Computational Aspects in Econometrics - Python II module

Pieter Kleer

Table of contents

Chapter 1

About

1.1 Welcome

Welcome to the online "book" for the Python II module of Computational Aspects in Econometrics. We will follow the content in this book during the lectures and it is the basis of the material that will appear on the exam, so you should read through this book carefully. Because this book is new, it is likely that we will make some edits throughout the course.

Please not that this is an advanced Python module. We assume familiarity with the basics of programming in Python. For students enrolled in the bachelor Econometrics and Operations Research, the topics of the course Programming for EOR is a good example of what I expect you to be familiar with. An online book covering most of these topis can be found here. This book also contains some topics not covered in Programming for EOR.

1.2 Goal

The goal of this module is to teach you the basics of scientific computing with Python. Here you should think mostly of implementing algorithmic tasks that you encounter during your Econometrics and Operations Research courses, such as, linear algebra, optimization, statistics and machine learning. We hope that the skills you are taught here can be useful for, e.g., numerical work in your bachelor or (perhaps later) master thesis. Furthermore, many companies nowadays program in Python, so the topics of this module can also be useful in your professional career later in life.

Next to teaching you how to implement certain mathematical task in a correct manner in Python, we also put emphasis on good coding practices. Especially if you start to write scripts with hundres of lines of code, it is important that you learn how to do this in a structured fasion using efficient Python functionality. Good coding practices are the topic of the next chapter. The idea is that you use these practices when doing the exercises corresponding to every lecture, as well as the group assignment.

1.3 Use of Spyder

This course document is based on the use of Spyder as integrated development environment (IDE) for creating Python code, i.e., the program that the code is written in. You can also use VS code or any other IDE to do the exercises and/or assignment in. Whenever this book contains screenshots to illustrate something, they will have been taken in Spyder.

To install the Anaconda distribution containing Spyder, follow the steps here. This is the installation that was alos recommended in the Python I module.

Chapter 2

Good coding practices

In this chapter we will discuss the good coding practices that have to applied when writing Python code: efficient computations, no hard coding, don't repeat yourself (DRY), single responsibility, and documentation. Overall, always try to keep your code simple and structured.

2.1 Efficient computations

Perhaps the most important topic of this course is that of efficient computation, i.e., to make efficient use of the mathematical functionality that Python has to offer, in particular the functionality of the NumPy package.

The way to think of this is as follows: Many of the mathematical tasks and exercises that we will see could, in theory, be solved using for- and/or while-loops, as well as if/else-statements. However, there are often more efficient functions programmed in the NumPy package in Python that can carry out these task in a quicker way using less code!

Let us look at an example. Suppose I am given a vector $x = [x_1, \dots, x_n] \in \mathbb{R}^n$, and I want to compute the L^2-norm

$$||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

of this vector. One way to solve this directly would be by using a for-loop to compute the sum inside the square root, and then take the square root of this number. We will illustrate this next for the vector $x = [1, 2, 3, \dots, 300.000]$.

```
# Length of vector
n = 300000

# Compute inner summation of ||x||_2
inner_sum = 0
for i in range(1,n+1):
    inner_sum = inner_sum + i**2

# Take square root
two_norm = (inner_sum)**(0.5)

# Print value of two-norm
print(two_norm)
```

```
94868566.97584295
```

Another way to do this is to define the vector x efficiently making use of the `arange()` function in Numpy, followed by the `linalg.norm()` function that can compute the L^2-norm for us directly. We will see these functions in more detail in a subsequent chapter.

It is standard convention to import the Numpy package under the alias `np`.

```
#Import Numpy package under the alias np
import numpy as np

# Length of vector
n = 300000

# Define vector x as array using arange() function
x = np.arange(1,n+1)

# Compute two-norm with built-in function
two_norm = np.linalg.norm(x)

# Print value of two-norm
print(two_norm)
```

```
94868566.97584295
```

This is a much cleaner, and in fact faster, way to compute the L^2-norm of the vector x. Especially for large values of n, the second piece of code that exploits the Numpy functionality can be much faster! The Python code below (which we will not discuss) illustrates this comparison by timing how long both approaches require to compute the norm for $n = 30.000.000$ (i.e., thirty million). If you run the code below on your own device, you might get different outputs, but, in general, there should be a significant (multiplicative) difference in execution time.

```
import numpy as np
import time

## Computing norm with for-loop

# We determine the time it takes to input the code between start and end
start = time.time()

# Length of vector
n = 30000000

# Compute inner summation of ||x||_2
inner_sum = 0
for i in range(1,n+1):
    inner_sum = inner_sum + i**2

# Take square root
two_norm = (inner_sum)**(0.5)

end = time.time()

# Time difference
```

```
loop_time = end - start
print('%.10f seconds needed for computing norm' % (loop_time),
      'with for-loop')

## Computing norm with Numpy functions

#We determine the time it takes to input the code between start and end
start = time.time()

# Length of vector
n = 30000000

# Define vector x as array using arange() function
x = np.arange(1,n+1)

# Compute two-norm with built-in function
two_norm = np.linalg.norm(x)

end = time.time()

# Time difference
numpy_time = end - start
print('%.10f seconds needed for computing norm' % (numpy_time),
      'with Numpy functions')


#This shows how many times faster Numpy approach is faster than for-loop solution
if numpy_time != 0:
    (print('NumPy\'s approach is %i times more efficient than for-loop approach.'
           % ((loop_time)/(numpy_time))))
```

```
12.3619461060 seconds needed for computing norm with for-loop
0.1451377869 seconds needed for computing norm with Numpy functions
NumPy's approach is 85 times more efficient than for-loop approach.
```

One important take-away of the above comparison is the following.

> When performing mathematical tasks, avoid the use of for- and while-loops, as well as if/else-statements, by efficient use of Python functionality.

2.2 No hard coding

Suppose we are given the function $f(x) = a \cdot x^2 + a \cdot b \cdot x - a \cdot b \cdot c$ and the goal is to compute $f(10)$ for $a = 3, b = 4$ and $c = -10$. One way of doing this would be to plug in all the variables and return the resulting function value.

```
# Hard coding
print(3*(10**2) + 3*4*10 + 4*-2*3)
```

```
396
```

However, this is inefficient for the following reason: If we would want to change the number $x = 10$ to, e.g., $x = 20$, we would have to twice replace 10 by 20. The same is true for the variable a, which appears in even three places. In general, in large scripts, variables can appear in more than a hundred places. To overcome

this inefficiency issue, it is always better to separate the input data (the x, a, b and c in this case) from the function execution

```
# No hard coding
def f(x, a, b, c):
    return a*x**2 + a*b*x + b*c*a

x = 10
a, b, c = 3, 4, -2
print(f(x, a, b, c))
```

```
396
```

The take-away of the above comparison is the following.

> Separate input data from the execution of functions and commands.

2.3 Don't repeat yourself (DRY)

If you have to carry out a piece of code for multiple sets of input data, always avoid copy-pasting the code. As a first step, try to to the repeated execution using a for-loop. For example, suppose we want to print the function values $f(x, 3, 4, -2)$ for $x = 1, 2, 3, 4$, with f as in the previous section. We can do this as follows.

```
def f(x, a, b, c):
    return a*x**2 + a*b*x + b*c*a

a, b, c = 3, 4, -2

# With repetition
x = 1
print('%.2f' % f(x,a,b,c))
x = 2
print('%.2f' % f(x,a,b,c))
x = 3
print('%.2f' % f(x,a,b,c))
x = 4
print('%.2f' % f(x,a,b,c))
```

```
-9.00
12.00
39.00
72.00
```

A more efficient way of typing this, is by using a for-loop that repeatedly executes the print statement.

```
import numpy as np

a, b, c = 3, 4, -2
x = [1,2,3,4]
# Determine all function values with list comprehension
y = [f(i,a,b,c) for i in x]

# Print values in y (could also print the whole vector y right away)
for j in y:
```

```
    print(j)
```

```
-9
12
39
72
```

The take-away of the above comparison is the following.

> Don't carry out the same task on different inputs twice by copy-pasting code, but use, e.g., a for-loop in which you iterate over the inputs.

In fact, later on in the course we will explain the concept of vectorizing a function, to make sure it can handle multiple input simultaneously. This is another way to avoid unnecessary repetition and in fact also the use of loops.

2.4 Single responsibility

When you write larger pieces of codes, it is often useful to split it up in smaller parts that all serve their own purpose. Suppose we want to implement Newton's method for finding a root x of a function $f : \mathbb{R} \to \mathbb{R}$, i.e., a point x that satisfies $f(x) = 0$.

Newton's methods starts with a (suitably chosen) initial guess $x_0 \in \mathbb{R}$ and repeatedly computes better approximations using the recursive formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

The goal is to implement this formula for the function $f(x) = (x-1)^2-1$, whose derivative is $f'(x) = 2(x-1)$. The roots of this function are $x = 0$ and $x = 2$.

```
# We implement the function y = x - f(x)/f'(x)
def y(x):
    return x - ((x-1)**2 - 1)/(2*(x-1))

x0 = 3
x1 = y(x0)

print(x1)
```

```
2.25
```

A clearer way to implement the formula $x_{i+1} = x_i - f(x_i)/f'(x_i)$ is to define separate functions for the evaluation of the functions f and f', and then combine these to create the recursive formula. In this way, we get three functions that are each responsible for one aspect of Newton's formula, namely implementing the function f, implementing the function f' and computing the recursive formula. This also makes it easier for a user of the code to understand what is happening.

```
# Single responsibility version

# Define f
def f(x):
    return (x-1)**2 - 1

# Define f'
```

```
def fprime(x):
    return 2*(x-1)

# Implement function y = x - f(x)/f'(x)
def y(x):
    return x - f(x)/fprime(x)

x0 = 3
x1 = y(x0)

print(x1)
```

```
2.25
```

The take-away of the above comparison is the following.

> If you are implementing a complex mathematical task involving multiple aspects, try to separate these aspects in different functions.

2.5 Documentation

The final good coding practice that we discuss is documentation. Ideally, you should always explain inside a function what the inputs and outputs are, and for larger scripts it is good to indicate what every part of the script does. For smaller functions and scripts this is not always necessary. We will next give an example for Newton's method as in the previous section

```
## Function that implements Newton's method
def newton(f,fprime,x0,iters):
    """
    This function implements Newton's iterative method
    x_{i+1} = x_i - f(x_i)/f'(x_i) for finding root of f.

    Input parameters
    ----------
    f : Function f
    fprime : Derivative of the function f
    x0 : Initial estimate for root x of f.
    iters : Number of iterations that we run Newton's method.

    Returns
    -------
    Approximation for x satisfying f(x) = 0.
    """

    # Initial guess
    x = x0

    # Repeatedly compute the recursive formula
    # by overwriting x for 'iters' iterations
    for i in range(iters):
        x_new = x - f(x)/fprime(x)
        x = x_new
    return x
```

```
## An example of Newton's method as implemented above

# Define f
def f(x):
    return (x-1)**2 - 1

# Define f'
def fprime(x):
    return 2*(x-1)

# Define intial guess and number of iterations
x0 = 10
iters = 6

# Run Newton's method
root = newton(f,fprime,x0,iters)

# Print output and explain what has been computed
print('Running Newton\'s method for %.i iterations with initial' % iters,
       'estimate %.2f' % x0,'\n','gives (estimated) root x = %.7f' % root,
       'with f(x) = %.7f' % f(root))
```

```
Running Newton's method for 6 iterations with initial estimate 10.00
 gives (estimated) root x = 2.0000013 with f(x) = 0.0000025
```

The following is a set of guidelines regarding how to add documentation to a function.

> Try to adhere to the following documention rules when writing complex functions: 1. Function documentation between triple double-quote characters. 2. Clearly describe what a function does and what its input and output arguments are. 3. Choose descriptive variable names, lines not longer than 80 characters. 4. Don't add comments for every line. Add comments for main ideas and complex parts. 5. A comment should not repeat the code as text (e.g. "time = time + 1 # increase time by one).

Chapter 3

NumPy arrays

3.1 Introduction

The NumPy package (module) is used in almost all numerical computations using Python. It is a package that provides high-performance vector, matrix and higher-dimensional data structures for Python. High-performance here refers to the fact that Python can perform computations on such data structures very quickly if appropriate functions are used for this.

To use NumPy you need to import the `numpy` module. This is typically done under the alias `np` so that you don't have to type `numpy` all the time when using a function from the module.

```
import numpy as np
```

We emphasize at this point that there is often not a unique way or command to achieve a certain outcome. When doing the exercises corresponding to the theory given in this chapter, it is, however, recommended to find a solution using the presented functionality.

3.2 Creating arrays

In the NumPy package the data type used for vectors, matrices and higher-dimensional data sets is an array. There are a number of ways to initialize new arrays, for example from

- a Python list or tuples;
- using functions that are dedicated to generating numpy arrays, such as `arange()` and `linspace()` (we will see those later);
- reading data from files.

3.2.1 Lists

For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array()` function. Since we imported NumPy under the alias `np`, we use `np.array()` for this.

To create a vector, the argument to the array function is a Python list

```
v = np.array([1,2,3,4]) #Array creation from list [1,2,3,4]
print(v)
```

```
[1 2 3 4]
```

To create a matrix, the argument to the array function is a nested Python list. Every element of the outer list is a list corresponding to a row of the matrix. For example, the matrix

$$M = \begin{bmatrix} 1 & 2 & 7 \\ 3 & -4 & 4 \end{bmatrix}$$

is created as follows.

```
M = np.array([[1, 2, 7], [3, -4, 4]])
print(M)
```

```
[[ 1  2  7]
 [ 3 -4  4]]
```

You can access the shape (number of rows and columns) , size (number of elements) and number of dimensions (number of axes in matrix) of the array with the `shape`, `size` and `ndim` attributes, respectively. Note that the size is simply the product of the numbers in the shape tuple, and the number of dimensions is the size of the shape tuple.

```
# Shape of matrix M
shape_M = M.shape  #np.shape(M) also works
print(shape_M)
```

```
(2, 3)
```

```
# Size of matrix M
size_M = M.size  #np.size(M) also works
print(size_M)
```

```
6
```

```
# Number of diemenions
ndim_M = M.ndim  #np.ndim(M) also works
print(ndim_M)
```

```
2
```

NumPy arrays are of the type `ndarray` (short for n-dimensional array). You can access this type through the `type()` function.

```
# Type of matrix M
type_M = type(M)
print(type_M)
```

```
<class 'numpy.ndarray'>
```

So far a NumPy array looks awfully much like a Python list (or nested list). Why not simply use Python lists for computations instead of creating a new array type?

There are several reasons:

- Python lists are very general. They can contain any kind of object. They are dynamically typed. They do not support mathematical functions such as matrix and dot multiplications.
- Numpy arrays are statically typed and homogeneous. The type of the elements is determined when the array is created.
- Numpy arrays are memory efficient.

- Because of the static typing, fast implementation of mathematical functions such as multiplication and addition of numpy arrays can be implemented in a compiled language (C and Fortran are used).

Using the `dtype` (data type) attribute of an array, we can see what type the data inside an array has.

```
# Data type of elements in array
dtype_M = M.dtype
print(dtype_M)
```

```
int32
```

If we want, we can explicitly define the type of the array data when we create it, using the `dtype` keyword argument:

```
# Define data as integers
M = np.array([[1, 2], [3, 4]], dtype=int)
print('M = \n', M)
```

```
M =
 [[1 2]
 [3 4]]
```

```
# Define data as floats
N = np.array([[1, 2], [3, 4]], dtype=float)
print('N = \n', N)
```

```
N =
 [[1. 2.]
 [3. 4.]]
```

```
# Define data as complex floats
O = np.array([[1, 2], [3, 4]], dtype=complex)
print('O = \n', O)
```

```
O =
 [[1.+0.j 2.+0.j]
 [3.+0.j 4.+0.j]]
```

Common data types that can be used with dtype are: `int`, `float`, `complex`, `bool`, `object`, etc.

We can also explicitly define the bit size of the data types, such as: `int64`, `int16`, `float128`, `complex128`. For example, `int64` allows us to define an integer variable in the range [−264,…,264].

You can also change the data type of the elements using the `astype()` method.

```
M = np.array([[1,2], [3,4]])
print(M.dtype)
```

```
int32
```

```
# Define M_float as matrix whose elements are those of
# the matrix M, but then as floats.
M_float = M.astype(float)
print(M_float)
```

```
[[1. 2.]
 [3. 4.]]
```

```
print(M_float.dtype)
```

```
float64
```

3.2.2 Arrays from functions

There are various useful arrays that can be automatically created using functions from the NumPy package. These arrays are typically hard to implement directly as a list.

`arange(n)`: This function creates the array $[0, 1, 2, \dots, n-1]$ whose elements range from 0 to $n-1$.

```
n = 10
x = np.arange(n)

print(x)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

If you want to explicitly define the data type of the elements, you can add the `dtype` keyword argument (the same applies for all functions that are given below).

```
n = 10
x = np.arange(n, dtype=float)

print(x)
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

`arange(a,b)`: This function creates the array $[a, a+1, a+2, \dots, b-2, b-1]$.

```
a, b = 5,11
x = np.arange(a,b)

print(x)
```

```
[ 5  6  7  8  9 10]
```

`arange(a,b,step)`: This function creates the array $[a, a+step, a+2\cdot step, \dots, b-2\cdot step, b-step]$. That is, the array ranges from a to b (but not including b itself), in steps of size `step`.

```
a, b, step = 5, 11, 0.3
x = np.arange(a,b,step)

print(x)
```

```
[ 5.   5.3  5.6  5.9  6.2  6.5  6.8  7.1  7.4  7.7  8.   8.3  8.6  8.9
  9.2  9.5  9.8 10.1 10.4 10.7]
```

`linspace(a,b,k)`: Create a discretization of the interval $[a, b]$ containing k evenly spaced points, including a and b as the first and last element of the array.

```
a,b,k = 5,10,20
x = np.linspace(a,b,k)

print(x)
```

```
[ 5.          5.26315789  5.52631579  5.78947368  6.05263158  6.31578947
  6.57894737  6.84210526  7.10526316  7.36842105  7.63157895  7.89473684
  8.15789474  8.42105263  8.68421053  8.94736842  9.21052632  9.47368421
  9.73684211 10.        ]
```

`diag(x)`: This function creates a matrix whose diagonal contains the list/vector/array `x`.

```
x = np.array([1,2,3])
D = np.diag(x)

print(D)
```

```
[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

`np.zeros(n)`: This function create a vector of length n with zeros.

```
n = 5
x = np.zeros(n)

print(x)
```

```
[0. 0. 0. 0. 0.]
```

`np.zeros((m,n))`: This function create a matrix of size $m \times n$ with zeros. Note that we have to input the size of the matrix as a tuple `(m,n)`; using `np.zero(m,n)`

```
m, n = 2, 5
M = np.zeros((m,n))

print(M)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

`np.ones(n)` and `np.ones((m,n))`: These functions create a vector of length n with ones, and a matrix of size $m \times n$ with ones, respectively.

```
m, n = 2, 5
x = np.ones(n)

print(x)
```

```
[1. 1. 1. 1. 1.]
```

```
M = np.ones((m,n))

print(M)
```

```
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

3.2.3 Reading data from files

The third option, which you might use most often in a professional context, is to read in data from a file directly into a NumPy array. You can do this using the `loadtxt()` function.

If you want to try this yourself, download the file numerical_data.dat here and store it in the same folder as where you are storing the Python script in which you execute the code snippet below.

```
# Load data into NumPy array
data_dat = np.loadtxt('numerical_data.dat')

# Print the data
print(data_dat)
```

```
[[ 1.   5.   4.  -9.   1. ]
 [ 3.   5.   6.   7.   7. ]
 [ 4.   3.   2.   1.   0.5]]
```

Python puts every row in the data (DAT) file into a separate row of the NumPy array; note that the numbers in the data file are separated by a whitespace character.

We can also save data from a Numpy array into a DAT-file using the `savetxt()` function. The first argument of this function is the name of the file in which you want to store the array, and the second argument is the array to be stored.

```
# Matrix M
M = np.array([[1,2,3],[5,6,7],[10,11,12],[14,15,16]])

# Save matrix to DAT file
np.savetxt('matrix.dat', M)
```

This should have created the file matrix.dat in the same folder as where you stored the Python script that ran the code above. You might notice that the numbers are stored using the scientific notation. For example, the number 1 appears as `1.00000000000000000e+00` in the CSV-file.

You can suprress this behaviour by explicitly specifying the data type in which you want the numbers in the matrix to be stored using the `fmt` keyword argument. For example, `fmt = '%i'` stores the numbers as integers.

```
# Matrix M
M = np.array([[1,2,3],[5,6,7],[10,11,12],[14,15,16]])

# Save matrix to DAT file
np.savetxt('matrix_int.dat', M, fmt='%i')
```

This should have created the file matrix_int.dat in the same folder as where you stored the Python script that ran the code above.

3.3 Accessing

In this section we will describe how you can access, or index, the data in a NumPy array.
We can index elements in an array using square brackets and indices, just like as with lists. In NumPy

indexing starts at 0, just like with a Python list.

```
v = np.array([12,4,1,9])

# Element in position 0
print(v[0])

# Element in position 2
print(v[2])

# Element in position -1 (last element)
print(v[-1]) # Same as v[3]

# Element in position -3 (counted backwards)
print(v[-3]) # Same as v[1]
```

```
12
1
9
4
```

3.3.1 Basic indexing

If you want to access the element at position (i, j) from a two-dimensional array, you can use the double bracket notation `[i][j]`, but with arrays you can also use the more compact syntax `[i,j]`.

```
M = np.array([[10,2,6,7], [-15,6,7,-8], [9,10,11,12],[3,10,6,1]])

# Element at position (1,1)
print('List syntax:',M[1][1])

# Element at position (1,1)
print('Array syntax', M[1,1])
```

```
List syntax: 6
Array syntax 6
```

If you want to access row i you can use `M[i]` or `M[i,:]`.

```
print(M[2]) # Gives last row

print(M[2,:]) # Gives last row
```

```
[ 9 10 11 12]
[ 9 10 11 12]
```

If you want to access column j you can use `M[:,j]`. Both here and in the previous command, the colon `:` is used to indicate that we want all the elements in the respective dimension. So `M[:,j]` should be interpreted as, we want the elements from all rows in the j-th column.

3.3.2 Index slicing

Index slicing is the technical name for the index syntax that returns a slice, a consecutive part of an array.

```
v = np.array([12,4,1,9,11,14,17,98])

print(v)
```

```
[12  4  1  9 11 14 17 98]
```

`v[lower:upper]`: This return the elements in `v` at positions `lower, lower+1,...,upper-1`. Note that the element at position `upper` is not included.

