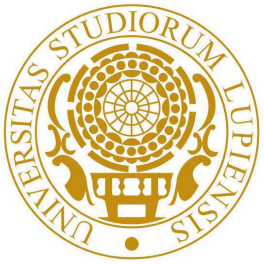



**UNIVERSITY OF SALENTO**

**COLLEGE ON ENGINEERING**

**DEGREE COURSE IN COMPUTER ENGINEERING**

---

**PROJECT DOCUMENTATION**
**OF ESTIMATION AND DATA ANALYSIS**

# COMPARISON OF DIFFERENT VERSIONS OF KF FOR THE PREDICTION OF THE PANDEMIC PROGRESSION OF COVID

**TEACHERS:**
**Prof. Daniela DE PALMA**
**Prof. Gianfranco PARLANGELI**

**STUDENT:**
**Danilo GIOVANNICO**
**Matr. 20039574**

---

**ACADEMIC YEAR 2020/2021**

INDEX

CHAPTER 1 – INTRODUCTION
CHAPTER 2–SHORT-TERM FORECAST MODEL BASED ON THE KALMAN FILTER FOR THE SPREAD OF COVID
2.1 IMPLEMENTATION OF KF WITH PYTHON AND R ON THE COVID CASE FOR THE PREDICTION OF SPREADS
2.2 SHORT-TERM FORECAST MODEL BASED ON A ROBUST VERSION TO THE OUTLIERS OF THE KALMAN FILTER
CHAPTER 3 - INTEGRATION OF UKF (UNSCENTED KALMAN FILTER) AND EKF (EXTENDED KALMAN FILTER) IN THE SEIR AND SIRD EPIDEMIOLOGICAL MODEL TO PREVENT THE TREND OF EPIDEMIC CURVES
3.1 IDEA OF INTEGRATION OF EKF (EXTENDED KALMAN FILTER) INTO COMPARTMENTAL MODELS
3.2 IDEA OF INTEGRATION OF THE UKF (UNSCENTED KALMAN FILTER) INTO COMPARTMENTAL MODELS
APPENDIX
BIBLIOGRAPHY

## CHAPTER 1 – INTRODUCTION

The COVID-19 pandemic has fascinated scientific activity since its earliest days. Particular attention was paid to identifying the underlying dynamics and forecasting future trends. Numerous studies have been proposed in this regard using different study models and methodologies; to be more successful we highlight the compartmental models, whose types differ from type to type. Among the most important we mention the SIR and the SEIR, and then see numerous other types proposed in the literature.
The dataset from which to draw data is provided by the civil protection and can be downloaded at the following link https://github.com/pcm-dpc/COVID-19, and provides countless national, regional and provincial statistics on the daily progress of epidemic in Italy.
Our study will focus on individual regions and not nationally, since I believe that with the color division used in the last period to contain the epidemic, a study limited to the single region is more correct.

In the first part of the document, a study on the spread of epidemic curves will be performed by exploiting the potential that the Kalman Filter offers as a tool to estimate the state of a linear dynamic system perturbed by noise, on the basis of measurements (or observations) linearly dependent on the state and noisy. Furthermore, we will see a robust version of the Kalman filter that uses a robust estimation algorithm for outliers called LEL, always applied to the case study of the spread.

In the second part, we will see a case study of nonlinear estimation applied to the SIRD (susceptible-infectious-recovered-dead) compartmental model. In that case, we will use the extended Kalman filter. Also, we will see another case study which sees the involvement of compartmental models, in particular the SEIR (susceptible-exposed-infectious-recovered), combined with the Unscented Kalman Filter, a different version of the Kalman conceived for non-linear models and therefore applied to non-linear estimates.

The purpose of this article is purely for educational purposes, in order to refine these estimates it is necessary to set optimal initial parameters (which certainly could be found from publications on the net) in order to obtain the most realistic case possible.

## CHAPTER 2 – SHORT-TERM FORECAST MODEL BASED ON THE KALMAN FILTER FOR THE SPREAD OF COVID

**Kalman Filter**

The KF uses a linear state space model to estimate the true (hidden) states, including past, present and future, of a process given a set of observations, in which the mean square error is minimized [**Appendix [1 ]**]. We denote with $x_t \in R^n$ the true state and with $y_t \in R^m$ the observation, the linear dynamic model that the KF tries to face can be specified as follows,

$$x_t = Ax_{t-1} + w_{t-1}$$

$$y_t = Hx_t + v_t$$

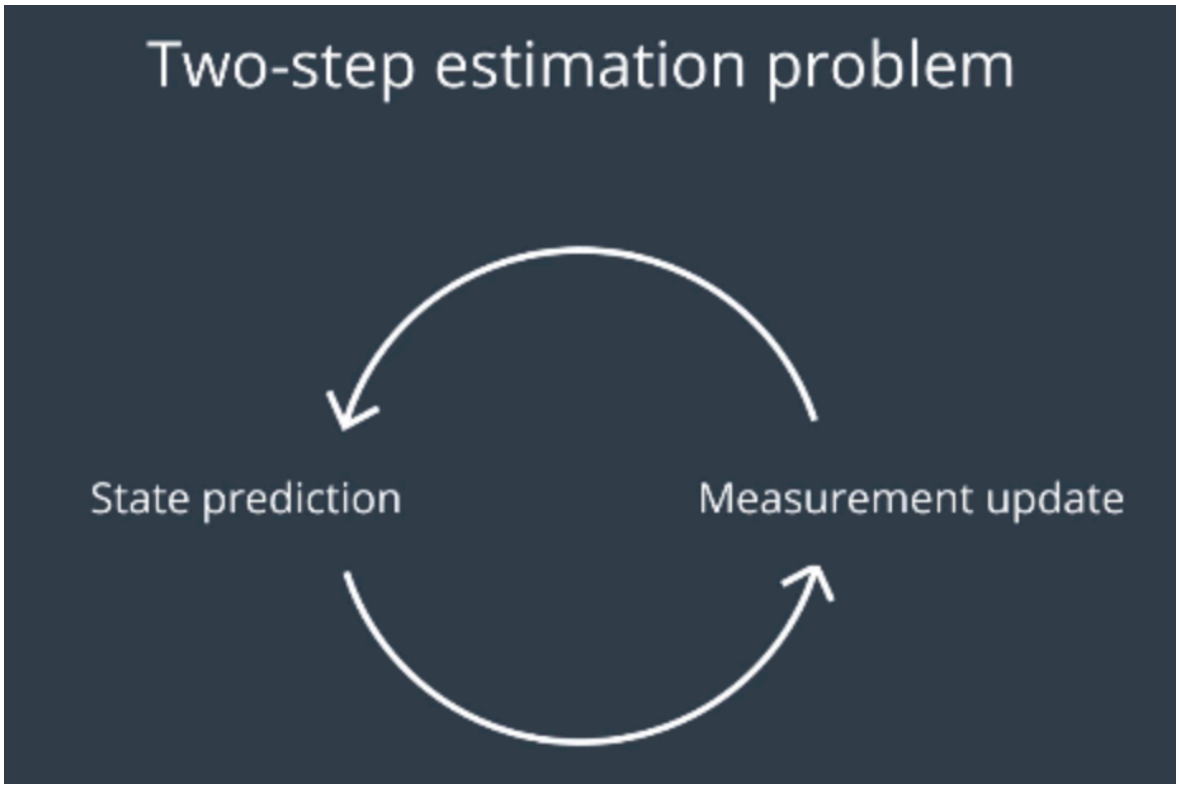
where:

- $w_t \sim N(0,Q)$ is the state error matrix, with Q process noise covariance matrix;
- $v_t \sim N(0,R)$ is the matrix of the observation error, with R the covariance matrix of the measurement noise;
- A (nxn) is the transition matrix, i.e. it denotes the relationships between the states of the previous time phase and the current when the process noise is absent;
- H (mxn) is the observation matrix, it establishes the relationship between state and measures.

**PLEASE NOTE.** The matrices A, B, H, Q and R may change over time, but in that article they are assumed to be constants.

Also in KF, $(x_t \mid x_{t-1})$ and $(y_t \mid x_t)$ are supposed to be Gaussian.

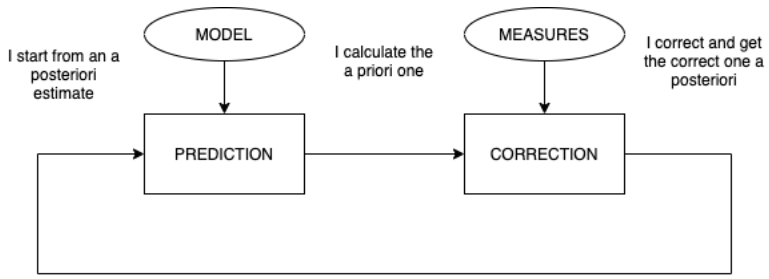
The Kalman Filter alternates two steps: the "prediction" step and the "update" step. In the state prediction and covariance step of the estimation error they are projected forward in time by performing a prediction thanks to the model provided; in the update one, the available measures are used to correct the estimate and the covariance previously calculated.

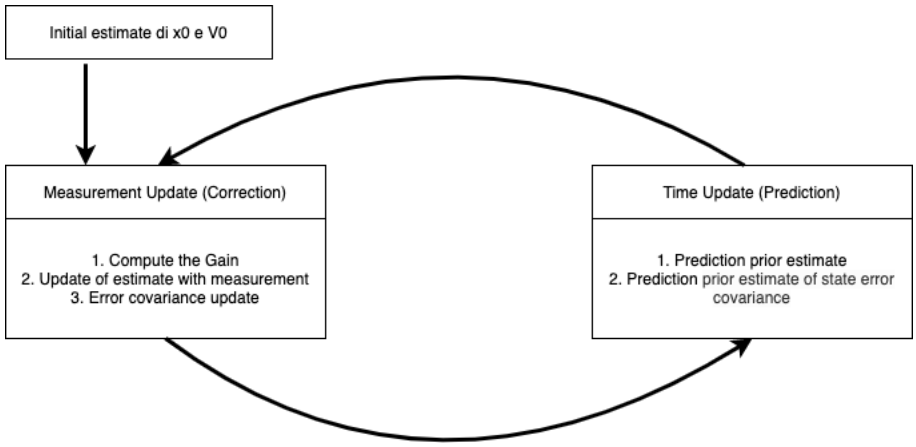



Be:

- $X_{t|t-1}$ the a priori estimate at time t given the information available at time t-1,
- $V_{t|t-1}$ the own estimate of the state error covariance,
- $\hat{X}_t$ the a posteriori estimate given by observation $Y_t$,
- $V_t$ the a posteriori estimate of the state error covariance.

The Kalman filter can be expressed as:






*KF Prediction*

$$\hat{X}_{t|t-1} = A_{t-1}\hat{X}_{t-1}$$

$$V_{t|t-1} = A_{t-1}V_{t-1}A_{t-1}^T + Q$$

*KF Updating*

$$E_t = Y_t - H_t\hat{X}_{t|t-1}$$

$$S_t = H_tV_{t|t-1}H_t^T + R$$

$$K_t = V_{t|t-1}H_t^TS_t^{-1}$$

$$\hat{X}_t = \hat{X}_{t|t-1} + K_tE_t$$

$$V_t = V_{t|t-1} - K_tS_tK_t^T$$

where:

- $E_t$ it is called Innovation or Residue and represents the difference between real and predicted measure, that is, it contains the new information.
- $S_t$ is the measurement prediction covariance matrix at step t.
- $K_t$ is called Kalman gain, and is the one who allows the correction of the state, defining how much the forecast needs correction giving more importance to the model or measure depending on the covariance.

**Representation of the state-space of the dynamic system (discrete-time) used for the prediction of Spreads**

In the case studies relating to the simple Kalman filter and the robust case (LEL) we used a simple linear model, which allows us to represent the rate of increase / decrease of

positive individuals (also applicable to the healed or deceased case); represented as follows:

$$x_k = v\Delta t + x_{k-1}$$

We write the state vector as follows:

$$r = \begin{bmatrix} x \\ y \end{bmatrix}$$

where x represents the number of positives and y the rate of increase / decrease. We describe the process and measurement equations as follows:

*Process model*

$$x_k = x_{k-1} + y_{k-1}\Delta t + v_{1_{k-1}}$$

$$y_k = y_{k-1} + v_{2_{k-1}}$$

Where the vector $v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$.

Of course, to obtain the Kalman equations we have to transcribe the following equations in matrix form; we have:

$$r_k = \begin{bmatrix} x_k \\ y_k \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{k-1} \\ y_{k-1} \end{bmatrix} + v_k$$

that is:

$$r_k = Fr_{k-1} + v_k$$

*Measurement model*

$$z_k = x_k + w_k$$

The matrix form can be written as:

$$z_k = Hr_k + w_k = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \end{bmatrix} + w_k.$$

For more information see Bibliography 1.

## 2.1 IMPLEMENTATION OF KF WITH PYTHON AND R ON THE COVID CASE FOR THE PREDICTION OF SPREADS

In our case study, the KF was used to predict the variation of the Spread, limited to a single region, of positive cases (alternatively we could also evaluate that on deaths and recoveries) over time, using the dataset provided by the civil protection.

*Preprocessing Dataset*

We extrapolate the useful data by reading from the statistical CSV on the regions, in our case we take: date, state, region name, latitude, longitude, total positive cases, total cases since the beginning of the pandemic, deceased and discharged healed.

```
[2]: dataframe = pd.read_csv("../DATASET/COVID-19/dati-regioni/dpc-covid19-ita-regioni.csv", delimiter = ',')
     dataframe.head()
```

[2]:

| | data | stato | codice_regione | denominazione_regione | lat | long | ricoverati_con_sintomi | terapia_intensiva | totale_ospedalizzati | isolamento_domiciliare | ... | deceduti | casi_da_sospetto_diagnostico | casi_da_s |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2020-02-24T18:00:00 | ITA | 13 | Abruzzo | 42.351222 | 13.398438 | 0 | 0 | 0 | 0 | ... | 0 | NaN | |
| 1 | 2020-02-24T18:00:00 | ITA | 17 | Basilicata | 40.639471 | 15.805148 | 0 | 0 | 0 | 0 | ... | 0 | NaN | |
| 2 | 2020-02-24T18:00:00 | ITA | 18 | Calabria | 38.905976 | 16.594402 | 0 | 0 | 0 | 0 | ... | 0 | NaN | |
| 3 | 2020-02-24T18:00:00 | ITA | 15 | Campania | 40.839566 | 14.250850 | 0 | 0 | 0 | 0 | ... | 0 | NaN | |
| 4 | 2020-02-24T18:00:00 | ITA | 8 | Emilia-Romagna | 44.494367 | 11.341721 | 10 | 2 | 12 | 6 | ... | 0 | NaN | |

5 rows × 24 columns

```
[3]: cleared_dataframe = dataframe[['data', 'stato','denominazione_regione','lat','long','totale_positivi','totale_casi','deceduti','dimessi_guariti']]
     cleared_dataframe.head()
```

[3]:

| | data | stato | denominazione_regione | lat | long | totale_positivi | totale_casi | deceduti | dimessi_guariti |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 2020-02-24T18:00:00 | ITA | Abruzzo | 42.351222 | 13.398438 | 0 | 0 | 0 | 0 |
| 1 | 2020-02-24T18:00:00 | ITA | Basilicata | 40.639471 | 15.805148 | 0 | 0 | 0 | 0 |
| 2 | 2020-02-24T18:00:00 | ITA | Calabria | 38.905976 | 16.594402 | 0 | 0 | 0 | 0 |
| 3 | 2020-02-24T18:00:00 | ITA | Campania | 40.839566 | 14.250850 | 0 | 0 | 0 | 0 |
| 4 | 2020-02-24T18:00:00 | ITA | Emilia-Romagna | 44.494367 | 11.341721 | 18 | 18 | 0 | 0 |

We perform further processing on the extracted dataframe so that we can build a time series by region:

```
[4]: header = ['region','lat','long','population']
     for date in cleared_dataframe['data']:
         if date.split('T')[0] not in header :
             header.append(date.split('T')[0])

     regions_array = cleared_dataframe['denominazione_regione'].iloc[:21]
     lat_array = cleared_dataframe['lat'].iloc[:21]
     long_array = cleared_dataframe['long'].iloc[:21]
     population_array = ['1305770','556934','1924701','5785861','4467118','1211357','5865544','1543127',
                         '10103969','1518400','302265','106951','117417','4341375','4008296','1630474',
                         '4968410','3722729','880285','125501','4907704']

     confirmed_df = pd.DataFrame(columns=header)
     confirmed_df['region'] = regions_array
     confirmed_df['lat'] = lat_array
     confirmed_df['long'] = long_array
     confirmed_df['population'] = population_array
     for count, column in enumerate(confirmed_df.drop(['region','lat','long','population'], axis = 1)):
         confirmed_df[column] = cleared_dataframe['totale_positivi'].iloc[count*21:(count+1)*21].tolist()
     confirmed_df.to_csv(r'confirmed_ts.csv', index = False, header=True)

     death_df = pd.DataFrame(columns=header)
     death_df['region'] = regions_array
     death_df['lat'] = lat_array
     death_df['long'] = long_array
     death_df['population'] = population_array
     for count, column in enumerate(death_df.drop(['region','lat','long','population'], axis = 1)):
         death_df[column] = cleared_dataframe['deceduti'].iloc[count*21:(count+1)*21].tolist()
     death_df.to_csv(r'death_ts.csv', index = False, header=True)

     recover_df = pd.DataFrame(columns=header)
     recover_df['region'] = regions_array
     recover_df['lat'] = lat_array
     recover_df['long'] = long_array
     recover_df['population'] = population_array
     for count, column in enumerate(recover_df.drop(['region','lat','long','population'], axis = 1)):
         recover_df[column] = cleared_dataframe['dimessi_guariti'].iloc[count*21:(count+1)*21].tolist()
     recover_df.to_csv(r'recover_ts.csv', index = False, header=True)

     recover_df.head()
```

[4]:

| | region | lat | long | population | 2020-02-24 | 2020-02-25 | 2020-02-26 | 2020-02-27 | 2020-02-28 | 2020-02-29 | ... | 2020-12-19 | 2020-12-20 | 2020-12-21 | 2020-12-22 | 2020-12-23 | 2020-12-24 | 2020-12-25 | 2020-12-26 | 2020-12-27 | 2020-12-28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Abruzzo | 42.351222 | 13.398438 | 1305770 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 19195 | 19307 | 19739 | 20235 | 20705 | 21248 | 21408 | 21419 | 21507 | 21812 |
| 1 | Basilicata | 40.639471 | 15.805148 | 556934 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 3760 | 3804 | 3865 | 3964 | 4120 | 4200 | 4251 | 4271 | 4280 | 4321 |
| 2 | Calabria | 38.905976 | 16.594402 | 1924701 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 11999 | 12260 | 12511 | 12588 | 13099 | 13225 | 13308 | 13507 | 13665 | 13749 |
| 3 | Campania | 40.839566 | 14.250850 | 5785861 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 92927 | 94031 | 95128 | 98167 | 100527 | 101601 | 102065 | 102580 | 103370 | 104549 |
| 4 | Emilia-Romagna | 44.494367 | 11.341721 | 4467118 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 85872 | 87594 | 88011 | 91411 | 94188 | 96529 | 96910 | 100333 | 100659 | 101255 |

5 rows × 313 columns

We define a function that from the new elaborated dataframe can prepare the data for plotting and spread prediction. The dataframe obtained using this function will be the

following:

```
[5]: def create_ts(df):
         ts=df
         ts=ts.drop(['lat','long','population'], axis=1)
         ts.set_index('region')
         ts=ts.T
         ts.columns=ts.loc['region']
         ts=ts.drop('region')
         ts=ts.fillna(0)
         ts=ts.reindex(sorted(ts.columns), axis=1)
         return (ts)
```

```
[6]: ts_confirmed=create_ts(confirmed_df)
     ts_death=create_ts(death_df)
     ts_recover=create_ts(recover_df)
     ts_confirmed.head()
```

[6]:

| region | Abruzzo | Basilicata | Calabria | Campania | Emilia-Romagna | Friuli Venezia Giulia | Lazio | Liguria | Lombardia | Marche | ... | P.A. Bolzano | P.A. Trento | Piemonte | Puglia | Sardegna | Sicilia | Toscana | Umbria | Valle d'Aosta | Veneto |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2020-02-24 | 0 | 0 | 0 | 0 | 18 | 0 | 2 | 0 | 166 | 0 | ... | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 32 |
| 2020-02-25 | 0 | 0 | 0 | 0 | 26 | 0 | 2 | 1 | 231 | 0 | ... | 1 | 0 | 3 | 0 | 0 | 3 | 2 | 0 | 0 | 42 |
| 2020-02-26 | 0 | 0 | 0 | 0 | 46 | 0 | 0 | 11 | 249 | 1 | ... | 1 | 0 | 3 | 0 | 0 | 3 | 2 | 0 | 0 | 69 |
| 2020-02-27 | 1 | 0 | 0 | 3 | 96 | 0 | 0 | 19 | 349 | 3 | ... | 1 | 0 | 2 | 1 | 0 | 2 | 2 | 0 | 0 | 109 |
| 2020-02-28 | 1 | 0 | 1 | 4 | 143 | 0 | 0 | 19 | 474 | 6 | ... | 1 | 0 | 11 | 3 | 0 | 2 | 7 | 0 | 0 | 149 |

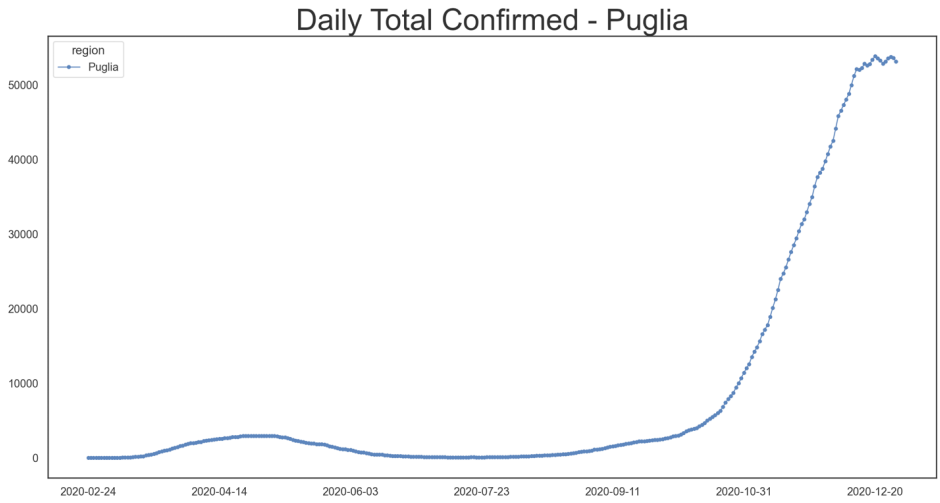
5 rows × 21 columns

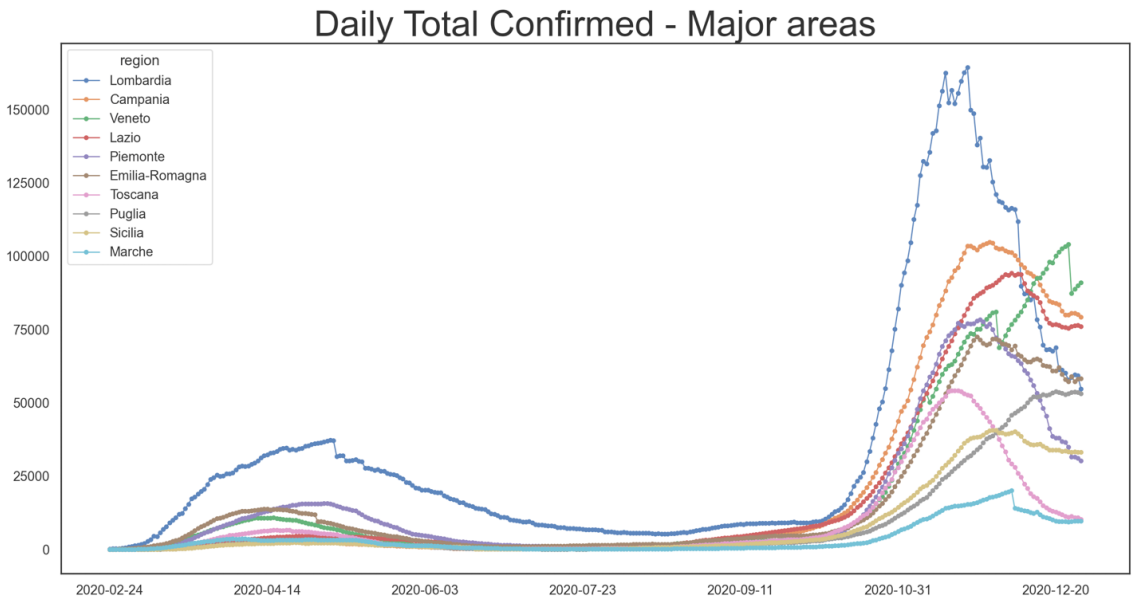
## *Plotting Dataset*

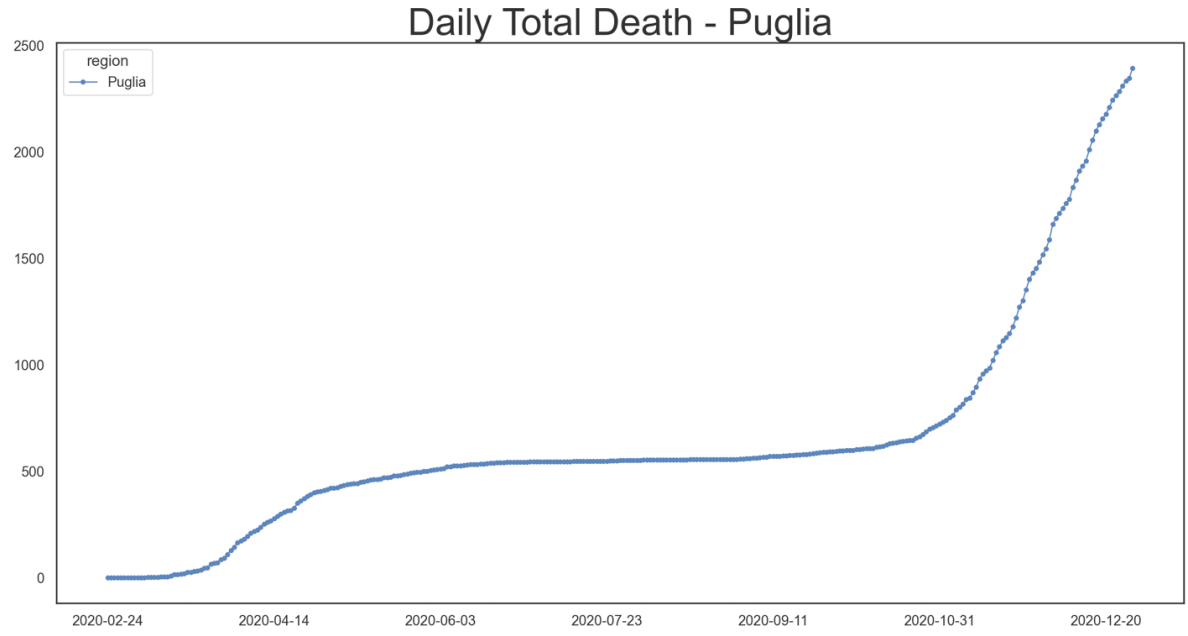
We plot the data provided by the civil protection and view the spreads for confirmed, deceased and cured cases:

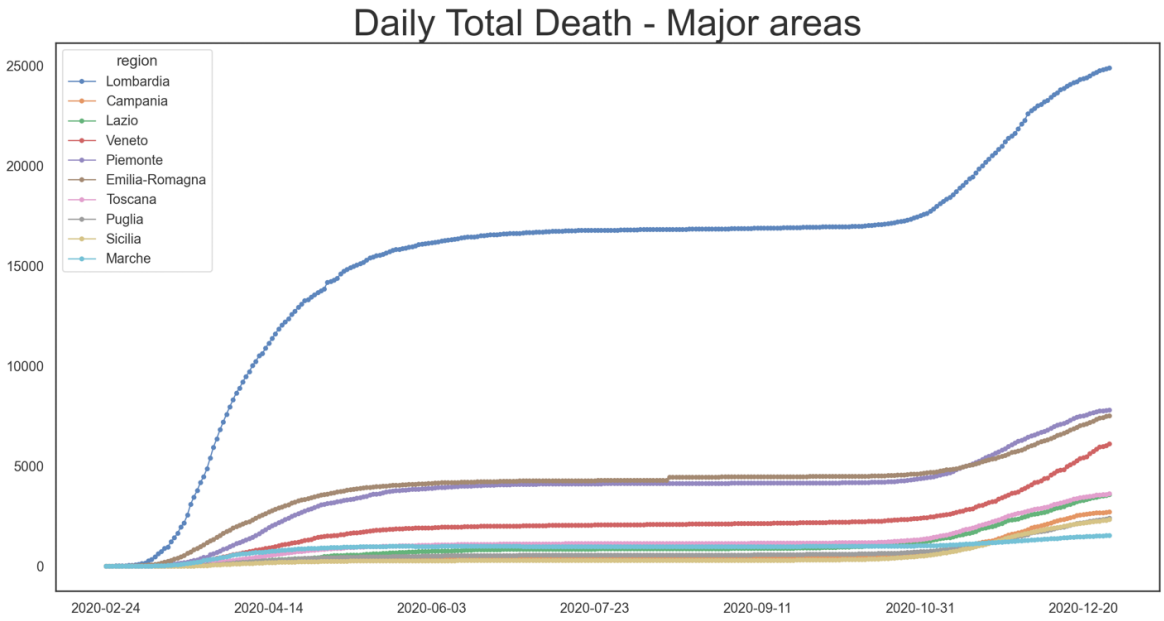
```
p=ts_confirmed.reindex(ts_confirmed.max().sort_values(ascending=False).index, axis=1)
p.iloc[:,7:8].plot(marker='o',linewidth=1,markersize=3,figsize=(16,8)).set_title('Daily Total Confirmed - Puglia',fontdict={'fontsize': 30})
p.iloc[:,0:10].plot(marker='o',linewidth=1,markersize=3,figsize=(16,8)).set_title('Daily Total Confirmed - Major areas',fontdict={'fontsize': 30})

p_d=ts_death.reindex(ts_confirmed.mean().sort_values(ascending=False).index, axis=1)
p_d.iloc[:,7:8].plot(marker='o',linewidth=1,markersize=3,figsize=(16,8)).set_title('Daily Total Death - Puglia',fontdict={'fontsize': 30})
p_d.iloc[:,0:10].plot(marker='o',linewidth=1,markersize=3,figsize=(16,8)).set_title('Daily Total Death - Major areas',fontdict={'fontsize': 30})

p_r=ts_recover.reindex(ts_confirmed.mean().sort_values(ascending=False).index, axis=1)
p_r.iloc[:,7:8].plot(marker='o',linewidth=1,markersize=3,figsize=(16,8)).set_title('Daily Total Recoverd - Puglia',fontdict={'fontsize': 30})
p_r.iloc[:,0:10].plot(marker='o',linewidth=1,markersize=3,figsize=(16,8)).set_title('Daily Total Recoverd - Major areas',fontdict={'fontsize': 30})

mplcursors.cursor(hover=True)
```

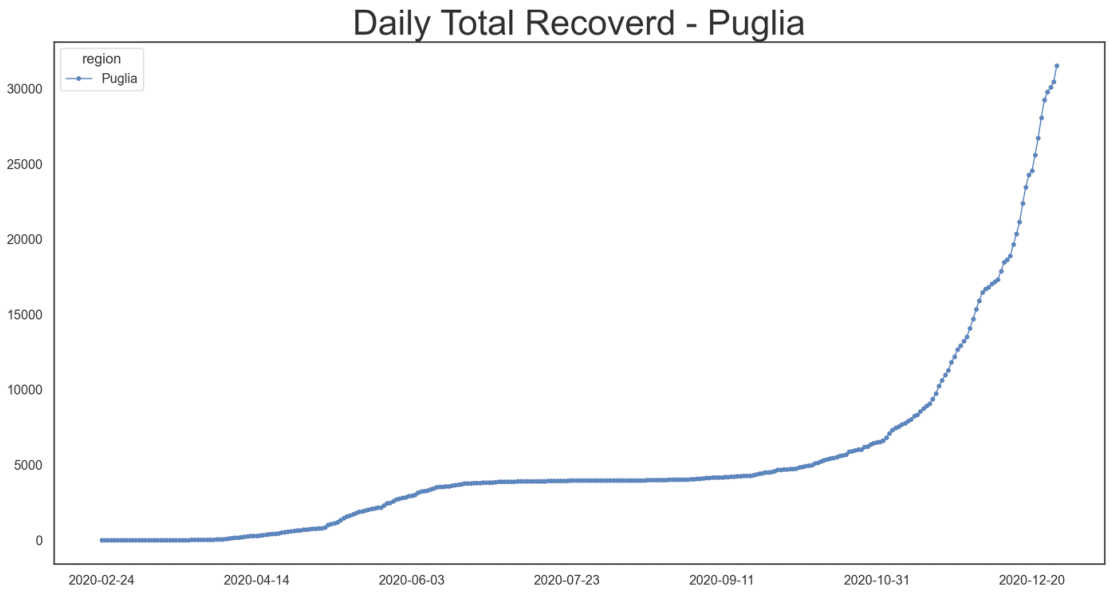
The graphs obtained will be the following:

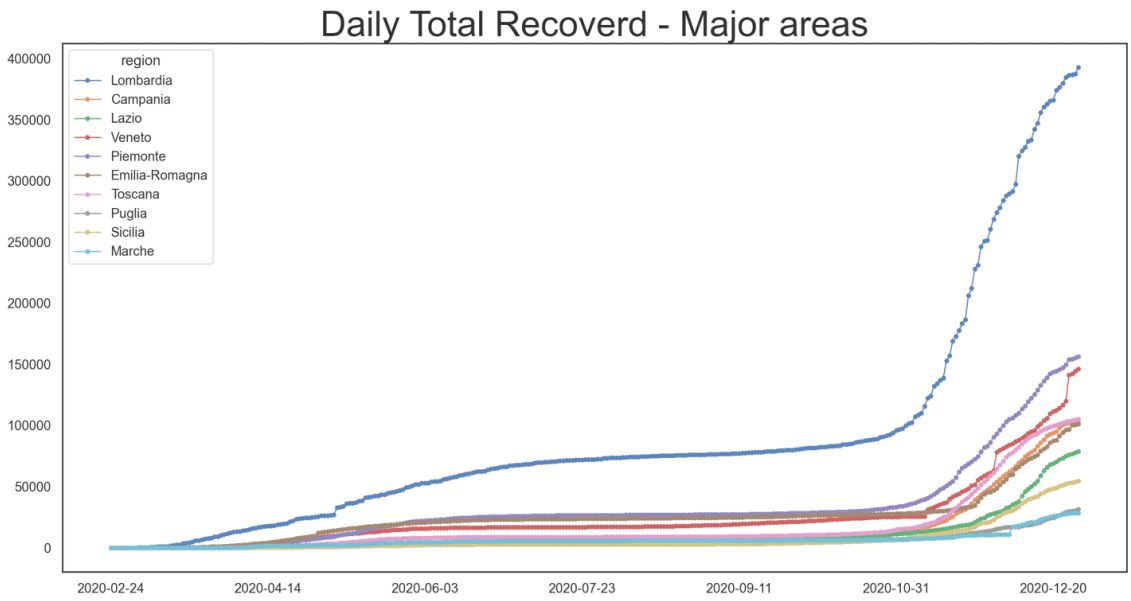



Daily Total Confirmed - Major areas
region
Lombardia
Campania
Veneto
Lazio
Piemonte
Emilia-Romagna
Toscana
Puglia
Sicilia
Marche
150000
125000
100000
75000
50000
25000
0
2020-02-24
2020-04-14
2020-06-03
2020-07-23
2020-09-11
2020-10-31
2020-12-20


Daily Total Death - Puglia
region
Puglia
2500
2000
1500
1000
500
0
2020-02-24
2020-04-14
2020-06-03
2020-07-23
2020-09-11
2020-10-31
2020-12-20


Daily Total Death - Major areas
region
Lombardia
Campania
Lazio
Veneto
Piemonte
Emilia-Romagna
Toscana
Puglia
Sicilia
Marche
25000
20000
15000
10000
5000
0
2020-02-24
2020-04-14
2020-06-03
2020-07-23
2020-09-11
2020-10-31
2020-12-20


Daily Total Recoverd - Puglia
region
Puglia
30000
25000
20000
15000
10000
5000
0
2020-02-24
2020-04-14
2020-06-03
2020-07-23
2020-09-11
2020-10-31
2020-12-20




From the graph on the spread of confirmed cases it can clearly be seen how the epidemic curve increases and decreases in correspondence with the two waves, it is also possible to note how regions such as Lombardy have anticipated the course of the infection and are in a phase of decline compared to regions like Puglia. Of course, these graphs are not true since they do not consider asymptomatic patients, who most likely will not undergo a swab, but are authentic only to the cases recorded from region to region.

*One day prediction*

Let's save the new dataframe obtained in a csv file:

```
ts_r=ts_confirmed.reset_index()
ts_r=ts_r.rename(columns = {'index':'date'})
ts_r['date']=pd.to_datetime(ts_r['date'] ,errors ='coerce')
ts_r.tail()
ts_r.to_csv(r'ts_r.csv', index = False, header=True)
```

Recall package R via the rpy2 library for prediction of spreads through KF, import the libraries useful for prediction, load the pre-processed dataset by adding a day for prediction and initialize the matrices for the construction of the filter. The prediction is cycled by region column by column:

```
%%R
library(pracma)
library(Metrics)
library(readr)
all<- read.csv("/Users/danilogiovannico/Desktop/PROGETTO-Estimation\ and\ Data\ Analysis\ with\ Applications/COVID\ PREDICTIONS/ts_r.csv")
all$X1<-NULL
date<-all[,1, drop=FALSE]
# La funzione rbind () combina vettori, matrici o frame di dati per righe
date<-rbind(date, data.frame(date = format(as.Date(all[nrow(all),1])+1)))
#date[nrow(date) + 1,1] <- as.Date(all[nrow(all),1])+1
pred_all<-NULL
# ncol ritorna il numero di colonne presenti nel dataset, nrow il numero di righe
# Iteriamo per colonna sulle regioni
for (n in 2:ncol(all)-1) {
    # Costruisco la time series
    #La funzione ts () convertirà un vettore numerico in un oggetto della serie temporale R.
    #Il formato è ts (vector, start=, end=, frequency=) dove inizio e fine sono i tempi della prima e dell'ultima osservazione e la frequenza è
    #il numero di osservazioni per unità di tempo (1 = annuale, 4 = trimestrale, 12 = mensile, ecc.).
    Y<-ts(data = all[n+1], start = 1, end =nrow(all)+1)
    # Time Step
    t<-1
    # Costruiamo la matrice A 2x2
    A<-matrix(c(1,0,t,1),2,2)
    #Matrice di osservazione
    H<-matrix(c(1,0),1,2)
    # Condizioni iniziali filtro di Kalman
    # Vettore di stato 2x1
    x0_0<-matrix(c(0,0),2,1)
    # Vettore 2x2
    p0_0<-matrix(c(1,0,0,1),2,2)
    #Rumore di processo
    Q<-0.01
    #Rumore di misura
    R<-0.01
    X<-NULL
    X2<-NULL
    pred<-NULL
```

The variables are as follows:

- Y: time series of the measures provided by the civil protection dataset;
- t: time step, discretization step $\Delta t = 1\ day$;
- A: transition matrix initialized as follows $A = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$;
- H: observation matrix initialized as follows $H = [1\ 0]$;
- $X_{0|0}$: state matrix $X_{0|0} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ according to a linear model;
- $P_{0|0}$: state error covariance matrix $P_{0|0} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$;
- Q: process noise matrix $Q = \begin{bmatrix} B_x & 0 \\ 0 & B_v \end{bmatrix}$, where $B_x = 0.01$ is the variance of the total positives and $B_v$ is the variance of the rate of increase / decrease of positives;
- R: measurement noise matrix $R = [B_x]$, where $B_x = 0.01$ is the variance of the total positives.

After initialization we launch the filter, cycling on the measures, and alternating the PREDICTION and UPDATE steps according to the previously defined formulas; therefore, we isolate the estimated state components and at each cycle we remove the previous variables related to two steps before to disallocate the occupied memory and free space by not keeping track of everything.

```
    # Iteriamo per riga sui dati giornalieri
    for (i in 0:nrow(all)) {
        #paste() Concatena i vettori dopo la conversione in caratteri.
        #PREDICTION
        namx <- paste("x", i+1,"_",i, sep = "")
        assign(namx,A%*%get(paste("x", i,"_",i, sep = "")))

        namp <-paste("p", i+1,"_",i, sep = "")
        #assign() Assegna un valore ad una variabile
        # %*% è la moltiplicazione di matrici
        # t(MATRICE) esegue la trasposta
        assign(namp, A%*%(get(paste("p", i,"_",i, sep = "")))%*%t(A)+Q)

        #UPDATE
        namE <- paste("E", i+1, sep = "")
        assign(namE,Y[i+1]-H%*%get(paste("x", i+1,"_",i, sep = "")))

        namk <- paste("k", i+1, sep = "")
        assign(namk,get(paste("p", i+1,"_",i, sep = ""))%*%t(H)%*%(inv(H%*%get(paste("p", i+1,"_",i, sep = ""))%*%t(H)+R)))

        namx2 <- paste("x", i+1,"_",i+1, sep = "")
        assign(namx2,get(paste("x", i+1,"_",i, sep = ""))+get(paste("k", i+1, sep = ""))%*%get(paste("E", i+1, sep = "")))

        namp2 <- paste("p", i+1,"_",i+1, sep = "")
        assign(namp2,(p0_0-get(paste("k", i+1, sep = ""))%*%H)%*%get(paste("p", i+1,"_",i, sep = "")))

        #Creo il vettore X appendando i valori predetti 1° riga di x
        X<-rbind(X,get(paste("x", i+1,"_",i,sep = ""))[1])
        #Creo il vettore X2 appendando i valori predetti 2° riga di x
        X2<-rbind(X2,get(paste("x", i+1,"_",i,sep = ""))[2])
        # rimuovo le variabili create 2 step prima
        if(i>2){
            # remove è usata per rimuovere oggetti creati
            remove(list=(paste("p", i-1,"_",i-2, sep = "")))
            remove(list=(paste("k", i-1, sep = "")))
            remove(list=(paste("E", i-1, sep = "")))
            remove(list=(paste("p", i-2,"_",i-2, sep = "")))
            remove(list=(paste("x", i-1,"_",i-2, sep = "")))
            remove(list=(paste("x", i-2,"_",i-2, sep = "")))
        }
    }
```

We post-process the results including date, region denomination and growth rates, then save the new csv to pass to python:

```
    pred<-NULL
    # Combino i vettori Y, X ed X2 in una matrice
    pred<-cbind(Y,X,round(X2,4))
    pred<-as.data.frame(pred)
    # appendo il nome della regione
    pred$region<-colnames(all[,n+1, drop=FALSE])
    # appendo la data
    pred$date<-date$date
    #Definisco i tassi di crescita o decrescita
    pred$actual<-rbind(0,(cbind(pred[2:nrow(pred),1]-pred[1:nrow(pred)-1,1])/pred[1:nrow(pred)-1,1]))*100
    pred$predict<-rbind(0,(cbind(pred[2:nrow(pred),2]-pred[1:nrow(pred)-1,2])/pred[1:nrow(pred)-1,2]))*100
    pred$pred_rate<-(pred$X/pred$Y-1)*100
    pred$X2_change<-rbind(0,(cbind(pred[2:nrow(pred),3]-pred[1:nrow(pred)-1,3])))
    pred_all<-rbind(pred_all,pred)
}
pred_all<-cbind(pred_all[,4:5],pred_all[,1:3])
names(pred_all)[5]<-"X2"
# ordino i valori secondo alla regione e la data
pred_all=pred_all[with( pred_all, order(region, date)), ]
pred_all<-pred_all[,3:5]

write.csv(pred_all,"/Users/danilogiovannico/Desktop/PROGETTO-Estimation\ and\ Data\ Analysis\ with\ Applications/COVID\ PREDICTIONS/ts_r_KF.csv", row.names = FALSE)
```

Let's fix a package problem for the dataframe:

```
p=pd.read_csv("./ts_r_KF.csv", delimiter = ',')
print(p)
# Uniamo l'output di R a causa di un problema con il pacchetto
t=ts_confirmed
t=t.stack().reset_index(name='confirmed')
t.columns=['date', 'region','confirmed']
t['date']=pd.to_datetime(t['date'] ,errors ='coerce')
t=t.sort_values(['region', 'date'])

# Estraggo le prime 3 colonne
temp=t.iloc[:,:3]
temp=temp.reset_index(drop=True)
for i in range(1,len(t)+1):
    #controllo se le regioni sono diverse
    if(temp.iloc[i,1] is not temp.iloc[i-1,1]):
        # aggiungo la nuova riga
        temp.loc[len(temp)+1] = [temp.iloc[i-1,0]+ pd.DateOffset(1),temp.iloc[i-1,1], 0]
temp=temp.sort_values(['region', 'date'])
p.set_index(temp.index,inplace=True)
#temp=temp.reset_index(drop=True)
temp['Y']=p['Y']
temp['X']=p['X']
temp['X2']=p['X2']
```

```
          Y             X         X2
0         0      0.000000     0.0000
1         0      0.000000     0.0000
2         0      0.000000     0.0000
3         1      0.000000     0.0000
4         1      1.109938     0.7528
...     ...           ...        ...
6505  87385  105243.292191  2249.2175
6506  88842   89370.697001 -7559.5096
6507  90021   85469.042731 -7849.8982
6508  90942   86240.394754 -5349.7204
6509     32   88278.316686 -2767.3479

[6510 rows x 3 columns]
```

Now let's start building the dataframe structure to pass to the matplotlib library for data plotting, and we also include the columns relating to the values and rates of change for 1, 3 and 7 days respectively:

```
t=ts_confirmed
t=t.stack().reset_index(name='confirmed')
t.columns=['date', 'region','confirmed']
t['date']=pd.to_datetime(t['date'] ,errors ='coerce')
t=t.sort_values(['region', 'date'])

# Aggiungo 1 giorno futuro per la previsione
t=t.reset_index(drop=True)
for i in range(1,len(t)+1):
    if(t.iloc[i,1] is not t.iloc[i-1,1]):
        t.loc[len(t)+1] = [t.iloc[i-1,0]+ pd.DateOffset(1),t.iloc[i-1,1], 0]
t=t.sort_values(['region', 'date'])
t=t.reset_index(drop=True)
t.head()
```

| | date | region | confirmed |
|---|---|---|---|
| **0** | 2020-02-24 | Abruzzo | 0 |
| **1** | 2020-02-25 | Abruzzo | 0 |
| **2** | 2020-02-26 | Abruzzo | 0 |
| **3** | 2020-02-27 | Abruzzo | 1 |
| **4** | 2020-02-28 | Abruzzo | 1 |

```
t['1_day_change']=t['3_day_change']=t['7_day_change']=t['1_day_change_rate']=t['3_day_change_rate']=t['7_day_change_rate']=t['last_day']=0
for i in range(1,len(t)):
    #controllo se la riga attuale è differente dalla precedente per nome regione
    if(t.iloc[i,1] is t.iloc[i-2,1]):
        print()
        # setto il valore di cambiamento ad un giorno come differenza dei confermati alla riga attuale - la precedente
        t.iloc[i,3]=t.iloc[i-1,2]-t.iloc[i-2,2]
        # setto 1_day_change_rate in modo analogo
        t.iloc[i,6]=(t.iloc[i-1,2]/t.iloc[i-2,2]-1)*100
        # setto last_day al valore predente dei confermati
        t.iloc[i,9]=t.iloc[i-1,2]
    # Analogamente setto il cambiamento per i 3 giorni
    if(t.iloc[i,1] is t.iloc[i-4,1]):
        t.iloc[i,4]=t.iloc[i-1,2]-t.iloc[i-4,2]
        t.iloc[i,7]=(t.iloc[i-1,2]/t.iloc[i-4,2]-1)*100
    # Analogamente setto il cambiamento per i 7 giorni
    if(t.iloc[i,1] is t.iloc[i-8,1]):
        t.iloc[i,5]=t.iloc[i-1,2]-t.iloc[i-8,2]
        t.iloc[i,8]=(t.iloc[i-1,2]/t.iloc[i-8,2]-1)*100

# Setto a zero i valori NaN
t=t.fillna(0)
# Eseguo il merge con i valori predetti dal filtro di kalman
t=t.merge(temp[['date','region', 'X']],how='left',on=['date','region'])
t=t.rename(columns = {'X':'kalman_prediction'})
t=t.replace([np.inf, -np.inf], 0)
t['kalman_prediction']=round(t['kalman_prediction'])
train=t.merge(confirmed_df[['region','population']],how='left',on='region')
train=train.rename(columns = {'Population':'population'})
# train['population']=train['population'].str.replace(r" ", '')
# train['population']=train['population'].str.replace(r",", '')
train['population']=train['population'].fillna(1)
train['population']=train['population'].astype('int32')
train['infected_rate'] =train['last_day']/train['population']*10000
#train=train.merge(w,how='left',on=['date','region'])
train=train.sort_values(['region', 'date'])
### Compiliamo i valori mancanti
for i in range(0,len(train)):
    if(np.isnan(train.iloc[i,12])):
        if(train.iloc[i,1] is train.iloc[i-1,1]):
            train.iloc[i,10]=train.iloc[i-1,10]
            train.iloc[i,12]=train.iloc[i-1,12]
```

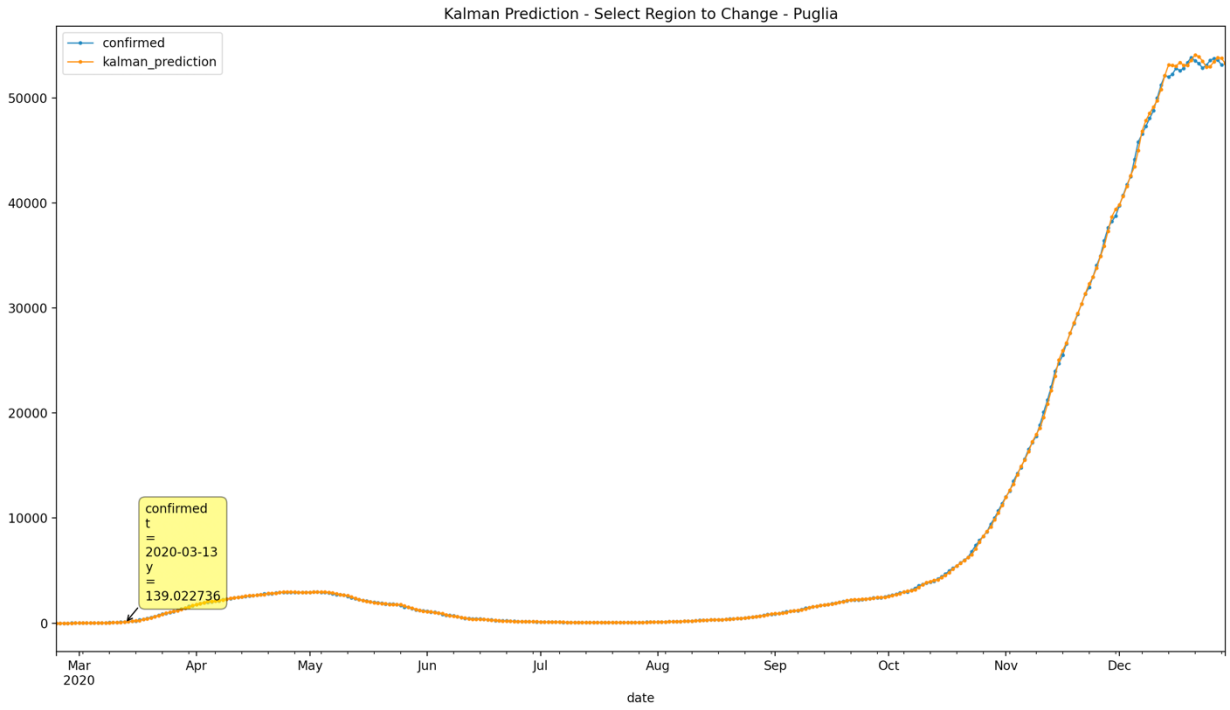
Now we are ready for the plot, we select the region of interest and we evaluate different statistical indices (these metrics are described in the **Appendix** respectively: **MSE [1], RMSE [2], MAE [3]**) to understand the goodness of the forecast obtained with respect the real values provided by civil protection; finally we display the prediction table and the relative plot for the 1 day prediction:

