For-Python-casting state-space using Python： a simple/practical approach

Call the required packages

```
import numpy as np
import pandas as pd
import re
import requests
from matplotlib import pyplot as plt
from scipy.stats import chi2_contingency
from scipy.stats import chi2
```

Working directory

It is important to establish the working directory as it will be the space where you will save data, scripts etc...For example, suppose you want to work with your memory stick (USB) this is the command to digit:

> Create and modify the working directory :
> PyCharm——>Settings——>Appearance&Behavior——>System Setting——>Project Opening——>Default directory

You can use Pycharm for many types of analysis.

Save your Script

If you want to save your code, in order to use it later, go to "File --> Save as" and give it the name you want. Just remember that the extension of the Script is "**.py**"
For example suppose you save a file and you call it MYFILE, then your script is saved in the working directory as follows: "**MYFILE.py**"

Save your data

Suppose that you want to save the dataset your are working on. You can use "**np.save**" and "**np.load**".

```
import numpy as np
a=[1,2,3,4]
np.save('a.npy',a)
b=np.load('a.npy')
print('b is',b)
```

Running Reslut:

```
b is [1 2 3 4]
```

Help when you do not know a command

Suppose you want to know what a command do when you use it. For example you want to know what the command "sum()". You can do it by using "**help**":

```
help(sum)
```

Running Result:

```
Help on built-in function sum in module builtins:

sum(iterable, start=0, /)
    Return the sum of a 'start' value (default: 0) plus an iterable of numbers

    When the iterable is empty, return the start value.
    This function is intended specifically for use with numeric values and may
    reject non-numeric types.
```

Vectors, Matrix and other stuff

Create a vector

```
x=np.array([2,6,1,3,11])  #创建一个数组
print('x是',x)
print('x的形状是',x.shape)
y=x.T                    #数组的转置没有任何作用
print('y是',y)
print('x的形状是',y.shape)
x_t=x.reshape(1,5)        #通过reshape将其转化成数组
print('x_t是',x_t)
print('x_t的形状是',x_t.shape)

x1=x_t.T
print('x_t形状是',x1.shape)
#x1.T.shape #返回数组形状
```

Running Result:

```
x是 [ 2  6  1  3 11]
x的形状是 (5,)
y是 [ 2  6  1  3 11]
x的形状是 (5,)
x_t是 [[ 2  6  1  3 11]]
x_t的形状是 (1, 5）
x_t形状是 (5, 1)
```

Create a sequence

Supposing you want to create a column from=-4, to=5,by=1.5.

```
x2=np.arange(-4,5.1,1.5)  #与R语言的不同点：R语言包含5，py不包含5
print(x2)
#注：如果未指定步长：默认以1为间隔
x3=np.arange(-4,5.1)
print(x3)
#注：只有上界时：
x5=np.arange(6)      #从0到5，不含6
print(x5)
#注：如果想要序列具有特定的长度：(4, 5, length.out = 5)
x4=np.linspace(4,5,5)      #注：此处R与py都包含5这个上界
print(x4)
```

Running Result:

```
[-4.  -2.5 -1.   0.5  2.   3.5  5. ]
[-4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
[0 1 2 3 4 5]
[4.   4.25 4.5  4.75 5.  ]
```

Create duplicate numbers

```
#创建一个向量（一列数字）,该向量重复数字2 5次
x6=np.ones(5)
x6_1=x6*2
x6_2=x6_1.reshape(5,1)
print(x6)
print(x6_2.shape)
#创建一个向量（一列数字）,该向量重复数字3 4次
x7=np.ones(4)
x7_1=x7*3
x7_2=x7_1.reshape(4,1)
print(x7_2)
#下面的命令创建一个3 x 3的矩阵，在矩阵的每个元素上都是4 （4，3，3）
x8=np.ones((3,3))
x8_1=x8*4
print(x8_1)
#下面的命令将向量按照指定形状生成矩阵（创建一个2 x 2矩阵）。（11，5，9，2）
x9=np.array([11,5,9,2]).reshape(2,2)
print(x9)
#下面的命令使用向量元素创建向量矩阵。(4*1)
x10=np.array([11,5,9,2]).reshape(4,1)
print(x10)
#下面的命令使用向量元素创建向量矩阵。(1*4)
x11=np.array([11,5,9,2]).reshape(1,4)
print(x11)
```

Running Result:

```
[1. 1. 1. 1. 1.]
(5, 1)
[[3.]
 [3.]
 [3.]
 [3.]]
```

```
[[4. 4. 4.]
 [4. 4. 4.]
 [4. 4. 4.]]
[[11  5]
 [ 9  2]]
[[11]
 [ 5]
 [ 9]
 [ 2]]
[[11  5  9  2]]
```

Create a specific sequence

```
#由0组成的数组
a=np.zeros((3,4))
#由1组成的数组
a1=np.ones((2,3,4),dtype=np.int16)  #3行4列2层
#向量拼接 (21, 5, 2, 15) (- 3, - 6, 1, - 7)(102, 10, - 13, 4) 4行3列
a2_1=np.array([21,5,2,15]).reshape(1,4)
a2_2=np.array([- 3,- 6,1,- 7]).reshape(1,4)
a2_3=np.array([102,10,- 13,4]).reshape(1,4)
a2=np.r_[a2_1,a2_2,a2_3]#按照行拼接row
print(a2)
a2_4=np.c_[a2_1.T,a2_2.T,a2_3.T]#按照列拼接column
print(a2_4)
#向量的第n个元素  从0开始
a3=np.array([21,5,2,15])
print(a3[3])
#向量第m到n个元素
print(a3[0:2])#不包含截止位置的元素
#向量特定位置的几个元素   两个位置的中括号
print(a3[[0,2,3]])
#m到n之间大小的元素
e=(a3>1) & (a3<16)
print(a3[e])
#矩阵第i行第j列元素
print(a2_1[0,2])#返回第一行第三列元素
#矩阵第i行
print(a2_1[0,:])#返回第一行第元素
#矩阵第j列
print(a2_1[:,2])#返回第三列元素
```

Running Result:

```
[[ 21   5   2  15]
 [ -3  -6   1  -7]
 [102  10 -13   4]]
[[ 21  -3 102]
 [  5  -6  10]
 [  2   1 -13]
 [ 15  -7   4]]
15
[21  5]
[21  2 15]
[ 5  2 15]
```

```
2
[21  5  2 15]
[2]
```

Delete element & create an object containing a series of matrices

```
#假设您有一个向量，并且想要删除一些元素（4，22，56，77，26，88，100）
b1=np.array([4,22,56,77,26,88,100])
#删除第4个    减掉1
b2=np.delete(b1,3)
b3=np.delete(b1,[3,4])
print(b2)
#删除第1-2个
b4=np.delete(b1,np.arange(0,1.1))
print(b4)
#创建16个来自（0，1）正态分布的数，转成4*4矩阵
b5=np.random.normal(loc=0,scale=1,size=16)#均值mean,标准差std,数量
b5_1=b5.reshape(4,4)
print(b5_1)
#去掉2 4行
b5_2=np.delete(b5_1,[1,3],0)
print(b5_2)
#去掉2 4列
b5_3=np.delete(b5_1,[1,3],1)
print(b5_3)
#去掉1 4行，1 2列
b5_4=np.delete(b5_1,[0,3],0)
b5_5=np.delete(b5_4,[0,1],1)
print(b5_5)
#创建包含一系列矩阵的对象
#用1-8这8个数创建一个2*4的矩阵 1 : 3，ç（2，4）
b6=np.arange(1,8.1)
b6_1=b6.reshape(2*4)
print(b6)
#用1-12这12个数创建3个2*2的矩阵对象 1 : 8，ç（2，2，3）
b7=np.arange(1,12.1)
b7_1=b7.reshape(2,2,3)
print(b7_1)
```

Running Result:

```
[  4  22  56  26  88 100]
[ 56  77  26  88 100]
[[-0.55922145  1.41685327  1.58616196 -1.80010491]
 [ 1.54532396  0.21644551  0.84566025 -0.26671643]
 [ 0.72692367  0.48878484  0.49671579 -1.36137115]
 [-0.35157659 -1.14070682 -0.87116449 -1.09381494]]
[[-0.55922145  1.41685327  1.58616196 -1.80010491]
 [ 0.72692367  0.48878484  0.49671579 -1.36137115]]
[[-0.55922145  1.58616196]
 [ 1.54532396  0.84566025]
 [ 0.72692367  0.49671579]
 [-0.35157659 -0.87116449]]
[[ 0.84566025 -0.26671643]
```

```
 [ 0.49671579 -1.36137115]]
[1. 2. 3. 4. 5. 6. 7. 8.]
[[[ 1.  2.  3.]
  [ 4.  5.  6.]]

 [[ 7.  8.  9.]
  [10. 11. 12.]]]
```

Matrix Computation

You can add/subtract/multiply/divide vectors with the same dimensions. Some examples are below:

```
#矩阵加，减，乘，除 （3，2，6） （- 2，- 1，5）
v1=np.array([3,2,6])
v2=np.array([-2,-1,5])
print(v1+v2)
print(v1-v2)
print(v1*v2)
print(v1/v2)
#可以使用符号*乘以具有不同尺寸的向量,创造矩阵（对应位置相乘）
v1_1=v1.reshape(1,3)
v2_1=v2.reshape(3,1)
print(v1_1*v2_1)
#当第一个矩阵的列数与第二个矩阵的行数相同时，作矩阵乘法(RNORM（6），2，3) (RNORM（9），3，3)
v3=np.random.normal(loc=0,scale=1,size=6)
v3_1=v3.reshape(2,3)
v4=np.random.normal(loc=0,scale=1,size=9)
v4_1=v4.reshape(3,3)
v5=np.matmul(v3_1,v4_1)
print(v5)
print(v5.shape)
#逆矩阵求法:   需要先转化成矩阵类型
v4_2=np.matrix(v4_1)
v6=v4_2.I
print(v6)
```

Running Result:

```
[ 1  1 11]
[5 3 1]
[-6 -2 30]
[-1.5 -2.   1.2]
[[ -6  -4 -12]
 [ -3  -2  -6]
 [ 15  10  30]]
[[-1.02336926  2.26229548 -0.01003356]
 [-0.0818821   1.78253906  0.87510632]]
(2, 3)
[[ 0.81327773  0.31466828  0.06991784]
 [ 0.17244778  0.32910265 -0.374298  ]
 [ 1.3315214  -0.34057171 -0.47523605]]
```

Create a list of objects

Suppose that you have several object of different types and you want to collect them in a list (one object). You can do that as follows:

```
c1=3
c2=np.ones(15)
c3=np.arange(1,8.1)
c3_1=c3.reshape(2,4)
c3_2=np.matrix(c3_1)
c=[c1,c2,c3_2]
print(c)
print(c[0])
```

Running Result:

```
[3, array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]),
matrix([[1., 2., 3., 4.],
        [5., 6., 7., 8.]])]
3
```

Writing your own function

Suppose you want to create a function you want to use several times or in different context. R allows you to do this. For example:

```
#函数定义：这是一个简单的函数，返回x对象的均值
def my_mean(x):
    return sum(x)/len(x)
#函数调用
my_x=np.arange(1,4.1)
my_Mean=my_mean(my_x)
print(my_Mean)
```

Running Result:

```
2.5
```

> Correlation functions:
>
> Function: the variance of a variable.
>
> Function: the covariance between two variables.
>
> Function: the correlation between two variables.

Remember that the correlation is the ratio between the covariance of the variables and the product of the standard deviations of them. Also, remember that the covariance of x and y is the mean of x times y minus the product of the mean of x times the mean of y. Also, remember that the standard deviation is the squared root of the variance. Finally, the variance of x is the mean of the squared values of x minus the square of the mean of x. That is Variance of x is equal to $\left[mean(x^2) - (mean(x))^2\right]$.

Note that while the correlation function that you created returns the same value of the "cor" function of R, there is a difference between the "var" function and your variance and the "cov" function and your covariance. This is because R divides by n-1 while you divide by n. The n-1 is a correction used for small samples. Here we do not need to discuss this further.

The for loop and the if condition in Python

Loop function: Suppose you want to repeat the same procedure several times

If condition: The condition (if) makes a procedure when something happens or not (does not make it).

```
#for循环打印1：5
for i in range(1,6):
    print(i)
#runif（n，min，max）??在最小值和最大值之间生成n个随机均匀值。在下面的示例中，当数字<6时，您会看到打印件，否则看不到。
np.random.seed(12)
d1=np.random.uniform(3,4,(1,7))
print(d1)
index2=np.arange(0,d1.shape[1])
print(index2)
for i in index2:   #shape[0]表示行，shape[1]表示列
    if d1[0,i]<3.5 :
        print("小于3.5")
    elif (d1[0,i]>=3.5)&(d1[0,i]<3.7):
        print("大于3.5，小于3.7")
    else:
        print("大于等于3.7")
```

Running Result:

```
1
2
3
4
5
[[3.15416284 3.7400497  3.26331502 3.53373939 3.01457496 3.91874701
  3.90071485]]
[0 1 2 3 4 5 6]
小于3.5
大于等于3.7
小于3.5
大于3.5，小于3.7
小于3.5
大于等于3.7
大于等于3.7
```

Consider another example:

```
#实例：
#下面创建随机均匀值的矢量y（20个元素）。其次，创建另一个称为z的向量（未知维），并且如果y的第i个值<= 0，则将相对的z[i]值指定为-1。否则，如果y> 0，则会将z [i]的值指定为1
np.random.seed(13)
y2=np.random.uniform(-10,10,(1,20))
print(y2)
z=[]
for i in range(0,20):
    if y2[0,i]<=0:
        z=np.append(z,-1)          #向量插入
    else:
        z=np.append(z,1)
print(z)
```

Running Result:

```
[[ 5.55404821 -5.2491756   6.48557065  9.31498396  9.45202228 -0.93101505
   2.18084926  5.51053029  2.8322669   4.44036459 -9.29926952 -4.03101058
  -8.82975016  7.14121885 -2.54291944  3.59695903 -4.87440101 -3.0483757
  -9.8117446  -2.83332435]]
[ 1. -1.  1.  1.  1. -1.  1.  1.  1.  1. -1. -1. -1.  1. -1.  1. -1. -1.
 -1. -1.]
```

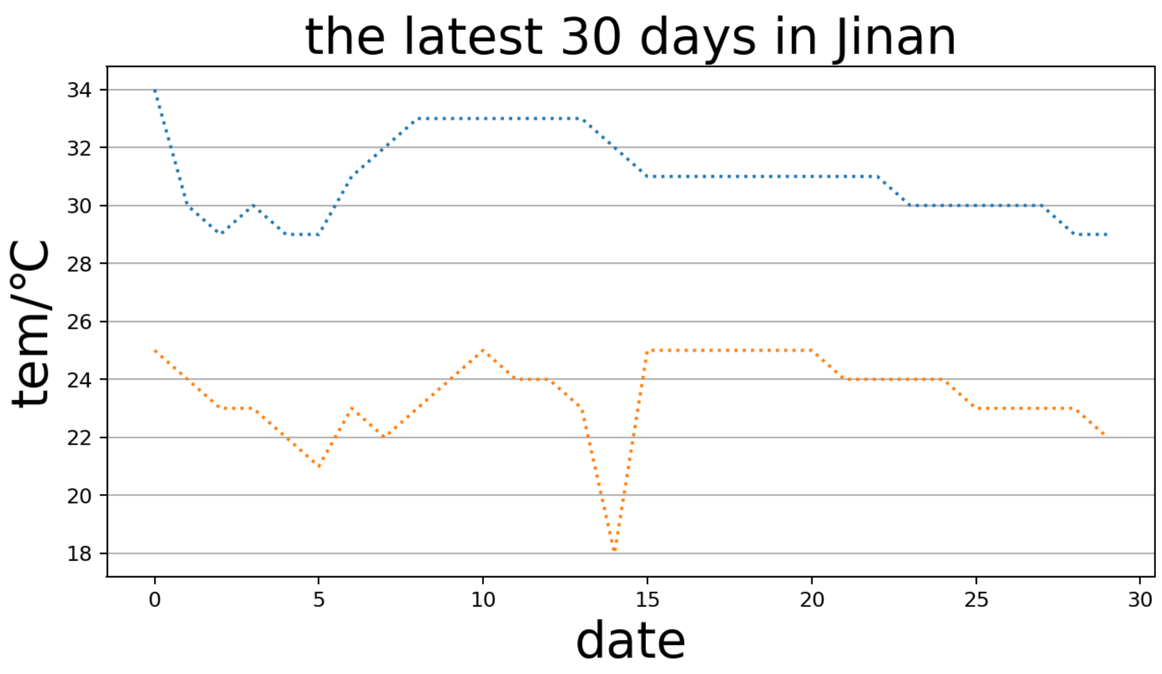
Data scraping from the Web: Reading data from HTML tables (web-scraping).

Suppose you want to work with data that are available from the Web.

```
#爬取数据并作图(由于无法访问国外网站，所以选取了一个国内网站做示例)
import re
import requests
from matplotlib import pyplot as plt
# 获取济南近30天的最低温和最高温
html = requests.get('https://www.yangshitianqi.com/jinan/30tian.html').text
#使用正则提取数据
pattern_temperature = r'<div class="fl i3 nz">(\d+~\d+)℃</div>'
pattern_date = r'<div class="t2 nz">(\d\d\.\d\d)</div>'
temperature = re.findall(pattern_temperature, html)
date = re.findall(pattern_date, html)
# 整理数据
max_d = [int(i.split('~')[1]) for i in temperature]
print(max_d)
min_d = [int(i.split('~')[0]) for i in temperature]
print(min_d)
# 定义图像质量
plt.figure(figsize=(9, 4.5), dpi=180)
# 解决中文乱码
plt.rcParams['font.sans-serif'] = ['SimHei']  # 用来正常显示中文标签
plt.rcParams['axes.unicode_minus'] = False  # 用来正常显示负号
# 绘制图像
plt.plot(range(30), max_d, linestyle=':')
plt.plot(range(30), min_d, linestyle=':')
# xy轴标识
plt.xlabel('date', size=24)
plt.ylabel('tem/℃', size=24)
plt.title('the latest 30 days in Jinan', size=24)
# 显示网格
plt.grid(axis='y')
# 保存图像
plt.savefig('a.png')
# 显示图像
plt.show()
```

Running Result:

```
[34, 30, 29, 30, 29, 29, 31, 32, 33, 33, 33, 33, 33, 33, 32, 31, 31, 31, 31, 31,
31, 31, 31, 30, 30, 30, 30, 30, 29, 29]
[25, 24, 23, 23, 22, 21, 23, 22, 23, 24, 25, 24, 24, 23, 18, 25, 25, 25, 25, 25,
25, 24, 24, 24, 24, 23, 23, 23, 23, 22]
```



Read xlsx data and do simple chart operation

```
#读取excel数据  dataRH.xlsx
dataRH=pd.read_excel('/home/yinaihua/Desktop/时间序列/期末作
业/weihai_pdf/dataRH.xlsx' )
#查看数据集的前6行
print(dataRH.head(6))
```

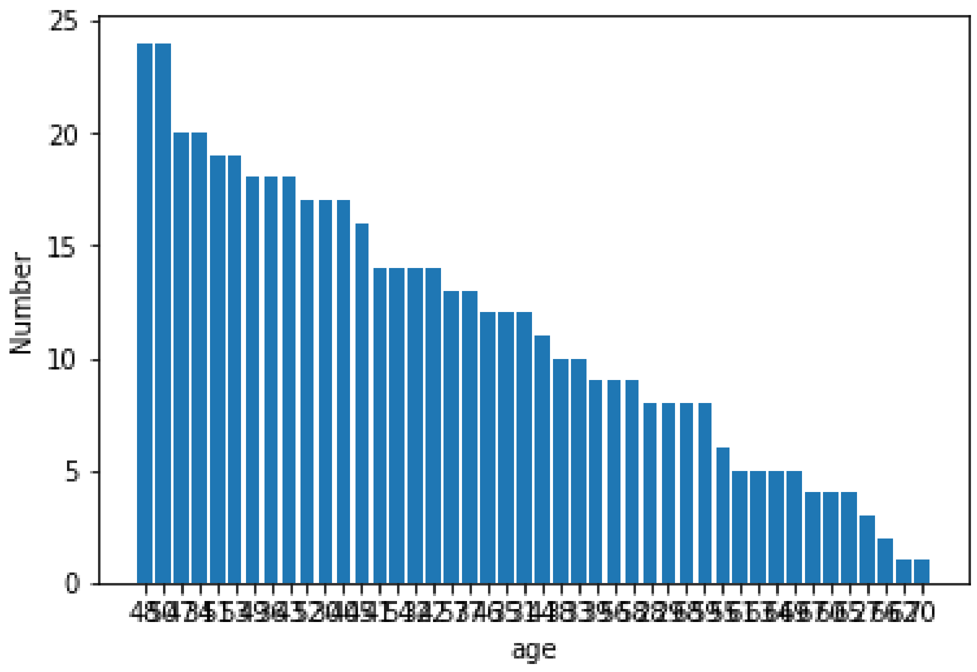
Running Result:

```
   sex  workexp  age  jobtitle  wage  edu
0    1        9   39         1  2370    1
1    1       16   31         1  3125    2
2    0        8   53         2  2620    1
3    1        6   35         1  2270    1
4    0        7   38         2  3315    3
5    0        8   59         3  3375    3
```

```
#查看读取的数据类型
print(type(dataRH))
#查看变量
#print(dataRH['age'].value_counts())
#绘制柱状图
plt.xlabel("age")
plt.ylabel("Number")
plt.xticks(np.arange(len(dataRH["age"].value_counts()))+0.5,dataRH["age"].value_
counts().index)
plt.bar(np.arange(len(dataRH["age"].value_counts()))+0.5,dataRH["age"].value_cou
nts())
plt.savefig('Age.png')  #保存图片
plt.show()
```

Running Result:

```
<class 'pandas.core.frame.DataFrame'>
```



```
#考虑两个变量来构建一个双变量表(!!!!!!!!!必须得是列联表)
dataRH_2 = pd.crosstab(dataRH['sex'],dataRH['wa'])
print(dataRH_2)
#练习：通过将变量切成不同的切片（片段）来修改变量
```

Running Reslut:

```
jobtitle    1    2   3
sex
0          41  135  36
1         198   17  73
```

Testing the difference between two means (quantitative variables)

Remember that the Null-hypothesis (H0 hypothesis) is that the difference between the two means is not significantly different. The alternative hypothesis (H1 hypothesis) is that the two means are significantly different.

```
import numpy as np
import pandas as pd
from scipy import stats
titanic = pd.read_excel('dataRH.xlsx')
# print(titanic.head(10))

table1 = pd.crosstab(titanic['sex'],titanic['jobtitle'],margins=False)
#交叉表，用于统计两个变量之间的数据个数。

print(table1)
result = stats.chi2_contingency(table1)#卡方检验函数
################################################################################
####
#第一个值为卡方值
#第二个值为P值
#第三个值为自由度
#第四个为与原数据数组同维度的对应理论值
################################################################################
#####
print(result)
```

Running results:

```
(62249.99999999997, 0.2394555650038306, 62001, array([[0.004, 0.004, 0.004, ...,
0.004, 0.004, 0.004],
       [0.004, 0.004, 0.004, ..., 0.004, 0.004, 0.004],
       [0.004, 0.004, 0.004, ..., 0.004, 0.004, 0.004],
       ...,
       [0.004, 0.004, 0.004, ..., 0.004, 0.004, 0.004],
       [0.004, 0.004, 0.004, ..., 0.004, 0.004, 0.004],
       [0.004, 0.004, 0.004, ..., 0.004, 0.004, 0.004]]))
```

> When the p-value is >0.05 than we accept the Null-hypothesis to conclude that there is no significant difference between the two means
>
> When the p-value is <0.05 than we reject the Null-hypothesis to conclude that there is a significant difference between the two means.

Consider this example:

```
data1 = np.random.randn(250,2)
#print(data1)
table1 = pd.crosstab(data1[:,0],data1[:,1],margins=False)#交叉表，用于统计两个变量之
间的数据个数。

result = stats.chi2_contingency(table1)#卡方检验函数
print(result)
```

Running results:

```
(2000.0000000000002, 0.0, 16, array([[23.762, 21.8  , 23.544, 21.364, 18.53 ],
       [21.8  , 20.   , 21.6  , 19.6  , 17.   ],
       [23.544, 21.6  , 23.328, 21.168, 18.36 ],
       [21.364, 19.6  , 21.168, 19.208, 16.66 ],
       [18.53 , 17.   , 18.36 , 16.66 , 14.45 ]]))
```

Are you able to interpret it? Both the way I generate the data and the results?

Now consider another example:

```
data2 = np.random.randint(5,size=500)
#print(data2)
data3 = np.ones(500)
data4 = np.add(data2,data3)
#print(data4)
table1 = pd.crosstab(data2,data4,margins=False)#交叉表，用于统计两个变量之间的数据个
数。
result = stats.chi2_contingency(table1)#卡方检验函数
print(result)
```

Running results:

```
(2000.0, 0.0, 16, array([[19.208, 21.168, 16.268, 21.56 , 19.796],
       [21.168, 23.328, 17.928, 23.76 , 21.816],
       [16.268, 17.928, 13.778, 18.26 , 16.766],
       [21.56 , 23.76 , 18.26 , 24.2  , 22.22 ],
       [19.796, 21.816, 16.766, 22.22 , 20.402]]))
```

Testing the difference between two means (quantitative variables)

Remember that the Null-hypothesis (H0 hypothesis) is that the difference between the two means is not significantly different. The alternative hypothesis (H1 hypothesis) is that the two means are significantly different.

```
from scipy import stats
data = pd.read_excel('dataRH.xlsx')

data1 = data['wage'][data['sex']==0]
data2 = data['wage'][data['sex']==1]

result = stats.ttest_ind(data1,data2)
print(result)
```

Running results:

```
Ttest_indResult(statistic=4.7142451538877355, pvalue=3.153616316732939e-06)
```

You can see the statistics together with the degrees of freedom and the p-value.

> When the p-value is >0.05 than we accept the Null-hypothesis to conclude that there is no significant difference between the two means
>
> When the p-value is <0.05 than we reject the Null-hypothesis to conclude that there is a significant difference between the two means.

Consider this example:

```
import math
from random import gauss

random_numbers = [gauss(1, 2) for i in range(500)]
data1 = [random_numbers[i]*25 +2000 for i in range(500)]
data1 = np.array(data1)
data = data1.reshape(250,2)
#print(data)
result = stats.ttest_ind(data[:,0],data[:,1])
print(result)
```

Running results:

```
Ttest_indResult(statistic=-1.095622052871073, pvalue=0.2737737728838749)
```

Are you able to interpret it? Both the way I generate the data and the results?

Now consider another example:

```
random_numbers = [gauss(1, 2) for i in range(500)]
data1 = [random_numbers[i]*20 +2500 for i in range(500)]
data1 = np.array(data1)
data1 = data1.reshape(250,2)

data2 = [random_numbers[i]*15 +2000 for i in range(500)]
data2 = np.array(data2)
data2 = data2.reshape(250,2)
#print(data)
result = stats.ttest_ind(data1,data2)
print(result)
```

Running results:

```
Ttest_indResult(statistic=array([144.58156265, 157.97710992]), pvalue=array([0.,
0.]))
```

Data visualization

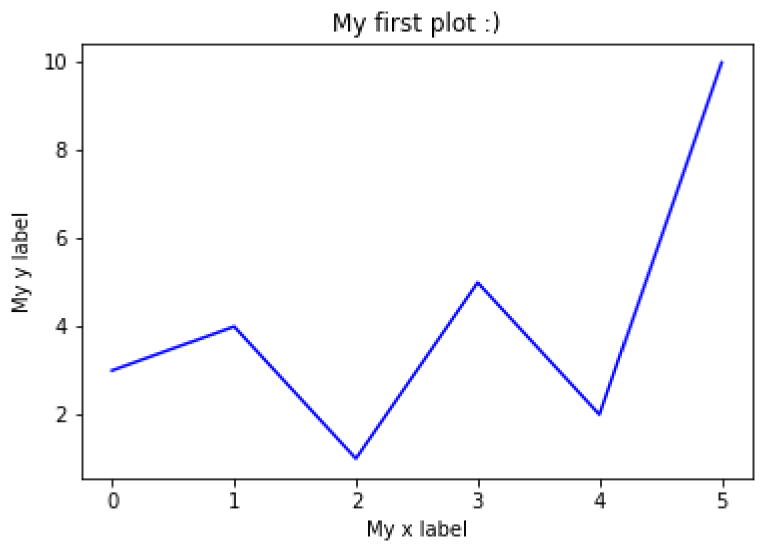
A quick introduction

We now step into the world of graphical respresentation with R. The material provided is an introduction about the potentials of R for data visualization. However, if you want to further increase your knowledge, you can find many R packages online that help you making amizing graphics.