```
# Returns v[1], v[2], v[3], v[4], v[5]
print(v[1:6])
```

```
[ 4  1  9 11 14]
```

You can also omit the `lower` or `upper` value, in which case it is set to be position 0 or the last position -1, respectively.

```
# Returns v[3],...,v[8]
print(v[3:])

# Returns v[0],...,v[4]
print(v[:5])
```

```
[ 9 11 14 17 98]
[12  4  1  9 11]
```

`v[lower:upper:step]`: This returns the elements in `v` at position `lower,lower+step,lower+2*step,...(upper-1)-step, (upper-1)`. It does the same as `[lower:upper]`, but now in steps of size `step`.

```
v = np.array([12,4,1,9,11,14,17,98])

# Returns v[1], v[3], v[5]
print(v[1:6:2])
```

```
[ 4  9 14]
```

You can omit any of the three parameters `lower`,`upper` and `step`

```
# lower, upper, step all take the default values
print(v[::])

# Index in step is 2 with lower and upper defaults
print(v[::2])

# Index in steps of size 2 starting at position 3
print(v[3::2])
```

```
[12  4  1  9 11 14 17 98]
[12  1 11 17]
[ 9 14 98]
```

You can also use slicing with negative index values.

```
# The last three elements of v
print(v[-3:])
```

```
[14 17 98]
```

Furthermore, the same principles apply to two-dimensional arrays, where you can specify the desired indices for both dimensions

```
M = np.array([[10,2,6,7], [-15,6,7,-8], [9,10,11,12],[3,10,6,1]])

print(M)
```

```
[[ 10   2   6   7]
 [-15   6   7  -8]
 [  9  10  11  12]
 [  3  10   6   1]]
```

`[a:b, c:d]`: This returns the submatrix consisting of the rows `a,a+1,...,b-1` and rows `c,c+1,...,d`. You can also combine this with a step argument, i.e., use `[a:b:step1, c:d:step2]`.

```
# Returns elements in submatrix formed by rows 2,3 (excluding 4)
# and columns 1,2 (excluding 3)
print(M[2:4,1:3])
```

```
[[10 11]
 [10  6]]
```

If you want to obtain a submatrix whose rows and/or columns do not form a consecutive range, or if you want to specify these list manually, you can use the `ix_()` function from NumPy. Its arguments should be a list of row indices, and a list of column indices specifying the indices of the desired submatrix.

```
i = [0,2,3]
j = [0,3]

# Returns submatrix formed by rows 0,2,3 and columns 0,3
print(M[np.ix_(i,j)])
```

```
[[10  7]
 [ 9 12]
 [ 3  1]]
```

3.3.3 Fancy indexing

Fancy indexing is the name for when an array or list is used instead of indices, to access part of an array. For example, if you want to access elements in the locations $(0,3),(1,2)$ and $(1,3)$, you can define a list of row indices `[0,1,1]` and columns indices `[3,2,3]` and access the matrix with these lists.

```
i = [0,1,1]
j = [3,2,3]

# Returns M[0,3] = 7, M[1,2] = 7, M[1,3] = -8
print(M[i,j])
```

```
[ 7  7 -8]
```

Another way of fance indexing is by using a Boolean list, that indicates for every element whether it should be index (True) or not (False). Such a list is sometimes called a mask.

```
v = np.array([1,6,2,3,9,3,6])

# Tell for every element whether is should be index
mask = [False, True, True, True, False, True, False]

print(v[mask])
```

```
[6 2 3 3]
```

Typically, the mask is generated from a Boolean statement. For example, suppose we want to select all elements strictly smaller than 3 and greater or equal than 7 from the array `v`.

The following statements achieve this. Recall that you can use `&` if you want the first AND the second statement to be satisfied, and `|` if either the first OR the second has to be satisfied (or both).

```
mask_37 = (v < 3) | (v >= 7)

# Boolean vector indiciating for ever element in v
# whether the conditions v < 3 and v >= 7 are satisfied
print(mask_37)
```

```
[ True False  True False  True False False]
```

We can now access the elements satisfying these conditions by indexing `v` with this mask

```
print(v[mask_37])
```

```
[1 2 9]
```

3.4 Modifying

3.4.1 Elements, rows or columns

Using similar ways of indexing as in the previous section, we can also modify the elements of an array

```
M = np.array([[1,1,1,1], [2,2,2,2], [3,3,3,3],[4,4,4,4]])

print(M)
```

```
[[1 1 1 1]
 [2 2 2 2]
 [3 3 3 3]
 [4 4 4 4]]
```

```
# Modify individual element
M[0,1] = -1

print(M)
```

```
[[ 1 -1  1  1]
 [ 2  2  2  2]
```

```
 [ 3  3  3  3]
 [ 4  4  4  4]]
```

```
  # Modify (part of a) row
  M[1,[1,2,3]] = [-2,-2,-2]

  print(M)
```

```
[[ 1 -1  1  1]
 [ 2 -2 -2 -2]
 [ 3  3  3  3]
 [ 4  4  4  4]]
```

```
  # Modify third column to ones
  M[:,3] = np.ones(4)

  print(M)
```

```
[[ 1 -1  1  1]
 [ 2 -2 -2  1]
 [ 3  3  3  1]
 [ 4  4  4  1]]
```

3.4.2 Broadcasting

There does not necessarily have to be a match between the part of the matrix that we index, and the dimensions of the data that we want to overwrite that part with.

```
  M = np.array([[1,1,1,1], [2,2,2,2], [3,3,3,3],[4,4,4,4]])

  print(M)
```

```
[[1 1 1 1]
 [2 2 2 2]
 [3 3 3 3]
 [4 4 4 4]]
```

For example, in order to replace the third column of M by ones, we can also do the command below, instead of using `np.ones(4)`.

```
  # Modify third column to ones
  M[:,3] = 1

  print(M)
```

```
[[1 1 1 1]
 [2 2 2 1]
 [3 3 3 1]
 [4 4 4 1]]
```

Although there is a mismatch between the indexed part on the left (a column) and the data on the right (single number), Python broadcasts the data to an appopriate format by copying it to the correct size. That is, it copies the 1 to an array `[1,1,1,1]` of ones, which it then places in the third column.

This works similar in higher dimensions. Suppose we want to overwrite the second and third row with `[1,6,2,3]`. Then the indexed part is a 2×4 array, but the data a 1×4 array.

```
# Modify second and third row
M[2:4,:] = [1,6,2,3]

print(M)
```

```
[[1 1 1 1]
 [2 2 2 1]
 [1 6 2 3]
 [1 6 2 3]]
```

Python here first copies the data to `[[1,6,2,3],[1,6,2,3]]` and then modifies M with this array.

3.4.3 Transpose

Another useful function, in the context of linear algebra, is to take the transpose of a two-dimensional array `M`, which modifies the entries along the diagonal.

```
M = np.array([[1,2,3],[3,4,-1]])

print(M)
```

```
[[ 1  2  3]
 [ 3  4 -1]]
```

```
transpose_M = M.T #np.transpose(M) also works
print(transpose_M)
```

```
[[ 1  3]
 [ 2  4]
 [ 3 -1]]
```

3.5 Repeating and stacking

We can also use existing matrices and build new ones from it by stacking them either horizontally or vertically.

`tile(M,(k,r))`: This function takes an array `M` and copies it k times vertically and r times horizontally, resulting in a "tiling" of the original array `M`.

```
M = np.array([[1,2],[3,4]])

M_tile = np.tile(M,(2,3))
print(M_tile)
```

```
[[1 2 1 2 1 2]
 [3 4 3 4 3 4]
 [1 2 1 2 1 2]
 [3 4 3 4 3 4]]
```

If you do not input a tuples with two arguments, but only a number, then `tile()` does the tiling only horizontally.

```
M = np.array([[1,2],[3,4]])

M_tile = np.tile(M,4)
print(M_tile)
```

```
[[1 2 1 2 1 2 1 2]
 [3 4 3 4 3 4 3 4]]
```

`repeat(M,k)`: This function takes every element of `M`, repeats it k times, and puts all these numbers in a one-dimension array.

```
M = np.array([[1,2],[3,4]])

M_repeat = np.repeat(M,3)
print(M_repeat)
```

```
[1 1 1 2 2 2 3 3 3 4 4 4]
```

`vstack((a,b))`: This stacks two arrays `a` and `b` vertically, provided they have the correct dimensions to do this. Note that a and b should be inputted as a tuple `(a,b)`.

```
a = np.array([7,8])
M = np.array([[1,2],[3,4]])

M_a = np.vstack((M,a))
print(M_a)
```

```
[[1 2]
 [3 4]
 [7 8]]
```

`hstack((a,b))`: This stacks two arrays `a` and `b` horizontally, provided they have the correct dimensions to do this.

Note that in the example below we define `a` as a 1×2 array, i.e., a column array, to make sure we can stack it right of `M`. If we would have kept `a = np.array([7,8])` then Python will give an error, because it cannot stack a row vector next to a two-dimensional array.

```
a = np.array([[7],[8]])
M = np.array([[1,2],[3,4]])

M_a = np.hstack((M,a))
print(M_a)
```

```
[[1 2 7]
 [3 4 8]]
```

3.6 Reshaping

It is possible to adjust the shape of an array, while keeping the data of the array the same. For example, consider the $x = [1, 2, 3, \ldots, 12]$.

```
x = np.arange(1,13)

print(x)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12]
```

We can reshape it into the 3×4 matrix

$$M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 11 & 10 & 11 & 12 \end{bmatrix}$$

by using the `reshape(a,b)` method. It reshapes x to an $a \times b$ array provided that $a \cdot b$ equal the size (i.e., number of elements) of x.

```
# Reshape x to a 3-by-4 matrix
M = x.reshape(3,4)

print(M)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

We can also reshape two-dimensional arrays, for example, we can reshape M again to a 2×6 matrix.

```
# Reshape M to a 2-by-6 matrix
N = M.reshape(2,6)

print(N)
```

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
```

You should observe that Python does the reshaping in a very specific way: When we transform x to M above, Python fills the matrix M in a row-by-row fashion (instead of column-by-column). This is because of what is called the largest (axis) index changest fastest principle.

To understand this idea, recall that we can access the element at position (i, j) of a matrix M with `M[i,j]`. Here i is the row-index at position 0 of the index list `[i,j]`, and j is the column index at position 1 of the index list `[i,j]`. We said that the row indices form the 0-axis of the matrix, and the column indices the 1-axis.

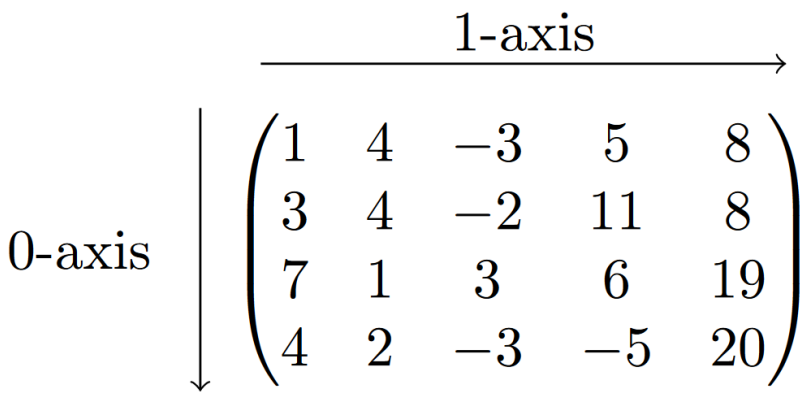


Figure 3.1: Axes of a two-dimensional array

Largest (axis) index changing fastest means that an $m \times n$ matrix gets filled first along the 1-axis, i.e., it fills the positions $(0,0), (0,1), ..., (0,n)$ while keeping the row index 0 fixed. It then moves up one row index, i.e., one position along the 0-axis and fills the elements $(1,0), (1,1), ..., (1,n)$, i.e., the elements along the 1-axis. It continues in this fashion until the complete matrix is full.

Another convenient method for reshaping is `flatten()`, which turns a matrix of any size into a one-dimensional array.

```
# Define 2-by-3 matrix
M = np.array([[9,1,3],[2,4,3]])

# Turn into one-dimensional array
x = M.flatten()
print(x)
```

```
[9 1 3 2 4 3]
```

If you want to turn a one-dimensional array $x = [x_0, \dots, x_{n-1}]$ into a column array of shape $(n, 1)$, you can do this as follows.

```
x = np.array([1,2,4,3,8])
n = np.size(x)

x = x.reshape(n,1)
print(x)
```

```
[[1]
 [2]
 [4]
 [3]
 [8]]
```

A more direct way of doing this, is by using `x[:,None]`.

```
x = np.array([1,2,4,3,8])
x = x[:,None] # Turns x into column array of shape (n,1)

print(x)
```

```
[[1]
 [2]
 [4]
 [3]
 [8]]
```

3.7 Copy vs. view

In the last sections we have seen various ways of using arrays to create other arrays. One point of caution here is whether or not the new array is a view or a copy of the original array.

3.7.1 View

A view y of an array x is another array that simply displays the elements of the array x in a different array, but the elements will always be the same. This means that if we would change an element in the array x, the same element will change in y and vice versa.

```
x = np.array([[4,2,6],[7,11,0]])
y = x # This create a view of x

print('y = \n', y)
```

```
y =
 [[ 4  2  6]
 [ 7 11  0]]
```

We next change an element in x. Note that the same element changes in y.

```
# Change element in x
x[0,2] = -30

# y now also changes in that position
print('y = \n',y)
```

```
y =
 [[  4   2 -30]
 [  7  11   0]]
```

The same happens the other way around: If we change an element in y, then the corresponding element in x also changes.

```
# Change element in y
y[1,1] = 100

# x now also changes in that position
print('x = \n', x)
```

```
x =
 [[  4   2 -30]
 [  7 100   0]]
```

Note that the same behaviour occurs in we apply the `reshape()` method.

```
# Define x = [1,2,...,12]
x = np.arange(1,13)

# Reshape x to a 3-by-4 matrix
M = x.reshape(3,4) # Creates view of x

print(M)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

If we now change an element in M, then the corresponding element changes in x. This mean that M is a view of the original array x.

```
# Change element in M
M[1,3] = 50

# x now also changes in that position
print(x)
```

```
[ 1  2  3  4  5  6  7 50  9 10 11 12]
```

3.7.2 Copy

A copy of an array x is an array z that is completely new and independent of x, meaning that if we change an element in x, then the corresponding element in z does not change, and vice versa. To obtain a copy of x, we can simply apply the `copy()` method to it.

```
# Define x = [1,2,...,12]
x = np.arange(1,13)

z = x.copy() # Create copy of x
z[0] = -10 # Change element of z

print('z = \n', z)
print('x = \n', x) # x has not changed
```

```
z =
 [-10   2   3   4   5   6   7   8   9  10  11  12]
x =
 [ 1  2  3  4  5  6  7  8  9 10 11 12]
```

Note that in the above example, x remains unchanged when we modify the element of z at position 0.

Similarly, to turn a reshaped array into a copy, we can apply the `copy()` method to it.

```
# Define x = [1,2,...,12]
x = np.arange(1,13)

# Reshape x to a 3-by-4 matrix
M = x.reshape(3,4).copy() # Create copy
M[0,0] = -10 # Change element of x
```

```
  print('M = \n', M)
  print('x = \n', x) # x has not changed
```

```
M =
 [[-10   2   3   4]
 [  5   6   7   8]
 [  9  10  11  12]]
x =
 [ 1  2  3  4  5  6  7  8  9 10 11 12]
```

The `flatten()` method actually directly creates a copy of the original array.

```
  # Define 2-by-3 matrix
  M = np.array([[9,1,3],[2,4,3]])

  # Turn into one-dimensional array
  x = M.flatten() # Creates copy of M
  x[0] = 100 # Change element in x

  print('x = \n', x)
  print('M = \n', M) # M has not changed
```

```
x =
 [100   1   3   2   4   3]
M =
 [[9 1 3]
 [2 4 3]]
```

It is important to know whether a Python function or command creates a copy or a view of the original array. You can typically look this up in the documentation of Python. Otherwise, experiment with the function or command to be sure how it behaves.

Chapter 4

Vectorization

In this chapter we will explore the power of NumPy arrays by studying the concept of vectorization. Let us first import the NumPy package.

```
import numpy as np
```

The idea of vectorization is that functions are designed so that they can efficiently handle multiple inputs simultaneously.

As an example, let's implement the Heavyside function H, which is defined by

$$H(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}.$$

That is, the function returns 1 if x is nonnegative, and 0 otherwise. If we are only interested in computing $H(x)$ for a single value x, then the following function suffices.

```
# Heavyside function
def Heavyside(x):
    if x >= 0:
        return 1
    else:
        return 0

print('H(2) =', Heavyside(2))
print('H(-3) =', Heavyside(-3))
```

```
H(2) = 1
H(-3) = 0
```

Suppose now that we would want to compute the value $H(x)$ for many values given in an array $x = [x_0, \ldots, x_{n-1}]$; this you would need to do, e.g., if you want to visualize a function. One way to do this is to use a for-loop and append the value of $H(x_i)$ to an (initially empty) list in iteration i.

```
n = 8
x = np.arange(-n,n)

h_values = []
for i in x:
    h_values.append(Heavyside(i)) # append H(i) to list h_values
```

```
print(h_values)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]
```

As mentioned earlier, for-loops are typically very time-inefficient and should be avoided whenever possible. It would be better if we could instead use another approach that can compute the function values more efficiently.

One approach to avoid the for-loop is with a Boolean statement. Recall from Chapter 2 that for an array x, the command `x >= 0` will return a Boolean array containing True at position i if $x_i \geq 0$, and False if not. This command is an example of a vectorized operation: Although, mathematically speaking, it is defined to compare a number x with 0, the command also works if x is an array, in which case each of its elements get compared to 0.

```
comparison = (x >= 0) # Compare each element in x with 0
print(comparison)
```

```
[False False False False False False False False  True  True  True  True
  True  True  True  True]
```

The statement `x >= 0` returns a Boolean array containing True and False, but the Heavyside function should ouput 1 and 0, respectively. To achieve this, we can convert the Boolean array to an array with ones and zeros.

This can be done with the `astype()` method, or by a clever multiplication: In Python, multiplying True with 1 gives 1, and False with 1 gives 0. Note that these are also examples of vectorized operations. For example, in the latter case, we multiply an array with one number, which Python executes by multiplying every element in the array with that number.

```
# Convert Boolean values to integers (True becomes 1, False becomes 0)
H = comparison.astype('int')
print(H)

# Multiply Boolean array with 1 (True*1 = 1, and False*1 = 0)
H2 = comparison*1
print(H2)
```

```
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
```

This means we can define the vectorized Heavyside function as follows, using the "multiplication with one" approach. We actually need less code for the vectorized version than in the original approach.

```
# Vectorized Heavyside function
def Heavyside(x):
    return (x >= 0)*1

print(x)
print(Heavyside(x))
```

```
[-8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
```

A vectorized function can handle higher-dimensional inputs. For example, a function that is (mathematically speaking) defined for a single number can also handle one- or two-dimensional

arrays as input, in which case the function computes the function value for every number in the array. Or a function that is defined for a one-dimensional array, can also handle two-dimensional arrays as input, in which case it computes the function value for every (one-dimensional) row (inner list) of the two-dimensional array.

We remark that vectorizing a function should not result in additional for-loops, but should exploit the functionality that Python, and in particular NumPy, have to offer. We will see many examples of this later in this chapter.

In this course, you will typically be told which types of input data your function should be able to handle, so, e.g., it could be that you have to write a function that performs mathematical operations on a one-dimensional array, but that it should be able to handle two-dimensional arrays as well (again, in which case your function should compute the function value for every row of that array).

As an example of what you should not do, the following definition of the Heavyside function is also able to handle "higher dimensional" inputs, but it does so by looping over the elements of the input, which makes it again a slow function. To summarize, you should not "hide" additional for-loops in the function itself.

```
# Heavyside function
def Heavyside(x):
    h = []
    for i in x:
        if i >= 0:
            h.append(0)
        else:
            h.append(1)
    return h

x = np.array([-2,-1,0,1,2])
print(Heavyside(x))
```

```
[1, 1, 0, 0, 0]
```

Also, it is not allowed to use functions like `vectorize()` from NumPy, because these are implemented essentially as a loop (as stated in the documentation of that function).

We continue with exploring vectorized functions within Python and NumPy.

4.1 Arithmetic operations

All basic arithmetic operations (addition, substraction, division, multiplication and comparison) are vectorized in Python. We will illustrate this with the addition operation `+`, but the same commands can be applied to the other arithmetic operations `-`, `/`, `*`, and `>=, ==, <=, !=`.

The addition operation `+` can be used to add two numbers together, as you well know. It can also add two arrays, i.e., it works as well for one- and two-dimensional arrays if they have the same shape. This is the usual addition operation you learn about when studying linear algebra.

```
x = np.array([1,4,7])
y = np.array([2,4,3])

print('x + y =\n',x+y)
```

```
x + y =
 [ 3  8 10]
```

```
A = np.array([[1,2],[1,4]])
N = np.array([[7,9],[3,4]])

print('A + N = \n',A+N)
```

```
A + N =
 [[ 8 11]
 [ 4  8]]
```

Python is also able to handle addition of arrays of different shapes in certain cases, using the concept of broadcasting that we have seen before. For example, we can add a single number to any array, in which case Python adds this number to every element in the array. This can be seen as an instance of vectorization.

```
c = 5

print('A + c =\n', A+c)
```

```
A + c =
 [[6 7]
 [6 9]]
```

We can also add either a one-dimensional array x of size n to an $m \times n$ matrix A. In this case, the array x gets added to every row of A:

$$A + x = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0(n-1)} \\ a_{10} & a_{11} & \dots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & \dots & a_{(m-1)(n-1)} \end{bmatrix} + \begin{bmatrix} x_0 & x_1 & \dots & x_{(n-1)} \end{bmatrix}$$
$$= \begin{bmatrix} a_{00} + x_0 & a_{01} + x_1 & \dots & a_{0(n-1)} + x_{n-1} \\ a_{10} + x_0 & a_{11} + x_1 & \dots & a_{1(n-1)} + x_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)0} + x_0 & a_{m1} + x_1 & \dots & a_{(m-1)(n-1)} + x_{n-1} \end{bmatrix}$$

In the example below, we have $m = 3$ and $n = 2$.

```
x = np.array([10,12])
print('Shape of x:', np.shape(x))

A = np.array([[1,2],[1,4],[3,1]])
print('Shape of A:', np.shape(A))

print('A + x = \n', A + x)
```

```
Shape of x: (2,)
Shape of A: (3, 2)
A + x =
 [[11 14]
 [11 16]
 [13 13]]
```

Again, this can be seen as an instance of vectorization, since Python automatically adds x to every row of the matrix A.

