COVID-19 Trend per Country and Predictions using SIR-D Model

July 7, 2020

0.1 Introduction

This notebook shows the visualization of the coronavirus in country level. The purpose is to further analyze the behavior of the spread in the top countries where confirmed cases went up. We have to separate this to other notebooks because we will solve for the basic reproduction number of COVID-19 and compare it to other coronaviruses (our references were made in Python).

Using a mathematical epidemic model, this notebook will predict the number of cases infected with COVID-19.

- Arrangement of dataset (All countries)
- Trend analysis
- Improvement of model (SIR, SIR-D)

Note: “Infected” means the currently infected cases. This can be calculated as “Confirmed” - “Deaths” - “Recovered”

0.2 Part 1: Country-level Analysis

0.2.1 Importing the Required Packages

[1]:
```
from datetime import timedelta
from dateutil.relativedelta import relativedelta
import os
from pprint import pprint
import warnings
from fbprophet import Prophet
from fbprophet.plot import add_changepoints_to_plot
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from matplotlib.ticker import ScalarFormatter
%matplotlib inline
import numpy as np
import optuna
optuna.logging.disable_default_handler()
import pandas as pd
pd.plotting.register_matplotlib_converters()
import seaborn as sns
```

```
from scipy.integrate import solve_ivp

np.random.seed(2019)
os.environ["PYTHONHASHSEED"] = "2019"

plt.style.use("seaborn-ticks")
plt.rcParams["xtick.direction"] = "in"
plt.rcParams["ytick.direction"] = "in"
plt.rcParams["font.size"] = 11.5
plt.rcParams["figure.figsize"] = (9, 6)
```

0.2.2 Visualizing COVID-19 in Country Level

[2]:
```
raw = pd.read_csv("covid_19_data.csv", parse_dates = ['ObservationDate','Last␣
↪Update'])

print (raw.shape)
print ('Last update: ' + str(raw.ObservationDate.max()))
```

```
(6722, 8)
Last update: 2020-03-18 00:00:00
```

[3]:
```
checkdup = raw.groupby(['Country/Region','Province/State','ObservationDate']).
↪count().iloc[:,0]
checkdup[checkdup>1]
```

[3]:
```
Country/Region  Province/State  ObservationDate
Mainland China  Gansu           2020-03-11         2
                                2020-03-12         2
                Hebei           2020-03-11         2
                                2020-03-12         2
Name: SNo, dtype: int64
```

[4]:
```
raw = raw.drop(['SNo', 'Last Update'], axis=1)
raw = raw.rename(columns={'Country/Region': 'Country', 'ObservationDate':
↪'Date'})
# To check null values
raw.isnull().sum()
```

[4]:
```
Date                 0
Province/State    2766
Country              0
Confirmed            0
Deaths               0
Recovered            0
dtype: int64
```

[5]:
```
# Update for Mar 15 data to clean up data
raw.Country = raw.Country.str.replace('Hong Kong SAR', 'Hong Kong')
raw.Country = raw.Country.str.replace('Taipei and environs', 'Taiwan')
raw.Country = raw.Country.str.replace('Republic of Korea', 'South Korea')
raw.Country = raw.Country.str.replace('Viet Nam', 'Vietnam')
raw.Country = raw.Country.str.replace('occupied Palestinian territory',␣
↪'Palestine')
raw.Country = raw.Country.str.replace('Macao SAR', 'Macau')
raw.Country = raw.Country.str.replace('Russian Federation', 'Russia')
raw.Country = raw.Country.str.replace('Republic of Moldova', 'Moldova')
raw.Country = raw.Country.str.replace('Holy See', 'Vatican City')
raw.Country = raw.Country.str.replace('Iran \(Islamic Republic of\)', 'Iran')
```