The simplest and easier plot in R can be shown as follows:

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

y = [3,4,1,5,2,10]
plt.plot(y,color = "b")
plt.xlabel('My x label')
plt.ylabel('My y label')
plt.title('My first plot :)')
plt.savefig("1-1.png")
```

Running results:

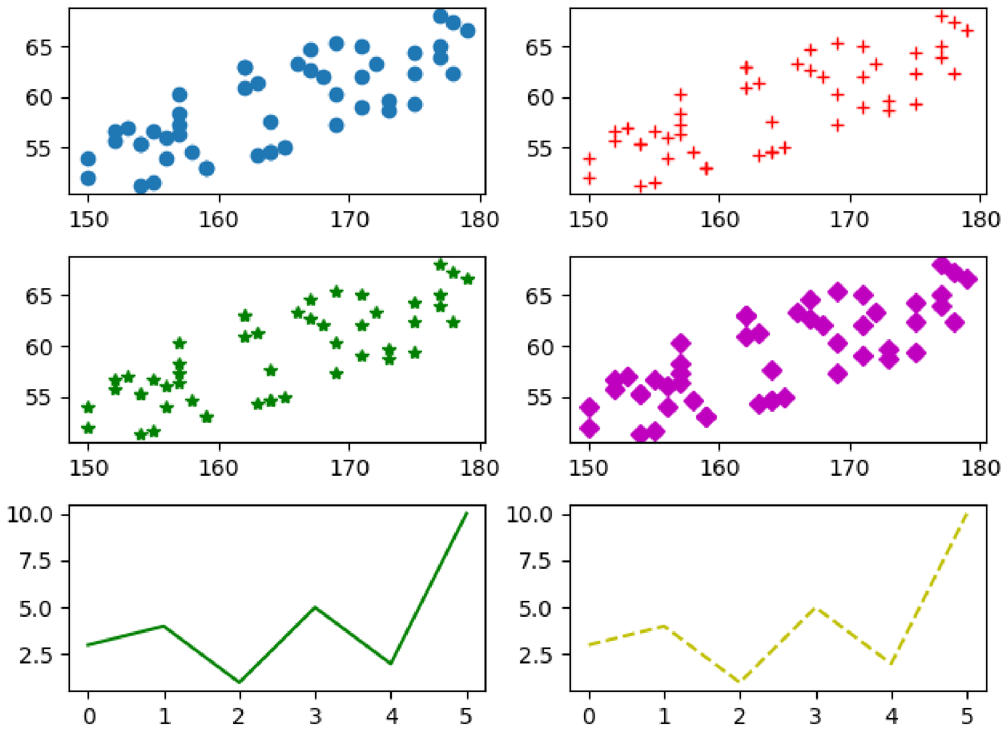


Consider the folliwing example such as we have 50 students providing their height and weight. Data are generated in the following chunck of code.

```
n = 50
height = np.random.randint(30,size = n)
#print(height)
height1 = np.add(150,height)
height = height1.reshape(n,1)
#print(height)
```

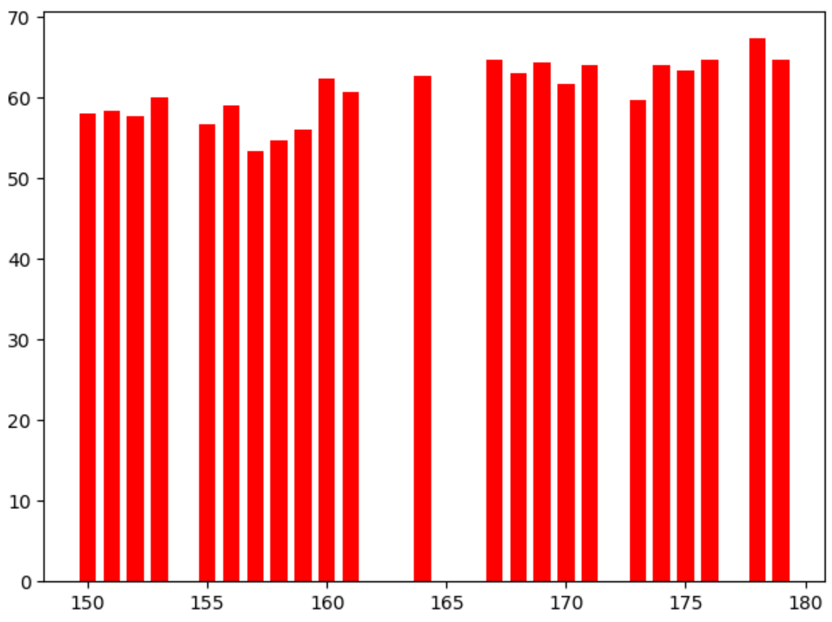
```
weight = np.random.randint(10,size = n)
#print(height)
weight = np.add(height1/3,weight)
weight = weight.reshape(n,1)
#print(weight)


#创建图形
plt.figure(1)

#第一行第一列图形
ax1 = plt.subplot(3,2,1)
#第一行第二列图形
ax2 = plt.subplot(3,2,2)
#第二行
ax3 = plt.subplot(3,2,3)
ax4 = plt.subplot(3,2,4)
ax5 = plt.subplot(3,2,5)
ax6 = plt.subplot(3,2,6)

#选择ax1
plt.sca(ax1)
plt.plot(height,weight,'o')

#选择ax2
plt.sca(ax2)
plt.plot(height,weight,'r+')
#plt.plot(height,weight,'+','k')


#选择ax3
plt.sca(ax3)
plt.plot(height,weight,'g*')

plt.sca(ax4)
plt.plot(height,weight,'mD')

plt.sca(ax5)
plt.plot(y,'g-')

plt.sca(ax6)
plt.plot(y,'y--')

plt.show()
plt.savefig("1-2.png")
```

Running results:

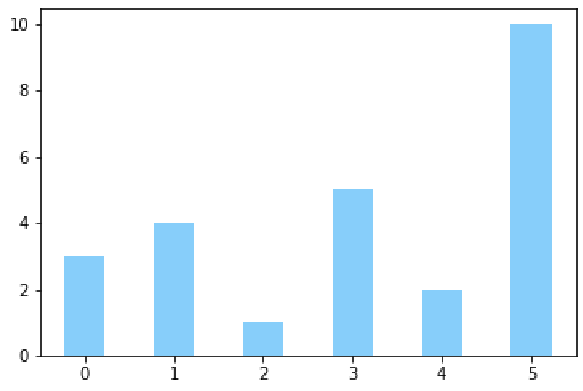


Note however that in one case the we plot the values of the matrix both in a two dimensional graphic, while in the other case we plot the two vectors as separated.
Here below we introduce the barplot:

```
#设置图形宽度
bar_width = 0.7

n = 50
height = np.random.randint(30,size = n)
#print(height)
height = np.add(150,height)

weight = np.random.randint(10,size = n)
#print(height)
weight = np.add(height/3,weight)

#绘制图形
plt.bar(height,weight,bar_width,align='center',color='r')

# #加图例
plt.show()
plt.savefig("1-3.png")
```



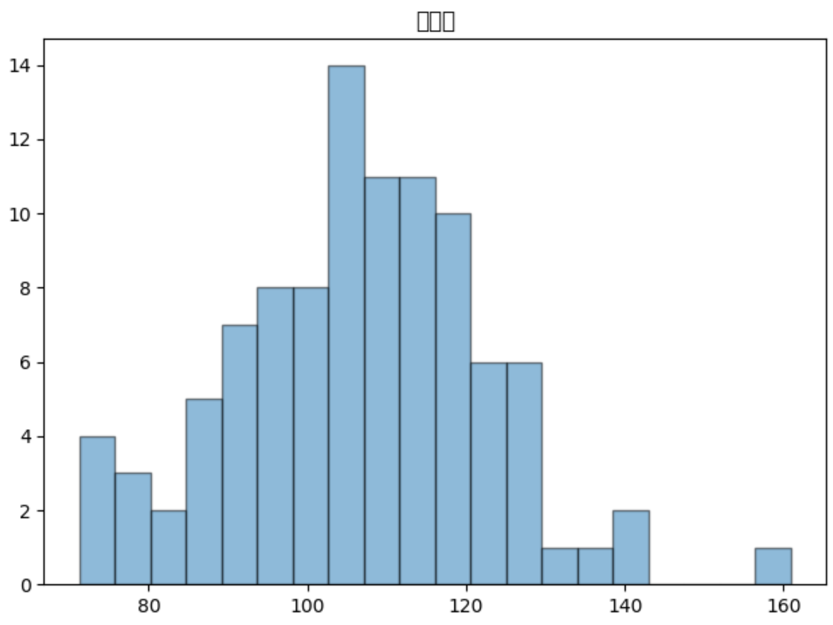
```
values = [3,4,1,5,2,10]
# 包含每个柱子下标的序列
index = np.arange(6)
# 柱子的宽度
width = 0.45

p2 = plt.bar(index, values, width, color="#87CEFA")
plt.savefig("1-4.png")
```

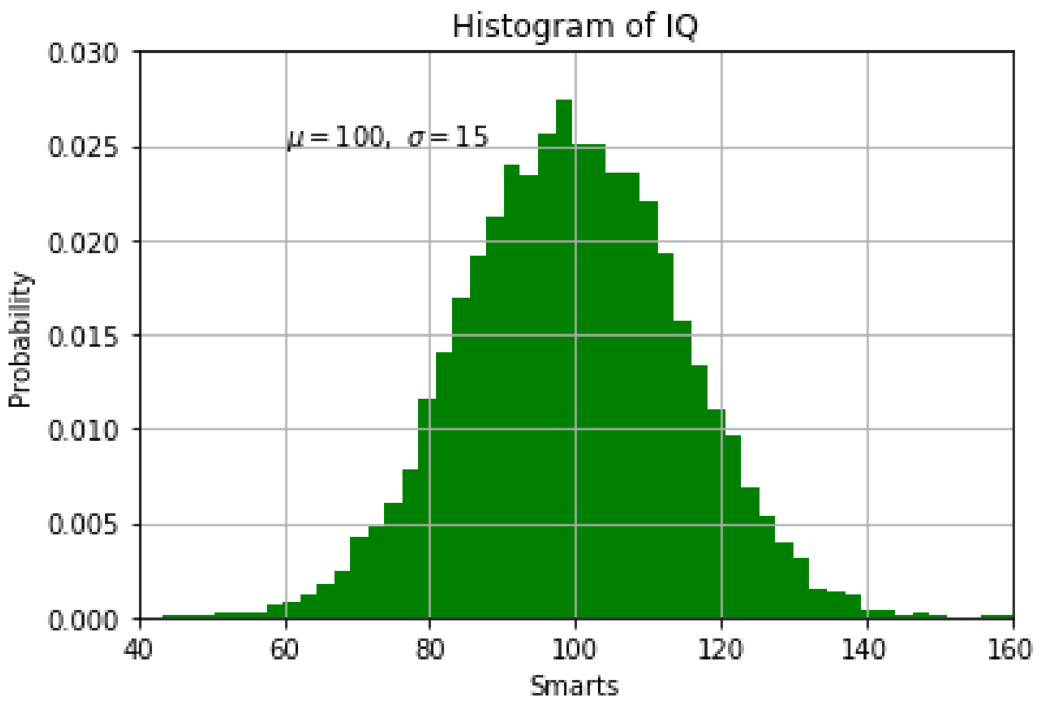


Here we introduce the histogram plot.

```
#均值与方差
mu,sigma = 100,20
a = np.random.normal(mu,sigma,size=100)

#绘制图形
plt.hist(a,20,normed=0,histtype='bar',edgecolor='k',alpha=0.5)

plt.title('直方图')
plt.show()

#直方图
import numpy as np
import matplotlib.pyplot as plt
```

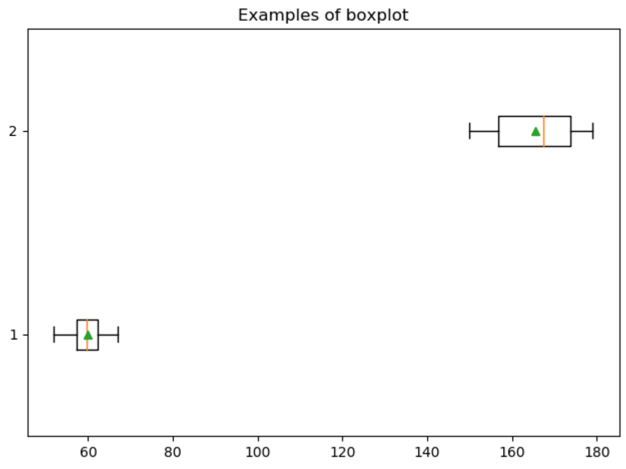
```
# Fixing random state for reproducibility
np.random.seed(19680801)

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50,density=True ,facecolor='g')#density=True 绘制
并返回概率密度
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.xlim(40, 160)
plt.ylim(0, 0.03)
plt.grid(True)
plt.show()
```

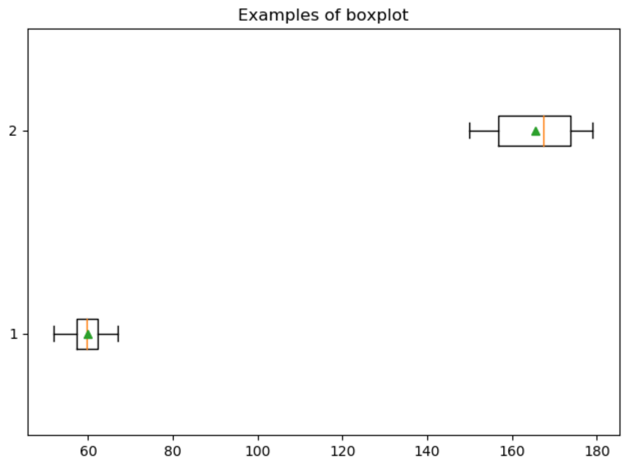




Another important reprentation is the boxplot that allows showing the interval in which values are concentrated. Below you see them in horizontal and vertical positions.

```
box_1, box_2= weight,height
plt.title('Examples of boxplot')#标题，并设定字号大小

#vert=False:水平箱线图；showmeans=True：显示均值
plt.boxplot([box_1, box_2], vert=False,showmeans=True )
plt.show()#显示图像


plt.boxplot([box_1, box_2], showmeans=True )
plt.show()#显示图像
```



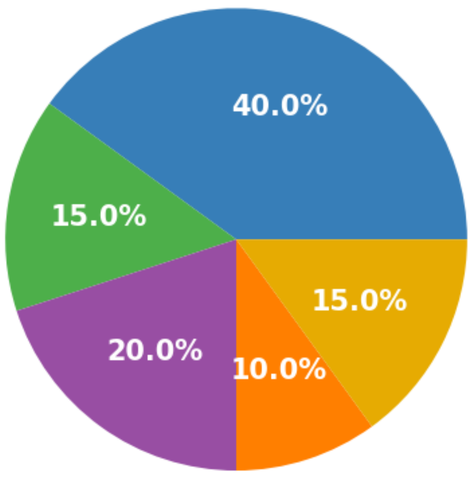


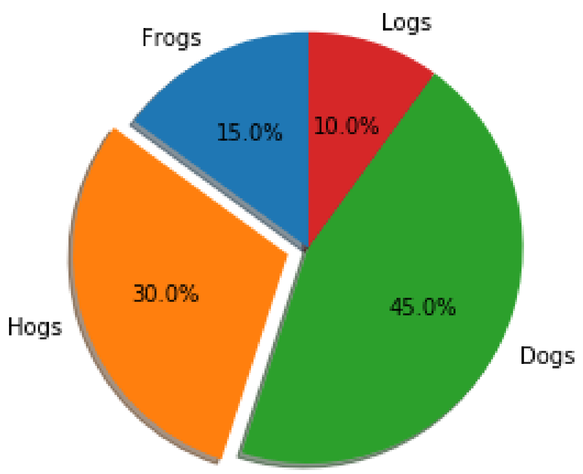
Clearly we can see that people that appreciated the most are the oldest one and they like to drink White wine. This is an important marketing information. Those who do not drink wine they do not like cheese (and they are aged between 50 and 60).

Below you can see how to make a pie.

```
elements = ["A", "B", "C", "D", "E"]
weight = [40, 15, 20, 10, 15]
colors = ["#377eb8", "#4daf4a", "#984ea3", "#ff7f00", "#e6ab02"]

wedges, texts, autotexts = plt.pie(weight,
                                   autopct="%3.1f%%",
                                            textprops=dict(color="w"),
                                            colors=colors)

plt.setp(autotexts, size=15, weight="bold")
plt.setp(texts, size=12)

plt.title("FIGURE")
plt.show()

#饼状图
import matplotlib.pyplot as plt

# Pie chart, where the slices will be ordered and plotted counter-clockwise:
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)  # 偏离相邻半径的大小

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()
```

FIGURE

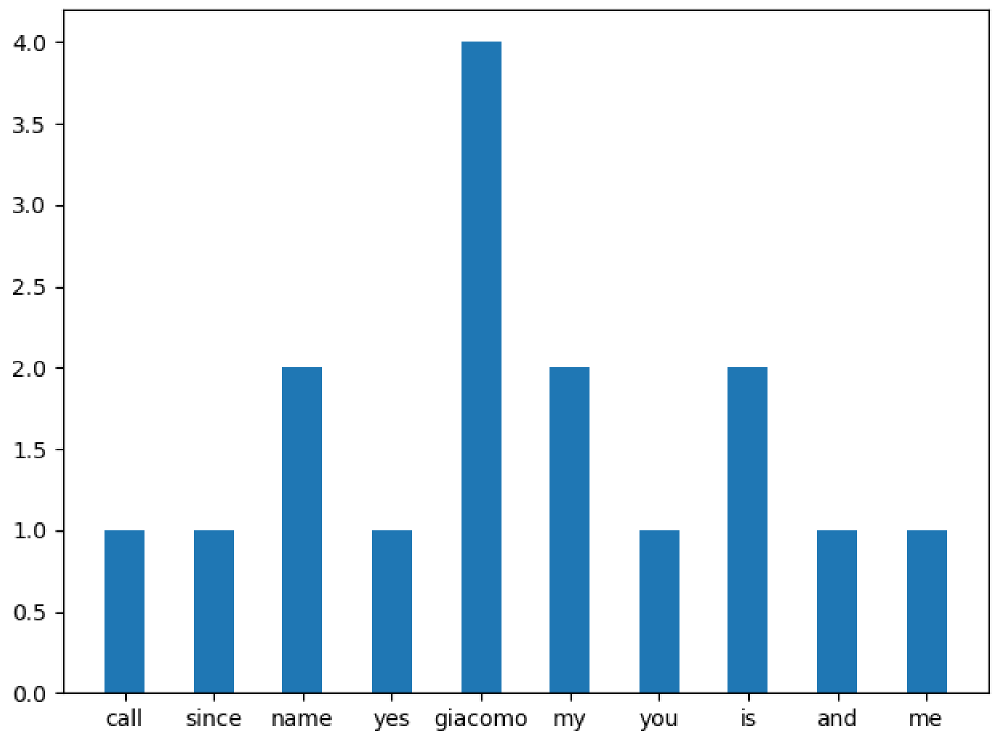




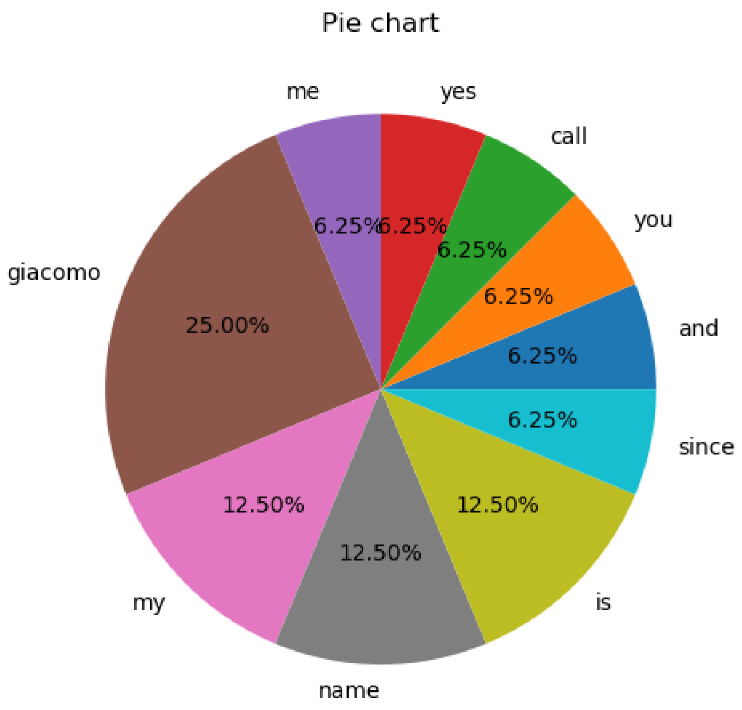
Text analysis? A simple example to visualize words!

Suppose you listen to me and you hear this:

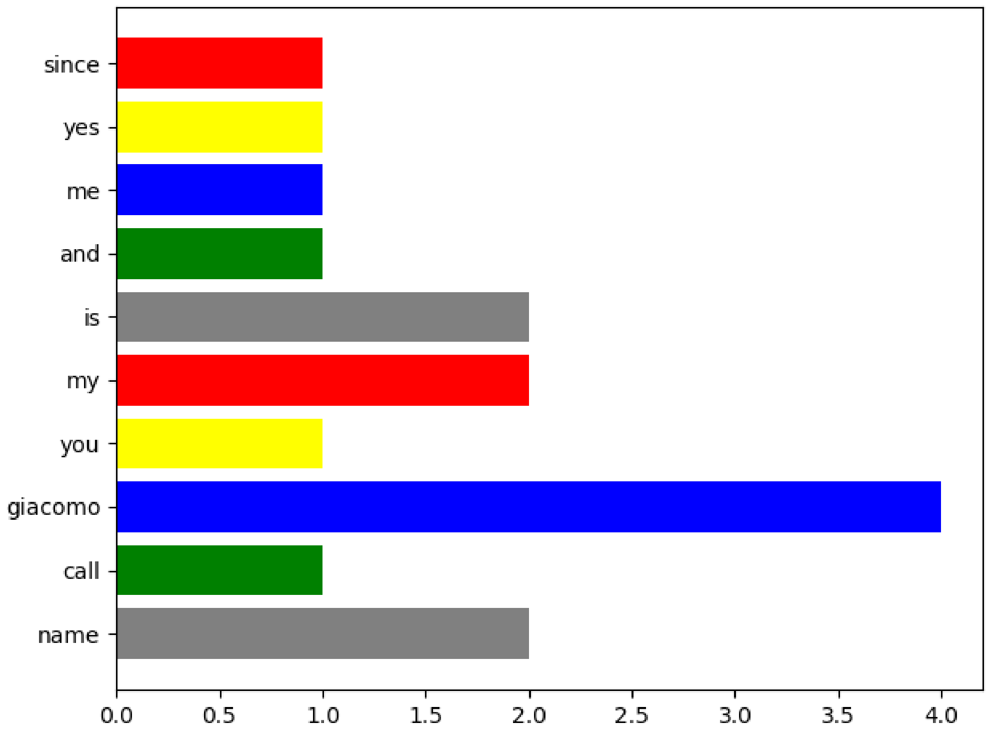
```
width = 0.45
myspeech = "giacomo is my name yes giacomo and you call me giacomo since giacomo
is my name"
newspeech = myspeech.split( )

count_set = set(newspeech)
count_list = list()
name_list = list()
for item in count_set:
    count_list.append((newspeech.count(item)))
    name_list.append(item)
print(count_list)
print(name_list)
plt.bar(name_list, count_list, width)
plt.show()
```



```
fig = plt.figure()
plt.pie(count_list,labels=name_list,autopct='%1.2f%%')
plt.title("Pie chart")

plt.show()
```



```
colors = ['red', 'yellow', 'blue', 'green', 'gray']
colors.reverse()

plt.barh(name_list, count_list, tick_label=name_list, color=colors)
plt.show()
```

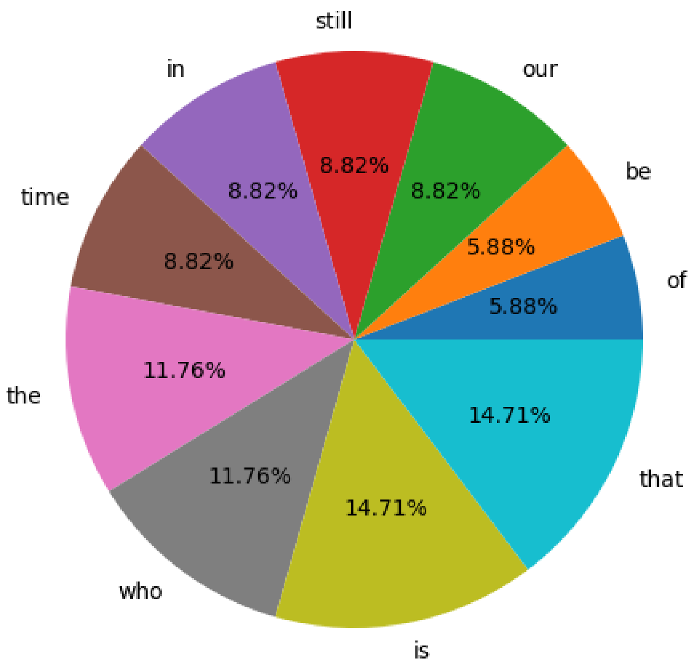


Below the victory speech made by Obama in 2008.

Now using a pie show the distribution of the 10 most used words.

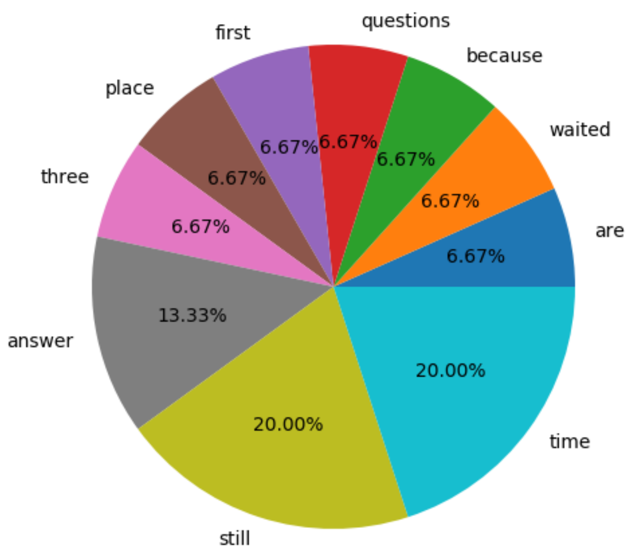
```
a1 = "If there is anyone out there who still doubts that America"
a2 = "is a place where all things are possible who still wonders"
a3 = "if the dream of our founders is alive in our time who still questions"
```

```
a4 = "the power of our democracy tonight is your answer It is the answer"
a5 = "told by lines that stretched around schools and churches in numbers"
a6 = "this nation has never seen by people who waited three hours and four"
a7 = "hours many for the first time in their lives because they believed that"
a8 = "this time must be different, that their voices could be that difference "

new_a1 = a1.split( )
new_a2 = a2.split( )
new_a3 = a3.split( )
new_a4 = a4.split( )
new_a5 = a5.split( )
new_a6 = a6.split( )
new_a7 = a7.split( )
new_a8 = a8.split( )

new = new_a1 + new_a2+ new_a3+ new_a4+ new_a5+ new_a6+ new_a7+ new_a8

count_set = set(new)
count_data = list()
count_list = list()
name_list = list()
for item in count_set:
    count_data.append([item,new.count(item)])

count_data.sort(key=lambda count_data : count_data[1])
count_data = count_data[-10:]
print(count_data)
name_list = [count_data[i][0] for i in range(len(count_data))]
count_list = [count_data[i][1] for i in range(len(count_data))]
# print(name_list)
# print(count_list)
plt.pie(count_list,labels=name_list,autopct='%1.2f%%')
plt.show()
```

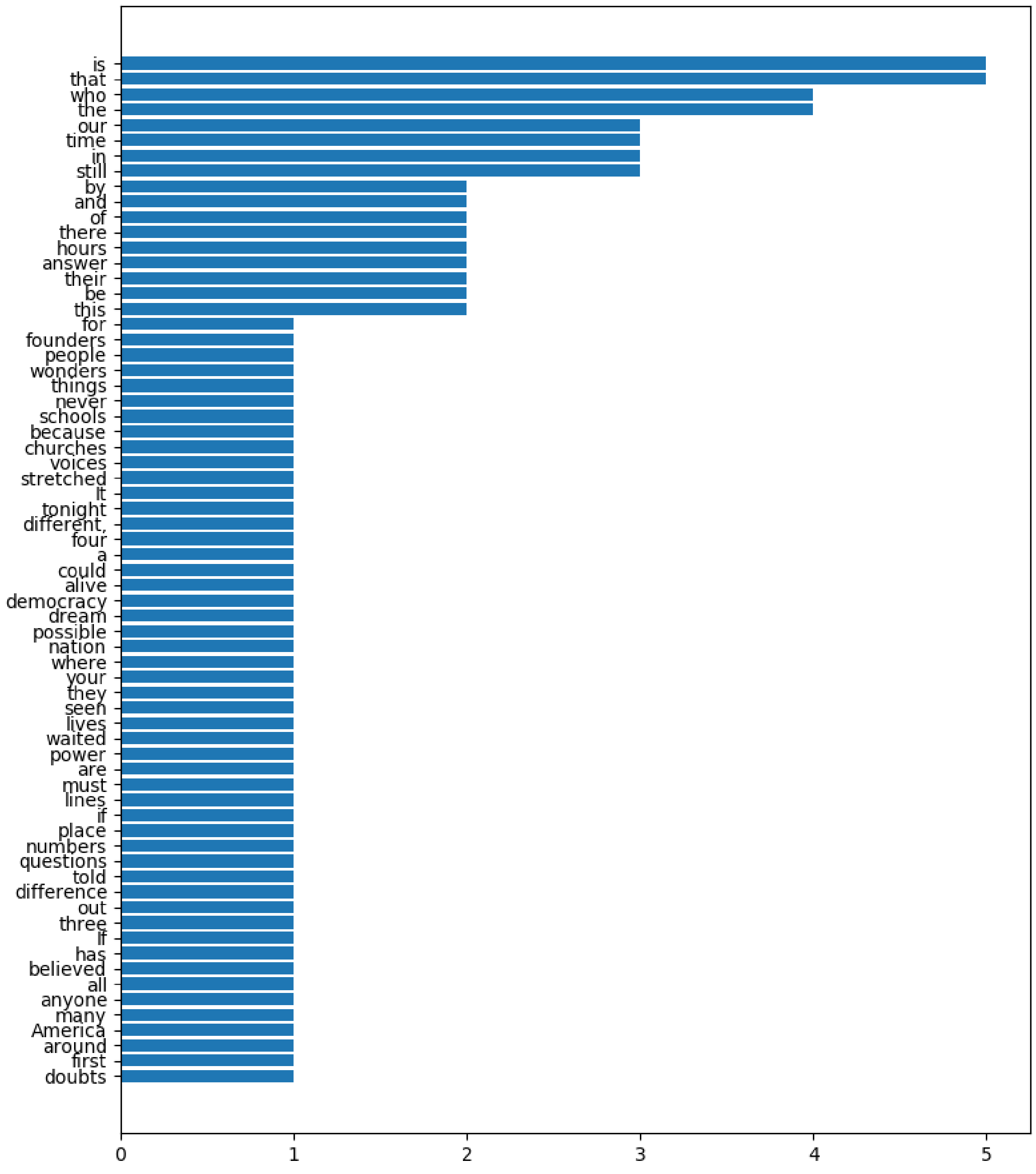


However, if you want to consider the main words only you need to take out the others such that "if", "is" etc...
This is the way to do that.