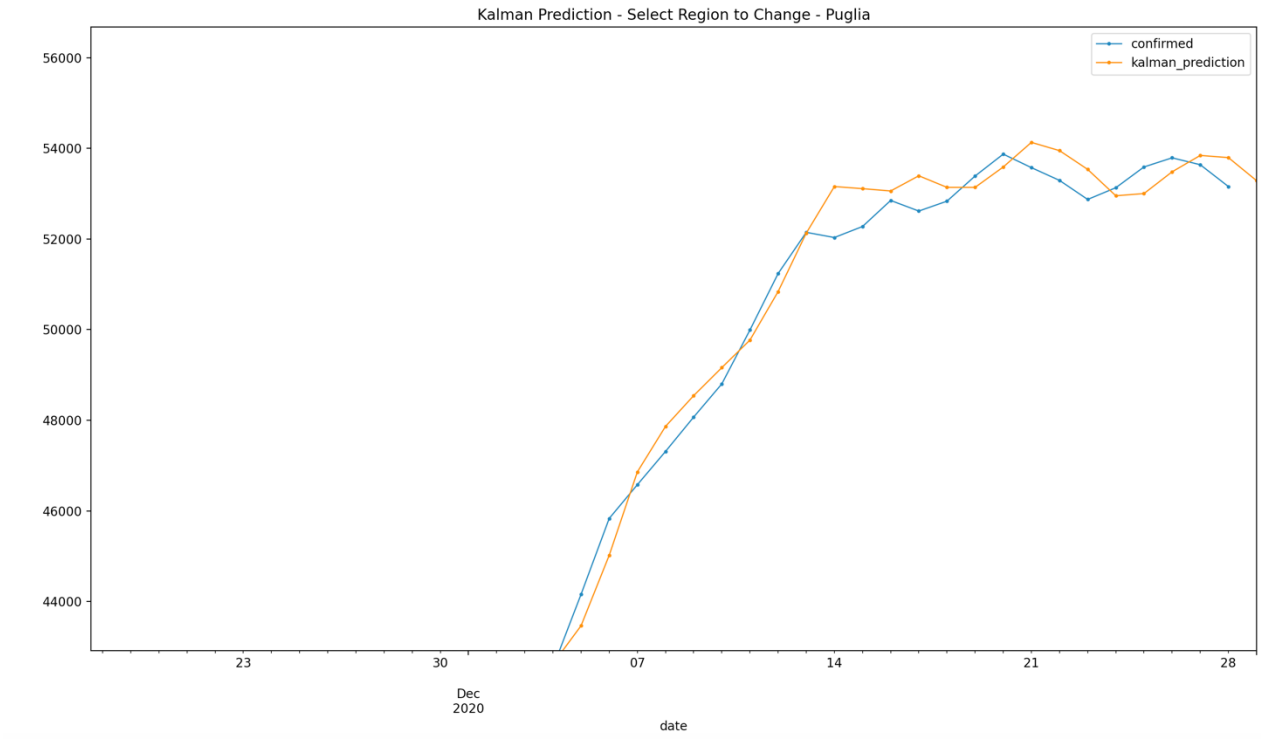
```
# Select region
region='Puglia'

#Eseguiamo una valutazione sulle metriche MSE, RMSE e MAE sui valori predetti
evaluation=pd.DataFrame(columns=['region','mse','rmse','mae'])
place=0
for i in range(1,len(t)):
    if(t.iloc[i,1] is not t.iloc[i-1,1]):
        # array valori predetti
        ex=np.array(t.iloc[i-len(ts_confirmed):i,10])
        # array valori confermati
        pred=np.array(t.iloc[i-len(ts_confirmed):i,2])
        evaluation=evaluation.append({'region': t.iloc[i-1,1], 'mse': np.power((ex - pred),2).mean(),'rmse':sqrt(mean_squared_error(ex,pred)),'mae': (abs(ex - pred)).mean()}, ignore_index=True)
t.head()
p=t[t['region']==region][['date','region','confirmed','kalman_prediction']]
# p=p.rename(columns = {'confirmed':'recoverd'})
p.iloc[len(p)-1,2]=None
p=p.set_index(['date'])
# Plottiamo valori reali e valori con relativa previsione di kalman
p.iloc[:,1:].plot(marker='o',linewidth=1,markersize=2,figsize=(16,8)).set_title('Kalman Prediction - Select Region to Change - {}'.format(p.iloc[0,0]))
mplcursors.cursor(hover=True)
print(evaluation[evaluation['region']==p.iloc[0,0]])
# print(evaluation)
p=t[t['region']==region][['date','region','confirmed','kalman_prediction']]
p.tail(10)
```

| | region | mse | rmse | mae |
|---|---|---|---|---|
| 14 | Puglia | 9.222937e+06 | 3036.928887 | 257.498382 |

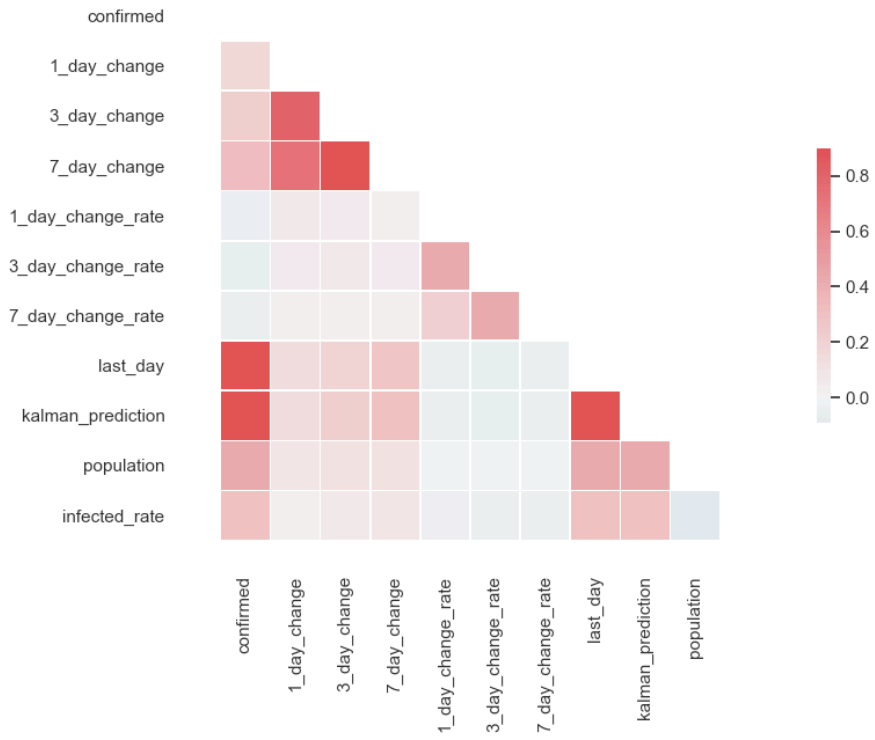
| | date | region | confirmed | kalman_prediction |
|---|---|---|---|---|
| **4640** | 2020-12-20 | Puglia | 53872 | 53588.0 |
| **4641** | 2020-12-21 | Puglia | 53574 | 54131.0 |
| **4642** | 2020-12-22 | Puglia | 53292 | 53949.0 |
| **4643** | 2020-12-23 | Puglia | 52872 | 53535.0 |
| **4644** | 2020-12-24 | Puglia | 53131 | 52953.0 |
| **4645** | 2020-12-25 | Puglia | 53589 | 53002.0 |
| **4646** | 2020-12-26 | Puglia | 53790 | 53482.0 |
| **4647** | 2020-12-27 | Puglia | 53638 | 53843.0 |
| **4648** | 2020-12-28 | Puglia | 53157 | 53795.0 |
| **4649** | 2020-12-29 | Puglia | 0 | 53287.0 |

The plot obtained for Puglia is the following:






After plotting the one-day prediction, we can plot the correlation matrix of the variables (this matrix will provide values between 0 and 1; respectively 0 means total correlation, while 1 total correlation):



From this matrix we can note:

• A high correlation between confirmed cases with the prediction obtained through KF and the predicted value on the previous day (last_day);

• A high correlation between the change noted for 1, 3 and 7 days respectively.

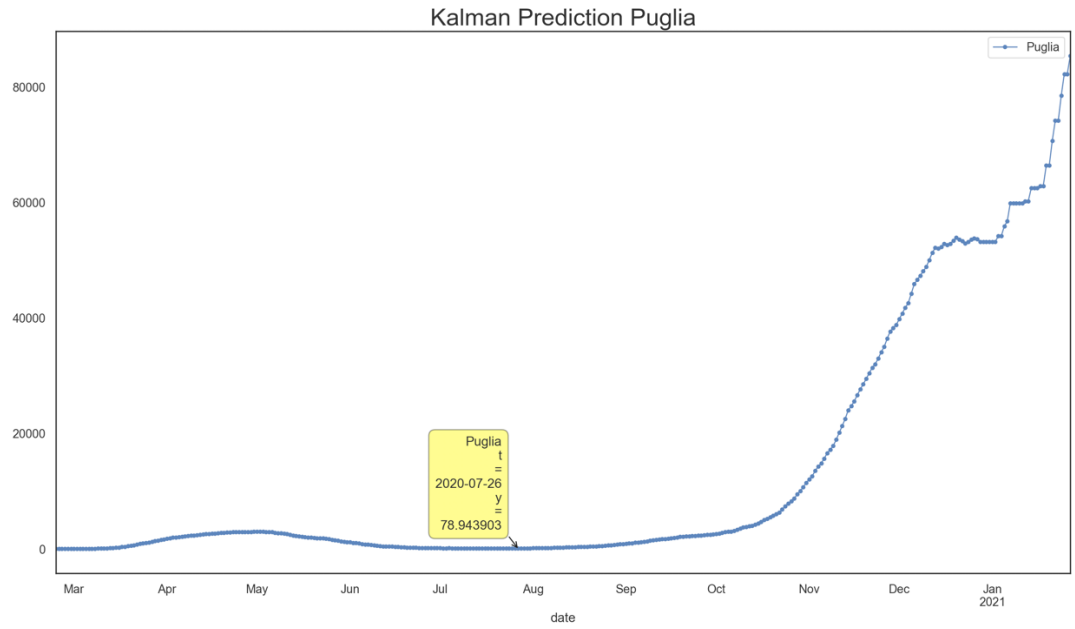
*N-day prediction*

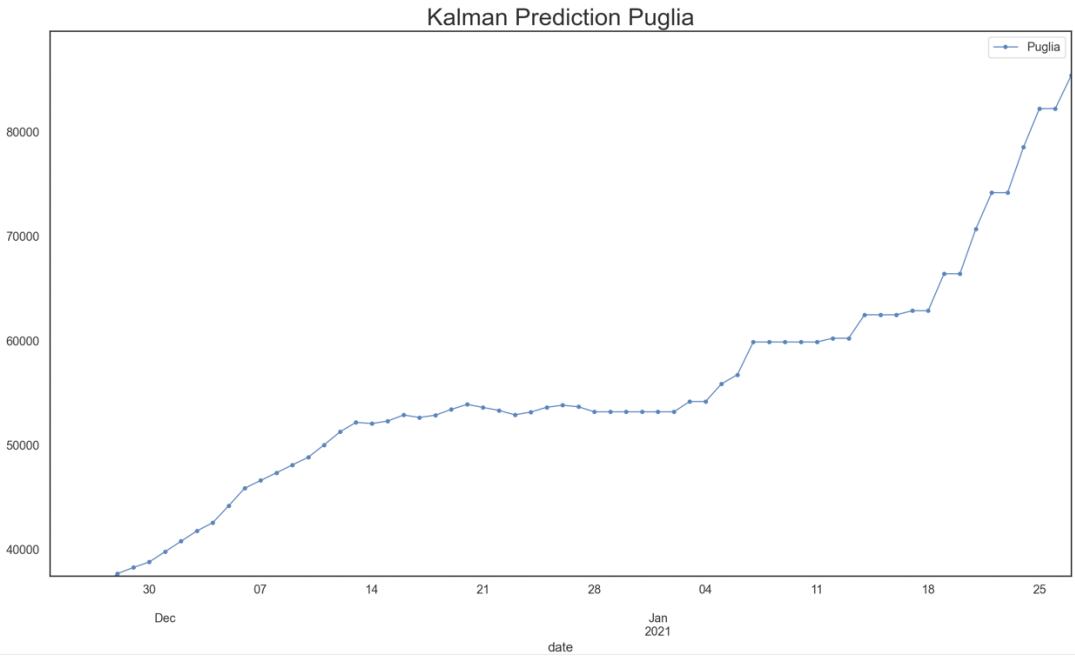
We then reinterpreted the prediction by extending it forward up to 30 days, using the last aforementioned measure from time to time, the corresponding code used is the following:

```
%%R
library(pracma)
library(Metrics)
library(readr)
library(reshape)
all<- read.csv("/Users/danilogiovannico/Desktop/PROGETTO-Estimation\ and\ Data\ Analysis\ with\ Applications/COVID\ PREDICTIONS/ts_r.csv")
all$X1<-NULL
# Imposto i giorni di previsione
for (i in 1:30) {
    if( i>1) {
        all<-all_new
    }
    date<-all[,1, drop=FALSE]
    #date[nrow(date) + 1,1] <- format(as.Date(all[nrow(all),1])+1)
    date<-rbind(date, data.frame(date = format(as.Date(all[nrow(all),1])+1)))
    pred_all<-NULL
    for (n in 2:ncol(all)-1) {
        Y<-ts(data = all[n+1], start = 1, end =nrow(all)+1)
        t<-1
        A<-matrix(c(1,0,t,1),2,2)
        H<-matrix(c(1,0),1,2)
        #Kalman
        x0_0<-matrix(c(0,0),2,1)
        p0_0<-matrix(c(1,0,0,1),2,2)
        Q<-0.01
        R<-0.01
        X<-NULL
        X2<-NULL
        pred<-NULL

    for (i in 0:nrow(all)) {
        #PREDICTION
        namx <- paste("x", i+1,"_",i, sep = "")
        assign(namx,A%*%get(paste("x", i,"_",i, sep = "")))
        namp <-paste("p", i+1,"_",i, sep = "")
        assign(namp, A%*%(get(paste("p", i,"_",i, sep = "")))%*%t(A)+Q)
        #UPDATE
        namE <- paste("E", i+1, sep = "")
        assign(namE,Y[i+1]-H%*%get(paste("x", i+1,"_",i, sep = "")))
        namk <- paste("k", i+1, sep = "")
        assign(namk,get(paste("p", i+1,"_",i, sep = ""))%*%t(H)%*%(inv(H%*%get(paste("p", i+1,"_",i, sep = ""))%*%t(H)+R)))
        namx2 <- paste("x", i+1,"_",i+1, sep = "")
        assign(namx2,get(paste("x", i+1,"_",i, sep = ""))+get(paste("k", i+1, sep = ""))%*%get(paste("E", i+1, sep = "")))
        namp2 <- paste("p", i+1,"_",i+1, sep = "")
        assign(namp2,(p0_0-get(paste("k", i+1, sep = ""))%*%H)%*%get(paste("p", i+1,"_",i, sep = "")))
        X<-rbind(X,get(paste("x", i+1,"_",i,sep = ""))[1])
        X2<-rbind(X2,get(paste("x", i+1,"_",i,sep = ""))[2])
        if(i>2){
            remove(list=(paste("p", i-1,"_",i-2, sep = "")))
            remove(list=(paste("k", i-1, sep = "")))
            remove(list=(paste("E", i-1, sep = "")))
            remove(list=(paste("p", i-2,"_",i-2, sep = "")))
            remove(list=(paste("x", i-1,"_",i-2, sep = "")))
            remove(list=(paste("x", i-2,"_",i-2, sep = "")))}
        }
        pred<-NULL
        pred<-cbind(Y,X,round(X2,4))
        pred<-as.data.frame(pred)
        pred$region<-colnames(all[,n+1, drop=FALSE])
        pred$date<-date$date
        pred$actual<-rbind(0,(cbind(pred[2:nrow(pred),1]-pred[1:nrow(pred)-1,1])/pred[1:nrow(pred)-1,1]))*100
        pred$predict<-rbind(0,(cbind(pred[2:nrow(pred),2]-pred[1:nrow(pred)-1,2])/pred[1:nrow(pred)-1,2]))*100
        pred$pred_rate<-(pred$X/pred$Y-1)*100
        pred$X2_change<-rbind(0,(cbind(pred[2:nrow(pred),3]-pred[1:nrow(pred)-1,3])))
        pred_all<-rbind(pred_all,pred)
    }
```

```
    pred_all<-cbind(pred_all[,4:5],pred_all[,1:3])
    names(pred_all)[5]<-"X2"
    pred_all<-pred_all[,1:5]
    pred_all_today=pred_all[with( pred_all, order(region, date)), ]
    all_new=all
    #all_new[nrow(all_new),1]<-all_new[nrow(all),1]+1
    # iteriamo modificando la data per la previsione
    temp<-with(pred_all_today, pred_all_today[date == format(as.Date(all[nrow(all),1,])+1), ])
    temp<-cbind(temp[,1:2],temp[,4])
    temp2<-reshape(temp, direction = "wide", idvar = "date", timevar = "region")
    # runif fornisce informazioni sulla distribuzione uniforme sull'intervallo da min a max
    rand_num<-runif(ncol(temp2)-1, 0.9, 1.05)
    temp2[,2:ncol(temp2)]<-temp2[,2:ncol(temp2)]*rand_num
    colnames(temp2)=colnames(all_new)
    all_new<-rbind(all_new,temp2)
    all_new[,2:ncol(all_new)]<-round(all_new[,2:ncol(all_new)])
    for (i in 2:ncol(all_new)) {
        #Assegno il valore massimo tra il valore al tempo t e quello a t-1
        all_new[nrow(all_new),i]=max(all_new[nrow(all_new)-1,i],all_new[nrow(all_new),i])
    }
}
write.csv(all_new,"/Users/danilogiovannico/Desktop/PROGETTO-Estimation\ and\ Data\ Analysis\ with\ Applications/COVID\ PREDICTIONS/ts_r_KF_ahead.csv", row.names = FALSE)
```

The plot obtained and the corresponding prediction table at one month are as follows:





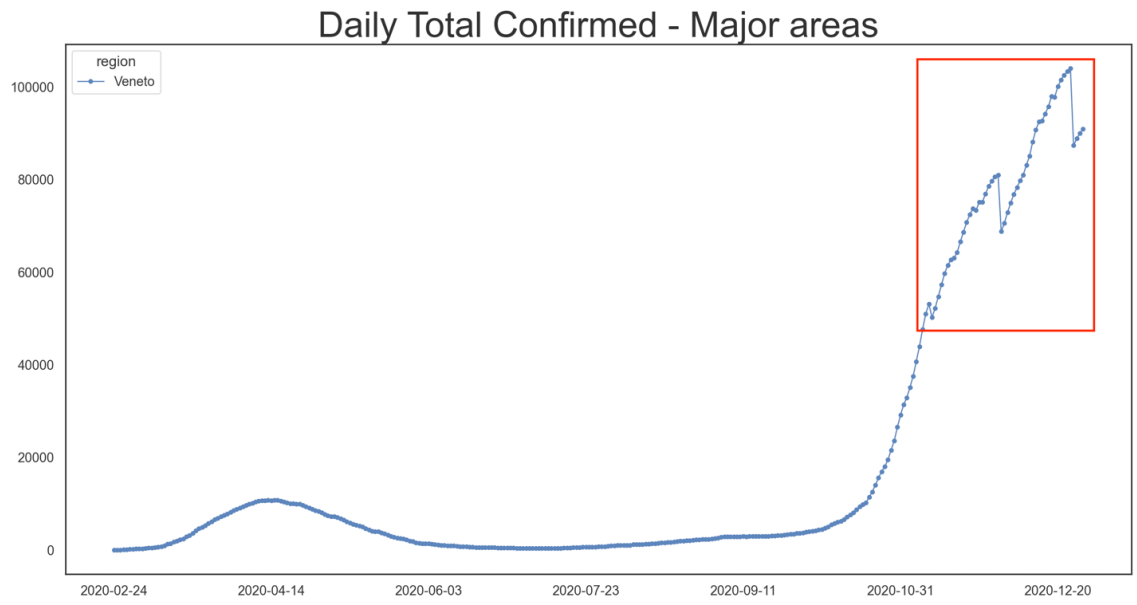

| | Puglia |
|---|---|
| **date** | |
| **2020-12-29** | 53157 |
| **2020-12-30** | 53157 |
| **2020-12-31** | 53157 |
| **2021-01-01** | 53157 |
| **2021-01-02** | 53157 |
| **2021-01-03** | 54143 |
| **2021-01-04** | 54143 |
| **2021-01-05** | 55834 |
| **2021-01-06** | 56715 |
| **2021-01-07** | 59845 |
| **2021-01-08** | 59845 |
| **2021-01-09** | 59845 |
| **2021-01-10** | 59845 |
| **2021-01-11** | 59845 |
| **2021-01-12** | 60204 |
| **2021-01-13** | 60204 |
| **2021-01-14** | 62452 |
| **2021-01-15** | 62452 |
| **2021-01-16** | 62452 |
| **2021-01-17** | 62840 |
| **2021-01-18** | 62840 |
| **2021-01-19** | 66381 |
| **2021-01-20** | 66381 |
| **2021-01-21** | 70677 |
| **2021-01-22** | 74148 |
| **2021-01-23** | 74148 |
| **2021-01-24** | 78512 |
| **2021-01-25** | 82203 |
| **2021-01-26** | 82203 |
| **2021-01-27** | 85371 |

*Conclusion*

The KF proves to be an excellent ally for the forecast of short-term spreads with excellent results, this is because the filter is able to promptly identify the rapid changes on the trend of the curve and will readjust the updated forecast. In the long run, however, the prediction is more challenging, in any case it provides an approximate future trend of what may happen. Therefore, this tool is useful in understanding when the peak will be reached and when the decrease phase will begin, also providing excellent forecast values in the short term.

## 2.2 SHORT-TERM FORECAST MODEL BASED ON A ROBUST VERSION TO THE OUTLIERS OF THE KALMAN FILTER

Here we will present a robust version of the Kalman Filter assuming the measurements are corrupted by outliers. For the study in question we will take the Veneto region which has several outliers in the months of November, December and January, represented by instantaneous peaks.




The idea behind this approach for estimating the robust state is based on the minimization of a suitably defined non-linear cost function. This cost function is composed of the sum of two terms: a quadratic term of the prediction based on the model (as in simple KF) and a nonlinear function of the output residuals, which is called the minimum cost of entropy (LEL). This residual cost function is comparable to the one used for M-Estimators, also used to perform robust estimates in the presence of outliers; functions like Huber's could be applied but could cause a loss of information due to their nature in saturating values.

***System model, state estimators based on Kalman and M-Estimator***

The state estimation problem, as already seen for the simple Kalman filter, consists in determining the state vector $x_k$ given the knowledge of the model matrices (including the noise covariance), of the inputs and outputs up to time t. Of course, supposing that the noises $w_k$ and $v_k$ (of process and observation) are zero Gaussian white and uncorrelated, and that we are in adequate observability conditions, then the optimal solution is represented by the Kalman filter.

However, we can think of the Kalman filter as the solution to a weighted LS (Least Square)

problem: in particular it can be derived by minimizing the following objective solution consisting of two terms, the first associated with the dynamics of the model and the second associated with the observations relating to KF estimate and covariance:

$$\hat{\mathbf{x}}_k^+ = \arg\min_{\mathbf{x}_k} J_{KF}$$

$$J_{KF} = \underbrace{\frac{1}{2}\left((\mathbf{x}_k - \hat{\mathbf{x}}_k^-)^\top (P_k^-)^{-1}(\mathbf{x}_k - \hat{\mathbf{x}}_k^-)\right)}_{J_{\text{dynamical model}}} + \underbrace{\frac{1}{2}\left(\mathbf{r}_k^\top R_k^{-1}\mathbf{r}_k\right)}_{J_{\text{observations}}}$$

$$\mathbf{r}_k = \mathbf{y}_k - C_k\mathbf{x}_k$$

To make the $J_{KF}$ function robust to outliers we use the M-Estimator methodology where we replace the term $J_{Observations}$ with a function, eg. Huber, Winsorized etc. The idea behind this method is that the cost functions of the M-Estimators are designed to grow more slowly than the quadratic term $J_{Observations}$ for residuals that are normally large enough. To obtain the KF structure, the solution is obtained by approximating the M-Estimator functions with a weighted LS term; robustness is sought by giving a finite weight to the individual residues that exceed a threshold. Most methods use a structure:

$$J_{Observations} = \sum_{i=1}^{k} \rho(r_i) \quad ove\ r_i = y_i - \hat{y}_i$$

$\rho(r_i)$ is a symmetric function with a single minimum in zero (as already mentioned $\rho$ can be of different types Huber, Winsorized). This approach is good and achieves good performance, but the break down point is still limited by a function that reduces the parameter to be estimated.