Note that the shape of x is $(n,)$ which is the syntax that Python uses to denote that x only has one dimension. You can define $x = [x_0, \dots, x_{n-1}]$ explicitly as a row vector of shape $(1, n)$ by defining `x = np.array([[x_0,...,x_{n-1}]])`, that is, with double brackets. It is sometimes needed to change the shape of an array from $(n,)$ to $(1, n)$ or $(n, 1)$ to be able to use a function from NumPy.

Addition works in the same way if we define x explicitly as an array of shape $(1, n)$.

```
x = np.array([[10,12]])
print('Shape of x:', np.shape(x))

A = np.array([[1,2],[1,4],[3,1]])
print('Shape of A:', np.shape(A))

print('A + x =\n', A + x)
```

```
Shape of x: (1, 2)
Shape of A: (3, 2)
A + x =
 [[11 14]
 [11 16]
 [13 13]]
```

The same works if we define $x = [x_0, \dots, x_{m-1}]^T$ as a column array of shape $(m, 1)$, in which case it gets added to every column of the matrix A of shape (m, n):

$$A + x = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0(n-1)} \\ a_{10} & a_{11} & \dots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & \dots & a_{(m-1)(n-1)} \end{bmatrix} + \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{(m-1)} \end{bmatrix}$$

$$= \begin{bmatrix} a_{00} + x_0 & a_{01} + x_0 & \dots & a_{0(n-1)} + x_0 \\ a_{10} + x_1 & a_{11} + x_1 & \dots & a_{1(n-1)} + x_1 \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)0} + x_{m-1} & a_{(m-1)1} + x_{m-1} & \dots & a_{(m-1)(n-1)} + x_{m-1} \end{bmatrix}$$

```
x = np.array([[10],[12],[14]])
print('Shape of x:', np.shape(x))

A = np.array([[1,2],[1,4],[3,1]])
print('Shape of A:', np.shape(A))

print('A + x =\n', A + x)
```

```
Shape of x: (3, 1)
Shape of A: (3, 2)
A + x =
 [[11 12]
 [13 16]
 [17 15]]
```

We cannot add arrays of any dimensions to each other. For example, if we would try to add a a 2×2 array to a 4×2 array, then Python will return `ValueError: operands could not be broadcast together with shapes (4,2) (2,2)`, i.e., Python cannot perform this addition.

4.1.1 Multiplication broadcasting

We emphasize that the broadcasting concepts above also apply to the multiplication operator `*`. That is, if x is a column array then `A*x` multiplies every column of A in a pointwise fashion with the array x. Similarly, for two matrix A and B, the syntax `A*B` returns a matrix in which all elements of A and B are pointwise multiplied with each other, that is, entry (i, j) contains $a_{ij} \cdot b_{ij}$.

This is not the same as, e.g., the matrix-vector multiplication Ax in the linear algebra sense, i.e,

$$
\begin{aligned}
Ax &= \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & \cdots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & \cdots & a_{(m-1)(n-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{(m-1)} \end{bmatrix} \\
&= \begin{bmatrix} a_{00}x_0 + a_{01}x_1 + \cdots + a_{0(n-1)}x_{n-1} \\ a_{10}x_0 + a_{11}x_1 + \cdots + a_{1(n-1)}x_{n-1} \\ \vdots \\ a_{(m-1)0}x_0 + a_{(m-1)1}x_1 + \cdots + a_{(m-1)(n-1)}x_{n-1} \end{bmatrix}
\end{aligned}
$$

We will see matrix-vector and matrix-matrix multiplications in the linear algebra sense later in this book.

4.2 Mathematical functions

Many mathematical functions in NumPy are also vectorized by default. Here you should think of functions like

- Trigonometry: `sin()`, `cos()`, `tan()`
- Exponentation and logarithms: `exp()`, `log()`, `log10()`, `log2()`
- Rounding: `around()`, `floor()`, `ceil()`
- Division with remainder: `mod()`, `divmod()`
- Power computation: `sqrt()`, `abs()`, `power()`

You access them using `np.function_name()`. Let us look at some examples; you can check out the documentation of the other functions yourself.

```
x = np.array([2,1,6])

# Compute sin(i) for every element i in x
y = np.sin(x)
print(y)
```

```
[ 0.90929743  0.84147098 -0.2794155 ]
```

```
A = np.array([[2,1,6],[1,1,3]])

# Compute e^i for every element i in A
y = np.exp(A)
print(y)
```

```
[[  7.3890561    2.71828183 403.42879349]
 [  2.71828183   2.71828183  20.08553692]]
```

```
x = np.array([1.249583, 3.110294, 4.51139])

# Round every number in x to two decimals
x = np.around(x, decimals=2)
print(x)
```

```
[1.25 3.11 4.51]
```

```
x = np.array([10,9,4])
y = np.array([2,4,5])

# Compute x_i (mod y_i) for all i
# and output divisor and remainder
z = np.divmod(x,y)
print(z)
```

```
(array([5, 2, 0]), array([0, 1, 4]))
```

`np.divmod()` outputs two arrays: the divisors and the remainder. For example, looking at `x[1] = 9` and `y[1] = 4`: the number 4 fits twice in 9, after which 1 is left, i.e., $9 = 2 \cdot 4 + 1$. The number 2 appears in the second position of the first array, and the remainder 1 in the second position of the second array.

```
A = np.array([[2,3,6],[4,2,3]])
N = np.array([[1,2,3],[1,2,3]])

# Pointwise compute a_{ij}^n_{ij} for all i,j
P = np.power(A,N)
print(P)
```

```
[[  2   9 216]
 [  4   4  27]]
```

4.3 Operations along array axes

Another efficient way to perform vectorized operations is to exploit the fact that many NumPy functions that perform an operation on a one-dimensional array, can also be used for two-dimensional arrays where the operation is then either performed on every column (i.e., along the 0-axis), or on every row (i.e., along the 1-axis).

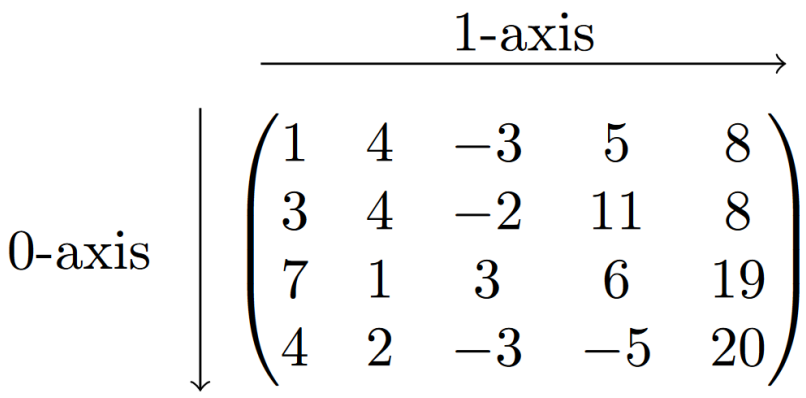


Figure 4.1: Axes of a two-dimensional array

We will look at some examples of this in the next sections.

4.3.1 Sorting and searching

The function `sort()` can be used to sort the elements in a one-dimensional array in ascending order, i.e., smallest to largest.

```
x = np.array([0.89, 0.5,  0.57, 0.34])

# Sort and print the elements in x
x_ascending = np.sort(x)
print(x_ascending)
```

```
[0.34 0.5  0.57 0.89]
```

It is not possible to use `sort()` to sort in descending order, i.e., largest to smallest, but this can be accomplished by reversing the sorted array. We can do this using index slicing with a step size of -1, starting at position -1, meaning that Python goes backwards through the array.

```
# Access x from beginning till end with step size -1
x_descending = x_ascending[-1::-1]

print(x_descending)
```

```
[0.89 0.57 0.5  0.34]
```

Vectorizing the (ascending) sort operation means we want to have a function that can take as input a two-dimensional array, and return for every row (or column) the sorted list of numbers. It turns out that `sort()` can do this right away, by adding an additional keyword argument `axis`.

Adding `axis=0` means that Python will sort the numbers in every column, i.e., along the 0-axis, and `axis=1` will sort numbers in every row, i.e., along the 1-axis.

```
A = np.array([
 [0.89, 0.5,  0.57, 0.34],
 [0.61, 0.12, 0.04, 1.  ],
 [0.27, 0.26, 0.28, 0.25],
 [0.9,  0.84, 0.15, 1.  ]])
```

```
# Sort elements in every column
A_col_ordered = np.sort(A,axis=0)
print(A_col_ordered)
```

```
[[0.27 0.12 0.04 0.25]
 [0.61 0.26 0.15 0.34]
 [0.89 0.5  0.28 1.  ]
 [0.9  0.84 0.57 1.  ]]
```

```
# Sort elements in every row
A_row_ordered = np.sort(A,axis=1)
print(A_row_ordered)
```

```
[[0.34 0.5  0.57 0.89]
 [0.04 0.12 0.61 1.  ]
 [0.25 0.26 0.27 0.28]
 [0.15 0.84 0.9  1.  ]]
```

We remark that `sort()` creates a copy, and not a view of the original matrix (see Chapter 3.7).

Another useful sorting function is `argsort()` that, for a given array $x = [x_0,...,x_{n-1}] outputs an array whose i-th element is the position of the number in x that appears in place i in the ordered array `np.sort(x)`.

```
x = np.array([0.89, 0.5,  0.57, 0.34])

# Position in original array or elements in ordered list
order = np.argsort(x) # np.sort(x) = [0.34, 0.5, 0.57, 0.89]
print(order)

# Obtaining sort() from argsort()
print(x[order])
```

```
[3 1 2 0]
[0.34 0.5  0.57 0.89]
```

In the example above, in the ordered list `np.sort(x)` the first number is 0.34, which appears at position 3 in x; the second number is 0.5, which appears at position 1 in x, the third number is 0.57 which appears in position 2 in x; and the fourth number is 0.89, which appears in position 0 in x.

This function also works for two-dimensional arrays. For example, determining the relative order of the elements in every column can be done by adding `axis=0` (and similarly in every row by using `axis=1`).

```
A = np.array([
 [0.89, 0.5,  0.57, 0.34],
 [0.61, 0.12, 0.04, 1.  ],
 [0.27, 0.26, 0.28, 0.25],
 [0.9,  0.84, 0.15, 1.  ]])

# Determine relative order in every column
N = np.argsort(A,axis=0)
print(N)
```

```
[[2 1 1 2]
 [1 2 3 0]
```

```
[0 0 2 1]
[3 3 0 3]]
```

4.3.2 Summary statistics

There are various other mathematical functions that can perform operations along axes by adding the `axis` keyword argument. Here we list some common ones from NumPy, that yield so-called summary statistics of a (one-dimensional) array:

- Sum and product: `sum()`, `prod()`,
- Mean and standard deviation: `mean()`, `std()`,
- Maximum and minimum: `max()`, `min()`.

We will illustrate the use of these six functions using `max()`, but the same code applies to all other functions (if the task at hand is mathematically well-defined).

```
A = np.array([
[2,3,6],
[4,2,3]
])
```

If we apply the `max()` function directly to a (two-dimensional) array, it will give the maximum value in the whole array.

```
# Gives maximum of all elements in A
A_max = np.max(A)

print(A_max)
```

```
6
```

If we add the `axis` keyword argument, we can either obtain the maximum of every row, or every column.

```
# Gives maximum of every column
A_column_max = np.max(A,axis=0)

print(A_column_max)
```

```
[4 3 6]
```

```
# Gives max of every row
A_row_max = np.max(A,axis=1)

print(A_row_max)
```

```
[6 4]
```

Another useful function is `argmax()` than can return the index (position) at which the maximum in an array is attained.

```
# Gives position of maximum in every column
A_col_argmax = np.argmax(A,axis=0)

print(A_col_argmax)
```

```
[1 0 0]
```

```
# Gives position of maximum in every row
A_row_argmax = np.argmax(A,axis=1)

print(A_row_argmax)
```

```
[2 0]
```

Note that the array containing the positions of the maxima is given as a row array. If you want to turn this into a column array (because the rows are ordered vertically in a two-dimensional array), recall you can do this as follows.

```
A_row_argmax = A_row_argmax[:,None]

print(A_row_argmax)
```

```
[[2]
 [0]]
```

If we try `np.argmax(A)` without using the `axis` keyword argument, then Python first flattens the matrix into a one-dimensional array, after which it returns the position of the maximum in the flattened array. Note that this flattening happens according to the largest index changing fasted principle (so it it places all the rows after each other, and not all the columns under each other).

Also note that if the maximum is attained in multiple places, then Python only returns the position of the first element that attains the maximum.

```
# Gives position of maximum
N = np.array([
[2,3,4],
[4,4,3]
])

N_argmax = np.argmax(N) # Turns N into [2,3,4,4,4,3];
                        # returns first position with maximum

print(N_argmax)
```

```
2
```

There are also more advance functions that give some summative information about an array:

- Cumulative sum: `cumsum()`,
- Cumulative product: `cumprod()`.

The function `cumsum()` return the cumulative sum of a one-dimensional array. As an example, if $x = [1, 4, 2, 5]$, then the cumulative sums will be given by

$$x_{\text{cumsum}} = [1, 1+4, 1+4+2, 1+4+2+5] = [1, 5, 7, 12].$$

```
x = np.array([1,4,2,5])

# Cumulative sum of x
x_cumsum = np.cumsum(x)
```

```
print('x_cumsum =', x_cumsum)
```

```
x_cumsum = [ 1  5  7 12]
```

The function can also be vectorized using the `axis` keyword argument.

The function `cumprod()` returns the cumulative product. Again, if $x = [1, 4, 2, 5]$, then the cumulative products will be given by

$$x_{\text{cumprod}} = [1, 1 \cdot 4, 1 \cdot 4 \cdot 2, 1 \cdot 4 \cdot 2 \cdot 5] = [1, 4, 8, 40].$$

```
x = np.array([1,4,2,5])

# Cumulative product of x
x_cumprod = np.cumprod(x)
print('x_cumprod =', x_cumprod)
```

```
x_cumprod = [ 1  4  8 40]
```

This function can also be vectorized using the `axis` keyword argument.

Chapter 5

Linear algebra and optimization

In this chapter we will see how Python can be used for performing linear algebra tasks, like solving systems of linear equations, and (integer) linear optimization.

Linear algebra forms the basis of many statistical tasks like principle component analysis (PCA), which can be used to identify important properties/dimensions of large data sets. This typically makes, e.g., machine learning algorithms more efficient as this avoid the so-called curse of dimensionality.

Applications of (integer) linear optimization range from scheduling, routing, transportation and food aid problems to solving Sudoku puzzles.

Let us first again import NumPy. Later on we will also use its linear algebra package (or module) `linalg`.

```
import numpy as np
```

5.1 Linear algebra

In this section we will often refer to one-dimensional arrays as row or column vectors, and to two-dimensional arrays as matrices.

5.1.1 Matrix multiplications

Recall from Section 4.1.1 that for an $m \times n$ matrix A and $m \times 1$ column vector x, the command `A*x` gives a matrix in which every column of A gets pointwise multiplied by the column vector x, that is,

$$A * x = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0(n-1)} \\ a_{10} & a_{11} & \dots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & \dots & a_{(m-1)(n-1)} \end{bmatrix} * \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{(m-1)} \end{bmatrix}$$

$$= \begin{bmatrix} a_{00} \cdot x_0 & a_{01} \cdot x_0 & \dots & a_{0(n-1)} \cdot x_0 \\ a_{10} \cdot x_1 & a_{11} \cdot x_1 & \dots & a_{1(n-1)} \cdot x_1 \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)0} \cdot x_{m-1} & a_{(m-1)1} \cdot x_{m-1} & \dots & a_{(m-1)(n-1)} \cdot x_{m-1} \end{bmatrix}$$

```
A = np.array([
[1,2],
```

```
  [3,4]])

  x = np.array([1,5])
  x = x[:,None] # Turn x into (2,1) shaped column vector

  print('A = \n', A)
  print('x = \n', x)
  print('A*x = \n', A*x)
```

```
A =
 [[1 2]
 [3 4]]
x =
 [[1]
 [5]]
A*x =
 [[ 1  2]
 [15 20]]
```

In the linear algebra sense, the multiplication Ax gives an $m \times 1$ column vector defined by

$$Ax = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & \cdots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & \cdots & a_{(m-1)(n-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{(m-1)} \end{bmatrix}$$
$$= \begin{bmatrix} a_{00}x_0 + a_{01}x_1 + \cdots + a_{0(n-1)}x_{n-1} \\ a_{10}x_0 + a_{11}x_1 + \cdots + a_{1(n-1)}x_{n-1} \\ \vdots \\ a_{(m-1)0}x_0 + a_{(m-1)1}x_1 + \cdots + a_{(m-1)(n-1)}x_{n-1} \end{bmatrix}$$

This can be achieved in Python with the `@` ('at') operator, that is, by writing `A @ x` (the white spaces are not needed; they are included here for readability).

```
  print('A = \n', A)
  print('x = \n', x)


  print('Ax = \n', A @ x)
```

```
A =
 [[1 2]
 [3 4]]
x =
 [[1]
 [5]]
Ax =
 [[11]
 [23]]
```

If we talk about matrix multiplication, it will always be clear from context whether we mean `A*x` or `A @ x`.

We can also use `@` to multiply a $k \times m$ and $m \times n$ matrix with each other in the linear algebra sense.

```
A = np.array([
[1,2],
[3,4],
[3,7]])

B = np.array([
[2,5,1,7],
[3,4,-1,8]])

# Note that k = 3, m = 2, n = 4
print('AB = \n',A @ B)
```

```
AB =
 [[ 8 13 -1 23]
 [18 31 -1 53]
 [27 43 -4 77]]
```

A special case of this is if we multiple a column vector

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

with its transpose (being a row vector) resulting in a matrix that contains $x_i x_j$ on position (i, j).

```
x = np.array([
[1],
[2],
[4]])

print('xx^T = \n', x @ x.T)
```

```
xx^T =
 [[ 1  2  4]
 [ 2  4  8]
 [ 4  8 16]]
```

5.1.2 Matrix properties

Using the `linalg` subpackage from NumPy, we can compute various well-known properties of a matrix (that you should recall from your Linear Algebra course).

- `linalg.matrix_rank(A)`: Computes rank of matrix A.
- `linalg.det(A)`: Computes determinant of square A.
- `linalg.eig(A)`: Computes eigenvalues and (right) eigenvectors of matrix A, i.e., values λ that satisfy $Av = \lambda v$ for some (eigen)vector v.
- `linalg.inv(A)`: Computes the inverse matrix A^{-1} of A, that is, the matrix that satisfies $A^{-1}A = AA^{-1} = I$, where I is the identity matrix.
- `linalg.norm(A)`: Computes the norm of matrix (or vector).

We can access these function with the syntac `np.linalg.function_name(A)`.

Finally, there is also the trace function `trace(A)` in NumPy that computes the trace of A, i.e., the sum of the elements on the diagonal. This function is implemented directly in NumPy, so you don't have to go to

the subpackage `linalg` first.

Let us look at some examples of these commands.

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

```
# Rank of A
print('Rank of A: ', np.linalg.matrix_rank(A))
```

```
Rank of A:  2
```

```
# Determinant of A
print('Determinant of A: ', np.linalg.det(A))
```

```
Determinant of A:  -2.0000000000000004
```

```
# Trace of A
print('Trace of A: ', np.trace(A))
```

```
Trace of A:  5
```

```
# Inverse of A
print('Inverse of A: \n', np.linalg.inv(A))
```

```
Inverse of A:
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

The function `linalg.eig()` is special in that it does not output one, but two arrays.

The first array is one-dimensional and contains the eigenvalues $[\lambda_0, \dots, \lambda_{n-1}]$ of A; the second array is two-dimensional, and contains the corresponding eigenvectors $v^0, \dots, v^{n-1}$ as columns of the matrix. The eigenvalues and eigenvectors are paired in the sense that $Av^i = \lambda_i v^i$ for $i = 0, \dots, n-1$.

```
# Eigenvalues and eigenvectors of A
lambdas, V = np.linalg.eig(A)

print('Eigenvalues of A: \n', lambdas)
print('Matrix with eigenvectors of A: \n', V)
```

```
Eigenvalues of A:
 [-0.37228132  5.37228132]
Matrix with eigenvectors of A:
 [[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

You can output only the eigenvalues or the eigenvectors by suppressing the other output argment with `_`.

```
# Only eigenvalues of A
lambdas, _ = np.linalg.eig(A)

# Only eigenvectors of A
```

```
_, V = np.linalg.eig(A)
```

If you want to recall how you can handle multiple outputs, and suppress the ones that your are not interested, have a look at Appendix B.

Let us check that the eigenvalues and eigenvectors indeed satsify $Av^i = \lambda_i v^i$ for $i = 0, 1$.

```
v0 = V[:,0]
v1 = V[:,1]

print('Verify first eigenvector:', A @ v0 - lambdas[0]*v0)
print('Verify second eigenvector:', A @ v1 - lambdas[1]*v1)
```

```
Verify first eigenvector: [0.00000000e+00 5.55111512e-17]
Verify second eigenvector: [0. 0.]
```

The function `linalg.norm(A)` computes the L^2-norm of a matrix $A = (a_{ij})_{i=1,\dots,m,j=1,\dots,n}$, also known as the Frobenius norm. It is defined as

$$||A||_{\mathrm{F}} = \sqrt{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} a_{ij}^2}.$$

If you apply the function two a row or column vector $x = [x_0, \dots, x_{n-1}]$, you get the usual L^2-norm defined as

$$||x||_2 = \sqrt{\sum_{i=0}^{n-1} x_i^2}.$$

You can also use this function in a vectorized manner. For example, if you want to compute the L^2-norm of every row in a matrix, you can use the `axis` keyword argument.

```
A = np.array([
[1,2],
[3,4],
[5,6]])

row_norms = np.linalg.norm(A,axis=1)

print(row_norms)
```

```
[2.23606798 5.         7.81024968]
```

5.1.3 Equation solving

For a given $m \times n$ matrix $A = (a_{ij})$ and $m \times 1$ column vector $b = [b_0, \dots, b_{m-1}]^T$, perhaps the most important question in linear algebra is to compute an $x = [x_0, \dots, x_{n-1}]^T$ that satisfies $Ax = b$, i.e., the linear systems of equalities

$$\begin{array}{ccccccccc} a_{00}x_0 & + & a_{01}x_1 & + & \dots & + & a_{0(n-1)}x_{n-1} & = & b_0 \\ a_{10}x_0 & + & a_{11}x_1 & + & \dots & + & a_{1(n-1)}x_{n-1} & = & b_1 \\ & & & & \vdots & & & & \\ a_{(m-1)0}x_0 & + & a_{(m-1)1}x_1 & + & \dots & + & a_{(m-1)(n-1)}x_{n-1} & = & b_{m-1} \end{array}.$$

If A is a square, invertible matrix then this can be done with `linalg.solve(A,b)`. This gives the unique solution $x = A^{-1}b$.