```
a1 = "If there is anyone out there who still doubts that America"
a2 = "is a place where all things are possible who still wonders"
a3 = "if the dream of our founders is alive in our time who still questions"
a4 = "the power of our democracy tonight is your answer It is the answer"
a5 = "told by lines that stretched around schools and churches in numbers"
a6 = "this nation has never seen by people who waited three hours and four"
a7 = "hours many for the first time in their lives because they believed that"
a8 = "this time must be different, that their voices could be that difference "

new_a1 = a1.split( )
new_a2 = a2.split( )
new_a3 = a3.split( )
new_a4 = a4.split( )
new_a5 = a5.split( )
new_a6 = a6.split( )
new_a7 = a7.split( )
new_a8 = a8.split( )

new = new_a1 + new_a2+ new_a3+ new_a4+ new_a5+ new_a6+ new_a7+ new_a8
cut_list =
["is","by","I","a","if","If","in","the","that","of","our","be","there","this","their","and","they","your","hours","told","who"]

for i in cut_list:
    while i in new:
        new.remove(i)

count_set = set(new)
count_data = list()
count_list = list()
name_list = list()
for item in count_set:
    count_data.append([item,new.count(item)])

count_data.sort(key=lambda count_data : count_data[1])
count_data = count_data[-10:]
print(count_data)
name_list = [count_data[i][0] for i in range(len(count_data))]
count_list = [count_data[i][1] for i in range(len(count_data))]
# print(name_list)
# print(count_list)
plt.pie(count_list,labels=name_list,autopct='%1.2f%%')
plt.show()
```

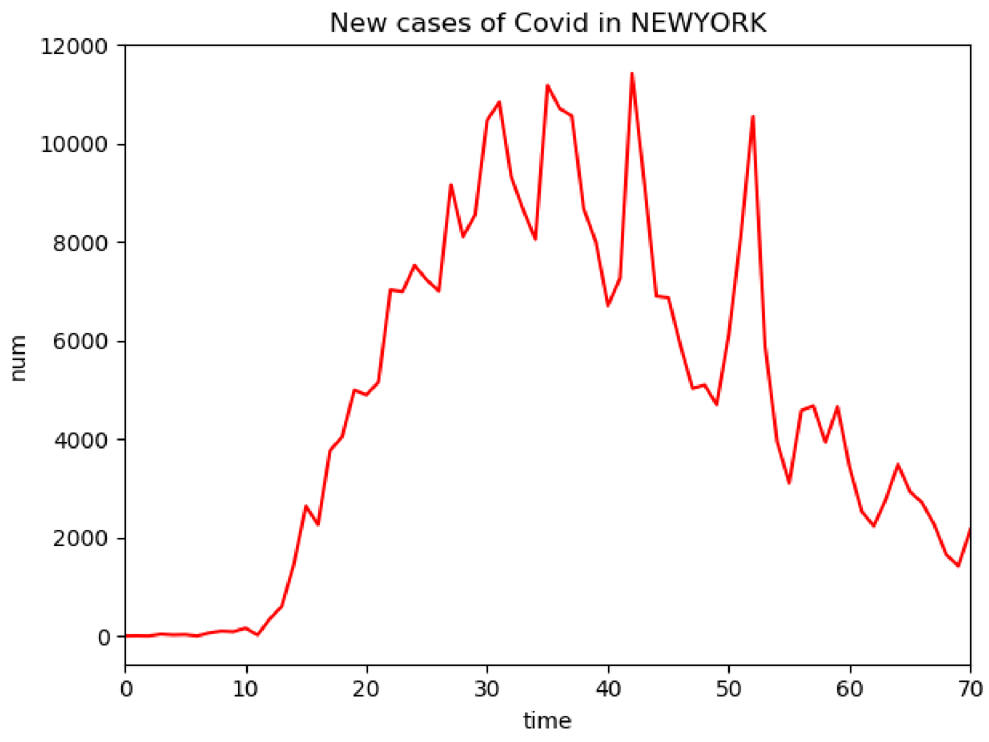


The barplot below may be nicer to show:



Below the graphic of the Covid new cases observed in Italy as shown above but in a smarter format.

```
import pandas as pd
from scipy import stats
data = pd.read_excel('CoronavirusSpreadUSAregions.xlsx')
y = data["NEWYORK"]
plt.xlabel('time')
plt.ylabel('num')
plt.title('New cases of Covid in NEWYORK')
y.plot(color='r',title='New cases of Covid in NEWYORK')
plt.show()
```



Exercise Human Resources data

The following script generate data relative to the information of 200 workers working in a firm. Please run the following script:

```
import numpy as np
edumat,jobmat, wag, age  = [],[],[],[]
n = 200
sexmat0 = np.random.rand(200)
sexmat = [round(i) for i in sexmat0]
sexmat = np.array(sexmat)
sexmat = sexmat.reshape(200,1)
print(sexmat)
for i in range(len(sexmat)):
    if(sexmat[i] == 0):
        edumat[i] = np.random.randint(2, 4)
        jobmat[i] = np.random.randint(1, 4)
        wag[i] = 2.3+0.5*np.random.rand(1)
        age[i] = round(40+10*np.random.rand(1))
    else:
        edumat[i] = np.random.randint(1, 4)
        jobmat[i] = np.random.randint(1, 2)
        wag[i] = 2+0.3*np.random.rand(1)
        age[i] = round(40+3.5*np.random.rand(1))
```

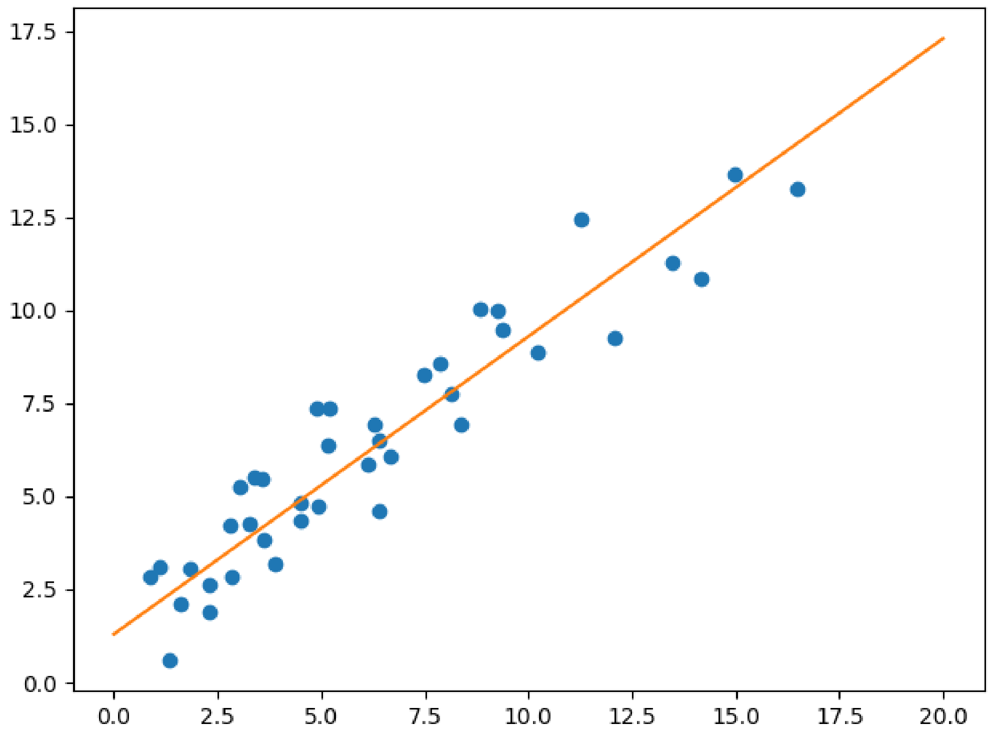
Simple linear regression model

The simple linear regression model can be written as follows:

$y_i = a + b \times x_i + e_i$

Where y_i is the dependent variable with i=1,2,.,n. x_i is defined as the independent variable while a and b are the regression coefficients. In particular, b measures the impact of x on the y while a is the intercept (constant term) that adds up. Finally, e_i represents the error term containing everything the model does not know about the dependent variable. Note that e_i is also defined as the residuals since we can write: $e_i = y_i - a - b \times x_i$
The parameters of this model are estimated using the so-called Least Squares method (or Ordinary least squares). Lets first generate the model and try to figure out its dynamics using the following code:

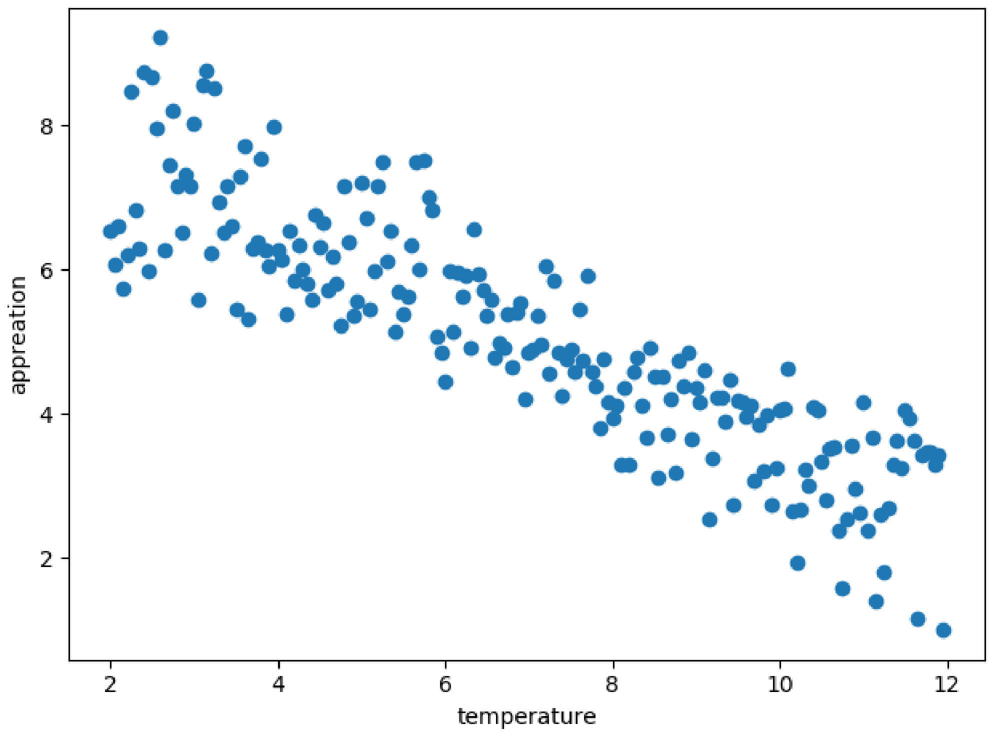
```
from scipy import stats
import matplotlib.pyplot as plt
import numpy as np
pointsline = list()
x = stats.chi2.rvs(6,size=40)
e = stats.norm.rvs(size=40)

y = 1.3+0.8*x+e
pointsline = [1.3+0.8*i for i in range(21)]
# print(list(y))
y = list(y)
print(y)
plt.plot(x,y,'o')
plt.plot(pointsline)
plt.show()
```

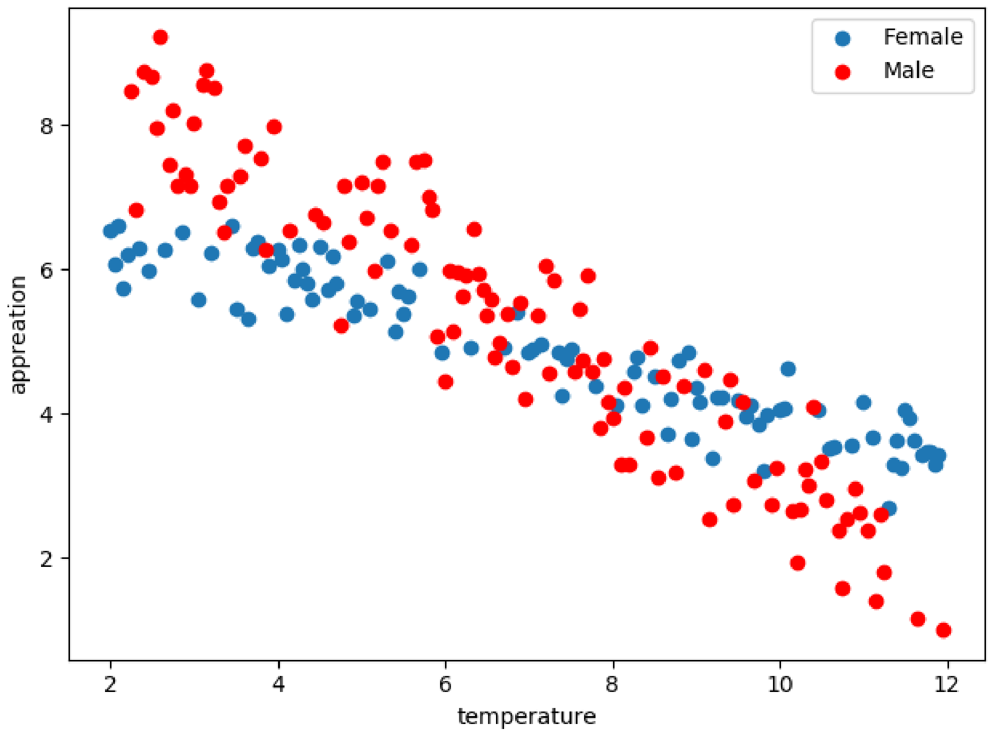


For example, we know that the level of appreciation of a beer depends (among other factors) upon the temperature of the beer. Suppose we ask 200 clients (both female and male) to express their appreciation for the beer for each temperature included beteen 2 degres and 12 degrees Celsius. The chunk below is a simulation:

```
n = 200
sex = list()
appreciation = list()
data0 = list()
data1 = list()
data0_x = list()
data1_x = list()
temperature = [0,]
gender = np.random.rand(n)
gender = np.round(1.1*gender)

for i in range(len(gender)):
    if gender[i] == 0:
        sex.append("Femall")
    else:
        sex.append(("Male"))

temperature = [2+(x*5)/100 for x in range(0,200)]
ef = 0.4* stats.norm.rvs(size=n)
em = 0.8* stats.norm.rvs(size=n)

ef = list(ef)
em = list(em)

i = 0
for i in range(n):
    if(gender[i] == 0):
        appreciation.append(7 - 0.3 * temperature[i] + ef[i])
    else:
        appreciation.append(10 - 0.7 * temperature[i] + em[i])
```

```python
plt.plot(temperature,appreciation,'o')
plt.xlabel("temperature")
plt.ylabel("appreation")
plt.show()

for i in range(len(sex)):
    if(gender[i] == 0):
        data0.append(appreciation[i])
        data0_x.append(temperature[i])
    else:
        data1.append(appreciation[i])
        data1_x.append(temperature[i])

ax = plt.subplot()

ax.plot(data0_x,data0, 'o',label='Female')
ax.plot(data1_x,data1, 'ro',label='Male')

plt.xlabel("temperature")
plt.ylabel("appreation")
plt.legend()
plt.show()
```





Her we note that there is a negative relation between the temperature and the appreciation of clients in the supermarket. However, it seems that female clients appreciate the beer differently compared to male, when the temperature change. Can you sketch these differences?

We can evaluated these differences by running a linear regression considering the variable appreciation as y and the variable temperature as x (note that here temperature is a control variable that affects the appreciation of the beer). This can be done as follows:

```
import pandas as pd
OurRegression = np.polyfit(appreciation,temperature,1)
print(OurRegression)
```

One can see, indeed, that the level of appreciation decreses toghether with the raising temperature. In addition, each additional degree impact on reducing about 0.5 of the appreciation of the client.

Note that the R^2 is quite high so the temperature seems to be an important factor explaining the dynamics of y.

In addition, both constant and the slope coefficients are statistically significant since we have 3 stars $***$ on the right hand side of the regression coefficients. One has that No star means the parameter is not significantly different from zero (i.e. no impact); 1 star the parameter is moderately significant (at 10% level); 2 stars the parameter is quite significant (at 5% level); 3 stars the parameter is strongly significant (at 1% level).

Now suppose we want to evaluate the beer appreciation by gender. This is because we wish to know if there is the temperature impacts differently in the appreciation of female and that of male. This is the way to run it:

```
summary_across_rows = pd.DataFrame(OurRegression).describe() # across axis=0
print(summary_across_rows)
```

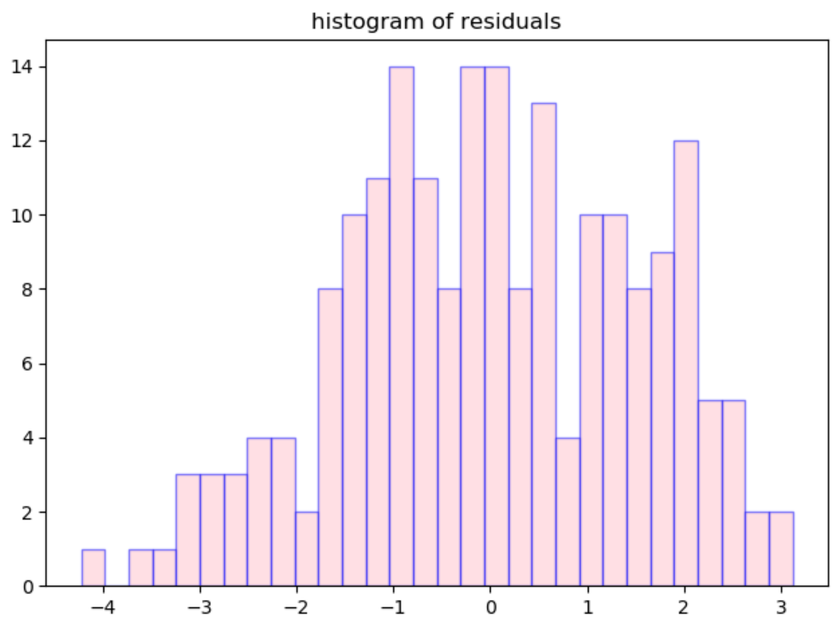
Running Results:

```
[-1.52629741 14.74281946]
                0
count   2.000000
mean    6.608261
std    11.504003
min    -1.526297
25%     2.540982
50%     6.608261
75%    10.675540
max    14.742819
```

We can see that temperature impacts more on male appreciation compared and less of female. The appreciation of male (say between 10 and 1) is more dispersed than female that tend to be less volatile since the range is between 7 and 3. Thefore, male are more concerned than female regarding the temperature of their beer!

Note that, among the validation criteria of the regression analysis, beside the R^2 and the significance of the parameters (t-test) there is also the normality of Residuals. Firstly, note that if you want to extract the errors or the residuals of your regression you have to type:

```
OurRegression = np.polyfit(appreciation,temperature,1)
p1 = np.poly1d(OurRegression)
print(OurRegression)
print(p1)

resid = list()
for i in range(len(appreciation)):
    resid.append(temperature[i] - p1(appreciation[i]))
print(resid)

plt.hist(resid, 30, edgecolor='blue',facecolor = 'pink', alpha=0.5)
plt.title('histogram of residuals')
plt.show()
```



The null hypothesis is Normality so:

> If the p-value is smaller than 5% you reject the Normality hypothesis.

> If the p-value is bigger than 5% you accept the Null-hypothesis. The residuals are normal.

Linear regression analysis can use more than one variables (more than one X). Suppose in the previous example we also record the age of the client, using this simulation:

```
import pandas as pd
n = 200
sex = list()
appreciation0 = list()
appreciation1 = list()
temperature0 = list()
age0 = list()
total = list()


gender = np.random.rand(n)
gender = np.round(1.1*gender)

for i in range(len(gender)):
    if gender[i] == 0:
        sex.append("Femall")
    else:
        sex.append(("Male"))

temperature = [2+(x*5)/100 for x in range(0,200)]
age = 22 + stats.chi2.rvs(5,size=n)
ef = 0.4* stats.norm.rvs(size=n)
em = 0.8* stats.norm.rvs(size=n)

ef = list(ef)
em = list(em)

for i in range(n):
    if(gender[i] == 0):
        appreciation0.append(7 - 0.3 * temperature[i] + 0.1*age[i]+ef[i])
        temperature0.append(temperature[i])
        age0.append(age[i])
        total.append(temperature[i]+age[i])
    else:
        appreciation1.append(10 - 0.7 * temperature[i] + 0.02*age[i]+em[i])


OurRegression = np.polyfit(appreciation0,total,1)
summary_across_rows = pd.DataFrame(OurRegression).describe() # across axis=0
print(summary_across_rows)
```

Running Result:

```
[-1.95274887 48.29920857]
               0
count   2.000000
mean   23.173230
std    35.533500
min    -1.952749
25%    10.610240
50%    23.173230
75%    35.736219
max    48.299209
```

Exercise Restaurant

The following code generate a dataset called $Resto$

```
n = 1000
Creditcard = list()
sex = [0*i for i in range(n)]
timing = [0*i for i in range(n)]
age = [0*i for i in range(n)]
day = [0*i for i in range(n)]
Creditcard0 = np.random.rand(n)
Creditcard0 = np.round(Creditcard0)

for i in range(len(Creditcard0)):
    if Creditcard0[i] == 0:
        Creditcard.append("Femall")
    else:
        Creditcard.append(("Male"))

spending = 19+stats.chi2.rvs(8,size=n)

for i in range(n):
    if(spending[i]<28):
        sex[i] = int(((np.round(np.random.rand(1))))[0])
        timing[i] = (40+stats.chi2.rvs(8,size=1))[0]
        age[i] = (np.round(25+stats.chi2.rvs(3)))
        day[i] = (1+3* np.round(np.random.rand(1)))[0]
    else:
        sex[i] = 1
        timing[i] = 90 + stats.chi2.rvs(3)
        age[i] = np.round(45+stats.chi2.rvs(6))
        day[i] = np.round(2+2*np.random.rand(1))[0]

for i in range(len(day)):
    if day[i] == 1:
        day[i] = "Mon-Thu"
    elif day[i] == 2:
        day[i] = "Fri"
    elif day[i] == 3:
        day[i] = "Sat"
    elif day[i] == 4:
        day[i] = "Sun"
```

```
Resto =
pd.DataFrame({'spending':spending,'age':age,'day':day,'sex':sex,'Creditcard':Cre
ditcard,'timing':timing})
print(Resto)
```

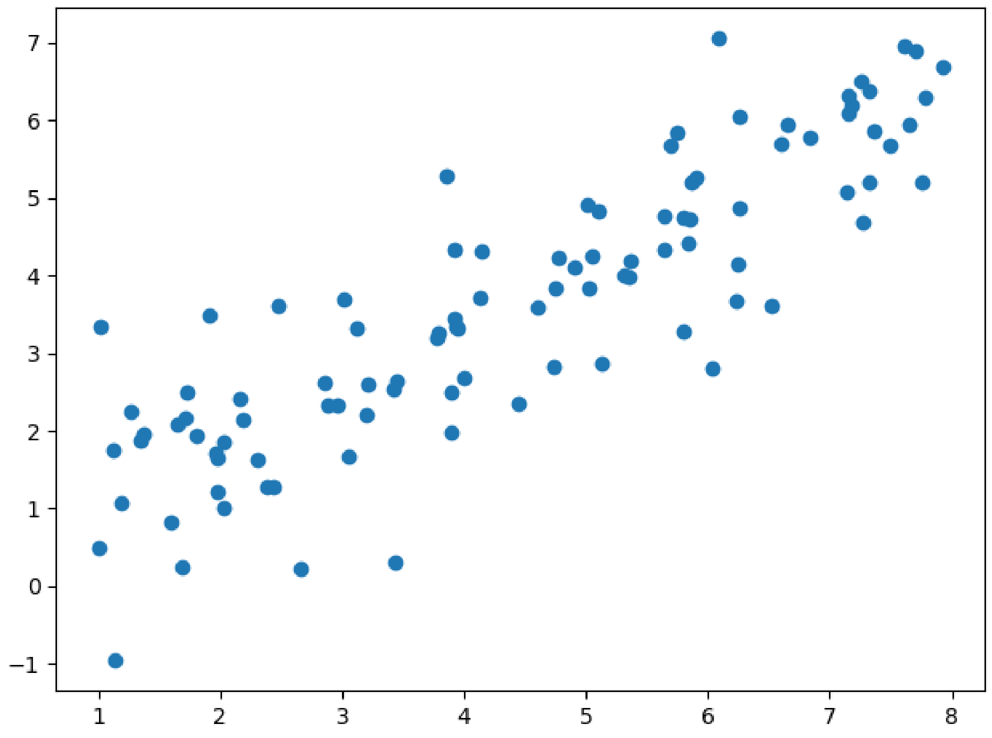
Running Result:

```
      spending   age      day  sex Creditcard     timing
0    26.314072  27.0  Mon-Thu    1     Femall  48.596835
1    29.676379  49.0      Sat    1     Femall  93.479555
2    26.630359  32.0      Sun    0     Femall  45.392964
3    30.925381  53.0      Sat    1     Femall  94.920864
4    23.183199  26.0  Mon-Thu    0     Femall  46.072180
..         ...   ...      ...  ...        ...        ...
995  22.005320  28.0  Mon-Thu    0       Male  45.927595
996  29.592402  51.0      Fri    1     Femall  92.187404
997  24.327652  27.0  Mon-Thu    1       Male  54.555624
998  22.503088  27.0      Sun    0     Femall  47.029779
999  24.713992  27.0      Sun    1     Femall  64.911918
```

Optimization with Python

Suppose we observe the following variables, i.e. x and y:

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

np.random.seed(1)
n = 100
x = 1+7*np.random.rand(n)
y = 1*np.random.randn(n) + 0.8 * x

plt.plot(x,y,'o')
plt.show()
```



Now suppose that we believe that there is a relation between x and y. For example we believe that the following relation holds $y = \beta x + \epsilon$. In other words, we believe that y is function of x and when x changes y changes according to the law expressed in the previous expression.

The following chunk makes the trick:

```
from scipy.optimize import minimize

def myfu(args):
    x , y = args
    # v = (y - b*x)*(y - b*x)
    v = lambda b: np.sum((y - b*x)*(y - b*x))
    return v

args = (x,y)
x0 = np.asarray((1))  # 初始猜测值
res = minimize(myfu(args), x0, method='SLSQP')
print(res.fun)
print(res.success)
print(res.x)
```

Running Result:

```
81.40590541320968
True
[0.80753537]
```

Note that we have a function (myfu) whose last element is $sum(e^2)$. Moreover we want to minimize the function by finding a specific value (this is a 1 dimensional optimization). There are several options to optimize a function in R. the $optim$ represents a widely used tool.