The estimator that we will see, on the other hand, is derived by minimizing a cost function having the same structure as that seen, given by the sum of $J_{Observations}$ and $J_{Dynamical\ Model}$, but with the difference that the term $J_{Observations}$ is taken from a robust identification algorithm of static parameters called LEL. The LEL function is designed with the aim of reducing most of the residuals to the square at the expense of a minority of them. This approach is close to the LMS one but with the additional advantage that the cost of the LEL can be numerically minimized by evaluating its gradient and Hessian; the limit that could be had is if we were in a local minimum instead of the global, in which case the estimate would be suboptimal.

***Robust state estimation at outliers by LEL algorithm***

Let's now see how the cost varies with respect to the case seen for the LEL. Cost $J_k$ can be rewritten as:

$$J_k = \underbrace{\frac{1}{2}\left((\mathbf{x}_k - \hat{\mathbf{x}}_k^-)^T (P_k^-)^{-1}(\mathbf{x}_k - \hat{\mathbf{x}}_k^-)\right)}_{J_{\text{dynamical model}}} + \underbrace{\alpha H_k(\mathbf{r}_1, \ldots, \mathbf{r}_k)}_{J_{\text{LEL}}}$$

where $r_i = y_i - C_i\hat{x}_i^+, i = 1, \ldots, k$ denotes the i-th residue and $H_k(\cdot)$ represents a loss function inspired by entropy. We define $D_k$ as the minimum square cost:

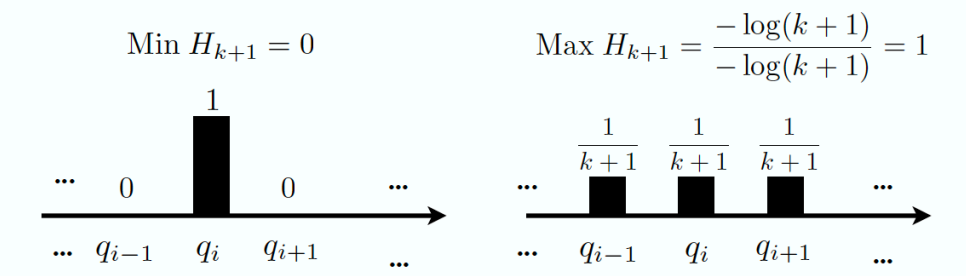
$$D_k = \sum_{j=1}^{k} \|\mathbf{r}_j\|^2 = \sum_{j=1}^{k} \mathbf{r}_j^\top \mathbf{r}_j$$

and its residual squared $q_i$ as,

$$\text{if } D_k \neq 0 \Rightarrow q_i := \frac{\|\mathbf{r}_i\|^2}{\sum_{j=1}^{k} \|\mathbf{r}_j\|^2} \quad : q_i \in [0, 1] \quad \text{and} \quad \sum_{i=1}^{k} q_i = 1,$$

the LEL loss function has the following form:

$$H_k = \begin{cases} 0 & \text{if } D_k = 0 \\ -\frac{1}{\log k} \sum_{i=1}^{k} q_i \log q_i & \text{otherwise,} \end{cases}$$




The purpose of this function is to give a global measure of the squared dispersion of the relative residuals. The idea behind the estimator is to make the relative residuals squared $q_i$ as less evenly distributed as possible; in this case, most of the residuals are small (compared to the constant $D_k$ normalization, i.e. the LS cost) and few are large (outliers). Technically, the goal is to make the majority of the relative squared residuals as small as possible to the detriment of the large ones and then minimize $H_k$ defined as Gibbs entropy for a set of discrete probabilities $p_i$. Consequently, minimizing the cost of LEL, $H_k$, is equivalent to finding the minimum entropy distribution of a set of discrete probabilities: the absolute minimum of the entropy of a set of discrete probabilities is obtained when all probabilities are zero except one. It should also be noted that, while the term $J_{\text{Dynamical Model}}$ does not have an upper limit, the term $H_k$ is limited in the interval [0, 1] due to the 1 / log k normalization factor. Consequently, the constant parameter $\alpha$ in the $J_{\text{LEL}}$ contribution is to be considered as a tuning gain that makes the two terms $J_{\text{Dynamical Model}}$ and $J_{\text{LEL}}$ comparable. In the absence of an adequate choice of $\alpha$, the overall cost $J_k$ could be dominated by only one of the two terms (if it were very small, the minimization of $J_k$ would correspond only to the prediction step).

Now we will see how to obtain the solution in closed form but before defining it we

observe that H (·) depends on the residuals {$r_1$, ..., $r_k$} (each residual $r_i$ in the set is a function of the state estimate $\hat{x}_i^+$ that is a quantity fixed for all i <k) and depends only on $x_k$; therefore by compactness notation we will write x→$x_k$, hence $H_k(x)$. Let us now pass to the computation of the closed form to the minimization problem, obtained by approximating $H_k(x)$ in a neighborhood of $\hat{x}_{k-1}^+$ with a quadratic function through 2° order Taylor expansion.

$$H_k(\mathbf{x}) = H_k(\hat{\mathbf{x}}_{k-1}^+) + \nabla_{\mathbf{x}} H_k(\hat{\mathbf{x}}_{k-1}^+)^\top (\mathbf{x} - \hat{\mathbf{x}}_{k-1}^+) + \\ + \frac{1}{2}(\mathbf{x} - \hat{\mathbf{x}}_{k-1}^+)^\top \mathcal{H}\left[H_k(\hat{\mathbf{x}}_{k-1}^+)\right](\mathbf{x} - \hat{\mathbf{x}}_{k-1}^+) + o\left(\|\mathbf{x} - \hat{\mathbf{x}}_{k-1}^+\|^2\right)$$

where the gradient and the Hessian $\mathcal{H}(\cdot)$ of the LEL cost function calculated around the point $\hat{x}_{k-1}^+$ are:

$$\nabla_{\mathbf{x}} H_k(\cdot)|_{\mathbf{x}=\hat{\mathbf{x}}_{k-1}^+} = \nabla_{\mathbf{x}} H_k\left(\hat{\mathbf{x}}_{k-1}^+\right)$$

$$\mathcal{H}[H_k(\cdot)]|_{\mathbf{x}=\hat{\mathbf{x}}_{k-1}^+} = \mathcal{H}\left[H_{k-1}\left(\hat{\mathbf{x}}_{k-1}^+\right)\right]$$

By direct calculation we obtain:

$$\nabla_{\mathbf{x}} H_k(\mathbf{x}_k) = \begin{cases} \mathbf{0} & \text{if } D_k = 0 \\ \frac{2}{D_k(\mathbf{x}_k)\log k}\left(\log\|\mathbf{r}_k(\mathbf{x}_k)\|^2 - \frac{S_k(\mathbf{x}_k)}{D_k(\mathbf{x}_k)}\right) C_k^\top \mathbf{r}_k(\mathbf{x}_k) & \text{otherwise} \end{cases}$$

$$\mathcal{H}[H_k(\mathbf{x}_k)] = \begin{cases} 0_{n\times n} & \text{if } D_k = 0 \\ \frac{2}{D_k^2(\mathbf{x}_k)\log k}\Big[2\, C_k^\top C_k\, \|\mathbf{r}_k(\mathbf{x}_k)\|^2 \cdot \\ \left(2\log\|\mathbf{r}_k(\mathbf{x}_k)\|^2 - 2\frac{S_k(\mathbf{x}_k)}{D_k(\mathbf{x}_k)} + 1\right) + \\ -C_k^\top C_k\left(D_k(\mathbf{x}_k)\log\|\mathbf{r}_k(\mathbf{x}_k)\|^2 - S_k(\mathbf{x}_k) + 2\,D_k(\mathbf{x}_k)\right)\Big] & \text{otherwise} \end{cases}$$

where $S_k$ is:

$$S_k(\mathbf{x}_k) = \left(\sum_{j=1}^{k-1}\left\|\mathbf{r}_j\left(\hat{\mathbf{x}}_j^+\right)\right\|^2 \log\left\|\mathbf{r}_j\left(\hat{\mathbf{x}}_j^+\right)\right\|^2\right) + \|\mathbf{r}_k(\mathbf{x}_k)\|^2 \log\|\mathbf{r}_k(\mathbf{x}_k)\|^2.$$

Assuming the limited and well-defined Hessian, the function $J_k$ becomes:

$$J_k = \frac{1}{2}(\mathbf{x}_k - \hat{\mathbf{x}}_k^-)^\top \left(P_k^-\right)^{-1}(\mathbf{x}_k - \hat{\mathbf{x}}_k^-) \\ + \alpha\left(H_k(\hat{\mathbf{x}}_k^+) + \nabla_{\mathbf{x}} H_k(\hat{\mathbf{x}}_k^+)^\top(\mathbf{x}_k - \hat{\mathbf{x}}_k^+) + \frac{1}{2}(\mathbf{x}_k - \hat{\mathbf{x}}_k^+)^\top \mathcal{H}[H_k(\hat{\mathbf{x}}_k^+)](\mathbf{x}_k - \hat{\mathbf{x}}_k^+)\right) \\ + o\left(\|\mathbf{x} - \hat{\mathbf{x}}_{k-1}^+\|^2\right)$$

Setting the gradient of $J_k$ with respect to $x_k$ equal to zero, the filter equations will be:

$$\hat{\mathbf{x}}_k^- = A_{k-1}\hat{\mathbf{x}}_{k-1}^+ + B_{k-1}\mathbf{u}_{k-1}$$

$$P_k^- = A_{k-1}P_{k-1}^+ A_{k-1}^\top + Q_{k-1}$$

$$K_k = \left((P_k^-)^{-1} + \alpha\mathcal{H}[H_k(\hat{\mathbf{x}}_{k-1}^+)]\right)^{-1} \alpha\mathcal{H}[H_k(\hat{\mathbf{x}}_{k-1}^+)]$$

$$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + K_k(\hat{\mathbf{x}}_{k-1}^+ - \hat{\mathbf{x}}_k^-) - \left((P_k^-)^{-1} + \alpha\mathcal{H}[H_k(\hat{\mathbf{x}}_{k-1}^+)]\right)^{-1} \alpha\nabla_{\mathbf{x}} H_k(\hat{\mathbf{x}}_{k-1}^+)$$

$$P_k^+ = \left((P_k^-)^{-1} + \alpha\mathcal{H}[H_k(\hat{\mathbf{x}}_{k-1}^+)]\right)^{-1}$$

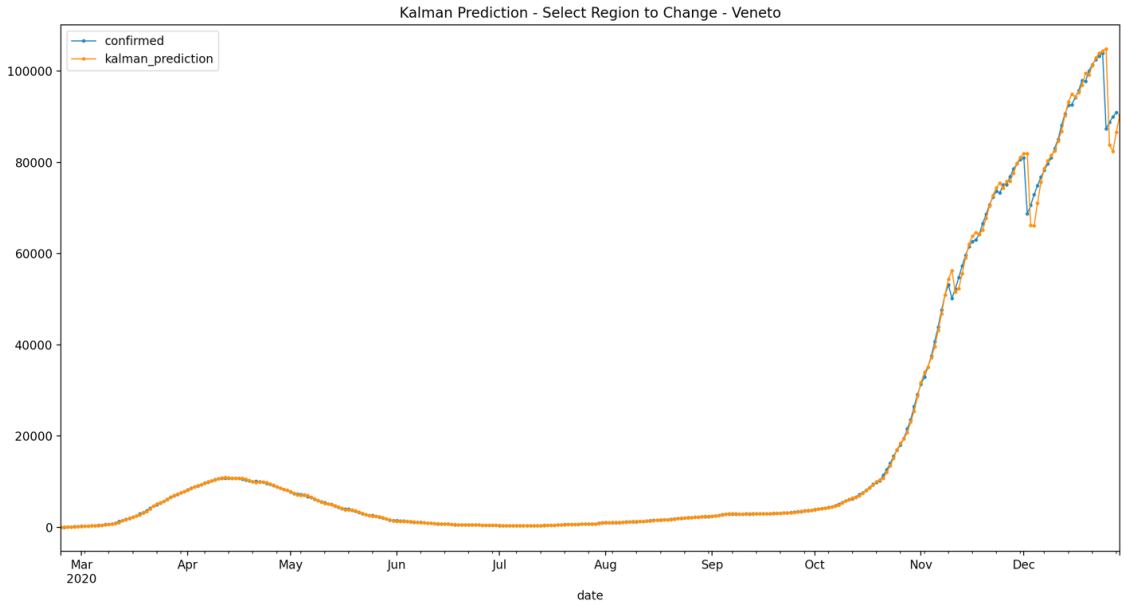
Where $(P_k^-)^{-1} + \alpha\mathcal{H}[(\hat{x}_{k-1}^+)] = (P_k^+)^{-1}$ is the Hessian of cost $J_k$.

For further information see 8 in the Bibliography section.

**Implementation of the Robust KF (LEL) with Python and R on the COVID case for the prediction of Spreads**

Here we will see how the code was implemented. The initial preprocessing phases follow the cases described above; the model used is always the same as in the previous cases for the prediction of spreads, therefore linear but with the presence of outliers in the measurements provided.

Let's see first the estimate obtained with a simple KF in order to denote the inefficiency in the presence of outliers:




It is visible how in the estimate the filter faithfully follows what is the measurement by virtue of the hypotheses on KF, that is, that process noise and measurement are independent white Gaussian stochastic variables with zero mean. In the presence of outliers such hypotheses are lacking and it is convenient, as already said, to switch to a

robust version, such as that of the M-Estimators or LEL. The case study in question implements the LEL through R code, below we show the pseudo-code that will be implemented:

**Require:** $\hat{\mathbf{x}}_{k-1}^{+}, P_{k-1}^{+}, \mathbf{y}_k, \|\mathbf{r}_{k-N+1}(\hat{\mathbf{x}}_{k-N+1}^{+})\|^2, \ldots, \|\mathbf{r}_{k-1}(\hat{\mathbf{x}}_{k-1}^{+})\|^2$
**Ensure:** $\hat{\mathbf{x}}_{k}^{+}, P_{k}^{+}$
$\hat{\mathbf{x}}_{k}^{-} \leftarrow A_{k-1}\hat{\mathbf{x}}_{k-1}^{+} + B_{k-1}\mathbf{u}_{k-1}$ ▷ Predict
$P_{k}^{-} \leftarrow A_{k-1}P_{k-1}^{+}A_{k-1}^{\top} + Q_{k-1}$
$\|\mathbf{r}_k(\hat{\mathbf{x}}_{k-1}^{+})\| \leftarrow \|\mathbf{y}_k - C_k\,\hat{\mathbf{x}}_{k-1}^{+}\|$
$j* \leftarrow \arg\max_j \{\|\mathbf{r}_j\| \;:\; j \in [k-N+1, k]\}$
$\mathbf{r}_{j*} \leftarrow \mathbf{r}_{j*}\beta$
$D_k \leftarrow \left(\sum_{j=k-N+1}^{k-1} \|\mathbf{r}_j(\hat{\mathbf{x}}_j^{+})\|^2\right) + \|\mathbf{r}_k(\hat{\mathbf{x}}_{k-1}^{+})\|^2$ ▷ Update
$S_k \leftarrow \left(\sum_{j=k-N+1}^{k-1} \|\mathbf{r}_j(\hat{\mathbf{x}}_j^{+})\|^2 \log\|\mathbf{r}_j(\hat{\mathbf{x}}_j^{+})\|^2\right) + \|\mathbf{r}_k(\hat{\mathbf{x}}_{k-1}^{+})\|^2 \log\|\mathbf{r}_k(\hat{\mathbf{x}}_{k-1}^{+})\|^2$
Compute $\nabla H_k(\hat{x}_{k-1}^{+})$ and $\mathcal{H}(H_k(\hat{x}_{k-1}^{+}))$ using (25–26)
**if** $\mathcal{H}(H_k(\hat{x}_{k-1}^{+})) > 0$ **then**
    $K_k \leftarrow ((P_k^{-})^{-1} + \alpha\mathcal{H}(H_k(\hat{\mathbf{x}}_{k-1}^{+}))^{-1}\alpha\mathcal{H}(H_k(\hat{\mathbf{x}}_{k-1}^{+}))$
    $P_k^{+} \leftarrow (P_k^{-1} + \alpha\mathcal{H}(H_k(\hat{\mathbf{x}}_{k-1}^{+}))^{-1}$
    $\hat{\mathbf{x}}_k^{+} \leftarrow \hat{\mathbf{x}}_k^{-} + K(\hat{\mathbf{x}}_{k-1}^{+} - \hat{\mathbf{x}}_k^{-}) - \alpha P_k^{+}\nabla H_k(\hat{\mathbf{x}}_{k-1}^{+})$
**else**
    $P_k^{+} \leftarrow P_k^{-}$
    $\hat{\mathbf{x}}_k^{+} \leftarrow \hat{\mathbf{x}}_k^{-}$
**end if**
$\mathbf{r}_{j*} \leftarrow \mathbf{r}_{j*}/\beta$
**return** $P_k^{+}, \hat{\mathbf{x}}_k^{+}$

The idea on the case study is to initially use the simple KF on the variation of positives and then move on to the LEL for the months of November, December and January. The initial parameter tuning is as follows:

- Y: time series of the measures provided by the civil protection dataset;
- t: time step, discretization step $\Delta t = 1\ giorno$;
- A: transition matrix initialized as follows $A = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$;
- H: observation matrix initialized as follows $H = [1\ 0]$;
- $X_{0|0}$: state matrix $X_{0|0} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ according to a linear model;
- $P_{0|0}$: state error covariance matrix $P_{0|0} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$;
- Q: process noise matrix $Q = \begin{bmatrix} B_x & 0 \\ 0 & B_v \end{bmatrix}$, where $B_x = 0.01$ is the variance of the total positives and $B_v$ is the variance of the rate of increase / decrease of positives;
- R: measurement noise matrix $R = [B_x]$, where $B_x = 0.01$ is the variance of the total positives;
- N: time window equal to 30 for the finished memory implementation. This is to overcome the problem that $D_k$ can decrease indefinitely, with the advantage that the filter forgets the old measures in favor of the new ones;

- SMALL: equal to 1e-10. Value used to regularize the null residuals, this is because, if $||\ r_k\ ||$ = 0 (a causa del termine log $||\ r_k\ ||^2$) then the Hessian matrix would be misplaced;
- alpha: weight of the LEL cost functional initially set equal to $H^T R^{-1} H$.

Below is the development of the code in R language on which we will not dwell much having provided the pseudo-code:

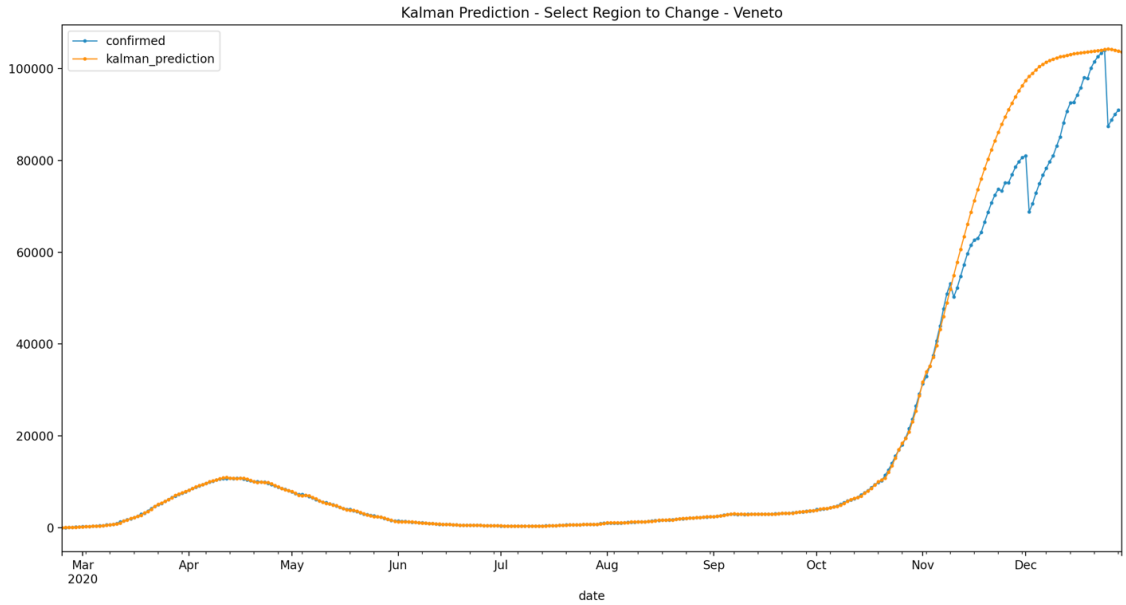
```
%%R
library(pracma)
library(Metrics)
library(readr)
library(matrixStats)
all<- read.csv("/Users/danilogiovannico/Desktop/PROGETTO-Estimation\ and\ Data\ Analysis\ with\ Applications/COVID\ PREDICTIONS/ts_r.csv")
all$X1<-NULL
date<-all[,1, drop=FALSE]
# La funzione rbind () combina vettori, matrici o frame di dati per righe
date<-rbind(date, data.frame(date = format(as.Date(all[nrow(all),1])+1)))
#date[nrow(date) + 1,1] <- as.Date(all[nrow(all),1])+1
pred_all<-NULL
# ncol ritorna il numero di colonne presenti nel dataset, nrow il numero di righe
# Iteriamo per colonna sulle regioni
for (n in 2:ncol(all)-1) {
    # Costruisco la time series
    #La funzione ts () convertirà un vettore numerico in un oggetto della serie temporale R.
    #Il formato è ts (vector, start=, end=, frequency=) dove inizio e fine sono i tempi della prima e dell'ultima osservazione e la frequenza è
    #il numero di osservazioni per unità di tempo (1 = annuale, 4 = trimestrale, 12 = mensile, ecc.).
    Y<-ts(data = all[n+1], start = 1, end =nrow(all)+1)
    # Time Step
    t<-1
    # Costruiamo la matrice A 2x2
    A<-matrix(c(1,0,t,1),2,2)
    #Matrice di osservazione
    H<-matrix(c(1,0),1,2)
    # Condizioni iniziali filtro di Kalman
    # Vettore di stato 2x1
    x0_0<-matrix(c(0,0),2,1)
    # Vettore 2x2
    p0_0<-matrix(c(1,0,0,1),2,2)
    #Rumore di processo
    Q<-matrix(c(0.01,0.01,0.01,0.01),2,2)
    #Rumore di misura
    R<-matrix(c(0.01),1,1)
    X<-NULL
    X2<-NULL
    pred<-NULL

    N<-30
    SMALL<-1e-10
    SMALL_LEL<-1e-9*diag(length(x0_0))
    residual_vec <- vector()
    residual_j_vec <- vector()
    residual_j_log_vec <- vector()
    normHessian_vec <- vector()
    normHRH=norm(t(H)%*%inv(R)%*%H)
    alpha=normHRH
```

```
    # Iteriamo per riga sui dati giornalieri
    for (i in 0:nrow(all)) {
        summ_rj<-0
        summ_log_rj<-0

        namx <- paste("x", i+1,"_",i, sep = "")
        assign(namx,A%*%get(paste("x", i,"_",i, sep = "")))

        namp <-paste("p", i+1,"_",i, sep = "")
        assign(namp, A%*%(get(paste("p", i,"_",i, sep = "")))%*%t(A)+Q)

        ppred = (1/2)*(t(get(namp)) + get(namp))

        r<-drop(Y[i+1]-H%*%get(paste("x", i,"_",i, sep = "")))
        r2<-drop(t(Y[i+1]-H%*%get(paste("x", i,"_",i, sep = ""))) %*% (Y[i+1]-H%*%get(paste("x", i,"_",i, sep = ""))))

        if(r2<SMALL){
            r2=SMALL
        }
        if(i>1){
            if(residual_vec[i-1]<SMALL) {
                residual_vec[i-1]=SMALL
            }
        }

        residual_vec <- c(residual_vec, r2)
```

```
if(i>255) {

    residual_j_vec <- residual_vec[(i-N+1):i]
    j<-which(residual_j_vec==max(residual_j_vec))
    residual_j_vec[j] = residual_j_vec[j]*1000
    #REGULARIZATION
    sc<-100
    r2max<-residual_j_vec[j]
    if(r2>r2max){
        r2Reg<-r2*sc
        r2<-r2Reg
    }else{
        r2Reg<-r2max*sc
    }

    residual_j_log_vec<-residual_j_vec
    for (q in 0:length(residual_j_log_vec)) {
        residual_j_log_vec[q]<-residual_j_log_vec[q]*log(residual_j_log_vec[q])
    }

    Dk<-sum(residual_j_vec) + (sc-1)*r2Reg
    Sk<-sum(residual_j_log_vec) + sc*r2max*log(sc)+(sc-1)*r2Reg*log(r2Reg)
    if(is.nan(Dk)) {
        print('Dk NAN')
    }
    if(is.nan(Sk)) {
        print('Sk NAN')
    }

    #Calculate functional cost C
    C = (-1/log(i))*(Sk/Dk - log(Dk))
    #Calculate gradient
    if(1/Dk**2<SMALL){
        invDkReg2 <- 1/(Dk**2 + SMALL)
    } else {
        invDkReg2 <- 1/Dk**2
    }
    gradH<-((invDkReg2*(2/log(i))) * (Dk*log(r2)-Sk)) %*% (r*H)

    #Calculate Hessian
    hessianH<-(invDkReg2*(2*log(i))) * ((2*t(r*H)%*%(r*H) * (2*log(r2)-2*Sk/Dk -Dk/r2 +1)) - t(H)%*%H*(Dk*log(r2)-Sk))

    #Compute alpha
    if(length(normHessian_vec)<10){
        normHessian_vec <- c(normHessian_vec, norm(hessianH))
    }
    lambda_val<-eigen(hessianH)
    alpha<-normHRH/median(normHessian_vec)

    if(is.infinite(alpha))
    {
        alpha=normHRH
    }
    alpha = alpha/2
    if(min(lambda_val$values)<0) {
        namx2 <- paste("x", i+1,"_",i+1, sep = "")
        assign(namx2,get(namx))

        namp2 <- paste("p", i+1,"_",i+1, sep = "")
        assign(namp2,get(namp))
    } else {

        namk <- paste("k", i+1, sep = "")
        assign(namk,inv(inv(ppred) + alpha*(hessianH+SMALL_LEL)) * alpha*(hessianH))

        namp2 <- paste("p", i+1,"_",i+1, sep = "")
        assign(namp2,inv(inv(ppred) + alpha*hessianH+SMALL_LEL))

        namx2 <- paste("x", i+1,"_",i+1, sep = "")
        assign(namx2,get(paste("x", i+1,"_",i, sep = "")) + get(namk)%*%(get(paste("x", i,"_",i, sep = ""))-get(paste("x", i+1,"_",i, sep = ""))) - alpha*get(namp2)%*%t(gradH))
    }
```

```
        } else {
            #UPDATE
            namR <- paste("R", i+1, sep = "")
            assign(namR,Y[i+1]-H%*%get(paste("x", i+1,"_",i, sep = "")))

            namk <- paste("k", i+1, sep = "")
            assign(namk,get(paste("p", i+1,"_",i, sep = ""))%*%t(H)%*%(inv(H%*%get(paste("p", i+1,"_",i, sep = ""))%*%t(H) + R)))

            namx2 <- paste("x", i+1,"_",i+1, sep = "")
            assign(namx2,get(paste("x", i+1,"_",i, sep = ""))+get(paste("k", i+1, sep = ""))%*%get(paste("R", i+1, sep = "")))

            namp2 <- paste("p", i+1,"_",i+1, sep = "")
            assign(namp2,(p0_0-get(paste("k", i+1, sep = ""))%*%H)%*%get(paste("p", i+1,"_",i, sep = "")))
        }

        #Creo il vettore X appendando i valori predetti 1° riga di x
        X<-rbind(X,get(paste("x", i+1,"_",i,sep = ""))[1])
        #Creo il vettore X2 appendando i valori predetti 2° riga di x
        X2<-rbind(X2,get(paste("x", i+1,"_",i,sep = ""))[2])
        # rimuovo le variabili create 2 step prima
        if(i>2){
            # remove è usata per rimuovere oggetti creati
            remove(list=(paste("p", i-1,"_",i-2, sep = "")))
            remove(list=(paste("k", i-1, sep = "")))
            remove(list=(paste("R", i-1, sep = "")))
            remove(list=(paste("p", i-2,"_",i-2, sep = "")))
            remove(list=(paste("x", i-1,"_",i-2, sep = "")))
            remove(list=(paste("x", i-2,"_",i-2, sep = "")))
        }
    }
    pred<-NULL
    # Combino i vettori Y, X ed X2 in una matrice
    pred<-cbind(Y,X,round(X2,4))
    pred<-as.data.frame(pred)
    # appendo il nome della regione
    pred$region<-colnames(all[,n+1, drop=FALSE])
    # appendo la data
    pred$date<-date$date
    #Definisco i tassi di crescita o decrescita
    pred$actual<-rbind(0,(cbind(pred[2:nrow(pred),1]-pred[1:nrow(pred)-1,1])/pred[1:nrow(pred)-1,1]))*100
    pred$predict<-rbind(0,(cbind(pred[2:nrow(pred),2]-pred[1:nrow(pred)-1,2])/pred[1:nrow(pred)-1,2]))*100
    pred$pred_rate<-(pred$X/pred$Y-1)*100
    pred$X2_change<-rbind(0,(cbind(pred[2:nrow(pred),3]-pred[1:nrow(pred)-1,3])))
    pred_all<-rbind(pred_all,pred)
}
pred_all<-cbind(pred_all[,4:5],pred_all[,1:3])
names(pred_all)[5]<-"X2"
# ordino i valori secondo alla regione e la data
pred_all=pred_all[with( pred_all, order(region, date)), ]
pred_all<-pred_all[,3:5]

write.csv(pred_all,"/Users/danilogiovannico/Desktop/PROGETTO-Estimation\ and\ Data\ Analysis\ with\ Applications/COVID\ PREDICTIONS/ts_r_KF.csv", row.names = FALSE)
```

