PYTHON PROGRAMMING AND SCIENTIFIC COMPUTATION

Tran Nam Trung

Institute of Mathematics, VAST, Hanoi - Vietnam

Email: tntrung@math.ac.vn

Contents

Preface

This document introduces the Python language from the basis. Our aim is to implement basic numerical problems and data analysis in Python such as: solving equations, optimization, graph theory. At the end of the book we sketch simple ideas and programs on Machine learning.

Chapter 1

Python

Source: https://realpython.com/ and https://www.w3schools.com/python/

1.1 Introduction

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

1. web development (server-side),
2. software development,
3. mathematics,
4. system scripting.

What can Python do?

1. Python can be used on a server to create web applications.
2. Python can be used alongside software to create workflows.
3. Python can connect to database systems. It can also read and modify files.
4. Python can be used to handle big data and perform complex mathematics.
5. Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

1. Python works on different platforms (Windows, Mac, Linux, etc).
2. Python has a simple syntax similar to the English language.
3. Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
4. Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
5. Python can be treated in a procedural way, an object-orientated way or a functional way.

Good to know

1. The most recent major version of Python is Python 3, which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
2. In this tutorial Python will be written in Jupyter Notebook. It is possible to write Python in an Integrated Development Environment, such as Pycharm, VS Code, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

Python Syntax compared to other programming languages

1. Python was designed for readability, and has some similarities to the English language with influence from mathematics.
2. Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
3. Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

1.2 Python Getting Started

1.2.1 Python Install

This this tutor we install Python 3.7 version from Anaconda; and illustrate by Jupyter Notebook.

Installing Anaconda on Windows

1. Visit Anaconda.com/downloads
2. Select Windows
3. Download the .exe installer
4. Open and run the .exe installer

Use Jupyter Notebook

1. Open **Anaconda navigator**
2. Click **Launch** in the item "Jupyther notebook"

1.2.2 Jupyter Notebooks

Jupyter Notebooks are a powerful way to write and iterate on your Python code for data analysis. Rather than writing and re-writing an entire program, you can write lines of code and run them one at a time. Then, if you need to make a change, you can go back and make your edit and rerun the program again, all in the same window.

1.2.3 Python Syntax

Execute Python Syntax. Python syntax can be executed by writing directly in the Command Line.

▷ The first program: print "Hello, World!"

```
# The first program
print("Hello,World!")
# Output: Hello, World!
```

Python Indentation.

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.

▷ Python Indentation

```
if 5 > 2:
    print("Hello,World!")
```

▷ Python will give you an error if you skip the indentation:

```
if 5 > 2:
print("Five is greater than two!")
```

▷ The number of spaces is up to you as a programmer, but it has to be at least one.

```
if 5 > 2:
 print("Five is greater than two!")
if 5 > 2:
        print("Five is greater than two!")
```

▷ You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

```
if 5 > 2:
    print("Five is greater than two!")
        print("Five is greater than two!")
```

Python Variables

In Python, variables are created when you assign a value to it. Python has no command for declaring a variable.

```
x = 5
y = "Hello, World!"
```

Comments

Python has commenting capability for the purpose of in-code documentation. Comments start with a #, and Python will render the rest of the line as a comment:

▷ Comments in Python:

```
# This is a comment.
print("Hello, World!")
```

Creating Variables

1. Variables are containers for storing data values.
2. Unlike other programming languages, Python has no command for declaring a variable.
3. A variable is created the moment you first assign a value to it.

```
x = 5
y = "John"
print(x)
print(y)
```

Variables do not need to be declared with any particular type and can even change type after they have been set.

```
x = 5 # x is of type int
x = "John" # x is now of type str
print(x)
```

String variables can be declared either by using single or double quotes:

```
x = "John"
#  is the same as
x = 'John'
```

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

1. A variable name must start with a letter or the underscore character.
2. A variable name cannot start with a number.
3. A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
4. Variable names are case-sensitive (age, Age and AGE are three different variables).

Assign Value to Multiple Variables

▷ Python allows you to assign values to multiple variables in one line:

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

▷ And you can assign the same value to multiple variables in one line:

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Listing 1.1: Example

Output Variables

▷ The Python **print** statement is often used to output variables. To combine both text and a variable, Python uses the + character:

```
x = "owesome"
print("Python is "+x)
```

▷ You can also use the + character to add a variable to another variable:

```
x = "Python is "
y = "owesome"
z = x + y
print(z)
```

▷ For numbers, the + character works as a mathematical operator:

```
x = 5
y = 10
print(x+y)
```

▷ If you try to combine a string and a number, Python will give you an error:

```
x = 5
y = "John"
print(x+y)
```

Global Variables

1. Variables that are created outside of a function (as in all of the examples above) are known as global variables.

2. Global variables can be used by everyone, both inside of functions and outside.

```
x = "awesome"
def myfunc():
    print("Python is "+x)

myfunc()
```

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

```
x = "awesome"
def myfunc():
    x = "fantastic"
```

```
    print("Python is "+x)

myfunc()
print("Python is "+x)
```

The global Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.

▷ To create a global variable inside a function, you can use the **global** keyword.

```
x = "awesome"
def myfunc():
    global x
    x = "fantastic"

myfunc()
print("Python is "+x)
```

1.3 Python Data Types

- In programming, data type is an important concept.
- Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

1. Text Type: **str**
2. Numeric Types: **int, float, complex**
3. Sequence Types: **list, tuple, range**
4. Mapping Type: **dict**
5. Set Types: **set, frozenset**
6. Boolean Type: **bool**
7. Binary Types: **bytes, bytearray, memoryview**

Getting the Data Type

▷ You can get the data type of any object by using the **type()** function.

```
x = 5
print(type(x))
```

Setting the Data Type

In Python, the data type is set when you assign a value to a variable.

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = True	bool

Python Numbers

▷ There are three numeric types in Python:

- int
- float
- complex

▷ Variables of numeric types are created when you assign a value to them.

```
x = 10 # int
y = 2.8 # float
z = 1j # complex
```

▷ To verify the type of any object in Python, use the **type()** function:

```
print(type(x))
print(type(y))
print(type(z))
```

▷ **int**, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

```
x = 1
y = 35656222554887711
z = -3255522
```

▷ **float**, or *"floating point number"* is a number, positive or negative, containing one or more decimals.

```
x = 1.10
y = 1.0
z = -35.36
```

▷ Float can also be scientific numbers with an **"e"** to indicate the power of **10**.

```
x = 35e3
y = 12E4
z = -87.7e100
```

▷ **Complex** numbers are written with a **"j"** as the imaginary part:

```
x = 3+5j
y = 5j
z = -5j
```

Type Conversion

▷ You can convert from one type to another with the **int()**, **float()**, and **complex()** methods.

```
x = 1 # int
y = 2.8 # float
z = 1j # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
```

```
print(type(c))
```

Random Number

Python does not have a **random()** function to make a random number, but Python has a built-in module called **random** that can be used to make random numbers:

▷ Import the random module, and display a random number between 1 and 9:

```
import random

print(random.randrange(1,10))
```

Python Casting

Specify a Variable Type: There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

- **int()** - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
- **float()** - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- **str()** - constructs a string from a wide variety of data types, including strings, integer literals and float literals

```
# Integers
x = int(1)   # x will be 1
x = int(2.8) # x will be 2
x = int("3") # x will be 3

# Floats
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2

# Strings
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

Python Strings

String Literals: String literals in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as **"hello"**.

▷ You can display a string literal with the **print()** function:

```
print("Hello")

print('Hello')
```

Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string.

Multiline Strings

▷ You can assign a multiline string to a variable by using three quotes (double quotes or single quotes):

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""

print(a)
```

▷ Three single quotes:

```
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''

print(a)
```

1.3.1 Python Booleans

∗ Booleans represent one of two values: **True** or **False**.

1.3.2 Python Operators

∗ Operators are used to perform operations on variables and values.

∗ Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	$x + y$
-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	x/y
%	Modulus	$x\%y$
//	Floor division	$x//y$

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	$x = 5$	$x = 5$
+=	x += 5	$x = x + 5$
+=	x -= 5	$x = x - 5$
*=	x *= 5	$x = x * 5$
/=	x /= 5	$x = x/5$
%=	x %= 5	$x = x\%5$
//=	x //= 5	$x = x//5$
**=	x **= 5	$x = x * *5$

Python Comparison Operators

Comparison operators are used to compare two values:

Operator	Example	Same As
==	Equal	$x == y$
!=	Not equal	$x != y$
>	Greater than	$x > y$
<	Less than	$x < y$
>=	Greater than or equal to	$x >= y$
<=	Less than or equal to	$x <= y$

Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns **True** if both statements are true	$x < 5$ and $y < 10$
or	Returns **True** if one of the statements is true	$x < 5$ or $y < 10$
not	Reverse the result, returns **False** if the result is true	not ($x < 5$ and $y < 10$)

Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns **True** if both variables are the same object	x is y
not is	Returns **True** if both variables are not the same object	x not is y

Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns **True** if a sequence with the specified value is present in the object	x in y
not in	Returns **True** if a sequence with the specified value is not present in the object	x not in y

1.3.3 Python Lists

Python Collections (Arrays): There are four collection data types in the Python programming language.

1. **List** is a collection which is ordered and changeable. Allows duplicate members.
2. **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
3. **Set** is a collection which is unordered and unindexed. No duplicate members.
4. **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

List: A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

▷ Create a List:

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

Access Items: You access the list items by referring to the index number. Remember that the first item has index 0.

▷ Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

Negative index: Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.

▷ Print the last item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

Range of Indexes: You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new list with the specified items.

▷ Return the third, fourth, and fifth item:

```
thislist = ["apple", "banana", "cherry",
            "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

Remark 1. The search will start at index 2 (included) and end at index 5 (not included).

By leaving out the start value, the range will start at the first item:

▷ This example returns the items from the beginning to "orange":

```
thislist = ["apple", "banana", "cherry",
            "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
```

▷ By leaving out the end value, the range will go on to the end of the list:

Example. This example returns the items from "cherry" and to the end:

```
thislist = ["apple", "banana", "cherry",
            "orange", "kiwi", "melon", "mango"]
print(thislist[:2])
```

Range of Negative Indexes

▷ Specify negative indexes if you want to start the search from the end of the list:

Example. This example returns the items from index -4 (included) to index -1 (excluded):

```
thislist = ["apple", "banana", "cherry",
            "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])
```

Change Item Value

To change the value of a specific item, refer to the index number:

▷ Change the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "cherry"
print(thislist)
```

Loop Through a List

You can loop through the list items by using a **for** loop:

▷ Print all items in the list, one by one:

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
    print(x)
```

Check if Item Exists

To determine if a specified item is present in a list use the **in** keyword:

▷ Check if "apple" is present in the list:

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
    print("Yes, 'apple' is in the fruits list")
```

List Length

To determine how many items a list has, use the **len()** function:

▷ Check if "apple" is present in the list:

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

Some operators on Lists

- Add item:
  1. Using the append() method to append an item: thislist.append("orange").
  2. To add an item at the specified index, use the insert() method: thislist.insert(1, "orange") will insert "orange" at position 1 in thislist.
- Remove Item:
  1. The remove() method removes the specified item: thislist.remove("banana").
  2. The pop() method removes the specified index, (or the last item if index is not specified): thislist.pop().
  3. The del keyword removes the specified index: del thislist[0].
  4. The clear() method empties the list: thislist.clear().

Copy lists

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

There are ways to make a copy:

- Make a copy of a list with the copy() method: list2 = list1.copy().
- Another way to make a copy is to use the built-in method list(): list2 = list(list1).

Join Two Lists

There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the + operator: list3 = list1 + list2.

Or you can use the extend() method, which purpose is to add elements from one list to another list. For example, use the extend() method to add list2 at the end of list1: list1.extend(list2).

1.3.4 Python Tuples

Definition 2. A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.

▷ Create a tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
# Output: ('apple', 'banana', 'cherry')
```

Access Tuple Items : You can access tuple items by referring to the index number, inside square brackets.

▷ Print the second item in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
# Output: banana
```

Remark 3. Everything you have learned about lists - they are ordered, they can contain arbitrary objects, they can be indexed and sliced, they can be nested - is true of tuples as well. But they cannot be modified.

Unpacking tuples

When creating a tuple we pack some objects into a single object. Unpacking a tuple, we extract those values into a single variable.

```
# Create a tuple
t = ('foo', 'bar', 'baz', 'qux')

# Unpack a tuple
(s1, s2, s3, s4) = t  # s1 = 'foo',s2 = 'bar',s3 = 'baz',s4 = 'qux'

print(s1) # Output: foo
```

Remark 4. In unpacking of tuple, number of variables on left hand side should be equal to number of values in given tuple.

1.3.5 Python Dictionaries

A **dictionary** is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

▷ Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Accessing Items: You can access the items of a dictionary by referring to its key name, inside square brackets. Get the value of the "model" key: x = thisdict["model"].

Change Values: You can change the value of a specific item by referring to its key name. Change the "year" to 2018: thisdict["year"] = 2018.

Loop Through a Dictionary: You can loop through a dictionary by using a for loop. When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

▷ Print all key names in the dictionary, one by one:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
   print(thisdict[x])
```

▷ You can also use the values() function to return values of a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict.values():
   print(x)
```

Check if Key Exists

To determine if a specified key is present in a dictionary use the in keyword: x in thisdict.

Dictionary Length

To determine how many items (key-value pairs) a dictionary has, use the len() method: len(thisdict).

Adding Items

▷ Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
```

Removing Items

There are several methods to remove items from a dictionary:

- The pop() method removes the item with the specified key name: thisdict.pop("model").
- The del keyword removes the item with the specified key name: del thisdict["model"].
- The clear() keyword empties the dictionary: thisdict.clear().

Copy a Dictionary

You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.

Here are ways to make a copy:

- One way is to use the built-in Dictionary method copy(): dict2 = dict1.copy().
- Another way to make a copy is to use the built-in method dict(): dict2 = dict(dict1).

Nested Dictionaries

A dictionary can also contain many dictionaries, this is called nested dictionaries.

▷ Create a dictionary that contain three dictionaries:

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
```

```
    "name" : "Linus",
    "year" : 2011
  }
}
```

▷ Or, if you want to nest three dictionaries that already exists as dictionaries:

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

The dict() Constructor

It is also possible to use the dict() constructor to make a new dictionary:

```
thisdict = dict(brand="Ford", model="Mustang", year=1964)
```

1.3.6 Python Sets

Source: `https://docs.python.org/2/library/sets.html`

Definition 5. A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

▷ Create a set:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
# Output: {'banana', 'cherry', 'apple'}
```

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

The set() Constructor: It is also possible to use the set() constructor to make a set.

```
thisset = set(("apple", "banana", "cherry")) # note the double round-
    brackets
print(thisset) # Output: {'banana', 'cherry', 'apple'}
```

Empty set: Empty set is set() not {}, because the latter is an empty dictionary.

Access Items: You cannot access items in a set by referring to an index, since sets are unordered the items has no index.

But you can loop through the set items using a for loop, or ask if a specified value is present in a set.

```
for x in thisset:
  print(x)
# Output:
  banana
  cherry
  apple
```

Check if an item is present in the set by using in:

```
print("banana" in thisset)
# Output: True

print("orange" in thisset)
# Output: False
```

Change Items: Once a set is created, you cannot change its items, but you can add new items.

Add Items: To add one item to a set use the add() method. To add more than one item to a set use the update() method.

```
thisset.add("orange")
print(thisset)
# Output: {'orange', 'banana', 'cherry', 'apple'}

thisset = {"apple", "banana", "cherry"}
thisset.update(["orange", "mango", "grapes"])
print(thisset)
# Output: {'apple', 'grapes', 'mango', 'banana', 'orange', 'cherry'}
```

Get the Length of a Set: To determine how many items a set has, use the len() method.

Remove Item: To remove an item in a set, use the remove(), or the discard() method.

```
thisset = {'orange', 'banana', 'cherry', 'apple'}
thisset.remove("cherry")
# Output: {'orange', 'banana', 'apple'}

thisset.discard("orange")
# Output: {'banana', 'apple'}
```

Note: If the item to remove does not exist, remove() will raise an error, but discard() will not raise an error.

You can also use the pop(), method to remove an item, but this method will remove the last item. Remember that sets are unordered, so you will not know what item that gets removed.

```
thisset = {"apple", "banana", "cherry"}

x = thisset.pop()
print(x)    # Output: banana

print(thisset)    # Output: {'cherry', 'apple'}
```

Note: Sets are unordered, so when using the pop() method, you will not know which item that gets removed.

Delete: Some method for deleting on sets:

1. The remove() method removes the specified element from the set..
2. The clear() method empties the set.
3. The del keyword will delete the set completely.

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("cherry")
print(thisset)  # Output: {'banana', 'apple'}

thisset.clear()
print(thisset)  # Output: set()

thisset = {"apple", "banana", "cherry"}
```

```
del thisset
print(thisset)  # NameError: name 'thisset' is not defined
```

Operations on sets: Let $S1$ and $S2$ be two sets.

Operator	Syntax
union	$S1 \mid S2$
intersection	$S1$ & $S2$
difference	$S1 - S2$
is disjoint	$S1$.isdisjoint($S2$)
is subset	$S1 <= S2$, or $S1$.issubset($S2$)
is proper subset	$S1 < S2$

1.4 Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: a == b.
- Not equals: a != b.
- Less than: a < b.
- Less than or equal to: a <= b.
- Greater than: a > b.
- Greater than or equal to: a >= b.

These conditions can be used in several ways, most commonly in **if statements** and **loops**.

An "if statement" is written by using the if keyword.

▷ If statement:

```
a = 33
b = 200
if a > b:
    print("b is greater than a")
```

Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

▷ If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a")
```

Elif

▷ The elif keyword is pythons way of saying *"if the previous conditions were not true, then try this condition"*:

```
a = 33
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal" )
```

Else

▷ The else keyword catches anything which isn't caught by the preceding conditions:

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal" )
else:
    print("a is greater than b")
```

▷ You can also have an else without the elif:

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

▷ One line if statement:

```
a = 200
b = 33
if a > b: print("a is greater than b")
```

The pass Statement

if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

▷ **pass** statement:

```
a = 200
b = 33
if b > a:
    pass
```

Remark 6. We can use the keywords and, or to combine conditional statements.

1.5 Python loops

Python has two primitive loop commands:

- while loops
- for loops

1.5.1 The while loop

With the while loop we can execute a set of statements as long as a condition is true.

▷ Print i as long as i is less than 6:

```
i = 0
while i < 6:
    print(i)
    i += 1
```

Remark 7. ▷ Remember to increment i, or else the loop will continue forever.

▷ The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 0.

The break Statement

With the break statement we can stop the loop even if the while condition is true:

▷ Exit the loop when i is 3:

```
i = 0
while i < 6:
    print(i)
    if i == 3: break
    i += 1
```

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

▷ Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
    i += 1
    if i == 3: continue
    print(i)
```

The else Statement

With the else statement we can run a block of code once when the condition no longer is true:

▷ Print a message once the condition is false:

```
i = 0
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

1.5.2 For Loops

▷ A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

▷ This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

▷ With the for loop we can execute a set of statements, once for each item in a list, tuple, set, etc.

▷ Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

Remark 8. The for loop does not require an indexing variable to set beforehand.

The range() function

▷ To loop through a set of code a specified number of times, we can use the range() function.

▷ The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

▷ Using the range() function:

```
for x range(6):
   print(x)   # the result is the sequence: 0, 1, 2, 3, 4, 5
```

Remark 9. ▷ range(6) is not the values of 0 to 6, but the values 0 to 5.

▷ The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6).

▷ The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a *third parameter*: range(2, 30, 3).

▷ Using the start parameter:

```
for x range(2,6):
    print(x)   # the result: 2, 3, 4, 5
```

▷ Increment the sequence with 3 (default is 1):

```
for x range(2,18,3):
    print(x)   # the result: 2, 5, 8, 11, 14, 17
```

Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished.

▷ Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x range(6):
    print(x)
else:
    print("Finally finished!")
```

The pass Statement

for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

▷ Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in [0,2,4,6]:
    pass
```

1.5.3 Stack and Queue in Python

Source: `https://docs.python.org/2/library/queue.html`

A **stack** is a container of objects that are inserted and removed according to the last-in first-out (*LIFO*) principle. A **queue** is a container of objects that are inserted and removed according to the first-in first-out (*FIFO*) principle.

In Python, the module **queue** includes stack and queue. Moreover, it also has *priority queue*.

▷ Create FIFO Queue and LIFO queue

```
# Import Queue
import queue
# create a queue
Q = queue.Queue()
#create a stack
S = queue.LifoQueue()
```

Both stack and queue have the following functions available in the module queue:

1. empty() - Return True if the queue is empty, False otherwise.
2. get() - Remove and return *an item* from the queue. If queue is *empty*, wait until an item is available.
3. put(item) - Put an *item* into the queue.
4. qsize() - Return the number of items in the queue.

▷ Using FIFO And LIFO Queues

```
import queue

# using queue
Q = queue.Queue()
for i in range(5):
    Q.put(i)
while not Q.empty():
    print(Q.get())
# Output: 0 1 2 3 4

# using stack
S = queue.LifoQueue()
for i in range(5):
    S.put(i)
```

```
while not S.empty():
    print(S.get())
# Output: 4 3 2 1 0
```

1.6 Python Functions

1.6.1 Functions

▷ A function is a block of code which only runs when it is called.

▷ You can pass data, known as parameters, into a function.

▷ A function can return data as a result.

Creating a Function

In Python a function is defined using the **def** keyword.

▷ Example:

```
def my_function():
    print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

▷ Example.

```
my_function()
```

Arguments

▷ Information can be passed into functions as arguments.

▷ Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

▷ The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_func(fname):
    print(fname + " Refsnes")

my_func("Emil")
my_func("Tobias")
my_func("Linus")
```

Remark 10. The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

▷ A **parameter** is the variable listed inside the parentheses in the function definition.

▷ An **argument** is the value that are sent to the function when it is called.

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

▷ This function expects 2 arguments, and gets 2 arguments:

```
def my_func(fname, lname):
    print(fname + " " + lname)

my_func("Emil", "Refsnes")
```

Remark 11. If you try to call the function above with 1 or 3 arguments, you will get an error.

▷ This function expects 2 arguments, but gets only 1 (get an error):

```
def my_func(fname, lname):
    print(fname + " " + lname)

my_func("Emil")
```

Arbitrary Arguments *args

▷ If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

▷ This way the function will receive a tuple of arguments, and can access the items accordingly.

▷ If the number of arguments is unknown, add a * before the parameter name:

```
def my_func(*kid):
    print("The youngest child is " + kids[2])

my_func("Emil", "Tobias", "Linus")
# result: The youngest child is Linus
```

Default Parameter Value

▷ The following example shows how to use a default parameter value.

▷ If we call the function without argument, it uses the default value.

Remark 12. You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

Return Values

▷ To let a function return a value, use the **return** statement:

```
def my_func(x):
    return 5 * x

print(my_func(2))
# result: 10
```

The pass Statement

The function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

▷ Example:

```
def my_function():
    pass
```

1.6.2 Recursion

▷ Python also accepts function recursion, which means a defined function can call itself.

▷ Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

▷ The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

▷ For example, we write functions to calculate the following examples:

1. The function $\mathsf{tri\_recursion(k)} = 1+2+\cdots+\mathsf{k}$.
2. The function $\mathsf{factorial(k)} = \mathsf{k!}$.
3. The function $\mathsf{fib(k)}$ defined by:
$$\mathsf{fib(0)} = 1,\ \mathsf{fib(1)}=1,\ \mathsf{fib(k)} = \mathsf{fib(k\text{-}1)}+\mathsf{fib(k\text{-}2)}\ \text{for}\ \mathsf{k} \geqslant 2.$$

▷ Implement in Python:

```
def tri_function(k):
    if k == 0:
        return 0
    else:
        return (k+tri_function(k-1))

def factorial(k):
    if k == 0:
       return 1
    else:
        return (k * factorial(k-1))

def fib(k):
    if k == 0:
        return 1
    if k == 1:
        return 1
    return (fib(k-1)+fib(k-2))
```

1.6.3 Python Lambda

▷ A lambda function is a small anonymous function.

▷ A lambda function can take any number of arguments, but can only have one expression.

Syntax: lambda *arguments : expression*

▷ The expression is executed and the result is returned:

Example. A lambda function that adds 10 to the number passed in as an argument, and print the result:

```
x = lambda a : a + 10

print(x(5)) # Output: 10
```

▷ Lambda functions can take any number of arguments.

Example. A lambda function that multiplies argument a with argument b and print the result:

```
x = lambda a, b : a * b

print(x(5,6)) # Output: 30
```

Why Use Lambda Functions?

▷ The power of lambda is better shown when you use them as an anonymous function inside another function.

▷ Use lambda functions when an anonymous function is required for a short period of time.

1.7 Object Oriented Programming

1.7.1 Procedural Programming

A programming paradigm is procedural programming which structures a program like a recipe in that it provides a set of steps, in the form of procedures and code blocks, which flow sequentially in order to complete a task.

This approach breaks down a programming task into a collection of **data** and **procedures**. Data consists of *variables* and *data structures*. Procedures, also known as routines, subroutines, or functions, simply contain a series of computational steps to be carried out. Data and procedures are treated as separate entities.

1.7.2 Object-Oriented Programming (OOP)

Object-oriented programming is an approach to solve a programming problem by creating *objects*. An object has two characteristics:

- **Data fields** (often known as *attributes* or *properties*).
- **Methods** (or *behaviors*) that are procedures which can access and often modify the data fields of the object with which they are associated.

In Python, an object is an instance of a class, and a class is a data type consists of data fields and methods.

Class: *A class is a blueprint for the object.*

Object: An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

▷ Below is a simple Python program that creates a class with single method:

```
# A simple example class
class Test:
    # A sample method
    def fun(self):
```

```
        print("Hello")

# Create an object
obj = Test()

# Invoke a method
obj.fun() # Output: Hello
```

The **self**:

1. Class methods must have an extra first parameter in method definition. We do not give a value for this parameter when we call the method, Python provides it.

2. If we have a method which takes no arguments, then we still have to have one argument - the self. See fun() in above simple example.

3. This is similar to this pointer in **C++** and this reference in **Java**.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) - this is all the special self is about.

The __init__ method:

The __init__ method is similar to constructors in *C++* and *Java*. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

```
# A sample class with init method
class Person:
    # init method or constructor
    def __init__(self, name):
        self.name = name
    #Sample method
    def say_hi(self):
        print("Hello, my name is', self.name)

p = Person("Micheal")
p.say_hi()

# Output: Hello, my name is Micheal
```

Class and Instance Variables (Or attributes)

1. In Python, instance variables are variables whose value is assigned inside a constructor or method with self.

2. Class variables are variables whose value is assigned in class.

```
class CSStudent:
    stream = 'cse'

    def __init__(self, roll):
        self.rol = roll

a = CSStudent(101)
b = CSStudent(102)

print(a.stream)  # prints "cse"
print(b.stream)  # prints "cse"
print(a.rol)  # prints 101
print(CSStudent.stream)  # prints "cse"
```

▷ We can define instance variables inside normal methods also:

```
class  CSStudent:
    stream = 'cse'
    def __init__(self,roll):
         self.rol = roll
    def  setAddress(self, address):
         self.address = address  #  Adds an instance variable
    def getAddress(self):
         return self.address  # Retrieve instance variable

a = CSStudent(101)
a.setAddress("Hanoi, Vietnam")

print(a.getAddress())  # prints "Hanoi, Vietnam"
```

How to create an **empty class**?

▷ We can create an empty class using pass statement in Python:

```
class Test:
    pass:
```

In Python, the concept of OOP follows some basic principles:

- Inheritance: A process of using details from a new class without modifying existing class.

- Encapsulation: Hiding the private details of a class from other objects.
- Polymorphism: A concept of using common operation in different ways for different data input.

Inheritance

Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

▷ Use of **Inheritance** in Python:

```
# Parent class
class Bird:
    def __Init__(self):
        print("Bird is ready")
    def whoisThat(self):
        print("Bird")
    def swim(self):
        print("Swim faster")

# Child class
class Penguin(Bird):
    def __Init__(self):
        # Call super() function
        super().__Init__()
        print("Peguin is ready")
    def whoisThat(self):
        print("Penguin")
    def run(self):
        print("Run faster")

peggy =  Penguin()
peggy.whoisThat()
peggy.swin()
peggy.run()
```

▷ When we run this program, the output will be:

```
Bird is ready
Penguin is ready
Penguin
Swin faster
Run faster
```

Remark 13. In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from swim() method. Again, the child class modified the behavior of parent class. We can see this from whoisThis() method. Furthermore, we extend the functions of parent class, by creating a new run() method.

Additionally, we use super() function before __init__() method. This is because we want to pull the content of __init__() method from the parent class into the child class.

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single "_" or double "__".

▷ Data Encapsulation in Python:

```
class Computer:
    def __init__(self):
        self.__maxprice = 900
    def sell(self):
        print("Sell price: {}".format(self.__maxprice))
    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell() # maxprice is still 900

# using setter function
c.setMaxPrice(1000)
c.sell() # maxprice now is 1000
```

Remark 14. In the above program, we defined a class Computer. We use __init__() method to store the maximum selling price of computer. We tried to modify the price. However, we can't change it because Python treats the __maxprice as private attributes. To change the value, we used a setter function i.e. setMaxPrice() which takes price as parameter.

Polymorphism

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

▷ Using Polymorphism in Python:

```
class Perrot:
    def fly(self):
        print("Parrot can fly")
    def swin(self):
        print("Parrot can't swim")

class Penguin:
    def fly(self):
        print("Penguin can't fly")
    def swim(self):
        print("Penguin can swim")

# common interface
def flying_test(bird):
    bird.fly()

# instantiate objects
blu = Perrot()
peggy = Penguin()

# passing the objects
flying_test(blu)  # result: Parrot can fly
flying_test(peggy)  # result: Penguin can't fly
```

Remark 15. In the above program, we defined two classes Parrot and Penguin. Each of them have common method fly() method. However, their functions are different. To allow polymorphism, we created common interface i.e. flying_test() function that can take any object. Then, we passed the objects blu and peggy in the flying_test() function, it ran effectively.

Key Points to Remember:

1. The programming gets easy and efficient.

2. The class is sharable, so codes can be reused.

3. The productivity of programmars increases.

4. Data is safe and secure with data abstraction.

Chapter 2

Data Visualization

In this chapter we will introduce the following modules:

- Numpy: to handle vectors and matrices.
- Matplotlib: to draw graphs and scatters.
- Pandas: to manage the dataframes.
- Networkx: to implement finite graphs.

2.1 Numpy: arrays and matrices

Source: https://numpy.org/devdocs/user/quickstart.html

NumPy is an extension to the Python programming language, adding support for large, multidimensional (numerical) arrays and matrices, along with a large library of high-level mathe- matical functions to operate on these arrays. In the following we use NumPy to analyze vectors and matrices.