As you can see the results are the same as before! You might want to change what you want to minimize. For example, suppose you want to minimize the sum of the absolute errors. Then the following code makes the trick.

```
from scipy.optimize import minimize

def myfu(args):
    x , y = args
    # v = (y - b*x)*(y - b*x)
    # v = lambda b: np.sum((y - b*x)*(y - b*x))
    v = lambda b: np.sum(abs(y - b * x))
    return v

args = (x,y)
x0 = np.asarray((1))  # 初始猜测值
res = minimize(myfu(args), x0, method='SLSQP')
print(res.fun)
print(res.success)
print(res.x)
```

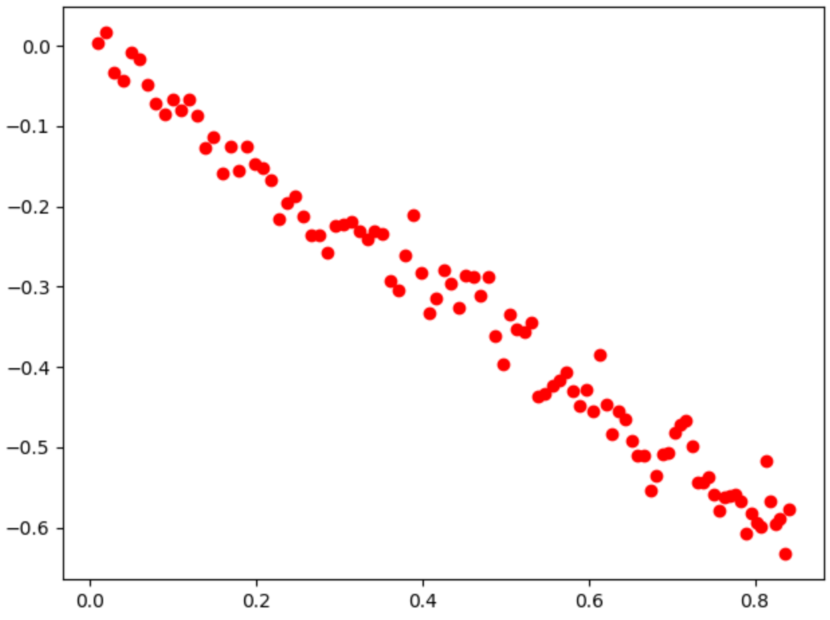
Running Result:

```
68.34517048885769
True
[0.82053164]
```

Exercise (unidimensional)

Consider the following code:

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

np.random.seed(1224)
x = [i/100 for i in range(1,101)]
x = np.sin(np.array(x))
b = np.random.rand(1)
y = -b *x + 0.03*(np.random.randn(len(x)))

plt.plot(x,y,'ro')
plt.show()
```



> Suppose we believe that there exist the following relation $y = \beta x + \epsilon$.
>
> Can you find the value of beta that minimize the sum of squared epsilon?
>
> Can you find the value of beta that minimize the sum of absolute epsilon?

```
def myfu(args):
    x , y = args
    v = lambda b: np.sum((y - b*x)*(y - b*x))
    return v

def con(args):
    bmin, bmax = args
    cons = ({'type': 'ineq', 'fun': lambda b: b - bmin},{'type': 'ineq', 'fun':
lambda b: -b + bmax})
    return cons

args = (x,y)
x0 = np.asarray((1))  # 初始猜测值
args1 = (-4,4)
cons = con(args1)
res = minimize(myfu(args), x0, method='SLSQP',constraints=cons)
print(res.fun)
print(res.success)
print(res.x)
```

Running Result:

```
0.07616873457636662
True
[-0.72539767]
```

Suppose that we want to maximize a function. Consider the function $y = \frac{(\exp -0.5(x-4)^2)}{\sqrt{2\pi}}$. For example:

```
np.random.seed(1224)
```

```
x = [i/100 for i in range(1,101)]
x = np.sin(np.array(x))
b = np.random.rand(1)
y = -b *x + 0.03*(np.random.randn(len(x)))

plt.plot(x,y,'ro')
# plt.show()

def myfu(args):
    x , y = args
    v = lambda b: -(np.sum((y - b*x)*(y - b*x)))
    return v

def con(args):
    bmin, bmax = args
    cons = ({'type': 'ineq', 'fun': lambda b: b - bmin},{'type': 'ineq', 'fun':
lambda b: -b + bmax})
    return cons

args = (x,y)
x0 = np.asarray((1))  # 初始猜测值
args1 = (-14,14)
cons = con(args1)
res = minimize(myfu(args), x0, method='SLSQP',constraints=cons)
print(res.fun)
print(res.success)
print(res.x)
```

Running Result:

```
-5989.634882530805
True
[14.]
```

Now suppose we want to minimize a function of two variables. For example, consider the following function $z = (-2 + x)^2 + (4 + y)^2$ and assume we want to find the values of x and y that minimize z. Then we can still use optim as follows:

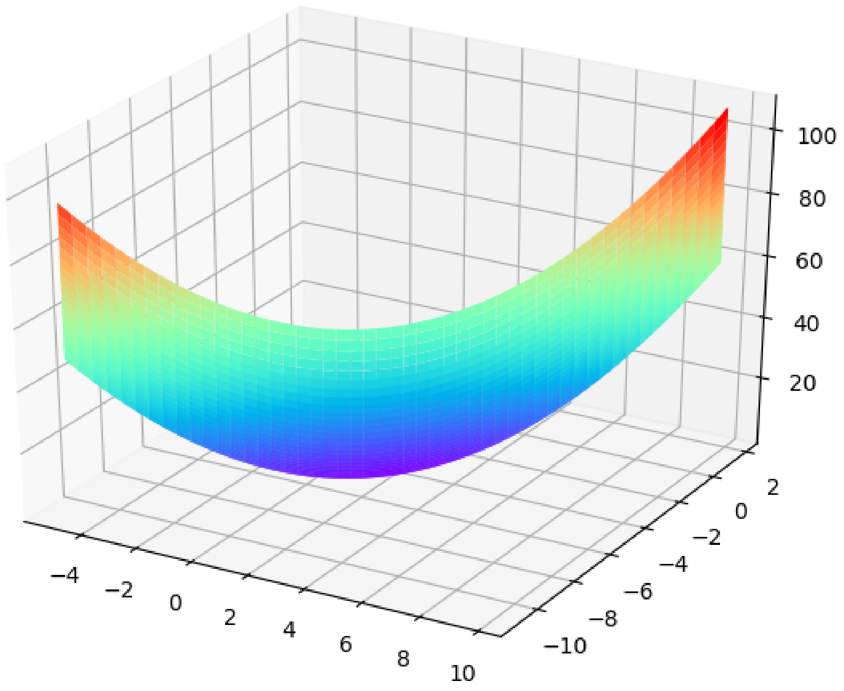
```
def fun(args):
    a, b = args
    v = lambda x: (a + x[0])**2 + (b + x[1])**2
    return v

# 定义常量值
args = (-2,4)  # a,b,c,d
# 设置初始猜测值
x0 = np.asarray((1,3))

res = minimize(fun(args), x0, method='SLSQP')
print(res.fun)
print(res.success)
print(res.x)
```

Running Reslut:

```
0.0
True
[ 2. -4.]
```

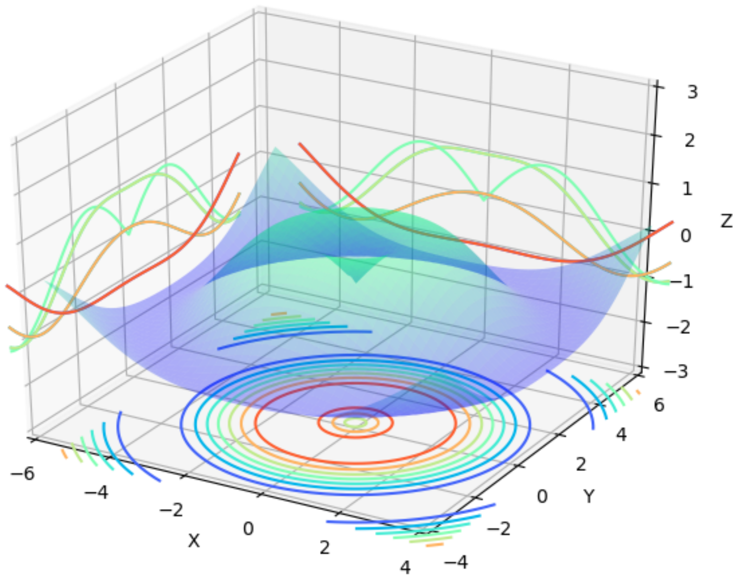
Suppose we want to have some fun by visualizing this z function as above. This is a nice code that allows you to see the beauty & power of python-graphics:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()  #定义新的三维坐标轴
ax3 = plt.axes(projection='3d')

#定义三维数据
xx = np.arange(-5,10,15/50)
yy = np.arange(-11,2,13/50)
X, Y = np.meshgrid(xx, yy)
# print(X)
Z = (-2+X)**2 + (4+Y)**2

#作图
ax3.plot_surface(xx,yy,Z,cmap='rainbow')
plt.show()
```



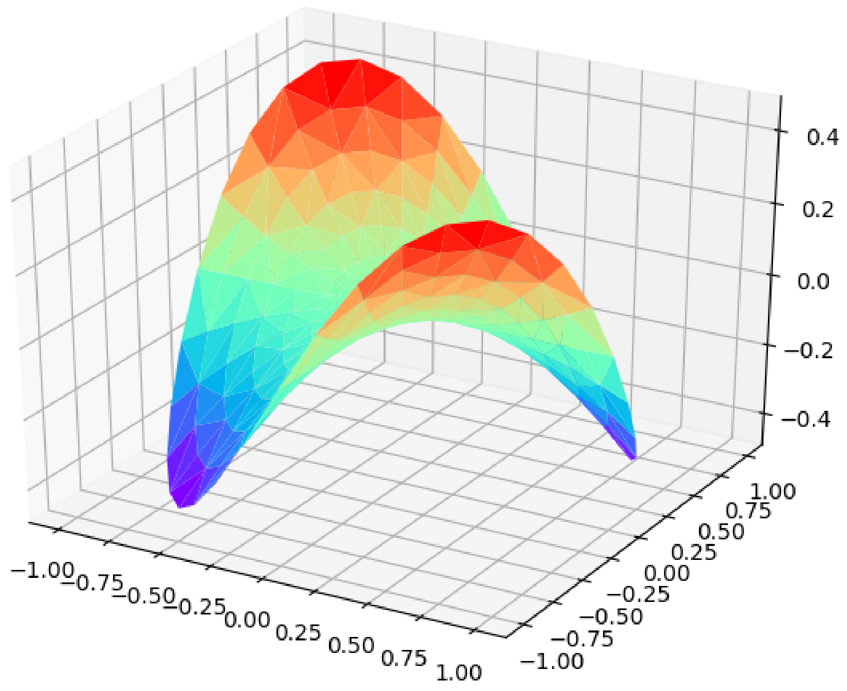
```
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

#定义坐标轴
fig4 = plt.figure()
ax4 = plt.axes(projection='3d')

#生成三维数据
```

```
xx = np.arange(-5,5,0.1)
yy = np.arange(-5,5,0.1)
X, Y = np.meshgrid(xx, yy)
Z = np.sin(np.sqrt(X**2+Y**2))

#作图
ax4.plot_surface(X,Y,Z,alpha=0.3,cmap='winter')     #生成表面， alpha 用于控制透明度
ax4.contour(X,Y,Z,zdir='z', offset=-3,cmap="rainbow")  #生成z方向投影，投到x-y平面
ax4.contour(X,Y,Z,zdir='x', offset=-6,cmap="rainbow")  #生成x方向投影，投到y-z平面
ax4.contour(X,Y,Z,zdir='y', offset=6,cmap="rainbow")   #生成y方向投影，投到x-z平面
#ax4.contourf(X,Y,Z,zdir='y', offset=6,cmap="rainbow")   #生成y方向投影填充，投到x-z
平面，contourf()函数

#设定显示范围
ax4.set_xlabel('X')
ax4.set_xlim(-6, 4)  #拉开坐标轴范围显示投影
ax4.set_ylabel('Y')
ax4.set_ylim(-4, 6)
ax4.set_zlabel('Z')
ax4.set_zlim(-3, 3)

plt.show()
```



```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

n_radii = 8
n_angles = 36


radii = np.linspace(0.125, 1.0, n_radii)
angles = np.linspace(0, 2 * np.pi, n_angles, endpoint=False)


angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
```

```
x = np.append(0, (radii * np.cos(angles)).flatten())
y = np.append(0, (radii * np.sin(angles)).flatten())
z = np.sin(-x * y)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_trisurf(x, y, z, linewidth=0.2, antialiased=True,cmap="rainbow")

plt.show()
```



The Normal distribution

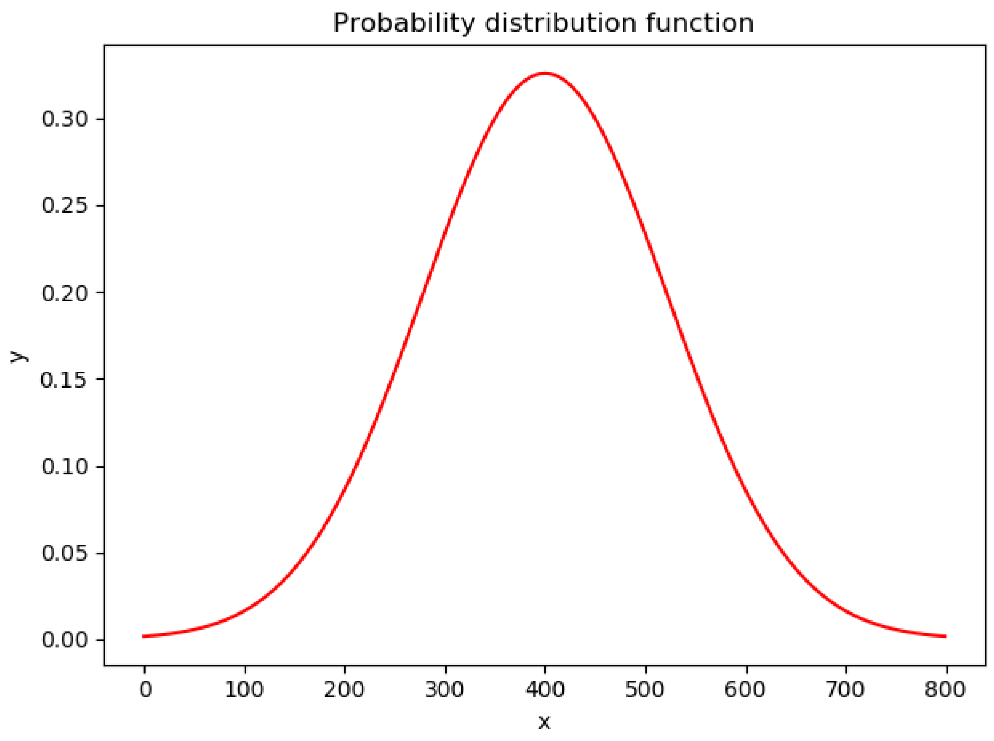
The normal (or Gaussian) probability function represents a crucial distribution in statistics. Most of the statistical inference is based on that distribution. One remarkable example is the so-called Central Limit Theorem, one of the most important result achieved in the last centuries! As a matter of fact, the properties of the (multivariate) Normal distribution represent also the cornerstone of the so-called Kalman filter, a crucial tool widely used in engeneering, finance and economics. Therefore, before introducing the Kalman filter and its use for filtering (forecasting) we will first focus on the univariate and bivariate Normal distribuion properties. For the sake of simplicity and since we will focus on the filtering of a single variable, We will consider only the bivariate Normal properties. However, similar results also hold for the generic multivariate Normal distribution. This is left for the future. The aim of the following sections is to provide a theoretical, yet practical, approach for understending **the basics of the Kalman filter and its potentialities in data analysis** . The reader will face simultaneously both theory and examples codes using R. This might be an interesting approach differing from most of the books in time series analysis. The emphasis here is not to follow the standard practice of teaching time series. On the contrary, the aim is to provide the basic knowledge to understand and simplify the state-space modelling approach by also intrudicing recently published results (by the author).

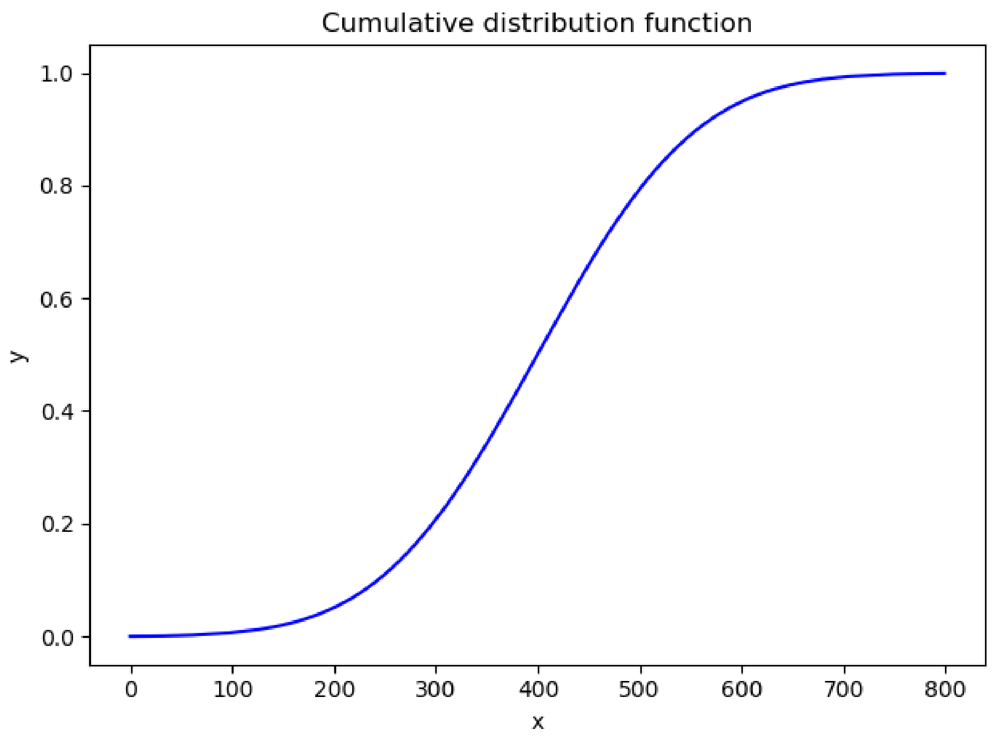
Consider a Normal distributed variable X having mean μ and standard deviation σ. We can then write the so called probability density function (pdf) for the univariate case as follows:

$$Probability(x) = \frac{1}{\sqrt{(2\pi\sigma^2)}} exp(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}]$$

Suppose we want to generete a normal distribution with mean 0.2 and variance 1.5 with R. This simple code does the job:

```
import numpy as np
import math
import matplotlib.pylab as plt

x = np.arange(-3.8,4.2,1/100)
mu = 0.2
sigma = np.sqrt(1.5)

pdf, cdf = list(), list()
for i in range(len(x)):
    pdf.append(1/np.sqrt(2*math.pi*sigma**2)*math.exp(-0.5*(x[i]-
mu)**2/sigma**2))
    cdf.append(round(np.sum(pdf)/100,4))
plt.plot(pdf,'r-')
plt.xlabel("x")
plt.ylabel("y")
plt.title("Probability distribution function")
plt.show()

plt.plot(cdf,'b-')
plt.xlabel("x")
plt.ylabel("y")
plt.title("Cumulative distribution function")
plt.show()
```





Where the last graph is the plot of the cumulative distribution function.

Any normal distribution can be ``standardized" as follows $z = \frac{x-\mu}{\sigma}$ This has the same bell shape as x but with mean 0 and variance 1. Use the above code to generate and plot the pdf and the cdf of the standard normal distribution.

Bivariate Normal distribution

Suppose that two variables, say x and y, are jointly normal distributed with mean respectively μ_x, μ_y and variance respectively σ_x^2 and σ_y^2. Moreover suppose that the covariance between the two variables is σ_{xy}. This can be represented as follows:

$$\begin{bmatrix} x \\ \\ y \end{bmatrix} \sim N\left(\begin{bmatrix} \mu_x \\ \\ \mu_y \end{bmatrix}; \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \\ \sigma_{xy} & \sigma_y^2 \end{bmatrix}\right)$$

Where the so called variance/covariance matrix is:

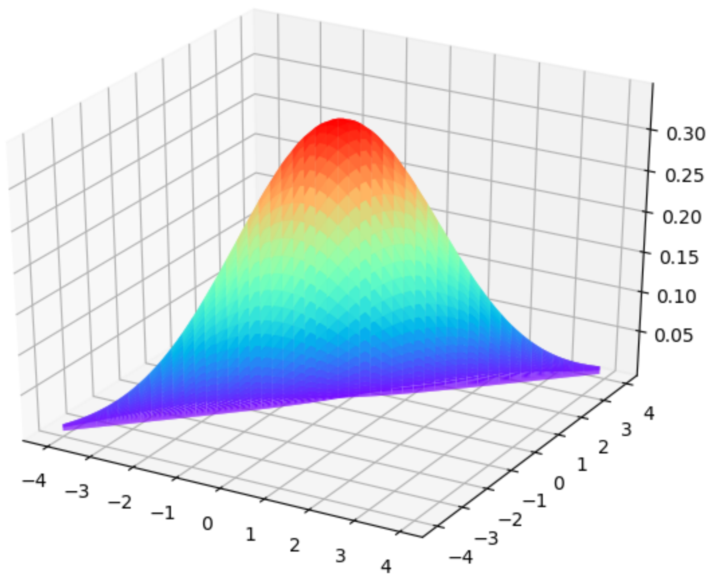
$$\Sigma = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \\ \sigma_{xy} & \sigma_y^2 \end{bmatrix}$$

For practical reason we need to derive the inverse of this matrix, say Σ^{-1} such that $\Sigma\Sigma^{-1} = I$. One can see that this matrix is:

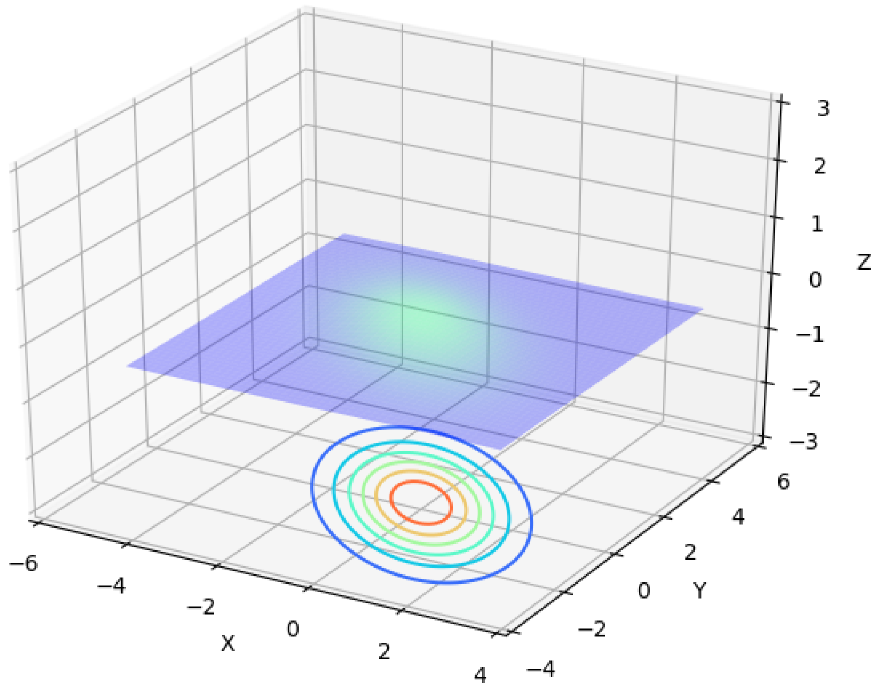
$$\Sigma^{-1} = \begin{bmatrix} \frac{\sigma_y^2}{\sigma_x^2\sigma_y^2-\sigma_{xy}^2} & \frac{-\sigma_{xy}}{\sigma_x^2\sigma_y^2-\sigma_{xy}^2} \\ \frac{-\sigma_{xy}}{\sigma_x^2\sigma_y^2-\sigma_{xy}^2} & \frac{\sigma_x^2}{\sigma_x^2\sigma_y^2-\sigma_{xy}^2} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}} & \frac{\frac{-\sigma_{xy}}{\sigma_y^2}}{\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}} \\ \frac{\frac{-\sigma_{xy}}{\sigma_y^2}}{\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}} & \frac{\frac{\sigma_x^2}{\sigma_y^2}}{\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}} \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 \\ -\frac{\sigma_{xy}}{\sigma_y^2} & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}} & 0 \\ 0 & \frac{1}{\sigma_y^2} \end{bmatrix} \begin{bmatrix} 1 & -\frac{\sigma_{xy}}{\sigma_y^2} \\ 0 & 1 \end{bmatrix}$$

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
import numpy.matlib
mx = 0
my = 0
varx = 0.5
vary = 0.6
covxy = -0.3
Sigma = np.matlib.ones((2,2))
Sigma[0,0] = varx
Sigma[1,1] = vary
Sigma[0,1] = Sigma[1,0] = covxy

x = np.arange(-4,4,8/100)
y = np.arange(-4,4,8/100)

fig = plt.figure()  #定义新的三维坐标轴
ax3 = plt.axes(projection='3d')


X, Y = np.meshgrid(x, y)
Z = (1/(2*np.pi*np.linalg.det(Sigma)**0.5))*np.exp(-0.5*((vary*(X-mx)**2+(Y-my)*
(-2*(X-mx)*covxy+varx*(Y-my)))/(varx*vary-2*covxy)))

ax3.plot_surface(x,y,Z,cmap='rainbow')
plt.show()
```



```
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

#定义坐标轴
fig4 = plt.figure()
ax4 = plt.axes(projection='3d')

#作图
ax4.plot_surface(X,Y,Z,alpha=0.3,cmap='winter')     #生成表面， alpha 用于控制透明度
ax4.contour(X,Y,Z,zdir='z', offset=-3,cmap="rainbow")  #生成z方向投影，投到x-y平面

