Table of Contents

# Python Introduction

Python is a general-purpose high-level programming language. It was invented in the late ’80s, and its current major version is 3. Important features and characteristics of Python include:

* Object-oriented
* Dynamically typed
* Open source
* High-level
* Interpreted
* First-class functions
* Higher-order functions

Python code is interpreted, meaning it is transformed into byte code one line at a time during execution. There is no separate compile phase, so no compile time is needed. Popular uses of Python include scripting and automation, microservices, statistics, and machine learning.

Due to its open-source nature and well-developed libraries for computation and visualization, such as NumPy and Matplotlib, Python has become quite popular in math-related fields, especially in universities and academia. It has increasingly replaced previously well-established software like “Wolfram Mathematica” and “MATLAB.”

## Basic Python commands

Traditionally, every tutorial for a programming language starts with an introduction on how to display output to the screen or use the print command. Printing is done with the function print.

**Example**: Print “Hello, World!”

# Print a string  
print("Hello World")

Hello World

While printing, strings and numbers can be concatenated using the , sign.

**Example**. Concatenate the words “Hello” and “World” and print them.

# Concatenating and printing strings  
print("Concatenating Hello and World with +: "+"Hello"+" World")  
print("Concatenating Hello and World with +: ","Hello"," World")

Concatenating Hello and World with +: Hello World  
Concatenating Hello and World with +: Hello World

## Declaring variables in Python

Python is a dynamic language, and memory is assigned at execution time. This means that variables don’t need to be associated with a type at the time of declaration; they receive their type at the time of initialization. As a result, only initialization is required. Additionally, because Python is dynamic, different types of data can be assigned to the same variable, although this is generally not considered good practice.

In Python, the symbol : can be used to provide hints on the variable type. These hints are used by development environments to provide information to the developer. Using type hints improves code quality and makes the coding process easier.

**Example**. Assign 1 and 2 to integer variables x, y and print their values

x = 1  
# int is a hint and not required in the following expression  
y:int = 2  
  
print("x is:")  
print(x)  
print("y is:")  
print(y)

x is:  
1  
y is:  
2

Almost anything can be printed using concatenation, but Python offers a more elegant way of formatting. Simple formatting is done by using the letter f before the string. Instead of concatenating, variables can be inserted directly into the string using {}.

**Example**: Using string formatting to print x=1, y=2 where x and y are integer variables.

x = 1  
# int is a hint and not required in the following expression  
y:int = 2  
  
print(f"x={x}, y={y}")

x=1, y=2

Similarly, a string can be formatted into two lines using "\n" as a line break.

**Example**. Print one string in two lines using the "\n" line brake.

print(f"To print in a new line use \\n \\n The following text will be printed in a new line:\\n'text in a new line'")

To print in a new line use \n \n The following text will be printed in a new line:\n'text in a new line'

## Basic data types in Python

* Text Type: str
* Numeric Types: int, float, complex
* Sequence Types: list, tuple, range
* Mapping Type: dict
* Set Types: set, frozenset
* Boolean Type: bool
* Binary Types: bytes, bytearray, memoryview
* None Type: NoneType

**Example**. Declare the most common data types in Python.

# Strings, datatypes that hold text.  
x1: str = "Hello World"  
  
# Integers, positive and negative whole numbers.  
x2: int = 20  
  
# Floats, datatypes that are generally used to represent real numbers.  
# Mathematically speaking, they are real numbers with a precision of usually 16-17 digits.  
x3: float = 20.5  
  
# Complex numbers, used to represent numbers with a real and an imaginary part.  
# They have precision similar to floats.  
x4: complex = 1j  
  
# List or array, collections of indexed elements. The elements don't have to be unique.  
x5: list = ["apple", "banana", "cherry"]  
# The indexing starts from 0. e.g., access the second element of a list  
x5\_2 = x5[1]  
  
# Tuple, collections of indexed elements similar to lists but are immutable once  
# created. This type of variable is not used in this book. It is used to  
# represent static data that is stored in memory.  
x6: tuple = ("apple", "banana", "cherry")  
  
# Range, immutable sequences of numbers, usually used for looping or iterating  
# in arrays or other data structures.  
x7: range = range(6)  
  
# Dictionary, key-value pair data structure where keys are unique. Usually used  
# when building JSON objects or storing data. This variable is not used in this  
# book.  
x8: dict = {"name": "John", "age": 36}  
  
# Access an element in the dictionary  
x8\_name = x8["name"]  
  
# Set, unordered collections of unique elements. Not used in this book. Usually  
# used to store or keep track of unique values.  
x9: set = {"apple", "banana", "cherry"}  
  
# Frozenset, unordered collections of unique elements. Unlike sets, they are  
# immutable. Not used in this book. Could be used to store certain static  
# parameters inside a client application or server from initialization until  
# the end of the application.  
x10: frozenset = frozenset({"apple", "banana", "cherry"})  
  
# Boolean, represents True or False values. Mostly used for conditional statements.  
x11: bool = True  
  
# Bytes, sequences of bytes. Once a bytes object is created, its elements cannot  
# be changed. Used when dealing with communication with embedded devices like  
# electronic cards. This variable is not of interest for the purposes/range of  
# this book.  
x12: bytes = b"Hello"  
  
# Bytearray, similar to bytes but mutable, meaning elements can be changed.  
# This variable is not of interest for the purposes/range of this book.  
x13: bytearray = bytearray(5)  
  
# None, used to represent the absence of a value.  
x14: None = None

***When writing high-quality code, it is essential to select variable types that satisfy the minimal requirements for the use case. This seminar work has an educational purpose and balances efficient code practices with clarity. For this reason, not necessarily the most effective but the most common and recognizable data types are used.***

## Basic arithmetic operations in Python

Python provides several built-in operators for basic arithmetic operations. These are the most common ones: \* Addition (+): Adds two numbers together. \* Subtraction (-): Subtracts one number from another. \* Multiplication (*): Multiplies two numbers.*  Division (/): Divides one number by another (returns a float). \* Integer Division (//): Divides one number by another and returns the integer part of the result. \* Modulo (%), returns the remainder when one number is divided by another \* Exponentiation (\*\*): Raises one number to the power of another.

**Example**. Demonstrate the usage the most common built-in operators in Python.

# Addition  
result = 5 + 3  
print("Addition:", result)  
# Subtraction  
result = 5 - 3  
print("Subtraction:", result)  
# Multiplication  
result = 5 \* 3  
print("Multiplication:", result)  
# Division  
result = 5 / 3  
print("Division:", result)  
# Integer Division  
result = 5 // 3  
print("Integer Division:", result)  
# Modulo  
result = 5 % 3  
print("Modulo:", result)  
# Exponentiation  
result = 2 \*\* 3  
print("Exponentiation:", result)

Addition: 8  
Subtraction: 2  
Multiplication: 15  
Division: 1.6666666666666667  
Integer Division: 1  
Modulo: 2  
Exponentiation: 8

## If - else conditions in Python

The general form of an if-else condition in Python is:

if <bool>:  
 <code block>  
elif <bool>:  
 <code block>  
 .....  
else:  
 <code block>

As shown, python uses the symbol : followed by new line with indentation to indicate a start of a new block.

**Example**. Given a number i, print one of the following “i is smaller or equal to 10”, “i is smaller than 20”, “i is equal to 20”, “i is bigger than 20” depending on i.

i\_:int = 12  
  
if i\_<=10:  
 print("i is smaller or equal to 10")  
elif i\_<20:  
 print("i is smaller than 20")  
elif i==20:  
 print("i is equal to 20")  
else:  
 print("i is bigger or equal than 20")

i is smaller than 20

## Loops and iterations in Python

As in other languages, commonly used loops in Python are while and for loops. Additionally, Python supports functional programming, and it is not uncommon for functions to be used instead of declarative loops for interactions, especially when working with data collections.

The while loop syntax form in python is the following:

while <boolean>:  
 <code block>

**Example**. Using while loop print the elements of a list, l = {1,2,3,"this is the last element of the list"}

l = [1,2,3,"this is the last element of the list"]  
  
index\_l = 0  
while(index\_l<4):  
 print(l[index\_l])  
 index\_l = index\_l + 1

1  
2  
3  
this is the last element of the list

The general form of a for loop in Python is:

for variable in sequence:  
 <code block>

**Example**. Using for loop, print the elements of the list l = {1,2,3,"this is the last element of the list"}

l = [1,2,3,"this is the last element of the list"]  
  
for element in l:  
 print(element)

1  
2  
3  
this is the last element of the list

Python supports functional programming, so it is becoming increasingly popular to use functions instead of loops for iterating over data structures. Built-in functions like map and reduce are used to apply a function to the elements of a collection, such as a list or set. This style of writing code is called declarative. In contrast, imperative programming uses loops like while or for to iterate over data structures. More detailed examples will be shown in the functions section of this seminar work.

**Example**. Using the map function print the elements of a list l = [1,2,3,"this is the last element of the list"] in Python

l = [1, 2, 3, "this is the last element of the list"]  
  
list(map(lambda x: print(x), l))

1  
2  
3  
this is the last element of the list  
  
  
  
  
  
[None, None, None, None]

## Functions in python

Depending on how they are defined, there are generally two types of functions in Python: lambda functions and standard functions. Standard functions, or just functions, are traditional and quite flexible, but they require more code to be defined. Lambda functions, on the other hand, require less code but are not as flexible, as they are limited to single-line expressions.

This seminar work will not use lambda functions, as they can be completely replaced by standard functions. In general, lambda functions are used with predefined functions like map or reduce when iterating over data structures. They are often inlined with these functions, which also contributes to code readability.

Standard function

def function\_name(parameters):  
 <Code block>  
 <Command 1>  
 <Command 2>  
 ...  
 </Code block>  
 <!-- optional return -->  
 return <value>

Lambda function

lambda parameters: expression

**Example**. Using a standard and a lambda function calculate the sum of two numbers and print the output.

def sum\_standard\_function(x, y):  
 result = x + y  
 return result  
# Call the standard function and print the output  
sum\_standard = sum\_standard\_function(3, 5)  
print(f"Using Standard Function: {sum\_standard}")  
  
sum\_lambda\_function = lambda x, y: x + y  
# Call the lambda function and print the output  
  
sum\_lambda = sum\_lambda\_function(3, 5)  
print(f"Using Lambda Function: {sum\_lambda}")

Using Standard Function: 8  
Using Lambda Function: 8

**Functions are “first-class citizens” in Python**, meaning they are treated as variables. A function can be assigned to a variable, passed to another function as a parameter, or returned from a function. Treating functions as variables is part of the [functional programming paradigm](https://en.wikipedia.org/wiki/Functional_programming). These features are supported out of the box in Python, and as a result, higher-order functions can be created quite easily. In the following sections related to optimization, these Python features and other elements of the functional programming paradigm will be used extensively.

**Example**: Create a function that numerically calculates the derivative of a function in ,

,

meaning instead of calculating the limit when approaching 0, use an extremely small interval around to approximate the derivative, where is very small ( e.g. ).

def f(x\_in:float):  
 return x\_in\*\*2  
  
def derivative\_of(function\_in):  
 def numerical\_derivative(x\_in):  
 \_delta\_x = 1e-5  
 \_derivate\_f = (function\_in(x\_in + \_delta\_x) - function\_in(x\_in - \_delta\_x))/(2\*\_delta\_x)  
 return \_derivate\_f  
 return numerical\_derivative  
  
derivative\_f = derivative\_of(f)  
  
print(f"The approximation of the derivate of a function x^2 in x = 3 is {derivative\_f(3)}")

The approximation of the derivate of a function x^2 in x = 3 is 6.000000000039306

### Libraries and modules

Basic arithmetic operations with functions in Python include finding the absolute value using abs and calculating the square root using math.sqrt. The abs function is part of Python’s standard library, while sqrt is part of the math library, which is commonly used but not included in the base library. Libraries are collections of modules, and modules are collections of functions and classes that are used for similar purposes. The math library needs to be imported before it is used. To do so, write import math before using it. Once imported, the functions and classes inside the library can be used by accessing them with dot notation. For example, math.sqrt(9) will return 3 as it calculates the square root of nine.

Modules, on the other hand, are files containing Python code within a library. For example, when using import matplotlib.pyplot, pyplot is a module in the matplotlib library that is used for plotting data. Modules and libraries can be imported with aliases instead of their full names. To define an alias for the module matplotlib.pyplot, the following code line can be used: import matplotlib.pyplot as plt.

*As with every programming language, the necessary libraries need to be imported and be part of the dependencies. The advantage of using Google Colab is that it handles the dependencies itself, so only an import statement for the required library is necessary.*

**Example**: Given a number, calculate the square root of the absolute value of that number.

import math  
  
def calculate\_sqrt\_abs(num\_in):  
 abs\_num = abs(num\_in)  
 sqrt\_abs\_num = math.sqrt(abs\_num)  
 return sqrt\_abs\_num  
  
number = -9  
result = calculate\_sqrt\_abs(number)  
print(f"The square root of the absolute value of {number} is {result}")

The square root of the absolute value of -9 is 3.0

## Logical Operators

Python uses True and False as boolean values.

The most popular logical operators are: \* and \* or \* not

Comparing values is done with: \* <, <= \* >, >= \* ==

**Examples:**

1. Given two numbers a and b, write a function that will return the larger number.
2. Given three numbers a, b, and c, write a function that will return the largest number.
3. Given two numbers a and b, write a function that returns True if a > b.
4. Write a function that, given a number n, prints the first n numbers.