▷ First we need to import NumPy:

```
1 import numpy as np
```

2.1.1 Create arrays

▷ Create ndarrays from lists. note: every element must be the same type (will be converted if possible):

```
data1 = [1, 2, 3, 4, 5]              # list
arr1 = np.array(data1)               # 1d array
data2 = [range(1, 5), range(5, 9)]   # list of lists
arr2 = np.array(data2)               # 2d array
arr2.tolist()                        # convert array back to list
```

▷ create special arrays:

```
np.zeros(10)                    # a zero-vector with 10 elements
np.zeros((3, 6))                # a zero-matrix with size 3 x 6
np.ones(10)                     # a vector with 10 elements 1
np.linspace(0, 1, 5)            # 0 to 1 (inclusive) with 5 points
```

2.1.2 Examining arrays

▷ Some information of arrays:

```
arr1.dtype     # type of arr1: float64
arr2.dtype     # int32
arr2.ndim      # 2 : the number of rows
arr2.shape     # (2,4) : axis 0 is rows, axis 1 is columns
arr2.size      # 8 - total number of elements
len(arr2)      # 2 - size of first dimension (aka axis)
```

2.1.3 Reshaping

It is not always possible to change the shape of an array **without copying the data**. If you want an error to be raised when the data is copied, you should assign the new shape to the shape attribute of the array:

```
# create a 10D vector a = [1,2,3,4,5,6,7,8,9,10]
a = np.array([1,2,3,4,5,6,7,8,9,10])

# create b = reshape a to be a 2x5 matrix
b = a.reshape((2,5))
# Output: array([[1,2,3,4,5],[6,7,8,9,10]])

# create c = reshape b to be a 5x2 matrix
c = b.reshape((5,2))
# Output: array([[1,2],[3,4],[5,6],[7,8],[9,10]])

# change c to be a vector
d = c.reshape(10)
```

```
# Output d = a

# Other reshape: the unspecified value (the row) is inferred to be 2
e = a.reshape((-1,5))
# Output: e = b
```

2.1.4 Stack arrays

▷ Stack flat arrays in rows:

```
u = np.array([1,2,3,4])
v = [5,6,7,8,9]
w = [-3,-2,-1,0]

# Stack arrays
f = np.stack((u,v,w))

# Output: array([[1,2,3,4],[5,6,7,8,9],[-3,-2,-1,0]])
```

2.1.5 Selection

Single item

```
arr = np.arange(10, dtype=float).reshape((2, 5)))

arr[0]    # the first row: array([0., 1.,2.,3., 4.])
arr[0,2]  # row 0, column 2 (indexed from 0): returns 2
arr[0][2] # alternative syntax
```

Slicing

▷ Syntax: *start:stop:step* with start (default 0) stop (default last) step (default 1)

```
arr[0, :] # row 0: return 1D array([0., 1.,2.,3.,4.])
arr[:, 0] # column 0: return 1D array([0.,5.])
arr[:, :2]  # columns strictly before index 2 (2 first rows)
arr[:, 2:]  # columns after index 2 included
arr2 = arr[:, 1:4]  # columns between index 1 (included) and 4 (
    excluded)

print(arr2) # Output: array([[1.,2.,3.],[6.,7.,8.]])
```

Remark. Slicing returns a view (not a copy).

```
arr2[0,0] = 33
print(arr2)
print(arr)
```

2.1.6 Vectorized operations

▷ All NumPy operations are vectorized, where you apply operations to the whole array instead of on each element individually:

```
nums = np.arange(5) # nums = [0,1,2,3,4]
nums * 10 # multiply each element by 10
nums = np.sqrt(nums)  # square root of each element
np.ceil(nums) # also floor, rint (round to nearest int)
np.isnan(nums) # checks for NaN
nums + np.arange(5) # # add element-wise
np.maximum(nums, np.array([1, -2, 3, -4, 5]))  # compare element-wise

# random numbers
np.random.seed(12234) # Set the seed
np.random.rand(2,3) # 2x3 matrix in [0,1]
np.random.randn(10) # random  normals (mean 0, and std 1)
np.random.randint(0,2,10) # 10 randomly picked 0 or 1


# Compute Euclidean distance between 2 vectors
vec1 = np.random.randn(10)
vec2 = np.random.randn(10)
dist = np.sqrt((vec1-vect2)**2)
```

2.2 Matplotlib: Visualization with Python

Source: https://matplotlib.org/tutorials/index.html#introductory

Matplotlib is the most popular plotting library in python. Using matplotlib, you can create pretty much any type of plot.

2.2.1 Getting Started

We carry out to plot by following steps:

1. *Prepare your data*: usually save in the a list or Numpy array.

2. *Create the plot/sub-plot.*

3. *Customize your plot*: adjust the linewidth, color, marker, axes, etc.

4. *Show/Save plot.*

We will illustrate this diagram in the following examples:

1. Plotting two graphs on the same plot.

2. Filled a region between two graphs.

3. Scatter plot.

4. Bar Chart plot.

First plot:

▷ Step 0: Import Libraries

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

▷ Step 1: Prepare Your Data

```
# Create a mesh with 256 points from −π to π
x = np.linspace(-np.pi, np.pi, 256)

# Find the y values accordingly
y1 = np.sin(x)
y2 = np.cos(x)
```

▷ Step 2: Create The Plot

```
plt.plot(x,y1)
plt.plot(x,y2)
```

▷ Step 3: Customize your plot

You can set the plot title, and labels for x and y axes.

```
plt.xlabel("angle")
plt.ylabel("sine/cosine")
plt.title('sine and cosine waves')
```

▷ Step 4: Show/Save Plot.

```
plt.show()
```

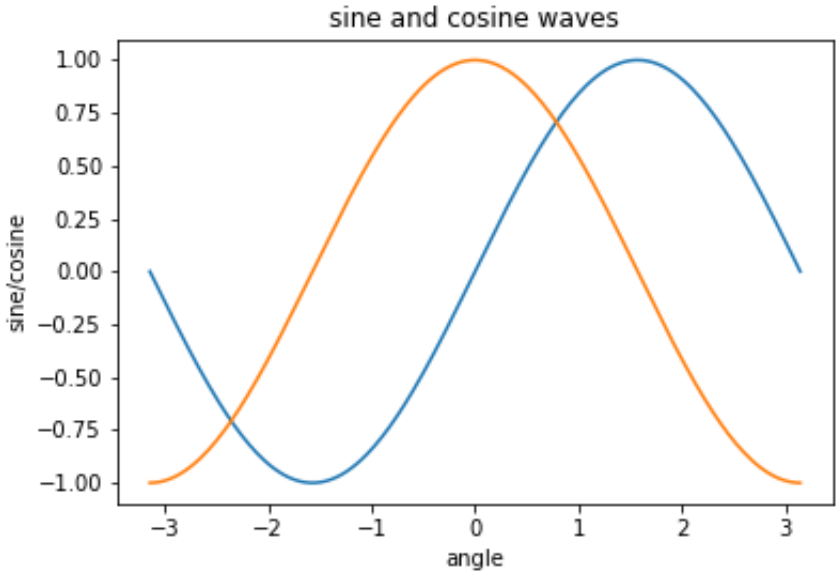


Figure 2.1: The first plot.

Well to do that, let's understand a bit more about what arguments plt.plot() expects. The plt.plot accepts *3 basic arguments* in the following order: (x, y, format).

This format is a short hand combination of {color}{marker}{line}. For example, the format 'go-' has 3 characters standing for: *'green colored dots with solid line'*. By omitting the line part ('-') in the end, you will be left with only green dots ('go'), which makes it draw a scatterplot.

Few commonly used short hand format examples are:

1. 'r*–': 'red stars with dashed lines'

2. 'ks.': 'black squares with dotted line' ('k' stands for black)

3. 'bD-.': 'blue diamonds with dash-dot line'.

```
x = np.linspace(0,10,50)
sinus = np.sin(x)
plt.plot(x,sinus, "go")
plt.show()
```

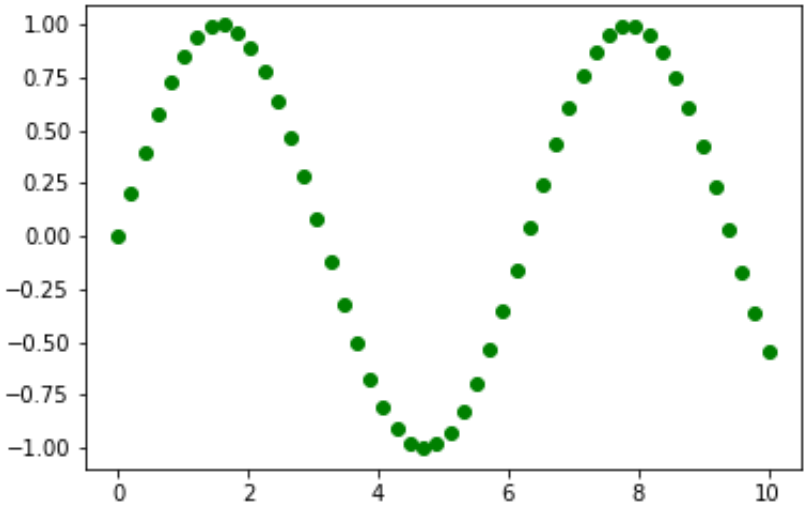


Figure 2.2: Graph of sine function using dots.

Filled a region between two graphs: Fill the region between the graphs of sine and cosine function from $-3\pi/4$ to $\pi/4$ from Figure 2.1.

```
# Step 1
x = np.linspace(-np.pi, np.pi, 256)
y1, y2 = np.sin(x), np.cos(x)
x3 = x[(x <= np.pi/4) & (x >= -3*np.pi/4)]
y3,y4 = np.sin(x3), np.cos(x3)

# Step 2
plt.plot(x,y1)
plt.plot(x,y2)
plt.fill_between(x3, y3, y4, facecolor='yellow')

# Step 3
plt.xlabel("angle")
plt.ylabel("sine/cosine")
plt.title('sine and cosine waves')

# Step 4
plt.show()
```

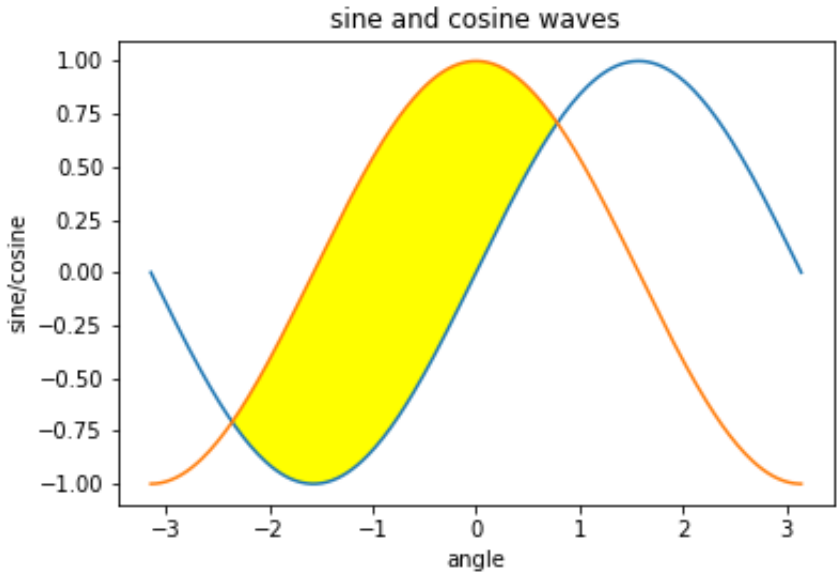


Figure 2.3: Region between the sine and cosine function from $-3\pi/4$ to $\pi/4$.

Scatter plot:

A scatter plot is a diagram where each value in the data set is represented by a dot (with color, shape and size).

The Matplotlib module has a method for drawing scatter plots, it needs two arrays of the same length, one for the values of the x-axis, and one for the values of the y-axis:

```
# Step 1
# Note: data can be a list of numpy array
x = [1, 2, 3, 4, 5]
y1 = [2, 3, 4, 5, 7]
y2 = [3, 4, 5, 5.5, 7.5]

# Step 2 & 3
plt.scatter(x, y1, color='blue')
plt.scatter(x, y2, color='red')

# Step 4
plt.show()
```

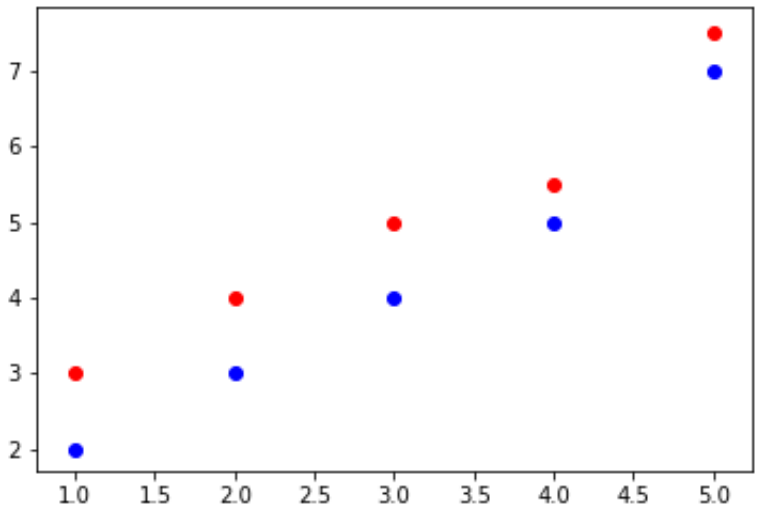


Figure 2.4: Scatter plot.

Bar Chart plot:

A bar chart or bar graph is a chart or graph that presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally. A vertical bar chart is sometimes called a column chart.

The function for Bar plot is ax.bar(x, height, width, color, ...), where:

1. x is a sequence of scalars representing the x coordinates of the bars.
2. height is a scalar or a sequence of scalars representing the height(s) of the bars.
3. width is scalar or an array-like, optional. the width(s) of the bars default 0.8.
4. color is scalar or an array-like, optional. the color(s) of the bars.

```
# Step 1
# Note: data can be a list of numpy array
x = [1, 2, 3, 4, 5]
y = [10, 24, 36, 48]
tick_label = ['one', 'two', 'three', 'five']

# Step 2 & 3
plt.bar(x, y, tick_label=tick_label,width=0.8, color=['red', 'green'])
plt.xlabel('x - axis')
plt.ylabel('y - axis')
```

```
plt.title('My bar chart!')

# Step 4
plt.show()
```

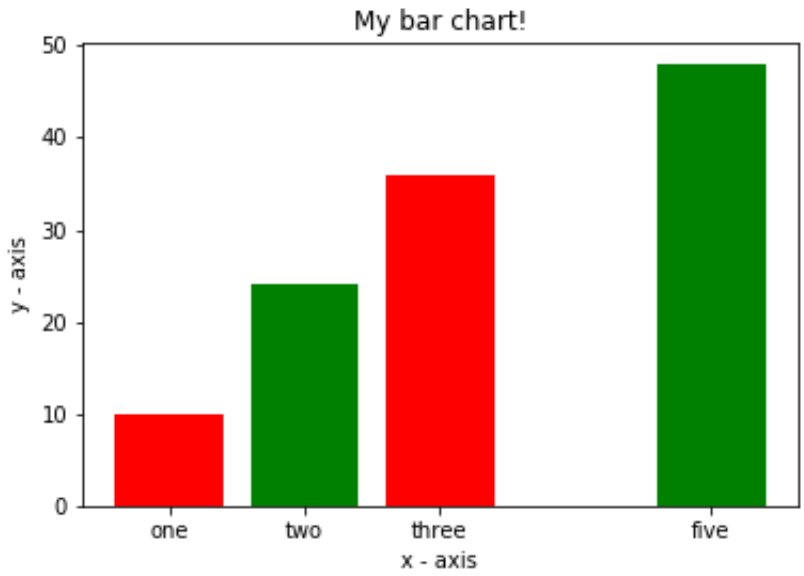


Figure 2.5: Bar Chart plot.

2.2.2 Figures and Axes

In *matplotlib*, *pyplot* is used to create **Figures** and change the *characteristics* of figures. The figure contains the overall window where plotting happens, contained within the figure are where actual graphs are plotted. All plotting is done with respect to an **Axes** (or *subplot*).

Plotting graphs via Figure and Axes:

Now let's create a figure, add an axes to the figure, and plot via axes.

```
x = np.linspace(-10, 9, 20)
y = x ** 3

fig = plt.figure()
ax = figure.add_axes([0.2, 0.2, 0.7, 0.8])

ax.plot(x, y, 'b')
ax.set_xlabel('X Axis')
ax.set_ylabel('Y Axis')
```

```
10 ax.set_title('Cube function')
11
12 plt.show()
```

In the code:

Line 4: The figure method called using pyplot class returns figure object.

Line 5: We can call add_axes method to add an axes to this object. The parameters $[0.2, 0.2, 0.7, 0.8]$ passed to the add_axes method are the *distance* from the *left* and *bottom* of the *default axis* (of Figure): $[0.2, 0.2]$; and the *width* and *height* of the axis: 0.7 and 0.8, respectively.

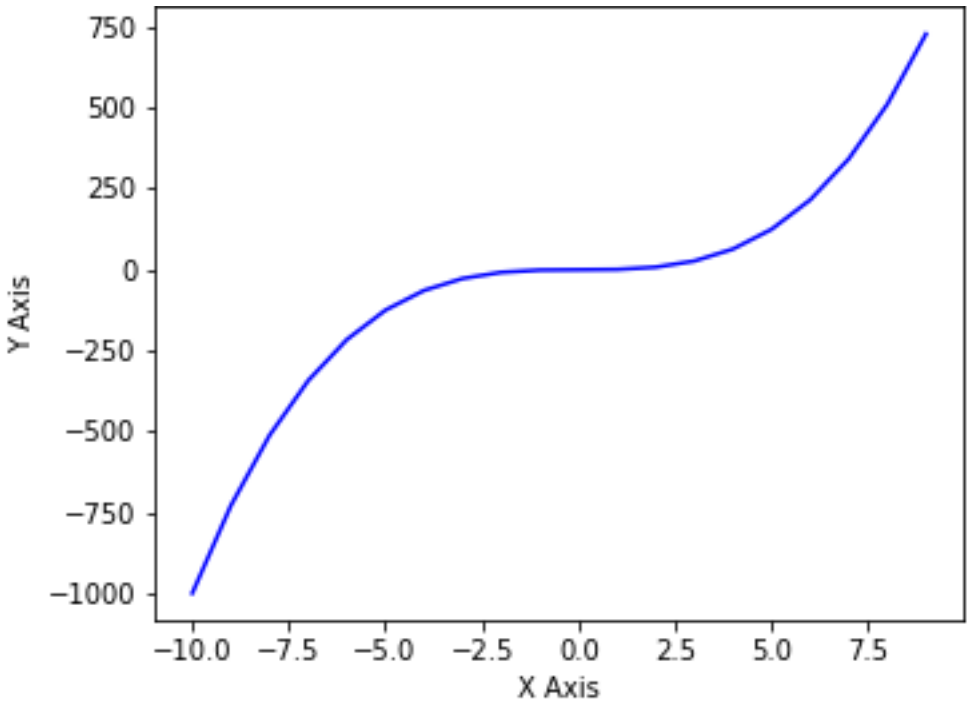


Figure 2.6: Plotting via Axes.

You can add as many axes as you want on one plot using the add_axes method. Take a look at the following example:

```
1 x = np.linspace(-10, 9, 20)
2 y = x ** 3
3 z = x ** 2
4
5 fig = plt.figure()
6 ax = fig.add_axes([0.16, 0.16, 0.8, 0.75])
7 ax2 = fig.add_axes([0.50, 0.30, 0.30, 0.20]) # inset axes
```

```
8
9  # Plot the cube function in Axes: ax
10 ax.plot(x, y, 'b')
11 ax.set_xlabel('X Axis')
12 ax.set_ylabel('Y Axis')
13 ax.set_title('Cube function')
14
15 # Plot the quadratic function in Axes: ax2
16 ax2.plot(x, z, 'r')
17 ax2.set_xlabel('X Axis')
18 ax2.set_ylabel('Y Axis')
19 ax2.set_title('Square function')
20
21 plt.show()
```

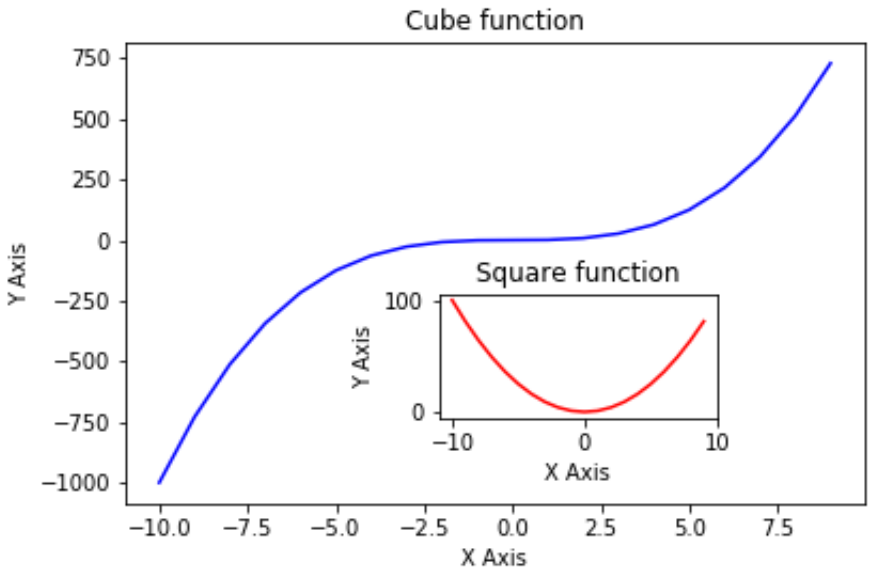


Figure 2.7: Two Axes.

Subplots:

Another way to create more than one plots at a time is to use subplots method. This method creates a figure and a grid of subplots with a single call, while providing reasonable control over how the individual plots are created.

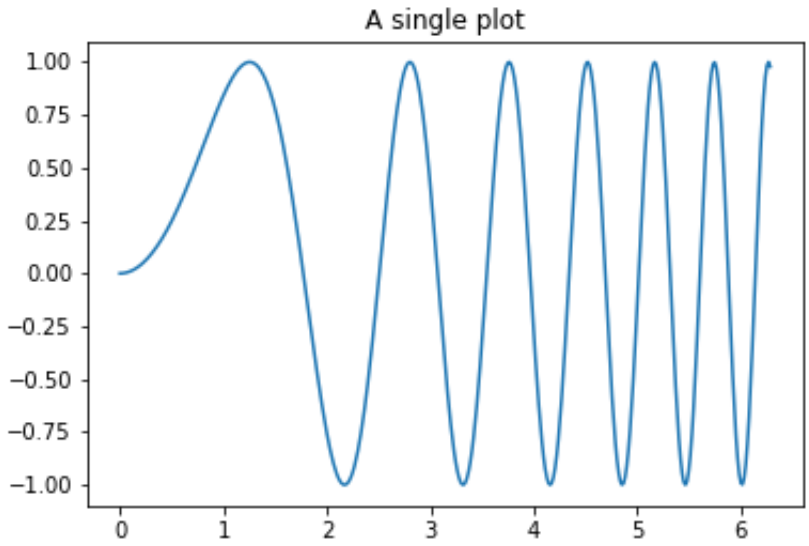
▷ Some example data to display:

```
1 x = np.linspace(0, 2 * np.pi, 400)
2 y = np.sin(x ** 2)
```

▷ A FIGURE WITH JUST ONE SUBPLOT: subplots() without arguments returns a Figure and a single Axes.

This is actually the simplest and recommended way of creating a single Figure and Axes.

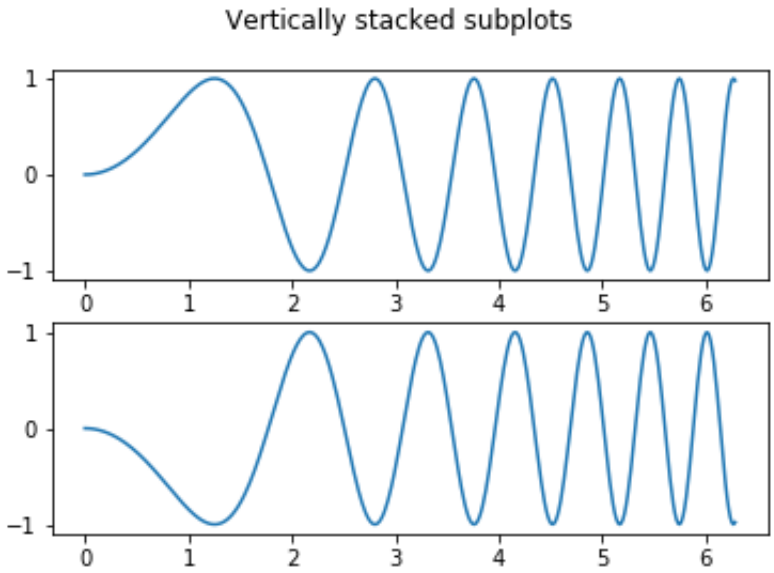
```
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('A single plot')
plt.show()
```



▷ STACKING SUBPLOTS IN ONE DIRECTION: The first two optional arguments of subplots define the number of **rows** and **columns** of the subplot grid.

When stacking in one direction only, the returned **axes** is a 1D numpy array containing the list of created **Axes**.

```
fig, axs = plt.subplots(2)
fig.suptitle('Vertically stacked subplots')
axs[0].plot(x, y)
axs[1].plot(x, -y)
```

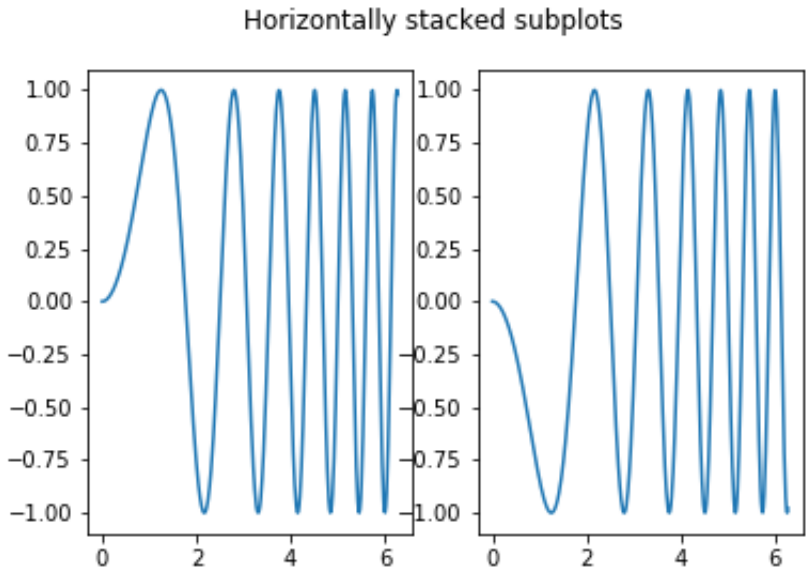


If you are creating just a few Axes, it's handy to unpack them immediately to dedicated variables for each Axes. That way, we can use ax1 instead of the more verbose axs[0].

```
fig, (ax1,ax2) = plt.subplots(2)
fig.suptitle('Vertically stacked subplots')
axs1.plot(x, y)
axs2.plot(x, -y)
```

▷ Side-by-side subplots: To obtain side-by-side subplots, pass parameters 1, 2 for one *row* and two *columns*.

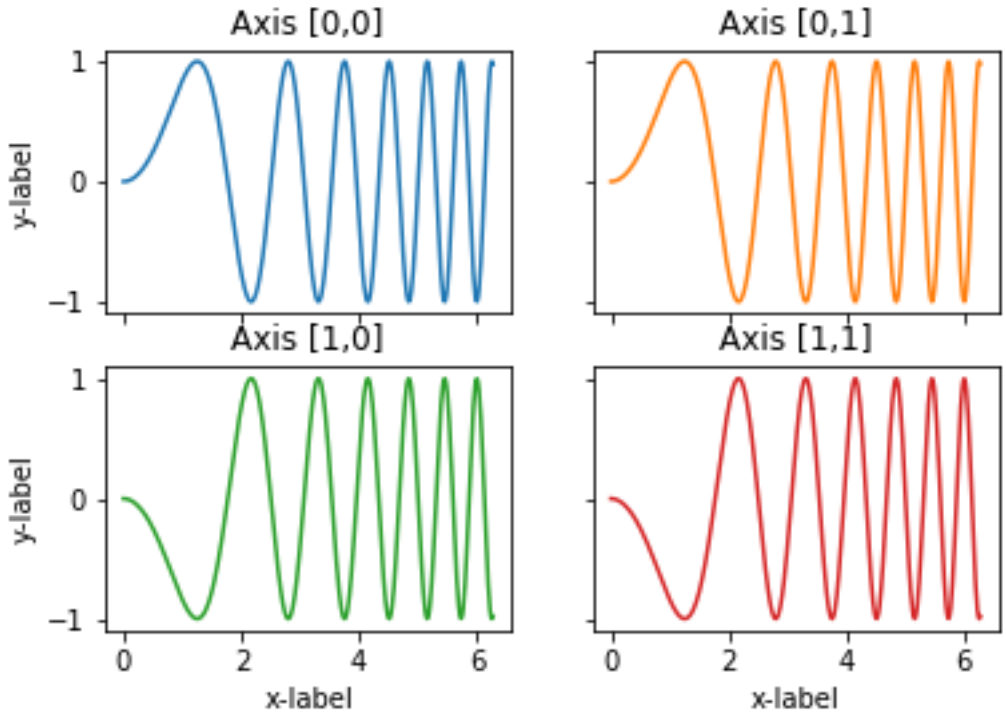
```
fig, (axs1,axs2) = plt.subplots(1,2)
fig.suptitle('Horizontally stacked subplots')
axs1.plot(x, y)
axs2.plot(x, -y)
```



▷ Stacking subplots in two directions: When stacking in two directions, the returned axs is a 2D numpy array.

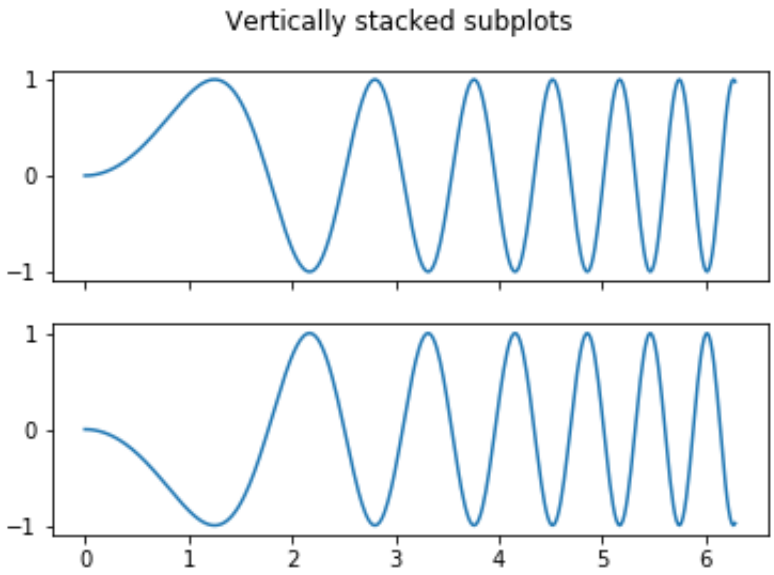
If you have to set parameters for each subplot it's handy to iterate over all subplots in a 2D grid using for ax in axs.flat:

```
fig, axs = plt.subplots(2, 2)
axs[0, 0].plot(x, y)
axs[0, 0].set_title('Axis [0,0]')
axs[0, 1].plot(x, y, 'tab:orange')
axs[0, 1].set_title('Axis [0,1]')
axs[1, 0].plot(x, -y, 'tab:green')
axs[1, 0].set_title('Axis [1,0]')
axs[1, 1].plot(x, -y, 'tab:red')
axs[1, 1].set_title('Axis [1,1]')

for ax in axs.flat:
    ax.set(xlabel='x-label', ylabel='y-label')

# Hide x labels and tick labels for top plots and y ticks for right
    plots.
for ax in axs.flat:
    ax.label_outer()
```



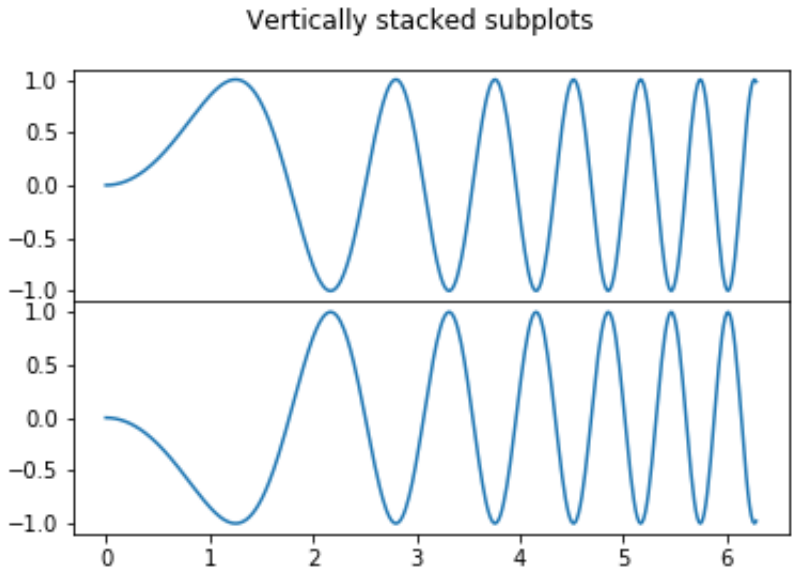
▷ SHARING AXES: You can use sharex or sharey to align the horizontal or vertical axis.

```
fig, (axs1,axs2) = plt.subplots(2, sharex=True)
fig.suptitle('Vertically stacked subplots')
axs1.plot(x, y)
axs2.plot(x, -y)
```



The parameter gridspec_kw of subplots controls the grid properties. For example, we can reduce the height between vertical subplots using gridspec_kw={'hspace': 0}.