#设定显示范围
ax4.set_xlabel('X')
ax4.set_xlim(-6, 4)  #拉开坐标轴范围显示投影
ax4.set_ylabel('Y')
ax4.set_ylim(-4, 6)
ax4.set_zlabel('Z')
ax4.set_zlim(-3, 3)

plt.show()
```



Using the results as above one obtains:

$$p(x,y) = \frac{1}{2\pi\left(\sigma_x^2\sigma_y^2 - \sigma_{xy}^2\right)^{\frac{1}{2}}} \times$$

$$\times exp(-0.5\left[\left(x-\mu_x-\frac{\sigma_{xy}}{\sigma_y^2}(y-\mu_y)\right) \quad y-\mu_y\right]\begin{bmatrix} \frac{1}{\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}} & 0 \\ 0 & \frac{1}{\sigma_y^2} \end{bmatrix}\left[\begin{matrix}\left(x-\mu_x-\frac{\sigma_{xy}}{\sigma_y^2}(y-\mu_y)\right) \\ y-\mu_y\end{matrix}\right])$$

Conditional Normal distribution

Since we discussed about both univariate and bivariate normal distribution, it is now relevant to introduce the conditional distribution. In probability, the definition of conditional distribution is the following:

$p(x|y) = \frac{p(x,y)}{p(y)}$

That is: the probability of x given a specific y is the ratio between the joint (bivariate) distribution and the univariate distribution of y.

Now define:

$$\left(x-\mu_x-\frac{\sigma_{xy}}{\sigma_y^2}(y-\mu_y)\right) = x - \mu_{x|y}$$

Then we have:

$$p(x,y) = \frac{1}{2\pi\left(\sigma_x^2\sigma_y^2-\sigma_{xy}^2\right)^{\frac{1}{2}}} exp(-\frac{1}{2}\times\left[\frac{\left(x-\mu_x-\frac{\sigma_{xy}}{\sigma_y^2}(y-\mu_y)\right)^2}{\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}}+\frac{(y-\mu_y)^2}{\sigma_y^2}\right])$$

We can now derive the conditional probability of x given y. This is:

$$p(x|y)=\frac{p(x,y)}{p(y)}=\frac{\frac{1}{2\pi\left(\sigma_x^2\sigma_y^2-\sigma_{xy}^2\right)^{\frac{1}{2}}}exp\left(-\frac{1}{2}\times\left[\frac{\left(x-\mu_x-\frac{\sigma xy}{\sigma_y^2}(y-\mu_y)\right)^2}{\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}}+\frac{\left(y-\mu_y\right)^2}{\sigma_y^2}\right]\right)}{\frac{1}{\sqrt{2\pi\sigma_y^2}}exp\left(-\frac{1}{2}\frac{\left(y-\mu_y\right)^2}{\sigma_y^2}\right)})$$

That can be written as:

$$p(x|y)=\frac{p(x,y)}{p(y)}=\frac{1}{\sqrt{2\pi}\left(\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}\right)^{\frac{1}{2}}}exp\left(-\frac{1}{2}\times\left[\frac{\left(x-\mu_x-\frac{\sigma_{xy}}{\sigma_y^2}(y-\mu_y)\right)^2}{\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}}+\frac{(y-\mu_y)^2}{\sigma_y^2}-\frac{(y-\mu_y)^2}{\sigma_y^2}\right]\right]$$

This simplifies as:

$$p(x|y)=\frac{p(x,y)}{p(y)}=\frac{1}{\sqrt{2\pi\left(\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}\right)}}exp\left(-\frac{1}{2}\times\left[\frac{\left(x-\mu_x-\frac{\sigma_{xy}}{\sigma_y^2}(y-\mu_y)\right)^2}{\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}}\right]\right]$$

Which is again a univariate (yet conditional!) normal distribution with mean:

$$\mu_{x|y}=\left(\mu_x+\frac{\sigma_{xy}}{\sigma_y^2}(y-\mu_y)\right)$$

and variance $\sigma_x^2-\frac{\sigma_{xy}^2}{\sigma_y^2}$. Note that $\left(\mu_x+\frac{\sigma_{xy}}{\sigma_y^2}(y-\mu_y)\right)$ is the mean of x conditional on the value of y. That is the mean of x imposing the information provided by the variable y. This identity is rather relevant since it is key in the derivation of the Kalman filter as shown below.

State-Space models and the Kalman filter

Consider the following State-Space model:

$$\begin{aligned} y_t &= z\alpha_{t-1}+e_t \\ \alpha_t &= c+w\alpha_{t-1}+u_t \end{aligned}$$

We define α_t as the state variable (the one which is unobserved) and y_t the data that we observe (some people define it the observational equation).

Note that c is a constant and $t=1,2,\cdots,n$ is the time such that we assume that we observe n observations. Assuming that e_t and u_t are uncorrelated normal distributed variables such that $e_t\sim N(0,\sigma_e^2)$ and $u_t\sim N(0,\sigma_u^2)$ with $E(e_{t-j}u_{t-i})=0$ for any j and i. Since a linear combination of normal (or Gaussian) variables remains normal we have that the vector

$$\begin{bmatrix} y_t \\ \alpha_t \end{bmatrix}$$

follows a bivariate normal distribution. Now defining $E(\alpha_t)=c+wE(\alpha_{t-1})$, we have that the previous vector is Normal with mean

$$\begin{bmatrix} zE(\alpha_{t-1}) \\ c+wE(\alpha_{t-1}) \end{bmatrix}$$

While the matrix/covariance matrix of the vector is

$$\Sigma = \begin{bmatrix} z^2E(\alpha_{t-1}-E[\alpha_{t-1}])^2+\sigma_e^2 & zwE(\alpha_{t-1}-E[\alpha_{t-1}])^2 \\ zwE(\alpha_{t-1}-E[\alpha_{t-1}])^2 & w^2E(\alpha_{t-1}-E[\alpha_{t-1}])^2+\sigma_u^2 \end{bmatrix}$$

Now imagine that we observe y_t, but we do not observe α_t. This may represents an obstacle since we are not able to know neither $E(\alpha_t)$ nor $E(\alpha_{t-1}-E[\alpha_{t-1}])^2$.
However, we can make use of the conditional distribution properties, as above, to derive $E(\alpha_t|y_t)$ and $p_{t-1}=E(\alpha_{t-1}-E[\alpha_{t-1}|y_{t-1}])^2$:

Therefore we can now derive the conditional probability of α_t given the information provided by y_t.
Define $E(\alpha_t|y_t)=a_t$ and $v_t=y_t-zE(\alpha_{t-1}|y_{t-1})=z[\alpha_{t-1}-a_{t-1}]+e_t$ such that

$$E(v_t^2)=z^2E[\alpha_{t-1}-E(\alpha_{t-1}|y_{t-1})]+e_t]^2=z^2p_{t-1}+E[e_t]^2=z^2p_{t-1}+\sigma_e^2$$

It then follows that

$$a_t=E(\alpha_t|y_t)=c+wE(\alpha_{t-1}|y_{t-1})+\frac{zwp_{t-1}}{z^2p_{t-1}+\sigma_e^2}(y_t-zE(\alpha_{t-1}|y_{t-1}))=c+wa_{t-1}+\frac{zwp_{t-1}}{z^2p_{t-1}+\sigma_e^2}v_t$$

For simplicity, we define $k_t=\frac{zwp_{t-1}}{z^2p_{t-1}+\sigma_e^2}$. In addition, we also have that the conditional variance is:

$$\begin{aligned} &p_t=E[\alpha_t-E(\alpha_t|y_t)]^2= \\ &E[c+w\alpha_{t-1}+u_t-c-wa_{t-1}-k_tv_t]^2= \\ &E[w(\alpha_{t-1}-a_{t-1})+u_t-k_t\left(z[\alpha_{t-1}-a_{t-1}]+e_t\right)]^2= \\ &E[(w-k_tz)\left(\alpha_{t-1}-a_{t-1}\right)+u_t-k_te_t]^2= \\ &(w-k_tz)^2E(\alpha_{t-1}-a_{t-1})^2+\sigma_u^2+k_t^2\sigma_e^2= \\ &w^2p_{t-1}-2wzk_tp_{t-1}+k_t^2(z^2p_{t-1}+\sigma_e^2)+\sigma_u^2= \\ &w^2p_{t-1}-wzk_tp_{t-1}+\sigma_u^2 \end{aligned}$$

Now defining the information up to time t-1 as follows: $Y_{t-1}=[y_1,y_2,\cdots,y_{t-1}]$. We can now write the following distribution for the vector

$$\begin{bmatrix} y_t \\ x_t \end{bmatrix}|Y_{t-1}\sim N\left(\begin{bmatrix} za_{t-1} \\ c+wa_{t-1} \end{bmatrix};\begin{bmatrix} z^2p_{t-1}+\sigma_e^2 & wzp_{t-1} \\ wzp_{t-1} & w^2p_{t-1}-\frac{(zwp_{t-1})^2}{z^2p_{t-1}+\sigma_e^2}+\sigma_u^2 \end{bmatrix}\right)$$

We can now collect the main recursions that we have been discussed so far:

$$\begin{aligned} p_t&=w^2p_{t-1}-wzk_tp_{t-1}+\sigma_u^2 \\ k_t&=\frac{zwp_{t-1}}{z^2p_{t-1}+\sigma_e^2} \\ a_t=E(\alpha_t|y_t)&=c+wa_{t-1}+\frac{zwp_{t-1}}{z^2p_{t-1}+\sigma_e^2}(y_t-za_{t-1})=c+wa_{t-1}+k_t\left(y_t-za_{t-1}\right) \\ v_t&=(y_t-za_{t-1}) \end{aligned}$$

These recursions are the well-known **Kalman filter**!

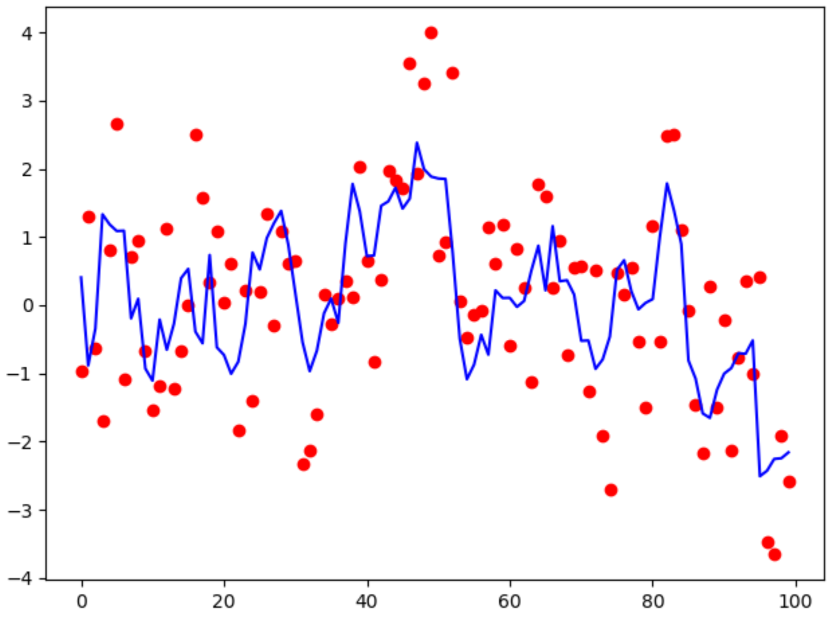
Kalman filter example

Consider the following model:

$$\begin{aligned} y_t&=\alpha_{t-1}+e_t \\ \alpha_t&=0.9\alpha_{t-1}+u_t \end{aligned}$$

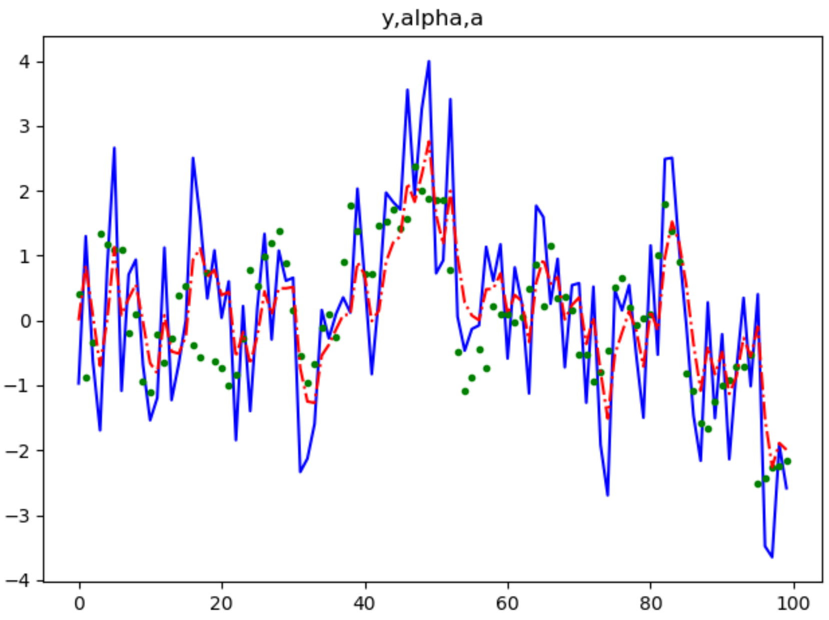
Where $\sigma_e^2=.8$ and $\sigma_u^2=.4$. Now we can generate it using the following code:

```
import numpy as np
import matplotlib.pylab as plt
n = 100
np.random.seed(123)
e = np.sqrt(0.8)*np.random.randn(n)
u = np.sqrt(0.4)*np.random.randn(n)
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
y[0] = e[0]
alpha[0] = u[0]
for t in range(1,n):
    y[t] = alpha[t-1] + e[t]
    alpha[t] = 0.9*alpha[t-1]+u[t]
plt.plot(y,'ro')
plt.plot(alpha,'b-')
plt.show()
```



We now implement the Kalman filter as in (2). First note that since the state equation α_t is an Autoregressive process of order one we can initialize $a_1 = 0$ and $p_1 = \frac{\sigma_u^2}{1-0.81} = 2.11$ (the last is the variance of α_t). Now, assuming that we know the variances of the noises (this is not the case in empirical situations) we can make use of the following code:

```
n = 100
sigmae = 0.8
sigmau = 0.4
w = 0.9
z = 1

k = [0*i for i in range(n)]
v = [0*i for i in range(n)]
a = [0*i for i in range(n)]
p = [0*i for i in range(n)]
a[0] = 0
p[0] = 2.11
for t in range(1,n):
    k[t] = (z*w*p[t-1])/(z**2 *p[t-1]+sigmae)
    p[t] = w**2 * p[t-1] - w*z * k[t] * p[t-1] + sigmau
    v[t] = y[t] - z*a[t-1]
    a[t] = w*a[t-1] + k[t]*v[t]
plt.plot(y,'b-')
plt.plot(alpha,'g.')
plt.plot(a,'r-.')
plt.title("y,alpha,a")
plt.show()
```



Wonderful and very easy to implement using Python!

Likelihood function and model estimation

We have shown the derivation of the Kalman filter and its implementation assuming we know the variances of the errors. However, the question that now comes: what are the parameters to be chosen in order to make an estimate of $E(\alpha_t|Y_t)$ as closed as possible to $E(\alpha_t)$?

This is very important since in practice we do not know these variances and we need to make a guess (an estimation).

First of all, since y_t is a combination of normal variables, it has also a normal distribution. We can therefore write the Probability density function of y as follows:

$$Probability(y) = \frac{1}{\sqrt{(2\pi\sigma_y^2)}} exp(-\frac{1}{2}\frac{(y - E(\alpha|Y))^2}{\sigma_y^2}]$$

Secondly, since our aim is to forecast y_t, we are looking for the set of parameters that help us contructing a_t as closed as possible to y_t.

Now, as shown above, we have that the variance of y is $\sigma_y^2 = z^2 p_{t-1} + \sigma_e^2$. It is important to focus on the following difference $y_t - a_{t-1}$ that we defined as v_t, representing the ``prediction error" when predicting y_t using a_{t-1}. Therefore we can write the probability distribution for a generic time y_t as:

$$Probability(y_t) = \frac{1}{\sqrt{2\pi(z^2 p_{t-1} + \sigma_e^2)}} exp(-\frac{1}{2}\frac{v_t^2}{(z^2 p_{t-1} + \sigma_e^2)})$$

Assuming that the observations y_t are independent, the joint distribution defined as likelihood function is:

$$Likelihood = Prob(y_1) \times Prob(y_2) \times Prob(y_3) \cdots Prob(y_n) = \prod_{t=1}^{n} Prob(y_t) =$$
$$\left(\Pi_{t=1}^{n} 2\pi^{-\frac{1}{2}} (z^2 p_{t-1} + \sigma_e^2)^{-\frac{1}{2}}\right) exp(-\tfrac{1}{2} \textstyle\sum_{t=1}^{n} \frac{v_t^2}{(z^2 p_{t-1} + \sigma_e^2)})$$

Therefore, the set of parameters to be chosen in order to make an estimate of $E(\alpha_t|Y_t)$ as closed as possible to y_t is the one that maximize the log L function. Or alternatively, the set that minimize minus the log L function.

One potential issue of using the Log-likeliood as above is related with the number of parameters. Indeed, the likelihood is a single value and gets maximized by all parameters. Therefore the less parameters we have, the more efficient the estimation. In the univariate framework, one trick that we can use is to concentrate out the likelihood function.

Rewrite the log-Likelihood above as follows:

$$\log L = -\frac{n}{2}\log(2\pi) - \frac{1}{2}\sum_{t=1}^{n}\log[(z^2\tilde{p}_{t-1} + 1) \times \sigma_e^2] - \frac{1}{2}\sum_{t=1}^{n}\left(\frac{v_t^2}{[(z^2\tilde{p}_{t-1} + 1) \times \sigma_e^2]}\right)$$

Where $\tilde{p}_{t-1}\sigma_e^2 = p_{t-1}$. Now defining $\frac{\sigma_u^2}{\sigma_e^2} = q$, the Kalman filter recursions for this reparametrized model can be written as:

$$\begin{aligned} \tilde{p}_t &= w^2\tilde{p}_{t-1} - wzk_t\tilde{p}_{t-1} + q \\ k_t &= \frac{zw\tilde{p}_{t-1}}{z^2\tilde{p}_{t-1} + 1} \\ a_t &= c + wa_{t-1} + k_t\left(y_t - za_{t-1}\right) \\ v_t &= (y_t - za_{t-1}) \end{aligned}$$

Interestingly, k_t is not affected by the fact that we consider $\tilde{p}_t$ in place of p_t, neither are a_t and v_t.

Now note that $\tilde{\sigma}_e^2 = \frac{1}{(n-1)}\sum_{t=2}^{n}\left(\frac{v_t^2}{z^2\tilde{p}_{t-1}+1}\right)$ represents a consistent estimator of σ_e^2. Therefore the concentrated log-Likelihood gets:

$$\log L = -\frac{n}{2}[\log(2\pi) + 1] - \frac{1}{2}\sum_{t=1}^{n}\log(z^2\tilde{p}_{t-1} + 1) - \frac{n}{2}\log[\tilde{\sigma}_e^2]$$

Thererfore, one can now choose the set of parameters (q; w; z ; c) that maximize the last expression. Once the set is estimated, we can estimate all the variances as follows:

$$\tilde{\sigma}_e^2 = \frac{1}{(n-1)}\sum_{t=2}^{n}\left(\frac{v_t^2}{z^2\tilde{p}_{t-1} + 1}\right)$$

$$\tilde{\sigma}_u^2 = q * \tilde{\sigma}_e^2$$

Initializing the Kalman filter recursions

These recursions need to be initialized. In particular, when we are at time $t = 1$ we need to determine the value of a_1. In principle the choice of this value depends on the type of process followed by x_t. For example, when x_t follows an AR(1) process (i.e. $0 \leq w \leq 1$) , the best choice is $a_1 = 0$ since the process is mean reverting. On the other hand, when x_t is a random walk (i.e. $w = 1$) it is better to use $a_1 = y_1$. We do not consider the case when $w > 1$ since this would be an explosive autoregressive process. Finally, when using the **standard Kalman filter recursions**, one should also initialize p_t.

For the AR(1) case, as shown above, one can initialize as follows $p_1 = \frac{\sigma_u^2}{1-\omega^2}$. When $\omega = 1$, as in the random walk process, one should use $p_1 = \infty$. Indeed, the so called diffuse initialization proposes, for the case considered, $p_1 = 10000$, an arbitrary high number.

Concentrated Log-likelihood in action!

The code below generate a model and estimate it using the Concentrated Log-likelihood with (2):

```
import numpy as np
from scipy.optimize import minimize
import matplotlib.pylab as plt
n = 100
np.random.seed(61)
su = 0.1
se = 0.4
qreal = su/se
e = np.sqrt(se)*np.random.randn(n)
u = np.sqrt(su)*np.random.randn(n)
z = 1
wreal = 0.97
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
y[0] = e[0]
alpha[0] = u[0]
for t in range(1,n):
    y[t] = z*alpha[t-1] + e[t]
    alpha[t] = wreal*alpha[t-1]+u[t]
############  Standard Kalman filter approach  ##########################
a = [0*i for i in range(n)]
p = [0*i for i in range(n)]
a[0] = 0
p[0] = 10
k = [0*i for i in range(n)]
v = [0*i for i in range(n)]

def myfun0(x):
    z = 1
    likelihood = 0
    sigmae = 0
    for t in range(1, n):
        k[t] = (z * x[0] * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = x[0] ** 2 * p[t - 1] - x[0] * z * k[t] * p[t - 1] + x[1]
        v[t] = y[t] - z * a[t - 1]
        a[t] = x[0] * a[t - 1] + k[t] * v[t]
        sigmae = sigmae + (v[t]**2/(z**2 * p[t-1] + 1))
        likelihood = likelihood + 0.5*np.log(2*np.pi)+0.5+0.5*np.log(z**2 * p[t-
1] + 1)
    likelihood = likelihood + 0.5 * n * np.log(sigmae / n)
    return likelihood

def myfu():
    v = lambda x: myfun0(x)
    return v

x0 = np.asarray((0.85,0.5))
```

```
res = minimize(myfu(), x0, method='SLSQP')
print(res)
print(res.x)
```

Running Result:

```
 fun: 122.64056133291024
     jac: array([0.02630234, 0.0004282 ])
 message: 'Optimization terminated successfully.'
    nfev: 36
     nit: 7
    njev: 7
  status: 0
 success: True
       x: array([0.90283944, 0.76331662])
[0.90283944 0.76331662]
```

One can see that the estimates are rather closed to the true parameters.

State-Space models and the Kalman filter in action!

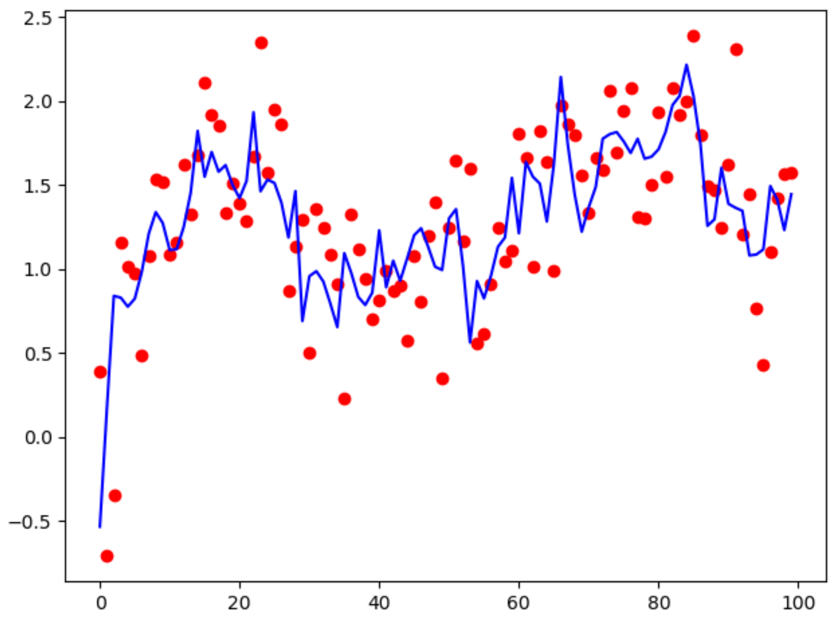
We can now stop with the theory and start getting fun using R and reproducing the results we obtained above.

For example suppose we want to generete the following state-space model:

$$\begin{aligned} y_t &= \alpha_{t-1} + e_t \\ \alpha_t &= .2 + .85\alpha_{t-1} + u_t \end{aligned}$$

where $e \sim Normal(\mu_e = 0; \sigma_e^2 = .1)$ and $u \sim Normal(\mu_u = 0; \sigma_u^2 = .05)$ where $n = 100$. In addition suppose we want to estimate the parameters of the model.

```
import numpy as np
import matplotlib.pylab as plt
n = 100
np.random.seed(1265)
e = np.sqrt(0.1)*np.random.randn(n)
u = np.sqrt(0.05)*np.random.randn(n)
constant = 0.2
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
y[0] = e[0]
alpha[0] = u[0]
for t in range(1,n):
    y[t] = alpha[t-1] + e[t]
    alpha[t] = constant + 0.85*alpha[t-1] + u[t]
plt.plot(y,'ro')
plt.plot(alpha,'b-')
plt.show()
```



Using the standard KF we obtain the following results:

```
import numpy as np
import matplotlib.pylab as plt
from scipy.optimize import minimize
import pandas as pd
n = 100
np.random.seed(1265)
e = np.sqrt(0.1)*np.random.randn(n)
u = np.sqrt(0.05)*np.random.randn(n)
constant = 0.2
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
y[0] = e[0]
alpha[0] = u[0]
for t in range(1,n):
    y[t] = alpha[t-1] + e[t]
    alpha[t] = constant + 0.85*alpha[t-1] + u[t]
plt.plot(y,'ro')
plt.plot(alpha,'b-')
# plt.show()

a = [0*i for i in range(n)]
p = [0*i for i in range(n)]
a[0] = 0
p[0] = 1
k = [0*i for i in range(n)]
v = [0*i for i in range(n)]


def myfun0(x):
    z = 1
    likelihood = 0
    sigmae = 0
    for t in range(1, n):
        k[t] = (z * x[0] * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = x[0] ** 2 * p[t - 1] - x[0] * z * k[t] * p[t - 1] + x[1]
        v[t] = y[t] - z * a[t - 1]
        a[t] = x[2] + x[0] * a[t - 1] + k[t] * v[t]
        sigmae = sigmae + (v[t]**2/(z**2 * p[t-1] + 1))
        likelihood = likelihood + 0.5*np.log(2*np.pi)+0.5+0.5*np.log(z**2 * p[t-
1] + 1)
    likelihood = likelihood + 0.5 * n * np.log(sigmae / n)
    return likelihood

def myfu():
    v = lambda x: myfun0(x)
    return v

x0 = np.asarray((0.9,1,0.1))
res = minimize(myfu(), x0, method='SLSQP')
# print(res)
# print(res.x)

v[1] = 0
w = abs(res.x[0])
q = abs(res.x[1])
co = abs(res.x[2])
from math import log,pi
z = 1
likelihood = sigmae = 0
for t in range(1,len(y)):
    k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
    p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + q
    v[t] = y[t] - z * a[t - 1]
    a[t] = co + w * a[t - 1] + k[t] * v[t]
    sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))
    likelihood = likelihood + 0.5 * np.log(2 * np.pi) + 0.5 + 0.5 * np.log(z **
2 * p[t - 1] + 1)
likelihood = likelihood + 0.5 * n * np.log(sigmae / n)
sigmae = sigmae/len(y)
sigmau = q * sigmae
result =
pd.DataFrame({'co':co,'w':w,'z':z,'sigmae':sigmae,'sigmau':sigmau},index=[0])
print(result)
```

Running Result:

```
         co         w  z  sigmae    sigmau
0  0.350461  0.750141  1    0.09  0.050488
```

The Local level model (or simple exponential smoothing)

The local level is one of the simplest state-space models. This model assumes $w = z = 1$ and $constant = 0$, such that we have:

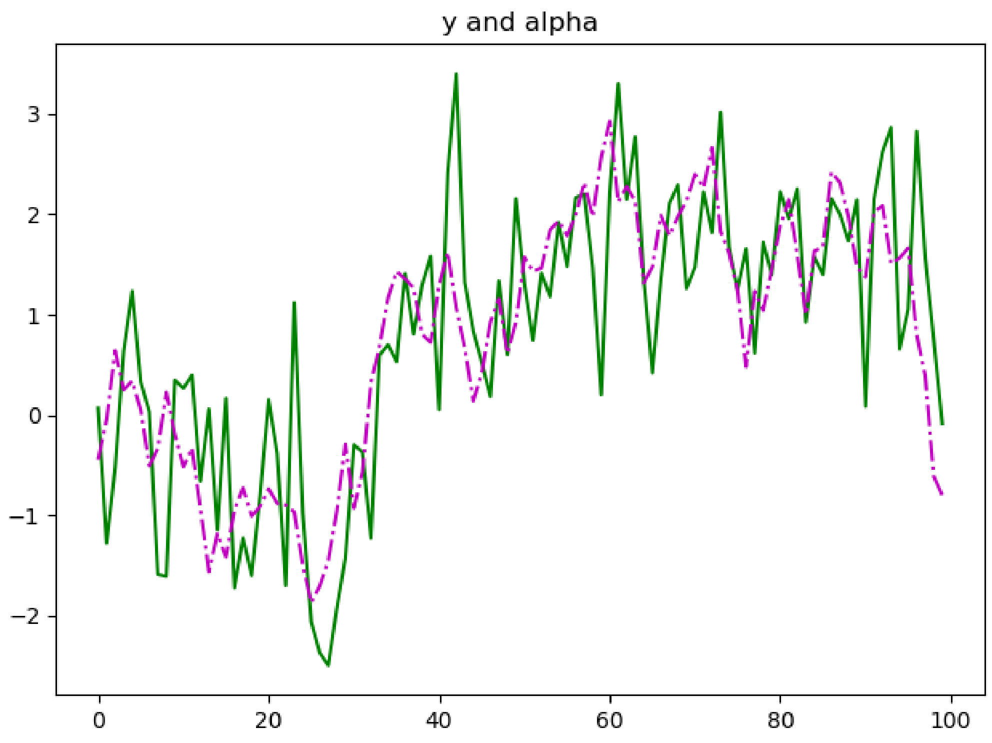
$$y_t = \alpha_{t-1} + e_t$$
$$\alpha_t = \alpha_{t-1} + u_t$$

As a consequence one has only one parameter to estimate q. This is the only ingredient that maximize the concentreted log Likelihood function.

For example suppose we want to generete the following state-space model:

where $e \sim Normal(\mu = 0; \sigma_e^2 = .5)$ and $u \sim Normal(\mu = 0; \sigma_u^2 = .2)$ where $n = 100$. Here q is 0.4.

```
import numpy as np
import matplotlib.pylab as plt
from scipy.optimize import minimize
import pandas as pd
np.random.seed(153)
n = 100
e = np.sqrt(0.5)* np.random.randn(n)
u = np.sqrt(0.2)* np.random.randn(n)
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
y[0] = e[0]
alpha[0] = u[0]
for t in range(1,n):
    y[t] = alpha[t-1] + e[t]
    alpha[t] = alpha[t-1] + u[t]
plt.plot(y,'g-')
plt.plot(alpha,'m-.')
plt.title("y and alpha")
plt.show()
```



We can now estimate this model with the KF recursions. The code below estimate the model with the 4 recursions.