# Example 1  
def max\_of\_two(a, b):  
 return a if a > b else b  
print("Max of 10 and 20:", max\_of\_two(10, 20)) # Output: 20  
  
# Example 2  
def max\_of\_three(a, b, c):  
 pom = a  
 if(b>a):  
 pom = b  
 if(c>pom):  
 pom = c  
 return pom  
print("Max of 50, 40, and 30:", max\_of\_three(50, 40, 30)) # Output: 50  
  
# Example 3  
def is\_a\_greater\_than\_b(a, b):  
 return a > b  
# Example 4  
def print\_first\_n(n:int):  
 print(f"First {n} positive integers:")  
 i=1  
 while(not is\_a\_greater\_than\_b(i,n)):  
 print(i)  
 i = i+1  
  
print\_first\_n(10)

Max of 10 and 20: 20  
Max of 50, 40, and 30: 50  
First 10 positive integers:  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10

## Plotting Functions

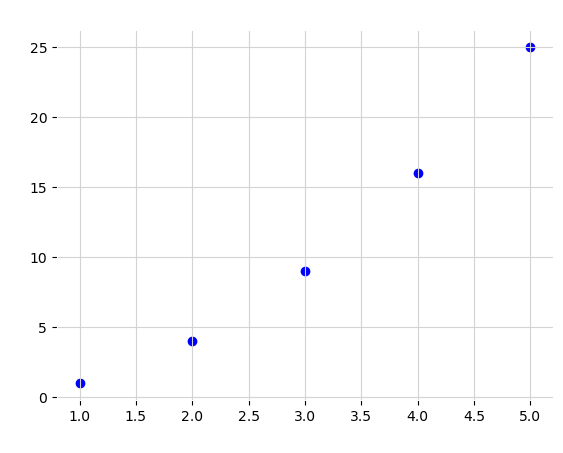
One of the reasons why Python is popular in mathematical-related areas is because of the Matplotlib library. This library is used to draw or plot on a coordinate system. It can be used for drawing functions, dots, bars, histograms, and pie charts. It offers other useful options like drawing in different colors, naming the axes of the coordinate system, and naming the entire plot. It also supports drawing in both 2D and 3D.

**Examples:**

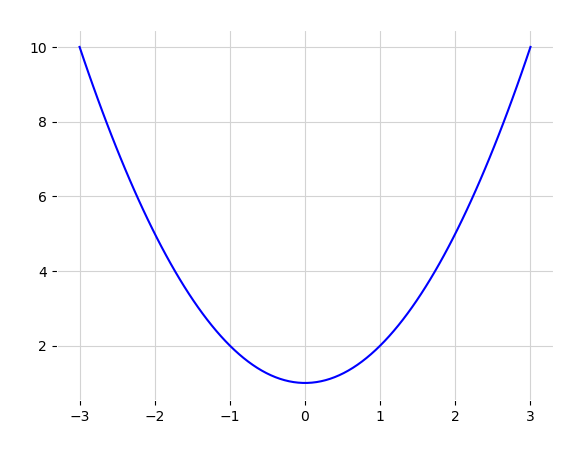
1. Draw dots in a coordinate system.
2. Draw a function in a coordinate system over a given domain.
3. Draw a line in a coordinate system that passes through two dots.

# Optional, Customizing Matplotlib styles  
# This step is not necessary and can be skipped, it makes sure that regardless of the system the plots will look the same.  
import matplotlib.pyplot as plt  
plt.rcParams['axes.prop\_cycle'] = plt.cycler('color', ['blue', 'green', 'red', 'cyan', 'magenta', 'yellow', 'black'])  
plt.rcParams['axes.facecolor'] = 'white'  
plt.rcParams['figure.facecolor'] = 'white'  
plt.rcParams['grid.color'] = 'lightgray'  
plt.rcParams['xtick.color'] = 'black'  
plt.rcParams['ytick.color'] = 'black'

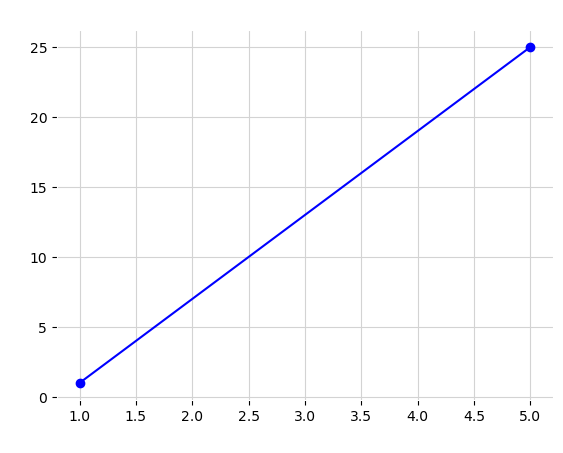
import matplotlib.pyplot as plt  
import numpy as np  
  
# Example 1  
# Draw dots  
  
# Define coordinates for the dots  
x\_dots = [1, 2, 3, 4, 5]  
y\_dots = [1, 4, 9, 16, 25]  
  
# Plot the dots  
plt.scatter(x\_dots, y\_dots)  
plt.xlabel('X')  
plt.ylabel('Y')  
plt.title('Dots in a Coordinate System')  
plt.grid()  
plt.show()  
  
# Example 2  
# Draw function  
  
# Define the function  
def f(x):  
 return x\*\*2 + 1  
  
# Define the domain  
x\_func = np.linspace(-3, 3, 100) # Adjust the domain as needed  
  
# Plot the function  
plt.plot(x\_func, f(x\_func))  
plt.xlabel('X')  
plt.ylabel('Y')  
plt.title('Function in a Coordinate System')  
plt.grid()  
plt.show()  
  
# Example 3  
# Draw line  
  
# Define the coordinates of the two dots  
x\_line = [1, 5]  
y\_line = [1, 25]  
  
# Plot the line passing through the two dots  
plt.plot(x\_line, y\_line, marker='o')  
plt.xlabel('X')  
plt.ylabel('Y')  
plt.title('Line Passing Through Two Dots')  
plt.grid()  
plt.show()



png



png



png

## Basic Mathematical Functions

**Absolute Value**

The absolute value using NumPy can be calculated with np.abs(<np.array>) or abs(x). The np.abs function is used on a NumPy array and returns an array where all elements are positive.

**Polynomial Functions**

A one-dimensional polynomial function is defined as:

Using the NumPy library, a polynomial function can be created with np.poly1d, where the are elements of a np.array.

**Derivatives**

Accordingly, the derivative of a polynomial function is:

and the derivative of a polynomial function f can be found using np.polyder(f) where f=np.poly1d(<np.array>).

Except for calculating derivatives of polynomial functions, NumPy does not offer a function to calculate the derivative of any function.

This leaves us two popular options for calculating a derivative of a general function:

### Custom function

Write a function to calculate the derivative according to the definition of a derivative. The function derivative\_of does exactly this and can differentiate any one-dimensional Python function.

**Advantages:** The function derivative\_of returns a function and as so supports higher-order derivatives. Additional advantage of this method is that no additional library.

**Disadvantages:** The function derivative\_of supports only one dimensional functions.

*derivative\_of can be only used with used with mathematical function, take a number as an input and return a number as an output*

### Sympy-library

The sympy library’s sympy.diff function with sympy.symbols is the most general and flexible solution for symbolic differentiation. It allows the calculation of the derivative of a function and can be used to calculate higher-order derivatives (second, third, etc.). It can also be used to calculate partial derivatives of a multi-dimensional function.

**Advantages:** It supports higher-order and partial derivatives.

**Disadvantages:** It can’t operate on standard Python functions but requires a special variable type (symbolic objects). Instead of creating a standard Python function for the mathematical function whose derivative is required, a symbolic object representing the mathematical function is created. The operations of differentiation are then performed on this object.

My suggestion is to use the derivative\_of function whenever possible and to use sympy.diff for multi-dimensional functions. Note that sympy.diff is not used in this seminar work.

**Integrals**

NumPy also does not offer functions for calculating bounded integrals or integrals in general; they need to be calculated manually. The scipy.integrate library can be used for approximation. Use scipy.integrate.nquad for single-value functions and for functions with more than one variable, and use scipy.integrate.trapz for functions with more than one dimension. Note that Integrals are not required or used in this seminar work.

*In general, multi-purpose programming languages do not offer methods or libraries that can manipulate mathematical functions. They usually offer a way to approximate them, perhaps because mathematical functions are not a standard type of variable in general-purpose languages.*

**Examples:**

1. Calculate the first derivative of the polynomial function and evaluate it in .
2. Calculate the first and second derivatives of the polynomial function and evaluate them in using the derivative\_of function.
3. Calculate the first and second derivatives of the function and evaluate them in using the sp.diff function.
4. Calculate the partial derivatives of with respect to x and y using the sp.diff function.

# Example 1  
import numpy as np  
  
def f(x):  
 return 2 \* x\*\*2 + 3 \* x - 5  
x = 3  
  
# Example 1  
# the polynomial function is represented as a Python list  
f\_array = np.poly1d([2, 3, -5])  
# df is first derivative of f  
df = f\_array.deriv()  
# First derivative of f(x) is 4 \* x + 3 - 5  
print(f"The first derivative of f(x) = 2x^2 + 3x - 5 in x = 3 is: {df(x)}")  
  
# Example 2  
# df is first derivative of f  
df = derivative\_of(f)  
# ddf is the second derivative of f  
ddf = derivative\_of(df)  
# First derivative of f(x) is 4 \* x + 3 - 5  
print(f"The first derivative of f(x) = 2x^2 + 3x - 5 in x = 3 is: {df(x)}")  
# Second derivative of f(x) is 4  
print(f"The second derivative of f(x) = 2x^2 + 3x - 5 in x = 3 is: {ddf(x)}")  
  
# Example 3  
import sympy as sp  
x = sp.symbols('x')  
f = 2 \* x\*\*2 + 3 \* x - 5  
# df is first derivative of f  
df = sp.diff(f, x)  
print(f"The first derivative of {f} is {df}")  
print(f"The first derivative of {f} in x = 3 is", df.evalf(subs={x: 3}))  
# ddf is the second derivative of f  
ddf = sp.diff(df, x)  
print(f"The second derivative of {f} is {ddf}")  
print(f"The first derivative of {f} in x = 3 is {ddf.evalf(subs={x: 3})}")  
  
# Example 4  
import sympy as sp  
x = sp.symbols('x')  
y = sp.symbols('y')  
f = 2 \* x\*\*2 + 3 \* y\*\*2 - 5  
# df is first derivative of f  
dxf = sp.diff(f, x)  
print(f"The partial derivative of {f} with respect {x} is {dxf}")  
# ddf is the second derivative of f  
dyf = sp.diff(f, y)  
print(f"The partial derivative of {f} with respect {y} is {dyf}")

The first derivative of f(x) = 2x^2 + 3x - 5 in x = 3 is: 15  
The first derivative of f(x) = 2x^2 + 3x - 5 in x = 3 is: 15.000000000142675  
The second derivative of f(x) = 2x^2 + 3x - 5 in x = 3 is: 4.000000330961484  
The first derivative of 2\*x\*\*2 + 3\*x - 5 is 4\*x + 3  
The first derivative of 2\*x\*\*2 + 3\*x - 5 in x = 3 is 15.0000000000000  
The second derivative of 2\*x\*\*2 + 3\*x - 5 is 4  
The first derivative of 2\*x\*\*2 + 3\*x - 5 in x = 3 is 4.00000000000000  
The partial derivative of 2\*x\*\*2 + 3\*y\*\*2 - 5 with respect x is 4\*x  
The partial derivative of 2\*x\*\*2 + 3\*y\*\*2 - 5 with respect y is 6\*y

# Arrays, Matrices, and Elements of Linear Algebra in Python

Python has a rich set of functions and libraries for operations with vectors and matrices. This is generally done using the libraries NumPy and SymPy. Matrices offer a powerful way to handle linear algebra operations. This text in the chapter on optimization algorithms will extensively use linear algebra and the NumPy library.

## Standard operations with lists in Python

Although not connected directly to linear algebra, standard lists in python will be used in developing the optimization algorithms in the next chapter of this text.

Most popular operations with the standard python list are:

* **Indexing**: list[i], where i is the index.
* **Appending**: list.append(x), where x is appended at the end of the list.
* **Removing**: list.pop(i), where i is the index of the element to be deleted.
* **Inserting**: list.insert(i, x), where x is the element to be inserted and i is it’s position index.
* **Slicing**: list[x:y], where x and y are the indices of the first and last elements of the returned list (inclusive of x and exclusive of y). x or y might be omitted, which would be interpreted as the start and end of the list, respectively.
* **Length of a list**: len(list) - getting the size of the list.

*When it comes to indexing, the first element of the list has index 0 and the last element can be accessed with -1.*

**Examples:**

1. Define list\_a with 10 elements and print its first, second, and last elements.
2. Print the size of list\_a.
3. Append an element to the already defined list\_a and print the list.
4. Remove the second element of list\_a and print the list.
5. Insert an element in list\_a between the second and third elements and print the list.
6. From list\_a, use the slicing operator to create a new list, list\_b, that contains the elements from list\_a starting from index 3 and ending at index 5, including the elements at index 3 and 5.
7. From list\_a, use the slicing operator to create a list, list\_b, that does not contain the first 3 elements of list\_a.
8. From list\_a, use the slicing operator to create a list, list\_b, that does not contain the first 4 elements of list\_a, in the same order as in list\_a.