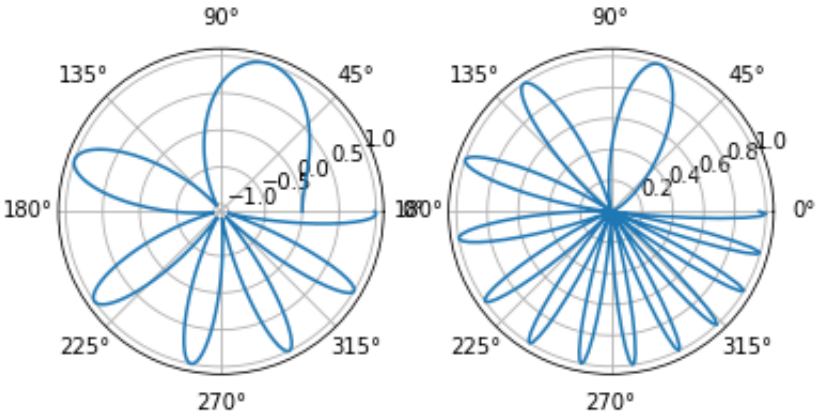
```
fig, (axs1,axs2) = plt.subplots(2, sharex=True,
                                gridspec_kw = {'hspace':0})
fig.suptitle('Vertically stacked subplots')
axs1.plot(x, y)
axs2.plot(x, -y)
```



Polar axes

The parameter subplot_kw of subplots controls the subplot properties. In particular, this can be used to create a grid of **polar Axes**.

```
fig, (ax1, ax2) = plt.subplots(1, 2,
                               subplot_kw=dict(projection='polar'))
ax1.plot(x, y)
ax2.plot(x, y ** 2)


plt.show()
```



2.2.3 Axis Ticks Positions, Labels and Legends

▷ Axis Ticks Positions and Labels

There are 3 basic things you will probably ever need in matplotlib when it comes to manipulating axis ticks:

1. Using plt.xticks() or ax.setxticks() and ax.setxticklabels() to control the position and tick labels.

2. Using plt.tick_params() to control axis's ticks (top/bottom/left/right) should be displayed.

3. Functional formatting of tick labels.

```
from matplotlib.ticker import FuncFormatter

def rad_to_degrees(x, pos):
    'converts radians to degrees'
    return round(x * 57.2985, 2)

plt.figure(figsize=(12,7), dpi=100)
X = np.linspace(0,2*np.pi,1000)
plt.plot(X,np.sin(X))
plt.plot(X,np.cos(X))

# 1. Adjust x axis Ticks
```

```
plt.xticks(ticks=np.arange(0, 440/57.2985, 90/57.2985), fontsize=12,
    rotation=30, ha='center', va='top')  # 1 radian = 57.2985 degrees

# 2. Tick Parameters
plt.tick_params(axis='both',bottom=True, top=True, left=True, right=
    True, direction='in', which='major', grid_color='blue')

# 3. Format tick labels to convert radians to degrees
formatter = FuncFormatter(rad_to_degrees)
plt.gca().xaxis.set_major_formatter(formatter)

plt.grid(linestyle='--', linewidth=0.5, alpha=0.15)
plt.title('Sine and Cosine Waves\n(Notice the ticks are on all 4 sides
     pointing inwards, radians converted to degrees in x axis)',
    fontsize=14)
plt.show()
```

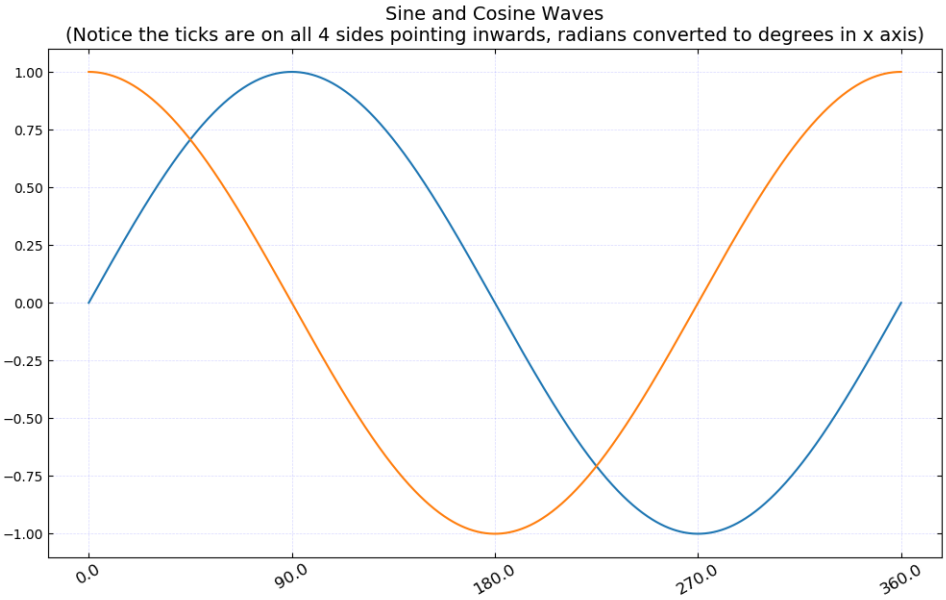


Figure 2.8: Modifying sticks.

In above code, plt.xticks takes the ticks and labels as required parameters but you can also adjust the label's fontsize, rotation, 'horizontalalignment' and 'verticalalignment' of the hinge points on the labels, like I've done in the below example.

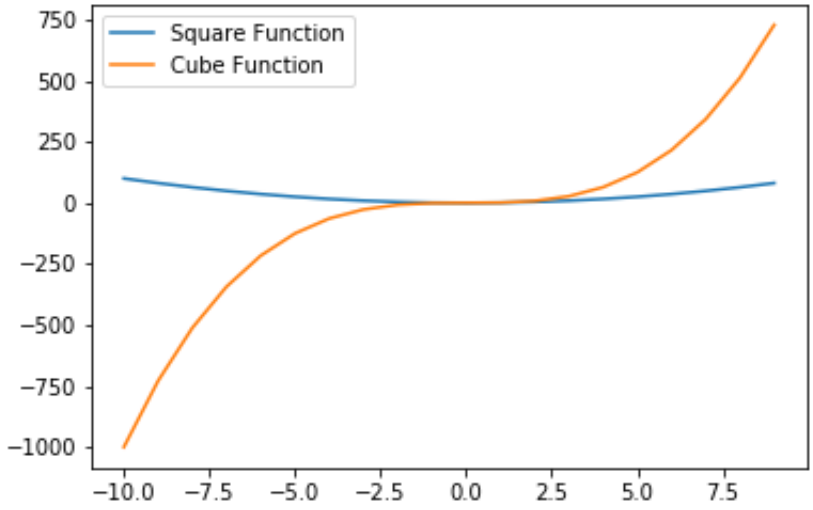
plt.tick_params() is used to determine which all axis of the plot ('top' / 'bottom' / 'left' / 'right') you want to draw the ticks and which direction ('in' / 'out') the tick should point to.

the matplotlib.ticker module provides the FuncFormatter to determine how the final tick label should be shown.

▷ Legend

Adding legends to a plot is very straightforward. All you have to do is to pass the value for the label parameter of the plot function. Then after calling the plot function, you just need to call the legend function. Take a look at the following example:

```
x = np.linspace(-10, 9, 20)
y = x ** 3
z = x ** 2

plt.plot(x, z, label="Square Function")
plt.plot(x, y, label="Cube Function")
plt.legend()

plt.show()
```



You can customize the Legend as you want:

```
plt.figure(figsize=(10,7), dpi=80)
X = np.linspace(0, 2*np.pi, 1000)
sine = plt.plot(X,np.sin(X)); cosine = plt.plot(X,np.cos(X))
sine_2 = plt.plot(X,np.sin(X+.5)); cosine_2 = plt.plot(X,np.cos(X+.5))

# Adjust the range of coordinates
plt.xlim(-0.5,7)
```

```
plt.ylim(-1.25,1.5)
plt.title('Custom Legend Example', fontsize=18)

# Modify legend
plt.legend([sine[0],cosine[0],sine_2[0],cosine_2[0]],  # plot items
        ['sine curve','cosine curve','sine curve 2','cosine curve 2'],
        frameon=True,                          # legend border
        framealpha=1,                          # transparency of border
        ncol=2,                                # num columns
        shadow=True,                           # shadow on
        borderpad=1,                           # thickness of border
        title='Sines and Cosines')             # title
plt.show()
```

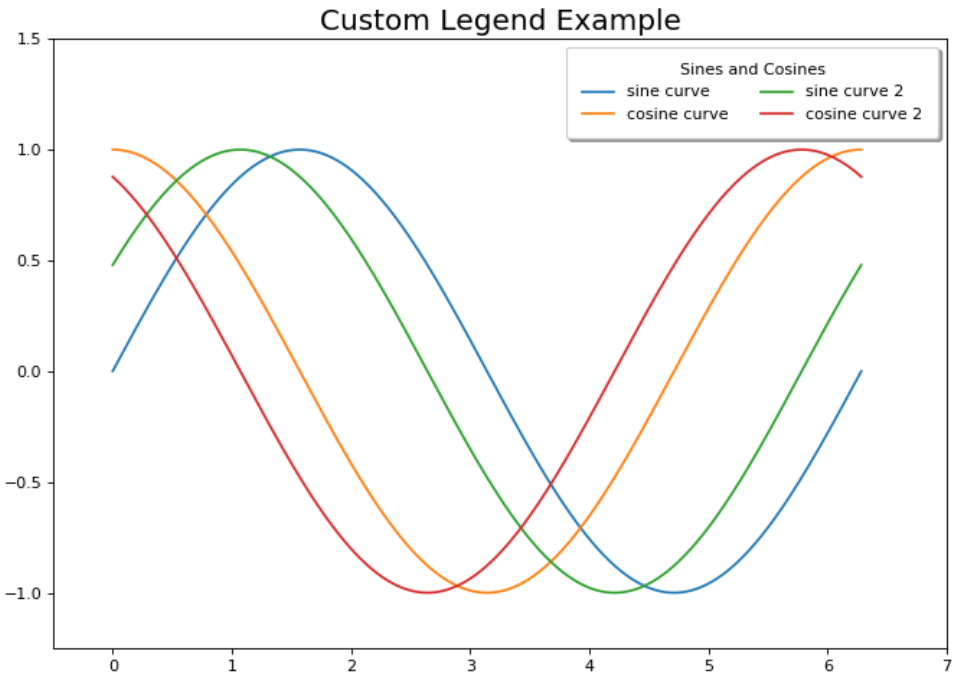


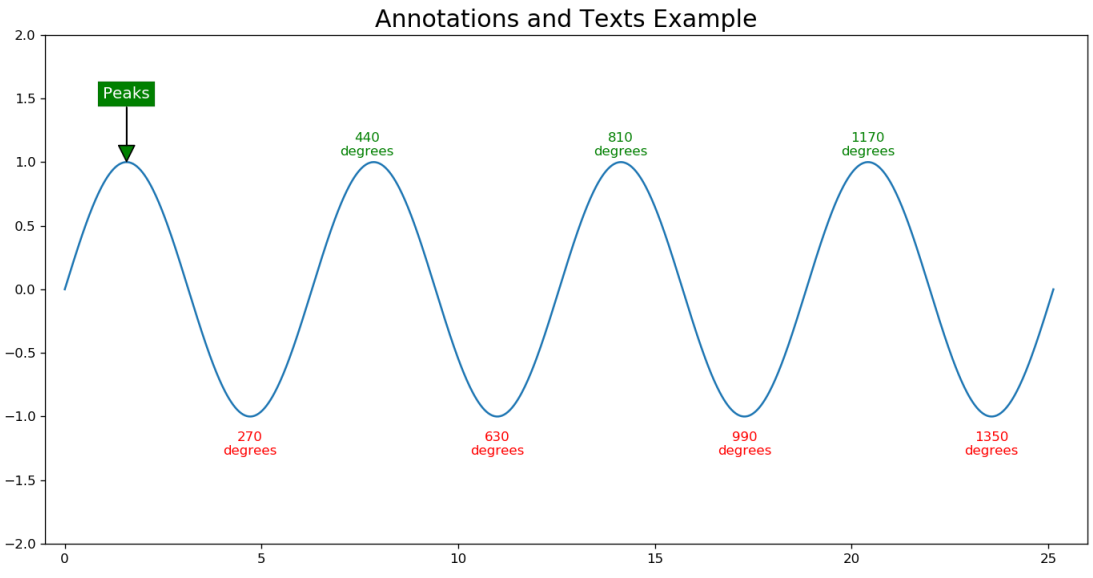
Figure 2.9: Customise the Legend.

2.2.4 Add Texts, Arrows and Annotations

plt.text and plt.annotate adds the texts and annotations respectively. If you have to plot multiple texts you need to call plt.text() as many times typically in a for-loop.

Let's *annotate* the *peaks* and *troughs* adding *arrowprops* and a *bbox* for the text.

```
# Texts, Arrows and Annotations Example
# ref: https://matplotlib.org/users/annotations_guide.html
plt.figure(figsize=(14,7), dpi=120)
X = np.linspace(0, 8*np.pi, 1000)
sine = plt.plot(X,np.sin(X), color='tab:blue');

# 1. Annotate with Arrow Props and bbox
plt.annotate('Peaks', xy=(90/57.2985, 1.0), xytext=(90/57.2985, 1.5),
             bbox=dict(boxstyle='square', fc='green', linewidth=0.1),
             arrowprops=dict(facecolor='green', shrink=0.01, width
    =0.1),
             fontsize=12, color='white', horizontalalignment='center')

# 2. Texts at Peaks and Troughs
for angle in [440, 810, 1170]:
    plt.text(angle/57.2985, 1.05, str(angle) + "\ndegrees", transform=
    plt.gca().transData, horizontalalignment='center', color='green')

for angle in [270, 630, 990, 1350]:
    plt.text(angle/57.2985, -1.3, str(angle) + "\ndegrees", transform=
    plt.gca().transData, horizontalalignment='center', color='red')

plt.gca().set(ylim=(-2.0, 2.0), xlim=(-.5, 26))
plt.title('Annotations and Texts Example', fontsize=18)
plt.show()
```



Notice, all the text we plotted above was in relation to the data.

That is, the x and y position in the plt.text() corresponds to the values along the x and y axes. However, sometimes you might work with data of different scales on different subplots and you want to write the texts in the same position on all the subplots.

In such case, instead of manually computing the x and y positions for each axes, you can specify the x and y values in relation to the axes (instead of x and y axis values). You can do this by setting transform=ax.transData.

The lower left corner of the axes has $(x, y) = (0, 0)$ and the top right corner will correspond to $(1, 1)$.

The function plt.cga() gives the current *Axes* instance on the current figure.

2.3 Pandas: data manipulation

Source:

`http://pandas.pydata.org/pandas-docs/stable/index.html`

Pandas DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

2.3.1 Basic manipulation

▷ Import pandas and numpy:

```
import pandas as pd
import numpy as np
```

▷ Create DataFrame

```
# Create names for columns of DataFrames
col_names = ['name', 'age', 'gender', 'job']

# user1 has two rows and four columns
user1 = pd.DataFrame([['alice', 19, "F", "student"],
                      ['john', 26, "M", "student"]],
                      columns = col_names)

user2 = pd.DataFrame([['eric', 22, "M", "student"],
                      ['paul', 58, "F", "manager"]],
```

```
                      columns=col_names)

user3 = pd.DataFrame(dict(name=['peter', 'julie'],
                          age=[33, 44], gender=['M', 'F']),
                     columns=col_names)
print(user3)
```

Output:

```
    name  age gender  job
0  peter   33      M  NaN
1  julie   44      F  NaN
```

▷ Concatenate DataFrames

```
user1.append(user2)
users = pd.concat([user1,user2, user3])
print(users)
```

Output:

```
   name   age   gender  job
0   alice   19       F   student
1   john    26       M   student
0   eric    22       M   student
1   paul    58       F   manager
0   peter   33       M    NaN
1   julie   44       F    NaN
```

▷ Columns Selection

```
users['gender']         # select one column
type(users['gender'])   # Series
users.gender            # select one column using the DataFrame

# select multiple columns
users[['age', 'gender']]      # select two columns
my_cols = ['age', 'gender']   # or, create a list...
users[my_cols]                # ...and use that list to select columns
type(users[my_cols])          # DataFrame
```

▷ Rows selection (basic)

iloc is strictly integer position based:

```
df = users.copy()   # Copy a DataFrame from users
df.iloc[0]          # first row
df.iloc[0, 0]       # first item of first row
df.iloc[0, 0] = 55  # Set first item of first row be 55
```

▷ Rows selection (filtering)

```
# only show users with age < 20
users[users.age < 20]

# use multiple conditions
users[(users.age > 20) & (users.gender == 'M')]
```

2.3.2 CSV Files

Source: `https://en.wikipedia.org/wiki/Comma-separated_values`

Definition 16. A *comma-separated values* (CSV) file is a delimited text file that uses a comma to separate values. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format.

To create a *CSV* file with a text editor, first choose your favorite text editor, such as Notepad or vim, and open a new file. Then enter the text data you want the file to contain, separating each value with a comma and each row with a new line. Often, the first row contains the names of columns.

Save this file with the extension .csv. You can then open the file by using a DataFrame.

▷ Create DataFrame from a CSV File

```
url ='https://raw.github.com/neurospin/pystatsml/master/datasets/
    salary_table.csv'
salary = pd.read_csv(url)

# Print first 5 rows in salary
salary.head()
```

Output:

```
   salary experience  education management
0  13876     1          Bachelor  Y
1  11608     1          Ph.D      N
2  18701     1          Ph.D      Y
3  11283     1          Master    N
4  11767     1          Ph.D      N
```

2.4 Graphs and NetworkX

2.4.1 Introduction to Graphs

Source: R. Diestel, Graph Theory, 5th Edition, Springer-Verlag, 2006.

A Graph is a non-linear data structure consisting of nodes and edges.

Definition 17. An *graph* $G = (V, E)$ is a set V of vertices (or nodes) and a set E of edges. Each edge $e \in E$ is associated with two vertices u and v from V , and we write $e = (u, v)$. We say that u is adjacent to v, u is incident to v, and u is a neighbor of v.

The graph in the definition above is an *undirected graph*. Sometimes we want to associate a direction with the edges to indicate a one-way relationship. For example, in making a graph representation of a road network where some streets were one-way. This naturally leads to directed graphs:

Definition 18. A *directed graph* $G = (V, E)$ is a set V of vertices (or nodes) and set E of edges. Each edge $e \in E$ is an ordered pair of vertices from V. We denote the edge from u to v by (u, v) and say that u points to v.

In many applications, each edge of a graph has an associated numerical value, called a *weight*. For example, when a graph representation of a road network, the weight of an edge is assigned by its length.

Definition 19. A *weighted graph* $G = (V, E, \omega)$ is a graph (V, E) with an associated *weight function* $\omega\colon E \to \mathbb{R}$. In other words, each edge e has an associated weight $\omega(e)$.

Visualize Graphs

We visualize graphs by using points to represent vertices and line segments, possibly curved, to represent edges, where the endpoints of a line segment representing an edge are the points representing the endpoints of the edge.

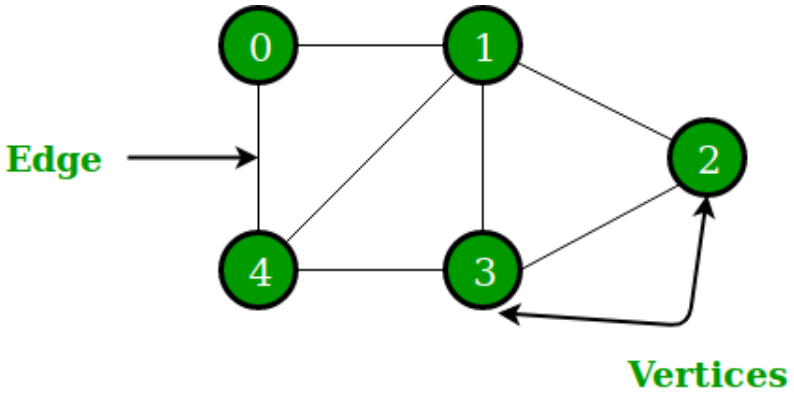


Figure 2.10: Visualize a graph.

Graph Terminology

A *path* in the graph G is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that the edge $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k-1$. The *length* of this path is $k-1$, it is the number of edges on the path. A path $v_1, v_2, \ldots, v_k$ is *simple* if $v_i \neq v_j$ for all $i \neq j$. For example, the graph G depicted in Figure 2.10 has a simple path $2, 1, 0, 4, 3$ of length 4.

A *cycle* of length k, $k \geqslant 3$, is a path $v_0, v_1, \ldots, v_k$ such that $v_i \neq v_j$ for all $0 \leqslant i < j < k$, and $v_0 = v_k$.

The *neighborhood* of a node u of G, denoted by $N(u)$, consists of all nodes v with $(u, v) \in E$.

In an undirected graph, the *degree* of a node $u \in V$, denoted by $\deg(u)$, is the number of edges $e = (u, v) \in E$, so that $\deg(u)$ is the number of neighbors of u. For example, consider the node 3 in the graph G depicted in Figure 2.10, we have

$$N(3) = \{1, 2, 4\} \text{ and } \deg(3) = 3.$$

In a directed graph, the *in-degree* of a node $u \in V$ is the number of edges that point to u, i.e., the number of edges $(v, u) \in E$. The *out-degree* of u is the number of edges that point away from u, i.e., the number of edges $(u, v) \in E$.

A *subgraph* of the graph G is a graph $H = (W, F)$, where $W \subseteq V$ and $F \subseteq E$. A subgraph H of G is a proper subgraph of G if $H \neq G$.

A graph is *connected* if there is a path from every vertex to every other vertex in the graph. A graph that is not connected consists of a set of *connected components*, which are maximal connected subgraphs.

Remark 20. In this course we only consider *simple graphs*, i.e. there is at most one edge connected between two nodes of the considered graph.

2.4.2 Trees

Tree is a discrete structure that represents hierarchical relationships between individual elements or nodes.

Definition 21. A *tree* is a connected undirected graph with no cycles.

In many applications of trees, a particular vertex of a tree is designated as the *root*. Once we specify a root, we can assign a *direction* to each edge as follows. Since there is a unique path from the root to each vertex of the tree, we direct each edge away from the root. Thus, a tree together with its root produces a directed graph called a *rooted tree*.

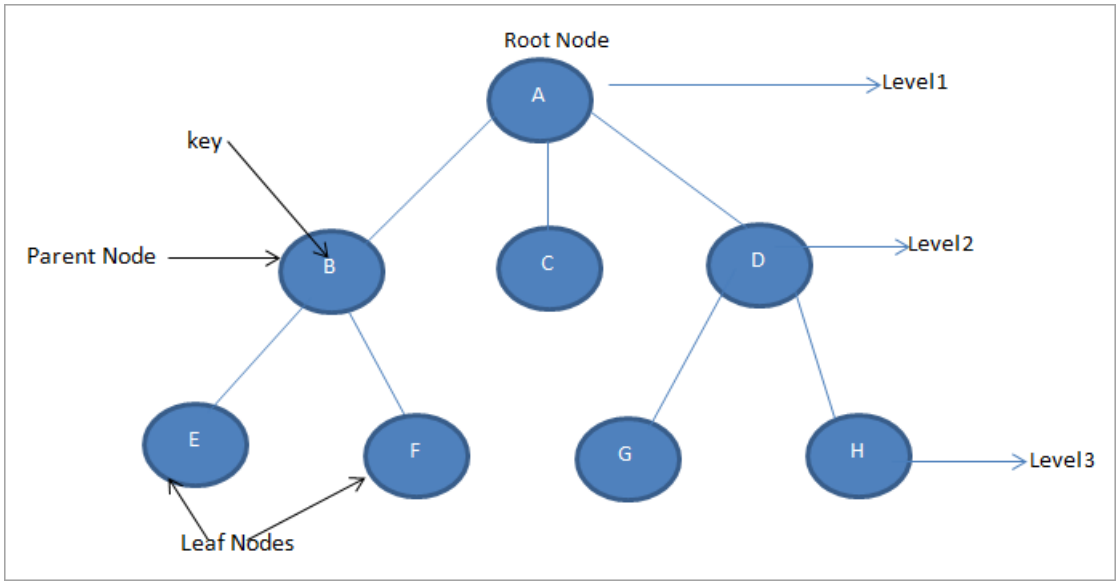


Figure 2.11: Rooted tree.

Definition 22. A rooted tree is a tree in which one vertex has been designated as the root and every edge is directed away from the root.

Suppose that T is a rooted tree. If v is a vertex in T other than the root, the *parent* of v is the unique vertex u such that there is a directed edge from u to v (such a vertex is unique). When u is the parent of v, v is called a *child* of u.

The *ancestors* of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root (that is, its parent, its parent's parent, and so on, until the root is reached).

The *descendants* of a vertex v are those vertices that have v as an ancestor.

A vertex of a rooted tree is called a *leaf* if it has no children. Vertices that have children are called *internal vertices*.

The *level* of a vertex v in a rooted tree is the length of the unique path from the root to this vertex plus 1. Thus the level of root is 1.

An *ordered rooted tree* is a rooted tree where the children of each internal vertex are ordered.

Definition 23. A rooted tree is called an m-ary tree if every internal vertex has no more than m children. A complete m-ary tree is an ordered rooted m-ary tree in which every level, except possibly the last, is completely filled (i.e. has m children), and all nodes are as far left as possible at the last level. An m-ary tree with $m = 2$ is called a binary tree.

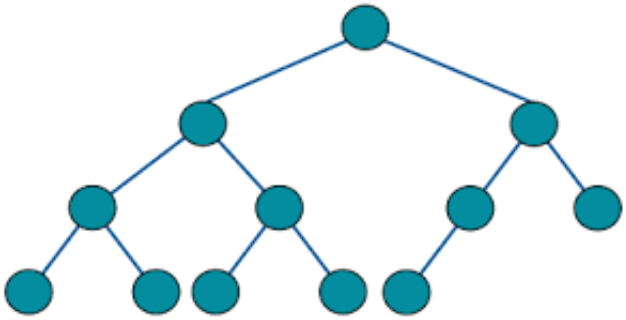


Figure 2.12: A Complete Binary Tree.

2.4.3 Represent a graph in computer

We can choose between two standard ways to represent a graph $G = (V, E)$ as a collection of *adjacency lists* or as an *adjacency matrix*.

▷ The adjacency-list

The representation of G consists of an array, say Adj, of n lists, $n = |V|$, one for each vertex in V. For each $u \in V$, the adjacency list Adj[u] contains the neighborhood of u (in some order).

We can readily adapt adjacency lists to represent weighted graphs. For a weighted graph (G, ω), we simply store the weight $\omega(u, v)$ of the edge (u, v) of E with vertex v in u's adjacency list.

A potential disadvantage of the adjacency-list representation is that it provides no quicker way to determine whether a given edge (u, v) is present in the graph than to search for in the adjacency list Adj[u]. An *adjacency-matrix* representation of the graph remedies this disadvantage.

▷ The adjacency-matrix

We assume that the vertices of G are numbered $\{0, 1, \ldots, n-1\}$ in some arbitrary manner. Then the adjacency-matrix representation of G is the $n \times n$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

For the weighted graph (G, ω), the adjacency-matrix representation should be

$$a_{ij} = \begin{cases} \omega(i, j) & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency-matrix representation always takes $O(|V|^2)$ space, so it is not appropriate for sparse graphs, i.e. graphs have relatively few edges.

2.4.4 NetworkX

NetworkX is a Python library for studying graphs and networks.

Source: `https://networkx.github.io/documentation/stable/index.html`

▷ Import the module NetworkX:

```
1 import networkx as nx
```

▷ Create the undirected graph $G = (V, E)$ where $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and

$$E = \{(1,2), (1,3), (2,3), (2,4), (2,5), (3,4), (4,5), (4,6), (5,7), (5,8), (7,8)\}$$

by the following code:

```
1 G = nx.Graph()
2 G.add_edges_from([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 4), (4,
     5), (4, 6), (5, 7), (5, 8), (7, 8)])
```

In this code, the line 1 creates an empty undirected graph G. The line 2, adding edges into G from the list, it automatically adds nodes into G from the list as well.

▷ In order to visualize G, we use matplotlib.pyplot:

```
1 import matplotlib.pyplot as plt
2 nx.draw_networkx(G)
3 plt.show()
```

The line 2 indicates that we draw G in a region generated by plt and the line 3 shows this image. Note that, in this context the coordinates of nodes of G are chosen randomly. It means that the appearance looks like different when we call the command again.

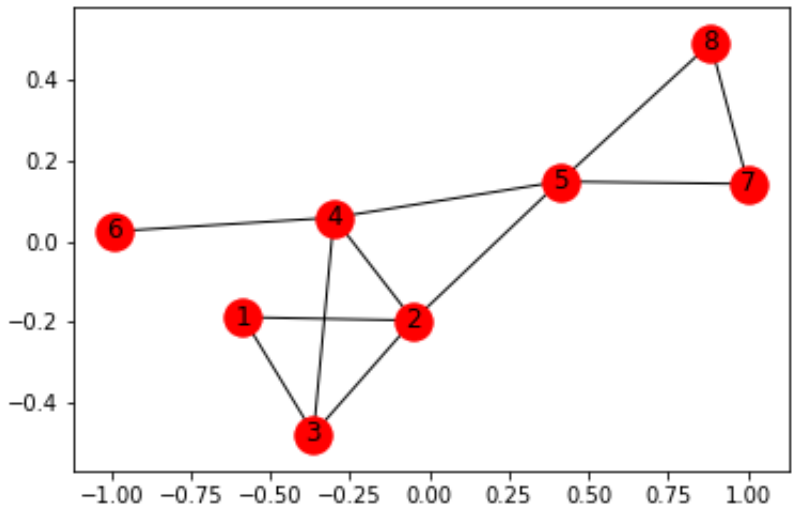


Figure 2.13: Draw the graph G with axes.