```
import numpy as np
import matplotlib.pylab as plt
from scipy.optimize import minimize
import pandas as pd

a = [0*i for i in range(n)]
p = [0*i for i in range(n)]
a[0] = 0
```

```
p[0] = 10000
k = [0*i for i in range(n)]
v = [0*i for i in range(n)]


def myfun0(x):
    z = w = 1
    likelihood = 0
    sigmae = 0
    for t in range(1, n):
        k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + x[0]
        v[t] = y[t] - z * a[t - 1]
        a[t] = w * a[t - 1] + k[t] * v[t]
        sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))
        likelihood = likelihood + 0.5 * np.log(2 * np.pi) + 0.5 + 0.5 * np.log(z
** 2 * p[t - 1] + 1)
    likelihood = likelihood + 0.5 * n * np.log(sigmae / n)
    return likelihood

def myfu():
    v = lambda x: myfun0(x)
    return v

def con(args):
    qmin, qmax = args
    cons = ({'type': 'ineq', 'fun': lambda q: q - qmin},{'type': 'ineq', 'fun':
lambda q: -q + qmax})
    return cons

x0 = np.asarray((0.2))
args = (0,1)
cons = con(args)
res = minimize(myfu(), x0, method='SLSQP',constraints=cons)
print(res)
print(res.x)
```

Running Result:

```
fun: 138.24154414349104
     jac: array([0.00025368])
 message: 'Optimization terminated successfully.'
    nfev: 16
     nit: 5
    njev: 5
  status: 0
 success: True
       x: array([0.17185308])
[0.17185308]
```

We now derive the estimates of the parameters (the two variances)

```
z = w = 1
likelihood = 0
sigmae = 0
q = res.x[0]
for t in range(1, n):
    k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
    p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + q
    v[t] = y[t] - z * a[t - 1]
    a[t] = w * a[t - 1] + k[t] * v[t]
    sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))
print(sigmae/(n-1))
print(q*(sigmae)/(n-1))
```

Running Result:

```
0.4851380906039108
0.2500557781404861
```

Consider the following model:

$$\begin{gathered} y_t = y_{t-1}\beta_{t-1} + e_t \\ \beta_t = \beta_{t-1} + u_t \end{gathered}$$

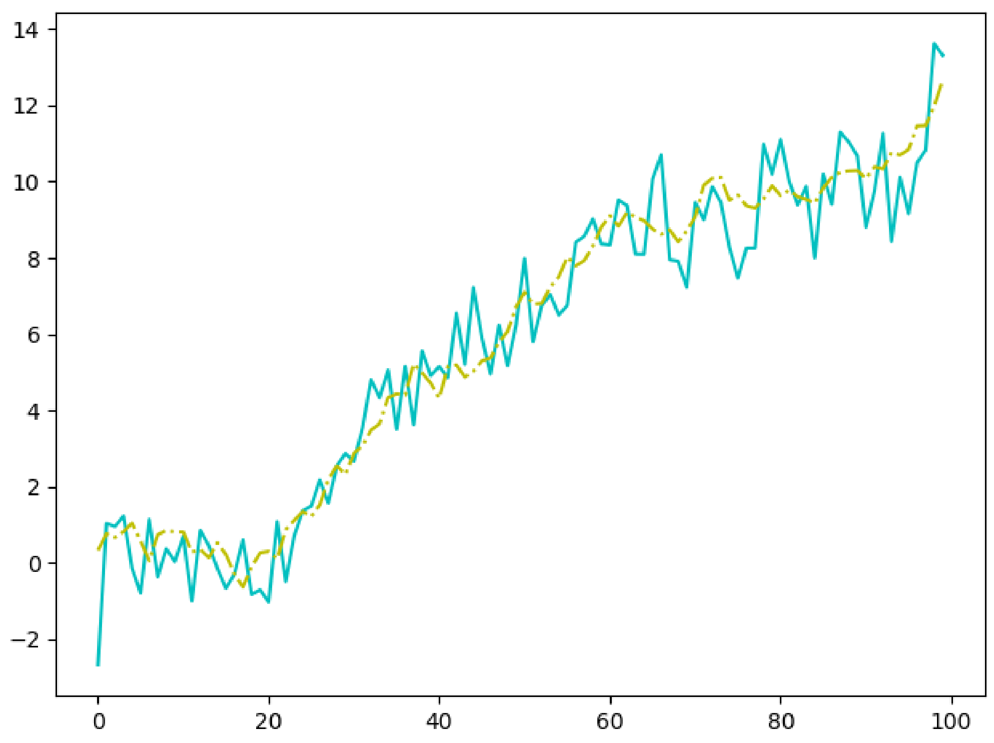
This model is a variant of model (3). Now try generate it and estimate it using the standard Kalman filter approach.

The Local Level with drift: (the Theta method)

This model is a simple variant of the local level but it is quite popular among the forecasters since it competes very well with other strong benchmark. This is also a reparametrization of the so-called Theta method as shown in (TOADD).

For example suppose we want $e \sim Normal(\mu = 0; \sigma_e^2 = .8)$, $u \sim Normal(\mu = 0; \sigma_u^2 = .1)$ and $constant = .1$, where $n = 100$. Here q is 0.125. This code generates the local level with drift:

```
import numpy as np
import matplotlib.pylab as plt
n = 100
np.random.seed(572)
e = np.sqrt(0.8)*np.random.randn(n)
u = np.sqrt(0.1)*np.random.randn(n)
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
co = 0.1
y[0] = e[0]
alpha[0] = u[0]
for t in range(1,n):
    y[t] = alpha[t-1] + e[t]
    alpha[t] = co + alpha[t-1]+u[t]
plt.plot(y,'c-')
plt.plot(alpha,'y-.')
plt.show()
```



The solid line is the observed data y_t while the dotted line is the state α_t. We now estimate the model maximizing the concentrated log-likelihood using the KF:

```
a = [0*i for i in range(n)]
p = [0*i for i in range(n)]
a[0] = y[0]
p[0] = 10000
k = [0*i for i in range(n)]
v = [0*i for i in range(n)]
v[0] = 0

def myfunTheta0(x):
    z = w = 1
    likelihood = 0
    sigmae = 0
    for t in range(1, n):
        print(t)
        k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + x[0]
        v[t] = y[t] - z * a[t - 1]
        a[t] = x[1] + w * a[t - 1] + k[t] * v[t]
        sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))
        likelihood = likelihood + 0.5 * np.log(2 * np.pi) + 0.5 + 0.5 * np.log(z
** 2 * p[t - 1] + 1)
    likelihood = likelihood + 0.5 * n * np.log(sigmae / n)
    return likelihood

def myfuTheta():
    v = lambda x: myfunTheta0(x)
    return v

x0 = np.asarray((0.6,0.2))
res = minimize(myfuTheta(), x0)
```

```
q = abs(res.x[0])
co = abs(res.x[1])
z = w = 1
sigmae = 0
for t in range(1,len(y)):
    k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
    p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + q
    v[t] = y[t] - z * a[t - 1]
    a[t] = co + w * a[t - 1] + k[t] * v[t]
    sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))

print(sigmae/(n-1))
print(q*(sigmae/(n-1)))
```

Running Result:

```
0.8155724083512
0.1534236175394
```

Suppose we want to create a function that generate a time series model and a function that estimate the model using the steady-stae approach, say the Theta model, and return the estimates of the parameters, the estimates of a_t and the prediction error v_t.

The first function can be created like that:

```
import numpy as np
def generateTheta(n,sigmae,sigmau,co):
    e = np.sqrt(sigmae)*np.random.randn(n)
    u = np.sqrt(sigmau)*np.random.randn(n)
    y = [0*i for i in range(n)]
    alpha = [0*i for i in range(n)]
    y[0] = e[0]
    alpha[0] = u[0]
    for t in range(1,n):
        alpha[t] = co + alpha[t-1] + u[t]
        y[t] = alpha[t-1] + e[t]
    return y
```

The second function can be created like that:

```
import numpy as np
from scipy.optimize import minimize
import pandas as pd

def myfunTheta0(x):
    n = len(y)
    z = w = 1
    likelihood = 0
    sigmae = 0

    a = [0*i for i in range(n)]
    p = [0*i for i in range(n)]
    a[0] = 0
    p[0] = 0
    k = [0 * i for i in range(n)]
    v = [0 * i for i in range(n)]
    for t in range(1, n):
        k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + x[0]
        v[t] = y[t] - z * a[t - 1]
        a[t] = x[1] + w * a[t - 1] + k[t] * v[t]
        sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))
        likelihood = likelihood + 0.5 * np.log(2 * np.pi) + 0.5 + 0.5 * np.log(z
** 2 * p[t - 1] + 1)
    likelihood = likelihood + 0.5 * n * np.log(sigmae / n)
    return likelihood

def myfuTheta():
    v = lambda x: myfunTheta0(x)
    return v

def EstimateTheta():
    x0 = np.asarray((0.5, 0.2))
    res = minimize(myfuTheta(), x0)
    v = [0 * i for i in range(n)]
    v[0] = 0
    w = z = 1
    q = abs(res.x[0])
    co = abs(res.x[1])
    sigmae = 0

    a = [0*i for i in range(n)]
    p = [0*i for i in range(n)]
    a[0] = 0
    p[0] = 0
    k = [0 * i for i in range(n)]
    v = [0 * i for i in range(n)]

    for t in range(1,len(y)):
        k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + q
        v[t] = y[t] - z * a[t - 1]
        a[t] = co + w * a[t - 1] + k[t] * v[t]
        sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))
    sigmae = sigmae/len(y)
    sigmau = q*sigmae
    thelist = pd.DataFrame({'sigmae':sigmae,'sigmau':sigmau,'co':co},index=[0])
    return  thelist
```

Let's test if they work!

```
np.random.seed(11)
y = generateTheta(100,.6,.2,1)
ans = EstimateTheta(generateTheta(100,.6,.2,1))
print(ans)
```

Running Result:

```
     sigmae    sigmau        co
 0  0.679695  0.184705  1.002001
```

Single Source of Error approach

The literature in the last two dacades has focused the attention on the state-space model with just one source of error.

Consider the following State-Space model:

$$y_t = z\alpha_{t-1} + e_t$$
$$\alpha_t = c + w\alpha_{t-1} + \gamma e_t$$

This model differs from (1) since there is only one noise determining both the state-equation and the observations. One important feature with this approach is that we do not need to run all the four KF recursions but only two, that is :

$$e_t = y_t - za_{t-1}$$
$$a_t = c + wa_{t-1} + \gamma e_t$$

The likelihood function for model (7) can be derived easily. First of all, when e_t is Normal then also y_t is. We can therefore write the Probability density function of y as follows:

$$Probability(y) = \frac{1}{\sqrt{(2\pi\sigma_y^2)}} exp(-\frac{1}{2}\frac{(y - za_{t-1})^2}{\sigma_y^2}) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp(-\frac{1}{2}\frac{e_t^2}{\sigma_y^2})$$

The main difference with the framework discussed above is that the prediction error estimated using $e_t = y_t - za_{t-1}$ corresponds to the error in the observational equation.

Assuming that the observations y_t are independent, the joint distribution defined as likelihood function is:

$$Likelihood = Prob(y_1) \times Prob(y_2) \times Prob(y_3) \cdots Prob(y_n) = \prod_{t=1}^{n} Prob(y_t) =$$
$$\Pi_{t=1}^{n}(2\pi^{-\frac{1}{2}}\sigma_y^2)^{-.5} exp\left(\sum_{t=1}^{n} -\frac{1}{2}\frac{e_t^2}{\sigma_y^2}\right)$$

The log-Likelihood function for this representation is the following

$$logL = -\frac{n}{2}\log(2\pi\sigma_y^2) - \frac{1}{2}\sum_{t=1}^{n}\frac{e_t^2}{\sigma_y^2}$$

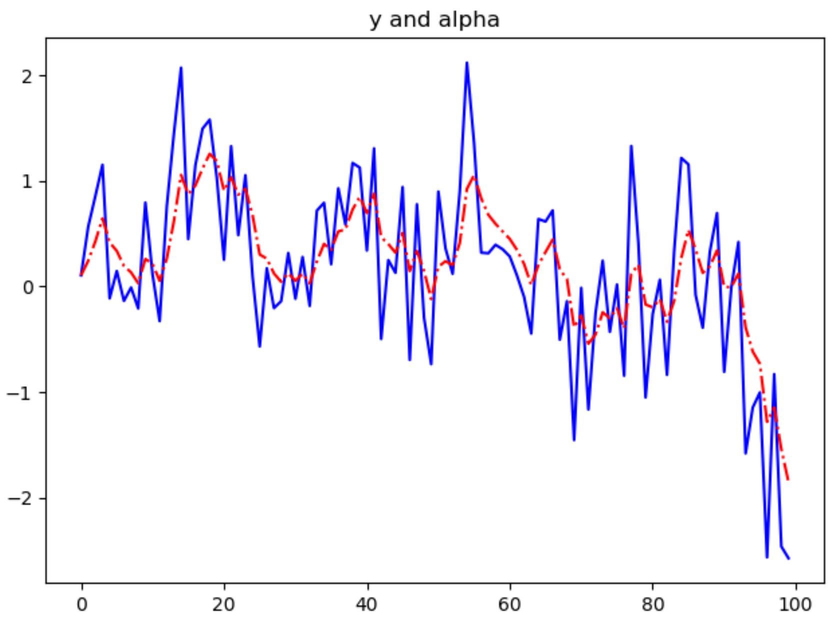
Note that the likelihood can be maximized by just minimizing $\sum e_t^2$. This is very simple!

The exponential smoothing with one source of error

Lets see the single source of error is practice. Below I generate an exponential smoothing model:

```
import numpy as np
import matplotlib.pylab as plt
np.random.seed(213)
n = 100
e = np.sqrt(0.6)*np.random.randn(n)
gamma = 0.3
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
y[0] = e[0]
```

```
alpha[0] = e[0]
for t in range(1,n):
    y[t] = alpha[t-1] + e[t]
    alpha[t] = alpha[t-1] + gamma * e[t]
plt.plot(y,'b-')
plt.plot(alpha,'r-.')
plt.title("y and alpha")
plt.show()
```



We can now estimate this model with the two recursions.

```
import numpy.matlib
from scipy.optimize import minimize

a = [0*i for i in range(n)]
a[0] = y[0]
e = np.matlib.zeros((len(y),1))

def myfun0(x):
    e2 = 0
    for t in range(1, n):
        e[t] = y[t] - a[t-1]
        e2 = e2+ (y[t] - a[t-1])**2
        a[t] = a[t-1] + x[0] *e[t]
    return e2/n

def myfu():
    v = lambda x: myfun0(x)
    return v

def con(args):
    bmin, bmax = args
    cons = ({'type': 'ineq', 'fun': lambda b: b - bmin},{'type': 'ineq', 'fun':
lambda b: -b + bmax})
    return cons

x0 = np.asarray((0.2))
args1 = (0,1)
cons = con(args1)
```

```
res = minimize(myfu(), x0, method='SLSQP',constraints=cons)
print(res)
print(res.x)
```

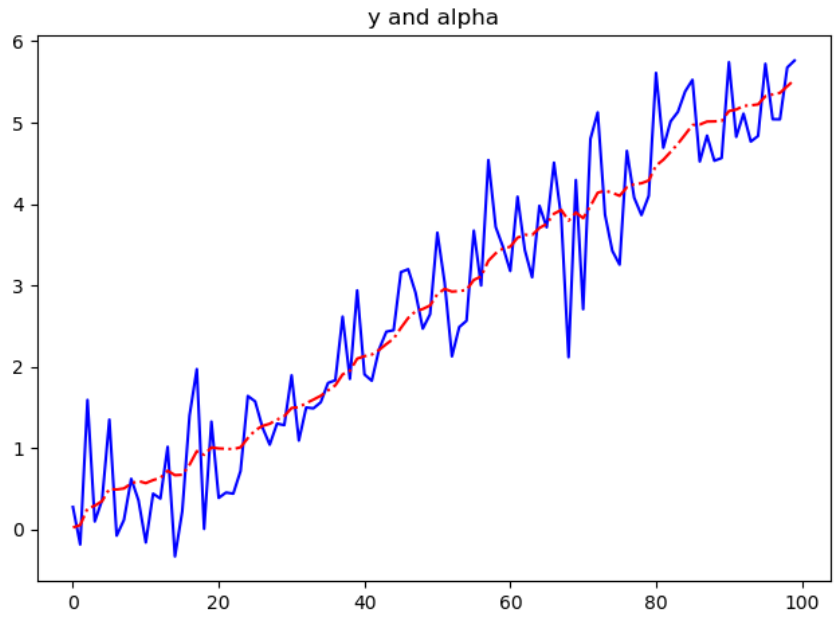
Running Result:

```
fun: 0.5314978130781608
     jac: array([-0.00024597])
 message: 'Optimization terminated successfully.'
    nfev: 13
     nit: 4
    njev: 4
  status: 0
 success: True
       x: array([0.3620233])
[0.3620233]
```

The Theta method with one source of error

Lets see the single source of error is practice. Below I generate an exponential smoothing model:

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

np.random.seed(5)
n = 100
e = np.sqrt(0.4)* np.random.randn(n)
gamma = 0.1
con = 0.05
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
y[0] = e[0]
alpha[0] = gamma*e[0]
for t in range(1,n):
    y[t] = alpha[t-1] + e[t]
    alpha[t] = con + alpha[t-1] + gamma*e[t]
plt.plot(y,'b-')
plt.plot(alpha,'r-.')
plt.title("y and alpha")
plt.show()
```



We can now estimate this model with the two recursions.

```
import numpy.matlib
from scipy.optimize import minimize
a = [0*i for i in range(n)]
a[0] = y[0]
e = np.matlib.zeros((len(y),1))

def myfun0(x):
    e2 = 0
    for t in range(1, n):
        e[t] = y[t] - a[t-1]
        e2 = e2 + (y[t] - a[t-1])**2
        a[t] = x[1] + a[t-1] + x[0] *e[t]
    return e2/n

def myfu():
    v = lambda x: myfun0(x)
    return v

x0 = np.asarray((0.2,0.1))
res = minimize(myfu(), x0, method='SLSQP')
print(res)
print(res.fun)
print(res.x)
```

```
     fun: 0.3617924244177702
     jac: array([-4.35039401e-05, -6.38384372e-04])
 message: 'Optimization terminated successfully.'
    nfev: 45
     nit: 10
    njev: 10
  status: 0
 success: True
       x: array([0.12396209 0.05532481])
0.3617924244177702
[0.12396209 0.05532481]
```

Note that the estimation returns the variance of the noise as well as the γ and the constant.

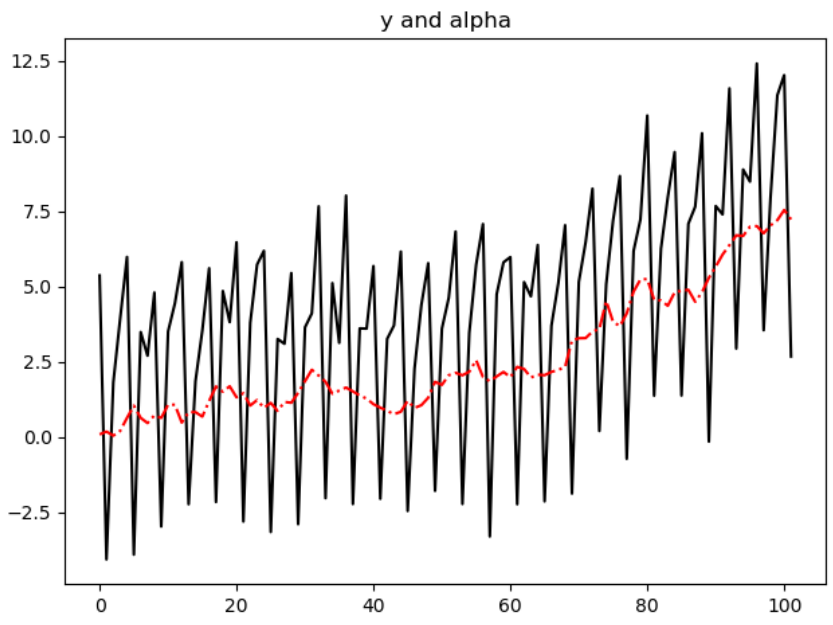
Seasonality

Sometimes data shown a dynamic behavior that tends to repeat from time to time depending on the specific time frequency.
In this case, the time series are affected by a seasonal component representing a non-negligiable feature of the time series dynamics. In this section we introduce two simple procedures that can be used in order to "clean" the series from this seasonal component. We first introduce the additive case and then the multiplicative case below.

Additive seasonality

Suppose we have a quarterly series, that is a series observed 4 times per year, like this:

```python
import numpy as np
import matplotlib.pylab as plt
import math

np.random.seed(1213)
n = 102
e = np.sqrt(0.5)* np.random.randn(n)
u = np.sqrt(0.1)* np.random.randn(n)
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
factor = [5, -4, 2,3]
s = 4
seasonal = (factor*(math.ceil(n/s)))[0:n]
y[0] = e[0] + seasonal[0]
alpha[0] = u[0]
for t in range(1,n):
    y[t] = seasonal[t] + alpha[t-1] + e[t]
    alpha[t] = alpha[t-1] + u[t]
plt.plot(y,'k-')
plt.plot(alpha,'r-.')
plt.title("y and alpha")
plt.show()
```



One way to threat this series is to remove the seasonal component by using the so called moving average appraoch. These are the steps:

> Take the centered moving average of the series, call it CMA_t
>
> Subtract the CMA from the original series $residuals_t = y_t - CMA_t$
>
> Average the elements of $residuals_t$ by season and obtain the seasonal factor
>
> Subtract the element of y_t by the corresponding seasonal factor

Step one requires to create a series of average values centered at time t. Define s the number of seasons considered. In the quarterly case we have 4 values. If we want to create a centered moving avearege we need to select values at time t , $t \pm 1$, $\cdots \pm \frac{s}{2}$. In the quarterly case we need to take 5 values t , $t \pm 1$, $t \pm 2$, but we want a moving average of 4 elements! So how to create this average? In the quarterly case for example we can take t , $t \pm 1$, $\frac{t \pm 2}{2}$, that is half of the extreme values $\pm\frac{t \pm 2}{2}$. In other terms we want

$$\frac{0.5 * x_{t-2} + x_{t-1} + x_t + x_{t+1} + 0.5 * x_{t+2}}{4}$$

The next code run an example of the procedure to deseasonalise a short series. The code is not smart but makes the trick:

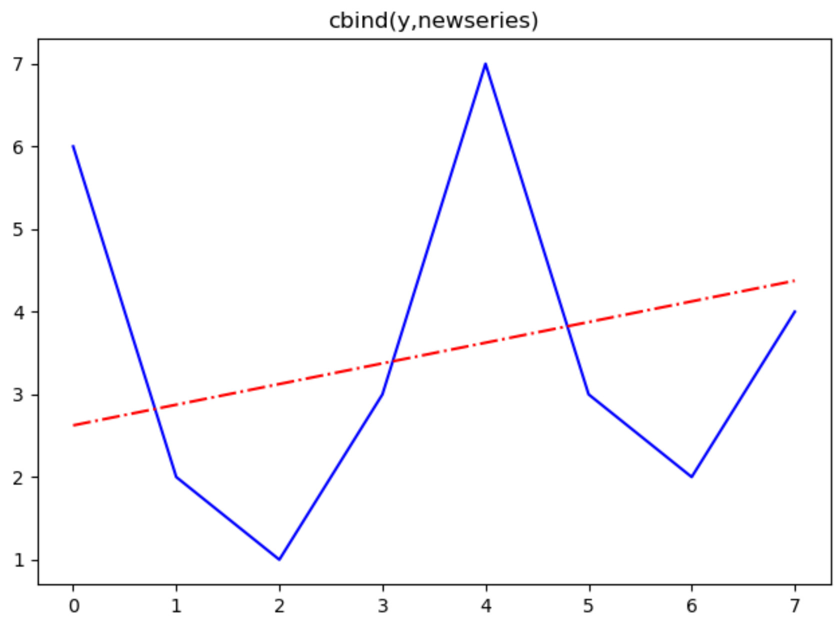
```
import pandas as pd

y = np.asarray([6,2,1,3,7,3,2,4])
cma = np.zeros(len(y))
residuals = np.zeros(len(y))
print(residuals)
cma[2] = (0.5*y[0]+y[1]+y[2]+y[3]+0.5*y[4])/4
cma[3] = (0.5*y[1]+y[2]+y[3]+y[4]+0.5*y[5])/4
cma[4] = (0.5*y[2]+y[3]+y[4]+y[5]+0.5*y[6])/4
cma[5] = (0.5*y[3]+y[4]+y[5]+y[6]+0.5*y[7])/4
for i in range(2,6):
    residuals[i] = y[i] - cma[i]
index = [i for i in range(len(y))]
y = np.array(y)
cma = np.asarray(cma)
result = pd.DataFrame({'y':y,'cma':cma,'residuals':residuals},index= index)
```

```
print(result)
```

Running Result:

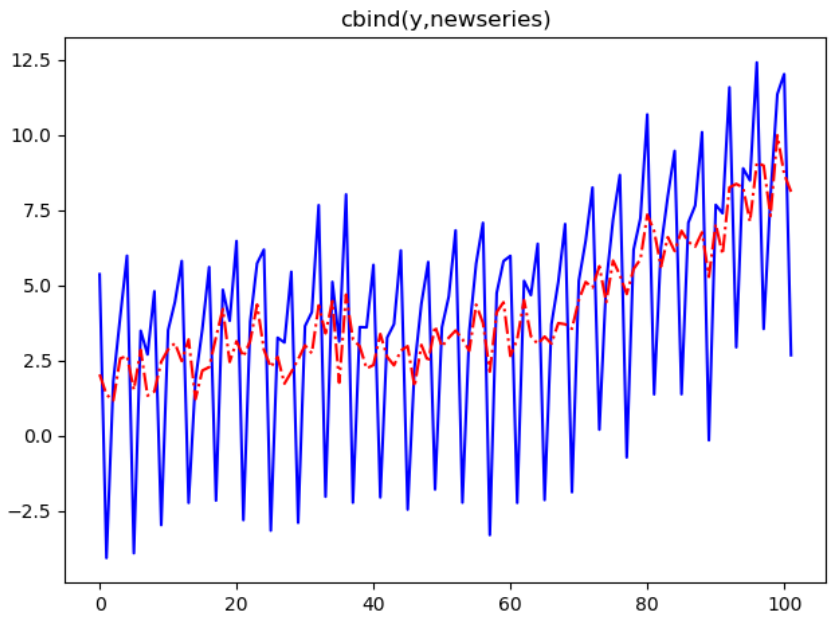
```
   y    cma  residuals
0  6  0.000      0.000
1  2  0.000      0.000
2  1  3.125     -2.125
3  3  3.375     -0.375
4  7  3.625      3.375
5  3  3.875     -0.875
6  2  0.000      0.000
7  4  0.000      0.000
```

```
newseries = [0*i for i in range(len(y))]
factors = [residuals[4]-residuals[0],residuals[5]-residuals[1],residuals[2]-
residuals[6],residuals[3]-residuals[7]]
factors = factors*2
for i in range(len(y)):
    newseries[i] = y[i] - factors[i]
plt.plot(y,'b-')
plt.plot(newseries,'r-.')
plt.title("cbind(y,newseries)")
plt.show()
```



The code makes what we want but it is not elegant and need to be generalized! Lets create a smarter code as below:

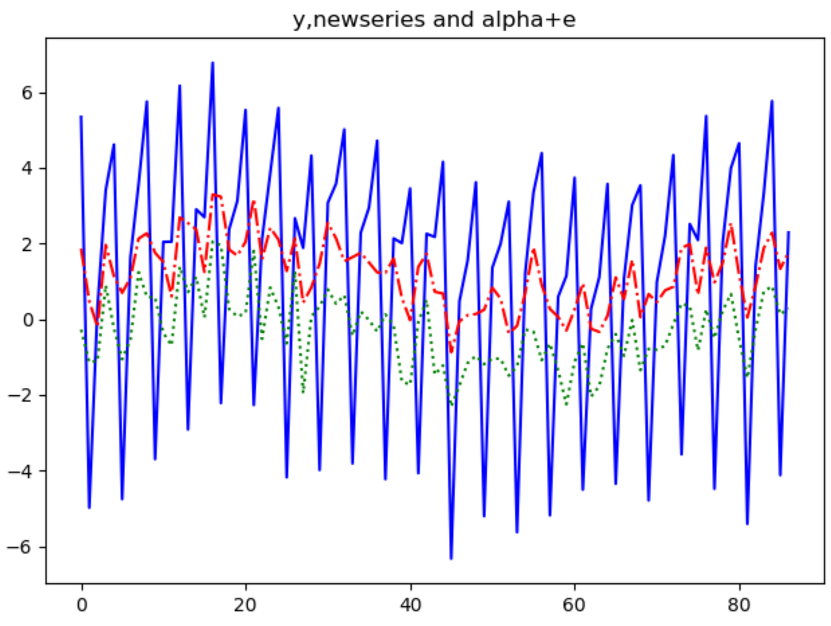
```
cma = np.zeros(len(y))
for g in range(0,(len(y)-s)):
    sum = 0
    for h in range(g,g+s+1):
        sum = sum + w[h-g]* y[h]
    cma[int(g+s/2)] = sum

residuals = [0*i for i in range(len(cma))]
for i in range(2,len(cma)-2):
    residuals[i] = y[i] - cma[i]

factors = [0*i for i in range(s)]
index0 = np.arange(0,len(y),4)
index1 = np.arange(1,len(y),4)
index2 = np.arange(2,len(y),4)
index3 = np.arange(3,len(y),4)
sum0 = 0
index =
np.arange(0,len(y),4),np.arange(1,len(y),4),np.arange(2,len(y),4),np.arange(3,le
n(y),4)
sum = [0*i for i in range(s)]
factors = [0*i for i in range(s)]

for seas in range(0,s):
    for i in index[seas]:
        sum[seas] = sum[seas] + residuals[i]

factors[0] = sum[0]/(len(index[0])-1)
factors[1] = sum[1]/(len(index[1])-2)
factors[2] = sum[2]/(len(index[2])-1)
factors[3] = sum[3]/(len(index[3]))

factors = (factors*math.ceil(n/s))[0:n]
newseries = [0*i for i in range(len(y))]
for i in range(len(y)):
    newseries[i] = y[i] - factors[i]
plt.plot(y,'b-')
plt.plot(newseries,'r-.')
plt.title("cbind(y,newseries)")
plt.show()
```