# Example 1  
# Define list\_a with 10 elements  
list\_a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
# Print the first, second, and last elements  
print("First element:", list\_a[0])  
print("Second element:", list\_a[1])  
print("Last element:", list\_a[-1])  
  
# Example 2  
# Print the list\_a size  
print("Size of list\_a:", len(list\_a))  
  
# Example 3  
# Append an element to list\_a and print the list  
list\_a.append(11)  
print("List after appending an element:", list\_a)  
  
# Example 4  
# Remove the second element of list\_a and print the list  
list\_a.pop(1)  
print("List after removing the second element:", list\_a)  
  
# Example 5  
# Insert an element in list\_a between the second and third elements and print the list  
list\_a.insert(2, 15)  
print("List after inserting an element:", list\_a)  
  
# Example 6  
# Create a new list list\_b from list\_a using the slicing operator from index 3 to 5  
list\_b = list\_a[3:6]  
print("List\_b from index 3 to 5 of list\_a:", list\_b)  
  
# Example 7  
# Create a new list list\_b from list\_a excluding the first 3 elements  
list\_b = list\_a[3:]  
print("List\_b excluding the first 3 elements of list\_a:", list\_b)  
  
# Example 8  
# Create a new list list\_b from list\_a excluding the first 4 elements  
list\_b = list\_a[4:]  
print("List\_b excluding the first 4 elements of list\_a:", list\_b)

First element: 1  
Second element: 2  
Last element: 10  
Size of list\_a: 10  
List after appending an element: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]  
List after removing the second element: [1, 3, 4, 5, 6, 7, 8, 9, 10, 11]  
List after inserting an element: [1, 3, 15, 4, 5, 6, 7, 8, 9, 10, 11]  
List\_b from index 3 to 5 of list\_a: [4, 5, 6]  
List\_b excluding the first 3 elements of list\_a: [4, 5, 6, 7, 8, 9, 10, 11]  
List\_b excluding the first 4 elements of list\_a: [5, 6, 7, 8, 9, 10, 11]

## Numpy library, standard operations

Creating a vector in Python can be done with the numpy library. Vectors created using the numpy library are usually referred to as numpy arrays or np arrays, and they are created from a list. Before using the numpy library, it needs to be imported. The standard way to import it is import numpy as np. This line imports numpy as np, and from now on in this text, numpy will be referenced and called np, as this is standard slang in Python.

import numpy as np  
my\_list = [1,2,3,4,5]  
my\_vector = np.array(my\_list)

The numpy library does not have any special data structure for a matrix, but matrixes are created as an array of arrays

my\_matrix = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])

It is important to note that all the previously described functions and operators that are applicable to the standard Python list are also applicable to np arrays. Other useful functions that are part of the np library but not directly connected to linear algebra are:

* np.full: Returns an array with a specified size and shape, prefilled with a provided element.
* np.linspace: Generates evenly spaced numbers over a specified interval, useful for plotting functions.
* np.concatenate: Joins together vectors and matrices, useful for solving linear systems or formalizing linear programs.
* np.vstack: Stacks arrays in sequence vertically (row-wise), useful for concatenating matrices.
* np.hstack: Stacks arrays in sequence horizontally (column-wise), useful for concatenating matrices.
* .shape: Returns the number of rows and columns of a matrix.

*In the context of NumPy, a matrix A with m rows and ncolumns is said to have* ***shape*** *[m, n].*

**Examples:**

1. Create an array with size 10, filled with the number 5.
2. Add additional 5 zeros to the end of the array.
3. Generate 10 values for the function on a domain between [1,10].
4. Create a matrix A1 with shape [2, 3].
5. Create a matrix A2 with shape [2, 2] and concatenate it with matrix A1 so that the new matrix A3 = [A1 A2] is created with a shape [2, 5].
6. Create a new matrix A4 with a shape [3, 5] and concatenate it with the matrix A3 so that A5 = [A3; A4] is created with a shape [5, 5].

import numpy as np  
  
# Example 1  
# Create an array with size 10, filled with the number 5  
array = np.full(10, 5)  
print("Array filled with 5s:", array)  
  
# Example 2  
# Add additional 5 zeros to the end of the array  
array = np.append(array, np.zeros(5))  
print("Array with 5 zeros at the end:", array)  
  
# Example 3  
# Generate 10 values for the function f(x) = x^2 on the domain [1, 10]  
x = np.linspace(1, 10, 10)  
f\_x = x \*\* 2  
print("Values of f(x)=x^2:", f\_x)  
  
# Example 3  
# Create a matrix A1 with shape [2,3]  
A1 = np.full((2, 3),1)  
print("Matrix A1 with shape [2,3]:", A1)  
  
# Example 4  
# Create a matrix A2 with shape [2,2]  
A2 = np.full((2, 2),2)  
print("Matrix A2 with shape [2,2]:", A2)  
  
# Example 5  
# Concatenate A2 with A1 to create A3 with shape [2,5]  
A3 = np.hstack((A1, A2))  
print("Matrix A3 with shape [2,5]:", A3)  
  
# Example 6  
# Create a matrix A4 with shape [3,5]  
A4 = np.full((3, 5),4)  
print("Matrix A4 with shape [3,5]:", A4)  
  
# Example 7  
# Concatenate A3 with A4 to create A5 with shape [5,5]  
A5 = np.vstack((A3, A4))  
print("Matrix A5 with shape [5,5]:", A5)

Array filled with 5s: [5 5 5 5 5 5 5 5 5 5]  
Array with 5 zeros at the end: [5. 5. 5. 5. 5. 5. 5. 5. 5. 5. 0. 0. 0. 0. 0.]  
Values of f(x)=x^2: [ 1. 4. 9. 16. 25. 36. 49. 64. 81. 100.]  
Matrix A1 with shape [2,3]:  
 [[1 1 1]  
 [1 1 1]]  
Matrix A2 with shape [2,2]:  
 [[2 2]  
 [2 2]]  
Matrix A3 with shape [2,5]:  
 [[1 1 1 2 2]  
 [1 1 1 2 2]]  
Matrix A4 with shape [3,5]:  
 [[4 4 4 4 4]  
 [4 4 4 4 4]  
 [4 4 4 4 4]]  
Matrix A5 with shape [5,5]:  
 [[1 1 1 2 2]  
 [1 1 1 2 2]  
 [4 4 4 4 4]  
 [4 4 4 4 4]  
 [4 4 4 4 4]]

## Linear algebra with Numpy

**The following text will use vectors and matrices with elements in the set of real numbers .**

Vector addition with np is quite intuitive as it uses the + operator. In mathematics, vector addition is defined between two vectors with the same dimension. If the two vectors have different dimensions, a ValueError exception is thrown.

**Example**: Sum two vectors v1 = [1, 2, 3] and v2 = [3, 4, 5], then print v1, v2, and v1 + v2.

import numpy as np  
  
v1 = np.array([1,2,3])  
v2 = np.array([3,4,5])  
  
v3 = v1 + v2  
  
print(f"Result of addition of {v1} and {v2} is {v3}")

Result of addition of [1 2 3] and [3 4 5] is [4 6 8]

Matrix addition with np is also possible and follows the same rules as vector addition, as matrices and vectors are represented with np.arrays.

**Example**. Sum two matrices

and

import numpy as np  
A = np.array([[1, 2], [3, 4]])  
B = np.array([[5, 6], [7, 8]])  
C = A + B  
print("Sum of A and B:")  
print(C)

Sum of A and B:  
[[ 6 8]  
 [10 12]]

### Transposed vectors and matrices

In mathematics, a transposed vector of a vector

is

Similarly, for a matrix, a transposed matrix of a matrix

is

Transposing a matrix in NumPy can be done using A.T, where A is the matrix. Since in NumPy a vector is a special case of a matrix or an array, the same method applies to vectors.

*The interpretation of transposing a matrix is creating a new matrix where the rows from the original matrix become columns and the columns become rows.*

**Examples:**

1. For a given vector, print the transposed vector.
2. For a given matrix, print the transposed matrix.
3. For a given matrix, print the transposed matrix.

import numpy as np  
# Example 1  
# Creating a 2D array (matrix)  
a = np.array([1, 2, 3])  
  
print("The transposed vector for the vector:")  
print(a)  
print("is:")  
print(a.T)  
  
  
# Example 2  
# Creating a 2D array (matrix)  
A = np.array([[1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 9]])  
print("The transposed matrix for the matrix:")  
print(A)  
print("is:")  
print(A.T)

The transposed vector for the vector:  
[1 2 3]  
is:  
[1 2 3]  
The transposed matrix for the matrix:  
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]  
is:  
[[1 4 7]  
 [2 5 8]  
 [3 6 9]]

### Dot Product of Vectors and Product of Matrices

The dot product for vectors and in where:

is defined as:

The expression is called the Euclidean norm of the vector and is denoted as .

*The interpretation of this norm is the size or length of the vector.*

The vector product is also equal to:

where is the angle between and

The cosine of the angle between two vectors can be calculated with: pmb

*The dot product is usually used for finding a projection of the vector on the vector ,* .

Similarly, for matrices, product of two matrices and with shapes [n,k] and [k,m] accordingly is an matrix with a shape [n,m] where the element is a dot product of the i’th row vector and the j’th column vector of .

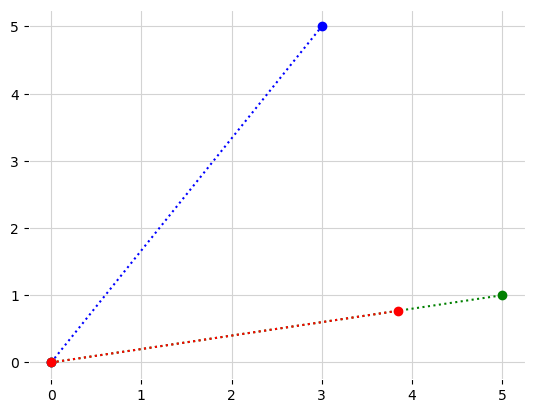
In Nympy the dot product can be found with np.dot(a, b) and contrary to the mathematical definition it does not require that one of the vectors be transposed.

**Examples:**

1. Using the properties of the dot product and given two vectors a=[2,3] and b=[4,5] find a vector a\_b that will be the projection of a on b and plot it.
2. Given the linear function f(x\_1..x\_4)=3x\_1+5x\_2-4x\_3+19x\_4 + 3 in x = [3,4,5,1], using the dot product x\*a where a are the parameters of the linear function a = [3,5,-4,19,3]

import numpy as np  
import matplotlib.pyplot as plt  
  
# Example 1:  
v1 = np.array([3,5])  
v2 = np.array([5,1])  
v1\_v2 = np.dot(v1, v2)  
v2\_v2 = np.dot(v2, v2)  
v1\_proj\_v2=(v1\_v2/(v2\_v2))\*v2  
plt.plot([0, v1[0]], [0, v1[1]], 'k:' ,color='b', label='v1', marker='o')  
plt.plot([0, v2[0]], [0, v2[1]], 'k:' ,color='g', label='v1', marker='o')  
plt.plot([0, v1\_proj\_v2[0]], [0, v1\_proj\_v2[1]], 'k:', color='r', label='v1', marker='o')  
plt.grid()  
plt.show()  
  
# Example 2:  
x = np.array([3,4,5,1])  
x = np.append(x, 1)  
a = np.array([3,5,-4,19,3])  
fx = np.dot(x,a)  
  
print(f"the value of f(x\_1..x\_4)=3x1+5x2-4x3+19x4+3 in x=(3,4,5,1) is {fx}")

/var/folders/4s/hhswlq455zv2br9bkfsygsfr0000gn/T/ipykernel\_74508/1885378213.py:10: UserWarning: color is redundantly defined by the 'color' keyword argument and the fmt string "k:" (-> color='k'). The keyword argument will take precedence.  
 plt.plot([0, v1[0]], [0, v1[1]], 'k:' ,color='b', label='v1', marker='o')  
/var/folders/4s/hhswlq455zv2br9bkfsygsfr0000gn/T/ipykernel\_74508/1885378213.py:11: UserWarning: color is redundantly defined by the 'color' keyword argument and the fmt string "k:" (-> color='k'). The keyword argument will take precedence.  
 plt.plot([0, v2[0]], [0, v2[1]], 'k:' ,color='g', label='v1', marker='o')  
/var/folders/4s/hhswlq455zv2br9bkfsygsfr0000gn/T/ipykernel\_74508/1885378213.py:12: UserWarning: color is redundantly defined by the 'color' keyword argument and the fmt string "k:" (-> color='k'). The keyword argument will take precedence.  
 plt.plot([0, v1\_proj\_v2[0]], [0, v1\_proj\_v2[1]], 'k:', color='r', label='v1', marker='o')



png

the value of f(x\_1..x\_4)=3x1+5x2-4x3+19x4+3 in x=(3,4,5,1) is 31

### Cross product

The cross product of two vectors is exclusively defined for vectors in three dimensions. It is denoted with , and for vectors and , it is defined as the vector :

Also, the norm of can be expressed as a product of the norms of and and the of the angle between them.