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

b = np.array([[1],[4]])

print('Solution to Ax = b: \n', np.linalg.solve(A,b))
```

```
Solution to Ax = b:
 [[ 2. ]
 [-0.5]]
```

In fact, you can compute x directly as $x = A^{-1}b$ using the inverse of A that we have seen earlier.

```
print('Solution to Ax = b: \n', np.linalg.inv(A) @ b)
```

```
Solution to Ax = b:
 [[ 2. ]
 [-0.5]]
```

If the system is overdetermined, then there is not necessarily a unique solution. This can happen when there are more constraints than variables, i.e., when $m > n$. In this case, we can find a solution x that is approximately optimal with the least squares method. It finds a solution x that solves the (non-linear) problem

$$\min_x ||Ax - b||_2.$$

The least squares method can be executed with `linalg.lstsq(A,b)`; we will see this method in the next chapter, when we consider non-linear optimization problems. This function can also be used to solve under-determined systems, where $n > m$.

5.2 Linear optimization

Recall that in a linear optimization problem the goal is to compute a vector $x = [x_0, \ldots, x_{n-1}]$ of decision variables that maximizes, or minimizes, a linear objective function subject to a collection of linear constraints, which can either be a $\leq$, $\geq$ or $=$ constraint.

In this section we will consider a general linear optimization problem of the form

$$\begin{array}{ll} \min & c^T x \\ \text{s.t.} & Ax = b \\ & Wx \leq z \\ & \ell \leq x \leq u \end{array}$$

where $c, \ell, u \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $U \in \mathbb{R}^{k \times n}$, and $z \in \mathbb{R}^k$ are the input data, or input parameters.

Written out this means we have a system with m equality constraints, k inequality constraints and upper and lower bounds on the decision variables.

$$
\begin{array}{rrcccccl}
\min & c_1x_1 & + & \cdots & + & c_nx_n & & \\
\text{s.t.} & a_{01}x_0 & + & \cdots & + & a_{0(n-1)}x_{n-1} & = & b_0 \\
 & & & \vdots & & & & \\
 & a_{(m-1)1}x_0 & + & \cdots & + & a_{(m-1)(n-1)}x_{n-1} & = & b_{m-1} \\
 & w_{01}x_0 & + & \cdots & + & w_{0(n-1)}x_{n-1} & \leq & z_0 \\
 & & & \vdots & & & & \\
 & w_{(k-1)1}x_0 & + & \cdots & + & w_{(k-1)(n-1)}x_{n-1} & \leq & z_{k-1} \\
\ell_0 \leq & x_0 & & & & & \leq & u_0 \\
 & & & \ddots & & & & \\
\ell_{n-1} \leq & & & & & x_{n-1} & \leq & u_{n-1}
\end{array}
$$

You can model a $\geq$ constraint by multiplying it with -1, so that it turns into a $\leq$ constraint.

If, in addition, we require that $x_i \in \mathbb{N}$ for $i = 1, \ldots, n$, i.e., that the variables are integral, then we call the above problem an integer linear optimization problem.

We will look at two packages that can be used to solve (integer) linear optimization problems. The difference is that the first package `linprog` allows us to explicitly input the data c, A, B, ℓ and u, whereas the second package `pulp` allows us to define constraints one-by-one. The latter is more convenient if the input data his not given explicitly.

5.2.1 Explicit input data

If the input data is given explicitly, we can solve linear optimization problems quickly with the `linprog` module, which is part of the `optimize` package of SciPy. We will see more functionality of SciPy in later chapters of this book.

We can import `linprog` as follows:

```
from scipy.optimize import linprog
```

This package is suitable for solving problem of the general form above. We will implement the following example:

$$
\begin{array}{rrrcrcr}
\max & z = & 15x_1 & + & 20x_2 & & \\
\text{s.t.} & & 2x_1 & + & 2x_2 & \leq & 8\frac{1}{2} \\
 & & x_1 & + & 2x_2 & \leq & 6 \\
 & & 12x_1 & + & 17x_2 & = & 51 \\
 & & x_1 & & & \geq & 0 \\
 & & & & x_2 & \geq & 0
\end{array}
$$

Because `linprog` works with explicitly input data, we first define these. We define all the input data as NumPy arrays, but you can actually also use plain lists for this.

Note that we multiply the objective function c with -1, so that our maximization problem turns into a minimization problem.

```
## Define input data

# Coefficients of the objective function
c = np.array([-15, -20])  # Minimize -15x1 - 20x2

# Coefficients of the equality constraints (Ax = b)
A = np.array([[12, 17]]) # 12x1 + 17x2 = 51
```

```
b = np.array([51])

# Coefficients of the inequality constraints (Ux <= z)
U = np.array([
[2, 2],  # 2x1 + 2x2 <= 8.5
[1, 2]]) # x1 + 2x2 <= 6

z = np.array([[8.5], [6]])
```

The bounds on x_i have to be inputted as tuple (ℓ_i, u_i). If the lower or upper bound is not present, meaning either $\ell_i = -\infty$ or $u_i = +\infty$, you can replace the entry with `None`.

```
# Bounds for the variables x1 and x2 (l = 0, u = infinity)
x1_bounds = (0, None)  # x1 >= 0
x2_bounds = (0, None)  # x2 >= 0
```

We can solve the problem with `linprogr()` It takes input arguments

- `c` : The vector with objective function coefficients
- `A_eq` : The constraint matrix for the equality constraints
- `b_eq` : The right hand side vector of the equality constraints
- `A_ub` : The constraint matrix for the inequality constraints
- `b_ub` : The right hand side vector of the inequality constraints
- `bounds` : List of tuples with each tuple having the upper and lower bound for the corresponding variable

Note that `A_eq` and `b_eq` can be left out as input arguments if there are no equality constraints; the same holds for `A_ub` and `b_ub`.

```
# Solve the linear programming problem
# (You can use \ to continue command on next line)
result = linprog(c, A_eq=A, b_eq= b,  \
                    A_ub=U, b_ub=z, \
                    bounds=[x1_bounds, x2_bounds])
```

The `result` variable contains various aspects of the solution. The two most important for us are

- `result.x` : The optimal solution x
- `result.fun` : The function value attained by the the optimal solution.

```
# Output the results
print('The problem is solved by', result.x, \
      'with objective function value', result.fun)
```

```
The problem is solved by [4.25 0.  ] with objective function value -63.75
```

If we want to solution to be integral, we add the input argument `integrality=1`. See the documentation to read about this and to find more options.

```
# Solve the integer linear optimization problem
# (You can use \ to continue command on next line)
result_integral = linprog(c, A_eq=A, b_eq= b,  \
                    A_ub=U, b_ub=z, \
                    bounds=[x1_bounds, x2_bounds], \
```

```
                    integrality=1)

# Output the results
print('The problem is solved by', result_integral.x, \
      'with objective function value', result_integral.fun)
```

```
The problem is solved by [0. 3.] with objective function value -60.0
```

The complete code of this section can be found below.

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

## Define input data

# Coefficients of the objective function
c = np.array([-15, -20])  # Minimize -15x1 - 20x2

# Coefficients of the equality constraints (Ax = b)
A = np.array([[12, 17]]) # 12x1 + 17x2 = 51

b = np.array([51])

# Coefficients of the inequality constraints (Ux <= z)
U = np.array([
[2, 2],  # 2x1 + 2x2 <= 8.5
[1, 2]]) # x1 + 2x2 <= 6

z = np.array([[8.5], [6]])

## Bounds for the variables x1 and x2 (l = 0, u = infinity)
x1_bounds = (0, None)  # x1 >= 0
x2_bounds = (0, None)  # x2 >= 0

## Solve the linear optimization problem
## (You can use \ to continue command on next line)
result = linprog(c, A_eq=A, b_eq= b,  \
                    A_ub=U, b_ub=z, \
                    bounds=[x1_bounds, x2_bounds])

# Output the results
print('The linear optimization problem is solved by', result.x, \
      'with objective function value', result.fun)

###################################
## With integrality constraints ##
###################################

# Solve the problem
result_integral = linprog(c, A_eq=A, b_eq= b,  \
                    A_ub=U, b_ub=z, \
                    bounds=[x1_bounds, x2_bounds], \
                    integrality=1)
```

```
# Output the results
print('The integer linear optimization problem is solved by', result_integral.x, \
      'with objective function value', result_integral.fun)
```

5.2.2 Implicit input data

If the constraints are not given explicitly, creating the input data matrices like A and U can be quite tedious. In this case, it works better to define the constraints directly, based on the given problem description.

In this section we will consider the maximum weight bipartite matching problem. We are given a (complete) bipartite graph $G = (V, W, E)$ with node sets $V = \{0, \dots, n-1\}$ and $W = \{0, \dots, n-1\}$ and edges $ij \in E = \{ab : a \in V, b \in W\}$ connecting the nodes $i \in V$ with $j \in W$. Every such edge has a weight w_{ij}. We want to match up every node in V with a node in W so that the total summed weight of the selected edges is maximal.

We introduce (binary) decision variables x_{ij} with the interpretation that $x_{ij} = 1$ if i and j get matched, and $x_{ij} = 0$ otherwise. The constraints then mean that every node $i \in V$ should get matched up with precisely one $j \in W$ and vice versa.

$$
\begin{array}{lllll}
\max & \displaystyle\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} w_{ij}x_{ij} & & & \\
\text{s.t.} & \displaystyle\sum_{i=0}^{n-1} x_{ij} & = & 1 & \text{for } j = 0, \dots, n-1 \\
& \displaystyle\sum_{j=0}^{n-1} x_{ij} & = & 1 & \text{for } i = 0, \dots, n-1 \\
& x_{ij} & \geq & 0 & \text{for all } i, j = 0, \dots, n-1 \\
& x_{ij} & \in & \{0, 1\} & \text{for all } i, j = 0, \dots, n-1
\end{array}
$$

In fact, the last conditions defining the decisions variables to be binary, are redundant: If we leave them out then the optimal solution that is found will satisfy them regardless (you might have seen this in a Combinatorial Optimization course). For now, we will ignore this mathematical fact, and define the decision variables as binary variables.

Defining the constraint matrix explicitly is quite a hassle, and given that many of its entries are zero, it is not very efficient for Python to store it explicitly, so using `linprog` here is not very convenient.

Instead, in this section we will use the `pulp` package to solve this problem.

```
import pulp
```

We create some input data (weights) for the problem that we are going to build.

```
# Size of the node sets
n = 4

# Weights of the edges ij
w = np.arange(1,n**2+1).reshape(n,n)

print(w)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
```

```
 [ 9 10 11 12]
 [13 14 15 16]]
```

Problem instantiation. We first initialize the problem object that we want to solve with `LpProblem()` that takes as input:

- Problem name: String variable
- Problem type: Maximization (`pulp.LpMaximize`), or minimization (`pulp.LpMinimize`).

```
prob = pulp.LpProblem("Weighted_Bipartite_Matching", pulp.LpMaximize)
```

As opposed to `linprog` we need to explicitly instantiate the decision variables, so that we can use them later to add constraints to our problem.

Decision variables. A decision variable y can be instantiated with `LpVariable()` that takes as input

- Variable name: String being name of variable
- Variable category: Keyword argument `cat` being Binary, Integer or Continuous (default if `cat` is not specified).

```
pulp.LpVariable('y',cat='Binary')
```

To initiate our decision variables we need to loop over the indices i and j, which can be done as follows. Note that the syntax `f"y_{i}_{j}"` allows us to incorporate/format the loop indices i and j into the variable name (which is a string).

```
x = np.zeros((n,n), dtype=object) # Data type of variable is 'object'
for i in range(n):
    for j in range(n):
        x[i,j] = pulp.LpVariable(f"x_{i}_{j}", cat='Binary')
```

This can be done more compactly using list comprehension, which is a quick alternative for doing the for-loops explicitly. Avoiding for-loops altogether here is not possible, because variables cannot be generated in a vectorized manner if we want to give them all a separate name.

```
# With list comprehension
x = np.array([[pulp.LpVariable(f"x_{i}_{j}", cat='Binary') \
               for j in range(n)] for i in range(n)])
```

In both cases we have created an $n \times n$ NumPy array whose elements are decision variables x_{ij}.

```
# Print names of decision variables
print(x)
```

```
[[x_0_0 x_0_1 x_0_2 x_0_3]
 [x_1_0 x_1_1 x_1_2 x_1_3]
 [x_2_0 x_2_1 x_2_2 x_2_3]
 [x_3_0 x_3_1 x_3_2 x_3_3]]
```

We will use these variables to define the objective function and constraints.

Objective function. We next construct the objective function

$$\max \quad \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} w_{ij} x_{ij}$$

Because both the weight matrix `w` and the decision variables `x` are stored in NumPy arrays, we can do a pointwise multiplication of these arrays, and sum up the elements of the resulting array, to get the objective.

Summing up decision variables in arrays, in our case the array `w*x`, can be done using `lpSum()` within PuLP. To add the objective to the problem, and later constraints, we use the syntax `prob += [objective]` where `[objective]` contains the objective function expresssion in terms of the decision variables that we instantiated. You should think of the syntax `+=` as adding something to the instantiated problem `prob`.

```
prob += pulp.lpSum(w*x)
```

Note that we write `lpSum()` and not `LpSum()` with a capital L. This has to do with the fact that `lpSum()` is a function, whereas `LpProblem` and `LpVariable` are classes (that have different naming conventions in Python modules).

```
# Print objective function
print(prob.objective)
```

```
x_0_0 + 2*x_0_1 + 3*x_0_2 + 4*x_0_3 + 5*x_1_0 + 6*x_1_1 + 7*x_1_2 + 8*x_1_3 + 9*x_2_0 + 10*x_2_1 + 11*x_
```

Constraints. Finally, we add the constraints

$$\begin{aligned} \sum_{i=0}^{n-1} x_{ij} &= 1 \quad \text{for } j = 0, \dots, n-1 \\ \sum_{j=0}^{n-1} x_{ij} &= 1 \quad \text{for } i = 0, \dots, n-1 \end{aligned}$$

to the problem, which can also be done with the `prob += [constraint]` syntax, where `[constraint]` is the constraint we want to add. We can add our constraints as follows.

```
for j in range(n):
    prob += pulp.lpSum([x[i,j] for i in range(n)]) == 1

for i in range(n):
    prob += pulp.lpSum([x[i,j] for j in range(n)]) == 1
```

To print the constraints, we need to realize that they are stored in an (ordered) dictionary. We can print the keys and values of this dictipnary as follows.

```
# Print constraints
for key, value in prob.constraints.items():
    print(f"{key}: {value}")
```

```
_C1: x_0_0 + x_1_0 + x_2_0 + x_3_0 = 1
_C2: x_0_1 + x_1_1 + x_2_1 + x_3_1 = 1
_C3: x_0_2 + x_1_2 + x_2_2 + x_3_2 = 1
_C4: x_0_3 + x_1_3 + x_2_3 + x_3_3 = 1
_C5: x_0_0 + x_0_1 + x_0_2 + x_0_3 = 1
_C6: x_1_0 + x_1_1 + x_1_2 + x_1_3 = 1
_C7: x_2_0 + x_2_1 + x_2_2 + x_2_3 = 1
_C8: x_3_0 + x_3_1 + x_3_2 + x_3_3 = 1
```

Solving the problem. We solve the problem with the `solve()` function.

We can access the values of the objective function and the variables using `pulp.value()`. The objective function of the problem is stored in `prob.objective`, and so its value in the optimized model can be

accessed with `pulp.value(prob.objective)`.

We want to represent the optial matching nicely in a binary matrix. Unfortunately, printing the values of the optimized model is difficult to do without for-loops since the function `pulp.value(x[i,j])` that returns the value of variable x_{ij} only works for numbers and not lists or arrays.

```
# Solve the problem
prob.solve()

# Store the results in matrix 'matching'
matching = np.zeros((n,n), dtype=int)
for i in range(n):
    for j in range(n):
        matching[i,j] = pulp.value(x[i,j])

# Print solution
print("The optimal matching is: \n", matching)
print(f"Optimal value of the objective function: {pulp.value(prob.objective)}")
```

```
The optimal matching is:
 [[1 0 0 0]
 [0 1 0 0]
 [0 0 0 1]
 [0 0 1 0]]
Optimal value of the objective function: 34.0
```

The complete code of this section can be found below.

```
import pulp

# Size of the node sets (input data)
n = 4

# Weights of the edges ij (input data)
w = np.arange(1,n**2+1).reshape(n,n)

# Instantiate problem
prob = pulp.LpProblem("Weighted_Bipartite_Matching", pulp.LpMaximize)

# Instantiate decision variables and store in NumPy array
x = np.array([[pulp.LpVariable(f"x_{i}_{j}", cat='Binary') \
               for i in range(n)] for j in range(n)])

# Set objective function
prob += pulp.lpSum(w*x)

# Set constraints
for j in range(n):
    prob += pulp.lpSum([x[i,j] for i in range(n)]) == 1

for i in range(n):
    prob += pulp.lpSum([x[i,j] for j in range(n)]) == 1

# Store the results in matrix 'matching'
```

```
matching = np.zeros((n,n), dtype=int)
for i in range(n):
    for j in range(n):
        matching[i,j] = pulp.value(x[i,j])

# Print solution and objective value
print("The optimal matching is: \n", matching)
print(f"Optimal value of the objective function: {pulp.value(prob.objective)}")
```

5.2.3 Remarks

The `linprog` package is typically convenient for small linear optimization problems, especially for problems whose input data (such as the constraint matrices) you can define explicitly.

The `pulp` package is more useful when the input data is defined implicitly, e.g., using different sets of inequalities defined throug indices. Furthermore, when problems become of larger scale, as you will typically encounter in practice, it is better to use state-of-the-art solvers such as CPLEX, GUROBI, or MOSEK. PuLP is able to be coupled to such solvers, that is, you can define a problem in PuLP and then have it solved with the external solver software.

Many of these solvers can also handle other problems such as mixed integer linear optimization problems, in which there is a mixture of continuous, integer and binary variables, and various non-linear optimization problems. We will see more of those in the next chapter.

Chapter 6

Nonlinear algebra and optimization

In this chapter we will see functions for solving systems of nonlinear equations and how to optimize a nonlinear function.

For equation solving, the main goal is to compute an $x \in \mathbb{R}^n$ that satisfies the system

$$\begin{cases} f_0(x) = 0 \\ f_1(x) = 0 \\ \vdots \\ f_{n-1}(x) = 0 \end{cases}$$

where $f_0, \ldots, f_{n-1} : \mathbb{R}^n \to \mathbb{R}$ are continuous, n-variate functions. That is, we have a systems of n equations and n variables. We will also see the least squares method for problems in which the number of equations is not equal to the number of variables (i.e., the sytem is under- or overdetermined).

For optimizing a function, we have a single continuous, n-variate function $f : \mathbb{R}^n \to \mathbb{R}$, and the goal is to compute an $x \in \mathbb{R}^n$, that attains

$$\min_{x \in \mathbb{R}^n} f(x).$$

When there is only one (nonlinear) equation in one variable, we will also see ways to specify the interval in which the solution to the respective problems should be searched for.

The Python package SciPy contains all the functions we will need to achieve these goals, in particular, the `optimize` submodule. We import it under the alias `optimize` for convenience, and also import NumPy.

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

6.1 Root finding

In this section we will discuss various root finding methods. We start by introducing various methods for finding the root of a univariate funtion, and then show similar methods for the multivariate case. We end with the least squares method, that finds an approximately optimal solution when a solution might not exist.

6.1.1 Univariate function

We present three type of methods and information that can be used to find roots of a function. We emphasize that none of these methods guarantee that a root will always be found if it exists, but in many cases they do.

Using `fsolve()`

The easiest-to-use function for finding the root of a univariate function $f : \mathbb{R} \to \mathbb{R}$ is `fsolve()` from the `optimize` module. It takes two mandatory input arguments:

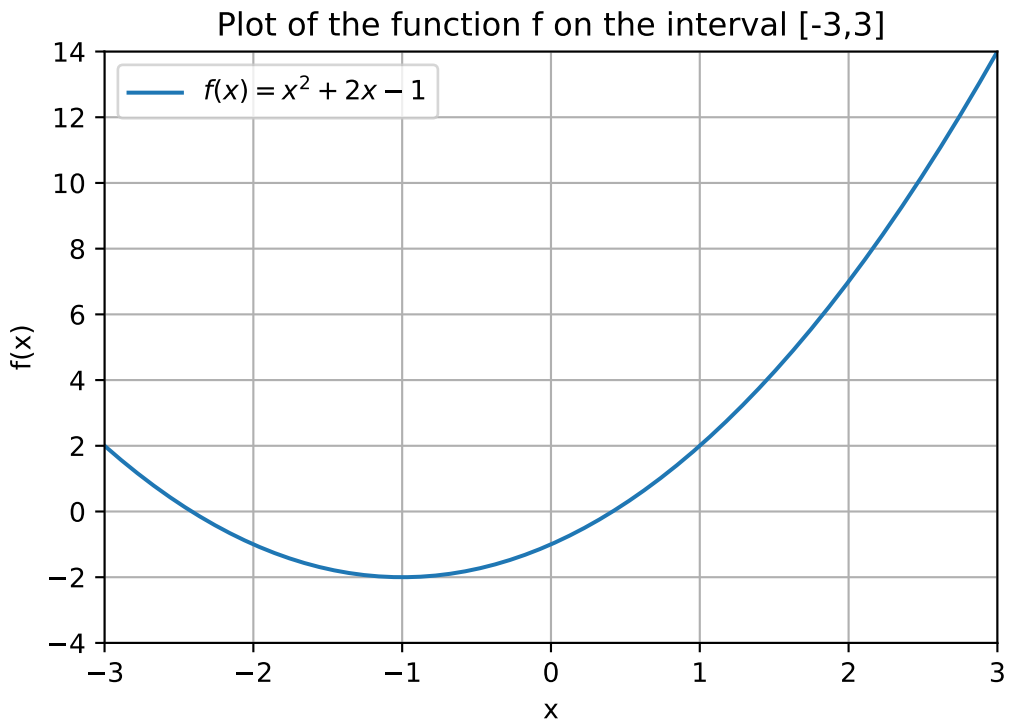
- The function we want to find the root of, and
- an initial guess for the root.