▷ We can omit the axes:

```
nx.draw_networkx(G)
plt.axis("off")
plt.show()
```

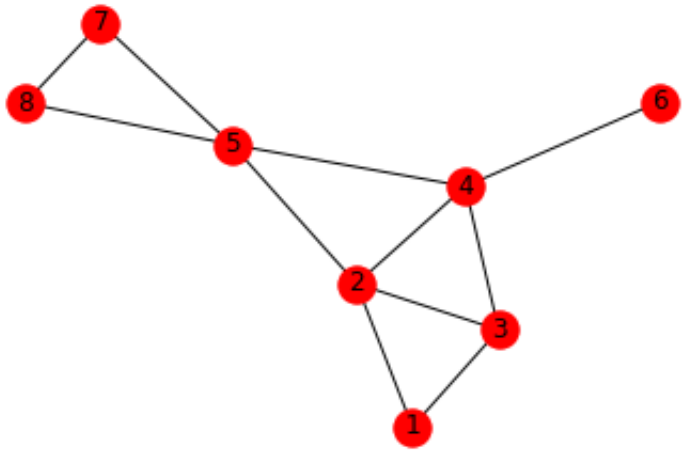


Figure 2.14: Draw the graph G without axes.

▷ NetworkX has many special graphs, for example, the complete bipartite graph $K_{3,3}$:

```
1 K33 = nx.complete_bipartite_graph(3,3)
2 nx.draw(K33, with_labels = True)
3 plt.show()
```

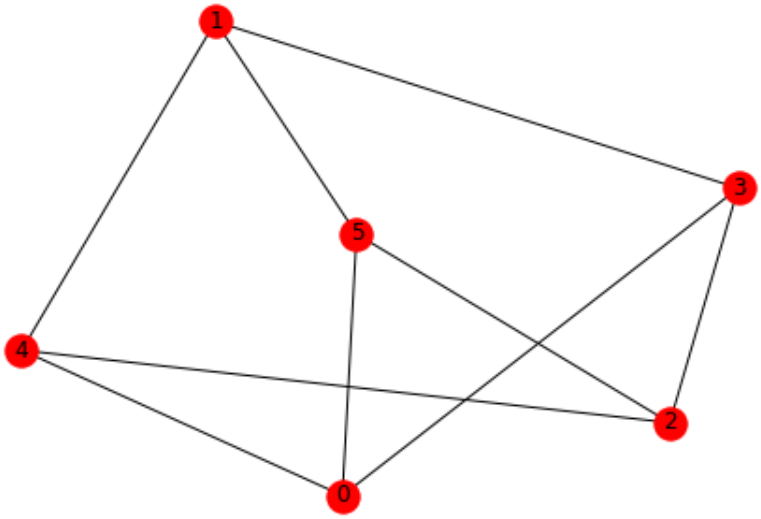


Figure 2.15: The graph $K_{3,3}$.

▷ In order to draw $K_{3,3}$ in a suitable appearance we set up the coordinates for nodes of $K_{3,3}$.

```
1 positions = {0:[-1,1], 1:[0,1], 2:[1,1], 3:[-1,-1], 4:[0,-1],
    5:[1,-1]}
2 nx.draw_networkx(K33, positions)
3 plt.show()
```

In the line 1, we let the coordinate for every node in the dictionary. Each item has the form **node : coordinate**.

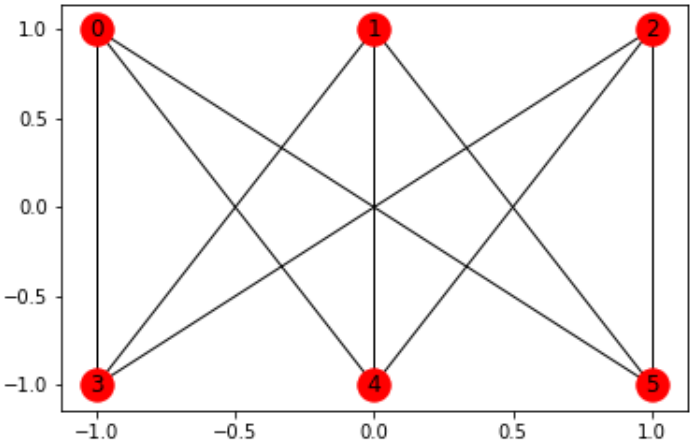


Figure 2.16: The graph $K_{3,3}$ with new apperance.

▷ Directed graphs:

```
1 H = nx.DiGraph()
2 H.add_nodes_from(['A', 'B', 'C'])
3 H.add_edges_from([('A', 'B'), ('B', 'C'), ('C', 'A'), ('A', 'C')])
4 nx.draw_networkx(H)
5 plt.show()
```

In this code, the line 1 creates an empty digraph H, the line 2 adds into G the nodes A, B and C, and the line 3, adds directed edges $(A, B), (B, C), (C, A), (A, C)$ into H

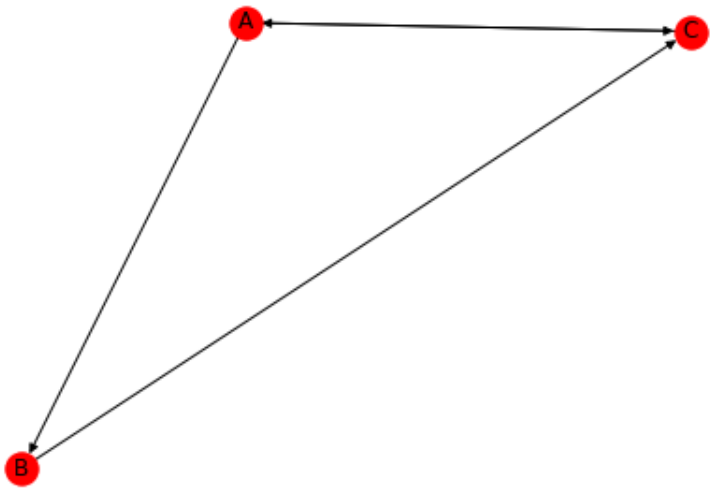


Figure 2.17: The directed graph H.

▷ Weighted graphs:

Recall that a weighted graph (G, ω) is a graph $G = (V, E)$ equipped with weight function $\omega \colon E \to \mathbb{R}$. The following code creates a weighted graph with weight = 'distance' (this is just name we called for the map ω).

```
1  G = nx.Graph()
2  G.add_nodes_from([1, 2, 3, 4, 5, 6])
3  G.add_weighted_edges_from([
4      (1, 2, 7),
5      (2, 4, 15),
6      (4, 5, 6),
7      (5, 6, 9),
8      (6, 1, 14),
9      (1, 3, 9),
10     (2, 3, 10),
11     (3, 4, 11)
12 ], weight='distance') # call the weight values 'distance'
13 G.add_edge(3,6, distance = 2)
```

In the code:

Lines $3 - 11$: each triple (u, v, w) means the edge (u, v) has weight w.

Line 13: we add a new edge $\{3, 6\}$ with weight 'distance' 2.

▷ In order to draw a graph with weight we need some extra works.

```
1  pos=nx.spring_layout(G) # pos = nx.nx_agraph.graphviz_layout(G)
2  nx.draw_networkx(G,pos)
3
4  edge_labels = nx.get_edge_attributes(G,'distance')
5  nx.draw_networkx_edge_labels(G,pos,edge_labels=edge_labels)
6  plt.axis("off")
7  plt.show()
```

In the code:

Line 1: gets the coordinate of nodes.

Line 2: draws the nodes.

Line 4: gets the labels of edges given by weight 'distance'.

Line 5: draws edges and labels of weight 'distance'.

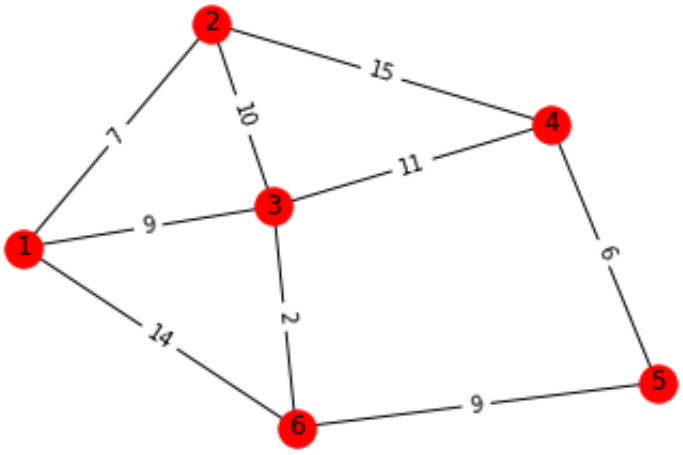


Figure 2.18: The weighted graph.

2.4.5 Some basic methods in NetworkX

Assume that G is a graph created by NetworkX. Let v be a node of G and (u, v) an edge of G.

▷ Information

```
G.nodes()    # Get the list of nodes
G.edges()   # Get the list of edges
G.number_of_nodes()   #  Get the number of nodes
G.number_of_edges()   # Get the number of edges
G.neighbors(v)   # Get the neighborhood of the node v
G.degree(v)   # Get the degree of the node v
```

▷ Delete

```
G.remove_node(v)   # Remove the node $v$ from $G$
G.remove_edge(u,v)   # Remove the edge $(u,v)$ from $G$
G.clear()   # Remove all nodes and edges from $G$
```

▷ In Directed Graphs

Assume more that G is a digraph.

```
G.out_degree(v)   # the out-degree of the node v
G.in_degree(v)    # the in-degree of the node v
```

```
G.degree(v)        # the degree of the node v
G.successors(v)    # the iterator over nodes out away from $v$
G.neighbors(v)     # the iterator over nodes going out from $v$
G.predecessors(v) # the iteration over nodes going to $v$
```

Chapter 3

Algorithms and Complexity

Source:

T. H. Cormen. C. E. Leiserson. R. L. Rivest and C. Stein, Introduction to Algorithms, Third Edition. The MIT Press.

3.1 Algorithms

Definition 24 (Algorithm)**.** Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*.

An algorithm is thus a *sequence* of computational steps that transform the input into the output.

▷ **Design**

Algorithm design refers to a method or a mathematical process for problem-solving and engineering algorithms.The three most important criteria for an algorithm are *correctness*, *efficiency*, and *simplicity*.

▷ **Complexity**

The complexity of an algorithm is a measure of the efficient use of the computer's resources. In particular, we would like them to run as fast as possible using as little memory as possible (though there is often a trade off here). To keep things simple, we will concentrate on the running time of the algorithms for a large part of this thread, and not worry too much about the amount of space (i.e., amount of memory) they use.

Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. To do so, we need to define the terms:

1. **The size of input**

   The input size is the memory that the computer needs to store the input. The best notion for input size depends on the problem being studied.

2. **The Running Time**

   The running time of the algorithm with input size n, denoted by $T(n)$, is the sum of running times for each statement executed.

▷ Big O-notation

In practice, when analyzing the complexity of an algorithm, it turns out that the exact number of operations is not as important as determining the most dominant part of the $T(n)$ function. In other words, as the problem gets larger, some portion of the $T(n)$ function tends to overpower the rest. This dominant term is what, in the end, is used for comparison. The order of magnitude function describes the part of $T(n)$ that increases the fastest as the value of n increases. Order of magnitude is often called Big-O notation (for "order") and written as O(f(n)). It provides a useful approximation to the actual number of steps in the computation. The function $f(n)$ provides a simple representation of the dominant part of the original $T(n)$.

Formally, we say that $T(n) = O(f(n))$ if there is a constant $c > 0$ such that

$$T(n) \leqslant cf(n) \text{ for all } n \gg 0.$$

Orders of common functions:

Notation	**Name**
$O(1)$	constant
$O(\log n)$	logarithmic
$O(\log n^c),\ c > 1$	polylogarithmic
$O(n^c),\ 0 < c < 1$	fractional power
$O(n)$	linear
$O(n \log n)$	loglinear
$O(n^2)$	quadratic
$O(n^c),\ c \geqslant 1$	polynomial
$O(c^n),\ c > 1$	exponential

3.1.1 Searching Algorithms

Problem: Given an array $a = [a_0, a_1, \ldots, a_{n-1}]$ of n elements, write a function to search the position of a given element x in a, and -1 if otherwise.

Example 25. 1. Let $a = [10, 20, 80, 30, 60, 50, 110, 100, 130, 170]$ and $x = 100$. The answer is 7 since the element x is present at index 7 (the start index is 0).

2. For $x = 175$, the answer is -1 since x is not in a.

▷ A simple approach is to do **linear search**:

1. Start from the leftmost element of a and one by one compare x with each element of a.

2. If x matches with an element, return the index.

3. If x doesn't match with any of elements, return -1.

▷ Linear search:

```
# Input: an array a of n numbers and an element x
# Output: The index i with a[i] == x or -1 if otherwise
def LinearSearch(a,x):
  for i in range(len(a)):
    if a[i] == x:
      return i
  return -1
```

Analyzing the linear search:

- The input size is n, the number of elements in the array.
- In order to analyze the complexity, we have:

- *In the worst case*, the target item is at the end of the list or not in the list at all. Then the algorithm must visit every item and perform n iterations for a list of size n. Thus, the worst-case complexity of a linear search is $O(n)$.

- *In the best case*, the algorithm finds the target at the first position, after making one iteration, for an $O(1)$ complexity.

- To determine *the average case*, you add the number of iterations required to find the target at each possible position and divide the sum by n. Thus, the algorithm performs

$$[n + (n-1) + (n-2) + \cdots + 1]/n,$$

or $(n+1)/2$ iterations. For very large n, the constant factor of $1/2$ is insignificant, so the average complexity is still $O(n)$.

This proves the following result.

Theorem 26. *The search algorithm is running in linear time.*

▷ **Binary Search**

A linear search is necessary for data that are not arranged in any particular order. When searching on a sorted array a, we can use a binary search.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

▷ Binary search:

```
#Input: an array a sorted in increasing order  and  item x
#Output: an index i with a[i]=x, if  i exists, or -1  otherwise
def BinarySearch(a,x):
  s = 0
  t = len(a)-1
  while s <= t:
    mid = (s+t) // 2
    if a[mid] == x:
      return mid
    if a[mid] < x:
      s = mid + 1
    else:
      t = mid - 1
  return -1
```

Theorem 27. *The running time of the binary search is* $T(n) = O(\log n)$.

Proof. Let say the iteration in Binary Search terminates after k iterations.

Let the length of the array a is n. Note that at each iteration, the array is divided by half. Thus,

1. At iteration 1: Length of array $= n$.
2. At iteration 2: Length of array $= n/2$.
3. At iteration 3: Length of array $= n/2^2$.

   ...

k. After iteration k: Length of array $= n/2^k$.

The worse case terminates when the length of the array is 1, so that $n/2^k \geqslant 1$. It follows that $k \leqslant \log_2 n$, and so $T(n) = O(\log n)$. □

▷ In Python, the index of an element x in list a is a.index(x):

```
a = [0,2,4,6,8,10]
a.index(4)  # result 2
```

3.1.2 Sorting Algorithms

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements.

Problem:

- **Input:** An array of n elements $a = [a_0, a_1, \ldots, a_{n-1}]$.
- **Output:** The array a with increasing order.

Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands. Insertion sort algorithm picks elements one by one and places it to the right position where it belongs in the sorted list of elements.

```
# Imput: an array a of n elements
# Output: the array a with increasing order

def InsertSort(a):
  for i in range(1,len(a)):
    key = a[i]
    # insert a[i] into the sorted array a[0...i-1]
    j = i - 1
    while j >= 0 and a[j] > key:
      a[j+1] = a[j]
      j = j - 1
    a[j+1] = key
```

Example 28. Let $a = [4, 3, 2, 10, 12, 1, 5, 6]$. The Insertion-Sort works as follows:

- Initialization: $[4, 3, 2, 10, 12, 1, 5, 6]$

Loops:

1. $[4, \leftarrow \mathbf{3}, 2, 10, 12, 1, 5, 6] \rightarrow [3, 4, 2, 10, 12, 1, 5, 6]$
2. $[3, 4, \leftarrow \mathbf{2}, 10, 12, 1, 5, 6] \rightarrow [3, 4, 2, 10, 12, 1, 5, 6]$
3. $[2, 3, 4, \mathbf{10}, 12, 1, 5, 6] \rightarrow [2, 3, 4, 10, 12, 1, 5, 6]$
4. $[2, 3, 4, 10, \mathbf{12}, 1, 5, 6] \rightarrow [2, 3, 4, 10, 12, 1, 5, 6]$
5. $[2, 3, 4, 10, 12, \leftarrow \mathbf{1}, 5, 6] \rightarrow [1, 2, 3, 4, 10, 12, 5, 6]$
6. $[1, 2, 3, 4, 10, 12, \leftarrow \mathbf{5}, 6] \rightarrow [1, 2, 3, 4, 5, 10, 12, 6]$
7. $[1, 2, 3, 4, 5, 10, 12, \leftarrow \mathbf{6}] \rightarrow [1, 2, 3, 4, 5, 6, 10, 12]$

The output is the increasing sequence $[1, 2, 3, 4, 5, 6, 10, 12]$.

Analyzing the insertion sort.

Once again, analysis focuses on the nested loop. The outer loop executes $n - 1$ times. In the worst case, when all of the data are out of order, the inner loop iterates once on the first pass through the outer loop, twice on the second pass, and so on, for a total of

$$1 + 2 + \cdots + \frac{n-1}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

times.

Thus, we have:

Theorem 29. *The running time of the insertion sort is $T(n) = O(n^2)$.*

Remark 30. The more items in the list that are in order, the better insertion sort gets until, in the best case of a sorted list, the sort's behavior is linear. In the average case, however, insertion sort is still quadratic.

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- The subarray which is already sorted.
- Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

```python
# Imput: an array a of n elements
# Output: the array a with increasing order

def SelectionSort(a):
  for i in range(len(a)):
    k = i
    for j in range(i+1,len(a)):
      if a[j] < a[k]:
        k = j
    # exchange a[i] and a[k]
    if i != k:
      a[i], a[k] = a[k], a[i]
```

By the same argument as in the analysis of the insertion sort, we have:

Theorem 31. *The running time of the selection sort is* $T(n) = O(n^2)$.

HeapSort

Heap sort is a comparison based sorting technique based on **binary heap** data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

A **binary heap** is a *complete binary tree* where the value stored at every parent node is greater (or smaller) than the values in its two children nodes. The former is called as *max heap* and the latter is called *min heap*.

The heap can be represented by an array as follows: for each index i of the array, the index of the left child is $2i + 1$ and the index of the right child is $2i + 2$ (assuming the indexing starts at 0).

We now use the max heap for sorting problems. Note that the smallest (or largest) element in a min-heap (or max-heap) is at the root. Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.

2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.

3. Repeat above steps while size of heap is greater than 1.

▷ Heapify

```
# To heapify subtree rooted at index i.
# n is size of heap

def heapify(arr, n, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[i] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i] # swap

        # Heapify the root.
        heapify(arr, n, largest)
```

Note the the heapify will halt when we reach a leave of the tree. Since the number we do heapify is at most the length from the root to a leave and this length is at most $\log(n)$, we have the complexity of heapify is at most $\log(n)$.

▷ Heap sort:

```
def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)
```

In the lines 5 and 6 we call heapify $n/2$ times, so the number of computations is

$O((n/2)\log n)$. In each for loop (lines $9-11$) (n loops) we do one swap and one heapify, so the number of computations is $O(n(1+\log n))$. Thus, the complexity of heap sort is $O(n\log n)$.

Theorem 32. *The running time of heap sort is* $T(n) = O(n\log n)$.

▷ Test the heap sort:

```
# Driver code to test above
arr = [12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print ("Sorted array is", arr)
# Output: Sorted array is [5, 6, 7, 11, 12, 13]
```

3.2 Graph Search

Definition 33. *Graph search* (also known as graph traversal) refers to a process of visiting every vertex and edge exactly once in a well-defined order.

Remark 34. The order of visiting the vertices are important and may depend upon the algorithm or question that you are solving.

Assume that every vertex of the graph we have an order on its neighborhood. We will consider two algorithms for traversing graphs: Breadth-first search and Depth-first search, which depend on the order in which the nodes are visited. These algorithms are used as a subroutine in many other algorithms.

3.2.1 Breadth-First Search (BFS)

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

BFS traverses the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer.

2. Move to the next layer.

Consider the following diagram:

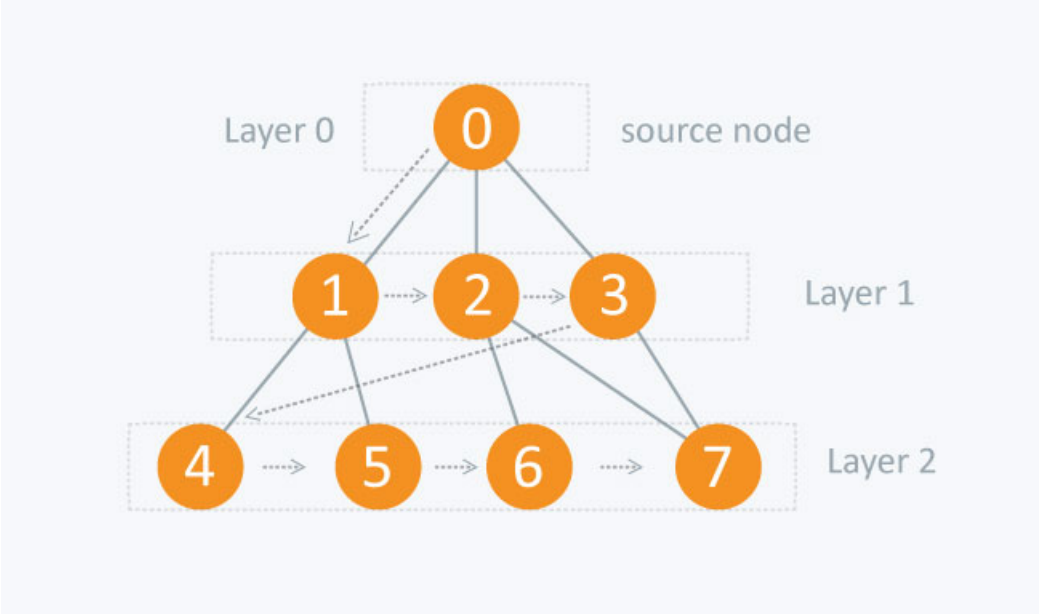


Figure 3.1: BFS.

That is, the algorithm discovers all vertices at distance k (the layer k) from the source before discovering any vertices at distance $k+1$ (the layer $k+1$). Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2 as indicated in Figure 3.4.

▷ Implement BFS:

During a traversal, it is important that you track which nodes have been visited. The most common way of tracking nodes is to mark them. We do this by coloring each node **white**, **gray**, or **black**. All nodes start out white and may later become gray and then black. A node is discovered the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If $(u, v) \in E$ and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever the search discovers a white vertex v in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree. We say that u is the **predecessor** or **parent**

of v in the **breadth-first tree**. Since a vertex is discovered at most once, it has at most one parent. Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on the simple path in the tree from the root s to vertex v, then u is an ancestor of v and v is a descendant of u.

To carries out BFS, we use a **queue** to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue follows the *First In First Out* (**FIFO**) queuing method, and therefore, the neigbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

First we need import queue:

```
import queue
```

```
# Properties of nodes: color, d, parent
# For example: {v: {'color': 'BLACK', 'd': 0, 'parent': -1}}

def BFS(G,s):
    for u in G.nodes():
        G.node[u]['color'] = 'WHITE'
        G.node[u]['d'] = -1
        G.node[u]['parent'] = None
    G.node[s]['color'] = 'GRAY'
    G.node[s]['d'] = 0
    G.node[s]['parent'] = -1
    Q = queue.Queue()
    Q.put(s)
    while not Q.empty():
        u = Q.get()
        for v in G.neighbors(u):
            if G.node[v]['color'] == 'WHITE':
                G.node[v]['color'] = 'GRAY'
                G.node[v]['d'] = G.node[u]['d']+1
                G.node[v]['parent'] = u
                Q.put(v)
        G.node[u]['color'] = 'BLACK'
```

In the algorithm above, G.node[v]['d'] is the distance from s to v, i.e. v is in layer k, where $k =$ G.node[v]['d'].

Analyzing

1. Initialization (Steps 1 to 13): the running time $O(|V|)$.

2. Maintenance: After initialization, BFS never visits a vertex twice, and thus the test in line 17 ensures that each vertex is enqueued at most once, and hence dequeued at most once.

   Because the algorithm scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $|E|$, the total time spent in scanning adjacency lists is $O(|E|)$.

3. Thus, the time running of the algorithm is $O(|V| + |E|)$.

3.2.2 Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

It explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source.

As in breadth-first search, depth-first search colors vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Besides creating a depth-first tree, depth-first search also *timestamps* each vertex. For vertex v, we assign two timestamps to it: the first timestamp *time_come* records when v is first discovered (and grayed), and the second timestamp *time_back* records when the search finishes examining v's adjacency list (and blackens v). These timestamps provide important information about the structure of the graph and are generally helpful in reasoning about the behavior of depth-first search.

```
1 # The graph G has attribite time to record timstamps
2 # Each node have properties: 'color', 'time_come', 'time_back', '
    parent'
3
4 def DFS_Visit(G,s):
```

```
    G.time += 1
    G.node[s]['color'] = 'GRAY'
    G.node[s]['time_come'] = G.time
    for v in G.neighbors(s):
        if G.node[v]['color'] == 'WHITE':
            G.node[v]['parent'] = s
            DFS_Visit(G,v)
    G.node[s]['color'] = 'BLACK'
    G.time += 1
    G.node[s]['time_back'] = G.time

def DFS(G,s):
    # Initializing for DFS
    G.time = 0
    for v in G.nodes():
        G.node[v]['color'] = 'WHITE'
        G.node[v]['time_come'] = -1
        G.node[v]['time_back'] = -1
        G.node[v]['parent'] = None

    # Carrying DFS
    DFS_Visit(G,s)
```

Analysis: By the same argument as in analyzing BFS, the running time of DFS is $O(|V| + |E|)$.

▷ Classification of edges via DFS

One interesting property of depth-first search is that the search can be used to classify the edges of the input graph $G = (V, E)$. We can define four edge types in terms of the **depth-first tree** G_π produced by a depth-first search on G:

1. *Tree edges* are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).

2. *Back edges* are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.

3. *Forward edges* are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.

4. *Cross edges* are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go

between vertices in different depth-first trees.

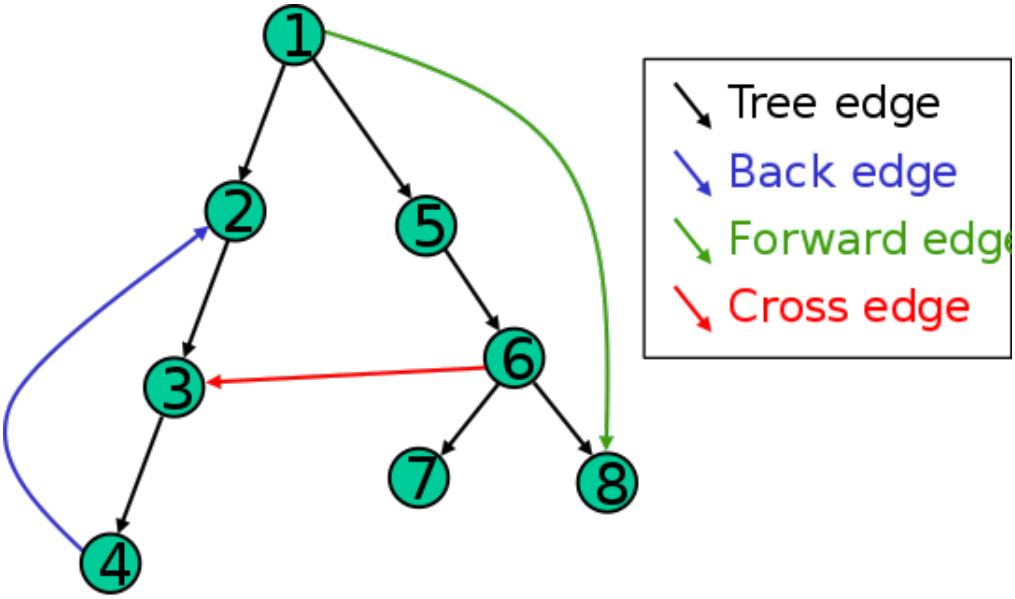


Figure 3.2: DFS works and Edge types (source 1).

The DFS algorithm has enough information to classify some edges as it encounters them. The key idea is that when we first explore an edge (u, v) the color of vertex v tells us something about the edge:

1. WHITE indicates a tree edge,
2. GRAY indicates a back edge, and
3. BLACK indicates a forward or cross edge.

3.2.3 BFS And DFS in NetworkX

Source: `https://networkx.github.io/documentation/stable/reference/algorithms/traversal.html`

Let H be a graph obtaining by choosing edges from G but these edges may equip new directions. In order to visualize graphs intuitively, we will draw G as follows:

1. Every edge of G that is not in H, draw it as usual.
2. Every edge of G that is in H, draw it thicker with color blue.

```
def DrawGraph(T, G, w = None):
    node_list = G.node()
    normalEdges = list(G.edges())
    darkEdges = list(T.edges())

    for e in darkEdges:
        if e in normalEdges:
            normalEdges.remove(e)
        if not G.is_directed():
            u,v = e
            e = (v,u)
            if e in normalEdges:
                normalEdges.remove(e)

    pos=nx.spring_layout(G)
    nx.draw_networkx_nodes(G, pos, node_color = 'r')
    nx.draw_networkx_labels(G, pos)

    nx.draw_networkx_edges(G, pos, edgelist=normalEdges,width=1)

    nx.draw_networkx_edges(T, pos, edgelist=darkEdges,width=2,
   edge_color='b')

    labels = nx.get_edge_attributes(G, w)
    nx.draw_networkx_edge_labels(G,pos,edge_labels=labels)

    plt.axis("off")
    plt.show()
```

Find the BFS tree

```
# Create a graph G
G = nx.Graph()
G.add_nodes_from(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'])
G.add_weighted_edges_from([
    ('A', 'B', 8),
    ('A', 'F', 10),
    ('A', 'H', 5),
    ('B', 'C', 4),
    ('B', 'E', 4),
    ('B', 'H', 4),
    ('B', 'F', 4),
    ('C', 'D', 3),
    ('C', 'F', 3),
```

```
    ('D', 'E', 1),
    ('D', 'F', 6),
    ('D', 'G', 3),
    ('E', 'G', 3),
    ('G', 'H', 5)
], weight = 'distance')

# Find the breadth first search tree from NetworkX
T = nx.breadth_first_search.bfs_tree(G,'A')

# Visualize
DrawGraph(T,G)
```

The result as follows:

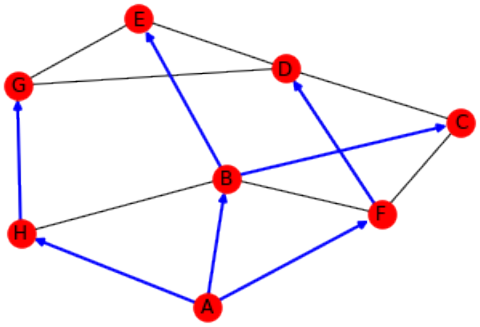


Figure 3.3: The breadth-first tree with color blue

Find the DFS tree

```
# Create a graph G

H = nx.DiGraph()
H.add_nodes_from([0, 1, 2, 3, 4, 5, 6])
H.add_weighted_edges_from([
    (0, 1,2),
    (0, 2,3),
    (1, 3,4),
    (2, 3,5),
    (3, 4,6),
    (3, 5,7),
    (4, 6,8),
    (5, 6,9),
], weight = 'distance')
# Find the breadth first search tree from NetworkX
```

```
16 T = nx.depth_first_search.dfs_tree(H,1)
17
18 # Visualize
19 DrawGraph(T,H)
```

The result as follows: two nodes 0 and 2 cannot reach from node 1.