Lets generate a series and we then implement the process of taking out the seasonal component:

```
import numpy as np
import math
import matplotlib.pylab as plt
```

```
np.random.seed(243)
n = 87
e = np.sqrt(0.3)* np.random.randn(n)
u = np.sqrt(0.1)* np.random.randn(n)
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
factor = [5, -4, 2,3]
s = 4
seasonal = (factor*(math.ceil(n/s)))[0:n]

y[0] = e[0] + seasonal[0]
alpha[0] = u[0]
for t in range(1,n):
    y[t] = seasonal[t] + alpha[t-1] + e[t]
    alpha[t] = alpha[t-1] + u[t]

w = [1/(2*s)]
w = w*(s+1)
for i in range(1,s):
    w[i] = 1/s


cma = np.zeros(len(y))
for g in range(0,(len(y)-s)):
    sum = 0
    for h in range(g,g+s+1):
        sum = sum + w[h-g]* y[h]
    cma[int(g+s/2)] = sum

residuals = [0*i for i in range(len(cma))]
for i in range(2,len(cma)-2):
    residuals[i] = y[i] - cma[i]

factors = [0*i for i in range(s)]
index0 = np.arange(0,len(y),4)
index1 = np.arange(1,len(y),4)
index2 = np.arange(2,len(y),4)
index3 = np.arange(3,len(y),4)
sum0 = 0
index =
np.arange(0,len(y),4),np.arange(1,len(y),4),np.arange(2,len(y),4),np.arange(3,le
n(y),4)
sum = [0*i for i in range(s)]
factors = [0*i for i in range(s)]

for seas in range(0,s):
    for i in index[seas]:
        sum[seas] = sum[seas] + residuals[i]

factors[0] = sum[0]/(len(index[0])-1)
factors[1] = sum[1]/(len(index[1])-2)
factors[2] = sum[2]/(len(index[2])-1)
factors[3] = sum[3]/(len(index[3]))

factors = (factors*math.ceil(n/s))[0:n]
newseries = [0*i for i in range(len(y))]
for i in range(len(y)):
    newseries[i] = y[i] - factors[i]
plt.plot(y,'b-')
plt.plot(newseries,'r-.')
plt.plot(alpha+e,'g:')
plt.title("y,newseries and alpha+e")
plt.show()

print(factor)
print(factors)
```



Running Result:

```
factor [5, -4, 2, 3]
factors [3.480871889587916, -5.453033030787276, 0.53208272649184,
1.4464002333253796]
```

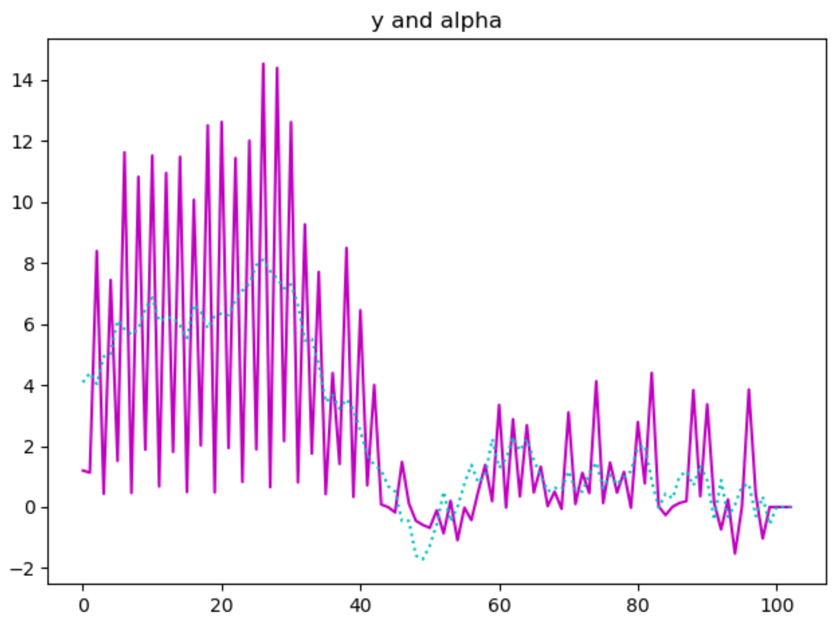
So this procedure allows to extract the seasonal component from a series. Note that a similar procedure can be run for the case where the seasonal component is multiplicative rather than additive. Below we show the procedure.

Multiplicative seasonality

Suppose we have a quarterly series, that is a series observed 4 times per year, like this:

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

np.random.seed(7)
n = 103
e = np.sqrt(0.5)* np.random.randn(n)
u = np.sqrt(0.4)* np.random.randn(n)
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
factor = [1.7, 0.3, 1.9, 0.1]
seasonal = (factor*(int(100/4)))

y[0] = e[0]
```

```
alpha[0] = 5 + u[0]
for t in range(1,len(seasonal)):
    y[t] = seasonal[t] * (alpha[t-1] + e[t])
    alpha[t] = alpha[t-1] + u[t]
plt.plot(y,'m-')
plt.plot(alpha,'c:')
plt.title("y and alpha")
plt.show()
```



The seasonal component is evident but is different from the one shown above. In this case, it tends to amplify when the series increases. Indeed, the level α_t is multiplied (not added) to the factor.

Here we can remove the seasonal component by using again the moving average appraoch but in a different way. These are the steps:

> Take the centered moving average of the series, call it CMA_t
>
> Divide the CMA from the original series $residuals_t = \frac{y_t}{CMA_t}$
>
> Average the elements of $residuals_t$ by season and obtain the seasonal factor
>
> Divide the element of y_t by the corresponding seasonal factor

This is the code (similar to the one above) that makes the trick:

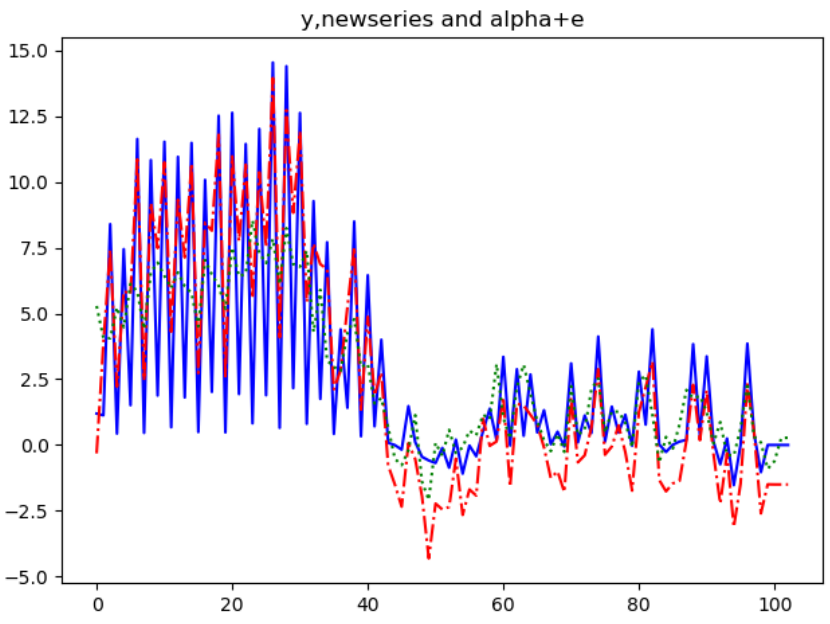
```
s = 4
n = len(y)
w = [1/(2*s)]
w = w*(s+1)
for i in range(1,s):
    w[i] = 1/s

cma = np.zeros(len(y))
for g in range(0,(len(y)-s)):
    sum = 0
    for h in range(g,g+s+1):
        sum = sum + w[h-g]* y[h]
    cma[int(g+s/2)] = sum
```

```
residuals = [0*i for i in range(len(cma))]
for i in range(2,len(cma)-2):
    residuals[i] = y[i] / cma[i]

factors = [0*i for i in range(s)]
index0 = np.arange(0,len(y),4)
index1 = np.arange(1,len(y),4)
index2 = np.arange(2,len(y),4)
index3 = np.arange(3,len(y),4)
sum0 = 0
index = 
np.arange(0,len(y),4),np.arange(1,len(y),4),np.arange(2,len(y),4),np.arange(3,le
n(y),4)
sum = [0*i for i in range(s)]
factors = [0*i for i in range(s)]

for seas in range(0,s):
    for i in index[seas]:
        sum[seas] = sum[seas] + residuals[i]

factors[0] = sum[0]/(len(index[0])-1)
factors[1] = sum[1]/(len(index[1])-2)
factors[2] = sum[2]/(len(index[2])-1)
factors[3] = sum[3]/(len(index[3]))

factors = (factors*math.ceil(n/s))[0:n]
newseries = [0*i for i in range(len(y))]
for i in range(len(y)):
    newseries[i] = y[i] / factors[i]
```

Let see how it works with the (multiplicative) seasonal series generate above:

```
import numpy as np
import math
import matplotlib.pylab as plt

np.random.seed(7)
n = 103
e = np.sqrt(0.5)* np.random.randn(n)
u = np.sqrt(0.4)* np.random.randn(n)
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
factor = [1.7, 0.3, 1.9, 0.1]
seasonal = (factor*(int(100/4)))

y[0] = e[0]
alpha[0] = 5 + u[0]
for t in range(1,len(seasonal)):
    y[t] = seasonal[t] * (alpha[t-1] + e[t])
    alpha[t] = alpha[t-1] + u[t]
plt.plot(y,'m-')
plt.plot(alpha,'c:')
plt.title("y and alpha")
plt.show()



s = 4
```

```
n = len(y)
w = [1/(2*s)]
w = w*(s+1)
for i in range(1,s):
    w[i] = 1/s

cma = np.zeros(len(y))
for g in range(0,(len(y)-s)):
    sum = 0
    for h in range(g,g+s+1):
        sum = sum + w[h-g]* y[h]
    cma[int(g+s/2)] = sum

residuals = [0*i for i in range(len(cma))]
for i in range(2,len(cma)-2):
    residuals[i] = y[i] / cma[i]

factors = [0*i for i in range(s)]
index0 = np.arange(0,len(y),4)
index1 = np.arange(1,len(y),4)
index2 = np.arange(2,len(y),4)
index3 = np.arange(3,len(y),4)
sum0 = 0
index = 
np.arange(0,len(y),4),np.arange(1,len(y),4),np.arange(2,len(y),4),np.arange(3,le
n(y),4)
sum = [0*i for i in range(s)]
factors = [0*i for i in range(s)]

for seas in range(0,s):
    for i in index[seas]:
        sum[seas] = sum[seas] + residuals[i]

factors[0] = sum[0]/(len(index[0])-1)
factors[1] = sum[1]/(len(index[1])-2)
factors[2] = sum[2]/(len(index[2])-1)
factors[3] = sum[3]/(len(index[3]))

print('factor',factor)
print('factors',factors)

factors = (factors*math.ceil(n/s))[0:n]
newseries = [0*i for i in range(len(y))]
for i in range(len(y)):
    newseries[i] = y[i] / factors[i]

plt.plot(y,'b-')
plt.plot(newseries,'r-.')
plt.plot(alpha+e,'g:')
plt.title("y,newseries and alpha+e")
plt.show()
```



Running Reslut:

```
factor [1.7, 0.3, 1.9, 0.1]
factors [1.012515472821411, 0.20922073614203296, 0.9413362751357984,
0.11620040927352396]
```

We can see that the difference is rather small.

Seasonal-state-Space representation

Another way to treat the seasonality is to modify the state-space model by taking into consideration the seasonal component.
For example, consider the Local level model in the SSOE framework with a seasonal behavior as follows:

$$y_t = \alpha_{t-s} + e_t$$
$$\alpha_t = \alpha_{t-s} + \gamma e_t$$

Where s represents the frequency of the data considered (e.g. monthly, quarterly, weekly, etc.). For example, suppose we observe a quarterly time series following a specific dynamics (say the local level). Moreover, we also observe that, every specific quarter, the time series tends to assume a value being similar to the one of the same quarter last year. We can represent this model as follows:

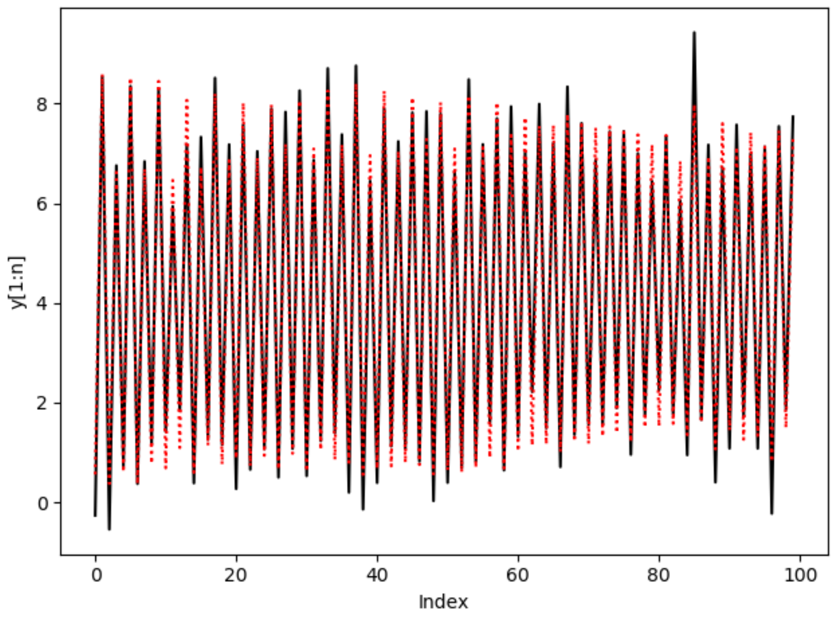
$$y_t = \alpha_{t-4} + e_t$$
$$\alpha_t = \alpha_{t-4} + \gamma e_t$$

For example suppose we want to generate the following quarterly local level model:

where $e \sim Normal(\mu = 0; \sigma_e^2 = .4)$ and $gamma = .3$ where $n = 100$.

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

np.random.seed(55)
n = 100
```

```
e = np.sqrt(.4) * np.random.randn(n)
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
s = 4
sfactor = np.array(10 * np.random.rand(s))
y[0] = sfactor[0] + e[0]
y[1] = sfactor[1] + e[1]
y[2] = sfactor[2] + e[2]
y[3] = sfactor[3] + e[3]
alpha[0] = sfactor[0] + 0.2 * e[0]
alpha[1] = sfactor[1] + 0.2 * e[1]
alpha[2] = sfactor[2] + 0.2 * e[2]
alpha[3] = sfactor[3] + 0.2 * e[3]

for t in range(4,n):
    alpha[t] = alpha[t - s] + 0.3 * e[t]
    y[t] = alpha[t-s]+e[t]

plt.plot(y[0:n],'b-')
plt.plot(alpha[0:n],'r:')
plt.xlabel("Index")
plt.ylabel("y[1:n]")
plt.show()
```



The good news is that the same results as shown above apply since we are assuming that the same factors appear every quarter. Therefore we can estimate the model as follows:

```
import numpy.matlib
from scipy.optimize import minimize

s = 4
state = np.matlib.zeros((n,1))
e = np.matlib.zeros((n,1))

a = [0*i for i in range(n)]
a[0] = y[0]
```

```
def logLikConc0(x):
    e2 = 0
    for t in range(s, n):
        e[t] = y[t] - state[t-s]
        e2 = e2 + e[t]**2
        state[t] = state[t-s] + x[0] * e[t]
    return (e2-e[0])/(n-1)

def logLikConc():
    v = lambda x: logLikConc0(x)
    return v

def con(args):
    bmin, bmax = args
    cons = ({'type': 'ineq', 'fun': lambda b: b - bmin},{'type': 'ineq', 'fun':
lambda b: -b + bmax})
    return cons

x0 = np.asarray((0.4))
args1 = (0,1)
cons = con(args1)
res = minimize(logLikConc(), x0, method='SLSQP',constraints=cons)
print(res)
print("this is the estimated gamma",res.x)
print("this is the estimated variance of e",res.fun)
```

Running Result:

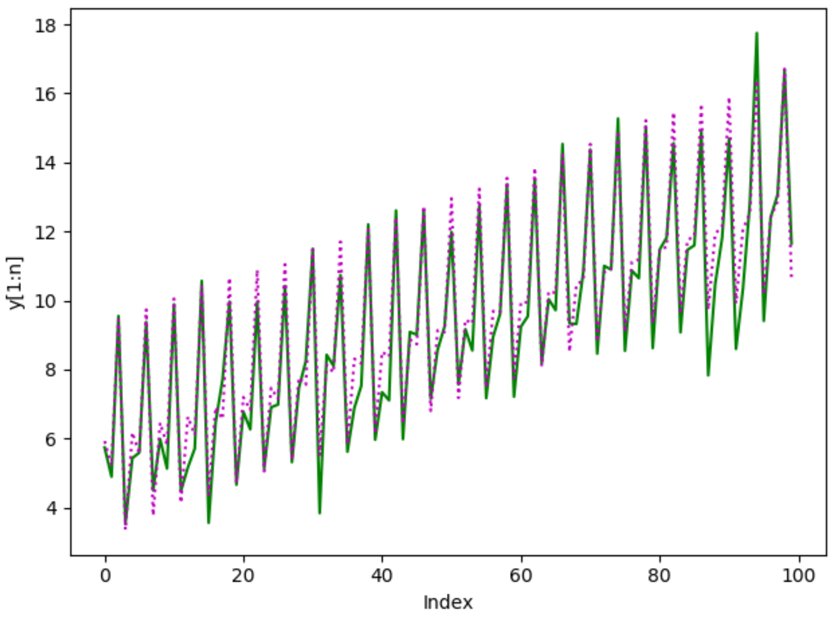
```
fun: 1.7552394098134867
     jac: array([0.0016399])
 message: 'Optimization terminated successfully.'
    nfev: 15
     nit: 5
    njev: 5
  status: 0
 success: True
       x: array([0.78777957])
this is the estimated gamma [0.78777957]
this is the estimated variance of e 1.7552394098134867
```

Where 0.78777957 is the estimated signal-to-noise ratio.

Suppose we want to generate a local level with drift (the model underlying the Theta method).

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

np.random.seed(1132)
n = 100
e = np.sqrt(.4) * np.random.randn(n)
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
s = 4
co = 0.3
sfactor = np.array(10 * np.random.rand(s))
y[0] = sfactor[0] + e[0]
y[1] = sfactor[1] + e[1]
```

```
y[2] = sfactor[2] + e[2]
y[3] = sfactor[3] + e[3]
alpha[0] = sfactor[0] + 0.2 * e[0]
alpha[1] = sfactor[1] + 0.2 * e[1]
alpha[2] = sfactor[2] + 0.2 * e[2]
alpha[3] = sfactor[3] + 0.2 * e[3]

for t in range(4,n):
    alpha[t] = co + alpha[t - s] + 0.1 * e[t]
    y[t] = alpha[t-s]+e[t]

plt.plot(y[0:n],'g-')
plt.plot(alpha[0:n],'m:')
plt.xlabel("Index")
plt.ylabel("y[1:n]")
plt.show()
```



We can now estimate the model as follows:

```
import numpy.matlib
from scipy.optimize import minimize

s = 4
v = np.matlib.zeros((n,1))
state = np.matlib.zeros((n,1))
for i in range(s):
    state[i] = y[i]

def logLikConc0(x):
    e2 = 0
    for t in range(s, n):
        v[t] = y[t] - state[t-s]
        e2 = e2 + e[t]**2
        state[t] = x[1] + state[t-s] + x[0] * v[t]
    return (e2-e[0])/(n-1)

def logLikConc():
    v = lambda x: logLikConc0(x)
    return v
```

```
x0 = np.asarray((0.2,0.2))
res = minimize(logLikConc(), x0, method='SLSQP')
print(res)
```

Running Result:

```
fun: 0.4538870155306315
     jac: array([0., 0.])
 message: 'Optimization terminated successfully.'
    nfev: 4
     nit: 1
    njev: 1
  status: 0
 success: True
       x: array([0.2, 0.2])
```

Forecasting time series

We have been discussing the estimation of state-space models but we did not discuss a crucial issue: how can we forecast time series?

One of the most important use of state-space model is to predict time series outside the estimation sample.

A crucial role is played by the state variable α_t. Indeed, the state variable (the unobserved component) is the key to predict our data h-steps ahead.

Consider again model (1) and suppose we want to forecast the y_t variable $1, 2, \cdots, h$-steps ahead. Then we have:

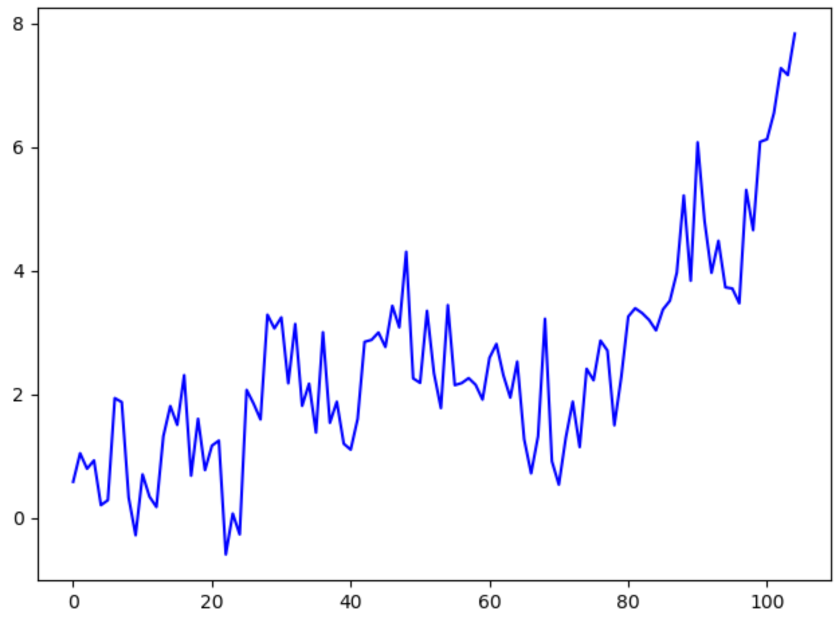
$$\begin{gathered}
\hat{y}_{t+1} = za_t = z(c + wa_{t-1} + kv_t) \\
\hat{y}_{t+2} = z * a_{t+1} = z * (c + w * a_t) = z * c + w * \hat{y}_{t+1} \\
\hat{y}_{t+3} = z * a_{t+2} = z * (c + w * a_{t+1}) = z * c + w * \hat{y}_{t+2} \\
\hat{y}_{t+4} = z * a_{t+3} = z * (c + w * a_{t+2}) = z * c + w * \hat{y}_{t+3} \\
\vdots \\
\hat{y}_{t+h} = z * c + w * \hat{y}_{t+h-1}
\end{gathered}$$

Note indeed that we know v_t but we do not know v_{t+1} as well as v_{t+2} etc.

For example consider the following Theta method:

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

np.random.seed(1347)
n = 105
e = np.sqrt(.5) * np.random.randn(n)
u = np.sqrt(.1) * np.random.randn(n)
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
co = 0.06
y[0] = e[0]
alpha[0] = u[0]
```

```
for t in range(1,n):
    alpha[t] = co + alpha[t-1] + u[t]
    y[t] = alpha[t-1] + e[t]
plt.plot(y,'b-')
plt.show()
```



Suppose we know the first 100 observations and we do not know the last 5 observations. If we want to forecast them we need first to estimate the parameters and then run the forecasts.

A simple code is the following:

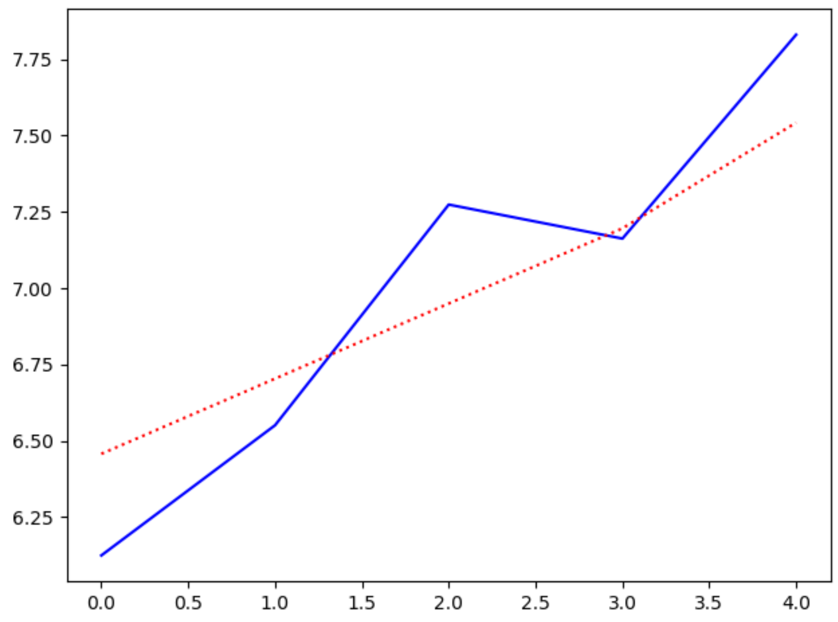
```
import numpy as np
import numpy.matlib
from scipy.optimize import minimize

obs = 100
xxx = [0*i for i in range(obs)]
for i in range(obs):
    xxx[i] = y[i]

a = [0*i for i in range(obs)]
p = [0*i for i in range(obs)]
a[0] = xxx[0]
p[0] = 10000
k = [0*i for i in range(obs)]
v = [0*i for i in range(obs)]

def funcTheta0(x):
    z = w = 1
    likelihood = sigmae = 0
    for t in range(1,obs):
        k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + x[0]
        v[t] = xxx[t] - z * a[t - 1]
        a[t] = x[1] + w * a[t - 1] + k[t] * v[t]
        sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))
        likelihood = likelihood + 0.5 * np.log(2 * np.pi) + 0.5 + 0.5 * np.log(z
** 2 * p[t - 1] + 1)
    likelihood = likelihood + 0.5 * n * np.log(sigmae / n)
    return likelihood

def funcTheta():
    v = lambda x: funcTheta0(x)
    return v

x0 = np.asarray((0.6,0.2))
res = minimize(funcTheta(), x0, method='SLSQP')
print(res)
print(res.x)

q = abs(res.x[0])
co = abs(res.x[1])
z = w = 1
sigmae = 0

for t in range(1,obs):
    k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
    p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + q
    v[t] = y[t] - z * a[t - 1]
    a[t] = co + w * a[t - 1] + k[t] * v[t]
    sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t] + 1))
print("co",co)
print("the variance of e",sigmae/(obs-1))
print("the variance of u",q*(sigmae/(obs-1)))
```

Running Result:

```
     fun: 121.07312427126719
     jac: array([ 0.00264263, -0.0026474 ])
 message: 'Optimization terminated successfully.'
    nfev: 32
     nit: 6
    njev: 6
  status: 0
 success: True
       x: array([0.56546551, 0.04629436])
[0.56546551 0.04629436]
co 0.04629435918816891
the variance of e 0.3380830359493047
the variance of u 0.19117429770680539
```

Now we can forecast x 5-steps ahead as follows:

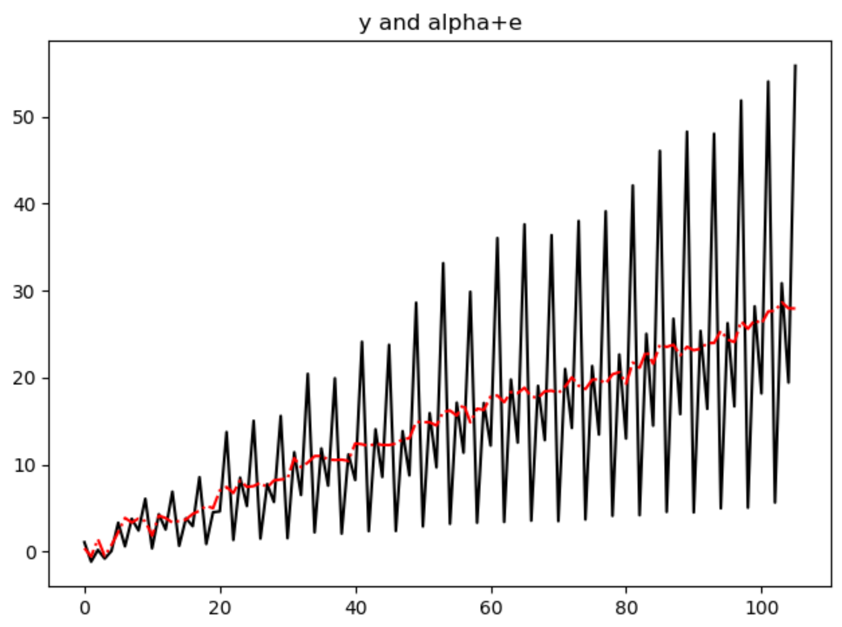
```
MyForecasts = [0*i for i in range(5)]
MyForecasts[0] = a[obs-1]
MyForecasts[1] = co + MyForecasts[0]
MyForecasts[2] = co + MyForecasts[1]
MyForecasts[3] = co + MyForecasts[2]
MyForecasts[4] = co + MyForecasts[3]
plt.plot(y[100:105],'b-')
plt.plot(MyForecasts,'r:')
plt.show()
```



Forecasting seasonal series

Now consider the case of series affected by a seasonal factor. Suppose we wish to forecast the last 6 observations of this series (considering that we know only the first 100 observations):

```
import numpy as np
import matplotlib.pylab as plt
import math

np.random.seed(115)
n = 106
e = np.sqrt(0.4)* np.random.randn(n)
u = np.sqrt(0.1)* np.random.randn(n)
y = [0*i for i in range(n)]
alpha = [0*i for i in range(n)]
con = 0.2
factor = [0.7, 2, 0.2 ,1.1]
seasonal = (factor*(math.ceil(n/4)))[0:n]
y[0] = e[0] + seasonal[0]
alpha[0] = u[0]
for t in range(1,n):
    y[t] = seasonal[t] *(alpha[t-1] + e[t])
    alpha[t] = con + alpha[t-1] + u[t]
plt.plot(y,'k-')
plt.plot(alpha+e,'r-.')
plt.title("y and alpha+e")
plt.show()
```



For this type of series the forecasting procedure goes like this: (1) Apply the additive or multiplicative seasonal adjustment procedure discussed above (2) forecast the deseasonalised series with the method you wish (3) add or multiply the seasonal factors to the forecast values obtained.