As an example, suppose we want to find the root of the equation

$$f(x) = x^2 + 2 \cdot x - 1,$$

that is plotted below. You do not have to look at the code generating this figure (but it is included for completeness).

```
import numpy as np
import matplotlib.pyplot as plt

# Define the x range
x = np.linspace(-3, 3, 600)

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Create the plot
plt.figure(figsize=(6, 4))
plt.plot(x, f(x), label='$f(x) = x^2 + 2x - 1$')

# Add labels and title
plt.title('Plot of the function f on the interval [-3,3]')
plt.xlabel('x')
plt.ylabel('f(x)')

# Add a grid
plt.grid(True)

# Set range
plt.xlim(-3,3)
plt.ylim(-4,14)

# Add a legend
plt.legend()

# Show the plot
plt.show()
```



Our initial guess will be `guess = 3`. There are various ways in which you can input the function and initial guess. Either as

- Keyword arguments: Use `func` for the function and `x0` for the initial guess.
- Positional arguments: Input function followed by initial guess, without keywords.

```
def f(x):
    return x**2 + 2*x - 1

guess = 3
x_root = optimize.fsolve(f,x0=guess)

# Also works:
# x_root = optimize.fsolve(func=f,x0=guess)
# x_root = optimize.fsolve(x0=guess, func=f)
# x_root = optimize.fsolve(f,guess), here the order matters!

# Does not work
# x_root = optimize.fsolve(x0=guess,f), f is positional, order matters!
# x_root = optimize.fsolve(guess,f), order matters!

print("A root of the function f is given by", x_root)

# x_root is an array one one element;
# can access index 0 to only get value
print("A root of the function f is given by", x_root[0])
```

```
A root of the function f is given by [0.41421356]
A root of the function f is given by 0.41421356237309503
```

For any function, in SciPy and beyond, you should look at the order of the required arguments of a function in its documentation.

> Because different root finding and optimization methods use a different keyword argument for the function of interest (e.g., `func`, `fun`, `f`), we will introduce the convention in this chapter that the first input argument of any SciPy function that we use is the function we want to find the root of, or optimize over, and all other input arguments will have a keyword (such as `x0`). The order of the keyword input arguments is not relevant, as long as they come after the function, which is the first input argument.

The choice of initial guess can determine which root we end up in. The function f has two roots as can be seen from the figure. If we start with another initial guess, such as `guess=4`, we find the other root of f.

```
def f(x):
    return x**2 + 2*x - 1

guess = -4
x_root = optimize.fsolve(f,x0=guess)

print("Another root of the function f is given by", x_root)
```

```
Another root of the function f is given by [-2.41421356]
```

If the function f would have had multiple input arguments, than `fsolve()` interprets the first argument of f as being the unknown variable that it needs to compute. Any remaining arguments of f should be specified in the `args` keyword argument.

For example, suppose that we would have defined $g(x, a, b, c) = a \cdot x^2 + b \cdot x + c$. Then executing `x_root = optimize.fsolve(g,x0=guess)` will result in an error because `fsolve()` cannot determine a root x if it does not know the values of a, b and c. Therefore, we need to specify these additional inputs in the `args` keyword argument of `fsolve()`.

```
def g(x,a,b,c):
    return a*x**2 + b*x + c

a, b, c = 1, 2, -1

guess = -4
x_root = optimize.fsolve(g,x0=guess,args=(a,b,c))

print("A root of the function g is given by", x_root)
```

```
A root of the function g is given by [-2.41421356]
```

We could have also stored the parameters a, b, c in an array so that g would have only had one additional input argument. This is illustrated below.

```
def g(x,coeff):
    return coeff[0]*x**2 + coeff[1]*x + coeff[2]

coeff = np.array([1,2,-1])

guess = -4
x_root = optimize.fsolve(g, x0=guess, args=(coeff))
```

```
print("A root of the function g is given by", x_root)
```

```
A root of the function g is given by [-2.41421356]
```

Bracket information

Another function that can find the root of a univariate function is `root_scalar()` from the `optimize` module.

Whereas `fsolve()` uses one fixed method in the background to find a root, `root_scalar()` allows the user to choose from a collection of methods. Some methods might perform better than others, depending on the type of function you are trying to find a root of. You can specify which method you want to use with the `method` keyword argument. Some methods require additional keyword arguments to be specified; see the documentation.

One such additional keyword argument is `bracket`, that allows you to specify the interval, or bracket, $[a, b]$ in which the root should be searched for. This does, however, come with the requirement that the function values in the points a and b should have a different sign: Either $f(a) < 0 < f(b)$ or $f(b) < 0 < f(a)$. The reason is that this guarantees, by the Intermediate Value Theorem, that there is at least one root in the interval $[a, b]$. If the bracket does not satisfy this condition, then Python will raise an error (try this yourself).

Let us look at an example where we use the Bisection method, called `'bisect'` in SciPy, with interval $[0, 4]$. The order in which you place the keyword argument `bracket` and `method` does not matter.

```
def f(x):
    return x**2 + 2*x - 1

interval = [0,4]

result = optimize.root_scalar(f, bracket=interval, method='bisect')
print(result)
```

```
      converged: True
           flag: 'converged'
 function_calls: 43
     iterations: 41
           root: 0.41421356237151485
```

Note that Python returns a lot of information regarding the root finding process. For example, it tells us whether the process has converged, meaning it found a point that satisfies $f(x) = 0$ up to a default precision. You can access these properties with the syntax `result.property_name` where `property_name` is the property of interest.

```
print("The root finding process converged:", result.converged)
```

```
The root finding process converged: True
```

For us, the most important property is `root`, which gives the value of the root.

```
print("A root of the function f is given by", result.root)
```

```
A root of the function f is given by 0.41421356237151485
```

Derivative information

If a function is differentiable, it is also possible to specify its derivative in some methods. This typically results in much faster root finding methods.

Recall, for example, Newton's method from Section Section 2.4 that finds a root by iteratively computing better approximations using the formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

starting from some initial guess x_0. The formula to compute the next iterate relies on

- The derivative $f'(x)$, and
- an initial guess x_0.

Let us look at an example of using Newton's method. The derivative f' should be defined as a function and can be entered in the `fprime` keyword argument. The initial guess is inputted in the `x0` keyword argument.

For our function $f(x) = x^2 + 2x - 1$ we have $f'(x) = 2(x + 1)$.

```
def f(x):
    return x**2 + 2*x - 1

def f_deriv(x):
    return 2*(x+1)

guess = 4
result = optimize.root_scalar(f,  method='newton', \
                              fprime=f_deriv, x0 = guess)
print(result)
```

```
     converged: True
          flag: 'converged'
 function_calls: 14
    iterations: 7
          root: 0.41421356237309503
```

Although it is a little bit like comparing apples and pears, the number of function calls and iterations (determining how long a method needs to converge) of Newton's method is much lower than that of the Bisection method.

Finally, we remark that the `args` keyword argument to specify additional input parameters can also be used in combination with methods that use bracket or derivative information.

6.1.2 Multivariate functions

Using `fsolve()`

The `fsolve()` function can also be used to compute a root of a system of n functions with n unknown variables. It again takes two input arguments:

- The system of function equations to be solved, and
- an initial guess for the (unknown) root.

The system of equations should be modelled as a Python function, i.e., we need a function that takes as input an array $x = [x_0, \ldots, x_{n-1}]$ and outputs the array $f(x) = [f_0(x), \ldots, f_{n-1}(x)]$. This function will then be the input for `fsolve()`.

As an example for $n = 2$, suppose we want to solve the system

$$\begin{cases} x_0^2 + x_1^2 &= 4 \\ x_0 + x_1 &= 1 \end{cases}.$$

That is, we have $f_0(x_0, x_1) = x_0^2 + x_1^2 - 4$ and $f_1(x_0, x_1) = x_0 + x_1 - 1$, and want to solve $f_0(x) = 0, f_1(x) = 0$. The array $[f_0(x), f_1(x)]$ can be defined as a function in the following way.

```
def f(x):
    # Input  : Array x = [x_0, x_1]
    # Output : Array f = [f_0(x), f_1(x)]

    f = np.array([x[0]**2 + x[1]**2 - 4, x[0] + x[1] - 1])
    return f
```

We emphasize that here the array $x = [x_0, x_1]$ is the input of the function, and not x_0 and x_1 separately. If we would define f as a function of two inputs, i.e., `f(x_0,x_1)`, then `fsolve()` would want to find a root with respect to its first argument `x_0` only, which is not what we want.

Using `fsolve()` to do the root finding gives us the following solution.

```
guess = np.array([1,1]) # Our initial guess
root = optimize.fsolve(f, x0=guess)

print(root)
```

```
[ 1.82287566 -0.82287566]
```

Note that the initial guess is an array in $\mathbb{R}^2$ this time, as we are considering a function with two variables.

Also here we can use the `args` keyword argument to specify additional input parameters. Suppose we want to solve, for $a = 2$ and $b = 4$, the system

$$\begin{cases} a \cdot x_0^2 + x_1^2 &= 4 \\ x_0 + b \cdot x_1 &= 1 \end{cases}.$$

```
def f(x,a,b):
    return np.array([a*x[0]**2 + x[1]**2 - 4, x[0] + b*x[1] - 1])

guess = np.array([1,1]) # Our initial guess
a, b = 2, 4
x_root = optimize.fsolve(f, x0=guess, args=(a,b))

print(x_root)
```

```
[ 1.41233385 -0.10308346]
```

You can double-check that the root x^* you found is indeed a root by plugging the solution into the system of equations, i.e., checking if it satisfies

$$f(x^*) = [f_0(x^*), \dots, f_{n-1}(x^*)] = [0, \dots, 0].$$

```
print(f(x_root,a,b)) # Both coordinates approximately equal to zero
```

```
[9.59232693e-14 0.00000000e+00]
```

Derivative information

Just as in the univariate case, it is also possible to use other function for finding a root. The analogue of `root_scalar()` is the function `root()`. Although it is not possible to input bracket information for

this function, it does have methods that use derivate information. These methods are typically faster than `fsolve()`.

Consider again the system

$$\begin{cases} x_0^2 + x_1^2 - 4 &= 0 \\ x_0 + x_1 - 1 &= 0 \end{cases}.$$

The "derivative" of the function $f : \mathbb{R}^n \to \mathbb{R}^n$ given by $f(x) = [f_0(x), \ldots, f_{n-1}(x)]$ is the Jacobian matrix

$$J(f) = \frac{\partial(f_0, \cdots, f_{n-1})}{\partial(x_0, \cdots, x_{n-1})} = \begin{bmatrix} \frac{\partial f_0}{\partial x_0} & \frac{\partial f_0}{\partial x_1} & \cdots & \frac{\partial f_0}{\partial x_n} \\ \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_{n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_{n-1}}{\partial x_0} & \frac{\partial f_{n-1}}{\partial x_1} & \cdots & \frac{\partial f_{n-1}}{\partial x_{n-1}} \end{bmatrix}.$$

For the two equations at hand, we have

$$J(f) = \begin{bmatrix} 2x_0 & 2x_1 \\ 1 & 1 \end{bmatrix}.$$

Just as in the univariate case when using Newton's method, we need to input an initial guess and the Jacobian matrix, where the latter should be defined as a function. The Jacobian is inputted in the `jac` keyword argument; the initial guess in `x0`.

```
def f(x):
    return np.array([x[0]**2 + x[1]**2 - 4, x[0] + x[1] - 1])

def jac_f(x):
    J = np.array([[2*x[0], 2*x[1]],[1,1]])
    return J

guess = np.array([1,1]) # Our initial guess
result = optimize.root(f, jac=jac_f,x0=guess)

print(result)
```

```
message: The solution converged.
success: True
 status: 1
    fun: [ 0.000e+00  0.000e+00]
      x: [ 1.823e+00 -8.229e-01]
   nfev: 27
   njev: 2
   fjac: [[-9.745e-01 -2.243e-01]
          [-2.243e-01  9.745e-01]]
      r: [-4.458e+00  6.983e-01  1.187e+00]
    qtf: [ 1.848e-10  4.255e-11]
```

We can access the properties of the result of the root finding procedure with the syntax `result.property_name` where `property_name` is the property of interest. Here the root is given by the property `x`.

```
print("A root of the function f is given by", result.x)
```

```
A root of the function f is given by [ 1.82287566 -0.82287566]
```

Note that this is the same root that we found with `fsolve()`.

6.1.3 Least squares method

If the number of functions is not equal to the number of variables, or if the system does not have a solution, the least squares method `optimize.least_squares` is your best pick to find a root.

The function (when using the default settings) tries to compute a point x that attains the minimum of the residual function

$$R(f_0, \dots, f_{n-1}) = \min_{x \in \mathbb{R}^n} \sum_{i=1}^{n-1} f_i(x)^2,$$

It should be observed that $R(f_0, \dots, f_{n-1}) \geq 0$ and $R(f_0, \dots, f_{n-1}) = 0$ if and only if the system $f_0(x) = 0, \dots, f_{n-1}(x) = 0$ has a root.

The function `optimize.least_squares` takes just like `fsolve()` two input arguments: The system of equations and an initial guess.

Let's look at an example. Consider the system

$$\begin{cases} f_0(x) &= \sin(x) &= 0 \\ f_1(x) &= (x - \pi + 0.1)^2 &= 0 \\ f_2(x) &= (x - \pi) &= 0 \end{cases}.$$

Without the 0.1-term, this system would have the unique solution $x = \pi$, but this small perturbation causes it to become infeasible. We can still find an approximately optimal solution with the least squares method. The number π can be accessed in NumPy using `np.pi`.

```
def system(x):
    return np.array([np.sin(x[0]), (x[0]-np.pi + 0.1)**2, x[0] - np.pi])

guess = 1
result = optimize.least_squares(system,x0=guess)

print("Least squares solution found is x = ", result.x) # Close to pi
```

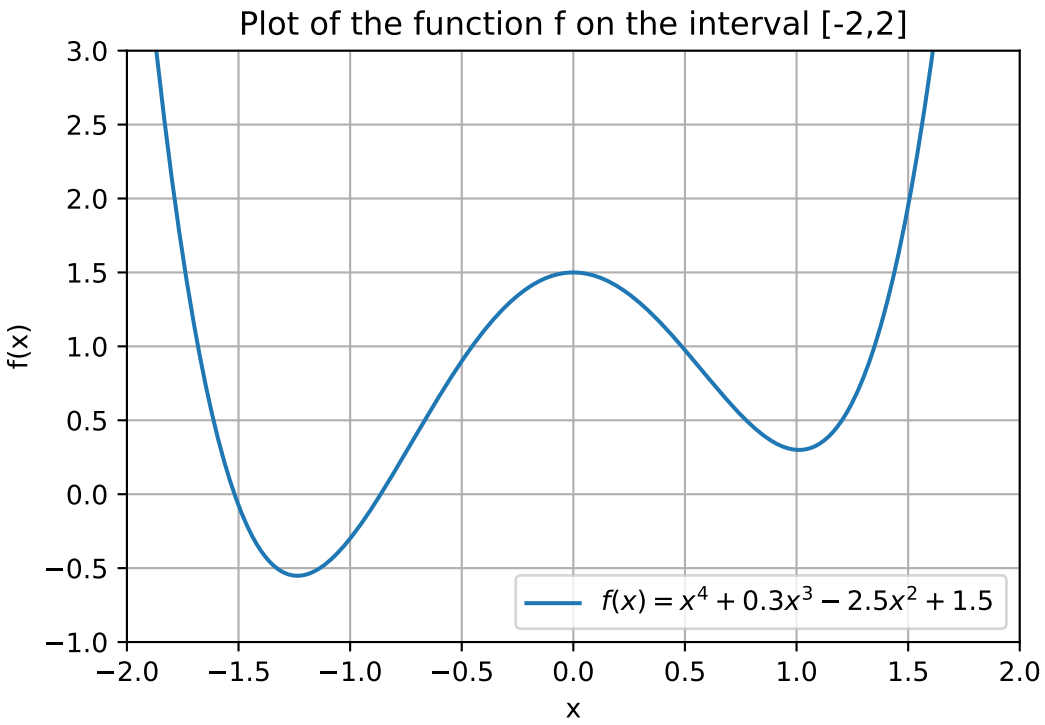
```
Least squares solution found is x =  [3.1406215]
```

Although `optimize.least_squares` often works well, it does not necessarily compute a root of the function even if one exists. What it actually does, is compute a local minimum of the residual function.

Let us illustrate this with an example. Consider the function $f(x) = x^4 + 0.3x^3 - 2.5x^2 + 1.5$, which has two (real) roots as can be seen from the figure below.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the x range
x = np.linspace(-2, 2, 600)

# Define the function f
def f(x):
    return x**4 + 0.3*x**3 - 2.5*x**2 + 1.5

# Create the plot
plt.figure(figsize=(6, 4))
```

```
plt.plot(x, f(x), label='$f(x) = x^4 + 0.3x^3 - 2.5x^2 + 1.5$')

# Add labels and title
plt.title('Plot of the function f on the interval [-2,2]')
plt.xlabel('x')
plt.ylabel('f(x)')

# Add a grid
plt.grid(True)

# Set range
plt.xlim(-2,2)
plt.ylim(-1,3)

# Add a legend
plt.legend()

# Show the plot
plt.show()
```



We execute both Newton's and the least squares method with the same initial guess.

```
# Function f
def f(x):
    return (x**4 + 0.3*x**3 - 2.5*x**2 + 1.5)

# Derivative of f
def f_deriv(x):
```

```
    return 4*x**3 + 0.9*x**2 -5*x

guess = 1.5

# Newton's method
result = optimize.root_scalar(f,  method='newton', \
                              fprime=f_deriv, x0 = guess)
print("Root found by Newton's method: \n x =", result.root,
        f"with f(x) =", f(result.root))

# Least squares method
result_ls = optimize.least_squares(f, x0=guess)
print("Root found by least squares method: \n x =", result_ls.x,
        f"with f(x) =", f(result_ls.x))
```

```
Root found by Newton's method:
 x = -1.5180670079327394 with f(x) = -8.881784197001252e-16
Root found by least squares method:
 x = [1.01118148] with f(x) = [0.29943799]
```

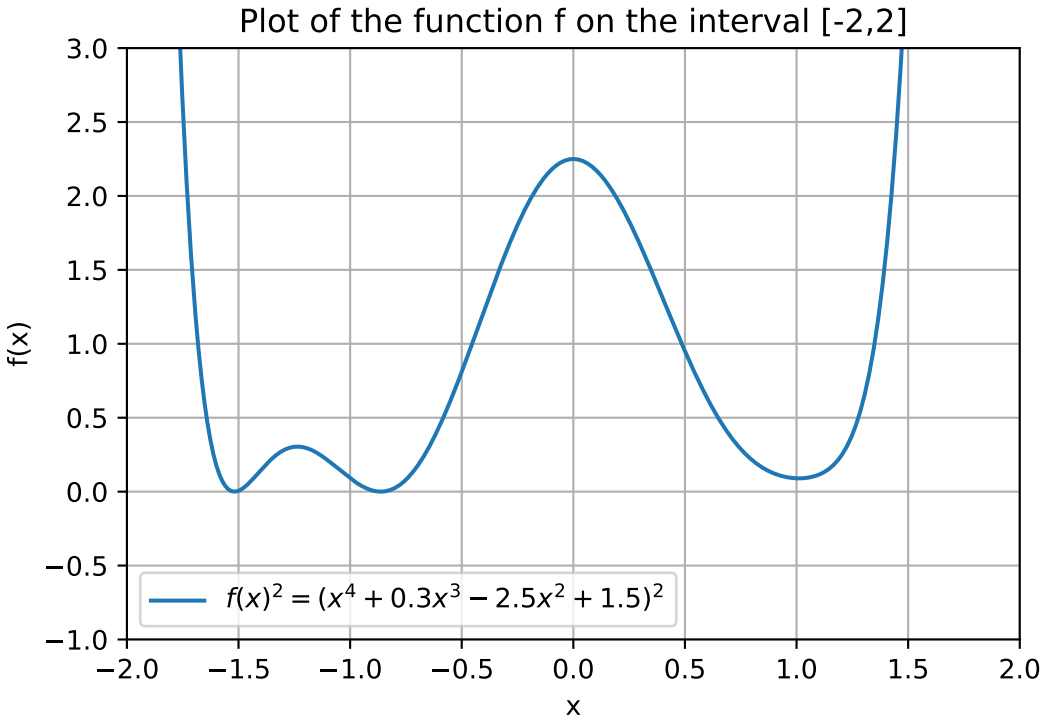
As you can see, the least squares method does not find a root, because the function value in the computed point is almost 0.3. What goes wrong here? Consider the residual problem

$$R(f) = \min_x (x^4 + 0.3x^3 - 2.5x^2 + 1.5)^2.$$

The function $g(x) = f(x)^2 = (x^4 + 0.3x^3 - 2.5x^2 + 1.5)^2$ is plotted below. As you can see, it has a local minimum around 1. Because we started out with the initial guess 1.5, the least squares method gets stuck in this local minimum.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the x range
x = np.linspace(-2, 2, 600)

# Define the function f
def f(x):
    return (x**4 + 0.3*x**3 - 2.5*x**2 + 1.5)**2

# Create the plot
plt.figure(figsize=(6, 4))
plt.plot(x, f(x), label='$f(x)^2 = (x^4 + 0.3x^3 - 2.5x^2 + 1.5)^2$')

# Add labels and title
plt.title('Plot of the function f on the interval [-2,2]')
plt.xlabel('x')
plt.ylabel('f(x)')

# Add a grid
plt.grid(True)

# Set range
plt.xlim(-2,2)
```

```
plt.ylim(-1,3)

# Add a legend
plt.legend()

# Show the plot
plt.show()
```



To close this section, we again emphasize that no method is guaranteed to always find a root for every function. For example, `fsolve()` is also not able to find a root of the function f in this section (try this yourself).

Therefore, always check whether a found solution is actually a root by evaluating the function in the solution found and checking if the resulting value is (almost) zero.

6.2 Nonlinear optimization

In this section we consider the minimization problem

$$\min_{x \in \mathbb{R}^n} f(x).$$

for a function $f : \mathbb{R}^n \to \mathbb{R}$. Note that maximization can be done by considering the function $-f$.

We again start with univariate functions and then switch to multivariate functions later on. In the last section we will also see some examples of constrained optimization, where the domain of f is a subset of $\mathbb{R}^n$. The syntax for using minimization functions from SciPy will be similar to the syntax we saw for root finding methods.

In particular, also here we can use the `args` keyword argument to specify any additional input arguments of the function we want to minimize over, so that the optimization can happen with respect to the unknown first input argument of the function.

6.2.1 Univariate function

If $f : \mathbb{R} \to \mathbb{R}$ is a univariate function then the easiest-to-use functions for minimization are `fmin()` and `minimize_scalar()` from the `optimize` module. We emphasize that these functions find a local minimum of the function f, which might or might not be a global minimum.