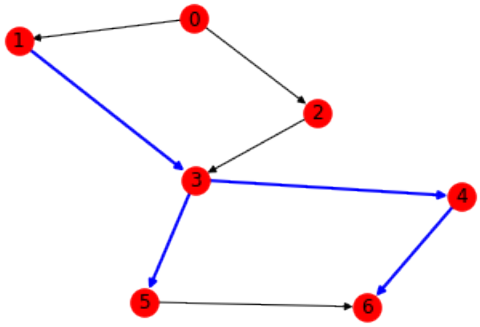


Figure 3.4: The depth-first tree with color blue

3.3 Graph optimization

In this section we present some greedy algorithms for solving several optimization problems on graphs. A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.

3.3.1 Minimal Spanning Tree

Let (G, ω) be a connected and weighted graph. A spanning tree of the graph G is a subgraph T that is a tree and connects all the nodes of G together. The weight of the spanning tree T is defined by

$$\omega(T) = \sum_{e \in E(T)} \omega(e),$$

i.e. the sum of weights given to each edge of the spanning tree.

Definition 35. A minimum spanning tree in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

In order to find minimum spanning trees: Prim's algorithm and Kruskal's algorithm.

Prim's algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding an edge of minimum weight that are incident to a vertex already in the tree and another not in it.

Kruskal's algorithm is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from a vertex, Kruskal's algorithm sorts all the edges from low weight to high and keeps adding the lowest edges, ignoring those edges that create a cycle.

We now implement Prim's algorithm. The key points are:

1. We use Priority Queue to store the weighted edges.

2. We use the list mark indicated the nodes of T in the process.

Algorithm 1 : Prim's Algorithm

Input: The weighted graph (G, ω), an initial node u.

Output: A minimum spanning tree T.

for each vertex v **do**
    mark$[v]$ = False
$T = \emptyset$
mark$[u]$ = True
Initialize a priority queue Q
for each neighbor v of u **do**
    Q.put$((u, v, \omega(u, v))$ ▷ The priority of Q is with respect to $\omega(u, v)$
while Q$\neq \emptyset$ **do**
    (a, b, w) = Q.$get()$
    if not mark$[b]$ **then**
        mark$[b]$ = True
        Adding the edge (a, b) to T
        for each neighbor v of b **do**
            if mark$[v]$ **then**
                Q.put$((b, v, \omega(b, v))$
return T

▷ Implement in Python

```
# Import queue to use PriorityQueue
import queue
```

```
3
4  def Prim(G, wt, u):
5      marks = {}
6      for v in G.nodes():
7          marks[v] = False
8      T = nx.Graph()
9      marks[u] = True
10     Q = queue.PriorityQueue()
11     for v in G.neighbors(u):
12         w = G.edges[u,v][wt]
13         Q.put([w,u,v])
14     n = G.number_of_nodes()
15     nNodes = 1
16     while nNodes < n:
17         e = Q.get()
18         u,v = e[1], e[2]
19         if not marks[v]:
20             nNodes += 1
21             marks[v] = True
22             T.add_edge(u,v)
23             for s in G.neighbors(v):
24                 if not marks[s]:
25                     w = G.edges[v,s][wt]
26                     Q.put([w,v,s])
27     return T
```

3.3.2 Shortest Path Problems

Let G be a graph with positive edge weight $\omega\colon E \to \mathbb{R}_{>0}$. We define the *length* of a path in G to be the sum of the weights of the edges of this path. Thus, for a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ in G, the length of p is

$$\omega(p) = \sum_{i=0}^{k-1} \omega(v_i, v_{i+1}).$$

Shortest path problem: Let a and z be two nodes of G which we called the *source* and the *target*, respectively. Determine the distance and a shortest path from the source vertex to the target vertex in G.

Dijkstra's algorithm finds a shortest path from the source a to the target z like BFS as follows: Start with finding a shortest path from a to a first node by visiting its neighbors, then a shortest path from a to a second node by exploring visited nodes

or visiting new nodes from neighbors of visited nodes, and so on, until a shortest path from a to z is found.

We now give the details of Dijkstra's algorithm. It begins by labeling a with 0 and the other vertices with ∞. We use the notation $L_0(a) = 0$ and $L_0(v) = \infty$ for these labels before any iterations have taken place (the subscript 0 stands for the "0th" iteration). These labels are the lengths of shortest paths from a to the vertices, where the paths contain only the vertex a. (Because no path from a to a vertex different from a exists, ∞ is the length of a shortest path between a and this vertex.)

Dijkstra's algorithm proceeds by forming a distinguished set of vertices. Let S_k denote this set after k iterations of the labeling procedure. We begin with S_0. The set S_k is formed from S_{k-1} by adding a vertex u not in S_{k-1} with the smallest label.

Once u is added to S_k, we update the labels of all vertices not in S_k, so that $L_k(v)$, the label of the vertex v at the kth stage, is the length of a shortest path from a to v that contains vertices only in S_k (that is, vertices that were already in the distinguished set together with u). Note that the way we choose the vertex u to add to S_k at each step is an optimal choice at each step, making this a greedy algorithm.

Let v be a vertex not in S_k. To update the label of v, note that $L_k(v)$ is the length of a shortest path from a to v containing only vertices in S_k. The updating can be carried out efficiently when this observation is used: A shortest path from a to v containing only elements of S_k is either a shortest path from a to v that contains only elements of S_{k-1} (that is, the distinguished vertices not including u), or it is a shortest path from a to u at the $(k-1)$st stage with the edge $\{u, v\}$ added. In other words,

$$L_k(a, v) = \min\{L_{k-1}(a, v), L_{k-1}(a, u) + \omega(a, u)\}.$$

This procedure is iterated by successively adding vertices to the distinguished set until z is added. When z is added to the distinguished set, its label is the length of a shortest path from a to z.

The **correctness of Dijkstra's algorithm** is a consequence of the following observations: at the k-th iteration (the proof is by induction on k)

- the label of every node v in S_k is the length of a shortest path from a to this vertex, and

- the label of every vertex not in S_k is the length of a shortest path from a to this vertex that contains only (besides the vertex itself) vertices in S_k.

Algorithm 2 : DIJKSTRA'S ALGORITHM

Input: The weighted graph (G, ω), a source s, a target t.

Output: A shortest path from a to t.

1: **for** each node v **do**
2: $\quad L(v) = \infty$
3: $\quad$ previous(v) = undefined
4: $L(a) = 0$
5: $S = \emptyset$
6: **while** $z \notin S$ **do**
7: $\quad$ Searching for a node u not in S with $L(u)$ minimal
8: $\quad S = S \cup \{u\}$
9: $\quad$ **for** each vertex $v \in N(u) \setminus S$ **do**
10: $\quad\quad$ **if** $L(u) + \omega(u, v) < L(v)$ **then**
11: $\quad\quad\quad L(v) = L(u) + \omega(u, v)$
12: $\quad\quad\quad$ previous$(v) = u$
13: **return** previous

Remark 36. When implementing Dijkstra algorithm we use Priority Queue for carrying out Step 6 in the algorithm above.

A shortest path from a to z is encoded in the the list previous, where previous(v) is the preceding node of v in the shortest path.

▷ Implement in Python

```
# Input: weighted graph (G,wt); source s, target t
# Output: the shortest path, and its length

def Dijkstra(G,wt,s,z):
    previous = {}
    optimal = {}
    L = {}
    for v in G.nodes():
        previous[v] = None
        optimal[v] = False
        L[v] = -1

    L[s] = 0
    Q = queue.PriorityQueue()
    Q.put([L[s],s])

```

```
    while not optimal[z]:
        u = Q.get()[1]
        optimal[u] = True
        for v in G.neighbors(u):
            if not optimal[v]:
                d = L[u] + G[u][v][wt]
                if L[v] == -1 or d < L[v]:
                    L[v] = d
                    previous[v] = u
                    Q.put([L[v],v])
    p = []
    v = z
    while v != s:
        p.append(v)
        v = previous[v]
    p.append(s)
    p.reverse()
    return p, L[z]
```

3.3.3 Flow in Networks

Definition 37. A *network* $N(G, s, t, c)$ consists of the following data:

1. A finite digraph $G = (V, E)$ with no self-loops and no parallel edges.

2. Two vertices s and t are specified; s is called the *source*; and t, the *sink*.

3. The *capacity* function $c\colon E \to \mathbb{R}_{>0}$. The positive real numbers, $c(e)$, is called the capacity of edge e.

For every vertex $v \in V$, let let $\alpha(v)$ denote the set of edges that enter v in G. Similarly, let $\beta(v)$ denote the set of edges that emanate from v.

A *flow function*, $f\colon E \to \mathbb{R}$, is an assignment of a real number $f(e)$ to each edge e such that the following two conditions hold:

- The Edge Rule: for every edge $e \in E$, $0 \leqslant f(e) \leqslant c(e)$.

- The Vertex Rule: For every vertex $v \in V \setminus \{s, t\}$,

$$\sum_{e \in \alpha(v)} f(e) = \sum_{e \in \beta(v)} f(e).$$

The *total flow*, of f in N, is defined by

$$|f| = \sum_{e \in \alpha(t)} f(e) - \sum_{e \in \beta(t)} f(e).$$

Namely, $|f|$ is the net sum of flow into the sink.

Maximum flow problem: Find a feasible flow through a network that obtains the maximum total flow.

Given a set $S \subset V$, let $\bar{S} = V \setminus S$. In the following, we shall discuss sets S, such that $s \in S$ and $t \in \bar{S}$. Also, $(S; \bar{S})$ denotes the set of edges which are directed from a vertex in S to a vertex in $\bar{S}$; this set of edges is called a *forward cut*. The set $(\bar{S}; S)$ is similarly defined, and is called a *backward cut*. The union of $(S; \bar{S})$ and $(\overline{S}; S)$ is called the *cut* defined by S. We define the capacity of the cut by

$$c(S) = \sum_{e \in (S, \bar{S})} c(e).$$

By definition, the total flow $|f|$ is measured at the sink. In fact, it can be measured at any cut.

Theorem 38. *For every $s \in S \subseteq V \setminus \{t\}$ and every flow function f, the following holds:*

$$|f| = \sum_{e \in (S; \bar{S})} f(e) - \sum_{e \in (\bar{S}; S)} f(e).$$

From this theorem we deduce that

$$|f| = \sum_{e \in (S; \bar{S})} f(e) - \sum_{e \in (\bar{S}; S)} f(e) \leqslant \sum_{e \in (S; \bar{S})} f(e) \leqslant \sum_{e \in (S; \bar{S})} c(e) = c(S).$$

This yields the following corollary.

Corrollary 39. *If $|f| = c(S)$ for some set S with $s \in S \subseteq V \setminus \{t\}$, then f is maximum.*

▷ Ford-Fulkerson Algorithm

Ford-Fulkerson Algorithm is a greedy algorithm that increases a flow f on the network N by using augmented path.

If the direction of the edges of G is ignored, a simple path P from s to t is called an *augmenting path* from s to t if there is a real number $\Delta > 0$, such that

- If an edge e of P is directed in the direction from s to t, then $c(e) \geqslant f(e) + \Delta$.
- If an edge e of P is directed in the opposite direction from s to t, then $f(e) \geqslant \Delta$.

Assume P is an augmented path, we define the function $f' \colon E \to \mathbb{R}$ as follows: for an edge e,

$$f'(e) = \begin{cases} f(e) & \text{if } e\text{is not in } P, \\ f(e) + \Delta & \text{if } e \text{ is an edge of } P \text{ with direction from } s \text{ to } t, \\ f(e) - \Delta & \text{otherwise.} \end{cases}$$

Then, f' is a flow on the network N with $|f'| = |f| + \Delta$, so that f' improves f.

The idea of augmented paths leads to Ford-Fulkerson algorithm:

Algorithm 3 : Ford-Fulkerson algorithm

Input: The weighted graph (G, ω), a source s, a sink t.

Output: A maximum flow f.

1: Start with initial flow as 0
2: **while** *there is a augmenting path from source to sink* **do**
3: Add this path-flow to flow.
4: **return** flow

Assume that the Ford - Fulkerson algorithm terminates. Then, we can not reach t by an augmented path from s. Let S be the set of all nodes that can be reached by augmented paths from s. It follows that $s \in S$ and $t \in \bar{S}$.

Observe that $f(e) = c(e)$ if $e \in (S; \bar{S})$, and $f(e) = 0$ if $e \in (\bar{S}; S)$. Together with Theorem 38, this fact gives

$$|f| = \sum_{e \in (S;\bar{S})} f(e) - \sum_{e \in (\bar{S};S)} f(e) = \sum_{e \in (S;\bar{S})} f(e) = c(S),$$

and thus f is a maximum flow by Corollary 39. This argument asserts the correctness of Ford-Fulkerson algorithm.

▷ Implement in Python

In the implementation of Ford-Fulkerson algorithm below, we use Dijkstra's algorithm for finding augmenting paths, this is the so-called Edmond-Karp algorithm.

```
# Using Diijstra's algorithm to find augmented paths

def AugmentedPath(G,wt,fw, ucp, s,z):
    L = {}
    previous = {}
    optimal = {}
```

```
    for v in G.nodes():
        L[v] = 0
        previous[v] = None
        optimal[v] = False
    L[s] = -ucp

    Q = queue.PriorityQueue()
    Q.put([L[s],s])

    while not optimal[z]:
        if Q.empty():
            return None, 0
        u = Q.get()[1]
        optimal[u] = True
        for v in G.successors(u):
            if not optimal[v]:
                d = max(L[u], G[u][v][fw]-G[u][v][wt])
                if d < L[v]:
                    L[v] = d
                    previous[v] = u
                    Q.put([L[v],v])
        for v in G.predecessors(u):
            if not optimal[v]:
                d = max(L[u],-G[v][u][fw])
                if d < L[v]:
                    L[v] = d
                    previous[v] = u
                    Q.put([L[v],v])
    p = []
    v = z
    while v != s:
        p.append(v)
        v = previous[v]
    p.append(s)
    p.reverse()
    return p, -L[z]

# Ford-Fulkerson algorithm
def Ford_Fulkerson(G,wt,fw,s,t):

    # Initialize flow
    for e in G.edges():
        G.edges[e][fw] = 0

```

```
    # The upper bound for augmented flows
    ucp = 0
    for v in G.successors(s):
        if ucp < G.edges[s,v][wt]:
            ucp = G.edges[s,v][wt]
    for v in G.predecessors(s):
        if ucp < G.edges[v,s][wt]:
            ucp = G.edges[v,s][wt]

    # Find feasible augmented flows and add to the flow
    p, d = AugmentedPath(G,wt,fw,ucp, s,t)
    while d > 0:
        for i in range(len(p)-1):
            u, v = p[i], p[i+1]
            if G.has_edge(u,v):
                G.edges[u,v][fw] += d
            else:
                G.edges[v,u][fw] -= d
        p, d = AugmentedPath(G,wt,fw,ucp, s,t)
```

Chapter 4

Numerical analysis

Source:

R. L. Burden and J. D. Faires, Numerical Analysis, 9th ed., Cengage Learning, 2011.

4.1 Vectors and Matrices

Let $\mathbb{R}$ denote the set of *real numbers*. Let m and n be positive integers.

Definition 40. An $m \times n$ (m by n) matrix is a rectangular array of elements with m rows and n columns in which not only is the value of an element important, but also its position in the array.

The notation for an $m \times n$ matrix will be a capital letter such as A for the matrix and lowercase letters with double subscripts, such as a_{ij}, to refer to the entry at the intersection of the ith row and jth column; that is,

$$A = [a_{ij}] = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

We will use the notation $A \in \mathbb{R}^{m \times n}$ to indicate that A is an $m \times n$ matrix with real entries.

The $n \times 1$ matrix

$$A = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{n1} \end{bmatrix}$$

is called an **n-dimensional column vector**. Usually the unnecessary subscripts are omitted for vectors, and a boldface lowercase letter is used for notation. Thus,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

denotes a column vector, and $\mathbf{y} = [y_1\ y_2 \dots y_n]$ denote an **n-dimensional row vector**. In addition, row vectors often have commas inserted between the entries to make the separation clearer. So you might see y written as $\mathbf{y} = [y_1, y_2, \dots, y_n]$.

Let $\mathbb{R}^{m\times n}$ denote the set of $m \times n$ matrices with entries in $\mathbb{R}$. Define operations on $\mathbb{R}^{m\times n}$ as follows : for $A = [a_{ij}], B = [b_{ij}] \in \mathbb{R}^{m\times n}$, and $\lambda \in \mathbb{R}$

- *Addition*: $A + B = [(a_{ij} + b_{ij}]$, and
- *Multiplication* with *scalar*: $\lambda A = [\lambda a_{ij}]$.

Then, $\mathbb{R}^{m\times n}$ is an mn dimensional vector space over $\mathbb{R}$. The vector space $\mathbb{R}^{n\times 1}$ of column vectors with n components will simply denote by $\mathbb{R}^n$ as usual.

For $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$, we define the *product* of A and B by

$$C = AB$$

where $C = [c_{ij}] \in \mathbb{R}^{m\times p}$ with

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

▷ In NumPy:

- A vector is a 1D array, and a matrix is a 2D array.
- The addition of two matrices A and B is $A + B$.
- The multiplication of the matrix A and a scalar λ is $\lambda * A$.
- The product of a matrix A with a matrix B is $A.\text{dot}(B)$.

4.2 Systems of Linear Equations

▷ Linear systems

A system of m linear equations in n variables is a set of m equations, each of which is linear in the same n variables:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = b_2 \\ \vdots & \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n & = b_m \end{cases} \tag{4.1}$$

A **solution** of the linear system (4.3) is a tuple $(s_1, s_2, \ldots, s_n)$ of numbers that makes each equation a true statement when the values s_1, s_2, ... , s_n are substituted for $x_1, x_2, \ldots, x_n$, respectively. The set of all solutions of a linear system is called the **solution set** of the system.

▷ Matrix form of linear systems

An $m \times (n+1)$-matrix can be used to represent the linear system (4.3) by first constructing

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{and } \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

then the system (4.3) can be represented by the matrix

$$\overline{A} = (A, b) = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{array}\right].$$

The matrix A is called the coefficient matrix and $\overline{A}$ is called the augmented matrix of the linear system (4.3), respectively.

The system (4.3) now can be written in matrix form:

$$A\mathbf{x} = \mathbf{b}. \tag{4.2}$$

4.2.1 Gauss Elimination

We now present Gaussian elimination, also known as row reduction, for solving the system of linear equations above.

Elementary row operations

There are three kinds of elementary row operations on matrices:

1. Multiply a row by a nonzero scalar.
2. Add to one row a scalar multiple of another.
3. Swap the positions of two rows.

Remark 41. Elementary row operations on the augmented matrix of a linear system change this linear system to another do not change solution set.

Row echelon forms

A matrix is said to be in **row echelon form** if it satisfies the following two conditions:

1. All zero rows are gathered near the bottom.
2. The first nonzero entry of a row, called the *leading entry* of that row, is ahead of the first nonzero entry of the next row.

Gauss Elimination

1. Write the augmented matrix of the system of linear equations.
2. Use elementary row operations to reduce the augmented matrix in row-echelon form.
3. Write the system of linear equations corresponding to the matrix in row-echelon form, and use back-substitution to find the solution.

Implement

We now implement Gauss Elimination for solving the system of linear equations:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = b_2 \\ \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n & = b_n \end{cases} \tag{4.3}$$

or in matrix form $A\mathbf{x} = \mathbf{b}$.

In practice, we often solve this linear system in the case it has unique solution. Using Gauss elimination reduces its augmented matrix to the upper triangle matrix:

$$\left[\begin{array}{cccc|c} b_{11} & b_{12} & \cdots & b_{1n} & b_{1,n+1} \\ 0 & b_{22} & \cdots & b_{2n} & b_{2,n+1} \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & b_{nn} & b_{n,n+1} \end{array}\right]$$

with $b_{ii} \neq 0$ for $i = 1, \ldots, n$.

Thus, the linear system is equivalent to

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + \cdots + b_{1n}x_n &= b_{1,n+1} \\ b_{22}x_2 + \cdots + b_{2n}x_n &= b_{2,n+1} \\ &\vdots \\ b_{nn}x_n &= b_{n,n+1} \end{aligned}$$

so *backward substitution* can be performed. Solving the nth equation for x_n gives

$$x_n = \frac{b_{n,n+1}}{b_{nn}}.$$

Solving the $(n-1)$st equation for x_{n-1} and using the known value for x_n yields

$$x_{n-1} = \frac{b_{n,n+1} - b_{n-1,n}x_n}{b_{n-1,n-1}}.$$

Continuing this process, we obtain

$$x_i = \frac{b_{i,n+1} - \sum_{j=i+1}^{n} b_{ij}x_j}{b_{ii}}.$$

Algorithm 4 : Gaussian Elimination

Input: number of unknowns and equations n, augmented $n \times (n+1)$ matrix $A = [a_{ij}]$

Output: solution $x_1, x_2, \ldots, x_n$ or message that the linear system has no unique solution

1: **for** $i = 1$ to n **do** ▷ Elimination process
2: Let p be the smallest integer with $i \leqslant p \leqslant n$ and $a_{pi} \neq 0$
3: **if** no integer p can be found **then**
4: **return** 'no unique solution exists'
5: **if** $p \neq i$ **then**
6: Exchange ith row and pth row
7: **for** $j = i + 1$ to n **do**
8: $m_{ji} = a_{ji}/a_{ii}$
9: Subtract from the jth row a multiple of m_{ji} with the ith row
10: $x_n = a_{n,n+1}/a_{nn}$ ▷ Start backward substitution
11: **for** $i = n - 1$ downto 1 **do**
12: $x_i = \left(a_{i,n+1} - \sum_{j=i+1}^{n} a_{ij}x_j\right)/a_{ii}$
13: **return** $(x_1, x_2, \ldots, x_n)$ ▷ Procedure completed successfully

Remark 42. The time running of the Gaussian Elimination Algorithm is $T(n) = O(n^3)$.

In Python, note that the array has index staring at 0.

```
#  Gaussian Elimination
#  Input: the aumented matrix [A | b] of the system Ax = b
#  Output: solution x_1, x_2, ..., x_n, or None if the system has no unique
    solution

def GaussElimination(A):
  n = len(A)  # the number of rows
  for i in range(n):  # Elimination process
    #  Find the smallest integer p with i <= p <= n-1 and a[p][i] != 0
    p = i
    while p < n:
      if A[p,i] != 0:
        break
      p += 1
    if p == n:
      return None
    if p != i:
      # swap the ith row and the pth row
      for j in range(i, n+1):
        A[i,j], A[p,j] = A[p,j], A[i,j]
    for j in range(i+1,n):
      m = A[j,i]/A[i,i]
      A[j] -= m * A[i]

    # Start backward substitution
    x = [0] * n
    x[n-1] = A[n-1,n] / A[n-1, n-1]
    for i in range(n-2,-1,-1):
      tmp = 0
      for j in range(i+1, n):
        tmp += A[i, j] * x[j]
      x[i] = (A[i, n] - tmp)/A[i,i]
    return x
```

Remark 43. We can solve systems of linear equations by using module NumPy.

Source: `https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html`

```
# Solve the system of equations 3 * x_0 + x_1 = 9 and x_0 + 2 * x_1 = 8:
```

```
import numpy as np

# Give A and b
A = np.array([[3,1], [1,2]])
b = np.array([9,8])

# Solve Ax = b
x = np.linalg.solve(A,b)

print(x) # Output: array([2.,  3.])
```

4.2.2 Iterative techniques

For large systems with a high percentage of 0 entries (sparse systems), the direct techniques such as Gaussian elimination is often not efficient. In this case, the iterative methods such as the Jacobi and the Gauss-Seidel iterative methods are efficient in terms of both computer storage and computation. Systems of this type arise frequently, for example, in the numerical solution of boundary-value problems and partial-differential equations.

An iterative technique to solve the $n \times n$ linear system $A\mathbf{x} = \mathbf{b}$ starts with an initial approximation $\mathbf{x}^{(0)}$ to the solution $\mathbf{x}$ and generates a sequence of vectors $\{\mathbf{x}^{(k)}\}_{k=0}^{\infty}$ that converges to $\mathbf{x}$. One prototype for this technique is the Banach fixed-point theorem.

Fixed-Point Iteration

Let (X, d) be a complete metric space and $f\colon X \to X$. A point x in X is called a *fixed-point* of f if $f(x) = x$.

Definition 44. f is called a *contraction mapping* on X if there exists $q \in [0, 1)$ such that

$$d(f(x), f(y)) \leqslant qd(x, y)$$

for all $x, y \in X$. The number q is called a *Lipschitz constant* for f.

Banach Fixed Point Theorem: Let f be a contraction mapping on X. Then, f has a unique fixed-point x^* in X.

Iterative method for finding x^*:

1. Start with an arbitrary element x_0 in X.
2. Define a sequence $\{x_n\}$ by $x_n = f(x_{n-1})$ for $n \geqslant 1$.
3. Then $x_n \to x^*$ as $n \to \infty$.

The speed of convergence: for any $n \geqslant 1$, we have

1. $d(x^*, x_n) \leqslant \dfrac{q^n}{1-q} d(x_1, x_0)$.
2. $d(x^*, x_{n+1}) \leqslant \dfrac{q}{1-q} d(x_{n+1}, x_n)$.
3. $d(x^*, x_{n+1}) \leqslant q d(x^*, x_n)$.

Algorithm 5 : FINDING THE FIXED-POINT

Input: the map f; an initial point x_0; a Lipschitz constant q for f; the tolerance TOL

Output: An approximation x_n of the fixed-point of f with tolerance TOL

1: $y = x_0$
2: $x = f(y)$
3: **while** $\dfrac{q}{1-q} d(x, z) > TOL$ **do** ▷ LOOP
4: $\quad y = x$
5: $\quad x = f(x)$
6: **return** x

Norms on $\mathbb{R}^n$

To define a distance in $\mathbb{R}^n$ we use the notion of a *norm*, which is the generalization of the absolute value on $\mathbb{R}$, the set of real numbers.

Definition 45. A **vector norm** in $\mathbb{R}^n$ is a function $\|\cdot\|$, from $\mathbb{R}^n$ to $\mathbb{R}$ with the following properties:

(i) $\|\mathbf{x}\| \geqslant 0$ for all $\mathbf{x} \in \mathbb{R}^n$;

(ii) $\|\mathbf{x}\| = 0$ in and only if $\mathbf{x} = 0$;

(iii) $\|\alpha \mathbf{x}\| = |\alpha| \, \|\mathbf{x}\|$ for all $\alpha \in \mathbb{R}$ and $\mathbf{x} \in \mathbb{R}^n$;

(iv) $\|\mathbf{x} + \mathbf{y}\| \leqslant \|\mathbf{x}\| + \|\mathbf{y}\|$.

In practice, the following norm is useful. For any $1 \leqslant p \leqslant \infty$, we define the ℓ_p norm of the vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$ by

$$\|x\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{1/p}.$$

In particular,

1. $\|x\|_1 = |x_1| + |x_2| + \cdots + |x_n|$.

2. $\|x\|_2 = (x_1^2 + x_2^2 + \ldots + x_n^2)^{1/2}$ (which is Euclidean norm).

3. $\|x\|_\infty = \max\{|x_1|, |x_2|, \ldots, |x_n|\}$.

Example 46. Determine the ℓ_2 norm and the ℓ_∞ norm of the vector $\mathbf{x} = (-1, 1, -2)^T$.

Solution. The vector $\mathbf{x} = (-1, 1, -2)^T$ in $\mathbb{R}^3$ has norms

$$\|\mathbf{x}\|_2 = \sqrt{(-1)^2 + (1)^2 + (-2)^2} = \sqrt{6}$$

and

$$\|\mathbf{x}\|_\infty = \max\{|-1|, |1|, |-2|\} = 2.$$

□

Definition 47. If $\|\cdot\|$ is a norm in $\mathbb{R}^n$, we can define distance between $\mathbf{x}$ and $\mathbf{y}$ by

$$\|\mathbf{x} - \mathbf{y}\|.$$

In particular, the ℓ_p distance between these vectors is

$$\|\mathbf{x} - \mathbf{y}\|_p = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}.$$

Example 48. The linear system

$$\begin{cases} 3.3330x_1 + 15920x_2 - 10.333x_3 = 15913 \\ 2.2220x_1 + 16.710x_2 + 9.6120x_3 = 28.544 \\ 1.5611x_1 + 5.1791x_2 + 1.6852x_3 = 8.4254 \end{cases}$$

has the exact solution $\mathbf{x} = (1, 1, 1)^T$, and Gaussian elimination performed using five-digit rounding arithmetic, produces the approximate solution

$$\tilde{\mathbf{x}} = (\tilde{x}_1, \tilde{x}_2, \tilde{x}_3)^T = (1.2001, 0.99991, 0.92538)^T.$$

Determine the ℓ_2 and ℓ_∞ distances between the exact and approximate solutions.

Solution. Measurements of $\mathbf{x} - \tilde{\mathbf{x}}$ are given by

$$\begin{aligned} \|\mathbf{x} - \tilde{\mathbf{x}}\|_2 &= [(1 - 1.2001)^2 + (1 - 0.99991)^2 + (1 - 0.92538)^2]^{1/2} \\ &= [(-0.2001)^2 + (0.00009)^2 + (0.07462)^2]^{1/2} = 0.21356 \end{aligned}$$

and

$$\begin{aligned} \|\mathbf{x} - \tilde{\mathbf{x}}\|_\infty &= \max\{|1 - 1.2001|, |1 - 0.99991|, |1 - 0.92538|\} \\ &= \max\{0.2001, 0.00009, 0.07462\} = 0.2001. \end{aligned}$$

Although the components $\tilde{x}_2$ and $\tilde{x}_3$ are good approximations to x_2 and x_3, the component $\tilde{x}_1$ is a poor approximation to x_1, and $|x_1 - \tilde{x}_1|$ dominates both norms. □

Definition 49 (Matrix Norms)**.** A matrix norm on $\mathbb{R}^{n\times n}$ is a norm $\|\cdot\|$ on $\mathbb{R}^{n\times n}$ satisfying

$$\|AB\| \leqslant \|A\|\,\|B\|$$

for all $A, B \in \mathbb{R}^{n\times n}$.

We often consider matrix norm obtained from a norm on $\mathbb{R}^n$.

Theorem 50. *Let $\|\cdot\|$ be a norm on $\mathbb{R}^n$. Then,*

$$\|A\| = \max_{\|\mathbf{x}\|=1} \|A\mathbf{x}\|$$

is a matrix norm on $\mathbb{R}^{n\times n}$.

Matrix norms defined by vector norms as in Theorem 50 are called the *natural*, or *induced*, matrix norm associated with the vector norm. In this text, all matrix norms will be assumed to be natural matrix norms unless specified otherwise.