[6]:
```
daily = raw.sort_values(['Date','Country','Province/State'])
```

[7]:
```
def get_place(row):
    if row['Country'] == 'Italy':
        return 'Italy'
    elif row['Country'] == 'Mainland China':
        return 'China'
    elif row['Country'] == 'South Korea':
        return 'South Korea'
    elif row['Country'] == 'Iran':
        return 'Iran'
    elif row['Country'] == 'Philippines':
        return 'Philippines'
    else: return 'Other Countries'

daily['segment'] = daily.apply(lambda row: get_place(row), axis=1)
```

0.2.3 Latest Update

[8]:
```
latest = daily[daily.Date == daily.Date.max()]
```

[9]:
```
print ('Total confirmed cases: %.d' %np.sum(latest['Confirmed']))
print ('Total death cases: %.d' %np.sum(latest['Deaths']))
print ('Total recovered cases: %.d' %np.sum(latest['Recovered']))
```

```
Total confirmed cases: 214915
Total death cases: 8733
Total recovered cases: 83313
```

[10]:
```
segment1 = latest.groupby('segment').sum()
segment1['Death Rate'] = segment1['Deaths'] / segment1['Confirmed'] * 100
segment1['Recovery Rate'] = segment1['Recovered'] / segment1['Confirmed'] * 100
segment1
```

[10]:

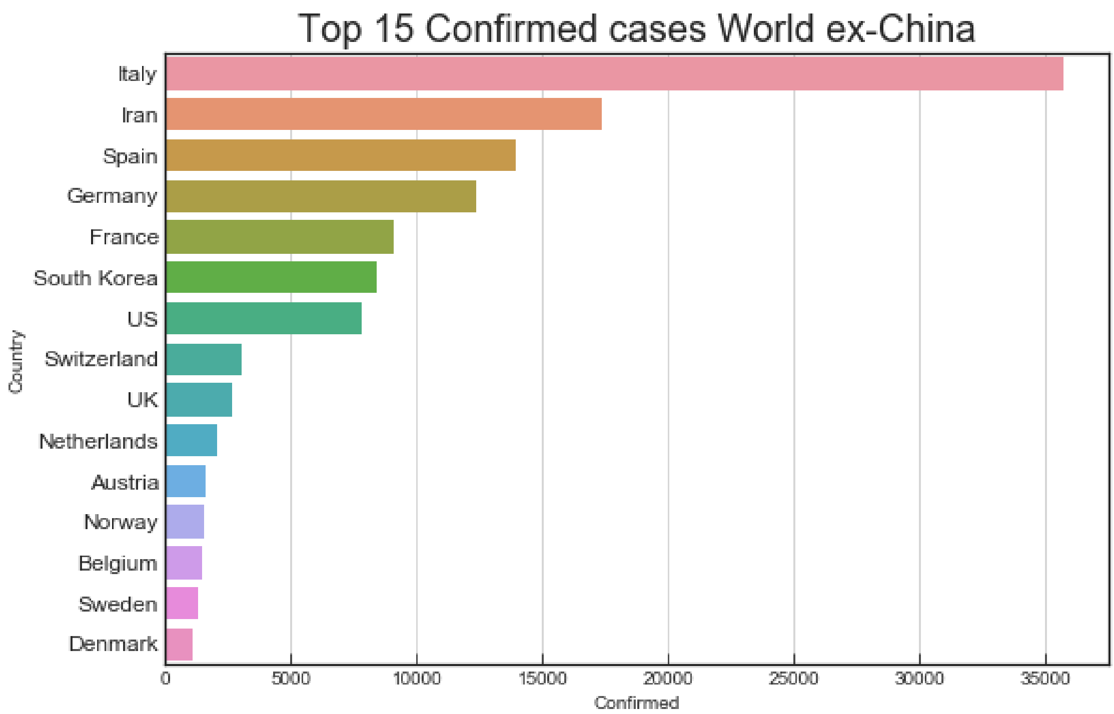
	Confirmed	Deaths	Recovered	Death Rate	Recovery Rate
segment					
China	80906.0	3237.0	69653.0	4.000939	86.091266
Iran	17361.0	1135.0	5389.0	6.537642	31.040839
Italy	35713.0	2978.0	4025.0	8.338700	11.270406
Other Countries	72320.0	1280.0	2701.0	1.769912	3.734790
Philippines	202.0	19.0	5.0	9.405941	2.475248
South Korea	8413.0	84.0	1540.0	0.998455	18.305004