The function `fmin()` works similar as `fsolve()` in that it requires an initial guess.

```
def f(x):
    return x**4 + 0.3*x**3 - 2.5*x**2 + 1.5

guess=1.5
x_min = optimize.fmin(f,x0=guess)

print("The minimum found is x =", x_min) # Local minimum
```

```
Optimization terminated successfully.
         Current function value: 0.299438
         Iterations: 15
         Function evaluations: 30
The minimum found is x = [1.01118164]
```

With a different initial guess the method is able to find the global minimum; see the figure in the previous section. Furthermore, you can suppress the output message of `fmin()` by setting the keyword argument `disp=False`; see the documentation of `fmin()`.

```
guess=-1.5
x_min = optimize.fmin(f,x0=guess,disp=False)

print("The minimum found is x =", x_min) # Local minimum
```

```
The minimum found is x = [-1.23618164]
```

The function `minimize_scalar()` gives more flexibility in terms of input arguments. First of all, we can execute it without any additional arguments, but then it more easily gets stuck in a local minimum.

```
result = optimize.minimize_scalar(f)
print(result)

# We access root with result.x
print("The minimum found  is x =", result.x)
```

```
 message:
          Optimization terminated successfully;
          The returned value satisfies the termination criteria
          (using xtol = 1.48e-08 )
 success: True
     fun: 0.29943799106751934
       x: 1.011179780213775
     nit: 11
    nfev: 14
The minimum found  is x = 1.011179780213775
```

We can also set an interval in which we want to find a minimum using the `bounds` keyword argument. Note that in the example below, the function `minimize_scalar()` does not get stuck in the local minimum around 1.

```
interval = [-2,2]
result = optimize.minimize_scalar(f,bounds=interval)

print("The minimum found  is x =", result.x)
```

```
The minimum found  is x = -1.2361799028533706
```

For more options and different methods that can be used with the `method` keyword argument, see the documentation of `minimize_scalar`.

6.2.2 Multivariate function

For multivariate functions, we can use the `minimize()` function from `optimize`, being the analogue of `root()` for root finding. This function can take into account derivative, or gradient, information with certain methods. The gradient of f is given by

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_0}, \ldots, \frac{\partial f}{\partial x_{n-1}}\right].$$

We can input the gradient again with the keyword argument `jac`. The default method that `minimize()` uses is `'BFGS'` and this method can use gradient information. You can choose your own method with the `method` keyword argument; see the documentation.

```
# Define the function
def f(x):
    return (x[0] - 3)**2 + (x[1] + 4)**2 - 1

# Define the gradient
def grad_f(x):
    return np.array([2 * (x[0] - 3), 2 * (x[1] + 4)])

# Initial guess
guess = np.array([0.0, 0.0])

# Minimization with gradient information
result = optimize.minimize(f, x0=guess, jac=grad_f)

# Print the result
print("The minimum found is:", result.x)
print("Function value at minimum:", result.fun)
```

```
The minimum found is: [ 3. -4.]
Function value at minimum: -1.0
```

6.2.3 Constrained optimization

Just as with `minimize_scalar()` the function `minimize()` can also include interval information using the `bounds` keyword argument. The syntax for this argument is a list of tuples that for every variable in $x = [x_0, \ldots, x_{n-1}]$ contains the lower and upper bound value for the variable. Recall that this is the same way how variable bounds were specified in the `linprog` package for solving (integer) linear optimization problems.

That is, if we have the constraints $\ell_i \leq x_i \leq u_i$ for $i = 0, \ldots, n-1$, then the input for `bounds` is `[(l_0, u_0), (l_1, u_1), ..., (l_{n-1},u_{n-1})]`. Just as in `linprog` you can set an upper or lower bound equal to `None` to model that the lower bound is $-\infty$, or that the upper bound is $+\infty$.

```python
# Define the function
def f(x):
    return (x[0] - 3)**2 + (x[1] + 4)**2 - 1

# Define the gradient
def grad_f(x):
    return np.array([2 * (x[0] - 3), 2 * (x[1] + 4)])

# Initial guess
guess = np.array([0.0, 0.0])
intervals = [(None,2), (-2,None)] # x_0 <= 2, x_1 >= -2

# Minimization with gradient information
result = optimize.minimize(f, x0=guess, bounds=intervals)

# Print the result
print("The minimum found is:", result.x)
print("Function value at minimum:", result.fun)
```

```
The minimum found is: [ 2. -2.]
Function value at minimum: 4.0
```

Next to specifying variable bounds, `minimize()` can also take into account more complex constraints that restrict the domain of f, i.e., the search space of `minimize()`, using the `constraints` keyword argument.

That is, we can solve the problem

$$\begin{array}{lll} \min_{x\in\mathbb{R}^n} & f(x) & \\ \text{s.t.} & g_j(x) \geq 0 & \text{for } j = 1, \dots, q-1 \\ & h_k(x) = 0 & \text{for } k = 0, \dots, r-1 \end{array} .$$

The syntax for adding constraints is to use a tuple containing dictionaries, where each dictionary models a constraints with keys

- `'type'`: Use `'ineq'` for a $\geq$-constraint and `'eq'` for an $=$- constraint;
- `'fun'` : Python function that describes g_j or h_k.

There is also the optional keyword argument `'args'` that allows you to specify additional parameters that appear in the function g_j or h_k.

As an example, suppose we want to solve the problem

$$\begin{array}{ll} \min_{x\in\mathbb{R}^n} & x_0^2 + 5x_1^2 \\ \text{s.t.} & x_0 + x_1 \geq 5 \\ & x_0 - 2x_1 = 3 \end{array}$$

Then we have $g_0(x) = x_0 + x_1 - 5$ and $h_0(x) = x_0 - 2x_1 - 3$.

```python
# Define functions of constraints
def g(x):
    return x[0] + x[1] - 5

def h(x):
    return x[0] - 2*x[1] - 3
```

```
# Define tuple containing constraints as dictionaries
cons = ({'type' : 'ineq', 'fun' : g},
        {'type' : 'eq', 'fun' : h})
```

Let us now do the optimization using these constraints.

```
def f(x):
    return x[0]**2 + x[1]**2

guess = np.array([0,0]) # minimize() always needs guess

result = optimize.minimize(f, x0=guess, constraints=cons)

print("The minimum found is:", result.x)
print("Function value at minimum:", result.fun)
```

```
The minimum found is: [4.33333333 0.66666667]
Function value at minimum: 19.2222222222229
```

This allows us, for example, to optimize a nonlinear function subject to linear constraints.

6.2.4 Remarks

Just as in the previous chapter, we remark here that there are various other packages in Python to perform optimization tasks with. For example, there is `cvxpy` for convex optimization, and you can also couple external solvers with Python, as explained in the last chapter.

Chapter 7

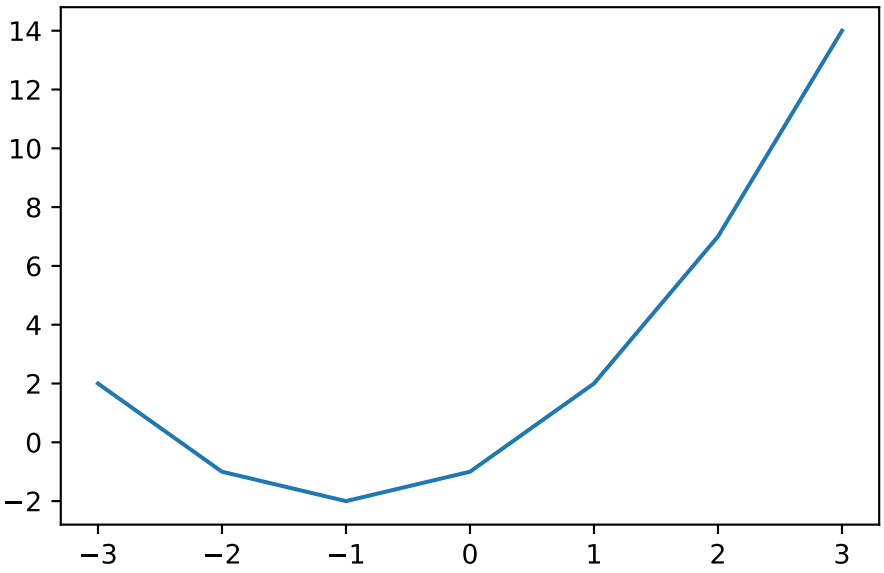
Visualization

In this chapter we will explain the basics for plotting functions and data, for which we will use the `matplotlib.pyplot` (sub)package. We import it under the name `plt`. You might wonder why we use the name `plt` and not the perhaps more obvious choise `plot`. This is because `plot()` is a command that we will be using, so we do not want to create any conflicts with this function when executing a Python script.

7.1 Basic plotting

In this section we will explain step-by-step how to generate the figure that we have seen in the previous two sections. We start with plotting the function $f(x) = x^2 + 2x - 1$ for some values of x in a two-dimensional figure.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Define the x range of x-values
x = np.array([-3,-2,-1,0,1,2,3])

# Compute the function values f(x[i]) of the elements x[i]
# and store them in the array y
y = f(x)

#Create the figure
plt.figure()

# Create the plot
plt.plot(x, y)

# Show the plot
plt.show()
```



You can view the figure in the Plots pane (or tab) in Spyder.

> If the resolution of the plots in the Plots pane is bad, you can increase it by going to "Tools > Preferences > IPython console > Graphics > Inline backend > Resolution" and set the resolution to, for example, 300 dpi.
>
> You can get the Plots pane in fullscreen by going to the button with the three horizontal lines in the top right corner and choose "Undock". You can "Dock" the pane again as well if you want to leave the fullscreen mode.

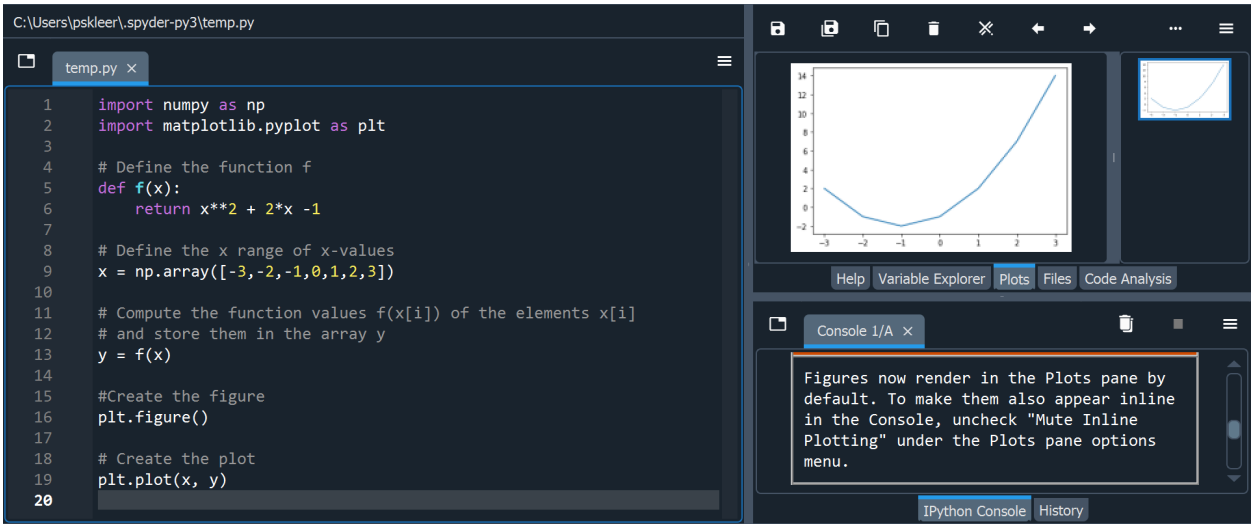


Figure 7.1: IPython Console

We will next explain what the code above is doing. After defining the function f, we create the vector (i.e., Numpy array)

$$x = [x_1, x_2, x_3, x_4, x_5, x_6, x_7] = [-3, -2, -1, 0, 1, 2, 3].$$

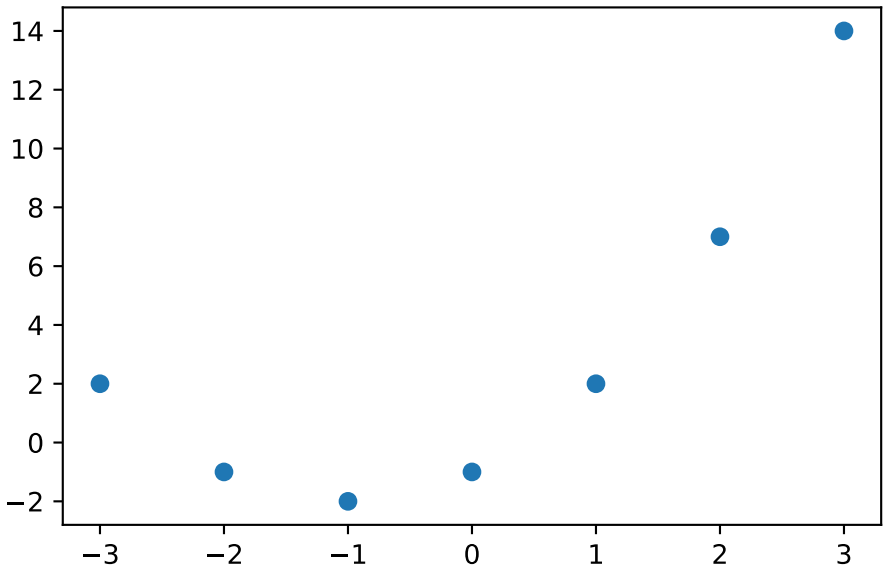
Because the function f is vectorized, we can right away compute all the function values in these points. We store them in the array `y = f(x)`, that is,

$$\begin{aligned} y = f(x) &= [f(x_1), f(x_2), f(x_3), f(x_4), f(x_5), f(x_6), f(x_7)] \\ &= [2, -1, -2, -1, 2, 7, 14]. \end{aligned}$$

Next, we create an (empty) figure using the command `plt.figure()`. Then comes the most important command, `plt.plot(x,y)`, that plots the elements in the vector x against the elements in the vector $y = f(x)$, and connects consecutive combinations (x_i, y_i) and (x_{i+1}, y_{i+1}) with a line segment. For example, we have $(x_1, y_1) = (-3, 2)$ and $(x_2, y_2) = (-2, -1)$. The left most line segment is formed by connecting these points.

If you only want to plot the points (x_i, y_i), and not the line segments, you can use `plt.scatter(x,y)` instead of `plt.plot(x,y)`.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Define the x range of x-values
x = np.array([-3,-2,-1,0,1,2,3])

# Compute the function values f(x[i]) of the elements x[i]
# and store them in the array y
y = f(x)

#Create the figure
plt.figure()

# Create the plot
plt.scatter(x, y)
```

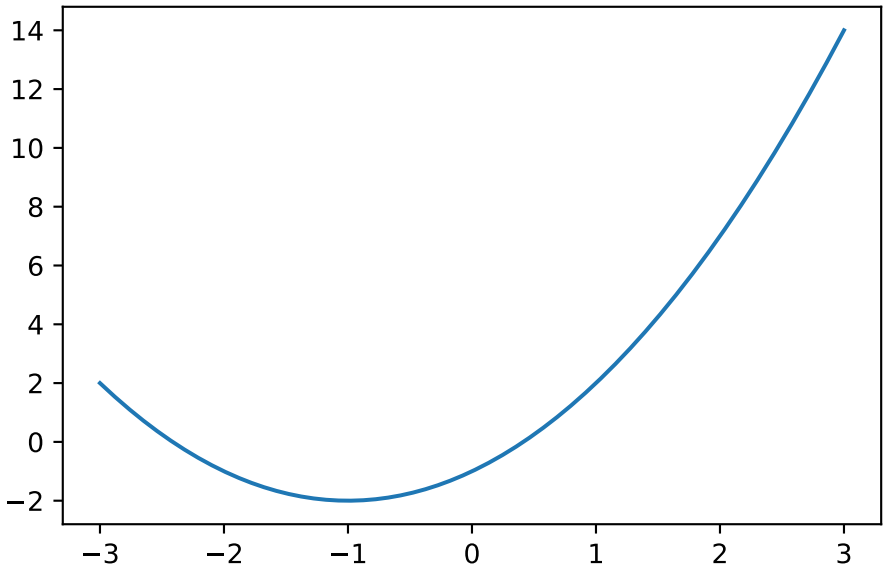


Observe that the (blue) line in the figure that was generated using `plt.plot(x,y)` is not as "smooth" as in the figures in the previous sections, where the function does not (visibly) have these segments. To get a

smoother function line, we can include more points in the vector x. This can be done, for example, with the `linspace()` function that we have seen in Chapter 3.

Let us plot again the function f, but this time with 600 elements in x in the interval $[-3, 3]$. We use `plt.plot()` again, instead of `plt.scatter()`. We now obtain a much smoother function line.

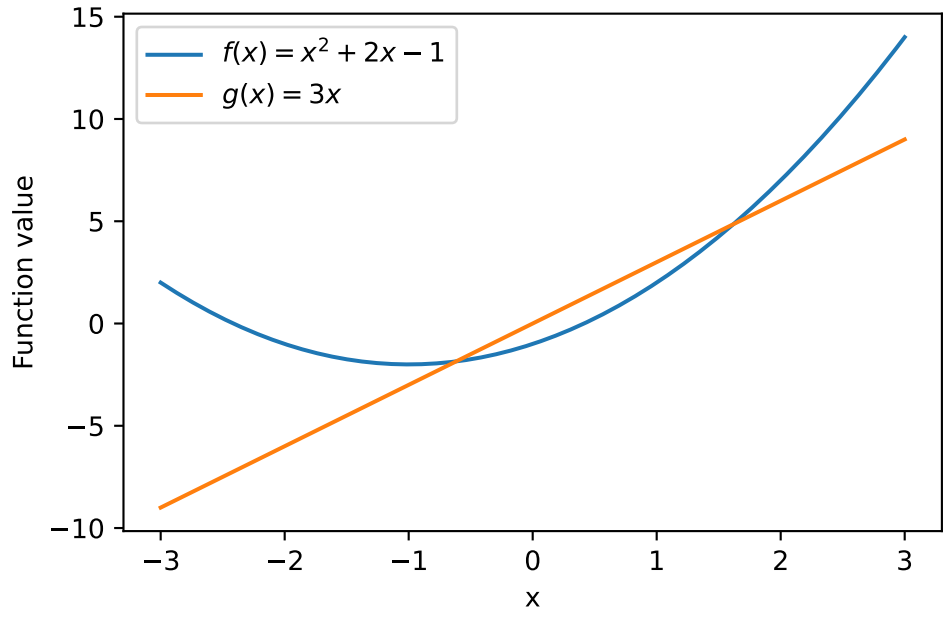
```
import numpy as np
import matplotlib.pyplot as plt

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Define the x range of x-values
x = np.linspace(-3,3,600)

# Compute the function values f(x[i]) of the elements x[i]
# and store them in the array y
y = f(x)

#Create the figure
plt.figure()

# Create the plot
plt.plot(x, y)
```



You can add a legend for the line/points that you plot by using the label argument of `plt.plot()`. For example we can add the function description using `plt.plot(x,y,label='$f(x) = x^2 + 2x - 1$')`. This is in particular useful if you plot multiple functions in one figure, as the example below illustrates. There we plot the functions f and g, with $g(x) = 3x$ a new function. To have the labels appear in the legend of the figure, you need to add a legend to the figure with `plt.legend()`.

If you want to add labels to the horizontal and vertical axis, you can use the commands `plt.xlabel()` and `plt.ylabel()`.

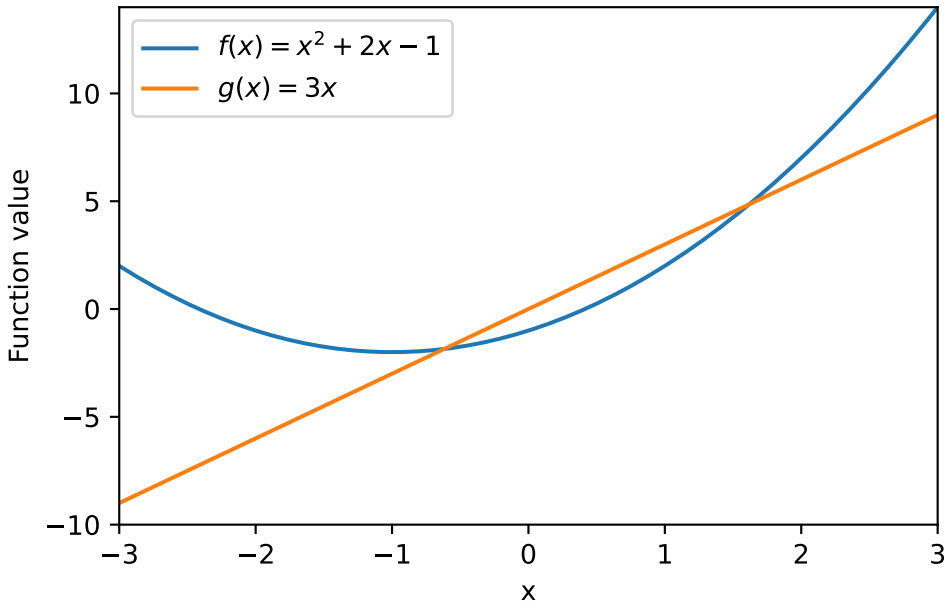
```
import numpy as np
import matplotlib.pyplot as plt

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Define the function g
def g(x):
    return 3*x

# Define the x range of x-values
x = np.linspace(-3,3,600)

# Compute the function values f(x[i]) of the elements x[i]
# and store them in the array y
y = f(x)
z = g(x)

#Create the figure
plt.figure()

# Create the plot
plt.plot(x, y, label='$f(x) = x^2 + 2x - 1$')
plt.plot(x, z, label='$g(x) = 3x$')

# Create labels for axes
plt.xlabel('x')
plt.ylabel('Function value')

# Create the legend with the specified labels
plt.legend()
```



You might observe that the range on the vertical axis changed now that we added a second function to the plot. When we only plotted the function f, the vertical axis ranged from -2 to 14, but now with the function g added to it, it ranges from -10 to 15.

You can fix the range $[c, d]$ on the vertical axis using the command `plt.ylim(c,d)`, and to fix the range of the horizontal axis to $[a, b]$, you can use `plt.xlim(a,b)`. In the figure below, we fix the vertical range to $[c, d] = [-10, 14]$ and the horizontal axis to $[a, b] = [-3, 3]$.