*Interpretation: In geometry the cross product of two vectors is an orthogonal vector to the plain that the vectors form, the size of the vector is the area of the parallelogram formed by the two vectors*

In np the cross product of a and b is calculated with the function np.cross(a, b).

**Examples:**

1. Calculate the cross product of the perpendicular vectors [1, 0, 0], [0, 1, 0].
2. Calculate the cross product of the non-perpendicular non-linear dependent vectors [1,2,3], [3,4,5].
3. Calculate the cross product of the linear dependent vectors [1, 0, 0], [5, 0, 0].

import numpy as np  
  
# Example 1  
v1 = np.array([1, 0, 0])  
v2 = np.array([0, 1, 0])  
c = np.cross(v1, v2)  
print(f'Example 1: Cross product of {v1} and {v2} = {c}')  
  
# Example 2  
v1 = np.array([1, 2, 3])  
v2 = np.array([3, 4, 5])  
c = np.cross(v1, v2)  
  
print(f'Example 2: Cross product of {v1} and {v2} = {c}')  
  
# Example 3  
v1 = np.array([1, 0, 0])  
v2 = np.array([5, 0, 0])  
c = np.cross(v1, v2)  
print(f'Example 3: Cross product of {v1} and {v2} = {c}')

Example 1: Cross product of [1 0 0] and [0 1 0] = [0 0 1]  
Example 2: Cross product of [1 2 3] and [3 4 5] = [-2 4 -2]  
Example 3: Cross product of [1 0 0] and [5 0 0] = [0 0 0]

### Determinant of a Matrix

The determinant of a matrix is generally used to solve a system of linear equations. It is used to determine if the system has a unique solution, no solution, or infinitely many solutions. Other uses of the determinant include finding an inverse of for a given matrix. In general, for a set of vectors, if the determinant is equal to 0, it indicates that there are at least two vectors that are linearly dependent.

np uses the formula np.linalg.det(A) to calculate the determinant of A.

**Example**: 1. Find the determinant of a matrix of linear dependent row vectors [[1, 2, 3, 4], [2, 4, 6, 8], [1, 0, 1, 0], [0, 1, 0, 1]] 2. Find the determinant of a matrix of linear independent row vectors: [[1, 2, 3, 4], [5, 7, 8, 9], [10, 11, 13, 14], [15, 16, 17, 19]]

import numpy as np  
  
# Example 1  
dependent\_matrix = np.array([  
 [1, 2, 3, 4],  
 [2, 4, 6, 8], # This row is 2 times the first row  
 [1, 0, 1, 0],  
 [0, 1, 0, 1]  
])  
  
det\_dependent = np.linalg.det(dependent\_matrix)  
print(f"Determinant of the linearly dependent matrix: {det\_dependent:.2f}")  
  
# Example 2  
independent\_matrix = np.array([  
[1, 2, 3, 4],  
 [5, 7, 8, 9],  
 [10, 11, 13, 14],  
 [15, 16, 17, 19]  
])  
  
det\_independent = np.linalg.det(independent\_matrix)  
print(f"Determinant of the linearly independent matrix: {det\_independent:.2f}")

Determinant of the linearly dependent matrix: 0.00  
Determinant of the linearly independent matrix: -26.00

### Unit/Identity Matrix and Inverse Matrix

The matrix with shape [n,n] is called an identity matrix or unite matrix if:

Numpy has a build in function for creating an identity matrix called np.identity(<size>). The identity matrix is defined only by the number of the columns or rows. Because of this np.identity(<size>) only requires one positive integer to create an identity matrix.

size = 5  
I = np.identity(size)

And an inverse matrix of is a matrix that multiplied with is equal to the inverse matrix .

Finding an inverse matrix for matrix A in np is done with the function inv\_A = np.linalg.inv(A), in case the determinant of is , does not have a inverse matrix and then is called a singular matrix. Calling inv\_A = np.linalg.inv(A) on a singular matrix will result in throwing np.linalg.LinAlgError exception. This behavior results in two possible options, first, before calling inv\_A = np.linalg.inv(A) check if the determinant is 0 or second, handle the exception np.linalg.LinAlgError.

Case one:

det\_A = np.linalg.det(A)  
if det\_A == 0:  
 print("Matrix A is singular")  
else:  
 A\_inv = np.linalg.inv(A)  
 print("Inverse Matrix of a A is")  
 print(A\_inv)

Case two:

try:  
 # Step 3: Compute the inverse of A  
 A\_inv = np.linalg.inv(A)  
 print("Inverse Matrix of a A is")  
 print(A\_inv)  
except np.linalg.LinAlgError:  
 print("Matrix A is singular")

**Examples:**

1. Create identity matrix I with size 5.
2. Create random not zero matrix A with shape [5,5] and find the inverse, A\_inv of A.
3. Verify that A\*A\_inv is equal to the identity **Use np.round(A, 3) to round the elements of the matrices to 3 decimals**.

import numpy as np  
  
def create\_random\_square\_matrix(size):  
 A = np.random.rand(size, size)  
 while np.linalg.det(A) == 0:  
 A = np.random.rand(size, size)  
 # returns the matrix only if the determinant is different than 0  
 return A  
  
# Example 1: Create an identity matrix I with size 5  
I = np.eye(5)  
print("Identity matrix I:")  
print(I)  
print()  
  
# Example 1: Create a random non-zero matrix A with shape [5, 5]  
A = np.round(create\_random\_square\_matrix(5),3)  
  
# Example 2: Find the inverse of A (A\_inv)  
A\_inv = np.round(np.linalg.inv(A),3)  
print("Random non-zero matrix A:")  
print(A)  
print("Inverse of matrix A (A\_inv):")  
print(A\_inv)  
print()  
  
# Example 3: Verify that A \* A\_inv is equal to the identity matrix  
A\_times\_A\_inv = np.round(np.dot(A, A\_inv))  
print("Product of A and A\_inv (should be close to identity matrix):")  
print(A\_times\_A\_inv)  
print()

Identity matrix I:  
[[1. 0. 0. 0. 0.]  
 [0. 1. 0. 0. 0.]  
 [0. 0. 1. 0. 0.]  
 [0. 0. 0. 1. 0.]  
 [0. 0. 0. 0. 1.]]  
  
Random non-zero matrix A:  
[[0.65 0.693 0.595 0.507 0.003]  
 [0.497 0.688 0.574 0.188 0.082]  
 [0.063 0.769 0.777 0.384 0.547]  
 [0.629 0.41 0.164 0.587 0.816]  
 [0.978 0.56 0.756 0.183 0.219]]  
  
Inverse of matrix A (A\_inv):  
[[-0.158 0.408 -0.964 0.414 0.717]  
 [-1.148 4.834 -0.836 0.627 -2.043]  
 [ 0.805 -3.406 1.607 -1.246 1.894]  
 [ 2.808 -3.133 0.491 0.07 -0.351]  
 [-1.483 0.196 0.488 0.792 0.346]]  
  
Product of A and A\_inv (should be close to identity matrix):  
[[ 1. 0. 0. 0. 0.]  
 [ 0. 1. 0. 0. 0.]  
 [-0. 0. 1. 0. 0.]  
 [ 0. 0. 0. 1. 0.]  
 [ 0. 0. 0. 0. 1.]]

### Eigenvalue and Eigenvectors of a matrix

Given a matrix , vector and scalar that satisfy the equation . Then is called eigenvalue of the matrix and and is called eigenvector of the matrix corresponding to the eigenvalue .

*Interpretation: Actually, and are such that the scalar multiple and the matrix multiple are one and same vector. In other words, the linear transformation maps the vector $ $ in its collinear vector . It means that the direction defined by is invariant under the linear mapping .*

Numpy offers an elegant option to calculate the Eigenvalue and the associated Eigenvectors to the value.

**Example**: Given a matrix [[1 2 3][4 5 6][7 8 9]], find the Eigenvalues and the corresponding Eigenvectors

import numpy as np  
  
A = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])  
  
eigenvalues, eigenvectors = np.linalg.eig(A)  
  
# iterates the elements in eigenvalues, where index and eigenvalue are the  
# index and the appropriate value in every iteration  
for index, eigenvalue in enumerate(eigenvalues):  
 print(f"The eigenvalue {eigenvalue} has the appropriate eigenvectors {eigenvectors}")

The eigenvalue 16.116843969807043 has the appropriate eigenvectors [[-0.23197069 -0.78583024 0.40824829]  
 [-0.52532209 -0.08675134 -0.81649658]  
 [-0.8186735 0.61232756 0.40824829]]  
The eigenvalue -1.1168439698070427 has the appropriate eigenvectors [[-0.23197069 -0.78583024 0.40824829]  
 [-0.52532209 -0.08675134 -0.81649658]  
 [-0.8186735 0.61232756 0.40824829]]  
The eigenvalue -1.3036777264747022e-15 has the appropriate eigenvectors [[-0.23197069 -0.78583024 0.40824829]  
 [-0.52532209 -0.08675134 -0.81649658]  
 [-0.8186735 0.61232756 0.40824829]]

### Solving equations

A system of linear equations can be represented with matrices in the following form .

In mathematics there are multiple ways of solving a system. Python uses numpy.linalg.solve(A,B)

**Examples:**

1. Solve the system A\*X = B where A =[[2, 3, 1],[4, 1, 2],[3, 2, 3]] and B = [1, 2, 3] and verify the result.
2. Solve the system A\*X = B where A = [[1,3,-2],[3,-2,-1]] and B=[1,-4].

import numpy as np  
  
# Example 1  
A = np.array([[2, 3, 1],  
 [4, 1, 2],  
 [3, 2, 3]])  
  
B = np.array([1, 2, 3])  
X = np.linalg.solve(A,B)  
  
print(f"X = {X}")  
print(f"A\*X = {np.dot(A,X)}")  
  
# Example 2  
  
A = np.array([[1,3,-2,1],[3,-2,-1,-4]])  
B = np.array([1, -4])  
  
X = np.linalg.solve(A,B)

X = [0. 0. 1.]  
A\*X = [1. 2. 3.]  
  
  
  
---------------------------------------------------------------------------  
  
LinAlgError Traceback (most recent call last)  
  
Cell In[123], line 19  
 16 A = np.array([[1,3,-2,1],[3,-2,-1,-4]])  
 17 B = np.array([1, -4])  
---> 19 X = np.linalg.solve(A,B)  
  
  
File /opt/anaconda3/lib/python3.12/site-packages/numpy/linalg/linalg.py:396, in solve(a, b)  
 394 a, \_ = \_makearray(a)  
 395 \_assert\_stacked\_2d(a)  
--> 396 \_assert\_stacked\_square(a)  
 397 b, wrap = \_makearray(b)  
 398 t, result\_t = \_commonType(a, b)  
  
  
File /opt/anaconda3/lib/python3.12/site-packages/numpy/linalg/linalg.py:213, in \_assert\_stacked\_square(\*arrays)  
 211 m, n = a.shape[-2:]  
 212 if m != n:  
--> 213 raise LinAlgError('Last 2 dimensions of the array must be square')  
  
  
LinAlgError: Last 2 dimensions of the array must be square

The system A\*x = B where A = [[1,3,-2],[3,-2,-1]] and B=[1,-4] from the last example cannot be solved with np.linalg.solve(A, B). This is because np.linalg.solve expects a square matrix. The solution for this kind of system is not automated. To solve this system, the Gaussian Elimination method can be used. The custom\_pivot\_column function from the next section of exercises can be used to transform a specified column of a matrix such that all elements in that column are zero except for the chosen pivot element, which is set to one.