The post processing and data plot phases are faithful to those of the one-day forecast case for the Spreads seen previously. The result obtained with the robust estimate is the following:




It is visible how the estimate has a regular trend compared to the simple case, and this makes sense if you had the correct data. It is unthinkable that the variation of positives could have a decrease of over 1000 units in a single day and then gradually rise again, it goes against the theory of compartmental models that applies to the study of pandemics;

on the contrary, the prediction with the robust version has a gradual climb and then stabilizes at around 10,000 units. What will happen next is not known correctly given the presence of the last outlier close to the final values, so we do not know if the actual subsequent values will actually decrease having exceeded the peak or will continue to rise to reach it; in any case we have obtained a forecast value for the following day very close to what we would actually have. Another observation can be made about the estimated curve: it is evident that it slightly overestimates the observed values but we cannot know who is right between the measurements provided and the estimate obtained given the presence of outliers (which will be verified with the new measurements of the days subsequent, in the absence of outliers of course); if there are underestimation or overestimation behaviors there is a need to re-tune with new values by varying the noise matrices and playing with the parameter $\alpha$, which must be kept constant once the optimal one is found.

## CHAPTER 3 – INTEGRATION OF UKF (UNSCENTED KALMAN FILTER) AND EKF (EXTENDED KALMAN FILTER) IN THE SEIR AND SIRD EPIDEMIOLOGICAL MODEL TO PREVENT THE TREND OF EPIDEMIC CURVES

For more information on compartmental models see Bibliography 12.

***Epidemiological models***

In epidemiology, a mathematical model is a symbolic model consisting of one or more equations that take into account the various parameters that are involved in the genesis and evolution of the phenomenon of health interest (generally: a disease) studied. The formulation of the mathematical models is the subject of the study of biomathematics, where the models originate from the deterministic description of the temporal evolution of the epidemic event studied, that is, the kinetics of the transformations that can make it up. The mathematical models used in epidemiology are built for different purposes, for example: predicting the course of a disease under certain conditions or predicting the effect on prevalence or incidence when certain control measures are adopted or calculating the risk of death or life expectancy during an epidemic or in specific environmental conditions. A good model allows you to simulate what will happen in nature and therefore can represent a very useful tool in the study of diseases. In order to allow the explication of these models, it is necessary to explain that those presented and analyzed in this work are based on a compartment organization. We mean that the total population or community taken into consideration as the subject of the spread of the epidemic is divided into a (usually small) number of discrete categories. The S-Susceptible class consists of those individuals who may have the disease but have not yet been infected by it. The class of the Infectious I, consists of those who are sick and who in turn are transmitting the disease to others, and that of the Repressed R, happy those who have been eliminated from the circle of the susceptible as or totally healed (and therefore resisi immune), or because they have been isolated or died (both from natural causes and as a result of the infection itself). These listed are only the main and most common categories that recur across all models, but many other subdivisions are possible, depending on the type of contagion considered.

***Initial assumptions and hypotheses***

In addition to this first characterization, we therefore see the other peculiar aspects of the models for epidemics in Biomathematics.

- In classical models, the assumption is made that the total N of the population is constant and also large enough to consider the amplitude of each class as a continuous variable rather than a discrete one.
- Models can be distinguished in those with vital dynamics or without vital dynamics. In the first case it is assumed that the births and deaths of individuals follow each other with equal weight within the vital process, and that all the new

born are totally among the susceptible. Individuals are then removed in the event of death from each class with the same constant of proportionality, called the mortality coefficient ($\delta$). The life span of each individual will therefore be the inverse of this coefficient. We can simplify the concept by talking about models with vital dynamics when the duration of the disease is such as to significantly influence the life trend of the population, unlike what happens in the cases of models without vital dynamics.

- The population is uniformly and homogeneously mixed within each class. A daily contact factor $\lambda$ indicates the average value of contacts per infectious per day. The contact of an infectious is an interaction that leads to the contagion of another individual, belonging to the category of susceptible. $\lambda$ is a fixed value and does not vary seasonally. From this definition it follows that the average number of susceptible infected by an infectious daily is given by $\lambda S$, and the average number of infected by the entire class of infectious, of size (NI), is $\lambda SIN$.
- Individuals recover and are removed from class I at a rate proportional to the infectious (we call the proportionality constant $\gamma$). The latency period (i.e. the time between the moment of exposure to the infection and the moment when the disease begins to manifest itself) is zero. Therefore the function that indicates the portion of individuals exposed (and immediately capable of transmitting the infection) at the initial instant $t_0$, and who are still infectious at the next instant of time $t_0 + t$, is $e^{-\gamma t}$. The average period in which the individual is contagious turns out to be the inverse of the constant of proportionality $\frac{1}{\gamma}$.

From the considerations made, it is clear that the rate of removal from the category of infectious both for death and for healing is given by $\gamma + \delta$, from which it immediately follows that the period of infection is $\frac{1}{\gamma + \delta}$. Similarly we can calculate the average number of contacts with susceptibles, during the period of infectivity of an individual, as $\rho = \frac{\lambda}{\gamma + \delta}$; while the average number of susceptible infected by an infectious during its period of illness is given by $\rho S$.

***Classic SIR model***

Let us consider the evolution of an epidemic within a host population, of which the total of individuals is taken constant, of value N. One of the most elementary models, but still extremely relevant and particularly appropriate for describing those infectious diseases that confer a lasting immunity, is the so-called SIR model, whose first formulation, around 1927, is thanks to the two scientists Kermack and McKendrick. This is a model that is based on the three main categories S, I, R and takes into consideration the case in which a small group of infectives is introduced into a large population: a classic problem could be that of wanting to describe the spread of the disease all over the world. population, as a

function of time. Clearly this phenomenon depends on a great variety of circumstances, including the very type of disease considered, but as a first attempt to formulate a model, we will make the following general assumptions.

We consider a disease that confers total immunity to the individual, once healing is achieved, or leads directly to death if lethal. In both cases these individuals will return to be part of the repressed, since in any case, they can no longer be infected. The process that regulates the spread of this disease can be schematized in this way S -> I -> R. In order to write a mathematical formulation of this phenomenon, we must introduce a series of differential equations, in order to indicate the transfer rates of individuals, from one compartment to another:

$$\frac{dS}{\mathrm{dt}} = f_1(I,S,R)$$
$$\frac{dI}{\mathrm{dt}} = f_2(I,S,R)$$
$$\frac{dR}{\mathrm{dt}} = f_3(I,S,R)$$

For this type of infectious process, in the passage from class S to class I the law of mass-action is typically assumed to be valid (rate at which susceptible become infectious), while to define the transfer from class I to R, a function is considered of exponential decrease. The simplest way to define the three functions $f_i$, i = 1; 2; 3; is the following:

$$f_1 = -\lambda IS$$
$$f_2 = \lambda IS - \gamma I$$
$$f_3 = \gamma I$$

Where $\gamma$ and $\lambda$ are positive constants representing the removal rate from the infectious category and the infection rate of the disease, respectively.

The SIR model thus described is valid if it is assumed that, when a susceptible is infected, it immediately becomes infectious, or no latency period of the disease is considered. If, on the other hand, there is an incubation time, an additional class containing those individuals subject to latency must be included in the model.

The system of differential equations thus obtained is the following:

$$\begin{cases} \frac{dS}{\mathrm{dt}} = -\lambda IS \\ \frac{dI}{\mathrm{dt}} = \lambda IS - \gamma I \\ \frac{dR}{\mathrm{dt}} = \gamma I \end{cases}$$

with initial conditions $S(0) = S_0 > 0$, $I(0) = I_0 > 0$, $R(0) = 0$. We are interested in solving this complete system, looking only for solutions that are positive for S, I, R, in how much other

solutions would not make sense from the perspective of biological mechanisms. From the sum of the equations:

$$\frac{dS}{\mathrm{dt}} + \frac{dI}{\mathrm{dt}} + \frac{dR}{\mathrm{dt}} = 0$$

it follows that:

$$S(t) + I(t) + R(t) = N,$$

where N is the total of the population.
A crucial problem in any epidemic situation is to try to determine if, given $\lambda$, $\gamma$ and $S_0$ and the initial value of infectious $I_0$, the infection is in the conditions to spread or if it will remain a restricted phenomenon of small scale, and at what value will begin its decline. From the report:

$$\frac{dI}{\mathrm{dt}} = (\lambda S - \gamma) I$$

it follows that:

$$\frac{dI}{\mathrm{dt}} > 0 \leftrightarrow (\lambda S - \gamma) > 0$$

in particular:

$$\frac{dI}{\mathrm{dt}} > 0 \leftrightarrow (\lambda S_0 - \gamma) > 0$$

This condition is met if $S_0 > \frac{\gamma}{\lambda}$, which means that the epidemic will spread. For a value of $S_0 < \frac{\gamma}{\lambda}$, the epidemic is extinguished. The ratio $\rho = \frac{\gamma}{\lambda}$ therefore assumes a threshold value with respect to which a bifurcation phenomenon occurs. We write

$$R_0 = \frac{\gamma}{\lambda} S_0$$

Where $R_0$ is the disease baseline reproductivity rate which represents the number of individuals who contracted the disease from those who were already infectious, within a totally susceptible population, (the number of susceptible equals the total of the population itself). The value $\frac{1}{\lambda}$, on the other hand, represents the average duration of the disease. This value of $R_0$ is extremely useful for characterizing the trend of the epidemic. With respect to this rate we can reformulate the threshold theorem in these terms:

- if $R_0 > 1$, the epidemic spreads;
- if $R_0 < 1$, the epidemic ends.

One way to reduce the value of $R_0$, in order to prevent the disease from spreading further, if its value tends to be too high, would be to reduce the number of susceptible $S_0$.

Vaccination, for example, is one of the most effective methods of enabling this, but it can be successful if applied to stable and not overly large communities.
The important thing to take into consideration is that an epidemic can still begin and spread very quickly despite the reproductivity value growing below the criticality threshold.
Starting from this classic model it is possible to derive alternative models, adding new compartments and modeling their differential equations: alternative models are the SEIR, SIRD etc.

## 3.1 IDEA OF INTEGRATION OF EKF (EXTENDED KALMAN FILTER) INTO COMPARTMENTAL MODELS

In this section, we propose the integration of the EKF with a slightly more complex compartmental model than the simple SIR, the SIRD; this model also includes the compartment for the deceased. The analysis as well as in other cases is limited to the individual regions of Italy.

*Differential equations SIRD model*

The compartmental model chosen used as a non-linear function to be passed to the UKF is the SIRD, this model takes into account individuals Susceptible to the disease, the Infected, the Healed and the Dead.


The dynamics of this model are characterized by a set of four ordinary differential equations that correspond to the stages of disease progression:

$$\frac{dS}{dt} = -\frac{\alpha}{N} I(t)S(t)$$

$$\frac{dI}{dt} = \frac{\alpha}{N} I(t)S(t) - (\gamma + \beta)I(t)$$

$$\frac{dR}{dt} = \beta I(t)$$

$$\frac{dD}{dt} = \gamma I(t)$$

The constants $\alpha$, $\beta$ and $\gamma$ are respectively the rate of infection, recovery and mortality. They are of vital importance to define the variation of the compartmental model curves, an incorrect setting of them can lead to incorrect predictions and consequently to a simulation of the wrong pandemic course. The main problem in their definition is due to the fact that they vary stochastically according to the restrictive measures adopted and according to the mobility and contact that people undergo; it goes without saying that if there was no contact then the disease would not progress.

***Enxtended Kalman Filter (EKF)***

We now describe the operation of the Extended Kalman Filter and then apply it to the SIRD-EKF practical case for the prediction of the course of the pandemic.
Consider the following discrete-time dynamic system with state evolution and (possibly) non-linear state-output link:

$$\begin{cases} x_{k+1} = f(x_k,\, u_k,\, w_k) \\ y_k = h(x_k,\, v_k) \end{cases}$$

where $w_k$ and $v_k$ are process and measurement noises with distributions $w_k \sim p_w(\cdot)$ and $v_k \sim p_v(\cdot)$ respectively, not necessarily Gaussian. It is assumed that these noises are white and unrelated to each other, $v_k \perp w_k$. The initial condition is $x(0) = x_0$, with $x_0$ random vector distributed according to $x_0 \sim p_0(\cdot)$. It is also assumed that, at any instant k, the controls $u_k$, $u_{k-1}$ ,. . . are known. The maps f and h can possibly be time variants, i.e. $f(\cdot) = f_k(\cdot)$ and $h(\cdot) = h_k(\cdot)$. The assumptions about the system do not allow to apply the classical Kalman filter. The calculation, moment by moment, of the distribution

$$p(x_k |\, y_0,\, \ldots,\, y_k,\, u_0,\, \ldots,\, u_{k-1})$$

it is, from a practical point of view, too difficult to perform. Also the optimal estimator:

$$\hat{x}_{k|k} = \mathbb{E}[x_k |\, y_0,\, \ldots,\, y_k,\, u_0,\, \ldots,\, u_{k-1}].$$

The algorithm is called Extended Kalman Filter (EKF), and is based on the idea of applying the classic Kalman filter to the linearized system moment by moment. It is assumed that $u_k$ is known and that the noises $v_k$ and $w_k$ have Gaussian distributions. The state $x_k$ will not have, in general, a Gaussian distribution, since the dynamics are non-linear. Consider the map:

$$z = g(\xi)$$

where ξ is a random variable with a known $p_\xi(\cdot)$ distribution. Let $\xi \in R^m$, $z \in R^n$ e $g : R^m \rightarrow R^n$ be a known nonlinear map. The procedure consists of two approximations:

- approximate $p_\xi$ with a Gaussian $\tilde{p}_\xi \sim \mathcal{N}(\bar{\xi},\ P_\xi)$
- approximate the map g with its linearization $\frac{\partial g}{\partial \xi}$

so that the variable z can in turn be approximated by a Gaussian variable with distribution $\tilde{p}_z \sim \mathcal{N}(\bar{z},\ P_z)$.

The classical Kalman filter equations are:

$$\begin{cases} \hat{x}_{k+1|k} = A\hat{x}_{k|k} + Bu_k \\ P_{k+1|k} = AP_{k|k}A^T + Q \end{cases} \qquad \text{Predizione}$$

e

$$\begin{cases} \hat{x}_{k+1|k+1} = \hat{x}_{k+1|k} - P_{k+1|k}C^T(CP_{k+1|k}C^T + R)^{-1}(y_{k+1} - \hat{y}_{k+1|k}) \\ P_{k+1|k+1} = P_{k+1|k} - P_{k+1|k}C^T(CP_{k+1|k}C^T + R)^{-1}CP_{k+1|k} \end{cases} \qquad \text{Aggiornamento}$$

based on assumptions: $x_k \sim \mathcal{N}(\hat{x}_{k|k}, P_{k|k}), w_k \sim \mathcal{N}(0, Q)$ *e* $v_k \sim \mathcal{N}(0, R)$.

The linearized dynamics thanks to the Taylor series development of the considered system is:

$$x_{k+1} \simeq \underbrace{f(\hat{x}_{k|k},\, u_k,\, 0)}_{\text{punto di media}} + \underbrace{\frac{\partial f}{\partial x_k}|_{(\hat{x}_{k|k},\, u_k,\, 0)}}_{A_k}(x_k - \hat{x}_{k|k}) + \underbrace{\frac{\partial f}{\partial w_k}|_{(\hat{x}_{k|k},\, u_k,\, 0)}}_{F_k} w_k,$$

from which it arises as an optimal estimator:

$$\begin{cases} \hat{x}_{k+1|k} = \mathbb{E}[x_{k+1}|\, y_0,\, \ldots,\, y_k,\, u_0,\, \ldots,\, u_k] = f(\hat{x}_{k|k},\, u_k,\, 0) \\ P_{k+1|k} = A_k P_{k|k} A_k^T + F_k Q F_k^T \end{cases}$$

Note that there is no theoretical guarantee that this estimator is the optimal one, and in fact not even that it is correct, since it is only an approximation. In particular, the matrices $P_{k+1|k}$ do not correspond to the true error variances of this estimator. Similarly, the linearized output is given by:

$$y_k \simeq h(\hat{x}_{k|k},\, 0) + \underbrace{\frac{\partial h}{\partial x_k}|_{(\hat{x}_{k|k},\, 0)}}_{C_k}(x_k - \hat{x}_{k|k}) + \underbrace{\frac{\partial h}{\partial v_k}|_{(\hat{x}_{k|k},\, 0)}}_{G_k} v_k$$

from which the update step results:

$$\begin{cases} \hat{x}_{k+1|k+1} = \hat{x}_{k+1|k} - P_{k+1|k}C_k^T(C_k P_{k+1|k} C_k^T + G_k R G_k^T)^{-1}(y_{k+1} - h(\hat{x}_{k+1|k},\, 0)) \\ P_{k+1|k+1} = P_{k+1|k} - P_{k+1|k}C_k^T(C_k P_{k+1|k} C_k^T + G_k R G_k^T)^{-1} C_k P_{k+1|k} \end{cases}$$

Obviously, in order to obtain the matrices $A_k$, $F_k$, $C_k$, $G_k$, the calculation of Jacobians, which is often difficult to calculate, is required. Specifically, the matrices in question are:

$$A_k = \left.\frac{\partial f(x, u_k, \overline{w})}{\partial x}\right|_{\hat{x}_k^-} \qquad F_k = \left.\frac{\partial f(\hat{x}_k^-, u_k, w)}{\partial w}\right|_{\overline{w}}$$
$$C_k = \left.\frac{\partial h(x, \bar{v})}{\partial x}\right|_{\hat{x}_k^-} \qquad G_k = \left.\frac{\partial h(\hat{x}_k^-, v)}{\partial v}\right|_{\bar{v}}$$

Compared to linear Kalman, the EKF version is a sub-optimal choice as an estimator but still widely accepted and used in practical applications. The extended Kalman filter, by its construction, achieves only a first order precision but still allows results close to the optimum in the case of operation of the filter in points where the second derivatives are zero.

Note that:

- the Extended Kalman Filter is a technique based on the hypothesis of Gaussianity and small variance of the noises and generally does not allow for guarantees of stability.
- the analysis in terms of performance and error variances is generally not possible for the EKF;
- it may happen that the EKF works well for long stretches, only to then diverge irremediably.

For more information see Bibliography 3.

**Implementation of the EKF-SIRD with Python**

Now we will see how the EKF-SIRD in Python language has been implemented.
Starting from the previous theory, we represent the equations of the SIRD model for discrete time (using Euler's method):

$$S_{t+1} = S_t - \frac{\alpha_t S_t I_t}{N}\Delta t$$
$$I_{t+1} = I_t + \left(\frac{\alpha_t S_t}{N} - \beta_t - \gamma_t\right) I_t \Delta t$$
$$R_{t+1} = R_t + \beta_t I_t \Delta t$$
$$D_{t+1} = D_t + \gamma_t I_t \Delta t$$

Where the parameters $\alpha$, $\beta$ and $\gamma$ are respectively the rate of infection, recovery and mortality.

We define the model as:

$$x_t \triangleq \begin{bmatrix} S_t \\ I_t \\ R_t \\ D_t \\ \alpha_t \\ \beta_t \\ \gamma_t \end{bmatrix} \qquad f_t(x_t) \triangleq \begin{bmatrix} S_t - \frac{\alpha_t S_t I_t}{N}\Delta t \\ I_t + \left(\frac{\alpha_t S_t}{N} - \beta_t - \gamma_t\right) I_t \Delta t \\ R_t + \beta_t I_t \Delta t \\ D_t + \gamma_t I_t \Delta t \\ \alpha_t \\ \beta_t \\ \gamma_t \end{bmatrix} \qquad h_t(x_t) \triangleq \begin{bmatrix} S_t \\ I_t \\ R_t \\ D_t \end{bmatrix}$$

Jacobians are:

$$\frac{\partial f_t}{\partial x_t}(x_t) = \begin{bmatrix} 1-(\alpha_t I_t/N)\Delta t & -(\alpha_t S_t/N)\Delta t & 0 & 0 & -(S_t I_t/N)\Delta t & 0 & 0 \\ (\alpha_t I_t/N)\Delta t & 1+(\alpha_t S_t/N-\beta_t-\gamma_t)\Delta t & 0 & 0 & (S_t I_t/N)\Delta t & -I_t\Delta t & -I_t\Delta t \\ 0 & \beta_t \Delta t & 1 & 0 & 0 & I_t \Delta t & 0 \\ 0 & \gamma_t \Delta t & 0 & 1 & 0 & 0 & I_t \Delta t \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$C \triangleq \frac{\partial h_t}{\partial x_t}(x_t) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

where $x_t$ is the augmented state vector (SIRD) and the total population is given by N = $S_t$ + $I_t$ + $R_t$ + $D_t$.

The dynamics are given by:

$$x_{t+1} \triangleq \begin{bmatrix} S_t - \frac{\alpha_t S_t I_t}{N}\Delta t \\ I_t + \left(\frac{\alpha_t S_t}{N} - \beta_t - \gamma_t\right) I_t \Delta t \\ R_t + \beta_t I_t \Delta t \\ D_t + \gamma_t I_t \Delta t \\ \alpha_t \\ \beta_t \\ \gamma_t \end{bmatrix} + w_t$$

$$y_t = \begin{bmatrix} S_t \\ I_t \\ R_t \\ D_t \end{bmatrix} + v_t$$

In out case $\Delta t = 1$ as the measurements are acquired daily.