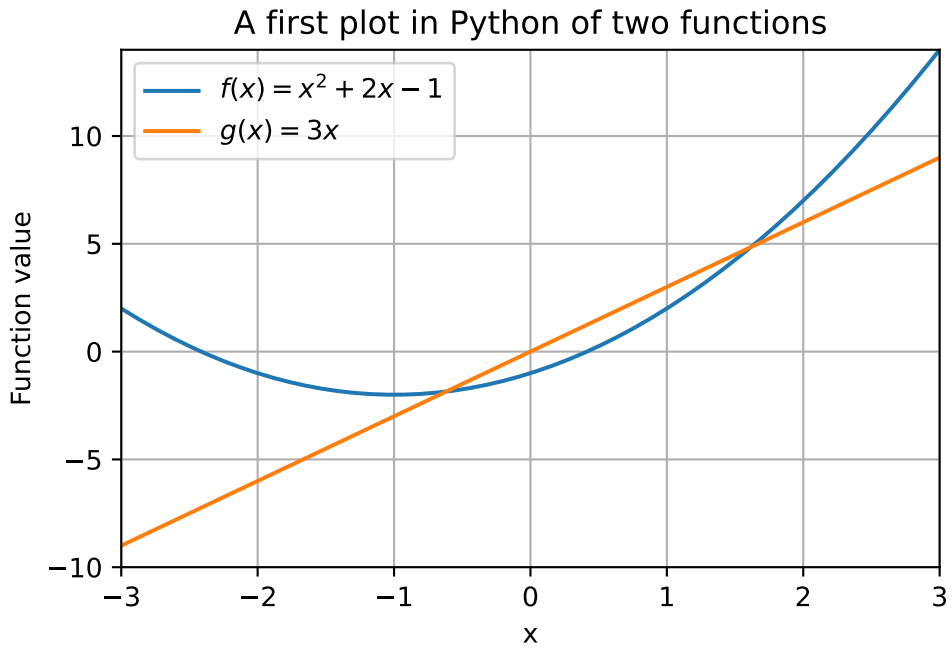
```
import numpy as np
import matplotlib.pyplot as plt

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Define the function g
def g(x):
    return 3*x

# Define the x range of x-values
x = np.linspace(-3,3,600)

# Compute the function values f(x[i]) of the elements x[i]
# and store them in the array y
y = f(x)
z = g(x)

#Create the figure object
plt.figure()

# Create the plot within the figure
plt.plot(x, y, label='$f(x) = x^2 + 2x - 1$')
plt.plot(x, z, label='$g(x) = 3x$')

# Create labels for axes
plt.xlabel('x')
plt.ylabel('Function value')

# Create the legend with the specified labels
plt.legend()

# Fix the range of the axes
plt.xlim(-3,3)
plt.ylim(-10,14)
```



Finally, you can also add a title to the plot using the command `plt.title()` as well as a grid in the background of the figure using `plt.grid()`. These are illustrated in the figure below.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Define the function g
def g(x):
    return 3*x

# Define the x range of x-values
x = np.linspace(-3,3,600)

# Compute the function values f(x[i]) of the elements x[i]
# and store them in the array y
y = f(x)
z = g(x)

#Create the figure
plt.figure()

# Create the plot
plt.plot(x, y, label='$f(x) = x^2 + 2x - 1$')
plt.plot(x, z, label='$g(x) = 3x$')

# Create labels for axes
plt.xlabel('x')
plt.ylabel('Function value')
```

```
# Create the legend with the specified labels
plt.legend()

# Fix the range of the axes
plt.xlim(-3,3)
plt.ylim(-10,14)

# Add title to the plot
plt.title('A first plot in Python of two functions')

# Add grid to the background
plt.grid()
```



This completes the description of the basics of plotting a figure. As a final remark, there are many more plotting options that we do not cover here, but which can be found in the documentation. For example, with the `plt.xticks()` and `plt.yticks()` commands you can specify the numbers you want to have displayed on the horizontal and vertical axis, respectively. Also, there are commands to specify line color, width, type (e.g., dashed) and much more!

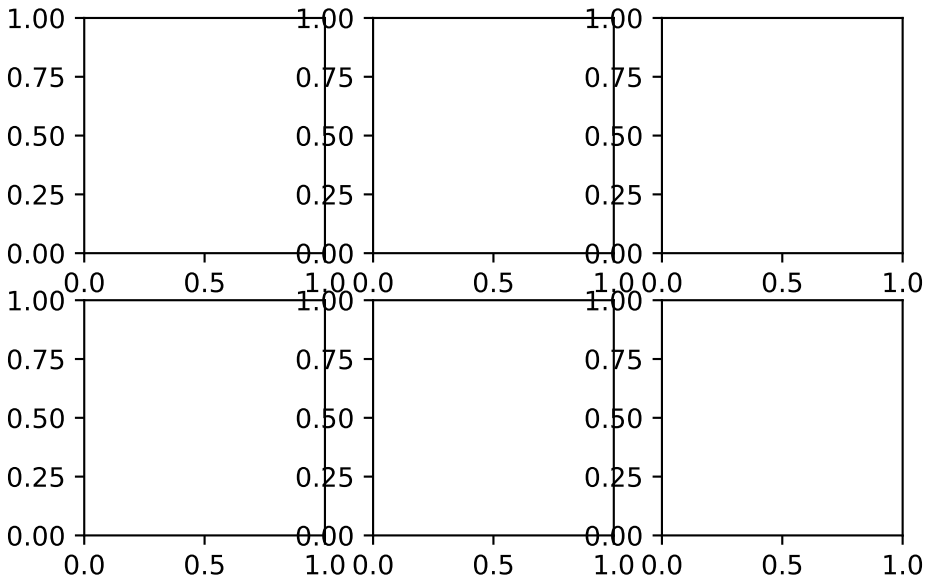
7.2 Subplots

In this section we will describe how you can create multiple subplots in one figure using the `subplots()` function. There are various ways to do this. You can to this in a predefined grid or on a plot-by-plot basis.

7.2.1 Fixed grid

We start with explaining the basics of the `subplots()` functoin. The syntax for creating a figure with a predefined grid on which plots can be placed is as follows.

```
m = 2
n = 3

# Create figure with six subplots in an n x m fashion
fig, ax = plt.subplots(m,n)

plt.show()
```



This creates a figure object, with name `fig` in this case, and a 2×3 array `ax` with so-called Axes objects. We are going to fill in the plots on the positions of the `ax` array.

```
print(np.shape(ax))
```

```
(2, 3)
```

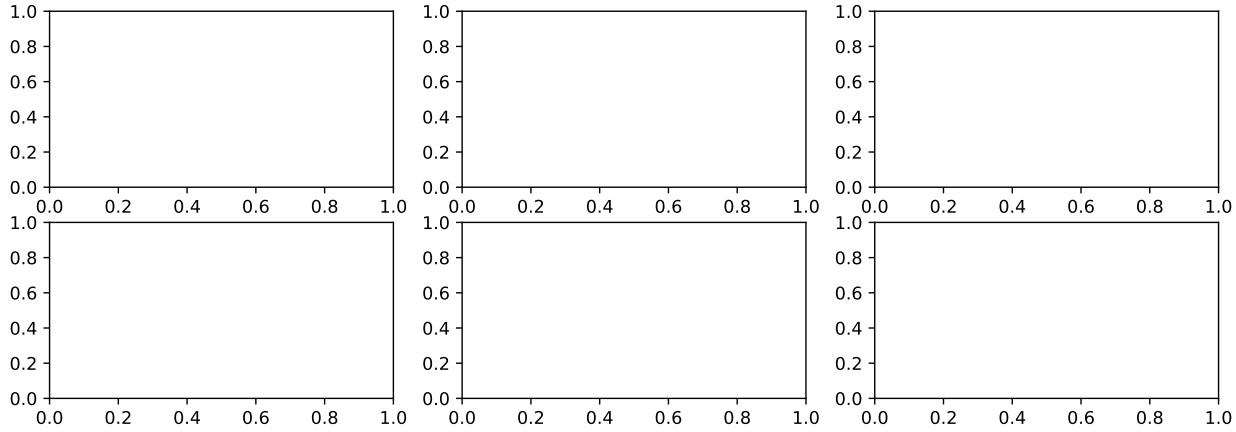
The fact that arrays can also store other objects besided numbers, is something we also already saw when using the `pulp` package for linear optimization in Chapter 5.

One might argue that the figure above is visually not very appealing, especially because the horizontally adjacent plots are very close to each other. You can get more control over the size of the figure (in which the plots are placed) by using the `figsize` keyword whose argument should be a tuple (w, h) indicating the width w and the height h of the figure.

Note that the input arguments of `figsize` are perhaps a bit counterintuitive, as for the shape of a NumPy array (like the command above) the first number is always the "height" of the matrix, and the second number the "width", but this is the other way around for the shape of a figure.

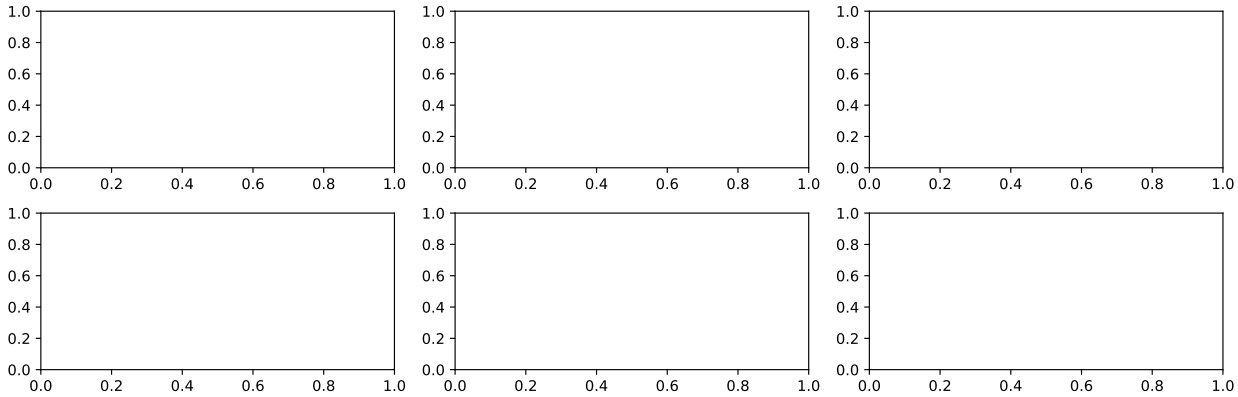
```
# Parameters for figure with n x m subplots
m, n = 2, 3

# Parameters w (width) and h (height) for figure size
w, h = 12, 4

# Create figure with six subplots in a 2 x 3 fashion
```

```
fig, ax = plt.subplots(m,n, figsize=(w,h))

plt.show()
```



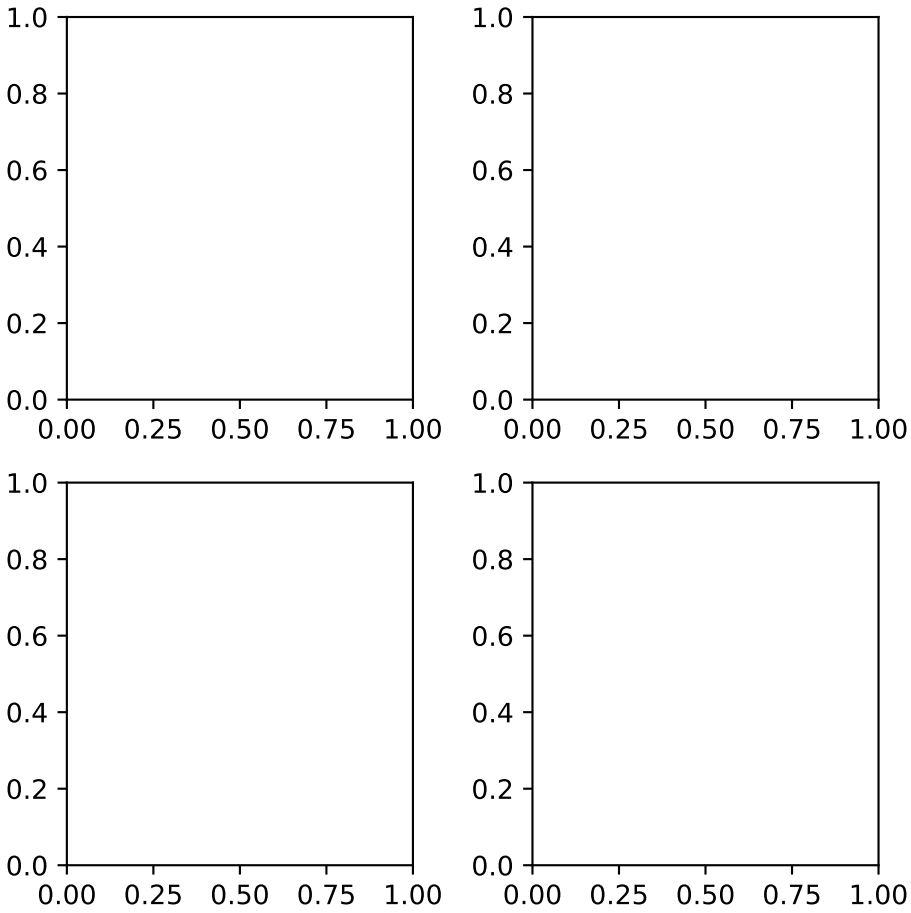
It usually also helps to put in the command `plt.tight_layout()` that prevents plots within a figure from overlapping by addings some spacing between them.

```
# Parameters for figure with n x m subplots
m, n = 2, 3

# Parameters w (width) and h (height) for figure size
w, h = 12, 4

# Create figure with six subplots in a 2 x 3 fashion
fig, ax = plt.subplots(m,n, figsize=(w,h))

plt.tight_layout()

plt.show()
```



We continue by explaining how you can add plotting information to the indiviual plots in the figure. For this, we will switch to a figure with a 2×2 array for in total four subplots.

```
# Parameters for figure with n x m subplots
n = 2

# Parameters w (width) and h (height) for figure size
w = 5

# Create figure with six subplots in a 2 x 2 fashion
fig, ax = plt.subplots(n,n, figsize=(w,w))

# Tighten layout
plt.tight_layout()

# Show plot
plt.show()
```

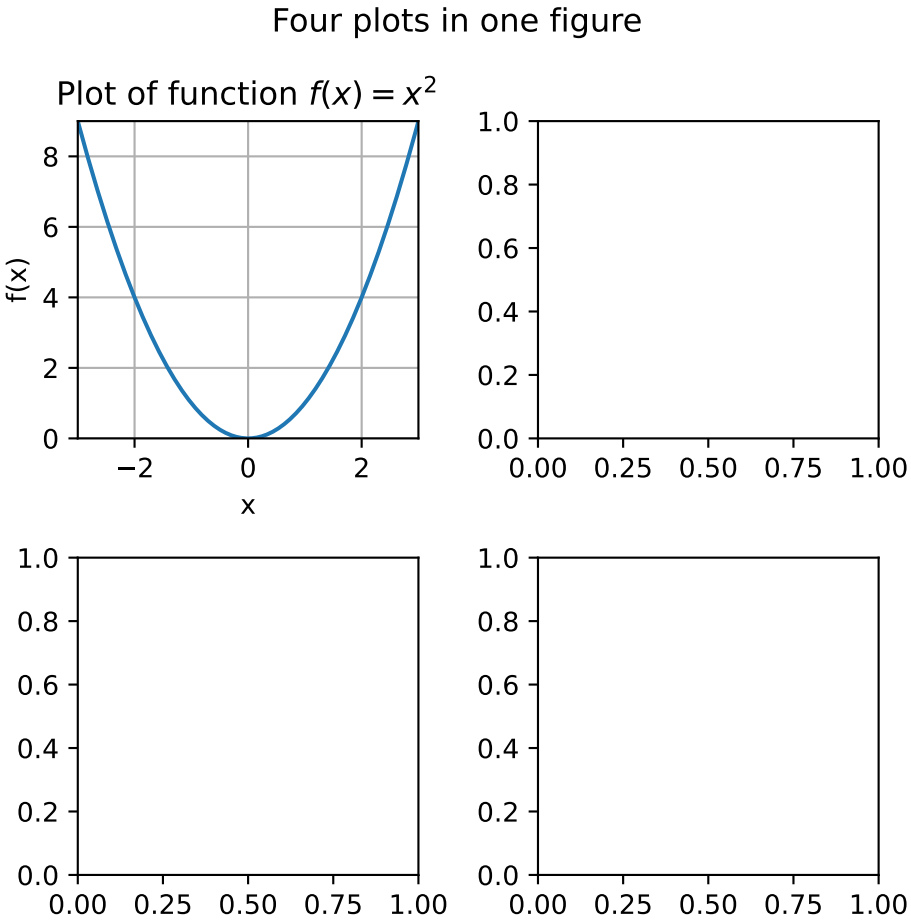


You can access the individual plot at position (i, j) of the array `ax` using `ax[i,j]`, and set properties of it using `ax[i,j].plot_option` where `plot_option` is a plotting command.

Sometimes you need to use a slightly different command than when you plot a single plot a figure. For many commands, you need to add `set_` to it. For example, instead of `plt.xlim(a,b)` you need to use `ax[i,j].set_xlim(a,b)`.

To set the title of the whole figure you can use `fig.suptitle()` instead of `plt.title()`.

```
# Define function to plot
def f(x):
    return x**2

a = -3
b = 3

# Define x-range
x = np.linspace(a,b,600)

# Create figure with six subplots in a 2 x 2 fashion
fig, ax = plt.subplots(n,n, figsize=(w,w))

# Title of whole figure
fig.suptitle("Four plots in one figure")

# Tighten layout
plt.tight_layout()

# Create plot on top-left position
ax[0,0].plot(x,f(x))
ax[0,0].set_xlim(a,b)
ax[0,0].set_ylim(0,9)
ax[0,0].set_xlabel("x")
ax[0,0].set_ylabel("f(x)")
ax[0,0].set_title("Plot of function $f(x) = x^2$")
ax[0,0].grid()

# Tighten layout
plt.tight_layout()

# Show plot
plt.show()
```



7.2.2 Iterative adding

Instead of predefining a grid in which the subplots will appear, it is also possible to add subplots in a more dynamic, iterative fashion to a grid using `add_subplot()`. The function typically gives you a bit more flexibility.

For example, we can create four plots of which one spans the whole first "row", and three smaller plots underneath. The numbering of the subplots follows the largest axis changing fasted principle, so the subplots are placed first along the first row, then the second row, etc.

If we would have a 2×3 grid, then the numbering of the subplots would be as follows:

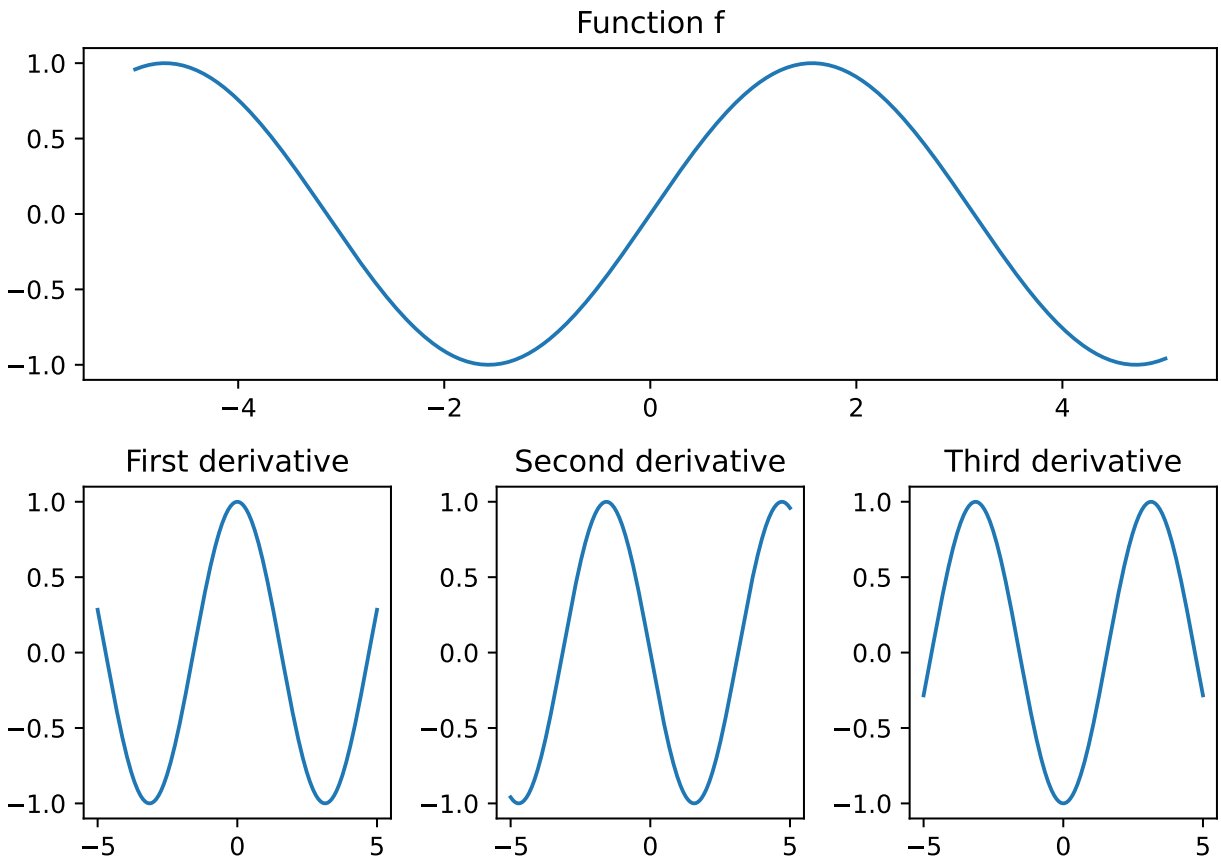
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

After having created a figure, we can add subplots to an $m \times n$ grid using `fig.add_subplot(m,n,(p,q))`. The tuple `(p,q)` indicates that we want to place the subplot on positions p through q in the figure. If there is only one position p at which you want to place the subplot, you can use `(p)`, or simply `p`, as the third argument of `add_subplot()`.

To avoid unnecessary repetition, it can often help to plot subplots using a for-loop. We will illustrate this for the figure below, in which we plot the function $f(x) = \sin(x)$ on the first row or our grid, and its first three derivatives in smaller subplots under it.

You can make the plot look nicer by adding labels, legends, different line colors, etc.