### Exercises with np

import numpy as np  
  
# E. Create an array with size 10, filled with the number 5  
array = np.full(10, 5)  
print(array)  
  
# E. add additional 5 zeros to the end of the array  
array = [1,2,3,4]  
np.concatenate((array, np.full(5, 0)))  
print(array)  
  
# E. Add a row to matrix  
matrix = np.array([[1, 2, 3],  
 [4, 5, 6]])  
new\_row = np.array([7, 8, 9])  
matrix\_with\_new\_row = np.vstack((matrix, new\_row))  
print("Matrix with the new row:")  
print(matrix\_with\_new\_row)  
  
# E. Get the number of rows and number of columns of a matrix  
matrix = np.array([[1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 9]])  
  
# Get the number of rows and columns of the matrix  
num\_rows, num\_cols = matrix.shape  
  
print(f"num\_rows, num\_cols = ({num\_rows},{num\_cols})")  
  
  
# E. Create sub matrix from a given matrix  
matrix = np.array([[1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 9]])  
  
# Get a sub-matrix from the original matrix  
# Define the row and column indices of the sub-matrix  
start\_row = 0  
end\_row = 2 # Note: End index is exclusive, so it will include rows up to end\_row - 1  
start\_col = 1  
end\_col = 3 # Note: End index is exclusive, so it will include columns up to end\_col - 1  
  
# Extract the sub-matrix using slicing  
sub\_matrix = matrix[start\_row:end\_row, start\_col:end\_col]  
  
print(sub\_matrix)  
  
# E. From matrix A create matrix B that has the same rows but only the first 4 columns  
matrix = np.array([[1, 2, 3, 4, 5],  
 [6,7,8,9,10],  
 [11, 12, 13,14,15]])  
sub\_matrix = matrix[0:,0:4]  
print(sub\_matrix)  
  
  
# E. From matrix A create matrix B that contains all rows except  
# the first one contains the first 4 columns  
matrix = np.array([[1, 2, 3, 4, 5],  
 [6,7,8,9,10],  
 [11, 12, 13,14,15]])  
sub\_matrix = matrix[0:,0:4]  
print(sub\_matrix)  
  
# E. Split the matrix A in to two sub matrixes vertically, so that all rows  
# from index 0 to 3 will be part of the first sub matrix and all rows from  
# index 4 to the end wil lbe part of the second matrix  
# the first one contains the first 4 columns  
matrix = np.array([[1, 2, 3, 4, 5],  
 [6,7,8,9,10],  
 [11, 12, 13,14,15]])  
  
sub\_matrix1 = matrix[0:,0:3]  
sub\_matrix2 = matrix[0:,3:]  
  
# todo reformat  
print(f"sub\_matrix1 = {sub\_matrix1}, sub\_matrix2 = {sub\_matrix2}")  
  
  
# OPERATIONS WITH VECTORS  
  
# E. Vector addition, Sum the two vectors A = (1, 2, 3, 4, 5) and B = (3, 0, 3, 4, 1)  
a = np.array([1, 2, 3, 4, 5])  
b = np.array([3, 0, 3, 4, 1])  
c = a + b  
print(f"sum of a ={a} and b = {b} is {c}")  
  
  
# E. Inverse matrix, create an inverse matrix for a given matrix A = ([1,2],[5,6])  
  
a = np.array([[1,2],[5,6]])  
print("Inverse of: ")  
print(a)  
print("is: ")  
A\_inverse = np.linalg.inv(a)  
print(A\_inverse)  
  
# E. Inverse matrix, create an inverse matrix for a give matrix A = ([1,2],[5,6]) manually  
print("Inverse of: ")  
print(np.linalg.det(a))

[5 5 5 5 5 5 5 5 5 5]  
[1, 2, 3, 4]  
Matrix with the new row:  
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]  
num\_rows, num\_cols = (3,3)  
[[2 3]  
 [5 6]]  
[[ 1 2 3 4]  
 [ 6 7 8 9]  
 [11 12 13 14]]  
[[ 1 2 3 4]  
 [ 6 7 8 9]  
 [11 12 13 14]]  
sub\_matrix1 = [[ 1 2 3]  
 [ 6 7 8]  
 [11 12 13]], sub\_matrix2 = [[ 4 5]  
 [ 9 10]  
 [14 15]]  
sum of a =[1 2 3 4 5] and b = [3 0 3 4 1] is [4 2 6 8 6]  
Inverse of:   
[[1 2]  
 [5 6]]  
is:   
[[-1.5 0.5 ]  
 [ 1.25 -0.25]]  
Inverse of:   
-3.999999999999999

# Pivoting  
  
# In a matrix, pick an element and using linear transformations,  
# (multiplying a row with a scalar row and adding it to a different one)  
# transform the matrix so that the picked element will be transformed to one and  
# all the elements of the matrix in the column of the picked element will be equal  
# to 0  
import numpy as np  
from sympy import Matrix  
  
def custom\_pivot\_column(matrix, pivot\_row, pivot\_col):  
 # Convert the matrix to a NumPy array for easier manipulation  
 np\_matrix = np.array(matrix)  
  
 # Divide the pivot row by the pivot element to make it 1  
 pivot\_element = np\_matrix[pivot\_row, pivot\_col]  
 np\_matrix[pivot\_row, :] /= pivot\_element  
  
 # Eliminate other elements in the same column  
 num\_rows = np\_matrix.shape[0]  
 for i in range(num\_rows):  
 if i != pivot\_row:  
 ratio = np\_matrix[i, pivot\_col]  
 np\_matrix[i, :] -= ratio \* np\_matrix[pivot\_row, :]  
  
  
 # Convert the modified NumPy array back to a SymPy Matrix  
 pivoted\_matrix = Matrix(np\_matrix)  
  
 return pivoted\_matrix  
  
m = Matrix([  
 [1, -1, 2],  
 [3, 4, 5],  
 [0, 2, 8]])  
  
print(custom\_pivot\_column(m, 1,1))  
print(custom\_pivot\_column(m, 0,0))

Matrix([[7/4, 0, 13/4], [3/4, 1, 5/4], [-3/2, 0, 11/2]])  
Matrix([[1, -1, 2], [0, 7, -1], [0, 2, 8]])

# Optimization, approximation algorithms and the scipy.optimize library

The following sections of this seminar-work describe different methods used for finding an extreme or a root of a mathematical function. By convention this function will be called an **objective function**.

**In the following text and code examples, the numerical methods or optimization methods are referred to as approximation methods and their results as approximations.**

**The methods will be explained for the optimization problem of minimum. In case the maximum of objective function is required, it will be converted to the problem of minimum of .**

All optimization algorithms start with initial list of values called approximations. The initial values are stored in a problem-specific data structure using a variable called approximations.

All optimization algorithms are iterative; they define an algorithm-specific step function. This function is called on the approximations variable and returns a new approximation. All algorithms iteratively call the step function on the approximations variable until a certain condition (stopping criterion) is satisfied.

The general form of an optimization algorithm is presented as follows:

def approximations = []  
  
def step\_function(approximations:list[float]):  
 ... <!- calculates a new approximation>  
 return approximation  
  
def stop\_criterion(approximations:list[float]):  
 ...  
 return <!-- returns true or false -->  
  
def approximation\_method(approximations\_in, step\_function\_in, stop\_criterion\_in)->list[float]  
 while(not stop\_criterion\_in(approximations[-1])):  
 approximation = step\_function\_in(approximations)  
 approximations\_in.append(approximation)  
 return approximations\_in

The general solution uses an approximations stack/array to store all the results from the step function. This is not required but is useful for learning and debugging purposes. For example, investigating intermediate approximations by printing or visualizing and plotting them.

In Python, the library scipy from the package optimize can be used to find an approximation to an extreme or a root of a function. In the following text, I will implement different approximation methods and also provide the appropriate python function from scipy.optimize.

## Golden Search

Golden search method is applicable for a function on a closed interval when has only one minimum under the closed interval. The algorithm works in a way that for every step requires a list of four values for :

so that and and .

Then it analyzes the function at these four values. Depending on whether or , the minimum will be located in or in . For the two intervals and , three values of are already known or previously calculated: for and for . In order to again analyze for a minimum under the new reduced interval, a new value for needs to be picked. is picked using the following optimization:

for resulting in a new input list of four values for : where .

or

for resulting in a new input list of four values for : where .

The scalar p in the golden search algorithm is optimized and in every step is equal to:

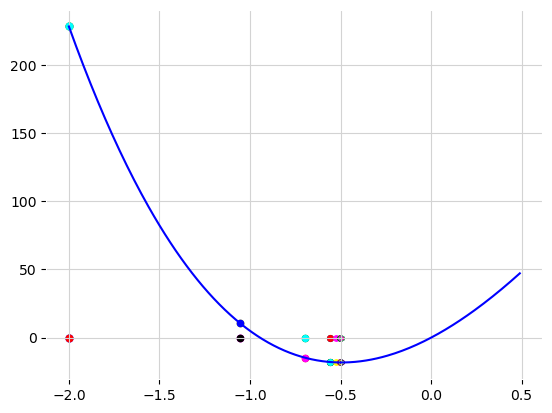
The described process is repeated with the new four values for , until the is at the required size.

The N-th step reduces the range(uncertainty interval) by factor:

**Example**: 1. Use the the golden search to find the value of x that minimizes the function: *([8], Example 7.1, “Introduction to Optimization”, page 94)* 2. Use the the golden search minimize\_scalar(f, method='golden') from the library scipy, package scipy.optimize to find the local minimum of the function

# tutorial: https://realpython.com/python-data-structures/  
  
#Example 1  
  
def F(x\_in):  
 return x\_in\*\*4-14\*x\_in\*\*3+60\*x\_in\*\*2+70\*x\_in  
  
# Approx, Approximations class holds tha values for x\_in and calculates f\_in for x\_in  
class Approx:  
 def \_\_init\_\_(self, x\_in, f\_in):  
 self.x = x\_in  
 self.\_f\_x = None  
 self.f = f\_in  
 # Lazily calculates f\_in at execution time and stores, for reuse  
 @property  
 def f\_x(self):  
 if(self.\_f\_x == None):  
 self.\_f\_x = self.f(self.x)  
 return self.\_f\_x  
  
# gives the range reduction for two Approximations  
def range\_reduction(a\_in:Approx, b\_in:Approx):  
 p = 0.382  
 return p\*abs(a\_in.x-b\_in.x)  
  
a0 = Approx(-2, F)  
b0 = Approx(2, F)  
a1 = Approx(a0.x + range\_reduction(a0, b0), F)  
b1 = Approx(b0.x - range\_reduction(a0, b0), F)  
  
x\_approximations = []  
x\_approximations.append((a0, a1, b1, b0))  
  
a0, a1, b1, b0 = x\_approximations[-1]  
  
  
def golden\_search\_minimizer\_step\_function(f\_in, approximations\_in):  
 a0, a1, b1, b0 = approximations\_in[-1]  
 if(a1.f\_x < b1.f\_x):  
 return (a0, Approx(a0.x + range\_reduction(a0,b1), f\_in), a1 ,b1)  
 else:  
 return (a1, b1, Approx(b0.x - range\_reduction(a1,b0), f\_in) ,b0)  
  
def golden\_search\_stop\_criterion(approximations\_in):  
 a0 = approximations\_in[-1][0]  
 b0 = approximations\_in[-1][3]  
 return b0.x - a0.x<0.0001  
  
def golder\_search\_method(approximations\_in, f\_in):  
 while(not golden\_search\_stop\_criterion(approximations\_in)):  
 current\_approx = golden\_search\_minimizer\_step\_function(f\_in, approximations\_in)  
 approximations\_in.append(current\_approx)  
 return approximations\_in  
  
x\_approximations = golder\_search\_method(x\_approximations, F)  
  
print(f"The minimizer using the manually developed algorithm is equal to X= {x\_approximations[-1][1].x}")  
  
# Plot  
import matplotlib.pyplot as plt  
import numpy as np  
import matplotlib.colors as mcolors  
  
x = np.arange(-2, +0.5, 0.01)  
a0\_list = [item[0] for item in x\_approximations]  
b0\_list = [item[3] for item in x\_approximations]  
plt.plot(x,F(x))  
c = 0  
for i in x\_approximations:  
 plt.scatter(i[0].x, 0, s=len(x\_approximations) - c)  
 plt.scatter(i[0].x, i[0].f\_x, s=len(x\_approximations) - c)  
 c = c+1  
plt.grid()  
  
#Example 2  
from scipy.optimize import minimize\_scalar  
minimizer\_python = minimize\_scalar(F, method='golden')  
print(f"The minimizer using the minimize\_scalar algorithm is equal to X= {minimizer\_python.x}")

The minimizer using the manually developed algorithm is equal to X= -0.49395485895564095  
The minimizer using the minimize\_scalar algorithm is equal to X= -0.4939286751012236



png

## Newton’s Method

The Newton method for minimizing a function requires that the first derivative and the second derivative are known and also , , can be calculated.

It is given by the iterative formula:

*Interpretation: The Newton method uses a Taylor series function to approximate the objective function at every value. By solving or it finds the local extreme of the function . This solution is the new approximation, and the step is repeated until some stop condition is reached*

**Example**: 1. Manually find the minimizer of given the derivatives and , *([8], Example 7.3, “Introduction to Optimization”, page 103)* 2. Use the scipy.optimize.newton to find the minimizer of given the derivatives and , *([8], Example 7.3, “Introduction to Optimization”, page 103)*

In the following code, several improvements will be implemented:

1. All the optimization methods are iterative, they call a specific step function until a certain stop\_criterion is reached. This behavior will be generalized in the general\_approximation\_iterator Python function.
2. The approximation method-specific step function will be generated from a different specific approximation method builder function and will be only dependent on the approximations list.
3. The approximation method-specific stop criterion function will be generated from a different specific approximation method builder function and will be only dependent on the approximations list.