Remember that the additive or multiplicative procedure depends upon the type of seasonality we observe. If the seasonal component is amplified when the level of the series increase we use the multiplicative procedure. On the other hand, if the seasonal component is constant and is not amplified when the series increases then use the additive procedure.

In this example we suppose we know the first 100 observations and we leave the other 6 as forecasting sample.

```
x = y[0:100]
s = 4
n = len(x)
w = [1/(2*s)]
w = w*(s+1)
for i in range(1,s):
    w[i] = 1/s

cma = np.zeros(len(y))
for g in range(0,(len(y)-s)):
    sum = 0
    for h in range(g,g+s+1):
        sum = sum + w[h-g]* y[h]
    cma[int(g+s/2)] = sum

residuals = [0*i for i in range(len(cma))]
for i in range(2,len(cma)-2):
    residuals[i] = y[i] / cma[i]

factors = [0*i for i in range(s)]
index0 = np.arange(0,len(y),4)
index1 = np.arange(1,len(y),4)
index2 = np.arange(2,len(y),4)
index3 = np.arange(3,len(y),4)
sum0 = 0
```

```
index =
np.arange(0,len(y),4),np.arange(1,len(y),4),np.arange(2,len(y),4),np.arange(3,le
n(y),4)
sum = [0*i for i in range(s)]
factors = [0*i for i in range(s)]

for seas in range(0,s):
    for i in index[seas]:
        sum[seas] = sum[seas] + residuals[i]

factors[0] = sum[0]/(len(index[0])-1)
factors[1] = sum[1]/(len(index[1])-2)
factors[2] = sum[2]/(len(index[2])-1)
factors[3] = sum[3]/(len(index[3]))
print('factor',factor)
print('factors',factors)
factors = (factors*math.ceil(n/s))[0:n]
newseries = [0*i for i in range(len(y))]
for i in range(len(y)):
    newseries[i] = y[i] / factors[i]
```

We can now forecast the newly deseasonalised series and then multiply the forecast by the factors to obtain the seasonal forecasts. The code below makes the trick:

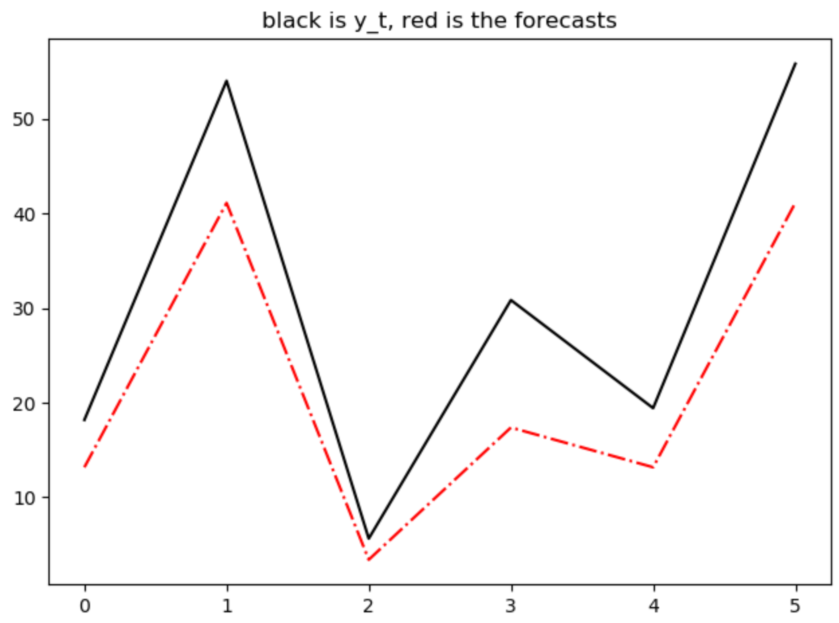
```
x = y[0:100]
s = 4
n = len(x)
w = [1/(2*s)]
w = w*(s+1)
for i in range(1,s):
    w[i] = 1/s

cma = np.zeros(len(y))
for g in range(0,(len(y)-s)):
    sum = 0
    for h in range(g,g+s+1):
        sum = sum + w[h-g]* y[h]
    cma[int(g+s/2)] = sum

residuals = [0*i for i in range(len(cma))]
for i in range(2,len(cma)-2):
    residuals[i] = y[i] / cma[i]

factors = [0*i for i in range(s)]
index0 = np.arange(0,len(y),4)
index1 = np.arange(1,len(y),4)
index2 = np.arange(2,len(y),4)
index3 = np.arange(3,len(y),4)
sum0 = 0
index =
np.arange(0,len(y),4),np.arange(1,len(y),4),np.arange(2,len(y),4),np.arange(3,le
n(y),4)
sum = [0*i for i in range(s)]
factors = [0*i for i in range(s)]

for seas in range(0,s):
    for i in index[seas]:
        sum[seas] = sum[seas] + residuals[i]

factors[0] = sum[0]/(len(index[0])-1)
factors[1] = sum[1]/(len(index[1])-2)
factors[2] = sum[2]/(len(index[2])-1)
factors[3] = sum[3]/(len(index[3]))
print('factor',factor)
print('factors',factors)
factors = (factors*math.ceil(n/s))[0:n]
newseries = [0*i for i in range(len(y))]
# print('000',len(y))
# print('000',len(factors))
for i in range(len(factors)):
    newseries[i] = y[i] / factors[i]

a = [0*i for i in range(len(newseries))]
p = [0*i for i in range(len(newseries))]
a[0] = newseries[0]
p[0] = 10000
k = [0*i for i in range(len(newseries))]
v = [0*i for i in range(len(newseries))]
v[0] = 0

from scipy.optimize import minimize

def funcTheta0(x):
    z = w = 1
    likelihood = sigmae = 0
    for t in range(1,len(newseries)):
        k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + x[0]
        v[t] = newseries[t] - z * a[t - 1]
        a[t] = x[1] + w * a[t - 1] + k[t] * v[t]
        sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))
        likelihood = likelihood + 0.5 * np.log(2 * np.pi) + 0.5 + 0.5 * np.log(z
** 2 * p[t - 1] + 1)
    likelihood = likelihood + 0.5 * n * np.log(sigmae / len(newseries))
    return likelihood

def funcTheta():
    v = lambda x: funcTheta0(x)
    return v

x0 = np.asarray((0.6,0.2))
res = minimize(funcTheta(), x0, method='SLSQP')
print(res)
print(res.x)

q = abs(res.x[0])
co = abs(res.x[1])
z = w = 1
sigmae = 0

for t in range(1,len(newseries)):
    k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
    p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + q
    v[t] = y[t] - z * a[t - 1]
    a[t] = co + w * a[t - 1] + k[t] * v[t]
    sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t] + 1))
print("co",co)
print("the variance of e",sigmae/(len(newseries)-1))
print("the variance of u",q*(sigmae/(len(newseries)-1)))
```

Running Result:

```
factor [0.7, 2, 0.2, 1.1]
factors [0.6492544542011102, 2.025257469568655, 0.1682601968158358,
0.8561348696297972]

     fun: 296.32236934884367
     jac: array([-0.00026321, -0.00117493])
 message: 'Optimization terminated successfully.'
    nfev: 35
     nit: 8
    njev: 8
  status: 0
 success: True
       x: array([1.30442755, 0.00160535])
[1.30442755 0.00160535]
co 0.0016053508093791138
the variance of e 84.10431754051116
the variance of u 109.7079889920933
```

Now we can forecast x 6-steps ahead and then multiply the factors to the out-of-sample forecasts, as follows:

```
w = z = 1
Myforecasts = [0*i for i in range(6)]
Myforecasts[0] = a[len(x)]
for o in range(1,6):
    Myforecasts[o] = co + Myforecasts[o-1]
print(np.array(Myforecasts))
print(factors)
SeasonalForcast = np.array(Myforecasts)* np.array(factors[0:6])
print(SeasonalForcast)
plt.plot(y[100:106],'k-')
plt.plot(SeasonalForcast,'r-.')
plt.title("black is y_t, red is the forecasts")
plt.show()
```



Unsurprisingly the 6 steps forecasts fit very well the series in black.

Comparing forecasting performance

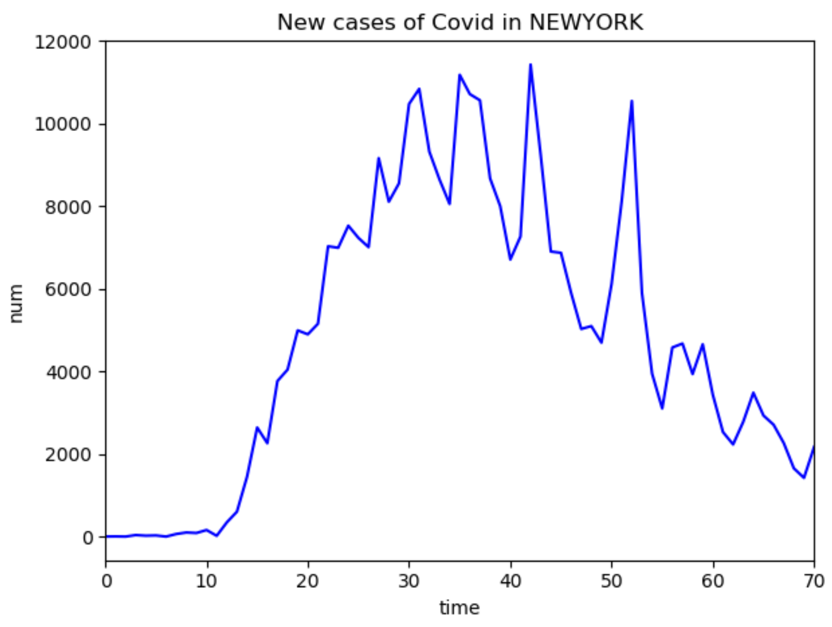
Suppose we aim to compare two models when forecasting a specific time series. An important issue is to decide a metric that allow us to establish which model provide better forecasts for the series. The forecasting literature has long discussed the forecast evaluation metrics. Define $y_{t+1}\ y_{t+2}\ \cdots\ y_{t+h}$ the future values of a series (that we do not know) and $\hat{y}_{t+1}\ \hat{y}_{t+2}\ \cdots\ \hat{y}_{t+h}$ the forecasts provided by a specific model. Two popular evaluation metrics are the so called Mean-Absolute-Scaled-Error (MASE) and the Mean-Absolute-Percentage-Error (MAPE) that can be calculated as follows:

$$MASE = \frac{1}{h}\frac{\sum_{t=n+1}^{n+h}|y_t-\hat{y}_t|}{\frac{1}{n-m}\sum_{t=m+1}^{n}|y_t-y_{t-m}|}$$

$$MAPE = \frac{2}{h}\sum_{t=n+1}^{n+h}\frac{|y_t-\hat{y}_t|}{|y_t|+|\hat{y}_t|}\times 100$$

Let's make an example. Suppose we want to forecast the number of $newcases$ of Coronavirus (Covid19) in the New York, USA .

```
import pandas as pd
import matplotlib.pylab as plt

data = pd.read_excel('CoronavirusSpreadUSAregions.xlsx')
y = data["NEWYORK"]
plt.xlabel('time')
plt.ylabel('num')
plt.title('New cases of Covid in NEWYORK')
y.plot(color='b',title='New cases of Covid in NEWYORK')
plt.show()
```



Here I estimate the Theta method using the first observations and leaving the last 5 observations to be forecasted:

```
obs = len(y)-5
xxx = y[0:obs]
a = [i for i in range(obs)]
p = [i for i in range(obs)]
a[0] = xxx[0]
p[0] = 10000
k = [i for i in range(obs)]
v = [i for i in range(obs)]
v[0] = 0
n = len(xxx)
newseries = y

def myfunTheta0(x):
    z = w = 1
    likelihood = 0
    sigmae = 0
    for t in range(1, obs):
        k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + x[0]
        v[t] = xxx[t] - z * a[t - 1]
        a[t] = x[1] + w * a[t - 1] + k[t] * v[t]
        sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))
        likelihood = likelihood + 0.5 * np.log(2 * np.pi) + 0.5 + 0.5 * np.log(z
** 2 * p[t - 1] + 1)
    likelihood = likelihood + 0.5 * n * np.log(sigmae / len(newseries))
    return likelihood

def myfuTheta():
    v = lambda x: myfunTheta0(x)
    return v

x0 = np.asarray((0.6,0.2))
res = minimize(myfuTheta(), x0, method='SLSQP')

q = abs(res.x[0])
co = abs(res.x[1])
```

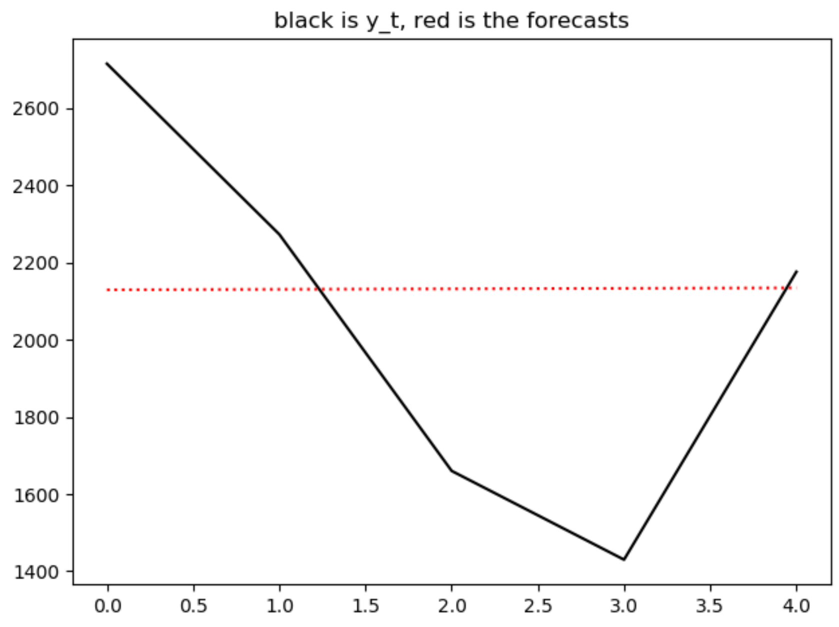
```
z = w = 1
sigmae = 0

for t in range(1,obs):
    k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
    p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + q
    v[t] = y[t] - z * a[t - 1]
    a[t] = co + w * a[t - 1] + k[t] * v[t]
    sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))

print("co",co)
print("the variance of e",sigmae/(len(newseries)-1))
print("the variance of u",q*(sigmae/(len(newseries)-1)))
```

Running Result:

```
co 1.25611179735503
the variance of e 60108.16222820192064
the variance of u 18354.3831263978
```

Here I forecast:

```
MyForecasts = [0*i for i in range(5)]
MyForecasts[0] = a[obs-1]
MyForecasts[1] = co + MyForecasts[0]
MyForecasts[2] = co + MyForecasts[1]
MyForecasts[3] = co + MyForecasts[2]
MyForecasts[4] = co + MyForecasts[3]
plt.plot(np.array(y[len(y)-5:len(y)]),'k-')
plt.plot(MyForecasts,'r:')
plt.title("black is y_t, red is the forecasts")
plt.show()
```



I can now compute the MASE for this method as follows:

```
Real = np.array(y[len(y)-5:len(y)])
```

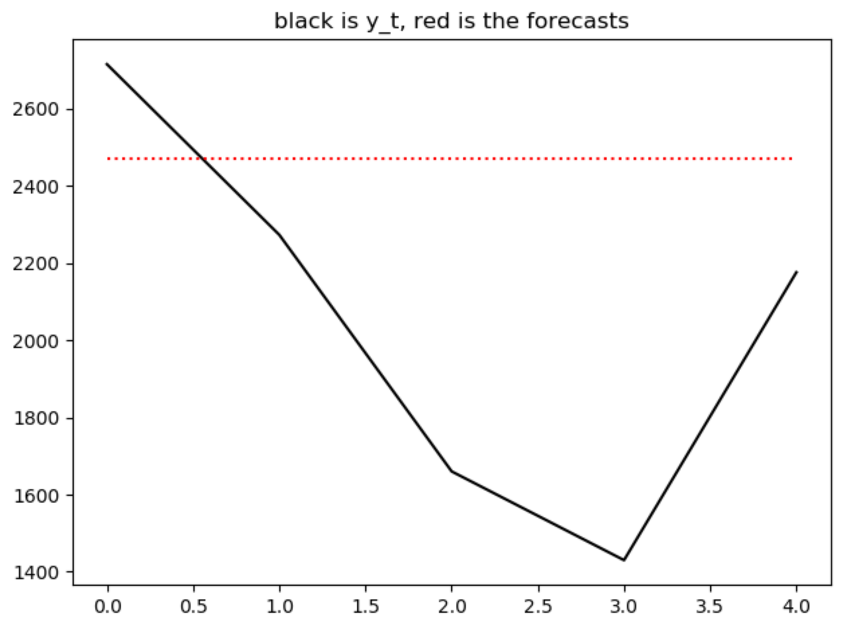
```
def MASE(y,Real,Fore,m):
    h = len(Fore)
    fenzi,fenmu = 0,0
    for i in range(h):
        fenzi = fenzi + abs(Fore[i] - Real[i])
    fenzi = fenzi/h
    for i in range(m,n):
        fenmu = fenmu + abs(y[i] - y[i-m])
    fenmu = fenmu/(n-m)
    return fenzi/fenmu

def MAPE(Real,Fore):
    h = len(Fore)
    ratio = 0
    fenzi, fenmu = 0, 0
    for i in range(h):
        fenzi = abs(Fore[i]-Real[i])
        fenmu = abs(Fore[i]) + abs(Real[i])
        ratio = ratio + (fenzi/fenmu)
    return ratio*(200/h)

print(MASE(y,Real,MyForecasts,4))
print(MAPE(Real,MyForecasts))
```

Running Result:

```
0.8384063379133543
53.685157661199085
```

Now suppose I want to compare the performance with the simple exponential smoothing (Local level model) using the SSOE assumption. I need to estimate and predict the series:

```
a = np.zeros(obs)
v = np.zeros(obs)


a[0] = xxx[0]
gamma = res.x[0]

for t in range(1,obs):
    v[t] = xxx[t] - z*a[t-1]
    a[t] = a[t-1] + gamma* v[t]

LLForecasts = [0*i for i in range(5)]
print(LLForecasts)

LLForecasts[0] = a[obs-1]
LLForecasts[1] = LLForecasts[0]
LLForecasts[2] = LLForecasts[1]
LLForecasts[3] = LLForecasts[2]
LLForecasts[4] = LLForecasts[3]

print('000',np.array(y[len(y)-5:len(y)]))
print('000',LLForecasts)
plt.plot(np.array(y[len(y)-5:len(y)]),'k-')
plt.plot(LLForecasts,'r:')
plt.title("black is y_t, red is the forecasts")
plt.show()
```



Now if we compare the results of the MASE we have:

```
Real = np.array(y[len(y)-5:len(y)])
print(MASE(y,Real,LLForecasts,4))
print(MAPE(Real,LLForecasts))
```

Running Result:

```
1.388755076524717
75.57251072205929
```

Forecast competion in action!

Suppose we have a dataset and we want to compare the forecast performance of different models using the MASE and the MAPE as above.

Below we provide the code for forecasting $h-steps$ ahead using: Model 1 in both the multiple source of error (MSOE) version and the SSOE version. Same for the Theta method in both versions and the damped trend model (SSOE only).

```
import numpy as np
import math
from scipy.optimize import minimize

def ForecastARkf0(x):
    a = [i for i in range(n)]
    p = [i for i in range(n)]
    a[0] = y[0]
    p[0] = 10000
    k = [i for i in range(n)]
    v = [i for i in range(n)]

    w = 1-math.exp(-abs(x[2]))
    likelihood = 0
    sigmae = 0
    z = 1
    for t in range(1, n):
        k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + x[0]
        v[t] = y[t] - z * a[t - 1]
        a[t] = x[1] + w * a[t - 1] + k[t] * v[t]
        sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))
        likelihood = likelihood + 0.5 * np.log(2 * np.pi) + 0.5 + 0.5 * np.log(z
** 2 * p[t - 1] + 1)
    likelihood = likelihood + 0.5 * n * np.log(sigmae / n)
    return likelihood

def ForecastARkf1():
    v_ans = lambda x: ForecastARkf0(x)
    return v_ans


def ForcastARkf(y,h):
    n = len(y)
    x0 = np.asarray((0.2,1,2))
    res = minimize(ForecastARkf1(), x0, method='SLSQP')

    a = [i for i in range(n)]
    p = [i for i in range(n)]
    a[0] = y[0]
    p[0] = 10000
    k = [i for i in range(n)]
    v = [i for i in range(n)]
    v[0] = 0
    z = 1
    q = abs(res.x[0])
    co = abs(res.x[1])
    w = 1-math.exp(-abs(res.x[2]))
    sigmae = 0
    for t in range(1,n):
        k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + q
        v[t] = y[t] - z * a[t - 1]
        a[t] = co + w * a[t - 1] + k[t] * v[t]
        sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))

    Forec = [0*i for i in range(h)]
    Forec[0] = a[len(y)]
    for i in range(1,h):
        Forec[i] = co + w*Forec[i-1]
    return Forec
```

```
def logLikConc0(x):
    state = np.zeros(len(y))
    v = np.zeros(len(y))
    state[0] = y[0]

    w = 1-math.exp(-abs(x[2]))
    v2 = 0
    for t in range(1, len(y)):
        v[t] = y[t] - state[t-1]
        v2 = v2 + v[t]**2
        state[t] = x[1] + w*state[t-1] + x[2]* v[t]
    return (v2-v[0])

def logLikConc1():
    v_ans = lambda x: logLikConc0(x)
    return v_ans


def logLikConc(y,h):
    n = len(y)
    state = np.zeros(len(y))
    v = np.zeros(len(y))

    x0 = np.asarray((0.2,1,2))
    res = minimize(ForecastARkf1(), x0, method='SLSQP')

    w = 1-math.exp(-abs(res.x[0]))
    gamma = abs(res.x[1])
    co = abs(res.x[2])

    for t in range(1,len(y)):
        v[t] = y[t] - state[t - 1]
        state[t] = co + w * state[t - 1] + gamma * v[t]

    Forec = [0*i for i in range(h)]
    Forec[0] = state[len(y)]
    for i in range(1,h):
        Forec[i] = co + w*Forec[i-1]
    return Forec
```

```
def logLikConc0(x):
    state = np.zeros(len(y))
    state[0] = y[0]
    v = np.zeros(len(y))
    w = 1
    v2 = 0
    for t in range(1, len(y)):
        v[t] = y[t] - state[t-1]
        v2 = v2 + v[t]**2
        state[t] = x[0] + w*state[t-1] + x[1]* v[t]
    return (v2-v[0])

def logLikConc1():
    v_ans = lambda x: logLikConc0(x)
    return v_ans


def ForecastTheta(y,h):
    n = len(y)
    state = np.zeros(len(y))
    v = np.zeros(len(y))

    x0 = np.asarray((0.3,1))
    res = minimize(ForecastARkf1(), x0, method='SLSQP')
    w = 1
    gamma = abs(res.x[0])
    co = abs(res.x[1])

    for t in range(1,len(y)):
        v[t] = y[t] - state[t - 1]
        state[t] = co + w * state[t - 1] + gamma * v[t]

    Forec = [0*i for i in range(h)]
    Forec[0] = state[len(y)]
    for i in range(1,h):
        Forec[i] = co + w*Forec[i-1]
    return Forec
```

```
def funcTheta0(x):
    k = np.zeros(len(y))
    p = np.zeros(len(y))
    a = np.zeros(len(y))
    v = np.zeros(len(y))
    a[0] = y[0]
    p[0] = 10000
    z = w = 1
    likelihood = 0
    for t in range(1, n):
        k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + x[0]
        v[t] = y[t] - z * a[t - 1]
        a[t] = x[1] + w * a[t - 1] + k[t] * v[t]
        sigmae = sigmae + (v[t] ** 2 / (z ** 2 * p[t - 1] + 1))
        likelihood = likelihood + 0.5 * np.log(2 * np.pi) + 0.5 + 0.5 * np.log(z
** 2 * p[t - 1] + 1)
    likelihood = likelihood + 0.5 * n * np.log(sigmae / n)
    return likelihood

def funcTheta1():
    v_ans = lambda x: logLikConc0(x)
    return v_ans


def ForecastThetakf(y,h):
    n = len(y)
    a = np.zeros(len(y))
    p = np.zeros(len(y))
    a[0] = y[0]
    p[0] = 10000
    k = np.zeros(len(y))
    v = np.zeros(len(y))
    v[0] = 0

    x0 = np.asarray((0.3,1))
    res = minimize(funcTheta1(), x0, method='SLSQP')

    z = w = 1
    q = abs(res.x[0])
    co = abs(res.x[1])
```

```python
    for t in range(1,len(y)):
        k[t] = (z * w * p[t - 1]) / (z ** 2 * p[t - 1] + 1)
        p[t] = w ** 2 * p[t - 1] - w * z * k[t] * p[t - 1] + q
        v[t] = y[t] - z * a[t - 1]
        a[t] = co + w * a[t - 1] + k[t] * v[t]


    Forec = [0*i for i in range(h)]
    Forec[0] = a[n]
    for i in range(1,h):
        Forec[i] = co + w*Forec[i-1]
    return Forec
```

```python
def fmsoe0(x):
    obs = len(y)
    damped = np.matlib.zeros((obs, 2))
    damped[0, 0] = y[0]
    damped[0, 1] = 0
    inn = np.matlib.zeros((obs, 1))
    e2 = 0
    for t in range(1, obs):
        inn[t] = y[t] - damped[t-1,0] - x[2]* damped[t-1,1]
        damped[t,0] = damped[t-1,0] + x[2] * damped[t-1,1] + x[0] * inn[t]
        damped[t,1] = x[2]*damped[t - 1, 1] + x[1] * inn[t]
        e2 = e2 + inn[t]**2
    return (e2-inn[0]**2)/obs

def fmsoe1():
    v_ans = lambda x: logLikConc0(x)
    return v_ans


def ForecastDamped(y,h):
    obs = len(y)
    damped = np.matlib.zeros((obs,2))
    damped[0,0] = y[0]
    damped[0,1] = 0
    inn = np.matlib.zeros((obs, 1))

    x0 = np.asarray((np.random.rand(1),np.random.rand(1),np.random.rand(1)))
    res = minimize(fmsoe1(), x0, method='SLSQP')

    k1 = abs(res.x[0])
    k2 = abs(res.x[1])
    k3 = abs(res.x[2])

    if(k3>1): k3 = 1
    for t in range(1,obs):
        inn[t] = y[t] - damped[t - 1, 0] - k3 * damped[t - 1, 1]
        damped[t, 0] = damped + k3 * damped[t - 1, 1] + k1 * inn[t]
        damped[t, 1] = k3 * damped[t - 1, 1] + k2 * inn[t]

    Forec = [0*i for i in range(h)]
    Forec[0] = damped[obs-1,0] + k3* damped[obs-1,1]
    for i in range(1,h):
        Forec[i] = Forec[i-1] + damped[obs-1,1] * k3 **i
    return Forec
```

We can now run a real forecast competition evaluating for each model the MASE and sMAPE and their mean and median across the series of the competition.