```
# Define x-range
x = np.linspace(-5,5,600)

# Function values
y = np.sin(x)

# Values of 1st, 2nd and 3rd derivative
deriv = np.vstack((np.cos(x),-np.sin(x),-np.cos(x)))

# Store function names in list
function_names = ["Function f", "First derivative",
                  "Second derivative",
                  "Third derivative"]

# Create figure
fig = plt.figure(figsize=(7,5))

# Will create an m x n grid with subplots
m, n = 2, 3

# Add first subplot
ax_f = fig.add_subplot(m,n,(1,3))
ax_f.plot(x,y)
ax_f.set_title(function_names[0])

# Add derivatives
for i in range(3):
    ax_deriv = fig.add_subplot(m,n,4+i)
    ax_deriv.plot(x,deriv[i])
    ax_deriv.set_title(function_names[1+i])

# Tighten layout
plt.tight_layout()

# Show plot
plt.show()
```



7.3 Bivariate functions

In Python it is also possible to plot a function with two variables, a so-called bivariate function. For example, consider $z = f(x, y) = x^2 + y^2$ that we would like to visualize on the (x, y)-domain $[0, 4] \times [3, 8]$.

Before going into the plotting commands we discuss in more detail how to efficiently compute the function values on the specified domain.

Just as with two-dimensional plotting, the idea is that we want compute the function values on a fine-grained discretization of the desired domain. In two dimensions, we can create such a discretization easily using the `mgrid` function from NumPy.

Suppose we discretize $[0, 4] \times [3, 8]$ by considering all the integer combinations. We can do this with `mgrid` by specifying for both dimensions the range that we are interested using index slicing.

```
# Note that the end index is not included
X, Y = np.mgrid[0:5, 3:9]
```

If you input two ranges then the output are two matrices. You could also use this function for higher dimensional problems.

```
print("X = \n", X)
```

```
X =
 [[0 0 0 0 0 0]
 [1 1 1 1 1 1]
```

```
 [2 2 2 2 2 2]
 [3 3 3 3 3 3]
 [4 4 4 4 4 4]]
```

```
  print("Y = \n", Y)
```

```
Y =
 [[3 4 5 6 7 8]
 [3 4 5 6 7 8]
 [3 4 5 6 7 8]
 [3 4 5 6 7 8]
 [3 4 5 6 7 8]]
```

The matrices X and Y together form a representation of all the integer points in the domain $[0, 4] \times [3, 8]$, namely

$$\begin{array}{cccccc} (0,3) & (0,4) & (0,5) & (0,6) & (0,7) & (0,8) \\ (1,3) & (1,4) & (1,5) & (1,6) & (1,7) & (1,8) \\ (2,3) & (2,4) & (2,5) & (2,6) & (2,7) & (2,8) \\ (3,3) & (3,4) & (3,5) & (3,6) & (3,7) & (3,8) \\ (4,3) & (4,4) & (4,5) & (4,6) & (4,7) & (4,8) \end{array}$$

To be precise, the matrix X contains the first element of every coordinate $(i, j) \in [a, b] \times [c, d]$, that is, the value i, and Y contains the second element, that is, the value j.

The same can be achieved with the function `meshgrid()` that takes as input the discretized ranges of x and y.

```
  x = np.arange(0,5)
  y = np.arange(3,9)

  X, Y = np.meshgrid(x,y)

  print("X = \n", X)
  print("Y = \n", Y)
```

```
X =
 [[0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]]
Y =
 [[3 3 3 3 3]
 [4 4 4 4 4]
 [5 5 5 5 5]
 [6 6 6 6 6]
 [7 7 7 7 7]
 [8 8 8 8 8]]
```

If we now want to compute the function values in the points (i, j), we can simply do the following

```
# Compute function values of f(x,y) = x^2 + y^2
Z = X**2 + Y**2 # For every (i,j), computes i**2 + j**2

print("Z = \n", Z)
```

```
Z =
 [[ 9 10 13 18 25]
 [16 17 20 25 32]
 [25 26 29 34 41]
 [36 37 40 45 52]
 [49 50 53 58 65]
 [64 65 68 73 80]]
```

We can also do this by defining the function f. Note that ** is a vectorized operation that is pointwise applied when executed on a two-dimensional array.

```
# Define function
def f(x,y):
    return x**2 + y**2

# Define grid
X, Y = np.mgrid[0:5, 3:9]

# f(X,Y) gives all function values of the grid points.
print("f(x,y) = \n", f(X,Y))
```

```
f(x,y) =
 [[ 9 16 25 36 49 64]
 [10 17 26 37 50 65]
 [13 20 29 40 53 68]
 [18 25 34 45 58 73]
 [25 32 41 52 65 80]]
```

We can create a more fine-grained plot by including more points in the ranges of the x- and y-values. Again, this can be done using slicing notation.

```
# Define function
def f(x,y):
    return x**2 + y**2

# Define grid with step size of 0.2 in x,y-coordinates
step = 0.2
X, Y = np.mgrid[0:2.1:step, 3:4.1:step]

# Print x-values of grid points
print("X = \n", X)

# Print y-values of grid points
print("Y = \n", Y)

# f(X,Y) gives all function values of the grid points.
print("f(X,Y) = \n", f(X,Y))
```

```
X =
```

```
 [[0.  0.  0.  0.  0.  0. ]
 [0.2 0.2 0.2 0.2 0.2 0.2]
 [0.4 0.4 0.4 0.4 0.4 0.4]
 [0.6 0.6 0.6 0.6 0.6 0.6]
 [0.8 0.8 0.8 0.8 0.8 0.8]
 [1.  1.  1.  1.  1.  1. ]
 [1.2 1.2 1.2 1.2 1.2 1.2]
 [1.4 1.4 1.4 1.4 1.4 1.4]
 [1.6 1.6 1.6 1.6 1.6 1.6]
 [1.8 1.8 1.8 1.8 1.8 1.8]
 [2.  2.  2.  2.  2.  2. ]]
Y =
 [[3.  3.2 3.4 3.6 3.8 4. ]
 [3.  3.2 3.4 3.6 3.8 4. ]
 [3.  3.2 3.4 3.6 3.8 4. ]
 [3.  3.2 3.4 3.6 3.8 4. ]
 [3.  3.2 3.4 3.6 3.8 4. ]
 [3.  3.2 3.4 3.6 3.8 4. ]
 [3.  3.2 3.4 3.6 3.8 4. ]
 [3.  3.2 3.4 3.6 3.8 4. ]
 [3.  3.2 3.4 3.6 3.8 4. ]
 [3.  3.2 3.4 3.6 3.8 4. ]
 [3.  3.2 3.4 3.6 3.8 4. ]]
f(X,Y) =
 [[ 9.   10.24 11.56 12.96 14.44 16.  ]
 [ 9.04 10.28 11.6  13.   14.48 16.04]
 [ 9.16 10.4  11.72 13.12 14.6  16.16]
 [ 9.36 10.6  11.92 13.32 14.8  16.36]
 [ 9.64 10.88 12.2  13.6  15.08 16.64]
 [10.   11.24 12.56 13.96 15.44 17.  ]
 [10.44 11.68 13.   14.4  15.88 17.44]
 [10.96 12.2  13.52 14.92 16.4  17.96]
 [11.56 12.8  14.12 15.52 17.   18.56]
 [12.24 13.48 14.8  16.2  17.68 19.24]
 [13.   14.24 15.56 16.96 18.44 20.  ]]
```

7.3.1 Contour plot

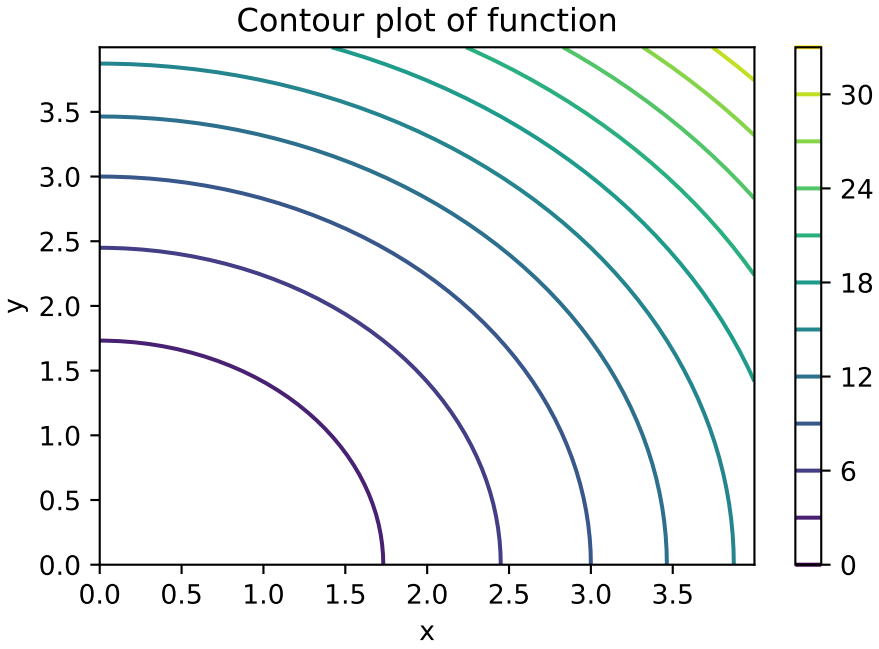
One way to visualize a function $f : \mathbb{R}^2 \to \mathbb{R}$ is using a contour plot with `plt.contour()`. What such a plot does is that it creates a two-dimensional plot where for any given value $z_0 \in \mathbb{R}$ it plots all points $(x, y) \in \mathbb{R}^2$ for which $f(x, y) = z_0$ with the same color.

The function `plt.contour()` takes as input the arrays X, Y and Z (the order is important here). Python plots the point $(X[i, i], Y[i, j])$ and assigns a common color to all such points with the same function value (i.e., the same $Z[i, j]$-value).

The `levels` keyword argument determines how many different colors, i.e., z_0-values, are plotted. For this Python uses a so-called colormap that has a shifting scale indicating a shift in the value of z_0. You can plot a color legend with `plt.colorbar()`.

```
# Define function
def f(x,y):
    return x**2 + y**2
```

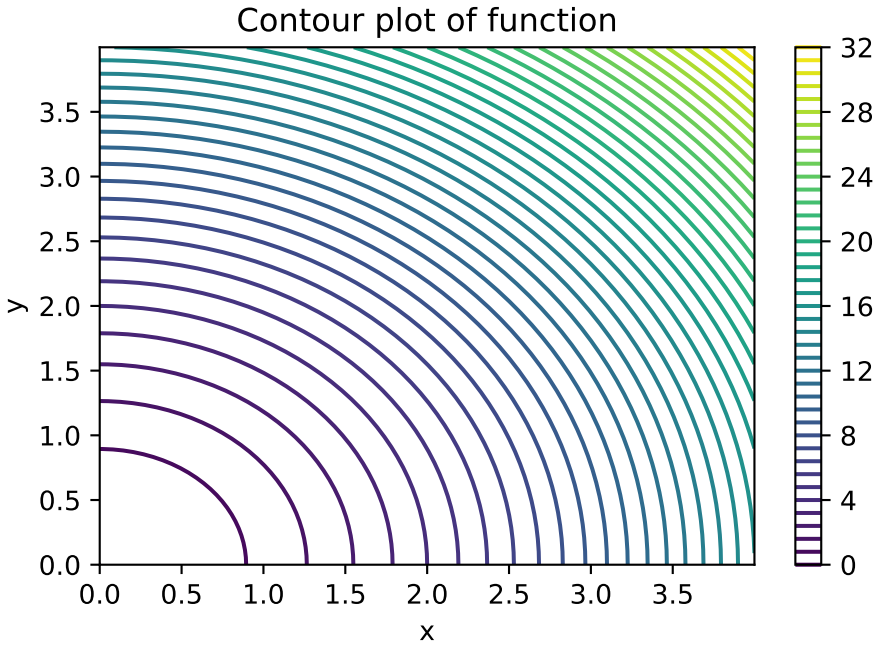
```
# Grid parameters
b = 4
step = 0.001

# Define grid [0,b]^2 with given step size
X, Y = np.mgrid[0:b:step, 0:b:step]

# Create figure
plt.figure()

# Create contour plot
plt.contour(X, Y, f(X,Y), levels=10)

# Add labels and title
plt.xlabel("x")
plt.ylabel("y")
plt.title("Contour plot of function")

# Show the plot
plt.colorbar()  # Add a color bar for reference

# Show plot
plt.show()
```



The same plot as above with 100 color levels is given below.

```
# Define function
def f(x,y):
    return x**2 + y**2
```

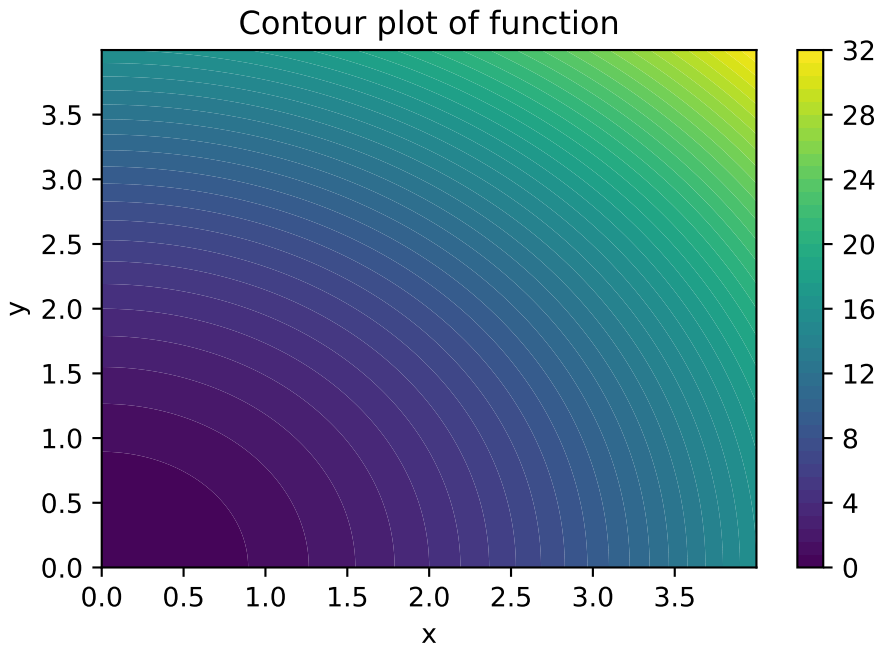
```
# Grid parameters
b = 4
step = 0.001

# Define grid [0,b]^2 with given step size
X, Y = np.mgrid[0:b:step, 0:b:step]

# Create figure
plt.figure()

# Create contour plot
plt.contour(X, Y, f(X,Y), levels=50)

# Add labels and title
plt.xlabel("x")
plt.ylabel("y")
plt.title("Contour plot of function")

# Show the plot
plt.colorbar()  # Add a color bar for reference

# Show plot
plt.show()
```



You can also choose to fill up the white space between the different contour lines. Then you should use `plt.contourf()` instead of `plt.contour()`. The same plot with 50 color levels and `plt.contourf()` is given below.

```
# Define function
def f(x,y):
```

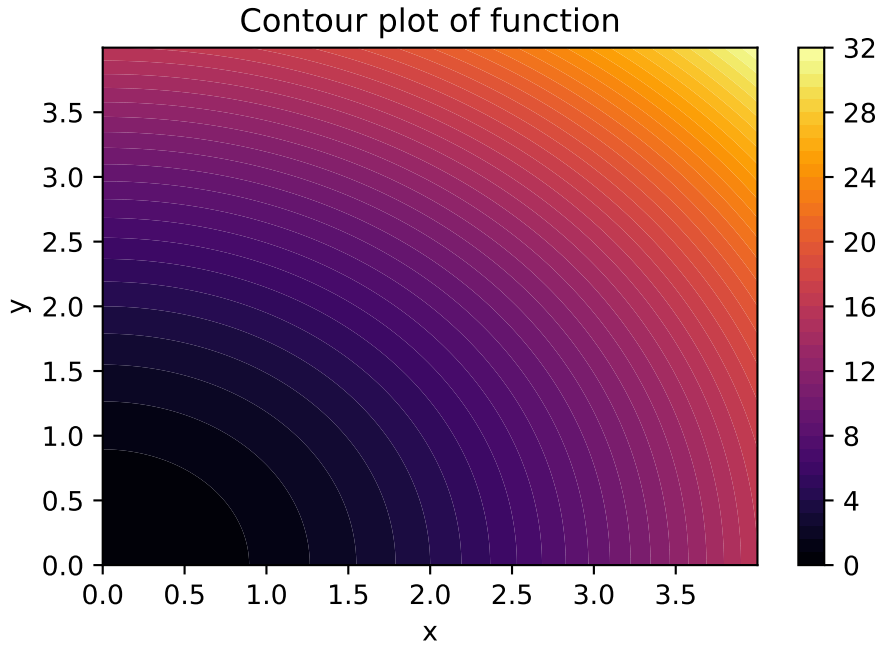
```
    return x**2 + y**2

# Grid parameters
b = 4
step = 0.001

# Define grid [0,b]^2 with given step size
X, Y = np.mgrid[0:b:step, 0:b:step]

# Create figure
plt.figure()

# Create contour plot with 50 levels
plt.contourf(X, Y, f(X,Y), levels=50)

# Add labels and title
plt.xlabel("x")
plt.ylabel("y")
plt.title("Contour plot of function")

# Show the plot
plt.colorbar()  # Add a color bar for reference

# Show plot
plt.show()
```



Finally, you can change the color chosen by Python using the `cmap` keyword argument. There are various color maps available; see the documentation.

Below we have plotted the figure above with the inferno colormap.

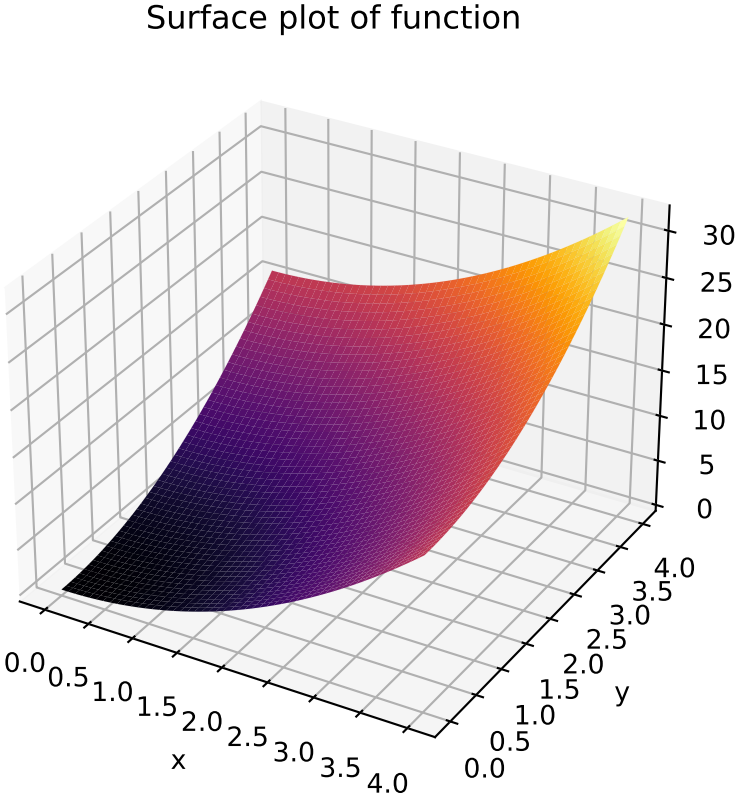
```
# Define function
def f(x,y):
    return x**2 + y**2

# Grid parameters
b = 4
step = 0.001

# Define grid [0,b]^2 with given step size
X, Y = np.mgrid[0:b:step, 0:b:step]

# Create figure
plt.figure()

# Create contour plot with 50 levels
plt.contourf(X, Y, f(X,Y), levels=50, cmap="inferno")

# Add labels and title
plt.xlabel("x")
plt.ylabel("y")
plt.title("Contour plot of function")

# Show the plot
plt.colorbar()  # Add a color bar for reference

# Show plot
plt.show()
```



7.3.2 3D plot

Another way of visualizing a bivariate function is using a 3D plot, in which we have an $x-$, $y-$ and $z-$axis. This you can do, e.g., with the `plot_surface()` function. To use this function we have to explicitly create an Axes object (plot), which we call `ax`, with three axes using `ax = plt.axes(projection='3d')`.

```
# Define function
def f(x,y):
    return x**2 + y**2

# Grid parameters
b = 4
step = 0.001

# Define grid [0,b]^2 with given step size
X, Y = np.mgrid[0:b:step, 0:b:step]

# Create figure
fig = plt.figure(figsize=(7,5))

# Create Axes object with three axes
ax = plt.axes(projection='3d')

# Create surface plot
ax.plot_surface(X, Y, f(X,Y), cmap="inferno")

# Add labels and title
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("Surface plot of function")

# Show plot
plt.show()
```



We close this chapter by remarking that we have only shown a small fraction of the plotting functionality that Python has to offer. There are many more options to create nice looking plots.

Appendix A

Corrections and changes

This chapter contains a list of corrections and chnages that were made during the teaching of the Python II module. If you spot a spelling error or mistake, feel free to let the teacher know! This will improve the quality of the book in the future significantly.

A.1 Section 3.2.2

- `mgrid()` explanation is moved to Chapter 5.

A.2 Section 4.3.1

- Corrected the explanation of `np.argsort()`.

A.3 Section 3.5

- Corrected `M_repeat = np.tile(M,3)` by `M_repeat = np.repeat(M,3)`
- Corrected `vstack(a,b)` to `vstack((a,b))`.
- Corrected `hstack(a,b)` to `hstack((a,b))`.

Appendix B

Function basics

In this chapter we will explain some basic commands for Python functions defined using `def`.

B.1 Output arguments

Suppose we have a function that takes two inputs, and yields four outputs.

```
def f(a,b):
    return a + b, a - b, a*b, a/b

a = 5
b = 3
```

There are various ways to get one or more specific outputs from Python.

```
# Returns all output variables in tuple
output = f(a,b)

print(output)
```

```
(8, 2, 15, 1.6666666666666667)
```

```
# Store outputs in variables w, x, y and z
w, x, y, z = f(a,b)

print(w, x, y, z)
```

```
8 2 15 1.6666666666666667
```

We can suppress one or more output arguments using `_`.

```
# Only store first and last output
w, _, _, z = f(a,b)

print(w, z)
```

```
8 1.6666666666666667
```

If we only want the first output, and suppress the other ones, we can use `*_`

```
# Only store first output
x, *_ = f(a,b) # This is the same as x, _, _, _ = f(a,b)

print(x)
```

```
8
```

If we only want the first two outputs, and suppress the remaining ones, we can do something similar.

```
# Only store first output
w, x, *_ = f(a,b) # This is the same as w, x, _, _ = f(a,b)

print(w,x)
```

```
8 2
```

If we only want the last output, we can do the following.

```
# Only store last output
*_, z = f(a,b) # This is the same as _, _, _, z = f(a,b)

print(z)
```

```
1.6666666666666667
```

Also here, if we would like the last two outputs, we can do the following.

```
# Only store last output
*_, y, z = f(a,b) # This is the same as _, _, y, z = f(a,b)

print(y,z)
```

```
15 1.6666666666666667
```