These changes will contribute to more flexibility in creating the approximation methods. Will alow combining, and reusing the methods.

import numpy as np  
  
  
# As all of the approximation methods will have the same interface a  
# a general function can be created that can take any starting values, any method  
# specific step function and any stop criterion function  
# and generate a approximation method  
def general\_approximation\_iterator(approximations\_in, stepfunction\_in, stop\_criterion\_in):  
 while(not stop\_criterion\_in(approximations\_in)):  
 approx = stepfunction\_in(approximations\_in)  
 approximations\_in.append(approx)  
 return approximations\_in  
  
  
# Example. 1  
def f\_derivative\_1(x\_in):  
 return x\_in - np.cos(x\_in)  
  
  
def f\_derivative\_2(x\_in):  
 return 1 + np.sin(x\_in)  
  
  
x\_approximations = []  
x\_approximations.append(0.5)  
  
  
def step\_newtons\_method\_builder(f\_dv\_1\_in, f\_dv\_2\_in):  
 def step\_function(approximations\_in):  
 der1 = f\_dv\_1\_in(approximations\_in[-1])  
 der2 = f\_dv\_2\_in(approximations\_in[-1])  
 return approximations\_in[-1] - der1/der2  
 return step\_function  
  
  
def stop\_criterion\_relative\_step(accuracy\_in: float):  
 def stop\_function(approximations\_in):  
 if(len(approximations\_in)>=2):  
 step\_size = abs((approximations\_in[-1]-approximations\_in[-2])/approximations\_in[-1])  
 return step\_size<accuracy\_in  
 return False  
 return stop\_function  
  
  
def newtons\_method(approximations\_in, step\_function\_in, stop\_criterion\_in):  
 return general\_approximation\_iterator(approximations\_in, step\_function\_in, stop\_criterion\_in)  
  
  
step\_newtons\_method = step\_newtons\_method\_builder(f\_derivative\_1, f\_derivative\_2)  
accuracy = 0.0000001  
stop\_newtons\_method = stop\_criterion\_relative\_step(accuracy)  
  
  
x\_approximations = newtons\_method(x\_approximations, step\_newtons\_method, stop\_newtons\_method)  
print(f"Minimizer calculated manually X={x\_approximations[-1]}")  
  
  
# Example. 2  
from scipy.optimize import newton  
root\_newton = newton(f\_derivative\_1, x0=0.5)  
print(f"Minimizer calculated with scipy.optimize.newton X={root\_newton}")

Minimizer calculated manually X=0.7390851332151607  
Minimizer calculated with scipy.optimize.newton X=0.7390851332151601

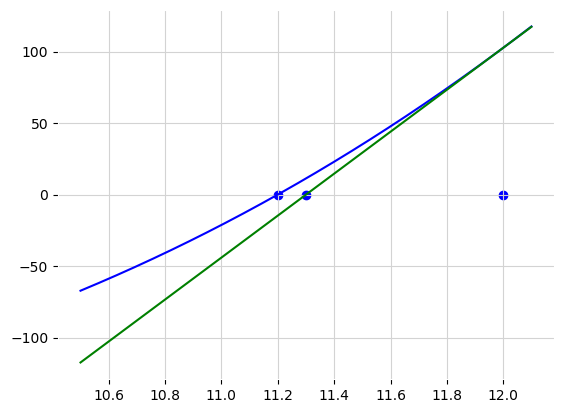
Since a function has the extreme at the same point where has value 0. With small modification the Newton’s method can be also used for finding the root of the function . This is the iterative formula for finding a root of a function :

**Example**: 1. Manually find the root of the function given its derivative, using the newton’s method. *([8], Example 7.4, “Introduction to Optimization”, page 105)*

1. Use the scipy.optimize.newton to find the root of the function $ g(x) = x^3 - 12.2x^2 + 7.45x + 42 = 0 $ given its derivative, using the newton’s method.

def g(x):  
 return x\*\*3 - 12.2\*x\*\*2 + 7.45\*x + 42  
def g\_derivative(x):  
 return 3\*x\*\*2 - 24.4\*x + 7.45  
  
  
# Example 1  
# this block uses the previous block functions  
step\_newtons\_method = step\_newtons\_method\_builder(g, g\_derivative)  
accuracy = 0.000001  
stop\_newtons\_method = stop\_criterion\_relative\_step(accuracy)  
  
  
x\_approximations = []  
x\_approximations.append(12)  
x\_approximations = newtons\_method(x\_approximations, step\_newtons\_method, stop\_newtons\_method)  
print(f"Root manually calculated, X={x\_approximations[-1]}")  
  
  
# plot the function and the approximations  
import matplotlib.pyplot as plt  
x\_plot = np.linspace(10.50, 12.1, 400)  
print(x\_approximations)  
plt.scatter([x\_approximations[0],x\_approximations[1],x\_approximations[2]], [0,0,0])  
plt.plot(x\_plot, g(x\_plot))  
plt.plot(x\_plot, g\_derivative(x\_approximations[0])\*x\_plot + g(x\_approximations[0])-g\_derivative(x\_approximations[0])\*x\_approximations[0])  
plt.grid()  
plt.show()  
  
  
# Example 2  
from scipy.optimize import newton  
root\_newton = newton(g, x0=0.5)  
print(f"Root calculated with scipy.optimize.newton, X={root\_newton}")

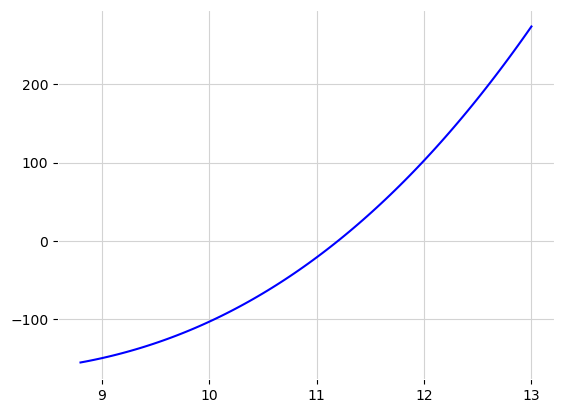
Root manually calculated, X=11.200000000000093  
[12, 11.300375042618478, 11.201895469742569, 11.200000695476783, 11.200000000000093]



png

Root calculated with scipy.optimize.newton, X=11.199999999999998

# Secant algorithm, Example 7,5 page 107  
  
import numpy as np  
import matplotlib.pyplot as plt  
from scipy.optimize import newton  
  
# Example 1  
  
# Find the root of the function  
def g(x\_in:float)->float:  
 return x\_in\*\*3 - 12.2\*x\_in\*\*2 + 7.45\*x\_in + 42  
  
# Plot  
x\_plot = np.linspace(8.8, 13, 100)  
# Generate data  
x\_plot = np.linspace(8.8, 13, 100)  
  
# Create the plot  
fig, ax = plt.subplots()  
  
# Plot the data  
plt.plot(x\_plot, g(x\_plot))  
plt.grid()  
  
# Show the plot  
plt.show()  
  
def secant\_method\_step\_builder(f\_in):  
 def step\_function(x\_approximations\_in):  
 \_a = (f\_in(x\_approximations\_in[-1])\*x\_approximations\_in[-2] - f\_in(x\_approximations\_in[-2])\*x\_approximations\_in[-1])  
 \_b = (f\_in(x\_approximations\_in[-1]) - f\_in(x\_approximations\_in[-2]))  
 return \_a/\_b  
 return step\_function  
  
  
def secant\_method(approximations\_in, function\_in, stop\_criterion\_in):  
 return general\_approximation\_iterator(approximations\_in,  
 secant\_method\_step\_builder(function\_in),  
 stop\_criterion\_in)  
  
x\_approximations = []  
x\_approximations.append(13)  
x\_approximations.append(12)  
  
stop\_criterion\_secant = stop\_criterion\_relative\_step(1e-12)  
  
secant\_method(x\_approximations, g , stop\_criterion\_secant)  
  
  
for element in enumerate(x\_approximations):  
 print(f"Step {element[0]} approximation value {element[1]},")  
  
# Example 2  
from scipy.optimize import newton  
# x0 x1 are the two starting values for secant method  
root\_secant = newton(g, x0=13, x1=12)  
print("Root found by scipy.optimize.newton Secant method:", root\_secant)



png

Step 0 approximation value 13,  
Step 1 approximation value 12,  
Step 2 approximation value 11.401574803149607,  
Step 3 approximation value 11.227209417308144,  
Step 4 approximation value 11.201027680731215,  
Step 5 approximation value 11.200005393481776,  
Step 6 approximation value 11.200000001073377,  
Step 7 approximation value 11.2,  
Step 8 approximation value 11.2,  
Root found by scipy.optimize.newton Secant method: 11.200000000000033

## Gradient method

The Gradient method is used to approximate an extremum of a function . Similar to the Newton method, it requires the knowledge of the derivative of in the approximation steps .

* Given , and $ c $.
* The gradient of at is a vector in , denoted by .
* points in the direction of the maximum rate of increase of .
* points in the direction of the maximum rate of decrease of .

For a starting value , the value will be a “move” in the direction of the local minimum:

For a starting the value , the value will be a “move” in the direction of the local maximum:

The scalar is the and is always bigger than 0. Different variations of the algorithm exist, that use different strategies to re-calculated on every iteration. Correctly choosing is quite important, picking a random might not result in convergence.

Unlike the previous example, where approximation was done on a one-dimensional function, the following gradient method algorithm approximates an n-dimensional function. As a result, the steps are n-dimensional dots/vectors. Consequently, the algorithm stops when the distance between and is smaller than a given accuracy, or when the norm of the vector is smaller than a required accuracy.

**Example**. Find the minimizer of , *([8], Page 118, Example 8.1, “An Introduction to Optimization”)*.

# Gradient method with fixed step alfa  
import numpy as np  
import matplotlib.pyplot as plt  
  
# Example 1  
  
def f(x\_in):  
 return (x\_in[0]-4)\*\*4 + (x\_in[1]-3)\*\*2 + 4\*(x\_in[2]+5)\*\*4  
  
def delta\_f(x\_in):  
 return np.array([4\*(x\_in[0]-4)\*\*3, 2\*(x\_in[1]-3), 16\*(x\_in[2]+5)\*\*3])  
  
def gradient\_method\_step\_builder(delta\_f\_in, alfa\_in):  
 def step\_function(approximations\_in):  
 return approximations\_in[-1] - alfa\_in\*delta\_f\_in(approximations\_in[-1])  
 return step\_function  
  
def gradient\_method(approximations\_in, function\_in, stop\_criterion\_in, alfa\_in):  
 return general\_approximation\_iterator(approximations\_in,  
 gradient\_method\_step\_builder(function\_in,  
 alfa\_in),  
 stop\_criterion\_in)  
  
def stop\_criteria\_relative\_multi\_dimensional(accuracy\_in:float):  
 def stop\_criterion(approximations\_in: list):  
 if len(approximations\_in) > 1:  
 a\_ = np.linalg.norm(approximations\_in[-1] - approximations\_in[-2])  
 b\_ = np.linalg.norm(approximations\_in[-2])  
 return (a\_/min(1,b\_))<accuracy\_in  
 return False  
 return stop\_criterion  
  
x\_approximations = []  
x0 = np.array([4,2,-1])  
x\_approximations.append(x0)  
  
approximations\_gradient = gradient\_method(approximations\_in = x\_approximations,  
 function\_in = delta\_f,  
 stop\_criterion\_in = stop\_criteria\_relative\_multi\_dimensional(accuracy\_in = 0.00001),  
 alfa\_in = 0.00001)  
  
# The algorithm is not really efficient as alpha is fixed  
# and conservatively selected.  
# The following code prints every 1000 approximation  
for i, element in enumerate(approximations\_gradient):  
 if i % 1000 == 0:  
 print(f" {element},")  
  
print(f"The algorithm achieved accuracy of {accuracy} in {len(approximations\_gradient)} steps")

[ 4 2 -1],  
 [ 4. 2.01980152 -3.38401547],  
 [ 4. 2.03921095 -3.80739339],  
 [ 4. 2.05823603 -4.01138817],  
 [ 4. 2.07688439 -4.13717301],  
 [ 4. 2.09516349 -4.22461701],  
 [ 4. 2.11308063 -4.28992785],  
 [ 4. 2.13064298 -4.34110152],  
 [ 4. 2.14785757 -4.3825981 ],  
 [ 4. 2.16473129 -4.41712757],  
 [ 4. 2.18127088 -4.44644404],  
 [ 4. 2.19748297 -4.4717397 ],  
 [ 4. 2.21337403 -4.49385691],  
 [ 4. 2.22895042 -4.51341003],  
 [ 4. 2.24421837 -4.53085906],  
 [ 4. 2.259184 -4.54655616],  
 [ 4. 2.27385329 -4.56077616],  
 [ 4. 2.2882321 -4.57373709],  
 [ 4. 2.30232619 -4.58561441],  
 [ 4. 2.31614119 -4.59655115],  
 [ 4. 2.32968264 -4.60666527],  
 [ 4. 2.34295594 -4.61605495],  
 [ 4. 2.35596641 -4.62480276],  
 [ 4. 2.36871926 -4.63297864],  
 [ 4. 2.38121958 -4.64064231],  
 [ 4. 2.39347237 -4.64784512],  
 [ 4. 2.40548254 -4.65463148],  
 [ 4. 2.41725489 -4.66104003],  
 [ 4. 2.42879413 -4.66710458],  
 [ 4. 2.44010488 -4.67285484],  
 [ 4. 2.45119166 -4.67831705],  
 [ 4. 2.4620589 -4.68351448],  
 [ 4. 2.47271095 -4.68846786],  
 [ 4. 2.48315208 -4.69319571],  
 [ 4. 2.49338645 -4.69771464],  
 [ 4. 2.50341817 -4.7020396 ],  
 [ 4. 2.51325125 -4.70618407],  
 [ 4. 2.52288962 -4.71016027],  
 [ 4. 2.53233713 -4.71397929],  
The algorithm achieved accuracy of 1e-06 in 38331 steps

## Gradient with step adjustment, Steepest descent

In-order to optimize when minimizing , it is requeued to finding a positive so for the next , the function will have the smallest possible value.

Since the vectors/dots, and are already know, optimizing would be finding a minimum of the one dimensional function:

The minimizer of can be found using some of the previous approximation methods that are applicable on one dimensional functions.