The EKF is used to regularize the observed values of the state variables for the SIRD model and to obtain a dynamic estimate of the parameters until we have measurements available. The simple idea to predict the evolution of the epidemic is to run the SIRD model in an open loop (i.e. without the measurement correction), considering:

- the final estimates for the variables $S$, $I$, $R$, $D$ given by the EKF as initial condition;
- the final estimates for the parameters $\alpha$, $\beta$, $\gamma$ given by the EKF as invariant parameters over time.

Below we represent the EKF prediction and update steps that will be implemented in the code:

**Predict**

$$\hat{X}(t+1|t) = F(\hat{X}(t|t))$$
$$P(t+1|t) = J_F(\hat{X}(t|t))P(t|t)J_F(\hat{X}(t|t))^T + Q_F(t)$$

**Update**

$$\hat{Y}(t+1) = Y(t+1) - C\hat{X}(t+1|t)$$
$$K(t+1) = P(t+1|t)C^T(CP(t+1|t)C^T + R_F(t))^{-1}$$
$$\hat{X}(t+1|t+1) = \hat{X}(t+1|t) + K(t+1)\hat{Y}(t+1)$$
$$P(t+1|t+1) = (I - K(t+1)C)P(t+1|t)$$

Where:

$$F(X_t) = F(\hat{X}_t) + J_F(\hat{X}_t)(X_t - \hat{X}_t)$$

where $X_t$ (augmented state vector) and $J_F$ (Jacobian relative to the nonlinear function of the SEIRD model f) have been written above.

PLEASE NOTE: the Jacobian output $\partial h_t$ / $\partial x_t$ is independent of the state $x_t$. In fact in this formulation the measurement model $h_t$ is a linear function of the state $x_t$. Consequently, in the EKF estimation we have a simplification consisting of a single invariant time matrix $C_t$= $C$ that can be calculated once and for all outside the FOR loop.

***Constants setting***

After reading the data from the civil protection dataset and after placing them in a dataframe, as we have already seen above, we proceed to set the parameters and the initialization matrices of the EKF.

```
region = "Puglia"
data_filtered = data_SIR[data_SIR['denominazione_regione']==region]
# DATASET
POPULATIONS = {
    #"Abruzzo":   1305770,
    #"Basilicata": 556934,
    #"Calabria": 1924701,
    #"Campania": 5785861,
    #"Emilia-Romagna": 4467118,
    #"Friuli Venezia Giulia": 1211357,
    #"Lazio": 5865544,
    #"Liguria": 1543127,
    #"Lombardia": 10103969,
    #"Marche": 1518400,
    #"Molise": 302265,
    #"P.A. Bolzano": 106951,
    #"P.A. Trento": 117417,
    #"Piemonte": 4341375,
    "Puglia": 4008296,
    #"Sardegna": 1630474,
    #"Sicilia": 4968410,
    #"Toscana": 3722729,
    #"Umbria": 880285,
    #"Valle d'Aosta": 125501,
    #"Veneto": 4907704
}

# observed infected
oI = data_filtered['totale_casi'].values

# observed recovered
oR = data_filtered['dimessi_guariti'].values

# observed dead
oD = data_filtered['deceduti'].values

# observed susceptibles
N = POPULATIONS[region]  # population size
T = oI.size       # observation horizon
DAYS_PREDICTION = 5

oS = np.zeros((T,))
for t in range(0, T):
    oS[t] = N - (oI[t] + oR[t] + oD[t])

y  = np.transpose(np.array([     oS,      oI,      oR,      oD]))

###############################################################################


# initializazions
thetaKF = np.zeros((T, 3))
x0      = np.concatenate([y[0], thetaKF[0]])
P0      = np.identity(x0.size)

mu, sigma, sigma2 = 0, 0.1, 0.001 # mean and standard deviation

Q       = np.identity(x0.size) * np.random.normal(mu, sigma, x0.size)
R       = np.identity(y[0, :].size) * np.random.normal(mu, sigma2, y[0, :].size)

# state estimation
x       = SIRDEKF(Q, R, x0, P0, y, DAYS_PREDICTION)

# parameter estimate
thetaKF = x[:, 4:]
print("MSE = ", quadcost(y, x[0:T:, 0:4]))   # mean squared error between the observed
###############################################################################
```

Specifically, the settings are as follows:

- Choice of the region and the number of the corresponding population (in our case Puglia);
- Reading and setting of the number of total positive cases, dead and healed per day;
- DAYS_PREDICTION: number of days to perform the prediction;

- y: vector containing the measures of Infected, Deceased, Healed and Susceptible (calculated as S = N - I + R + D);
- x0: initial state vector containing both the transposed vector y and the variability rates $\alpha$, $\beta$ and $\gamma$ initially set to 0;
- Q: 7x7 process noise matrix, initialized as standard Gaussian noise with zero mean and variance 0.1, thanks to the normal method of the numpy library in a random way;
- R: 4x4 measurement noise matrix, initialized as standard Gaussian noise with zero mean and variance 0.01, thanks to the normal method of the numpy library in a random way.

***EKF-SIRD algorithm***

After making these settings, the SIRDEKF method is launched (which receives the error matrices, the initial state, the measurements and the number of days to predict) and launches the algorithm that faithfully reflects that described theoretically for the EKF.

```
def SIRDEKF(Q, R, x0, P0, y, DAYS_PREDICTION):
    T   = y[:, 0].size       # observation horizon
    x_post = np.zeros((T+DAYS_PREDICTION, 7))   # corrected estimations time series

    # initializations
    x_prior = x0               # initial estimate
    P_prior  = P0                # initial estimate covariance

    # output Jacobian C = dh / dx
    C   = np.array([[1, 0, 0, 0, 0, 0, 0],
                    [0, 1, 0, 0, 0, 0, 0],
                    [0, 0, 1, 0, 0, 0, 0],
                    [0, 0, 0, 1, 0, 0, 0]])

    for t in range(1, T+DAYS_PREDICTION):
        if t < T:
            # correction step
            y_est   = y[t, :] - np.dot(C, x_prior)                          # innovation
            S   = np.dot(C, np.dot(P_prior, C.transpose())) + R             # innovation covariance
            K   = np.dot(P_prior, np.dot(C.transpose(), np.linalg.inv(S)))  # correction gain

            x_post[t] = x_prior + np.dot(K, y_est)                          # corrected estimate
            P_post = (np.identity(x0.size) - np.dot(K, C)).dot(P_prior)     # corrected estimate covariance

            # prediction step
            # state Jacobian A = df / dx
            A   = np.array([[1 - x_post[t, 1] / N * x_post[t, 4],                                   - x_post[t, 0] / N * x_post[t, 4], 0, 0, - x_post[t, 0] * x_post[t, 1] / N,          0,          0],
                            [    x_post[t, 1] / N * x_post[t, 4], 1 + x_post[t, 0] / N * x_post[t, 4] - x_post[t, 5] - x_post[t, 6], 0, 0,   x_post[t, 0] * x_post[t, 1] / N, - x_post[t, 1], - x_post[t, 1]],
                            [                                  0,                                                                   x_post[t, 5], 1, 0,                        0,   x_post[t, 1],          0],
                            [                                  0,                                                                   x_post[t, 6], 0, 1,                        0,          0,   x_post[t, 1]],
                            [                                  0,                                                                         0, 0, 0,                        1,          0,          0],
                            [                                  0,                                                                         0, 0, 0,                        0,          1,          0],
                            [                                  0,                                                                         0, 0, 0,                        0,          0,          1]])

            # non linear dynamic of the SIRD model
            hSp =  - x_post[t, 0] * x_post[t, 1] / N * x_post[t, 4]
            hIp =    x_post[t, 0] * x_post[t, 1] / N * x_post[t, 4] - x_post[t, 1] * x_post[t, 5] - x_post[t, 1] * x_post[t, 6]
            hRp =    x_post[t, 1] * x_post[t, 5]
            hDp =    x_post[t, 1] * x_post[t, 6]

            x_prior =  x_post[t] + np.array([hSp, hIp, hRp, hDp, 0, 0, 0])  # predicted estimate x_prior = f(x_post)
            P_prior =  np.dot(A, np.dot(P_post, A.transpose())) + Q         # predicted estimate covariance
        else:
            x_post[t] = x_prior                         # corrected estimate
            P_post = P_prior                            # corrected estimate covariance

            # prediction step
            # state Jacobian A = df / dx
            A   = np.array([[1 - x_post[t, 1] / N * x_post[t, 4],                                   - x_post[t, 0] / N * x_post[t, 4], 0, 0, - x_post[t, 0] * x_post[t, 1] / N,          0,          0],
                            [    x_post[t, 1] / N * x_post[t, 4], 1 + x_post[t, 0] / N * x_post[t, 4] - x_post[t, 5] - x_post[t, 6], 0, 0,   x_post[t, 0] * x_post[t, 1] / N, - x_post[t, 1], - x_post[t, 1]],
                            [                                  0,                                                                   x_post[t, 5], 1, 0,                        0,   x_post[t, 1],          0],
                            [                                  0,                                                                   x_post[t, 6], 0, 1,                        0,          0,   x_post[t, 1]],
                            [                                  0,                                                                         0, 0, 0,                        1,          0,          0],
                            [                                  0,                                                                         0, 0, 0,                        0,          1,          0],
                            [                                  0,                                                                         0, 0, 0,                        0,          0,          1]])

            # non linear dynamic of the SIRD model
            hSp =  - x_post[t, 0] * x_post[t, 1] / N * x_post[t, 4]
            hIp =    x_post[t, 0] * x_post[t, 1] / N * x_post[t, 4] - x_post[t, 1] * x_post[t, 5] - x_post[t, 1] * x_post[t, 6]
            hRp =    x_post[t, 1] * x_post[t, 5]
            hDp =    x_post[t, 1] * x_post[t, 6]

            x_prior =  x_post[t] + np.array([hSp, hIp, hRp, hDp, 0, 0, 0])  # predicted estimate x_prior = f(x_post)
            P_prior =  np.dot(A, np.dot(P_post, A.transpose())) + Q         # predicted estimate covariance

    return x_post
```

This method uses the available measures to perform the Prediction and Update steps, and subsequently, launches the Prediction step on the number of days to predict.

***MSE function and plotting***

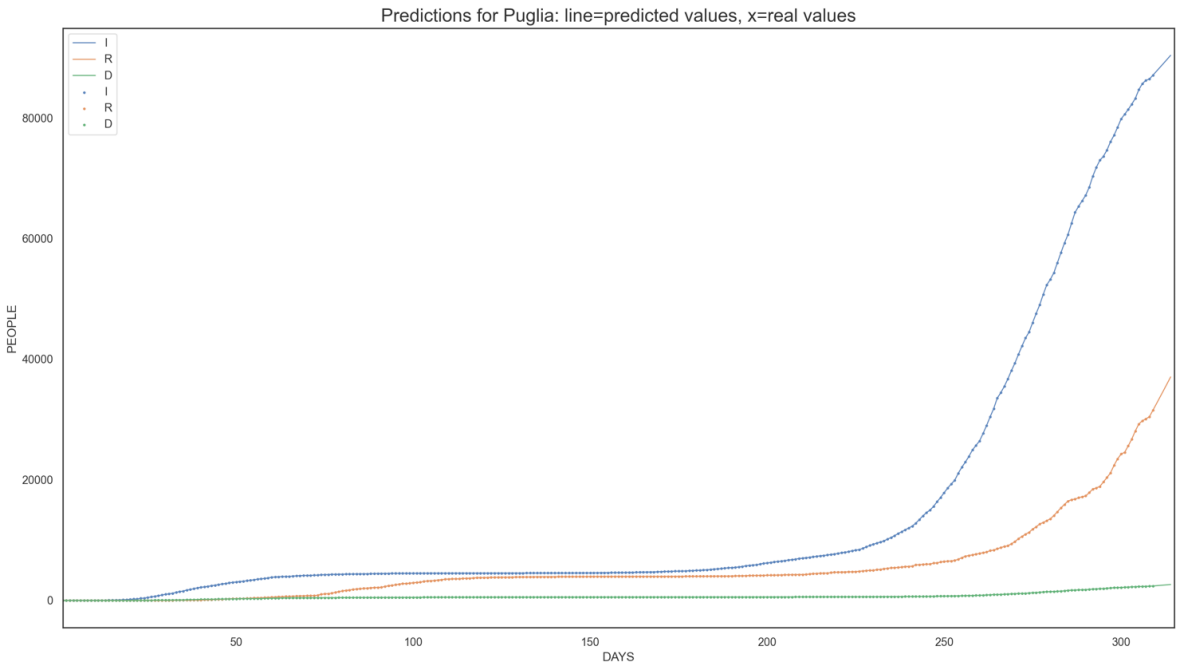
From here we obtain the vector containing the state estimates, which can be used to evaluate their goodness with the real values through MSE, and finally plot the curves. The function for calculating the MSE is:

```
def quadcost(y, hy):
    T = y[:,0].size
    cost = 0

    for t in range(1,T):
        error = y[t] - hy[t]
        cost  += np.power(np.linalg.norm(error), 2)
    return cost
```

While the code for the corresponding plot will be:

```
# PLOTS
fig, axs = plt.subplots(1, 1, constrained_layout=True)

axs.set_title('Predictions for '+region+': line=predicted values, x=real values', fontsize=18)
axs.plot(range(1, T+DAYS_PREDICTION+1), x[:, 1], linewidth=1 )
axs.plot(range(1, T+DAYS_PREDICTION+1), x[:, 2], linewidth=1 )
axs.plot(range(1, T+DAYS_PREDICTION+1), x[:, 3], linewidth=1 )
axs.scatter(range(1, T+1), oI, marker ='x', s=3)
axs.scatter(range(1, T+1), oR, marker ='x', s=3)
axs.scatter(range(1, T+1), oD, marker ='x', s=3)
axs.set_xlim(1, T+DAYS_PREDICTION+1)
plt.xlabel('DAYS', fontsize=12)
plt.ylabel('PEOPLE', fontsize=12)
axs.legend("IRDIRD 1", loc="upper left")
```

Where are we going to plot the number of Infected, Healed and Deceased.




***Conclusions***

From the obtained results we have been able to ascertain that the predicted model works very well for short time ranges 1-7 days max, this because the values $\alpha$, $\beta$ and $\gamma$ will tend to remain constant only in the prediction phase by construction. This is reasonable if we think that in reality the contagion index does not vary for several days, but it is necessary

to wait longer, around 14 days, to re-calculate it; which is why our government tends to change restrictions periodically after n days. Also, the MSE calculation is usually a very low number e.g. 0.0012189730307774792, this is an indication that the EKF provides a good daily estimate of the parameters $\alpha$, $\beta$ and $\gamma$ according to the measurements provided.

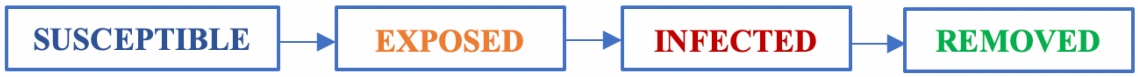
## 3.2 IDEA OF INTEGRATION OF THE UKF (UNSCENTED KALMAN FILTER) INTO COMPARTMENTAL MODELS

In this section, we propose the integration of the UKF with a more complex compartmental model, the SEIR, which also takes into account the incubation period of an individual; this study will be limited to the individual regions of Italy. The UKF-SEIR is shown to translate into a robust forecast of transmission dynamics for 1-4 months to estimate the course of the pandemic.

The SEIR model is associated with a set of differential equations that calculate values for an instantaneous event, while the time series of COVID cases we observe every day is a discrete event. Therefore, all daily events of COVID cases lack the instantaneous effect in the SEIR model based on differential equations. Here, we use the prediction-based UKF (Unscented Kalman Filter) to derive the dynamics. The UKF is a classic non-linear estimation algorithm that accurately and promptly predicts the dynamic state in a non-linear system; it has been used in various areas such as navigation, target tracking, structural dynamics, and vehicle positioning due to its high accuracy and rapid convergence merits.

*Differential equations SEIR model*

The compartmental model chosen used as a non-linear function to pass to the UKF is the SEIR, this model takes into account the individuals Susceptible to the disease, the Exposed in incubation, the Infected, the Removed (i.e. those no longer infected due to isolation or immunity and deceased).




The dynamics of this model are characterized by a set of five ordinary differential equations that correspond to the stages of disease progression:

$$\frac{dS}{dt} = -\frac{R_0}{T_{inf}}\frac{S}{N}I,$$
$$\frac{dE}{dt} = \frac{R_0}{T_{inf}}\frac{S}{N}I - T_{inc}^{-1}E,$$
$$\frac{dI}{dt} = T_{inc}^{-1}E - T_{inf}^{-1}I,$$
$$\frac{dR}{dt} = T_{inf}^{-1}I,$$

where:

$$R_0 = \left(\begin{array}{c}\text{Number of}\\ \text{contacts}\\ \text{per unit time}\end{array}\right)\left(\begin{array}{c}\text{Probability of}\\ \text{transmission}\\ \text{per contact}\end{array}\right)\left(\begin{array}{c}\text{Duration of}\\ \text{infection}\end{array}\right)$$

$$R_0 = r \times \beta \times \frac{1}{\gamma}$$
$$= \frac{r\beta}{\gamma}$$

For these equations, a choice of vital importance is that of constants; in order to have a faithful forecast for a certain time frame, these values must be set to those of the realistic case. Unfortunately, the difficulty in doing so is due to the high variability and evolution of the virus based on exogenous characteristics and various dynamics of the territory. We then describe these scalar values:

- $R_0$: it takes the name of the measure of contagiousness (Reproduction Number) at time t, and implies the number of secondary infections that each infected individual produces (if $R_0$ <1 the epidemic curve of the infected decreases towards 0, otherwise the pandemic grows);
- $T_{inc}$: duration of the incubation period;
- $T_{inf}$: duration of the period in which an individual broadcasts;
- $\mu$: disease-induced mortality rate,
- N: total population number.

Starting from the previous theory, we represent the equations of the SEIR model for discrete time (using Euler's method):

$$S_{t+1} = S_t - \frac{R_0}{T_{inf}}\frac{S_t}{N} I_t \Delta t$$
$$E_{t+1} = E_t + \left(\frac{R_0}{T_{inf}}\frac{S_t}{N} I_t - \frac{E_t}{T_{inc}}\right)\Delta t$$
$$I_{t+1} = I_t + \left(\frac{E_t}{T_{inc}} - \frac{I_t}{T_{inf}}\right)\Delta t$$
$$R_{t+1} = R_t + \frac{I_t}{T_{inf}}\Delta t$$

where we have a $\Delta t = 1$ as the measurements are acquired daily.

This model has been extrapolated and simplified from the paper available in Bibliography 2.

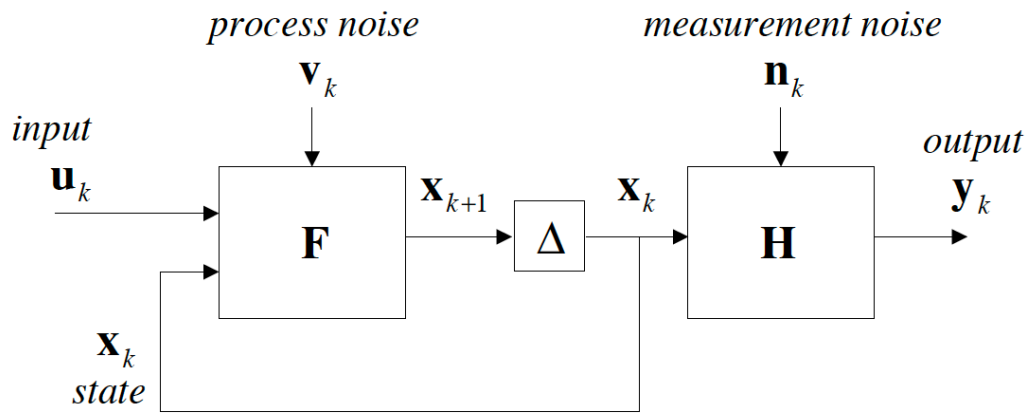
***Unscented Kalman Filter (UKF)***

UKF represents an alternative to EKF, providing superior performance with superior computational complexity.

The Extended Kalman Filter involves the estimation of the state of a linear dynamic discrete time system,

$$\begin{cases} x_{k+1} = F(x_k, u_k, v_k) \\ y_k = H(x_k, n_k) \end{cases},$$

where $x_k$ represents the unobserved state of the system, $u_k$ is known as the exogenous input, and $y_k$ is the observed measurement signal. The process noise $v_k$ drives the dynamics of the system, while the observation noise $n_k$ is relative to the measurements. The matrices of the discretized dynamic model of the system F and H are supposed to be known. We can represent the following block diagram:




In estimating the state, the EKF is the standard choice method to reach a recursive (approximate) maximum-likelihood estimate of the state $x_k$.

Before making any predictions starting from the nonlinear model, this model needs to be linearized using first order Taylor approximations; moreover, in order to be applied, the Jacobian computation on f and h is required, often difficult to solve and implement.

The UKF is able to manage EKF problems much better, using a much more efficient procedure than simple linearization. As for the EKF, the distribution of the state is represented by a GRV (Gaussian Random Variable), but with the difference that it is specified only using a minimal set of sample points. This sample of points completely captures the true mean and covariance of the GRV, and when propagated through the true nonlinear system, they capture the posterior mean and covariance accurately to 2° order (Taylor series expansion) for any nonlinearity. To deepen this concept, we must explain the unscented transformation.

*Unscented Trasformation*

UT is a method for calculating the statistics of a random variable that undergoes a non-linear transformation. Let us consider propagating an RV x (of dimension L) through a nonlinear function, y = f (x). Suppose x has mean $\bar{x}$ and covariance $P_x$. To compute the

statistics of y, we construct an *X* matrix of 2L + 1 sigma vectors $X_i$ like so:

$$\begin{aligned}
\mathcal{X}_0 &= \bar{\mathbf{x}} \\
\mathcal{X}_i &= \bar{\mathbf{x}} + \left(\sqrt{(L+\lambda)\mathbf{P}_{\mathbf{x}}}\right)_i \quad i = 1, \ldots, L \\
\mathcal{X}_i &= \bar{\mathbf{x}} - \left(\sqrt{(L+\lambda)\mathbf{P}_{\mathbf{x}}}\right)_{i-L} i = L+1, \ldots, 2L
\end{aligned}$$

where $\lambda = \alpha^2(L + \kappa) - L$ is a scalar parameter. The constant $\alpha$ determines the spread of sigma points around $\bar{x}$ and is usually a small number. The constant $\kappa$ is a secondary scalar parameter usually set to 0 or 3-L, and $\beta$ is used to incorporate a priori knowledge of the x distribution (for Gaussian distribution $\beta$ = 2 is optimal). $(\sqrt{(L + \lambda) + P_x})_i$ is the i-th Column of the square root matrix (for stability it is advisable to use the lower triangular Cholesky factorization). These sigma vectors are propagated through nonlinear functions,

$$\mathcal{Y}_i = f(\mathcal{X}_i) \quad i = 0, \ldots, 2L \quad ,$$

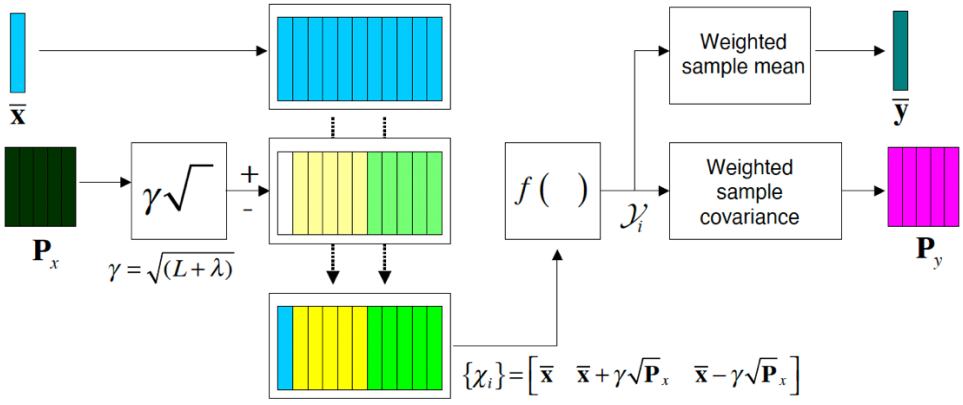
and the mean and covariance for y are approximated using a weighted mean and covariance of the posterior sigma points,

$$\begin{aligned}
\bar{\mathbf{y}} &\approx \sum_{i=0}^{2L} W_i^{(m)} \boldsymbol{\mathcal{Y}}_i \\
\mathbf{P}_{\mathbf{y}} &\approx \sum_{i=0}^{2L} W_i^{(c)} \left\{\boldsymbol{\mathcal{Y}}_i - \bar{\mathbf{y}}\right\} \left\{\boldsymbol{\mathcal{Y}}_i - \bar{\mathbf{y}}\right\}^T
\end{aligned}$$

with the weights $W_i$ given by:

$$\begin{aligned}
W_0^{(m)} &= \lambda/(L+\lambda) \\
W_0^{(c)} &= \lambda/(L+\lambda) + (1 - \alpha^2 + \beta) \\
W_i^{(m)} &= W_i^{(c)} = 1/\{2(L+\lambda)\} \qquad i = 1, \ldots, 2L.
\end{aligned}$$

This procedure can be represented in the following block diagram:

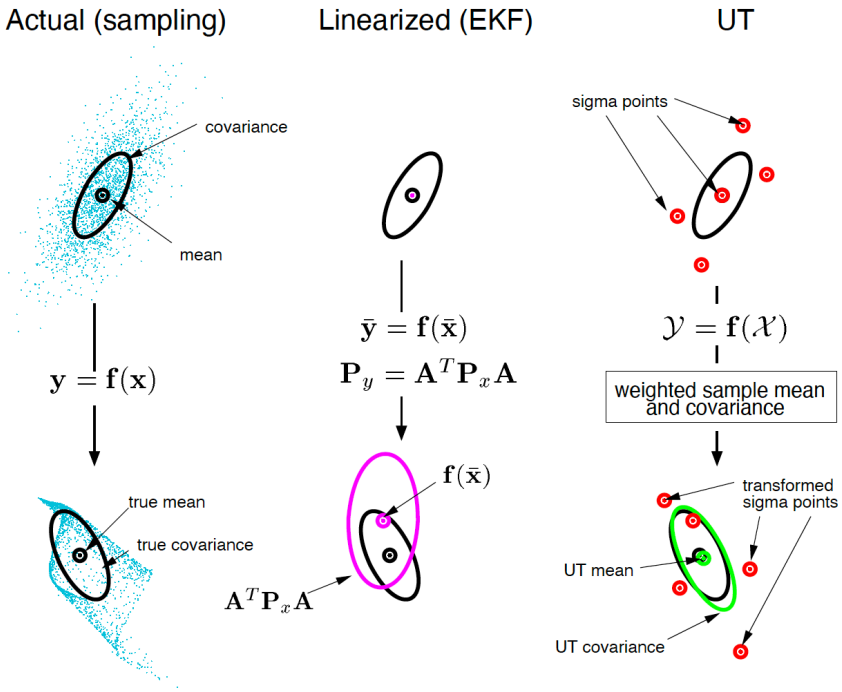



*Unscented Kalman Filter*

The UKF is a simple extension of the UT to recursive estimation

$$\hat{\mathbf{x}}_k \quad = \quad (\text{prediction of } \mathbf{x}_k) + \mathcal{K}_k \cdot [\mathbf{y}_k - (\text{prediction of } \mathbf{y}_k)]$$

Where the RV state is refined as the concatenation of the original state and noise variables: $x_k^a = [x_k^T v_k^T n_k^T]$. This scheme:




is applied to this new augmented RV state to compute the corresponding sigma matrix, $x_k^a$. The UKF equations are as follows:

Initialize with:

$$\begin{aligned}\hat{\mathbf{x}}_0 &= \mathbb{E}[\mathbf{x}_0] \\ \mathbf{P}_0 &= \mathbb{E}[(\mathbf{x}_0 - \hat{\mathbf{x}}_0)(\mathbf{x}_0 - \hat{\mathbf{x}}_0)^T] \\ \hat{\mathbf{x}}_0^a &= \mathbb{E}[\mathbf{x}^a] = [\hat{\mathbf{x}}_0^T \ \mathbf{0} \ \mathbf{0}]^T\end{aligned}$$

$$\mathbf{P}_0^a = \mathbb{E}[(\mathbf{x}_0^a - \hat{\mathbf{x}}_0^a)(\mathbf{x}_0^a - \hat{\mathbf{x}}_0^a)^T] = \begin{bmatrix} \mathbf{P}_0 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{R}^{\mathbf{v}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{R}^{\mathbf{n}} \end{bmatrix}$$

For $k \in \{1, \ldots, \infty\}$,

Calculate sigma points:

$$\boldsymbol{\mathcal{X}}_{k-1}^a = \begin{bmatrix} \hat{\mathbf{x}}_{k-1}^a & \hat{\mathbf{x}}_{k-1}^a + \gamma\sqrt{\mathbf{P}_{k-1}^a} & \hat{\mathbf{x}}_{k-1}^a - \gamma\sqrt{\mathbf{P}_{k-1}^a} \end{bmatrix}$$

Time update:

$$\begin{aligned}\boldsymbol{\mathcal{X}}_{k|k-1}^x &= \mathbf{F}[\boldsymbol{\mathcal{X}}_{k-1}^x, \mathbf{u}_{k-1}, \boldsymbol{\mathcal{X}}_{k-1}^v] \\ \hat{\mathbf{x}}_k^- &= \sum_{i=0}^{2L} W_i^{(m)} \mathcal{X}_{i,k|k-1}^x \\ \mathbf{P}_k^- &= \sum_{i=0}^{2L} W_i^{(c)} [\mathcal{X}_{i,k|k-1}^x - \hat{\mathbf{x}}_k^-][\mathcal{X}_{i,k|k-1}^x - \hat{\mathbf{x}}_k^-]^T \\ \boldsymbol{\mathcal{Y}}_{k|k-1} &= \mathbf{H}[\boldsymbol{\mathcal{X}}_{k|k-1}^x, \boldsymbol{\mathcal{X}}_{k-1}^n] \\ \hat{\mathbf{y}}_k^- &= \sum_{i=0}^{2L} W_i^{(m)} \mathcal{Y}_{i,k|k-1}\end{aligned}$$

Measurement update equations:

$$\begin{aligned}\mathbf{P}_{\tilde{\mathbf{y}}_k \tilde{\mathbf{y}}_k} &= \sum_{i=0}^{2L} W_i^{(c)} [\mathcal{Y}_{i,k|k-1} - \hat{\mathbf{y}}_k^-][\mathcal{Y}_{i,k|k-1} - \hat{\mathbf{y}}_k^-]^T \\ \mathbf{P}_{\mathbf{x}_k \mathbf{y}_k} &= \sum_{i=0}^{2L} W_i^{(c)} [\mathcal{X}_{i,k|k-1} - \hat{\mathbf{x}}_k^-][\mathcal{Y}_{i,k|k-1} - \hat{\mathbf{y}}_k^-]^T \\ \mathcal{K}_k &= \mathbf{P}_{\mathbf{x}_k \mathbf{y}_k} \mathbf{P}_{\tilde{\mathbf{y}}_k \tilde{\mathbf{y}}_k}^{-1} \\ \hat{\mathbf{x}}_k &= \hat{\mathbf{x}}_k^- + \mathcal{K}_k(\mathbf{y}_k - \hat{\mathbf{y}}_k^-) \\ \mathbf{P}_k &= \mathbf{P}_k^- - \mathcal{K}_k \mathbf{P}_{\tilde{\mathbf{y}}_k \tilde{\mathbf{y}}_k} \mathcal{K}_k^T\end{aligned}$$

where, $x^a = [x^T v^T n^T]^T$, $X^a = [(X^x)^T (X^v)^T (X^n)^T]^T$, $\gamma = \sqrt{(L + \lambda)}$, λ= composite scaling parameter, L = size of the augmented state, $R^v$ = process noise covariance, $R^n$ = measurement noise covariance, $W_i$ weights calculated as previously described .

In the event that the process and measurement noise are purely additive, the computational complexity of the UKF is reduced:

Initialize with:

$$\hat{\mathbf{x}}_0 = \mathbb{E}[\mathbf{x}_0]$$
$$\mathbf{P}_0 = \mathbb{E}[(\mathbf{x}_0 - \hat{\mathbf{x}}_0)(\mathbf{x}_0 - \hat{\mathbf{x}}_0)^T]$$

For $k \in \{1, \ldots, \infty\}$,

Calculate sigma points:

$$\boldsymbol{\mathcal{X}}_{k-1} = \begin{bmatrix} \hat{\mathbf{x}}_{k-1} & \hat{\mathbf{x}}_{k-1} + \gamma\sqrt{\mathbf{P}_{k-1}} & \hat{\mathbf{x}}_{k-1} - \gamma\sqrt{\mathbf{P}_{k-1}} \end{bmatrix}$$

Time update:

$$\begin{aligned}
\boldsymbol{\mathcal{X}}_{k|k-1} &= \mathbf{F}[\boldsymbol{\mathcal{X}}_{k-1}, \mathbf{u}_{k-1}] \\
\hat{\mathbf{x}}_k^- &= \sum_{i=0}^{2L} W_i^{(m)} \mathcal{X}_{i,k|k-1} \\
\mathbf{P}_k^- &= \sum_{i=0}^{2L} W_i^{(c)} [\mathcal{X}_{i,k|k-1} - \hat{\mathbf{x}}_k^-][\mathcal{X}_{i,k|k-1} - \hat{\mathbf{x}}_k^-]^T + \mathbf{R}^{\mathbf{v}} \\
\boldsymbol{\mathcal{Y}}_{k|k-1} &= \mathbf{H}[\boldsymbol{\mathcal{X}}_{k|k-1}] \\
\hat{\mathbf{y}}_k^- &= \sum_{i=0}^{2L} W_i^{(m)} \mathcal{Y}_{i,k|k-1}
\end{aligned}$$

Measurement update equations:

$$\begin{aligned}
\mathbf{P}_{\tilde{\mathbf{y}}_k \tilde{\mathbf{y}}_k} &= \sum_{i=0}^{2L} W_i^{(c)} [\mathcal{Y}_{i,k|k-1} - \hat{\mathbf{y}}_k^-][\mathcal{Y}_{i,k|k-1} - \hat{\mathbf{y}}_k^-]^T + \mathbf{R}^{\mathbf{n}} \\
\mathbf{P}_{\mathbf{x}_k \mathbf{y}_k} &= \sum_{i=0}^{2L} W_i^{(c)} [\mathcal{X}_{i,k|k-1} - \hat{\mathbf{x}}_k^-][\mathcal{Y}_{i,k|k-1} - \hat{\mathbf{y}}_k^-]^T \\
\mathcal{K}_k &= \mathbf{P}_{\mathbf{x}_k \mathbf{y}_k} \mathbf{P}_{\tilde{\mathbf{y}}_k \tilde{\mathbf{y}}_k}^{-1} \\
\hat{\mathbf{x}}_k &= \hat{\mathbf{x}}_k^- + \mathcal{K}_k(\mathbf{y}_k - \hat{\mathbf{y}}_k^-) \\
\mathbf{P}_k &= \mathbf{P}_k^- - \mathcal{K}_k \mathbf{P}_{\tilde{\mathbf{y}}_k \tilde{\mathbf{y}}_k} \mathcal{K}_k^T
\end{aligned}$$

Where $\gamma = \sqrt{(L + \lambda)}$, $\lambda$ = composite scaling parameter, L = dimension of the augmented state, $R^v$ = covariance of the process noise, $R^n$ = covariance of the measurement noise, $W_i$ weights calculated as previously described.

For more information see Bibliography 7.

**UKF-SEIR implementation with Python**

Let's see now the implementation of the Jupyter Notebook about the implementation of

the UKF-SEIR. Also in this case we take into consideration the civil protection dataset. Before describing the code, however, let's see how this case study was posed. This analysis is different from the previous one of the EKF, since it assumes that the SEIR model is modeled as a stochastic process; in this case the infection rate β is added as a status (the status in this case will consist of [S, E, I, R, β, dβ]) in the SEIR model. Next we used the Unscented Kalman Filter and a smooth, called Unscented Rauch-Tung-Striebel Smoother (see Bibliography 9), for the state estimation. After adding the parameter β in the state we define two equations:

$$\frac{d\beta}{dt} = \dot{\beta}$$

$$\frac{d\dot{\beta}}{dt} = v(t) \sim \mathcal{N}(0, q^2)$$

This means that β cannot be deterministically modeled (unlike the EKF case where the rates were modeled constant) but suppose that it changes with a certain speed and the time derivative of that speed is supposed to be white and normally distributed around zero with a standard deviation q. The term "blank" means that the process v(t) has no memory (the current state is sufficient to describe the current one and any previous information is redundant), i.e. it does not add information on the prediction of future states; while β can be seen as a sort of Markovian process.

For more information about this idea, visit the following link:
https://towardsdatascience.com/estimating-the-effect-of-social-distancing-in-sweden-c6c1e606c8f9

***Constants setting***

Let's start by setting the constants related to the SEIR model.

```
POPULATIONS = {
    #"Abruzzo":   1305770,
    #"Basilicata": 556934,
    #"Calabria": 1924701,
    #"Campania": 5785861,
    #"Emilia-Romagna": 4467118,
    #"Friuli Venezia Giulia": 1211357,
    #"Lazio": 5865544,
    #"Liguria": 1543127,
    #"Lombardia": 10103969,
    #"Marche": 1518400,
    #"Molise": 302265,
    #"P.A. Bolzano": 106951,
    #"P.A. Trento": 117417,
    #"Piemonte": 4341375,
    "Puglia": 4008296,
    #"Sardegna": 1630474,
    #"Sicilia": 4968410,
    #"Toscana": 3722729,
    #"Umbria": 880285,
    #"Valle d'Aosta": 125501,
    #"Veneto": 4907704
}
CASES_CONSIDERED = -1
DATA_SMOOTHING_WINDOW = 1
FORECAST_DAYS = 14
AVERAGE_R0_WINDOW = 7
SIGMA_CONSIDERED = 1

# Length of time an individual broadcasts
t_inf = 1.5
# Length of the incubation period
t_inc = 11.5
# Reproduction Number
R0_init = 5.5
```

Specifically we have:

- POPULATIONS: Json containing the number of inhabitants by corresponding region;
- CASES_CONSIDERED: Number of positive cases considered at the initial moment;
- DATA_SMOOTHING_WINDOW: Size of the smoothing window that will be used to smooth the curves in order to reduce uncertainty;
- AVERAGE_R0_WINDOW: window used to keep the contagiousness measurement constant, taking into consideration the average of the data contained in the window;
- SIGMA_CONSIDERED: scalar value used to define the errorbar, ie the standard error, on the prediction, in order to take into account a range of uncertainty;
- t_inf: average length of the period in which an individual broadcasts;
- t_inc: average duration of the incubation period;
- R0: measure of initial contagiousness.

And subsequently those relating to the UKF.

```
# Unscented Kalman Filter (UKF) setup.
dt = 1
dim_x, dim_z = 12, 2
z_std = (1e-2, 5e-3)
var_q = 5e-2
n = dim_x
alpha = 1e-3
beta = 2
kappa = 1
Wc, Wm = compute_weights(n, alpha, beta, kappa)
```

We have:

- dt: discretization step size;
- dim_x: state vector size;
- dim_z: dimension of the measurement vector;
- z_std: tuple of values used to initialize the measurement noise matrix R, according to the one mentioned above;
- var_q: scalar used to initialize the process noise matrix Q, according to the one mentioned above;
- n: scalar equal to the size of the state vector;
- alpha: scalar which determines the spread of the sigma points around $\bar{x}$;
- beta: scalar used to incorporate the a priori knowledge of the x distribution useful for the computation of sigma points;
- kappa: secondary scalar parameter useful for the calculation of sigma points;
- Wc, Wm: call the compute_weights() function to calculate the array of weights relative to the mean and the covariance. These weights will be used in the prediction and update formulas according to the algorithm described above.

Let's look specifically at the compute_weights function:

```
def compute_weights(n, alpha, beta, kappa):
    """ Computes the weights for the scaled unscented Kalman filter.
    Generates sigma points and weights according to Van der Merwe's. It
    parametizes the sigma points using alpha, beta, kappa terms, and
    is the version seen in most publications.

    Parameters
    ----------

    n : int
        Dimensionality of the state. 2n+1 weights will be generated.

    alpha : float
        Determins the spread of the sigma points around the mean.
        Usually a small positive value (1e-3) according to [3].

    beta : float
        Incorporates prior knowledge of the distribution of the mean. For
        Gaussian x beta=2 is optimal, according to [3].

    kappa : float, default=0.0
        Secondary scaling parameter usually set to 0 according to [4],
        or to 3-n according to [5].

    Return
    ----------

    Wm : np.array
        weight for each sigma point for the mean

    Wc : np.array
        weight for each sigma point for the covariance """

    lambda_ = alpha ** 2 * (n + kappa) - n

    c = .5 / (n + lambda_)
    #Return a new array of given shape and type, filled with fill_value.
    Wc = np.full(num_sigmas_func(n), c)
    Wm = np.full(num_sigmas_func(n), c)
    Wc[0] = lambda_ / (n + lambda_) + (1 - alpha ** 2 + beta)
    Wm[0] = lambda_ / (n + lambda_)
    return Wc, Wm
```

This function takes as input method parameters n, alpha, beta, kappa and proceeds to calculate the weights $W_i^{(m)}$ and $W_i^{(c)}$ according to the theory relating to the Ascendent Transformation described above.

After the first phase of initialization of the constants we pass to the reading and sorting of the dataset, which contains the data useful for the forecast.

```
plt.rcParams["figure.figsize"] = [16,9]
data = None
data = cleared_dataframe.head(2520).sort_values(["denominazione_regione", "data"], ascending = (True, True))
#data = cleared_dataframe.sort_values(["denominazione_regione", "data"], ascending = (True, True))
data.tail()
```

***UKF-SEIR initialization***

Now we have everything we need to be able to create the UKF-SEIR prediction algorithm, this algorithm will be cycled on the selected regions in the JSON seen previously; furthermore, the data relating to status and a posteriori covariance will be added (step by step) to the list() type variable called results, used at the end to generate the plots. We set the population of the region cycled by the for loop and filter the data read from the dataset for the corresponding region; we pre-process the data via the preprocess_data() function passing the filtered data.

```
# Run UKF by populations.
results = list()
for k, v in POPULATIONS.items():
    print(k)
    N = v
    data_country = data[data["denominazione_regione"] == k]
    zs = preprocess_data(data_country)
```

Let's see the preprocess_data() function in detail.

```
def preprocess_data(data_country):
    """
    Compute I and R from cumulative Cases, Recovered and Deaths data.
    :param data_country: dataset of the civil protection.
    :return: list of tuple daily (I, R) if the number of confirmed cases > CASES CONSIDERED.
    """
    global DATA_SMOOTHING_WINDOW, CASES_CONSIDERED
    # The zip () function accepts iterables (they can be zero or more), aggregates them into a tuple and returns them.
    # The rolling () function is used to provide the sliding window calculations. Mean in this case.
    return [(c - (r + d), r + d) for c, d, r
            in zip(data_country['totale_casi'].rolling(DATA_SMOOTHING_WINDOW).mean(),
                   data_country['deceduti'].rolling(DATA_SMOOTHING_WINDOW).mean(),
                   data_country['dimessi_guariti'].rolling(DATA_SMOOTHING_WINDOW).mean())
            if c > CASES_CONSIDERED]
```

We take the data from the dataset passed through the data_country variable, we apply the zip function used to aggregate iterable objects and we obtain the data relating to the total positive cases, the deceased and the discharged healed; on these data we apply the rolling function that applies a mobile calculation window obtaining the average, and from all this we go to calculate and return the Infected (total_cases - removed) and the Removed in a list of tuple.

Now let's create the vectors and matrices that we are going to initialize and use in the UKF.

```
# Initialize
x = np.zeros(dim_x)
P = np.eye(dim_x)
Q = np.eye(dim_x)
R = np.eye(dim_z)
# Number of sigma points for each variable in the state x
num_sigmas = num_sigmas_func(n)
# sigma points transformed through f(x) and h(x)
# variables for efficiency so we don't recreate every update
sigmas_f = np.zeros((num_sigmas, dim_x))
sigmas_h = np.zeros((num_sigmas, dim_z))
# Kalman gain
K = np.zeros((dim_x, dim_z))
# residual
y = np.zeros((dim_z))
# system uncertainty
S = np.zeros((dim_z, dim_z))
# these will always be a copy of x,P after predict() is called
x_prior = x.copy()
P_prior = P.copy()
# these will always be a copy of x,P after update() is called
x_post = x.copy()
P_post = P.copy()
```

In detail, the variables are the following:

- x (1x12): state vector defined as $x = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$;
- P (12x12): state error covariance matrix defined as $P = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$;
- Q (12x12): process error matrix defined as $Q = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$;
- R (2x2): measurement error matrix defined as $E = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$;
- num_sigmas: number of sigma points for each state variable x calculated by calling the num_sigma_func function which takes n as input and returns (2 * n + 1);

```
def num_sigmas_func(n):
    """ Number of sigma points for each variable in the state x"""
    return 2 * n + 1
```

- sigmas_f (25x12): matrix containing the sigma points transformed through the nonlinear function f (relative to the state) of the SEIR model defined as sigma_f = $\begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$;
- sigma_h (25x2): matrix containing the sigma points transformed through the nonlinear function h (relative to the measures) of the SEIR model defined as sigma_h = $\begin{bmatrix} 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \end{bmatrix}$;

- K (25x2): Kalman gain matrix defined as K = $\begin{bmatrix} 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \end{bmatrix}$;
- y (1x2): residual vector $y_k - \widehat{y_k}$;
- S (2x2): system uncertainty matrix S = $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$;
- x_prior (1x12): contains a copy of the a priori state after the prediction;
- P_prior (12x12): contains a copy of the covariance matrix of the a priori state error after the prediction;
- x_post (1x12): contains a copy of the status after the update;
- P_post (12x12): contains a copy of the covariance matrix of the post-update status error after the update.

Now we initialize x, P, R and Q as follows.

```
# Inital conditions.
x = np.array([v, 0, 0, 0, zs[0][0], 0, zs[0][1], 0, R0_init, 0, 0, 0])
# Noise setup
# factor on uncertainty of initial condition.
P *= 1e0
# process noise
R = np.diag([z ** 2 for z in list(z_std)])
# measurement noise
Q = Q_discrete_white_noise(dim=dim_z, dt=dt, var=var_q ** 2, block_size=int(dim_x / 2))

# Trace the status and covariance (x_post, P_post) of each step prediction+update
R_index = list()
# Trace Reproduction Number of each step prediction+update
R0 = list()
```

In detail we have:

- x: $x = [N° \, region \, population \quad 0 \quad 0 \quad 0 \quad Infected \, after \, preprocess \quad 0 \quad Removed \, after \, preprocess \quad 0 \quad R0 \, initial \quad 0 \quad 0 \quad 0]$ for Removed and Infected, reference is made to the first useful value in the dataset;
- P: the product of the matrix for a constant of uncertainty is executed;
- R: measurement error matrix whose components are set using the variable z_std previously defined R = $\begin{bmatrix} 1.0e-04 & 0 \\ 0 & 2.5e-05 \end{bmatrix}$;
- Q: process error matrix initialized using the standard function Q_discrete_white_noise which allows to obtain a matrix according to the Discrete Constant White Noise Model (see specifically the book: Bar-Shalom. "Estimation with Applications To Tracking and Navigation". John Wiley & Sons, 2001. Page 274.). This function is shown in the image below, we will not go into detail as it is a standard method for building matrices of this type.

```
def Q_discrete_white_noise(dim, dt, var, block_size):
    """Returns the Q matrix for the Discrete Constant White Noise
    Model. dim may be either 2, 3, or 4 dt is the time step, and sigma
    is the variance in the noise.

    Q is computed as the G * G^T * variance, where G is the process noise per
    time step. In other words, G = [[.5dt^2][dt]]^T for the constant velocity
    model.

    Parameters
    -----------

    dim : int (2, 3, or 4)
        dimension for Q, where the final dimension is (dim x dim)

    dt : float, default=1.0
        time step in whatever units your filter is using for time. i.e. the
        amount of time between innovations

    var : float, default=1.0
        variance in the noise

    block_size : int >= 1
        If your state variable contains more than one dimension, such as
        a 3d constant velocity model [x x' y y' z z']^T, then Q must be
        a block diagonal matrix.

    Examples
    --------
    >>> # constant velocity model in a 3D world with a 10 Hz update rate
    >>> Q_discrete_white_noise(2, dt=0.1, var=1., block_size=3)
    array([[0.000025, 0.0005  , 0.      , 0.      , 0.      , 0.      ],
           [0.0005  , 0.01    , 0.      , 0.      , 0.      , 0.      ],
           [0.      , 0.      , 0.000025, 0.0005  , 0.      , 0.      ],
           [0.      , 0.      , 0.0005  , 0.01    , 0.      , 0.      ],
           [0.      , 0.      , 0.      , 0.      , 0.000025, 0.0005  ],
           [0.      , 0.      , 0.      , 0.      , 0.0005  , 0.01    ]])

    References
    ----------

    Bar-Shalom. "Estimation with Applications To Tracking and Navigation".
    John Wiley & Sons, 2001. Page 274.
    """

    if not (dim == 2 or dim == 3 or dim == 4):
        raise ValueError("dim must be between 2 and 4")

    if dim == 2:
        Q = [[.25*dt**4, .5*dt**3],
             [ .5*dt**3,    dt**2]]
    elif dim == 3:
        Q = [[.25*dt**4, .5*dt**3, .5*dt**2],
             [ .5*dt**3,    dt**2,       dt],
             [ .5*dt**2,       dt,        1]]
    else:
        Q = [[(dt**6)/36, (dt**5)/12, (dt**4)/6, (dt**3)/6],
             [(dt**5)/12, (dt**4)/4,  (dt**3)/2, (dt**2)/2],
             [(dt**4)/6,  (dt**3)/2,   dt**2,     dt],
             [(dt**3)/6,  (dt**2)/2 ,  dt,        1.]]

    return block_diag(*[Q]*block_size) * var
```

This function takes as input: the variable dim (in our case it corresponds to dim_z: dimension of the measurement vector), the time step dt (1 day in our case), the variance in the noise var (in our case it corresponds to var_q$^2$), the variable block_size (equal to the previously defined dim_x divided by 2). The final matrix will be obtained by calling the block_diag function which accepts as input an array of arguments (matrices) to create a diagonal block matrix, at the end we will obtain a matrix (12x12) representing the White Gaussion noise.

```
def block_diag(*arrs):
    """Create a block diagonal matrix from provided arrays.

    Given the inputs `A`, `B` and `C`, the output will have these
    arrays arranged on the diagonal::

        [[A, 0, 0],
         [0, B, 0],
         [0, 0, C]]

    Parameters
    ----------
    A, B, C, ... : array_like, up to 2-D
        Input arrays.  A 1-D array or array_like sequence of length `n` is
        treated as a 2-D array with shape ``(1,n)``.

    Returns
    -------
    D : ndarray
        Array with `A`, `B`, `C`, ... on the diagonal.  `D` has the
        same dtype as `A`.

    Notes
    -----
    If all the input arrays are square, the output is known as a
    block diagonal matrix.

    Empty sequences (i.e., array-likes of zero size) will not be ignored.
    Noteworthy, both [] and [[]] are treated as matrices with shape ``(1,0)``.

    Examples
    --------
    >>> from scipy.linalg import block_diag
    >>> A = [[1, 0],
    ...      [0, 1]]
    >>> B = [[3, 4, 5],
    ...      [6, 7, 8]]
    >>> C = [[7]]
    >>> P = np.zeros((2, 0), dtype='int32')
    >>> block_diag(A, B, C)
    array([[1, 0, 0, 0, 0, 0],
           [0, 1, 0, 0, 0, 0],
           [0, 0, 3, 4, 5, 0],
           [0, 0, 6, 7, 8, 0],
           [0, 0, 0, 0, 0, 7]])
    >>> block_diag(A, P, B, C)
    array([[1, 0, 0, 0, 0, 0],
           [0, 1, 0, 0, 0, 0],
           [0, 0, 0, 0, 0, 0],
           [0, 0, 0, 0, 0, 0],
           [0, 0, 3, 4, 5, 0],
           [0, 0, 6, 7, 8, 0],
           [0, 0, 0, 0, 0, 7]])
    >>> block_diag(1.0, [2, 3], [[4, 5], [6, 7]])
    array([[ 1.,  0.,  0.,  0.,  0.],
           [ 0.,  2.,  3.,  0.,  0.],
           [ 0.,  0.,  0.,  4.,  5.],
           [ 0.,  0.,  0.,  6.,  7.]])"""

    if arrs == ():
        arrs = ([],)
    arrs = [np.atleast_2d(a) for a in arrs]

    bad_args = [k for k in range(len(arrs)) if arrs[k].ndim > 2]
    if bad_args:
        raise ValueError("arguments in the following positions have dimension "
                         "greater than 2: %s" % bad_args)

    shapes = np.array([a.shape for a in arrs])
    out_dtype = np.find_common_type([arr.dtype for arr in arrs], [])
    out = np.zeros(np.sum(shapes, axis=0), dtype=out_dtype)

    r, c = 0, 0
    for i, (rr, cc) in enumerate(shapes):
        out[r:r + rr, c:c + cc] = arrs[i]
        r += rr
        c += cc
    return out
```

• R_index: empty list that will contain the state and the covariance after each prediction and update step;

• R0: empty list that will contain the Reproduction Number after each prediction and update step.