Example 51. Let $\|\cdot\|_p$ be a matrix norm on $\mathbb{R}^{n\times n}$ defined by the norm ℓ_p on $\mathbb{R}^n$. If $A = [a_{ij}] \in \mathbb{R}^{n\times n}$, then

1. The ℓ_1 norm of A:
$$\|A\|_1 = \max_{1\leqslant j\leqslant n} \sum_{i=1} |a_{ij}|.$$

2. The ℓ_2 norm of A:
$$\|A\|_2 = \max\left\{\lambda \mid \lambda \text{ is an eigenvalue of } AA^T\right\}$$

3. The ℓ_∞ norm of A:
$$\|A\|_\infty = \max_{1\leqslant i\leqslant n} \sum_{j=1} |a_{ij}|.$$

Jacobi's Method

In order to solve the linear system $A\mathbf{x} = b$, in its ith equation

$$\sum_{j=1}^{n} a_{ij}x_j = b_i,$$

solve for the variable of x_i while assuming the other variable of x_j remain fixed. This gives

$$x_i = \sum_{\substack{j=1\\ j\neq i}}^{n} \left(-\frac{a_{ij}x_j}{a_{ii}}\right) + \frac{b_i}{a_{ii}}. \tag{4.4}$$

Let

$$T = \begin{bmatrix} 0 & -\frac{a_{12}}{a_{11}} & \cdots & -\frac{a_{1n}}{a_{11}} \\ -\frac{a_{21}}{a_{22}} & 0 & \cdots & -\frac{a_{2n}}{a_{22}} \\ \vdots & \vdots & \ddots & \vdots \\ -\frac{a_{n1}}{a_{nn}} & -\frac{a_{n2}}{a_{nn}} & \cdots & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{c} = \begin{bmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} \\ \vdots \\ \frac{b_n}{a_{nn}} \end{bmatrix}$$

Then, the linear system $A\mathbf{x} = \mathbf{b}$ can be written as $\mathbf{x} = T\mathbf{x} + \mathbf{c}$, so that $\mathbf{x}$ is a fixed point of the map $f\colon \mathbb{R}^n \to \mathbb{R}^n$, $f(\mathbf{x}) = T\mathbf{x} + \mathbf{c}$. Using the algorithm 5 to find this point is Jacobi's method for solving the linear system $A\mathbf{x} = \mathbf{b}$:

Start with $\mathbf{x}^{(0)}$ arbitrarily, we construct the sequence $\{\mathbf{x}^{(k)}\}$ by

$$x_i^{(k)} = \frac{1}{a_{ii}} \left[-\sum_{\substack{j=1 \\ j \neq i}}^{n} (a_{ij} x_j^{(k-1)}) + b_i \right], \text{ for } i = 1, 2, \ldots, n. \tag{4.5}$$

where $x_i^{(k)}$ is the ith component of $\mathbf{x}^{(k)}$.

Remark 52. For $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, we have

$$\|f(\mathbf{x}) - f(\mathbf{y})\|_\infty = \|T(x - y)\|_\infty \leqslant \|T\|_\infty \|\mathbf{x} - \mathbf{y}\|_\infty .$$

Note also that

$$|T|_\infty = \max \left\{ \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \mid i = 1, \ldots, n \right\},$$

so the Jacobi method will converge if the matrix A is diagonally dominant, that means

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|$$

for $i = 1, \ldots, n$.

Implement Jacobi's Method

```
# Input numpy

import numpy as np

# Input: mtrix A, vector b, tolerance; maximum number of iterations N

def jacobi(A, b, x_init, TOL=1e-10, N = 500):
```

```
    n = len(A)
    x0 = x_init.copy()
    x = np.zeros(n, dtype = float)
    for k in range(N):
        for i in range(n):
            x[i] = b[i]
            for j in range(i):
                x[i] -= A[i,j]*x0[j]
            for j in range(i+1,n):
                x[i] -= A[i,j]*x0[j]
            x[i] /= A[i,i]
        if np.linalg.norm(x-x0) < TOL:
            return x
        for i in range(n):
            x0[i] = x[i]
    return x

# problem data
A = np.array([
    [5, 2, 1, 1],
    [2, 6, 2, 1],
    [1, 2, 7, 1],
    [1, 1, 2, 8]
])
b = np.array([29, 31, 26, 19])

# you can choose any starting vector
x_init = np.zeros(len(b))
x = jacobi(A, b, x_init)
print(x)
# Ouput: [3.99275362 2.95410628 2.16183575 0.96618357]
```

The Gauss-Seidel Method

A possible improvement in Jacobi's method can be seen by reconsidering Equation (4.5). The components $\mathbf{x}_j^{(k-1)}$ is used to compute the components $x_i^{(k)}$ of $\mathbf{x}^{(k)}$. But, for $i > 1$, the components $x_1^{(k)}, \ldots, x_{i-1}^{(k)}$ have already computed are expected to be better approximations to the actual solutions $x_1, \ldots, x_{i-1}$. It seem reasonable, then, compute $x_i^{(k)}$ using these most calculated values. That is, to use

$$x_i^{(k)} = \frac{1}{a_{ii}} \left[-\sum_{j=1}^{i-1} (a_{ij} x_j^{(k)}) - \sum_{j=i+1}^{n} (a_{ij} x_j^{(k-1)}) + b_i \right] \tag{4.6}$$

for each $i = 1, 2, \ldots, n$ instead of Equation (4.5). This modification is called the

Gauss-Seidel method.

Implement The Gauss-Seidel Method

```
# Input numpy

import numpy as np

# Input: mtrix A, vector b, tolerance; maximum number of iterations N

def Gauss_Seidel(A, b, x_init, TOL=1e-10, N = 500):
    n = len(A)
    x0 = x_init.copy()
    x = np.zeros(n, dtype = float)
    for k in range(N):
        for i in range(n):
            x[i] = b[i]
            for j in range(i):
                x[i] -= A[i,j]*x[j]
            for j in range(i+1,n):
                x[i] -= A[i,j] * x0[j]
            x[i] /= A[i,i]
        if np.linalg.norm(x-x0) < TOL:
            return x
        for i in range(n):
            x0[i] = x[i]
    return x

# problem data
A = np.array([
    [5, 2, 1, 1],
    [2, 6, 2, 1],
    [1, 2, 7, 1],
    [1, 1, 2, 8]
])
b = np.array([29, 31, 26, 19])

# you can choose any starting vector
x_init = np.zeros(len(b))
x = Gauss_Seidel(A, b, x_init)
print(x)
# Ouput: [3.99275362 2.95410628 2.16183575 0.96618357]
```

4.3 Nonlinear Equations in One Variable

A *root-finding algorithm* is a numerical method or algorithm for finding a value x^* such that $f(x^*) = 0$, for a given function f. We call x^* to be a *root* of Equation $f(x) = 0$, or a *zero* of f.

4.3.1 Bisection method

Suppose f is a continuous function defined on the interval $[a, b]$, with $f(a)$ and $f(b)$ of opposite sign. The *Intermediate Value Theorem* implies that a number x exists in (a, b) with $f(x) = 0$.

The method calls for a repeated halving (or bisecting) of subintervals of $[a, b]$ and, at each step, locating the half containing a zero.

To begin, set $a_1 = a$ and $b_1 = b$, and let c_1 be the midpoint of $[a, b]$; that is,

$$c_1 = a_1 + \frac{b_1 - a_1}{2} = \frac{a_1 + b_1}{2}.$$

1. if $f(c_1) = 0$, the $x = c_1$ is a root, and we are done.

2. if $f(c_1) \neq 0$, then $f(c_1)$ has the same sign as $f(a_1)$ or $f(b_1)$.

    - if $f(a_1)$ and $f(c_1)$ has the opposite sign, then f has a zero in (a_1, c_1). Set $a_2 = a_1$ and $b_2 = c_1$.
    - if $f(a_1)$ and $f(c_1)$ has the opposite sign, then f has a zero in (c_1, b_1). Set $a_2 = c_1$ and $b_2 = b_1$.

Then reapply the process to the interval $[a_2, b_2]$. After n iterations we come to the interval $[a_n, b_n]$ containing a zero of f. If we take an approximation $x = c_n$ for a zero of f, then the tolerance is $(b_n - a_n)/2 = (b - a)/2^{n+1}$.

Implement Bisection Method

```
def BisectionMethod(f,a,b,TOL = 1.0e-10):
    f1 = f(a)
    f2 = f(b)
    while b-a > TOL:
        c = (a+b)/2.0
        fc = f(c)
        if fc == 0.0:
            return c
        if f1 * fc < 0:
            b = c
```

```
            f2 = fc
        else:
            a = c
            f1 = fc
    return (a+b)/2.0

# Find a root of x^3 - x^2 + x + 1 = 0 on [0,1] with tolerance 10^-10
x = BisectionMethod(lambda x:2*x**3-x**2+x-1, 0, 1)
print(x)
# Output: 0.7389836215006653
```

4.3.2 Newton's method

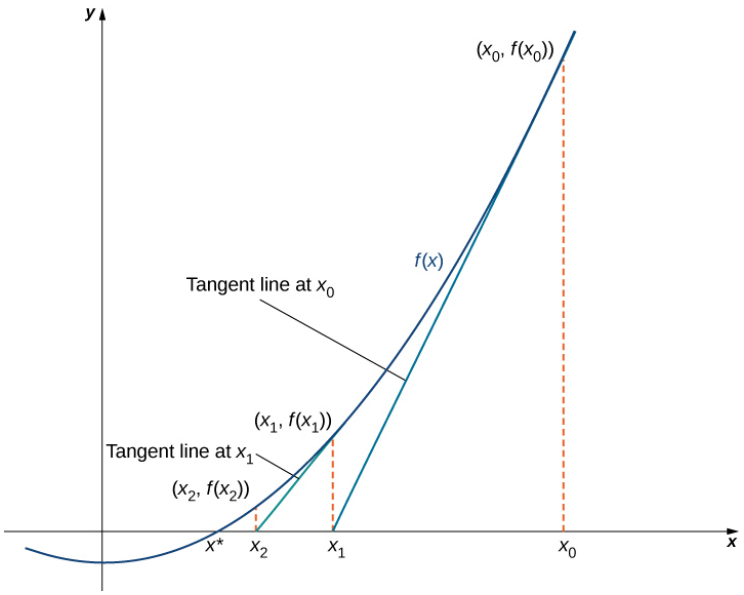


Figure 4.1: Newton's method.

The Newton's method starts with a single-variable function f defined for a real variable x, the function's derivative f', and an initial guess x_0 for a root of f. If the function satisfies sufficient assumptions and the initial guess is close, then

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

is a better approximation of the root than x_0. Geometrically, $(x_1, 0)$ is the intersection of the x-axis and the tangent of the graph of f at $(x_0, f(x_0))$: that is, the improved guess is the unique root of the linear approximation at the initial point. The process is repeated as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

until a sufficiently precise value is reached

Algorithm 6 : NEWTON(f, f', x_0, TOL)

Input: Function f and its derivative f'; initial point x_0; tolerance TOL

Output: Approximate solution

1: $x_1 = x_0 - f(x_0)/f'(x_0)$
2: **while** $|x_1 - x_0| > TOL$ **do**
3: $\quad x_0 = x_1$
4: $\quad x_1 = x_0 - f(x_0)/f'(x_0)$
5: **return** x_1

IMPLEMENT IN PYTHON

```
import math

#Newton's method
def NewtonMethod(f,df,x0, TOL = 1e-10):
    x = x0 - f(x0)/df(x0)
    while abs(x-x0) > TOL:
        x0 = x
        x = x0 - f(x0)/df(x0)
    return x

# find root of cos(x)-x in [0,π]
f = lambda x: math.cos(x)-x
df = lambda x : -math.sin(x)-1
x = NewtonMethod(f,df,0)

print(x)
# Output: 0.7390851332151607
```

QUADRATIC CONVERGENCE FOR NEWTON'S ITERATIVE METHOD

According to Taylor's theorem, any function $f(x)$ which has a continuous second derivative can be represented by an expansion about a point that is close to a root of $f(x)$. Suppose this root is α. Then the expansion of $f(\alpha)$ about x_n is

$$f(\alpha) = f(x_n) + f'(x_n)(\alpha - x_n) + \frac{1}{2!}f''(\xi_n)(\alpha - x_n)^2$$

for some ξ_n between α and x_n.

It follows that

$$0 = f(\alpha) = f(x_n) + f'(x_n)(\alpha - x_n) + \frac{1}{2!}f''(\xi_n)(\alpha - x_n)^2,$$

and hence

$$\frac{f(x_n)}{f'(x_n)} + (\alpha - x_n) = \frac{-f''(\xi_n)}{2f'(x_n)}(\alpha - x_n)^2.$$

Together with $x_{n+1} = x_n - f'(x_n)/f(x_n)$, one gets

$$\alpha - x_{n+1} = \frac{-f''(\xi_n)}{2f'(x_n)}(\alpha - x_n)^2.$$

This equation shows that the rate of convergence is at least *quadratic*.

The square root

Let $a \in \mathbb{R}$ with $a \geqslant 0$. Then, $\sqrt{a}$ is just the unique solution of $x^2 - a = 0$. By the Newton-Raphson's method we can find an approximation of $\sqrt{a}$ by:

$$x_0 = a,\ x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right),\quad \text{for } n \geqslant 0.$$

In particular, we can use this method to compute $\lfloor \sqrt{n} \rfloor$ of a positive integer n:

```
def isqrt(n):
    x=n
    y = (x + 1)//2
    while y < x:
        x=y
        y = (x + n // x)//2
    return x
```

This returns the largest integer x for which x^2 does not exceed n.

4.4 Nonlinear Equations of Serveral Variables

A system of nonlinear equations has the form

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{cases}$$

Let $F\colon \mathbb{R}^n \to \mathbb{R}^n$ be

$$F(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_n(\mathbf{x}))^T$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$. Then, the system above can be written as

$$F(\mathbf{x}) = 0.$$

The function $f_1, \ldots, f_n$ are called the coordinate functions of F.

4.4.1 Iterative Method

We can transform $F(\mathbf{x}) = 0$ into equivalent form $\mathbf{x} = G(\mathbf{x})$, and then find the fixed-points of $G(\mathbf{x})$. As a special case of the contraction mapping theorem, we have:

Theorem 53. *Let $D = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_n, b_n] \subset \mathbb{R}^n$, and let $G\colon D \to D$ be a continuous mapping. Then, G has a fixed point in D.*

Moreover, suppose that all components of G have continuous partial derivatives and a constant $K < 1$ exists with

$$\left|\frac{\partial g_i(\mathbf{x})}{\partial x_j}\right| < \frac{K}{n}, \text{ whenever } \mathbf{x} \in D,$$

for each $i = 1, 2, \ldots, n$ and each component function g_i. Then, the sequence $\{\mathbf{x}^{(k)}\}$ defined by an arbitrary element $\mathbf{x}^{(0)}$ in D and generated by

$$\mathbf{x}^{(k)} = G(\mathbf{x}^{(k-1)}), \text{ for each } k \geqslant 1$$

converges to the unique point $\mathbf{x}^*$ of $\mathbb{R}^n$ and

$$||\mathbf{x}_n - \mathbf{x}^*||_\infty \leqslant \frac{K^n}{1-K}||\mathbf{x}^{(1)} - \mathbf{x}^{(0)}||_\infty.$$

Example 54. Place the nonlinear system

$$\begin{aligned} 3x_1 - \cos(x_2 x_3) - \frac{1}{2} &= 0 \\ x_1^2 - 81(x_2 + 0.1)^2 + \sin x_3 + 1.06 &= 0 \\ e^{-x_1 x_2} + 20x_3 + \frac{10\pi - 3}{3} &= 0 \end{aligned}$$

in the fixed-point form $\mathbf{x} = G(\mathbf{x})$ by solving the ith equation for x_i.

1. Show that there is a unique solution on

$$D = \{(x_1, x_2, x_3) \mid -1 \leqslant x_i \leqslant 1, \text{ for each } i = 1, 2, 3\}.$$

2. Iterate starting with $\mathbf{x}^{(0)} = (0.1, 0.1, -0.1)$ until accuracy within 10^{-5} in the norm $||\cdot||_\infty$.

Proof. Solving the ith equation for x_i gives the fixed-point problem:

$$\begin{aligned} x_1 &= \frac{1}{3}\cos(x_2 x_3) + \frac{1}{6}, \\ x_2 &= \frac{1}{9}\sqrt{x_1^2 + \sin(x_3) + 1.06} - 0.1, \\ x_3 &= -\frac{1}{20}e^{-x_1 x_2} - \frac{10\pi - 3}{60}. \end{aligned} \tag{4.7}$$

Let $G\colon \mathbb{R}^3 \to \mathbb{R}^3$ be defined by $G(\mathbf{x}) = (g_1(\mathbf{x}), g_2(\mathbf{x}), g_3(\mathbf{x}))$, where

$$\begin{aligned} g_1(\mathbf{x}) &= \frac{1}{3}\cos(x_2 x_3) + \frac{1}{6}, \\ g_2(\mathbf{x}) &= \frac{1}{9}\sqrt{x_1^2 + \sin(x_3) + 1.06} - 0.1, \\ g_3(\mathbf{x}) &= -\frac{1}{20}e^{-x_1 x_2} - \frac{10\pi - 3}{60}. \end{aligned} \tag{4.8}$$

Clearly, G is continuous on $\mathbb{R}^3$. Moreover, we can check that $-1 \leqslant g_i(\mathbf{x}) \leqslant 1$ for each $i = 1, 2, 3$ and $\mathbf{x} \in D$. Thus $G(\mathbf{x}) \in D$ whenever $\mathbf{x} \in D$.

The partial derivatives of g_1, g_2, and g_3 are all continuous on D. For every $i, j = 1, 2, 3$ and $\mathbf{x} \in D$ we can check that

$$\left|\frac{\partial g_i}{\partial x_j}(\mathbf{x})\right| \leqslant 0.28.$$

Therefore, the condition in the second part of Theorem 53 holds with $K = 3(0.281) = 0.843$. Consequently, G has a unique fixed point in D, and the nonlinear system has a solution in D. □

IMPLEMENT IN PYTHON

```
# Input: function G=(G_1,G_2,...,G_n) encodes in a list; initial value x_0
#     tolerance TOL; and the maximal number of iterations N
#Output: an approximate root x

def IterationMethod(G,x0,TOL = 1e-10, N=500):
    n = len(x0)
    x = np.zeros(n)
    for k in range(N):
        for i in range(n):
            x[i] = G[i](x0)
        print('k=',k,' x=',x)
        if np.linalg.norm(x-x0) < TOL:
            return x
        for i in range(n):
            x0[i] = x[i]
    return x

# Find the fix-point of function G in Example above
#  with initial [0.1,0.1,-0.1]

G = [lambda y: np.cos(y[1]*y[2])/3+1/6,
     lambda y: np.sqrt(y[0]**2+np.sin(y[2])+1.06)/9-0.1,
     lambda y: -np.exp(-y[0]*y[1])/20-(10*np.pi-3)/60]
```

```
24
25 x = IterationMethod(G, [0.1,0.1,-0.1])
26 print(x)
27 # Ouput: [ 5.00000000e-01  6.10345108e-14 -5.23598776e-01]
```

4.4.2 Newton's method

The Newton's method discussed above for solving a single-variable equation $f(x) = 0$ can be generalized for solving multivariate equation systems:

$$F(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Same as in the single variable case of $n = 1$, to solve the equation $F(\mathbf{x}) = 0$, we first consider the Taylor series expansion of each of component of $F(\mathbf{x})$ around a guess point $\mathbf{x}^{(0)} = [x_1^{(0)}, \ldots, x_n^{(0)}]^T$:

$$f_i(\mathbf{x}) = f_i(\mathbf{x}^{(0)}) + \sum_{j=1}^{n} \frac{\partial f_i(\mathbf{x}^{(0)})}{\partial x_j}(x_j - x_j^{(0)}) + r_i(\mathbf{x}), \text{ for } i = 1, 2, \ldots, n,$$

with $\|r(\mathbf{x})\| = O(\left\|\mathbf{x} - \mathbf{x}^{(0)}\right\|^2)$ as $\left\|\mathbf{x} - \mathbf{x}^{(0)}\right\|$ small enough.

These n equations can be expressed in matrix form

$$F(\mathbf{x}) = F(\mathbf{x}^{(0)}) + \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}^{(0)}) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}^{(0)}) & \cdots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}^{(0)}) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}^{(0)}) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}^{(0)}) & \cdots & \frac{\partial f_2}{\partial x_n}(\mathbf{x}^{(0)}) \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1}(\mathbf{x}^{(0)}) & \frac{\partial f_n}{\partial x_2}(\mathbf{x}^{(0)}) & \cdots & \frac{\partial f_n}{\partial x_n}(\mathbf{x}^{(0)}) \end{bmatrix} \begin{bmatrix} x_1 - x_1^{(0)} \\ x_2 - x_2^{(0)} \\ \vdots \\ x_n - x_n^{(0)} \end{bmatrix} + \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix}$$

or equivalently,

$$F(\mathbf{x}) = F(\mathbf{x}^{(0)}) + J(\mathbf{x}^{(0)})(\mathbf{x} - \mathbf{x}^{(0)}) \tag{4.9}$$

where $J(\mathbf{x}^{(0)})$ is the Jacobian matrix of F evaluated at $\mathbf{x}^{(0)}$.

Recall that the Jacobian matrix of $F(\mathbf{x})$, also denoted by $\dfrac{\partial F}{\partial \mathbf{x}}$, is

$$J(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}) & \cdots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}) & \cdots & \frac{\partial f_2}{\partial x_n}(\mathbf{x}) \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1}(\mathbf{x}) & \frac{\partial f_n}{\partial x_2}(\mathbf{x}) & \cdots & \frac{\partial f_n}{\partial x_n}(\mathbf{x}) \end{bmatrix}$$

When $\left\|\mathbf{x}-\mathbf{x}^{(0)}\right\|$ is small, $\mathbf{r}(\mathbf{x})$ is neglected, so that from (4.9) we have

$$F(\mathbf{x}) \approx F(\mathbf{x}^{(0)}) + J(\mathbf{x}^{(0)})(\mathbf{x}-\mathbf{x}^{(0)}). \tag{4.10}$$

We now consider solving the equation system $F(\mathbf{x}) = 0$. Setting $F(\mathbf{x}) = 0$ and solving (4.10) for $\mathbf{x}$ gives

$$\mathbf{x} = \mathbf{x}^{(0)} - J(\mathbf{x}^{(0)})^{-1}F(\mathbf{x}^{(0)}).$$

Newton's method starts with a good initial choice $\mathbf{x}^{(0)}$ of the root's position, the formula above can be applied iteratively to obtain

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} - J(\mathbf{x}^{(k-1)})^{-1}F(\mathbf{x}^{(k-1)})$$

for $k = 1, 2, 3\ldots$. An initial point $\mathbf{x}^{(0)}$ that provides safe convergence of Newton's method is called an *approximate zero*.

Remark 55.

1. The stopping criteria for the iteration is that the iterates change by at most tolerance TOL if $||\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}||_\infty < TOL$.

2. In order to compute $J(\mathbf{x})^{-1}F(\mathbf{x})$, let $\mathbf{y} = J(x)^{-1}F(x)$, so that $J(\mathbf{x})\mathbf{y} = F(\mathbf{x})$. Then, solving this linear system to find $\mathbf{y}$.

▷ IMPLEMENT IN PYTHON

We now implement Newton's method in Python, and apply for finding a root of the following system

$$\begin{cases} x + e^{-x} + y^3 & = 0 \\ x^2 + 2xy - y^2 + \tan x & = 0 \end{cases}$$

with setting the initial guess as $\mathbf{x}^{(0)} = (3, -1.5)$ and the tolerance as $TOL = 10^{-14}$.

```
# Input: function F = (f_1, f_2, ..., f_n), Jacobian matrix J; initial value x_0
#     tolerance TOL; and the maximal number of iterations N
#Output: an approximate root x

def MultivariateNewton(F, J, x0, TOL=1e-10, N=500):
    n = len(x0)
    J1 = np.zeros((n,n)) # use to evaluate Jacobian matrix
    x = np.zeros(n)
    b = np.zeros(n)
    for k in range(N):
        for i in range(n):
```

```
            b[i] = F[i](x0)
            for j in range(n):
                J1[i,j] = J[i][j](x0)
        y = np.linalg.solve(J1,b)
        for i in range(n):
            x[i] = x0[i]-y[i]
        if np.linalg.norm(y) < TOL:
            return x
        for i in range(n):
            x0[i] = x[i]
    return x

# Find a root of the system:
#      x + e^{-x} + y^3 = 0,  x^2 + 2xy - y^2 + tan x = 0
#  with initial [3,-1.5]

F=[lambda y: y[0]+np.exp(-y[0])+y[1]**3,
     lambda y: y[0]**2+2*y[0]*y[1]-y[1]**2+np.tan(y[0])]
J=[[lambda y:1-np.exp(-y[0]),lambda y: 3*y[1]**2],
    [lambda y: 2*y[0]+2*y[1]+np.cos(y[0])**2,lambda y:2*y[0]-2*y[1]]]

x = MultivariateNewton(F, J,[3,-1.5], 1e-14)
print(x)
# Ouput: [ 3.59240993 -1.53544358]
```

▷ Numerical Differentiation

The Newton method has a disadvantage that it needs derivatives. Derivative of a function $f\colon \mathbb{R} \to \mathbb{R}$ at $x_0 \in \mathbb{R}$ is

$$f'(x_0) = \lim_{h\to 0} \frac{f(x_0+h) - f(x_0)}{h}$$

This formula gives an obvious way to generate an approximation to $f''(x_0)$; simply compute

$$\frac{f(x_0+h) - f(x_0)}{h}$$

for small values of h. Although this may be obvious, it is not very successful, due to our old nemesis round-off error. One improvement is to use Lagrange interpolation formula to get an approximation for $f'(x_0)$.

To obtain general derivative approximation formulas, suppose that $\{x_0, x_1, \ldots, x_n\}$ are $(n+1)$ distinct numbers. Let $L_k(x)$ denote the kth Lagrange coefficient polynomial for f at $\{x_0, x_1, \ldots, x_n\}$, that is

$$L_k(x) = \frac{(x-x_0)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)} = \prod_{i\neq k} \frac{x-x_i}{x_k-x_i}.$$

Then the Lagrange interpolation formula is

$$f(x) = \sum_{k=0}^{n} f(x_k)L_k(x) + \frac{(x-x_0)\cdots(x-x_n)}{(n+1)!} f^{(n+1)}(\xi(x))$$

for some $\xi(x)$ lies between x_0 and x.

Differentiating this expression gives

$$f'(x) = \sum_{k=0}^{n} f(x_k)L'_k(x) + D_x\left[\frac{(x-x_0)\cdots(x-x_n)}{(n+1)!}\right] f^{(n+1)}(\xi(x)) + \frac{(x-x_0)\cdots(x-x_n)}{(n+1)!} D_x f^{(n+1)}(\xi(x)).$$

In particular, for $x = x_j$ we have

$$f'(x_j) = \sum_{k=0}^{n} f(x_k)L'_k(x) + \frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{k\neq j}(x_k - x_j) \tag{4.11}$$

which is called an $(n+1)$-point formula to approximate $f'(x_j)$.

▷ Implement of Numeric Differentiation

We now illustrate the *Five-Point Midpoint Formula* to approximate $f'(x)$. For 5 points $\{x-2h, x-h, x, x+h, x+2h\}$, the formula (4.11) becomes:

$$f'(x) = \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} + \frac{h^4}{30} f^{(5)}(\xi)$$

where ξ lies between $x-2h$ and $x+2h$.

In the following code we compute $\partial F(x_1, \ldots, x_n)/\partial x_i$ at a point $\mathbf{x}^0$ and use it for Newton's method.

```
# Input: function F, point x, index ith in ∂F/∂x_i, step h
def diff5(F,i,x, h = 1e-4):
    a = x.copy()
    a[i] -= 2*h
    d = F(a)
    a[i] += h
    d -= 8 * F(a)
    a[i] += 2*h
    d += 8* F(a)
    a[i] += h
    d -= F(a)
    d /= (12*h)
    return d

```

```
# Newton's method for solving nonlinear system without derivatives
def Newton2(F, x0, TOL=1e-10, N=500):
    n = len(x0)
    h = TOL ** (1./3)
    x = np.zeros(n)
    b = np.zeros(n)
    J = np.zeros((n,n))
    for k in range(N):
        # Compute Jacobian matrix using diff5
        for i in range(n):
            for j in range(n):
                J[i,j] = diff5(F[i],j,x0,h)

        for i in range(n):
            b[i] = F[i](x0)
        y = np.linalg.solve(J,b)
        for i in range(n):
            x[i] = x0[i]-y[i]
        if np.linalg.norm(y) < TOL:
            return x
        for i in range(n):
            x0[i] = x[i]
    return x

# Find a root of the system with initial [3,-1.5]:
#      x + e^{-x} + y^3 = 0,  x^2 + 2xy - y^2 + tan x = 0
F=[lambda y: y[0]+np.exp(-y[0])+y[1]**3,
     lambda y: y[0]**2+2*y[0]*y[1]-y[1]**2+np.tan(y[0])]
x = Newton2(F,[3,-1.5], 1e-14)

print(x)
# Ouput: [ 3.59240993 -1.53544358]
```

4.5 Introduction to Optimization

An optimization problem can be represented in the following way:

- ▷ *Given*: a function $f\colon D \to \mathbb{R}$ for some subset D of $\mathbb{R}^n$.
- ▷ *Sought*: an element $\mathbf{x}^0 \in D$ such that $f(\mathbf{x}^0) \leqslant f(\mathbf{x})$ for all $\mathbf{x} \in D$ ("minimization") or such that $f(\mathbf{x}^0) \geqslant f(\mathbf{x})$ for all $\mathbf{x} \in D$ ("maximization").

Since the following is valid

$$f(\mathbf{x}^0) \geqslant f(\mathbf{x}) \Longleftrightarrow \tilde{f}(\mathbf{x}^0) \leqslant \tilde{f}(\mathbf{x})$$

with $\tilde{f}(\mathbf{x}) = -f(\mathbf{x})$ for $\mathbf{x} \in D$.

It is more convenient to solve minimization problems. However, the opposite perspective would be valid, too. The minimization optimization problem can be stated as

$$\min_{\mathbf{x}} f(\mathbf{x})$$
$$\text{subject to } \mathbf{x} \in D.$$

Typically, D is often specified by a set of *constraints*, *equalities* or *inequalities* that the members of D have to satisfy. The domain D is called the *search space* or the *choice set*, while the elements of D are called *candidate solutions* or *feasible solutions*.

The function f is called an *objective function*. A feasible solution that minimizes (or maximizes, if that is the goal) the objective function is called an *optimal solution*.

While *global minimum* is at least as good as every feasible element, a *local minimum* is at least as good as any nearby elements.

A *local minimum* $\mathbf{x}^*$ is defined as an element in D for which there exists some $\delta > 0$ such that

$$f(\mathbf{x}^*) \leqslant f(\mathbf{x}) \tag{4.12}$$

for all $\mathbf{x} \in D$ where $\|\mathbf{x} - \mathbf{x}^*\| \leqslant \delta$; that is to say, on some region around $\mathbf{x}^*$ all of the function values are greater than or equal to the value at that element. If the inequality (4.12) is strict whenever $\mathbf{x} \neq \mathbf{x}^*$, then $\mathbf{x}^*$ is a *strict local minimum*. Local maxima and strict local minima are defined similarly.

▷ Convex Optimization:

A convex optimization problem is an optimization problem in which the objective function is a *convex function* and the feasible set is a *convex set*.

A set $D \subseteq \mathbb{R}^n$ is *convex* if for all members $\mathbf{x}, \mathbf{y} \in D$ and all $\lambda \in [0, 1]$, we have that

$$\lambda\mathbf{x} + (1 - \lambda)\mathbf{y} \in D.$$

A function f is *convex* on a convex set D if for all $\lambda \in [0, 1]$ and all $\mathbf{x}, \mathbf{y} \in D$, the following condition holds:

$$f(\lambda\mathbf{x} + (1 - \lambda)\mathbf{y}) \leqslant \lambda f(\mathbf{x}) + (1 - \lambda)f(\mathbf{y}). \tag{4.13}$$

If the equality occurs if and only if either $\mathbf{x} = \mathbf{y}$ or $\lambda = 0$ or $\lambda = 1$, then f is called *strict convex* on D.

A function f is *convex* if its domain is convex and f is convex on this domain.

wThe following are useful properties of convex optimization problems:

- every local minimum is a global minimum;
- the optimal set is convex;
- if the objective function is strictly convex, then the problem has at most one optimal point.

4.5.1 Unconstrained optimization

Let

$$f\colon\ \mathbb{R}^n \to \mathbb{R}\,.$$

The unconstrained optimization problem in $\mathbb{R}^n$ can be stated as,

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{with respect to} \quad & x \in \mathbb{R}^n\,. \end{aligned}$$

In this section we will give a necessary condition for local minima and then represent some methods for finding them.