**Examples:**

1. Develop an algorithm for the Steepest descent method and find the minimizer of where the step is adjusted using the secant\_method method, *([8], Page 118, Example 8.1 of “An Introduction to Optimization”)*.
2. Find the minimizer of using the python’s minimize from the scipy.optimize import minimize, *([8], Page 118, Example 8.1 of “An Introduction to Optimization”)*.

# Gradient with step adjustment alpha or Steepest descent  
  
import numpy as np  
  
def f(x\_in) -> float:  
 return (x\_in[0]-4)\*\*4 + (x\_in[1]-3)\*\*2 + 4\*(x\_in[2]+5)\*\*4  
  
def delta\_f(x\_in):  
 return np.array([4\*(x\_in[0]-4)\*\*3, 2\*(x\_in[1]-3), 16\*(x\_in[2]+5)\*\*3])  
  
# This structure creates the arg\_min initially dependent on the objective  
# function function\_in and then on the current approximation of function\_in  
# in the respective iteration.  
def arg\_min\_builder(function\_in, delta\_function\_in):  
 def alpha\_arg\_min\_approximation(current\_approximation\_in):  
 def arg\_min(alpha\_in: float):  
 delta\_x = delta\_function\_in(current\_approximation\_in)  
 alpha\_delta = alpha\_in\*delta\_x  
 return function\_in(current\_approximation\_in-alpha\_delta)  
 return arg\_min  
 return alpha\_arg\_min\_approximation  
  
# The derivate is used to approximate the alpha  
def derivative\_of(function\_in):  
 def numerical\_derivative(x\_in):  
 \_delta\_x = 1e-5  
 \_derivate\_f = (function\_in(x\_in + \_delta\_x) - function\_in(x\_in - \_delta\_x))/(2\*\_delta\_x)  
 return \_derivate\_f  
 return numerical\_derivative  
  
# Let's create a general function that will take an approximation\_method from the methods we have previously defined,  
# the approximations\_in, and the stop\_criterion\_in. It will return an approximation\_method function that will be able to  
# approximate any input function given the approximation\_method\_in, approximations\_in, and stop\_criterion\_in.  
def approximation\_method\_builder(approximations\_in, approximation\_method\_in, stop\_criterion\_in):  
 def approximation\_method(function\_in):  
 return approximation\_method\_in(approximations\_in.copy(), function\_in, stop\_criterion\_in)  
 return approximation\_method  
  
# Here Im defining a step function for the gradient method with adjustable alpha step  
# This function already uses the gradient\_method\_step\_builder function with fixed alpha  
# and approximates alpha based on the  
def gradient\_method\_step\_builder\_with\_adjustable\_alpha(f\_in, delta\_f\_in, alpha\_approximation\_method\_in):  
 alpha\_arg\_min\_builder\_from\_approximation = arg\_min\_builder(f, delta\_f)  
 def approximate\_alpha(approximations\_in):  
 alpha\_arg\_min\_function\_ = alpha\_arg\_min\_builder\_from\_approximation(current\_approximation\_in = approximations\_in[-1])  
 derivative\_alpha\_function = derivative\_of(alpha\_arg\_min\_function\_)  
 alpha\_approximations = alpha\_approximation\_method\_in(derivative\_alpha\_function)  
 alpha = alpha\_approximations[-1]  
 return alpha  
 def step\_function(approximations\_in):  
 alpha = approximate\_alpha(approximations\_in)  
 gradient\_step\_function = gradient\_method\_step\_builder(delta\_f\_in, alpha)  
 return gradient\_step\_function(approximations\_in)  
 return step\_function  
  
def gradient\_method\_with\_alpha\_adjustment(f\_in,  
 delta\_f\_in,  
 approximations\_in,  
 stop\_criterion\_in,  
 alpha\_approximation\_method\_in):  
 return general\_approximation\_iterator(approximations\_in,  
 gradient\_method\_step\_builder\_with\_adjustable\_alpha(  
 f\_in,  
 delta\_f\_in,  
 alpha\_approximation\_method\_in  
 ),  
 stop\_criterion\_in)  
  
  
  
x0 = np.array([4, 2, -1])  
x\_approximations = [x0]  
  
gradient\_method\_with\_alpha\_adjustment(f\_in = f,  
 delta\_f\_in = delta\_f,  
 approximations\_in = x\_approximations,  
 stop\_criterion\_in = stop\_criteria\_relative\_multi\_dimensional(accuracy\_in=0.00001),  
 alpha\_approximation\_method\_in = approximation\_method\_builder(  
 approximations\_in = [0.01, 0.08],  
 approximation\_method\_in = secant\_method,  
 stop\_criterion\_in = stop\_criterion\_relative\_step(0.0001)  
 ))  
# Print the result  
print("The minimum of f using the manually written algorithm is in ",  
 x\_approximations[-1], " and the minimum value is : ",  
 f(x\_approximations[-1]))  
  
  
# Example 2  
from scipy.optimize import minimize  
  
# The method minimize is a general method for minimizing a function f given its  
# gradient delta\_f, it is not strictly using the steepest descent algorithm  
# In this case `method='Newton-CG' the minimize function will use the Newton-CG algorithm  
# https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize  
result = minimize(f, x0, method='Newton-CG', jac=delta\_f)  
  
# Print the result  
print("The minimum of f using scipy.optimize.minimize is in ",  
 result.x, " and the minimum value is : "  
 ,result.fun)

The minimum of f using the manually written algorithm is in [ 4. 3. -5.00095904] and the minimum value is : 3.383828899365421e-12  
The minimum of f using scipy.optimize.minimize is in [ 4. 3.00000003 -4.99737168] and the minimum value is : 1.908869103936226e-10

## Linear programming

This type of optimization tries to find an extreme of a linear function , **called the objective function**, given a set of constraints that are linear equations or inequalities. Based on the type of constraints and the extreme of the function, a linear program can be written in different forms.

The following proposition is called a linear program in **standard form**:

Minimize:

Subject to:

Or represented with matrices:

Minimize:

Subject to:

A linear program can be given in a **nonstandard form**, and the difference between a standard and nonstandard program is in the set of constraints. If the set of constraints defined by the matrices and contains inequalities, then the linear program is in a nonstandard form.

The solution of the linear program is the vector that maximizes/minimizes the function .

*Some authors define the standard form in a way that the goal of the problem is to maximize the objective function.*

Every solution for that satisfies the set of constraints is called a feasible point.

A solution to a standard the linear program is that minimizes for the given constraints. There can be more than one solution for the linear program.

Every linear program in a nonstandard form can be converted into a standard form. This is done by introducing additional variables for every inequality. In case that the goal of the linear problem is to maximize the objective function than the function can be replaced with a function just by multiplying with -1, as the is maximum for is achieved in the same value for as the maximum for .

For example: is transformed to a linear equation as , for .

Example, the following program is given in a nonstandard form:

The standard form of the program is:

or:

In the general case, a nonstandard linear program:

can be transformed to a standard one like this:

- all values of and are greater than or equal to and is an identity matrix.

### Simplex algorithm

The simplex method is a method that solves a liner program in standard form.

**Example**. Develop a linear program algorithm and solve the following program:

Minimize:  
 F(x\_1,x\_2) = 7\*x\_1 + 6\*x\_2  
 Subject to:  
 2\*x\_1 + x\_2 <= 3  
 x\_1 + 4\*x\_2 <= 4  
 x\_1, x\_2 >= 0

*([8], Example 16.3 (An introduction to optimization) Page 306)*

# Example 16.3 "An introduction to optimization" Page 306:  
#  
# Maximize:  
# G(x\_1,x\_2) = 7\*x\_1 + 6\*x\_2  
# Subject to:  
# 2\*x\_1 + x\_2 <= 3  
# x\_1 + 4\*x\_2 <= 4  
# x\_1, x\_2 >= 0  
#  
import numpy as np  
  
# By convection of this text we transform the Maximize to a Minimize problem  
#  
# Minimize:  
# F(x\_1,x\_2) = - 7\*x\_1 - 6\*x\_2  
# Subject to:  
# 2\*x\_1 + x\_2 <= 3  
# x\_1 + 4\*x\_2 <= 4  
# x\_1, x\_2 >= 0  
#  
  
# First step (1) is to prepare the linear program  
## 1.1 Transform the linear program in to it's standard form  
# 2\*x\_1 + 1\*x\_2 + 1\*s\_1 + 0\*s\_2 = 3  
# 1\*x\_1 + 4\*x\_2 + 0\*s\_1 + 1\*s\_2 = 4  
  
# C contains the params of the objective function  
C = np.array([-7,-6])  
A = np.array([[2,1],[1,4]])  
B = np.array([[3],[4]])  
I = np.array([[1,0,0],[0,1,0],[0,0,1]])  
  
# Creates a simplex table from the linear expressions parameters A, I, B and F  
def create\_simplex\_table(A,B,C,I):  
 A\_C = np.vstack((A,C))  
 print(A\_C)  
 A\_C\_I = np.hstack((A\_C,I))  
 print(A\_C\_I)  
 A\_C\_I\_B = np.hstack((A\_C\_I, np.vstack((B,[0]))))  
 return A\_C\_I\_B  
  
  
# This helper function returns the row and colum of the smallest value in the last row of a matrix  
def find\_smallest\_value\_in\_the\_last\_row(table\_in):  
 # Get the last row of the table\_in  
 last\_row = table\_in[-1]  
 # Find the index of the smallest value in the last row  
 min\_value\_index = np.argmin(last\_row[:len(last\_row)-1])  
 # Return the smallest value, its row index, and column index  
 return min\_value\_index, last\_row[min\_value\_index]  
  
# Return the smallest ratio, matrix[i,lastColumn]/matrix[i,column]  
def find\_smallest\_ratio(table\_in, column\_in):  
 ratios = [] # Store the ratios  
 for i in range(len(table\_in) - 1): # Exclude the last row  
 if table\_in[i][column\_in] != 0: # Avoid division by zero  
 ratio = table\_in[i][-1] / table\_in[i][column\_in]  
 ratios.append((i, ratio)) # Store (index, ratio) tuple  
 else:  
 ratios.append((i, None)) # Set value to None if division by zero  
 min\_index, min\_value = min((pair for pair in ratios if pair[1] is not None), key=lambda x: x[1], default=(None, None))  
 return min\_index, min\_value  
  
from sympy import Matrix  
  
def custom\_pivot\_column(table\_in, pivot\_row, pivot\_col):  
 # Convert the matrix to a NumPy array for easier manipulation  
 np\_matrix = np.array(table\_in)  
 # Divide the pivot row by the pivot element to make it 1  
 pivot\_element = np\_matrix[pivot\_row, pivot\_col]  
 np\_matrix[pivot\_row, :] /= pivot\_element  
 # Eliminate other elements in the same column  
 num\_rows = np\_matrix.shape[0]  
 for i in range(num\_rows):  
 if i != pivot\_row:  
 ratio = np\_matrix[i, pivot\_col]  
 # //todo add it in python tutorial for matrixes, manipulations with row and column  
 np\_matrix[i, :] -= ratio \* np\_matrix[pivot\_row, :]  
 # Convert the modified NumPy array back to a SymPy Matrix  
 pivoted\_table = Matrix(np\_matrix)  
 return np.array(pivoted\_table)  
  
def simplex\_method\_step\_function(simplex\_matrix\_in):  
 # Step 1 simplex algorithm, find the column, i of the value that has biggest negative value,  
 # This is the steppes direction of increase for the F function  
 column, value = find\_smallest\_value\_in\_the\_last\_row(simplex\_matrix\_in)  
 # Find the row of the constraint that gives the smallest constraint for xi.  
 row, value = find\_smallest\_ratio(simplex\_matrix\_in, column)  
 return custom\_pivot\_column(Matrix(simplex\_matrix\_in),row,column)  
  
  
# The algorithm requires the data form the equations to be in a form of A\_C\_I\_B matrix, lets rename this matrix to simplex\_matrix, for context  
# simplex\_table = A\_C\_I\_B  
simplex\_table = create\_simplex\_table(A,B,C,I)  
  
print("Simplex Table--")  
print(simplex\_table)  
print("--Simplex Table")  
  
x\_approximations = []  
x\_approximations.append(simplex\_table)  
  
def stop\_conditions\_simplex\_method(approximations\_in):  
 column, min\_column\_value = find\_smallest\_value\_in\_the\_last\_row(approximations\_in[-1])  
 row, min\_row\_value = find\_smallest\_ratio(approximations\_in[-1], column)  
 return min\_column\_value < 0 and min\_row\_value != None and min\_row\_value > 0  
  
  
def basic\_simplex\_method(approximations\_in, stop\_conditions\_function\_in):  
 while(stop\_conditions\_function\_in(approximations\_in)):  
 simplex\_step\_table = simplex\_method\_step\_function(approximations\_in[-1])  
 approximations\_in.append(simplex\_step\_table)  
  
  
basic\_simplex\_method(x\_approximations, stop\_conditions\_simplex\_method)  
  
print("Minimum is achieved in the vector")  
print(x\_approximations[-1][:len(C),-1])  
print("The minimum is")  
print(x\_approximations[-1][-1][-1])

[[ 2 1]  
 [ 1 4]  
 [-7 -6]]  
[[ 2 1 1 0 0]  
 [ 1 4 0 1 0]  
 [-7 -6 0 0 1]]  
Simplex Table--  
[[ 2 1 1 0 0 3]  
 [ 1 4 0 1 0 4]  
 [-7 -6 0 0 1 0]]  
--Simplex Table  
Minimum is achieved in the vector  
[8/7 5/7]  
The minimum is  
86/7

## Two phase simplex

The two phase simplex algorithm is a generalization of the simplex algorithm, it is required since the the basic feasible solution is not always obvious, this method standardizes the simplex algorithm. Example when to use the Two phase simplex: \* Usually the basic feasible solution is for every . If this does not solve the function than it can be said that the basic feasible solution is not obvious. In this case the the two phase simplex method can be used.