***We start the UKF forecasting algorithm***

Now let's get to the heart of the code, where we show how the UKF is launched and the forecast and update function defined. First we cycle on the measures and then alternate the prediction and the update; from time to time we keep track of the predictions in the list variables seen above.

```
# RUN UKF
# Derive all hidden variables from past and present.
for z in zs:
    #Prediction
    x, P, sigmas_f = predict(x, P, Wm, Wc, Q, sigmas_f)
    x_prior, P_prior = x, P
    #Update
    x, P, y, z, sigmas_h, S = update(z, R, sigmas_f, Wm, Wc, x_prior, P_prior)
    x_post, P_post = x, P

    x, y = x_post, P_post
    R0.append(x[8])
    R_index.append((x, y))
```

*Predict function*

```
def predict(x, P, Wm, Wc, Q, sigmas_f):
    global n, alpha, kappa, dt

    """
    Performs the predict step of the UKF. On return, x and
    P contain the predicted state (x) and covariance (P). """

    # calculate sigma points for given mean and covariance
    sigmas = sigma_points(x, P)

    for i, s in enumerate(sigmas):
        sigmas_f[i] = fx(s, dt)

    #and pass sigmas through the unscented transform to compute prior
    x_pior, P_pior = unscented_transform(sigmas_f, Wm, Wc, Q)
    return x_pior, P_pior, sigmas_f
```

This function deals with the prediction step, taking as input the state vector x, the covariance matrix of the state error P, the weights relative to the mean and covariance Wm and Wc, the process error matrix Q and the sigma_f matrix containing the sigma points transformed through the function f. First we pass x and P to the sigma_points function (this function faithfully reflects the theoretical part for the calculation of these points seen previously) to calculate the new sigma points for a given mean and covariance.

```python
def sigma_points(x, P):
    global n, alpha, kappa
    """ Computes the sigma points for an unscented Kalman filter
    given the mean (x) and covariance(P) of the filter.
    Returns tuple of the sigma points and weights.

    Works with both scalar and array inputs:
    sigma_points (5, 9, 2) # mean 5, covariance 9
    sigma_points ([5, 2], 9*eye(2), 2) # means 5 and 2, covariance 9I

    Parameters
    ----------

    x : An array-like object of the means of length n
    Can be a scalar if 1D.
    examples: 1, [1,2], np.array([1,2])

    P : scalar, or np.array
    Covariance of the filter. If scalar, is treated as eye(n)*P.

    Returns
    -------

    sigmas : np.array, of size (n, 2n+1)
    Two dimensional array of sigma points. Each column contains all of
    the sigmas for one dimension in the problem space.

    Ordered by Xi_0, Xi_{1..n}, Xi_{n+1..2n}
    """

    if n != np.size(x):
        raise ValueError("expected size(x) {}, but size is {}".format(n, np.size(x)))

    #Returns True if the type of element is a scalar type.
    if np.isscalar(x):
        #Convert the input to an array.
        x = np.asarray([x])

    #Returns True if the type of element is a scalar type.
    if  np.isscalar(P):
        P = np.eye(n)*P
    else:
        # numpy.atleast_2d() function is used when we want to Convert inputs to arrays with at least two dimension.
        # Scalar and 1-dimensional inputs are converted to 2-dimensional arrays, whilst higher-dimensional inputs are preserved.
        P = np.atleast_2d(P)

    lambda_ = alpha**2 * (n + kappa) - n
    #Compute the Cholesky decomposition of a matrix.
    # Returns the Cholesky decomposition, A = L L^* or A = U^* U of a Hermitian positive-definite matrix A.
    U = scipy.linalg.cholesky((lambda_ + n)*P)

    sigmas = np.zeros((2*n+1, n))
    sigmas[0] = x
    for k in range(n):
        # pylint: disable=bad-whitespace
        sigmas[k+1]   = np.subtract(x, -U[k])
        sigmas[n+k+1] = np.subtract(x, U[k])

    return sigmas
```

Once the vectors containing the sigma points have been obtained, it is necessary to propagate them through the non-linear SEIR model, that is the function fx, to obtain the updated sigma_f matrix. The function fx accepts as input the vector relating to the sigma points (state vector 1x12) and the step size dt and faithfully applies the differential equations of the SEIR model seen previously and returns an array containing the 12 calculated values relating to the state variables [ dS, 0, dE, 0, dI, 0, dR, 0, dR0 if dR> 0 otherwise 0, ddR0, 0, 0] added to the old x containing Susceptible, Infected and Removed (compartment update).

```
def fx(x, delta_t):

    global t_inc, t_inf, N

    S = x[0]
    E = x[2] if x[2] >= 0 else 0
    I = x[4]
    R0 = x[8] if x[8] >= 0 else 0

    dS_temp = (R0 / t_inf) * (S / N) * I * delta_t
    dE_temp = (E / t_inc) * delta_t

    dS = - dS_temp
    dE = dS_temp - dE_temp
    dR = (I / t_inf) * delta_t
    dI = dE_temp - dR
    dR0 = x[9] * delta_t
    ddR0 = x[10] * delta_t

    x[1] = dS
    x[3] = dE
    x[5] = dI
    x[7] = dR
    x[9] = ddR0

    return x + np.array([dS, 0, dE, 0, dI, 0, dR if dR > 0 else 0, 0, dR0, ddR0, 0, 0], dtype=float)
```

After passing the sigma points to the non-linear function, we proceed with the a priori calculation for x and P through the unscented_transform function that allows you to perform the UT according to the UKF theory already seen. This function accepts as input the sigmas_f matrix, the weights Wm and Wc and the White Gaussian process noise matrix (NB this function is used both for the prediction phase and for the update phase), then applies the formulas of updated views for x and P (obtaining the state estimate and a priori covariance).

```
def unscented_transform(sigmas, Wm, Wc, noise_cov=None):
    """
    Computes unscented transform of a set of sigma points and weights.
    returns the mean and covariance in a tuple.

    This works in conjunction with the UnscentedKalmanFilter class.


    Parameters
    ----------

    sigmas: ndarray, of size (n, 2n+1)
        2D array of sigma points.

    Wm : ndarray [# sigmas per dimension]
        Weights for the mean.


    Wc : ndarray [# sigmas per dimension]
        Weights for the covariance.

    noise_cov : ndarray, optional
        noise matrix added to the final computed covariance matrix.

    Returns
    -------

    x : ndarray [dimension]
        Mean of the sigma points after passing through the transform.

    P : ndarray
        covariance of the sigma points after passing throgh the transform.
    """

    # dot = \Sigma^n_1 (W[k]*Xi[k])
    x = np.dot(Wm, sigmas)

    # new covariance is the sum of the outer product of the residuals
    # times the weights

    # this is the fast way to do this - see 'else' for the slow way
    #numpy.newaxis is used to increase the dimension of the existing array by one more dimension, when used once.
    y = sigmas - x[np.newaxis, :]
    #np.diag convert vector Wc in diagonal matrix
    P = np.dot(y.T, np.dot(np.diag(Wc), y))

    if noise_cov is not None:
        P += noise_cov

    return (x, P)
```

## *Update function*

```
def update(z, R, sigmas_f, Wm, Wc, x_prior, P_prior):
    global log_likelihood, likelihood, mahalanobis
    """
    Update the UKF with the given measurements. On return,
    x and P contain the new post mean and covariance of the filter.

    Parameters
    ----------

    z : numpy.array of shape (dim_z)
        measurement vector

    R : numpy.array((dim_z, dim_z)), optional
        Measurement noise. If provided, overrides R for
        this function call.

    sigmas: ndarray, of size (n, 2n+1)
        2D array of sigma points.

    Wm : ndarray [# sigmas per dimension]
        Weights for the mean.


    Wc : ndarray [# sigmas per dimension]
        Weights for the covariance.

    x_prior: prior predicted state.

    P_prior: prior predicted covariance."""

    # pass prior sigmas through h(x) to get measurement sigmas
    # the shape of sigmas_h will vary if the shape of z varies, so
    # recreate each time
    sigmas_h = []
    for s in sigmas_f:
        sigmas_h.append(hx(s))

    sigmas_h = np.atleast_2d(sigmas_h)

    # mean and covariance of prediction passed through unscented transform
    zp, S = unscented_transform(sigmas_h, Wm, Wc, R)

    # compute cross variance of the state and the measurements
    Pxz = cross_variance(x_prior, zp, sigmas_f, sigmas_h, Wc)

    # Kalman gain
    K = np.dot(Pxz, np.linalg.inv(S))
    # Residual
    y = np.subtract(z, zp)

    # update Gaussian state estimate (x, P) posterior state
    x = x_prior + np.dot(K, y)
    P = P_prior - np.dot(K, np.dot(S, K.T))

    # save measurement
    #A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.
    z = deepcopy(z)

    return  x, P, y, z, sigmas_h, S
```

This function deals with the update step, taking as input the vector of measurements z, the measurement noise matrix R, the sigma calculated through the SEIR model, the weights Wc and Wm and the a priori predictions relating to state and covariance. First we pass the sigma calculated through the SEIR model to the prediction step to the function h to obtain the sigma relative to the measure, called sigma_h. These sigma calculated together with the weights and the measurement noise matrix R will be passed to the unscented_transform function (seen in the prediction step) to calculate the estimate of the measure zp and covariance S. For the gain calculation formula it will be necessary to calculate the cross covariance Pxz (according to the theory seen) by means of the cross_variance function which accepts as input the a priori state of the calculated measurement estimate, the sigma relative to the calculation using the function f and h and the weights Wc:

```
def cross_variance(x, z, sigmas_f, sigmas_h, Wc):
    """
    Compute cross variance of the state `x` and measurement `z`.
    """
    #np.zeros: Return a new array of given shape and type, filled with zeros.
    Pxz = np.zeros((sigmas_f.shape[1], sigmas_h.shape[1]))
    # shape: Return the shape of an array.
    N = sigmas_f.shape[0]
    for i in range(N):
        dx = np.subtract(sigmas_f[i], x)
        dz = np.subtract(sigmas_h[i], z)
        #np.outer: Compute the outer product of two vectors.
        Pxz += Wc[i] * np.outer(dx, dz)
    return Pxz
```

The cross variance calculated is used to calculate the gain K, we also calculate the residuals of measurement as the difference between real and estimated measurement. Now we have everything to be able to apply the a posteriori calculation of state and covariance according to the above formulas.

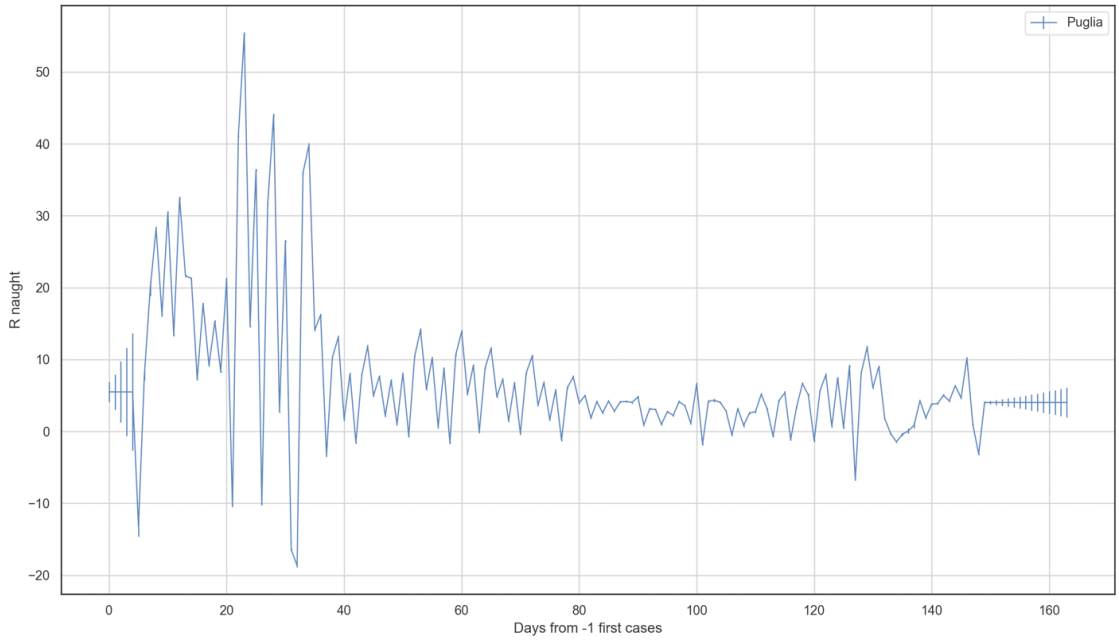
After cycling through the available measures we can proceed with the forward prediction on the FORECAST_DAYS time window; in this case we will only carry out the prediction step given the last posterior estimate.

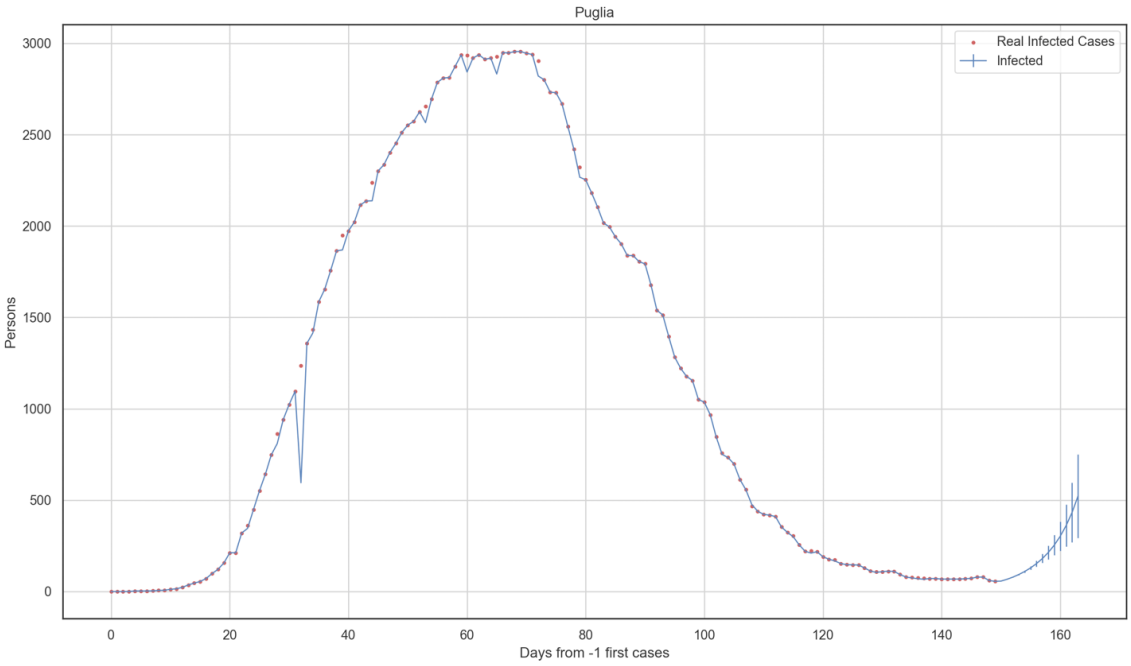
```
    # Forward prediction
    for i in range(FORECAST_DAYS):
        # Keep R0 constant for predictions.
        x[8] = np.mean(R0[-AVERAGE_R0_WINDOW:])
        x[9] = 0
        x[10] = 0

        try:
            x, P, sigmas_f = predict(x, P, Wm, Wc, Q, sigmas_f)
            x_prior, P_prior = x, P
            x, y = x_prior, P_prior
            R_index.append((x, y))
        except np.linalg.LinAlgError:
            print("Cannot predict %d" % i)
    results.append(R_index)
```

Finally, we proceed to the plot of the Rt index, of the predicted positive and removed individuals, with relative errorbar on the predicted time window forward in time. This uncertainty is calculated by multiplying the square root of the relative component of the state error covariance matrix by a preset SIGMA_CONSIDERED coefficient.

```
# Plot population curves I, E and R.
for r, t in zip(results, POPULATIONS.keys()):
    xs = range(len(r))

    # Pass array of predicted values to smooth function, with a certain smoothing windows
    # Use the Smooth function with window size other than 1 to smooth the response data.
    I = smooth([a[0][4] for a in r], DATA_SMOOTHING_WINDOW)
    # Return the non-negative square-root of an array, element-wise. Multiplied for a costant
    sI = [SIGMA_CONSIDERED * np.sqrt(a[1][4, 4]) for a in r]

    #R = smooth([a[0][6] for a in r], DATA_SMOOTHING_WINDOW)
    #sR = [SIGMA_CONSIDERED * np.sqrt(a[1][6, 6]) for a in r]

    # E = smooth([a[0][2] for a in r], DATA_SMOOTHING_WINDOW)
    # sE = [SIGMA_CONSIDERED * np.sqrt(a[1][2, 2]) for a in r]

    '''Traccia y contro x come linee e / o marker con errorbars attaccati.
    x, y definiscono le posizioni dei dati, xerr, yerr definiscono le dimensioni dell'errorbar. Per impostazione predefinita,
    questo disegna i marcatori / linee di dati e le errorbars. Usa fmt = 'none' per disegnare errorbars senza alcun indicatore di dati.'''
    plt.errorbar(x=xs, y=I, yerr=sI, elinewidth=1, markeredgewidth=1, linewidth=1)
    #plt.errorbar(x=xs, y=R, yerr=sR, elinewidth=1, markeredgewidth=1)
    # plt.errorbar(x=xs, y=E, yerr=sE, elinewidth=1, markeredgewidth=1)

    temp_data = data[data['denominazione_regione']==t]
    tvec_real_data_L=np.arange(0,len(temp_data[temp_data['denominazione_regione']==t][['data']]))
    plt.scatter(tvec_real_data_L, temp_data[temp_data['denominazione_regione']==t][['totale_positivi']], marker ='o', c = 'r',s=5)
    #plt.scatter(tvec_real_data_L, temp_data[temp_data['denominazione_regione']==t][['dimessi_guariti']], marker ='o', c = 'g',s=1)

    plt.title(t)
    plt.xlabel("Days from "+str(CASES_CONSIDERED)+" first cases")
    plt.ylabel("Persons")
    #plt.legend(["Real Infected Cases", "Real Recovered Cases", "Infected", "Recovered", "Exposed"])
    plt.legend(["Real Infected Cases", "Infected", "Recovered", "Exposed"])
    plt.grid(True)
    plt.figure()

# Plot R0.
for r, t in zip(results, POPULATIONS.keys()):
    xs = range(len(r))
    R0 = smooth([a[0][8] for a in r], DATA_SMOOTHING_WINDOW)
    sR0 = [SIGMA_CONSIDERED * np.sqrt(a[1][8, 8]) for a in r]
    plt.errorbar(x=xs, y=R0, yerr=sR0, elinewidth=1, markeredgewidth=1, linewidth=1)

plt.legend(POPULATIONS.keys())
plt.xlabel("Days from "+str(CASES_CONSIDERED)+" first cases")
plt.ylabel("R naught")
plt.grid(True)
plt.show()
```

The plots obtained are the following:




***Conclusion***

As you can see and understand, the following case study deals with the prediction of the SEIR model curves stochastically, this represents an advantage in the analysis of compartmental models as it offers medium-long term predictions. This advantage is linked to the modeling seen for the variable infection rate $\beta$, which as seen in the plot is accurately predicted. Of course, the main difficulty for the prediction with compartmental models, as already seen above, is related to the initialization of their initial conditions, additive to those relating to the initialization of the UKF. An a priori informational study is therefore necessary for accurate tuning and consequently an accurate prediction; it is advisable to observe the previous pandemic evolution combined with data known in the

literature. We can therefore conclude that, although the real values have not been found to perform a correct tuning (an incorrect initial setting could lead to divergence), the plot relative to the Infected assumes a fairly accurate prediction. This finding was possible because the measures relating to the first 150 days were used starting from 02.24.2020 (therefore from the end of February to the end of July) and then predicting the following 14 days; in fact, from past knowledge of the pandemic evolution in the summer period, the curve relating to the infected was almost close to 0, subsequently (from August) it started to rise again and restrictive measures are still in force on the national territory.

**APPENDIX**

**[1] Mean Squared Error**

In statistics, Mean Squared Error (MSE) indicates the mean square discrepancy between the observed data values and the estimated data values. The Mean Squared Error (MSE) of an estimator $\hat{\theta}$ with respect to the estimated parameter $\theta$ is defined as:

$$MSE(\hat{\theta}) = E\left[(\hat{\theta} - \theta)^2\right].$$

The mean square error is equal to the sum of the variance and the square of the bias of an estimator:

$$MSE(\hat{\theta}) = Var(\hat{\theta}) + (Bias(\hat{\theta}, \theta))^2.$$

The mean square error therefore gives us a measure for judging the quality of an estimator in terms of its variation and distortion.

**[2] Root Mean Square Error**

The RMSE (Root Mean Squared Error) value is a measure of absolute error in which deviations are squared to prevent positive and negative values from canceling each other out. Furthermore, with this measure, the errors of greater value are amplified, a characteristic that can facilitate the elimination of the methods that have the most significant errors.

$$RMSE(\hat{\theta}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(S_i -= O_i)^2}$$

where $O_i$ are the observations, $S_i$ the predicted values and n the number of observations available for analysis. RMSE is a good measure of accuracy, but only for comparing prediction errors of different models or model setups for a particular variable and not between variables, as it depends on scale.

**[3] Mean Absolute Error**

MAE is a model evaluation metric used with regression models. The average absolute error of a model against a test set is the average of the absolute values of the individual prediction errors across all instances of the test set. Each prediction error is the difference between the true value and the predicted value for the instance.

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} = \frac{\sum_{i=1}^{n} |e_i|}{n}$$

The EAW is therefore an arithmetic mean of the absolute errors $| e_i | = | y_i - x_i |$, where $y_i$

are the predicted values and $x_i$ the true values.
Note that alternative formulations may include relative frequencies as weight factors. The mean absolute error uses the same scale as the measured data. This is known as a scale-dependent measure of accuracy and therefore cannot be used to make comparisons between series using different scales. Mean absolute error is a common measure of forecast error in time series analysis, sometimes used in confusion with the more standard definition of mean absolute deviation.

**BIBLIOGRAPHY**


1. https://link.springer.com/article/10.1007/s10489-020-01948-1
2. https://www.medrxiv.org/content/10.1101/2020.04.27.20079962v1.full.pdf
3. https://www.researchgate.net/publication/342097277_A_new_estimation_method_for_COVID-19_time-varying_reproduction_number_using_active_cases
4. https://www.medrxiv.org/content/10.1101/2020.10.14.20212878v1.full#disp-formula-2
5. https://link.springer.com/article/10.1007/s10489-020-01948-1#Sec2
6. https://link.springer.com/article/10.1007/s00466-020-01911-4
7. https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjbhYWM06juAhUSPOwKHfzhDhUQFjAAegQIARAC&url=https%3A%2F%2Fwww.researchgate.net%2Fprofile%2FMohamed_Mourad_Lafifi%2Fpost%2FHow_to_create_state_transition_function_of_a_AR2_model_for_unscentedKalmanFilter_object_in_MATLAB%2Fattachment%2F5ede7a2a39f1f300016271da%2FAS%253A900211564089344%2540591638570015%2Fdownload%2Fukf.wan_.chapt7_.pdf&usg=AOvVaw3r5tLFHkVeONS2srqSA9IW
8. INTERNATIONAL JOURNAL OF ADAPTIVE CONTROL AND SIGNAL PROCESSING. Output outlier robust state estimation. Daniela De Palma and Giovanni Indiveri.
9. https://users.aalto.fi/~ssarkka/pub/uks-preprint.pdf
10. https://www.medrxiv.org/content/10.1101/2020.02.11.20022186v5.full.pdf
11. Bar-Shalom. "Estimation with Applications To Tracking and Navigation". John Wiley & Sons, 2001
12. https://amslaurea.unibo.it/3093/1/Andraghetti_Sara_tesi.pdf