A vector $\mathbf{d} \in \mathbb{R}^n$ is said to be a *descent direction* for f at $\mathbf{x}^*$ if there exists $\delta > 0$ such that

$$f(\mathbf{x}^* + \lambda \mathbf{d}) < f(\mathbf{x}^*)$$

for all $\lambda \in (0, \delta)$.

If the function f is differentiable it is possible to give a simple condition guaranteeing that a certain direction is a descent direction. Recall that the gradient vector of f at $\mathbf{x}$ is defined by

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{x_1} \\ \frac{\partial f(\mathbf{x})}{x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{x_n} \end{bmatrix} \in \mathbb{R}^n\,.$$

Proposition 56. *Assume that Δf exists and is continuous. Let $\mathbf{x}^*$ and $\mathbf{d}$ be given. If $\nabla f(\mathbf{x}^*)^T \mathbf{d} < 0$, then $\mathbf{d}$ is a descent direction for f at $\mathbf{x}^*$.*

Proof. Note that $\nabla f(\mathbf{x}^*)^T\mathbf{d}$ is the directional derivative of f (which is differentiable by hypothesis) at x^* along $\mathbf{d}$, i.e.

$$\nabla f(\mathbf{x}^*)^T\mathbf{d} = \lim_{\lambda\to 0}\frac{f(\mathbf{x}^*+\lambda\mathbf{d})-f(\mathbf{x}^*)}{\lambda}$$

and this is negative by hypothesis. As a result, for $\lambda > 0$ and sufficient small, we have

$$f(\mathbf{x}^*+\lambda\mathbf{d}) < f(\mathbf{x}^*)$$

and hence the proposition follows. □

We are now ready to give some necessary conditions and some sufficient conditions for a local minimum.

Theorem 57 (First order necessary condition)**.** *Assume $\nabla f(\mathbf{x}^*)$ exists and is continuous. The point $\mathbf{x}^*$ is a local minimum of f only if*

$$\nabla f(\mathbf{x}^*) = 0.$$

Proof. If $\nabla f(\mathbf{x}^*) \neq 0$, the direction $\mathbf{d} = -\nabla f(\mathbf{x}^*)$ is a descent direction by Proposition above. Therefore, in a neighborhood of $\mathbf{x}^*$ there is a point $\mathbf{x}^* + \lambda d = \mathbf{x}^* - \lambda\nabla f(\mathbf{x}^*)$ such that $f(\mathbf{x}^* - \lambda\nabla f(\mathbf{x}^*)) < f(\mathbf{x}^*)$, and this contradicts the hypothesis that $\mathbf{x}^*$ is a local minimum. □

Remark 58. A point $\mathbf{x}^*$ such that $\nabla f(\mathbf{x}^*) = 0$ is called a *stationary point* of f.

Recall that the Hessian matrix of f is just the Jacobian matrix of ∇f, i.e.

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{bmatrix}$$

Theorem 59 (Second order sufficient condition)**.** *Assume $\nabla^2 f(\mathbf{x})$ exists and and is continuous. The point $\mathbf{x}^*$ is a strict local minimum of f if*

$$\nabla f(\mathbf{x}^*) = 0$$

and

$$\mathbf{x}^T\nabla^2 f(\mathbf{x}^*)\mathbf{x} > 0$$

for all non-zero $\mathbf{x} \in \mathbb{R}^n$.

Generally, unless the objective function is convex in a minimization problem, there may be several local minima. In a convex problem, if there is a local minimum, it is also the global minimum, but a nonconvex problem may have more than one local minimum not all of which need be global minima.

Proposition 60 (Necessary and sufficient condition for convex functions)**.** *If f is convex and ∇f is continuous, then a point $\mathbf{x}^*$ is a global minimum of f if and only if $\nabla f(\mathbf{x}^*) = 0$.*

General unconstrained minimization algorithm

Consider the problem of minimizing the function $f\colon \mathbb{R}^n \to \mathbb{R}$ and suppose that ∇f exists and is continuous. General unconstrained minimization algorithms allow only to determine stationary points of f, i.e. to determine points in the set

$$\Omega = \{\mathbf{x} \in \mathbb{R}^n \mid \nabla f(\mathbf{x}) = 0\}.$$

An algorithm for the solution of the considered minimization problem is a sequence $\{\mathbf{x}^k\}$, obtained starting from an initial point $\mathbf{x}^0$, having some convergence properties in relation with the set Ω. Most of the algorithms that will be studied in this notes can be described in the following general way.

1. Fix a point $\mathbf{x}^0 \in \mathbb{R}^n$ and set $k = 0$.
2. If $\mathbf{x}^k \in \Omega$, stop.
3. Compute a descent direction $\mathbf{d}^k$ for f at $\mathbf{x}^k$.
4. Compute a step α_k along $\mathbf{d}^k$.
5. Let $\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha_k \mathbf{d}^k$. Set $k = k + 1$ and go back to 2.

The existing algorithms differ in the way the descent direction $\mathbf{d}^k$ is computed and on the criteria used to compute the step α_k.

Line Search

A *line search* is a method to compute the step α_k along a given direction $\mathbf{d}^k$. The choice of α_k affects both the convergence and the speed of convergence of the algorithm.

In any line search one considers the function of one variable $\phi\colon \mathbb{R} \to \mathbb{R}$ defined as

$$\phi(\alpha) = f(\mathbf{x}^k + \alpha \mathbf{d}^k) - f(\mathbf{x}^k).$$

The derivative of $\phi(\alpha)$ with respect to α is given by

$$\phi'(\alpha) = \nabla f(\mathbf{x}^k + \alpha \mathbf{d}_k)^T \mathbf{d}^k$$

provided that ∇f is continuous. In particular, $\phi'(0) = \nabla f(\mathbf{x}^k)^T \mathbf{d}^k$.

The *exact line search* consists in finding α_k that minimizes $\phi(\alpha)$ subject to $\alpha \geqslant 0$. However, it is usually undesirable to devote substantial resources to finding a value of α to precisely minimize f. It is therefore more convenient to use approximate methods, i.e. methods which are computationally simple and which guarantee particular convergence properties. Such methods are aimed at finding an interval of acceptable values for α_k subject to the following two conditions:

1. α_k has to guarantee a sufficient reduction of f;
2. $\mathbf{x}^k + \alpha_k \mathbf{d}^k$ has to be sufficient away from $\mathbf{x}^k$.

Armijo method

Armijo method was the first non-exact linear search method. Starting with a maximum candidate step size value $\alpha_0 > 0$, using search control parameters $\sigma \in (0, 1)$ and $\gamma \in (0, 1)$, the backtracking line search algorithm can be expressed as follows: find the smallest $j \geqslant 0$ such that

$$f(\mathbf{x}^k + \alpha_j \mathbf{d}) - f(\mathbf{x}^k) > \alpha_j t \leqslant \gamma \nabla f(\mathbf{x}^k)^T \mathbf{d}^k$$

where $\alpha_j = \alpha_0 \sigma^j$.

Remark 61. By Taylor expansion of $\phi(\alpha)$ and $\phi'(0) = \nabla f(\mathbf{x}^k)^T \mathbf{d}^k$, we imply that Armijo algorithm always terminates.

Armijo method can be implemented using the following (conceptual) algorithm.

Algorithm 7 : Armijo Algorithm

Input: Function f, $m = \nabla f(\mathbf{x}^k)^T \mathbf{d}^k$, point $\mathbf{x}$, descent direction $\mathbf{d}$, numbers α_0, σ, γ
Output: Step length α

```
Set α = α_0
Set t = γm
while f(x + αd) − f(x) > tα do
    α = σα
return α
```

The Steepest Descent

Consider the problem

$$\begin{aligned} &\text{minimize} && f(\mathbf{x}) \\ &\text{with respect to} && \mathbf{x} \in \mathbb{R}^n . \end{aligned}$$

In order to obtain the minimum point $\mathbf{x}^*$ by the iterative method, we start with the initial point $\mathbf{x}^{(0)}$, and generated the sequence $\{\mathbf{x}^{(k)}\}$ that converges to $\mathbf{x}^*$. The design variables are updated at each iteration k using:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}_k$$

where $\mathbf{p}_k$ is the search direction for the iteration k, and α_k is the accepted step length from the line search, that is the minimum of the function $t \mapsto f(\mathbf{x}^{(k)} + t\mathbf{p}_k)$.

This method uses the direction $\mathbf{p}_k = -\nabla f(\mathbf{x}^{(k)})$, where $\nabla f(\mathbf{x})$ is the gradient of f at $\mathbf{x}$ given by

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), \frac{\partial f}{\partial x_2}(\mathbf{x}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{x}) \right)^T.$$

Algorithm 8 : STEEPEST DESCENT

Input: Initial guess $\mathbf{x}^{(0)}$, convergence tolerances ε, the number of iterations N

Output: Optimum, x^*

1: **for** $k = 0$ to N **do**
2: $\quad p_k = -\nabla f(\mathbf{x}^{(k)})$
3: $\quad$ Perform the line search to find step length α_k
4: $\quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}_k$
5: $\quad$ **if** $||\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}|| < \epsilon$ **then**
6: $\quad\quad$ **return** $\mathbf{x}^{(k+1)}$
7: **return** "The Steepest Descent did not converge"

```python
import numpy as np

# Input: function f; gradient m; point x; descent direction d;
    m = ∇f(x^k)^T d^k; a, σ, γ
# Output: Step length

def Armijo(f, x, d, m, a, sm, gm):
    n = len(x)
    y = [0.0]*n
    m *= gm
    vf = f(x)
    mx = 32

    # Extend a
    for i in range(n):
        y[i] = x[i] + a*d[i]
```

```
    if f(y) <= vf + m * a:
        for i in range(mx):
            a /= sm
            for i in range(n):
                y[i] = x[i] + a*d[i]
            if f(y) > vf + m * a:
                break
        return a * sm
    else:
        # shrinks a
        while True:
            for i in range(n):
                y[i] = x[i] + a*d[i]
            if f(y) > vf + m * a:
                a *= sm
            else:
                return a

# Input: Function f; gradient gradf; initial x_0; tolerance TOL; number
    of iterations N
#Output: an approximate local minimum

def SteepestDescent(f, gradf, x0, TOL = 1e-10, N=500):
    n = len(x0)
    x = x0.copy()
    d = [0.0] * n
    y = np.zeros(n)
    a = 1.5
    m = 0.0
    for k in range(N):
        m = 0.0
        for i in range(n):
            d[i] = -gradf[i](x)
            m += d[i]**2
        if m == 0:
            return x
        m = -m
        a = Armijo(f,x,d,m, 1.5,0.8,0.5)
        for i in range(n):
            y[i] = x[i] + a * d[i]
        if np.linalg.norm(y-x) < TOL:
            return y
        for i in range(n):
            x[i] = y[i]
```

```
    return x

# minimize F = x1 - x2 + 2x1^2 + 2x1x2 + x2^2; initial [0,0]

F = lambda x: x[0]-x[1]+2*x[0]**2+2*x[0]*x[1]+x[1]**2
GF = [lambda x:1+4*x[0]+2*x[1], lambda x:-1+2*x[0]+2*x[1]]
x = SteepestDescent(F,GF, [0,0])
print(x)

#Output: [-1.   1.5]
```

NEWTON'S METHOD

Let $f\colon \mathbb{R}^n \to \mathbb{R}$ be a given function and assume that $\nabla^2 f$ is continuous. Newton's method for the minimization of f can be derived from solving the system

$$\nabla f(\mathbf{x}) = 0$$

by Newton's method.

The method can be described by the simple scheme. Start with a good initial choice $\mathbf{x}^0$ of a local minimum, generate iteratively the sequence

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} - (\nabla^2 f(\mathbf{x}^{k-1}))^{-1} \nabla f(\mathbf{x}^{k-1})$$

for $k = 1, 2, 3 \ldots$.

Second Derivative Formula

The Newton's method in this case evolves the first partial derivative and the second partial derivative. We have given an approximation for first derivative, we now want to do the same for second derivative.

Let $u\colon \mathbb{R}^2 \to \mathbb{R}$ and $(a, b) \in \mathbb{R}^2$. We will derive approximations for $u_{xx}(a, b), u_{yy}(a, b)$ and $u_{xy}(a, b)$ by using a step size $h > 0$.

For a function $g\colon \mathbb{R} \to \mathbb{R}$, expand it in a third Taylor polynomial about a point x, we have

$$g(x + h) = g(x) + g'(x)h + \frac{1}{2}g''(x)h^2 + \frac{1}{6}g'''(x)h^3 + \frac{1}{24}g^{(4)}(\xi_1)h^4$$

and

$$g(x - h) = g(x) - g'(x)h + \frac{1}{2}g''(x)h^2 - \frac{1}{6}g'''(x)h^3 + \frac{1}{24}g^{(4)}(\xi_{-1})h^4$$

where $\xi_1 \in (x, x + h)$ and $\xi_{-1} \in (x - h, x)$. If we add these equations, then

$$g(x + h) + g(x - h) = 2f(x) + f''(x)h^2 + \frac{1}{24}\left[f^{(4)}(\xi_1) + f^{(4)}(\xi_{-1}\right] h^4.$$

On the other hand, by the intermediate value theorem,

$$\frac{f^{(4)}(\xi_1) + f^{(4)}(\xi_{-1})}{2} = f^{(4)}(\xi)$$

for some $\xi \in [\xi_{-1}, \xi_2]$. Thus, we have

$$g''(x) = \frac{g(x+h) - 2g(x) + g(x-h)}{h^2} - \frac{h^2}{12} g^{(4)}(\xi).$$

By applying this formula to compute $u_{xx}(a,b)$ and $u_{xy}(a,b)$ we obtain

$$u_{xx}(a,b) = \frac{u(a+h,b) - 2u(a,b) + u(a-h,b)}{h^2} - \frac{h^2}{12} u_{xxxx}(\xi, b)$$

and

$$u_{yy}(a,b) = \frac{u(a,b+h) - 2u(a,b) + u(a,b-h)}{h^2} - \frac{h^2}{12} u_{yyyy}(a, \eta)$$

where $\xi \in (a-h, a+h)$ and $\eta \in (b-h, b+h)$.

We next compute $u_{xy}(a,b)$. First by applying Formula (4.11) for 3 points $\{x-h, x, x+h\}$, we have

$$g'(x) = \frac{g(x+h) - g(x-h)}{2h} - \frac{h^2}{6} g^{(3)}(\xi)$$

for some $\xi \in (x-h, x+h)$.

Using this formula to compute $u_{xy}(a,b)$ we obtain

$$u_{xy}(a,b) = \frac{u(a+h,b+h) - u(a+h,b-h) - u(a-h,b+h) + u(a-h,b-h)}{4h^2} + R$$

where

$$R = -\frac{h^2}{6}[\theta u_{xxxx}(\xi, \eta_0) + u_{xxxy}(\xi, \eta_0) + u_{xyyy}(a, \eta_1)]$$

with $0 < \theta < 1$, $\xi \in (a-h, a+h)$ and $\eta_0, \eta_1 \in (b-h, b+h)$.

Thus we have approximate formulas for second partial derivatives with order of error $O(h^2)$.

Implement

```
# Compute ∂²F/∂x_i∂x_j
def secondDiff(F,i,j,x,h=1e-4):
    a = x.copy()
    if i == j:
        a[i] -= h
        d = F(a) - 2*F(x)
        a[i] = x[i]+h
        d += F(a)
        return d/(h**2)
```

```
    a[i] += h
    a[j] += h
    d = F(a)
    a[i] = x[i]-h
    a[j] = x[j]-h
    d += F(a)
    a[i] = x[i] + h
    d -= F(a)
    a[i] = x[i]-h
    a[j] = x[j]+h
    d -= F(a)
    return (d/(4*h**2))

#Newton method
def NewtonOpitization(F, x0, TOL = 1e-10, N = 500):
    n = len(x0)
    h = TOL ** (1./3) # For gradient vector
    h2 = np.sqrt(TOL) # For Hessian matrix
    x = np.zeros(n)
    b = np.zeros(n)
    H = np.zeros((n,n))
    for k in range(N):
        # Compute Hessian matrix using diff5
        for i in range(n):
            for j in range(n):
                H[i,j] = secondDiff(F,i,j,x0,h2)

        # Compute gradient
        for i in range(n):
            b[i] = diff5(F,i,x,h)

        y = np.linalg.solve(H,b)
        for i in range(n):
            x[i] = x0[i]-y[i]
        if np.linalg.norm(y) < TOL:
            return x
        for i in range(n):
            x0[i] = x[i]
    return x

# minimize F = x_1 - x_2 + 2x_1^2 + 2x_1x_2 + x_2^2; initial [0,0]
F = lambda x: x[0]-x[1]+2*x[0]**2+2*x[0]*x[1]+x[1]**2
x = NewtonOpitization(F, [0,0])

```

```
print(x)
#Output: [-1.    1.5]
```

4.5.2 Nonlinear programming

In this subsection we discuss the basic tools for the solution of optimization problems of the form

$$(P)\begin{cases} & \min\limits_{\mathbf{x}} f(\mathbf{x}) \\ \text{subject to} & g_i(\mathbf{x}) \leqslant 0,\ i=1,\dots,m \\ & h_i(\mathbf{x}) = 0,\ i=1,\dots,p \end{cases} \tag{4.14}$$

or in vector form

$$\begin{aligned} &\min_{\mathbf{x}} f(\mathbf{x}) \\ \text{subject to } &g(\mathbf{x}) \leqslant 0 \\ &h(\mathbf{x}) = 0 \end{aligned}$$

where

$$\mathbf{x} \in \mathbb{R}^n,\ f\colon \mathbb{R}^n \to \mathbb{R}, g = (g_1,\dots,g_m)\colon \mathbb{R}^n \to \mathbb{R}^m \text{ and } h = (h_1,\dots,h_p)\colon \mathbb{R}^n \to \mathbb{R}^p\,.$$

Karush–Kuhn–Tucker conditions

In order to state the conditions for optimality, we define the Lagrangian function of the problem P (i.e. Problem (4.14)):

$$L(\mathbf{x},\lambda,\mu) = f(\mathbf{x}) + \lambda^T g(\mathbf{x}) + \mu^T h(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i g_i(\mathbf{x}) + \sum_{i=1}^{p} \mu_i h_i(\mathbf{x}),$$

with $\lambda \in \mathbb{R}^m$ and $\mu \in \mathbb{R}^p$. The vectors λ and μ are called *multipliers*.

A point $\tilde{\mathbf{x}}$ is a *regular point* for the constraints of the problem P if the vectors

$$\{\nabla g_i(\tilde{\mathbf{x}}) \mid g_i(\mathbf{x}) = 0\} \cup \{\nabla h_j(\tilde{\mathbf{x}}) \mid j = 1,\dots,p\}$$

are linearly independent.

The definition of regular point is given because, the necessary and the sufficient conditions for optimality, in the case of regular points are relatively simple.

Theorem 62 (First order necessary condition)**.** *Consider problem P. Suppose $\mathbf{x}^*$ is a local solution of the problem P, and x^* is a regular point for the constraints. Then there exist (unique) multipliers λ^* and μ^* such that*

$$\begin{aligned} &\nabla_{\mathbf{x}} L(\mathbf{x}^*, \lambda^*, \mu^*) = 0 \\ &g(\mathbf{x}^*) \leqslant 0 \\ &h(\mathbf{x}^*) = 0 \\ &\lambda^* \geqslant 0 \\ &(\lambda^*)^T g(\mathbf{x}^*) = 0. \end{aligned} \tag{4.15}$$

Conditions (4.15) are known as Karush–Kuhn–Tucker conditions (or KKT conditions for short).

Let $\mathbf{x}^*$ be a local solution of the problem P and let λ^* be the corresponding (optimal) multiplier. At $\mathbf{x}^*$ the condition of *strict complementarity* holds if $\lambda_i^* > 0$ whenever $g_i(\mathbf{x}^*) = 0$.

Theorem 63 (Second order sufficient condition)**.** *Consider the problem P. Assume that there exist $\mathbf{x}^*$, λ^* and μ^* satisfying conditions (4.15). Suppose moreover that λ^* is such that the condition of strict complementarity holds at $\mathbf{x}^*$. Suppose finally that*

$$s^T \nabla^2_{xx} L(\mathbf{x}^*, \lambda^*, \mu^*) s > 0$$

for all $s \neq 0$ such that

$$\begin{bmatrix} \frac{\partial g(\mathbf{x}^*)}{\partial x} \\ \frac{\partial h(\mathbf{x}^*)}{\partial x} \end{bmatrix} s = 0.$$

Then $\mathbf{x}^*$ is strict constrained local minimum of the problem P.

Remark 64. In what follows, we will always tacitly assume that the conditions of regularity and of strict complementarity hold.

Dual Problem

Consider problem P (we call it *primal problem*). We define the Lagrange dual function of P by

$$G(\lambda, \mu) = \inf_{\mathbf{x} \in \mathbb{R}^n} L(\mathbf{x}, \lambda, \mu).$$

It is worth mentioning that $G(\lambda, \mu)$ is a *concave* function (i.e. $-G$ is convex).

The *Lagrane dual problem* (dual problem for short) of P is defined by

$$(P^*) \begin{cases} \max\limits_{\lambda, \mu} G(\lambda, \mu) \\ \text{subject to} \quad \lambda \geqslant 0. \end{cases} \tag{4.16}$$

Let α^* and β^* be the optimal values of the primal and the dual problem, respectively.

Theorem 65 (Weak duality)**.** $\alpha^* \geqslant \beta^*$.

Proof. The following is obvious for $\lambda \in \mathbb{R}^m$ and $\mu \in \mathbb{R}^p$ with $\lambda \geqslant 0$,

$$\begin{aligned}
\alpha^* &= \min\{f(\mathbf{x}) \mid g(\mathbf{x}) \leqslant 0, h(\mathbf{x}) = 0\} \\
&\geqslant \min\{f(\mathbf{x}) \mid \lambda^T g(\mathbf{x}) \leqslant 0, \mu^T h(\mathbf{x}) = 0\} \\
&\geqslant \min\{f(\mathbf{x}) + \lambda^T g(\mathbf{x}) + \mu^T h(\mathbf{x}) \mid \lambda^T g(\mathbf{x}) \leqslant 0, \mu^T h(\mathbf{x}) = 0\} \\
&\geqslant \min\{L(\mathbf{x}, \lambda, \mu) \mid \lambda^T g(\mathbf{x}) \leqslant 0, \mu^T h(\mathbf{x}) = 0\} \\
&\geqslant \min\{L(\mathbf{x}, \lambda, \mu) \mid \mathbf{x} \in \mathbb{R}^n\} \\
&= G(\lambda, \mu).
\end{aligned}$$

It follows that $\alpha^* \geqslant \max\{G(\lambda, \mu) \mid \lambda \geqslant 0\} = \beta^*$. □

Now we state a condition that guarantees $\alpha^* = \beta^*$. Assume that $f, g_1, \ldots, g_m$ are convex and $h_1, \ldots, h_p$ are *affine*. In this case, the problem P is called the *convex minimization problem*. The problem P is said to be satisfied the *Slater's condition* if it has a feasible solution $\mathbf{x}^*$ that is strictly feasible, i.e.

$$g_i(\mathbf{x}^*) < 0, \text{ for } i = 1, \ldots, m \text{ and } h_j(\mathbf{x}^*) = 0, \text{ for } j = 1, \ldots, p.$$

Theorem 66 (Strong duality)**.** *If the primal is a convex problem, and it satisfies Slater's condition, then* $\alpha^* = \beta^*$.

Nonlinear programming methods: introduction

We now introduce the idea about two methods of non-linear programming that are *penalty function method* and *augmented Lagrangian method*. We describe these approaches in the context of the equality-constrained problem

$$(P_1) \quad \begin{cases} \min\limits_{\mathbf{x}} f(\mathbf{x}) \\ h_i(\mathbf{x}) = 0, \ i = 1, \ldots, p. \end{cases} \tag{4.17}$$

which is a special case of (4.14).

Penalty function method

The idea is to add a term, to the objective function, which turns a constrained optimization problem to an unconstrained one. The *quadratic penalty function* $Q(\mathbf{x}; \lambda)$ for this formulation is

$$Q(\mathbf{x}; \lambda) = f(\mathbf{x}) + \frac{\lambda}{2} \|h(\mathbf{x})\|^2 = f(\mathbf{x}) + \frac{\lambda}{2} \sum_{i=1}^{n} h_i^2(\mathbf{x}) \tag{4.18}$$

where $\lambda > 0$ is the *penalty parameter*.

The function $Q(\mathbf{x}; \lambda)$ is called *penalty function*, the term $\frac{\lambda}{2} \|h(\mathbf{x})\|^2$ is called *penalty term*. The penalty term is a measure of violation of the constraints. It is nonzero when the constraints are violated and is zero in the region where constraints are not violated.

By driving λ to ∞, we penalize the constraint violations with increasing severity. It makes good intuitive sense to consider a sequence of values $\{\lambda_k\}$ with $\lambda_k \uparrow \infty$ as $k \to \infty$, and to seek the approximate minimizer $\mathbf{x}^k$ of $Q(\mathbf{x}; \lambda_k)$ for each k. Because the penalty term in Formula (4.18) is smooth, we can use the techniques from unconstrained optimization to search for $\mathbf{x}^k$.

Augmented Lagrangian functions

The augmented Lagrangian function of the problem P_1 includes the Lagrange function and the quadratic penalty term:

$$L_A(\mathbf{x}, \mu) = L(\mathbf{x}, \mu) + \frac{\lambda}{2} \|h(\mathbf{x})\|^2 = f(\mathbf{x}) + \sum_{i=1}^{p} \mu_i h_i(\mathbf{x}) + \frac{\lambda}{2} \sum_{i=1}^{p} h_i(\mathbf{x})^2$$

where $\lambda > 0$ is the penalty parameter.

We now design an algorithm that fixes the penalty parameter λ to some value $\lambda_k > 0$ at its kth iteration, fixes μ at the current state μ^k, and performs minimization with respect to $\mathbf{x}$. Using $\mathbf{x}^k$ to denote the approximate minimizer of $L_A(\mathbf{x}, \mu^k; \lambda_k)$, we have by the optimality conditions for unconstrained minimization (see Theorem 57) that

$$0 \approx \nabla_{\mathbf{x}} L_A(\mathbf{x}^k, \mu^k; \lambda_k) = \nabla f(\mathbf{x}^k) + \sum_{i=1}^{p} [\mu_i^k + \lambda_k h_i(\mathbf{x}^k)] \nabla h_i(\mathbf{x}^k).$$

By comparing with the KKT conditions (see Theorem 4.15), we can deduce that

$$\mu_i^* \approx \mu_i^k + \lambda_k h_i(\mathbf{x}^k).$$

This relation immediately suggests a formula for improving ourcurrent estimate μ^k of the Lagrange multiplier vector, using the approximate minimizer $\mathbf{x}^k$ just calculated: We can set

$$\mu_i^{k+1} = \mu_i^k + \lambda_k h_i(\mathbf{x}^k). \tag{4.19}$$

This discussion motivates the following algorithmic framework.

Algorithm 9 : Augmented Lagrangian Algorithm

Input: $\mathbf{x}^0 \in \mathbb{R}^n$, $\mu^1 \in \mathbb{R}^p$, $\lambda_1 > 0$, tolerance τ_1, number of iterations: N

Output: Optimum x^*

1: Set $k = 1$

2: Find a local minimum $\mathbf{x}^k$ of $L_A(\mathbf{x}, \mu^k; \lambda_k)$ using any unconstrained minimization algorithm, with starting point $\mathbf{x}^{k-1}$.

3: **if** $\left\|\nabla_{\mathbf{x}} L_A(\mathbf{x}^k, \mu^k; \lambda_k)\right\| \leqslant \tau_k$ **then**

4: **return** $\mathbf{x}^k$

5: Set $\mu^{k+1} = \mu^k + \lambda_k h(\mathbf{x}^k)$.

6: Set $\lambda_{k+1} = \beta\lambda_k$, with $\beta = 1$ if $\left\|h(\mathbf{x}^k)\right\| \leqslant \frac{1}{4}\left\|h(\mathbf{x}^{k-1})\right\|$ or $\beta > 1$ otherwise.

7: Select tolerance τ_{k+1}.

8: Set $k = k + 1$ and goto Step 2 if $k \leqslant N$.

9: **return** "the algorithm does not converge"

The major advantage of the method is that unlike the penalty method, it is not necessary to take $\lambda_k \to \infty$ in order to solve the original constrained problem. This intuitive discussion can be given a formal justification, as shown in the next statement.

Theorem 67. *Let $\mathbf{x}^*$ be a local solution of P_1 such that $\mathbf{x}^*$ is a regular point and the second-order conditions specified in Theorem 63 are satisfied at $\mu = \mu^*$. Then, there is a threshold value $\overline{\lambda}$ such that for all $\lambda > \overline{\lambda}$, $\mathbf{x}^*$ is a strict local minimizer of $L_A(\mathbf{x}, \mu^*; \lambda)$.*

Problem: Write a python code to implement the algorithm above.

Chapter 5

Machine learning: An introduction

5.1 Introduction

5.1.1 Definition

The term *Machine Learning* was coined by Arthur Samuel in 1959, an American pioneer in the field of computer gaming and artificial intelligence and stated that *"it gives computers the ability to learn without being explicitly programmed"* (See Wikipedia).

And in 1997, Tom Mitchell gave a "well-posed" mathematical definition of a learning algorithm that: *"A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E".*

The types of machine learning algorithms differ in their approach, the type of data they input and output, and the type of task or problem that they are intended to solve. Early classifications for machine learning approaches sometimes divided them into three broad categories. These were:

▷ **Supervised learning**: The computer is presented with example inputs and their desired outputs, given by a "teacher", and the goal is to learn a general rule that maps inputs to outputs.
▷ **Unsupervised learning**: No labels are given to the learning algorithm, leaving it on its own to find structure in its input. Unsupervised learning can be a goal in itself (discovering hidden patterns in data) or a means towards an end (feature learning).
▷ **Reinforcement learning**: It is a type of machine learning that has an agent (like a robot) that learns how to behave in an environment by taking actions and quantifying the results. If the agent makes a correct response, it gets a reward point, which it tries to maximise.

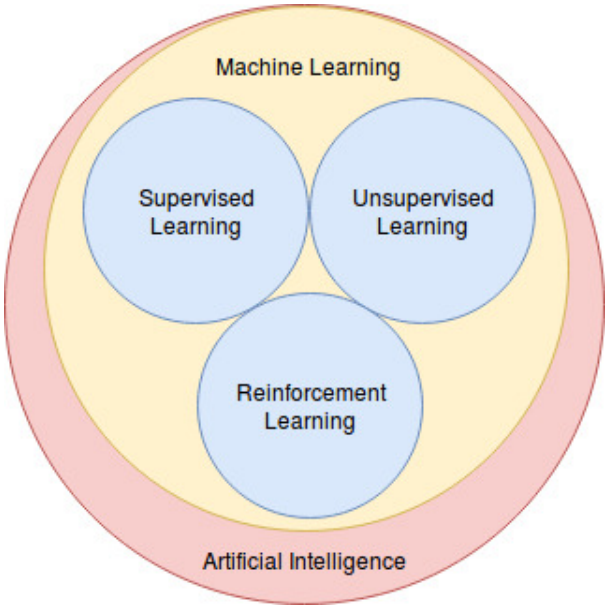


Figure 5.1: The AI Circle.

⋆ In the rest of this chapter we represent the ideas of the simplest algorithms of supervised learning and unsupervised learning, and use **Scikit-learn** to illustrate them. *Scikit-learn* (and also known as *sklearn*) is a free software machine learning library for the Python programming language.