The algorithm contains two phases. \* Phase 1 Is used to find a *basic feasible solution* \* Phase 2 Using the *basic feasible solution* find the optimal solution

**Canonical form of the simplex table:** Before applying the simplex method or the phase two of the “Two phase simplex method” the table needs to be in canonical form, this form needs to have 0 values for the columns of the base vectors in the last row, the row that holds the parameters of objective function. This is easily done with linear transformations.

**Example**. Develop an algorithm for the two phase simplex method and solve the linear program: Minimize : 2x\_1 + 3x\_2 Subject to : 4x\_1 + 2x\_2 >= 12 x\_1 + 4x\_2 >= 6 x1,x2 >= 0 *([8], Example 16.4, “Introduction to Optimization”, page 308)*

# Example 16.4, \*[8]\*, Introduction to Optimization, page 308  
# minimize :  
# 2x\_1 + 3x\_2  
# subject to :  
# 4x\_1 + 2x\_2 >= 12  
# x\_1 + 4x\_2 >= 6  
# x1,x2 >= 0  
  
  
# Initially, the problem is transformed in the standard form  
# minimize :  
# 2x\_1 + 3x\_2  
# subject to :  
# 4x\_1 + 2x\_2 - x\_3 - 0\*x\_4 = 12  
# x\_1 + 4x\_2 - 0\*x\_3 - x\_4 = 6  
# x1, x2, x\_3, x\_4 >= 0  
# Since x\_1, x\_2 = 0 is not a solution, it can be concluded that the initial  
# basic solution is not obvious  
  
  
# Then artificial variables, x\_5 and x\_6 >= 0 and a new  
# artificial objective function x\_5 + x\_6, that  
# needs to be maximized are introduced  
  
  
# The artificial problem will have the following form:  
# minimize:  
# x\_5+x\_6  
# subject to:  
# 4x\_1 + 2x\_2 - x\_3 - 0\*x\_4 +1x\_5 + 0x\_6 = 12  
# x\_1 + 4x\_2 - 0\*x\_3 - x\_4 + 0x\_5 +1x\_6 = 6  
# x1,x2,x\_3,x\_4,x\_5,x\_6 >= 0  
  
# Phase one solution, optimizing the artificial objective function  
  
import numpy as np  
from sympy import Matrix  
  
C = np.array([2,3])  
C\_a = np.array( [0,0,0,0,1,1] )  
A = np.array([[4,2,-1,0],  
 [1,4,0,-1]])  
B = np.array([[12],[6]])  
I\_a = np.array([[1,0],[0,1]])  
  
def create\_simplex\_table(A,B,C,I):  
 A\_I = np.hstack((A,I\_a))  
 A\_I\_C = np.vstack((A\_I,C))  
 A\_I\_C\_B = np.hstack((A\_I\_C, np.vstack((B,[0]))))  
 return A\_I\_C\_B  
  
simplex\_matrix = create\_simplex\_table(A,B,C\_a,I\_a)  
print(f"The starting table has this form: \n {simplex\_matrix}")  
  
# Before starting the last row or the coefficients of the optimization function  
# need to be adjusted so that the artificial values have 0 value  
simplex\_matrix[-1] -= np.sum(simplex\_matrix[:-1], axis=0)  
print(f"The canonical form of the table: \n {simplex\_matrix}")  
  
  
x\_approximations = []  
x\_approximations.append(simplex\_matrix)  
  
basic\_simplex\_method(x\_approximations, stop\_conditions\_simplex\_method)  
  
print(f"The phase 1 ends with the following matrix:\n {x\_approximations[-1]} \n")  
  
print("The maximum is")  
print(x\_approximations[-1][:len(C),-1])  
  
  
# Phase two  
# Remove the artificial variables  
def remove\_artificial\_variables(matrix, Ia):  
# todo, matrix[:,[-1]] is used instead of matrix[:,-1] as matrix[:,-1] will return only a row and not a matrix  
 return np.hstack((matrix[0:,0:-(len(Ia)+1)], matrix[:,[-1]]))  
  
  
def replace\_artificial\_objective\_function(table\_in, C\_in):  
 matrix = table\_in[0:-1,:]  
 zero\_values\_row = np.full(matrix.shape[1]-C\_in.shape[0], 0)  
 sum = np.hstack((C\_in, zero\_values\_row))  
 appended=np.vstack((matrix,sum))  
 return appended;  
  
  
from numpy.typing import NDArray  
  
def is\_base\_vector(array:NDArray):  
 non\_zero\_count = np.count\_nonzero(array)  
 return non\_zero\_count == 1  
  
# Expresses the objective function with the new basis, or in other words,  
# Transform the last row so that the zeros appear in the basis columns, base\_vectors  
def convert\_to\_canonical\_form(table\_in):  
 for column in range(table\_in.shape[1]):  
 if is\_base\_vector(table\_in[0:-1,column]):  
 row = np.flatnonzero(table\_in[0:-1,column])[0]  
 table\_in[row, :] /= table\_in[row][column]  
 table\_in[-1, :] -= table\_in[-1, column]\*table\_in[row][:]  
 return table\_in  
  
approximation = remove\_artificial\_variables(x\_approximations[-1], I\_a)  
print (f"Removed artificial variables from table: \n {approximation} \n")  
approximation = replace\_artificial\_objective\_function(approximation, C)  
print (f"Reverting back the original objective function: \n {approximation} \n")  
  
phase\_two\_table = convert\_to\_canonical\_form(approximation)  
  
print ("Phase two table")  
print (phase\_two\_table)  
x\_approximations.append(phase\_two\_table)  
basic\_simplex\_method(x\_approximations, stop\_conditions\_simplex\_method)  
  
print (f"The optimal solution to the linear program is: {x\_approximations[-1][:-1, -1]},")  
print (f"and the optimal cost is {-(x\_approximations[-1][-1, -1])}")

The starting table has this form:   
 [[ 4 2 -1 0 1 0 12]  
 [ 1 4 0 -1 0 1 6]  
 [ 0 0 0 0 1 1 0]]  
The canonical form of the table:   
 [[ 4 2 -1 0 1 0 12]  
 [ 1 4 0 -1 0 1 6]  
 [ -5 -6 1 1 0 0 -18]]  
The phase 1 ends with the following matrix:  
 [[1 0 -2/7 1/7 2/7 -1/7 18/7]  
 [0 1 1/14 -2/7 -1/14 2/7 6/7]  
 [0 0 0 0 1 1 0]]   
  
The maximum is  
[18/7 6/7]  
Removed artificial variables from table:   
 [[1 0 -2/7 1/7 18/7]  
 [0 1 1/14 -2/7 6/7]  
 [0 0 0 0 0]]   
  
Reverting back the original objective function:   
 [[1 0 -2/7 1/7 18/7]  
 [0 1 1/14 -2/7 6/7]  
 [2 3 0 0 0]]   
  
Phase two table  
[[1 0 -2/7 1/7 18/7]  
 [0 1 1/14 -2/7 6/7]  
 [0 0 5/14 4/7 -54/7]]  
The optimal solution to the linear program is: [18/7 6/7],  
and the optimal cost is 54/7

## Solving linear program with scipy.optimize.linprog

Python offers a convenient way for solving linear programming programs. The scipy library contains a module scipy.optimize.linprog that is designed for solving this type of problems.

Let’s demonstrate how to solve the previous linear program using the linprog library

# Example 16.4, Introduction to Optimization, page 308 [8]  
# Minimize :  
# 2x\_1 + 3x\_2  
# Subject to :  
# 4x\_1 + 2x\_2 >= 12  
# x\_1 + 4x\_2 >= 6  
# x\_1,x\_2 >= 0

1. Initially transform the linear program so it is a minimizer.
2. Create a list of all coefficients from the optimization function needs for example becomes c=[-2,-3].
3. Create matrix , this are the parameters defined by the values before the variables under the subject part of the program, but not including the last row. In this case, the parameters will form the following matrix A = [[-4, -2], [-1, -4]], it is important to point out that the sight before every parameter depends on the inequality operation or equality of the row. In case of:

* >= the sign of the parameters changes in the entire row changes
* <= the sign of the parameters stays the same in the entire row.
* == A new matrix A\_eq is created that contains this row, and will be plugged in the function separately.
* in case of < or >, manual work is required, as these rows are not excepted by the algorithm, by adding/removing a small number to the requeued row in the B-vector column.

1. Constructing the vector B, this is the last column of the linear program. In the current linear program it is the vector B=[12,6]. It has the same number of elements or rows as the matrix A. Also it follows the same rules as a concerning the inequality or equality operator.

import numpy as np  
from scipy.optimize import linprog  
  
# Example 16.4, Introduction to Optimization, page 308 [8]  
# minimize :  
# 2x\_1 + 3x\_2  
# subject to :  
# 4x\_1 + 2x\_2 <= 12  
# x\_1 + 4x\_2 <= 6  
# x1,x2 >= 0  
  
# Objective function coefficients to minimize  
c\_obj = np.array([-2, -3])  
  
# Negate for standard form (Ax <= b)  
A\_ineq = np.array([[4, 2], [1, 4]])  
b\_ineq = np.array([12, 6])  
  
# Bounds for variables x1 and x2 (non-negative)  
bounds = [(0, None), (0, None)]  
  
# Solve the linear programming problem  
# revised simplex uses the two phase simplex method descried above, but is deprecated  
# https://docs.scipy.org/doc/scipy-1.12.0/reference/optimize.linprog-revised\_simplex.html#optimize-linprog-revised-simplex  
res = linprog(c=c\_obj, A\_ub=A\_ineq, b\_ub=b\_ineq, bounds=bounds, method='revised simplex')  
  
# Print the results  
print("Optimal value:", res.fun)  
print("Optimal x values:", res.x)

Optimal value: -7.714285714285714  
Optimal x values: [2.57142857 0.85714286]  
  
  
/var/folders/4s/hhswlq455zv2br9bkfsygsfr0000gn/T/ipykernel\_74508/2743213689.py:25: DeprecationWarning: `method='revised simplex'` is deprecated and will be removed in SciPy 1.11.0. Please use one of the HiGHS solvers (e.g. `method='highs'`) in new code.  
 res = linprog(c=c\_obj, A\_ub=A\_ineq, b\_ub=b\_ineq, bounds=bounds, method='revised simplex')

# Bibliography

## Python Tutorial

1. W3Schools. “Plot Functions.” <https://www.w3schools.com/python/matplotlib_pie_charts.asp>.
2. Matplotlib. “Animations.” <https://matplotlib.org/stable/gallery/animation/index.html>.
3. W3Schools. “Language References.” <https://www.w3schools.com/python/>.
4. Turing. “Using sympy with Symbols.” <https://www.turing.com/kb/derivative-functions-in-python>.

### Linear Algebra

1. Wikipedia. “Dot-product.” <https://en.wikipedia.org/wiki/Dot_product#Definition>.
2. Wikipedia. “Cross Product.” <https://en.wikipedia.org/wiki/Cross_product>.
3. Hefferon, Jim. *LINEAR ALGEBRA, Fourth Edition.* <https://www.cs.ox.ac.uk/files/12921/book.pdf>.

## Optimization Methods

1. Chong, Edwin K. P., and Stanislaw H. Źak. *An Introduction to Optimization, Second Edition.* Wiley-Interscience, 2001.

## Simplex Method

1. Chong, Edwin K. P., and Stanislaw H. Źak. *An Introduction to Optimization, Second Edition.* Wiley-Interscience, 2001.
2. Schmidt, Jack. “Simplex Method: Lab Manual.” University of Kentucky, College of Arts and Sciences, Department of Mathematics. <https://www.ms.uky.edu/~jack/2010-01-MA162/2010-02-25-MA162Lab.pdf>.
3. Goemans, Michel. “Linear Programming Notes.” Massachusetts Institute of Technology, Department of Mathematics. <https://math.mit.edu/~goemans/18310S15/lpnotes310.pdf>.
4. Magnanti, Tom, and Jim Orlin. “Optimization Methods in Business Analytics: Solving Linear Programs.” <https://web.mit.edu/15.053/www/AMP-Chapter-02.pdf>.
5. Stojiljković, Mirko. “Hands-On Linear Programming: Optimization with Python.” Real Python. <https://realpython.com/linear-programming-python/>.
6. Martinich, Joseph S. “Linear Programming.” University of Kentucky. <https://www.uky.edu/~dsianita/300/online/LP.pdf>.
7. S, Tom. “Simplex Method Explained.” YouTube. <https://www.youtube.com/watch?v=E72DWgKP_1Y&t=652s>.

# Libraries

## Python

* SymPy, https://www.sympy.org/en/index.html
* Scipy, optimize https://docs.scipy.org/doc/scipy/reference/optimize.html
* matplotlib, https://matplotlib.org
* numpy, https://numpy.org/

## .md

* Latex, https://www.latex-project.org/