5.1.2 Supervised learning setting

▷ Dataset

Supervised learning algorithms build a mathematical model of a set of data (or *dataset*) that contains both the inputs and the desired outputs. The data is known as *training data*, and consists of a set of training examples. Each training example has one or more inputs and the desired output.

In this chapter, we assume that each data point in the dataset is represented by a D-dimensional vector of real numbers. We will use N to denote the number of examples in a dataset. So a dataset is the same as an $N \times D$ matrix, where a row is represented a particular example, and the column is represented a specific *feature*.

▷ Models as Functions

Assume that we are given a dataset consisting of N input points:

$$X = \{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N\},\ \mathbf{x}^n \in \mathbb{R}^D \text{ for } n = 1, \ldots, N$$

and N corresponding output values

$$Y = \{y_1, y_2, \ldots, y_N\},\ y_n \in \mathbb{R} \text{ for } n = 1, \ldots, N,$$

where $\mathbf{x}^n$ is the n-th feature vector of the dataset and y_n is its *label*.

A *learning algorithm* seeks a function f that models the relationship between X and Y. Typically, f is a parametric function, depending on parameters $\theta \in \mathbb{R}^p$.

The goal of learning is to find a model and its corresponding parameters such that the resulting function will perform well on unseen data, i.e. that can be used to predict the output associated with new inputs with reasonable error.

Intuitively, the model is said to fit the data well if the predicted vector $\hat{\mathbf{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_N)$, where $\hat{y}_n = f(\mathbf{x}^n)$, is close to the vector $\mathbf{y}$.

▷ Some concepts of machine learning:

1. The ability to perform well on previously unseen inputs is **generalization**.
2. **Underfitting** is the condition where the model could not fit the data well enough.
3. **Overfitting** is the opposite case of underfitting, i.e., when the model predicts very well on training data and is not able to predict well on test data or validation data.

▷ Types of Supervised Learning:

Supervised learning problems can be further grouped into *Regression* and *Classification* problems.

- Regression: A regression problem is when the output variable is a real or continuous value. Many different models can be used, the simplest is the *linear regression*. It tries to fit data with the best hyper-plane which goes through the points.
- Classification: A classification problem is when the output variable is a category, i.e. each input data is assigned a label from a given finite set (called class labels). There are a number of classification models, the simplest is *k-nearest neighbours* (k-NN for short) which takes the k nearest neighbours of a query instance and assigns the majority class.

5.1.3 Unsupervised learning setting

Unsupervised learning involves finding hidden structure in unlabeled data. The most commonly used unsupervised machine learning technique is *clustering*. Clustering can be defined as the process of organizing the dataset into groups whose members are similar in some way.

Figure 5.2: A dataset with 4 clusters.

K-means clustering is one of the simplest and popular unsupervised machine learning algorithms. The K-means algorithm identifies k number of centroids, and then allocates every data point in the dataset to the nearest cluster, while keeping the centroids as small as possible.

5.2 Linear Regression

5.2.1 Regression

Regression is a supervised learning task, where we assume we are given a dataset consisting of N input points

$$X = \{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N\},\ \mathbf{x}^n \in \mathbb{R}^D$$

and N corresponding output values $Y = \{y_1, y_2, \ldots, y_N\}$.

We wish to find a function $f\colon \mathbb{R}^n \to \mathbb{R}$ that represent the relationship between X and Y. In the case of linear regression, we consider the functions f that are linear

$$f(\mathbf{x}) = a_0 + a_1 x_1 + a_2 x_2 + \cdots + a_D x_D$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_N) \in \mathbb{R}^D$ is data input, and $\mathbf{a} = (a_0, a_1, \ldots, a_D) \in \mathbb{R}^{D+1}$ is the parameter vector of the linear regression model.

▷ Finding the Parameter Vector

For each vector $\mathbf{x}^n \in X$, write $\mathbf{x}^n = (x_1^n, x_2^n, \ldots, x_D^n)$. The predicted values of the model on the dataset X are

$$\hat{y}_n = f(\mathbf{x}^n) = a_0 + a_1 x_1^n + a_2 x_2^n + \cdots + a_D x_D^n, \quad \text{for } n = 1, 2, \ldots, N.$$

We define the residue by $e_n = y_n - \hat{y}_n$, for $n = 1, 2, \ldots, D$, and the loss function by

$$L(\mathbf{a}) = \frac{1}{2}\sum_{n=1}^{N} e_n^2 = \frac{1}{2}\sum_{n=1}^{N}(y_i - a_0 - a_1 x_1^n - \cdots - a_D x_D^n)^2.$$

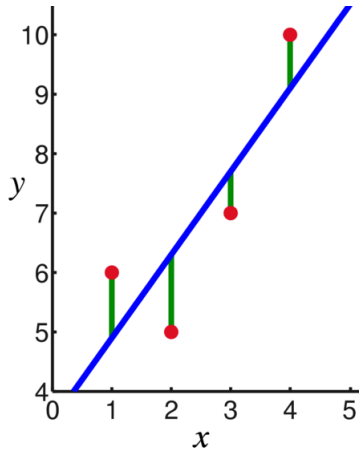


Figure 5.3: Dataset (red), linear function f (blue), residues (green).

The common techniques for finding $\mathbf{a}$ is the *least-squares estimation*: the parameter vector $\mathbf{a}$ is defined as such that minimizes the sum of mean squared loss

$$\mathbf{a} = \arg\min_{\mathbf{a}} L(\mathbf{a}).$$

Let

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}, \ \hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_N \end{bmatrix}, \ \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix} = \mathbf{y} - \hat{\mathbf{y}},$$

and

$$M = \begin{bmatrix} 1 & x_1^1 & x_2^1 & \cdots & x_D^1 \\ 1 & x_1^2 & x_2^2 & \cdots & x_D^2 \\ \vdots & & & & \\ 1 & x_1^N & x_2^N & \cdots & x_D^N \end{bmatrix} \in \mathbb{R}^{N\times(D+1)}.$$

We have

$$\hat{\mathbf{y}} = M\mathbf{a},\ \mathbf{y} = \widehat{\mathbf{y}} + \mathbf{e} = M\mathbf{a} + \mathbf{e},$$

and

$$\begin{aligned} L(\mathbf{a}) &= \frac{1}{2}\sum_{n=1}^{N} e_n^2 = \frac{1}{2}\left\|\mathbf{y} - \widehat{\mathbf{y}}\right\|^2 = \frac{1}{2}\left\|\mathbf{y} - M\mathbf{a}\right\|^2 = \frac{1}{2}\left\|M\mathbf{a} - \mathbf{y}\right\|^2 \\ &= \frac{1}{2}(M\mathbf{a} - \mathbf{y})^T(M\mathbf{a} - \mathbf{y}) = \frac{1}{2}\left(\mathbf{y}^T\mathbf{y} - \mathbf{y}^T M\mathbf{a} - \mathbf{a}^T M^T\mathbf{y} + \mathbf{a}M^T M\mathbf{a}\right). \end{aligned}$$

It follows that the gradient of the loss function is

$$\begin{aligned} \frac{\partial L(\mathbf{a})}{\partial \mathbf{a}} &= \frac{1}{2}\left(\mathbf{y}^T\mathbf{y} - \mathbf{y}^T M\mathbf{a} - \mathbf{a}^T M^T\mathbf{y} + \mathbf{a}M^T M\mathbf{a}\right) = -M^T\mathbf{y} + M^T M\mathbf{a} \\ &= (M^T M)\mathbf{a} - M^T\mathbf{y}. \end{aligned}$$

Setting the gradient to zero produces the system of linear equations for the optimum parameter vector $\mathbf{a}$:

$$(M^T M)\mathbf{a} = M^T\mathbf{y}.$$

Note that this linear system always has solutions. In the case $M^T M$ is inverse, we have $\mathbf{a} = (M^T M)^{-1}(M^T\mathbf{y})$.

5.2.2 Coefficient of Determination

When we use the dataset X and the output Y for training a model, i.e. finding a function $f\colon \mathbb{R}^n \to \mathbb{R}$. Then the predicted values of the model on the dataset X are

$$\hat{y}_n = f(\mathbf{x}^n),\ \text{for } n = 1, 2, \ldots, N.$$

It is important to know how the model could fit the data well enough. If it fits poorly then the model can not be used to predict anything. One measure we use for this purpose is the *coefficient of determination*, denoted R^2 or r^2 and pronounced *"R squared"*.

Define the *residuals* as $e_n = y_n - \hat{y}_n$, for $n = 1, 2, \ldots, N$. If $\overline{y}$ is the mean of the observed data:

$$\overline{y} = \frac{1}{N}\sum_{n=1}^{N} y_n$$

then we define

- The *total sum of squares*:

$$SS_{tot} = \sum_{n=1}^{N}(y_i - \overline{y})^2.$$

- The *regression sum of squares*, also called the *explained sum* of squares:

$$SS_{reg} = \sum_{n=1}^{N} (\hat{y}_i - \overline{y})^2.$$

- The *sum of squares of residuals*, also called the *residual sum of squares*:

$$SS_{res} = \sum_{n=1}^{N} (y_n - \hat{y_n})^2.$$

The most general definition of the *coefficient of determination* is

$$R^2 = 1 - \frac{SS_{reg}}{SS_{tot}}.$$

In the best case, the modeled values exactly match the observed values, which results in $SS_{res} = 0$ and $R^2 = 1$. Otherwise if R^2 is near 0, the relationship is poor, and if R^2 is near 1, the relationship is closed.

▷ Linear Regression

In the case the model is linear regression, we have $SS_{tot} = SS_{reg} + SS_{res}$, so that

$$R^2 = \frac{SS_{reg}}{SS_{tot}},$$

and R^2 ranges from 0 to 1.

The meaning of R^2 also write as

$$R^2 = \frac{\text{explained SS}}{\text{total SS}},$$

here *explained SS* means the sum of squares that can predict from linear regression.

5.2.3 Scikit-Learn

Source: https://scikit-learn.org/stable/

Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning (the module is known as *sklearn* in Python library). It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.

▷ Import necessary modules for Linear regression model from *sklearn*:

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4
5 from sklearn.linear_model import LinearRegression
6 import sklearn.metrics as metrics
```

where:

- Line 5: import module *LinearRegression* from *linear_model* in *sklearn*. This module use for the linear regression.

- Line 6: import module *metrics* from *sklearn*, which is used to compute R^2.

▷ Linear regression:

```
1 # Prepare data
2 x = np.array([5,7,8,7,2,2,9,4,11,12,9,6])
3 y = np.array([99,86,87,88,111,103,87,94,78,77,85,86])
4
5 # Reformat dataset
6 X = x.reshape(-1,1)
7
8 # Create a variable for Linear regression
9 reg = LinearRegression()
10
11 # Training model
12 reg.fit(X,y)
13
14 print(reg.coef_)
15 # Output: [−2.88416422]
16
17 print(reg.intercept_)
18 # Output: 109.79178885630498
```

where:

- Line 6: we need the dataset has the desired form which is each entry is a list (in this case each such list has only one element).

- Line 9: we create a variable for linear regression with name *reg* (the name is up to you).

- Line 12: we train the model. It gives a linear function

$$f(\mathbf{x}) = a_0 + a_1x_1 + a_2x_2 + \cdots + a_Nx_N$$

with minimum $L(\mathbf{a})$, where the vector $(a_1, a_2, \ldots, a_N)$ is *coef_* and a_0 is *intersept_*.

- Line 14: return the vector $(a_1, a_2, \ldots, a_D)$.

- Line 17: return the coefficient a_0.

▷ Visualization and R^2:

```
1  # Predict on the dataset
2  y_pread = reg.predict(X)
3
4  # Visualize the result
5  plt.scatter(x, y)
6  plt.plot(x, y_pred, color = 'red')
7  plt.show()
8
9  # Compute R^2 between y and y_pred
10 r2 = metrics.r2_score(y,y_pred)
11
12 print(r2)
13 #Output: 0.8604157164469495
```

where:

- Line 2: the syntax of prediction is $y = reg.predict(X)$, where X is an input dataset, and y is the output list.

- Line $5-7$: visualize the dataset (blue dots) and the linear function (red line) (see Figure 5.4).

- Line 10: compute R^2. The result is 0.8604157164469495, so that the model fits well on the training set. It indicates that we could use linear regression in future predictions.

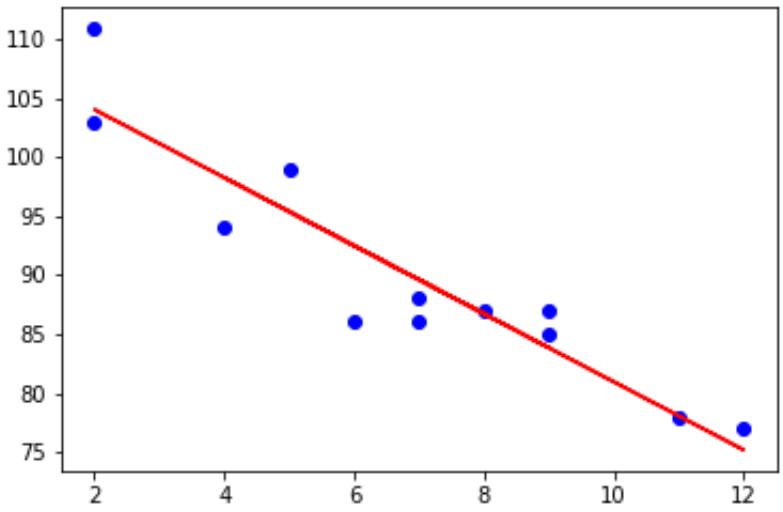


Figure 5.4: Linear Regression.

▷ Predict value at a point:

```
1 # Predict at point 10
2 x0 = 10
3 y0, = reg.predict([[x0]])
4
5 print(y0)
6 #Output: 80.95014662756597
```

Note: In Line 3, the result is a list with just one element, this command returns just such element not a list (*this a small trick in Python*).

▷ Training and Test Sets: Splitting Data

In Machine Learning when create a model to predict the outcome of certain events, we often divide the dataset into two subsets:

- *training set*: a subset to train the model.
- *test set*: a subset to test the model.

Train the model means create the model. Test the model means test the accuracy of the model.

▷ We may split the dataset by hand, or use the method *train_test_split* from *sklearn*:

```
1 # Import train_test_split
2 from sklearn.model_selection import train_test_split
```

```
3
4 # Create dataset
5 X = np.array([5,7,8,7,2,2,9,4,11,12,9,6])
6 y = np.array([99,86,87,88,111,103,87,94,78,77,85,86])
7
8 # Split dataset in to train set and test set
9 x_train, y_train, x_test,y_test = train_test_split(x,y, test_size
    =0.25)
```

where: in Line 9, we split the datasets (X, y) into train sets (X_train, y_train) and test sets (X_test, y_test) with 25% for test sets.

Example 68. We have a dataset about data sales (in thousands of units) depending upon advertising budgets (in thousands of dollars) for TV, radio, and newspaper media.

The source of this dataset is:

$$'https://raw.githubusercontent.com/neurospin/pystatsml/master/datasets/Advertising.csv'$$

Which has 4 columns and 200 rows, and here is the first 5 rows:

TV	Radio	Newspaper	Sales
230.1	37.8	69.2	22.1
44.5	39.3	45.1	10.4
17.2	45.9	69.3	9.3
151.5	41.3	58.5	18.5
180.8	10.8	58.4	12.9
...	...	...	...

For example in the row 1, if we invest the advertising budgets 230.1 thousand dollars for TV, 37.8 thousand dollars for Radio, 69.2 thousand dollars for Newspaper, then the sales is 22.1 thousand units of those products (with some scales).

▷ *We will create a machine learning for finding the relationship between advertising budgets and sales by using linear regression.*

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4
5 from sklearn.linear_model import LinearRegression
6 import sklearn.metrics as metrics
```

```
from sklearn.model_selection import train_test_split

# Read dataset

df = pd.read_csv('https://raw.githubusercontent.com/neurospin/
    pystatsml/master/datasets/Advertising.csv', index_col=0)

X = df[['TV', 'Radio', 'Newspaper']]
y =  df['Sales']

# Split dataset into train set and test set
X_train, X_test, y_train,y_test = train_test_split(X,y, test_size =
    0.2)

# Training model
reg = LinearRegression()
reg.fit(X_train,y_train)

# compute R^2 on train set
y_pred = reg.predict(X_train)
print("R squared =", metrics.r2_score(y_train, y_pred))
# Output: R squared = 0.8919137376029739

# compute R^2 on test set
y_pred_test = reg.predict(X_test)
print("R squared =", metrics.r2_score(y_test, y_pred_test))
# Output: R squared = 0.9140843342530077

# Predict: when we invest advertising budgets TV = 149, Radio = 22,
    Newspaper = 25, then Sales =?

sales, = reg.predict([[149,22,25]])
print(sales)
#Output: 13.871924633166394
```

Remark 69. R^2 of the linear regression on the train set and the test set indicate that the linear regression performs well on these sets.

5.3 k-Nearest Neighbors

5.3.1 Classification

Classification is a supervised learning task, where we assume we are given a dataset consisting of N input points

$$X = \{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N\},\ \mathbf{x}^n \in \mathbb{R}^D$$

and N corresponding output values $Y = \{y_1, y_2, \ldots, y_N\} \subseteq \mathcal{C}$, where $\mathcal{C}$ is a finite set.

Write $\mathcal{C}$ as $\mathcal{C} = \{c_1, c_2, \ldots, c_r\}$; each c_j is called a *class*. If $y_n = c_j$ for some j, we say that $\mathbf{x}^n$ belongs to the class c_j. It means that the dataset X is divided into classes.

The task of the *classification algorithm* is to find a mapping function

$$f\colon\ \mathbb{R}^n \to \mathcal{C}$$

that map the input $\mathbf{x} \in \mathbb{R}^n$ to a class $y \in \mathcal{C}$. The function f is called a *classifier*.

To measure the performance of the classifier, let $\hat{y}_n = f(\mathbf{x}^n)$, for $n = 1, \ldots, N$, and

$$\hat{N} = \#\{n \mid y_n = \hat{y}_n, n = 1, \ldots, N\},$$

then the *accuracy* of the classifier is

$$\frac{\hat{N}}{N} \cdot 100\%.$$

• The classification tasks can divide into two groups: *binary classifications* if that have two class labels and *multi-class classifications* if that have more than two class labels.

5.3.2 k-NN

Let k be a positive integer, typically small. The k-nearest neighbors algorithm (k-NN for short) is a non-parametric method used for classification as follows.

For an object $\mathbf{x} \in \mathbb{R}^D$, the algorithm first search for k points in the search space X which is the most closed to $\mathbf{x}$, and then assigns $\mathbf{x}$ to the class most common among its k nearest neighbors. If $k = 1$, then the object is simply assigned to the class of that single nearest neighbor.

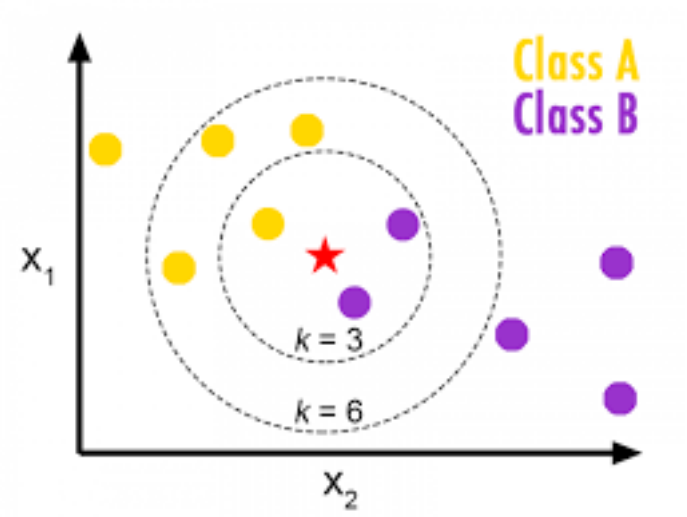


Figure 5.5: The k-NN works depending on k.

Remark 70. k-NN does not need training to get the parameter k for the classification; it does not learn anything from the training data, k is given, and simply uses the training data itself for classification.

▷ Example

We look at a simple binary classification example to predict customer's T-shirt sizes from their height and weight:

Height (cm)	Weight (kg)	Size
158	58	M
158	59	M
158	63	M
160	59	M
160	60	M
163	60	M
163	61	M
160	64	L
163	64	L
165	61	L
165	62	L
165	65	L
168	62	L
168	63	L
168	66	L
170	63	L
170	64	L
170	68	L

Suppose we have a test query that is a customer with height **161cm** and weight **61kg**.

▷ We will choose $k = 5$ nearest neighbours to make predictions.

```
import pandas as pd
import numpy as np
from sklearn import metrics
from sklearn.neighbors import KNeighborsClassifier


# Read dataset
df = pd.read_csv('https://raw.githubusercontent.com/trannamtrung/
    Database/master/Customer.csv')

# Declare k-NN with k = 5
clf = KNeighborsClassifier(n_neighbors=5)

# Declare the training set
```

```
X = df[['Height', 'Weight']]
y = df['Size']

# Put training set to k-NN
clf.fit(X,y)

# Compute the accuracy on the training set
y_pred = clf.predict(X)
print("Accuracy: %.2f" % metrics.accuracy_score(y, y_pred))
# Output: Accuracy: 1.00

# Predict with input [[161,61]]:
person = [[161,61]]
print(clf.predict(person))
# Output: ['M']
```

Thus, the answer is M, and k-NN performs very well on the dataset.

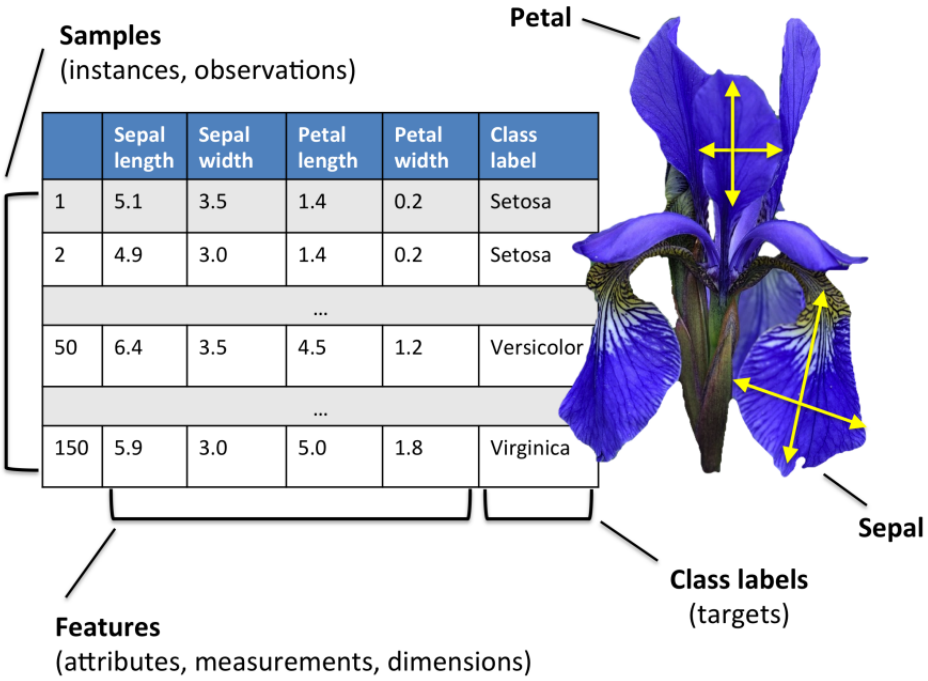
5.3.3 Iris Dataset

`Source: https://en.wikipedia.org/wiki/Iris_flower_data_set`

The **Iris dataset** consists of 50 samples from each of three species of **Iris** (*Iris setosa, Iris virginica* and *Iris versicolor*).

Figure 5.6: Iris flower family in Iris dataset.

Four features were measured from each sample: *the length* and *the width* of the **sepals** and **petals**, in *centimeters*.



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica

Figure 5.7: Iris dataset.

▷ Load Iris dataset

Source: `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html`

```
# Import Iris dataset from sklearn
from sklearn.datasets import load_iris

# Read dataset
iris = loat_iris()
```

Exploring *data fields* of iris:

- data: has 150 samples with 4 features each (sepal length, sepal width, petal length, petal width).
- target: A list corresponding to data which represents the species of each row: 0, 1, and 2 represent different species.
- target_names: Encoding scheme for species: 0 = *setosa*, 1 = *versicolor*, 2 = *virginica*.

- feature_names: The names of the four features.

▷ k-NN on Iris dataset

```
1  from sklearn import metrics
2  from sklearn.model_selection import train_test_split
3  from sklearn.neighbors import KNeighborsClassifier
4  from sklearn.datasets import load_iris
5
6  # Load Iris dataset
7  iris = load_iris()
8
9  # Create train sets and test sets
10 X, y = iris.data, iris.target
11
12 X_train, X_test, y_train, y_test = train_test_split(X,y, test_size =
       0.2)
13
14 # Create a k-NN with k = 3
15 clf = KNeighborsClassifier(n_neighbors=3)
16
17 clf.fit(X_train, y_train)
18
19 # Test the model on test set
20 y_pred = clf.predict(X_test)
21 print("Accuracy: %.2f" %metrics.accuracy_score(y_test, y_pred))
22 #Output: 0.97
23
24 # Predict
25 X_new = np.array([[3, 2, 4, 0.2], [  4.7, 3, 1.3, 0.2 ]])
26 y_new = clf.predict(X_new)
27 print("Species: {} and {}".format(iris.target_names[y_new[0]], iris.
       target_names[y_new[1]]))
28 # Output: Species: versicolor and setosa
```

Remark 71. Line 23 indicates that k-NN works well on Iris dataset.

5.4 K-Mean Clustering

Source: C. M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.

5.4.1 Definition

Typically, unsupervised algorithms make inferences from datasets using only input vectors without referring to known, or labelled, outcomes. The simplest unsupervised algorithm is *k-means clustering*.

Consider a datatset $X = \{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N\}$ and a positive integer $K \leqslant N$. The K-mean clustering algorithm aims to partition X into k sets $S_1, S_2, \ldots, S_K$ such that which minimize the following objective function:

$$J = \sum_{i=1}^{K} \sum_{\mathbf{x} \in S_i} \left\| \mathbf{x} - \mu^i \right\|^2,$$

where μ^i is the mean point of points in S_i.

5.4.2 Algorithm

The most common algorithm uses an iterative refinement technique. For each data point $\mathbf{x}^n$, we introduce a corresponding set of binary indicator variables $r_{nk} \in \{0, 1\}$, where $k = 1, \ldots, K$, that describes which of the K clusters the data point $\mathbf{x}^n$ is assigned to, so that if data point $\mathbf{x}^n$ is assigned to cluster k then $r_{nk} = 1$, and $r_{nj} = 0$ for $j \neq k$. We can then define an objective function, denoted *inertia*, as

$$J(r, \mu) = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \left\| \mathbf{x}^n - \mu^k \right\|^2,$$

which represents the sum of the squares of the Euclidean distances of each data point to its assigned vector μ^k. Our goal is to find values for the $\{r_{nk}\}$ and the $\{\mu^k\}$ so as to minimize the function $J(r, \mu)$. We can do this through an iterative procedure in which each iteration involves two successive steps corresponding to successive optimizations with respect to the r_{nk} and the μ^k.

• We start by choosing some initial values for the μ^k. Then in the first phase we minimize J with respect to the r_{nk}, keeping the μ^k fixed. In the second phase we minimize J with respect to the μ^k, keeping r_{nk} fixed. This two-stage optimization process is then repeated until convergence.

• The first phase: Consider first the determination of the r_{nk}. Because J in is a linear function of r_{nk}, this optimization can be performed easily to give a closed form solution. The terms involving different n are independent and so we can optimize for each n separately by choosing r_{nk} to be 1 for whichever value of k gives the minimum value of $\left\| \mathbf{x}^n - \mu^k \right\|^2$. In other words, we simply assign the nth data point to the closest

cluster centre. More formally, this can be expressed as

$$r_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_j \|\mathbf{x}^n - \mu^j\|^2, \\ 0 & \text{otherwise.} \end{cases}$$

• The second phase: Now consider the optimization of the μ^k with the r_{nk} held fixed. The objective function J is a quadratic function of μ^k, and it can be minimized by setting its derivative with respect to μ^k to zero giving

$$2\sum_{n=1}^{N} r_{nk}(\mathbf{x}^n - \mu^k) = 0$$

which we can easily solve for μ^k to give

$$\mu^k = \frac{\sum_{n=1}^{N} r_{nk}\mathbf{x}^n}{\sum_{n=1}^{N} r_{nk}}.$$

The denominator in this expression is equal to the number of points assigned to cluster k, and so this result has a simple interpretation, namely set μ^k equal to the mean of all of the data points $\mathbf{x}^n$ assigned to cluster k. For this reason, the procedure is known as the K-means algorithm.

The two phases of re-assigning data points to clusters and re-computing the cluster means are repeated in turn until there is no further change in the assignments (or until some maximum number of iterations is exceeded). Because each phase reduces the value of the objective function J, convergence of the algorithm is assured. However, it may converge to a local rather than global minimum of J.

▷ K-Mean Clustering in Sklearn

Source for using K-Mean Clustering in Sklearn:
https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

```
from sklearn import cluster, datasets
import matplotlib.pyplot as plt

iris = datasets.load_iris()
X = iris.data[:, :2]  # use only 'sepal length and sepal width'
y_iris = iris.target

km2 = cluster.KMeans(n_clusters=2).fit(X) # K-Mean  with k = 2
km3 = cluster.KMeans(n_clusters=3).fit(X) # K-Mean  with k = 3
```

```
km4 = cluster.KMeans(n_clusters=4).fit(X) # K-Mean    with k = 4

plt.figure(figsize=(9, 3))
plt.subplot(131)
plt.scatter(X[:, 0], X[:, 1], c=km2.labels_)
plt.title("K=2, J=%.2f" % km2.inertia_)
plt.subplot(132)
plt.scatter(X[:, 0], X[:, 1], c=km3.labels_)
plt.title("K=3, J=%.2f" % km3.inertia_)
plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1], c=km4.labels_)#.astype(np.float))
plt.title("K=4, J=%.2f" % km4.inertia_)
```

Result:

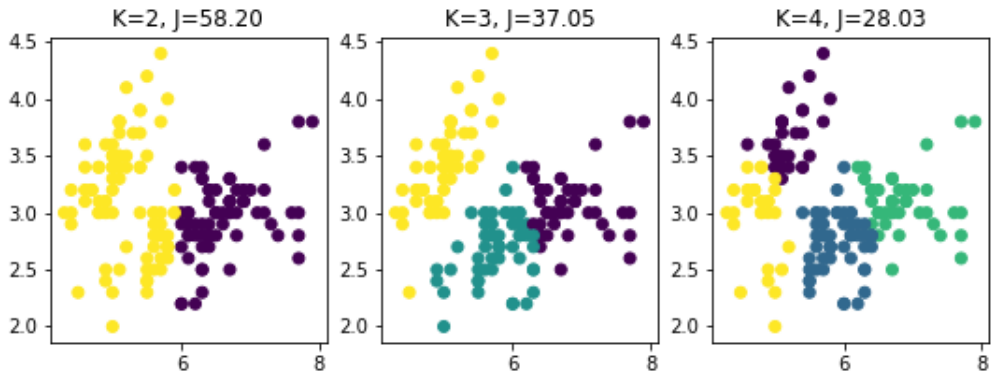


Figure 5.8: The K-Mean Clustering works depending on K.

Bibliography

[1] C. M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.

[2] R. L. Burden and J. D. Faires, Numerical Analysis, Ninth Edition. Cengage Learning, 2011.

[3] T. H. Cormen. C. E. Leiserson. R. L. Rivest and C. Stein, Introduction to Algorithms, Third Edition. The MIT Press.

[4] R. Diestel, Graph Theory, 5th Edition, Springer-Verlag, 2006.