[11]:
```
daily2 = raw.sort_values(['Date','Country','Province/State'])
```

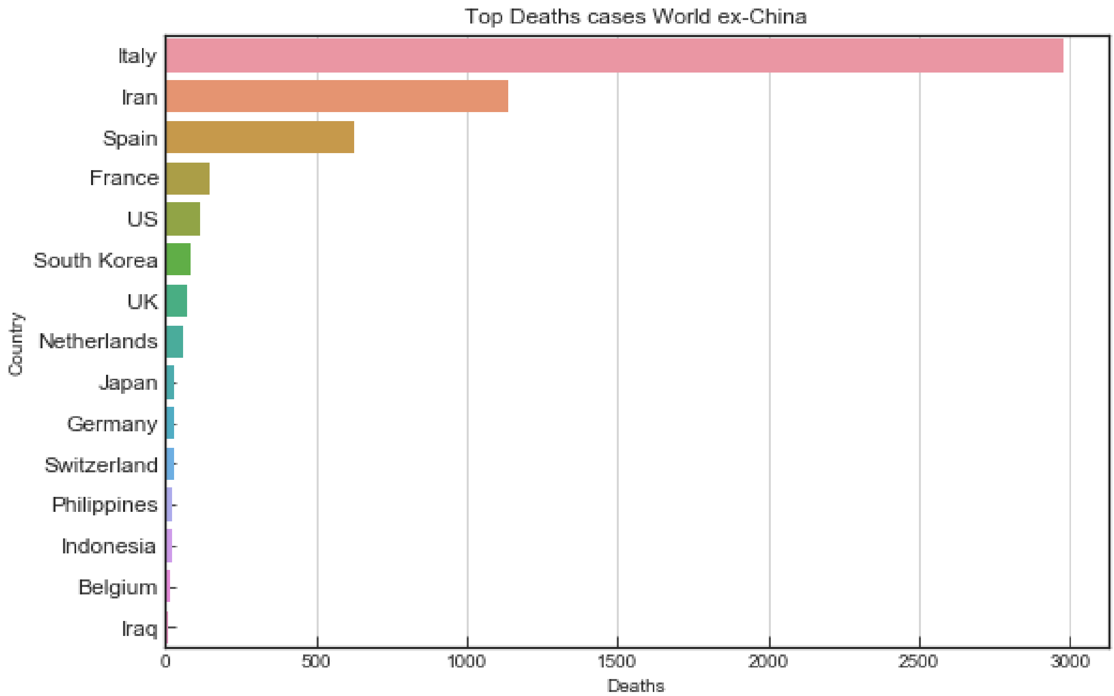
[12]:
```
def get_place_2(row):
    if row['Province/State'] == 'Hubei':
        return 'Hubei PRC'
    elif row['Country'] == 'Mainland China':
        return 'Others PRC'
    else: return 'World'

daily2['segment2'] = daily2.apply(lambda row: get_place_2(row), axis=1)
```

[13]:
```
latest2 = daily2[daily2.Date == daily2.Date.max()]
```

[14]:
```
# Confirmed Cases World ex-China
worldstat = latest2[latest2.segment2=='World'].groupby('Country').sum()
_ = worldstat.sort_values('Confirmed', ascending=False).head(15)
c10 = _.index.tolist() # Will be used in later part
plt.figure(figsize=(9,6))
sns.barplot(_.Confirmed, _.index)
plt.title('Top 15 Confirmed cases World ex-China', size = 20)
plt.yticks(fontsize=12)
plt.grid(axis='x')
plt.show()
```

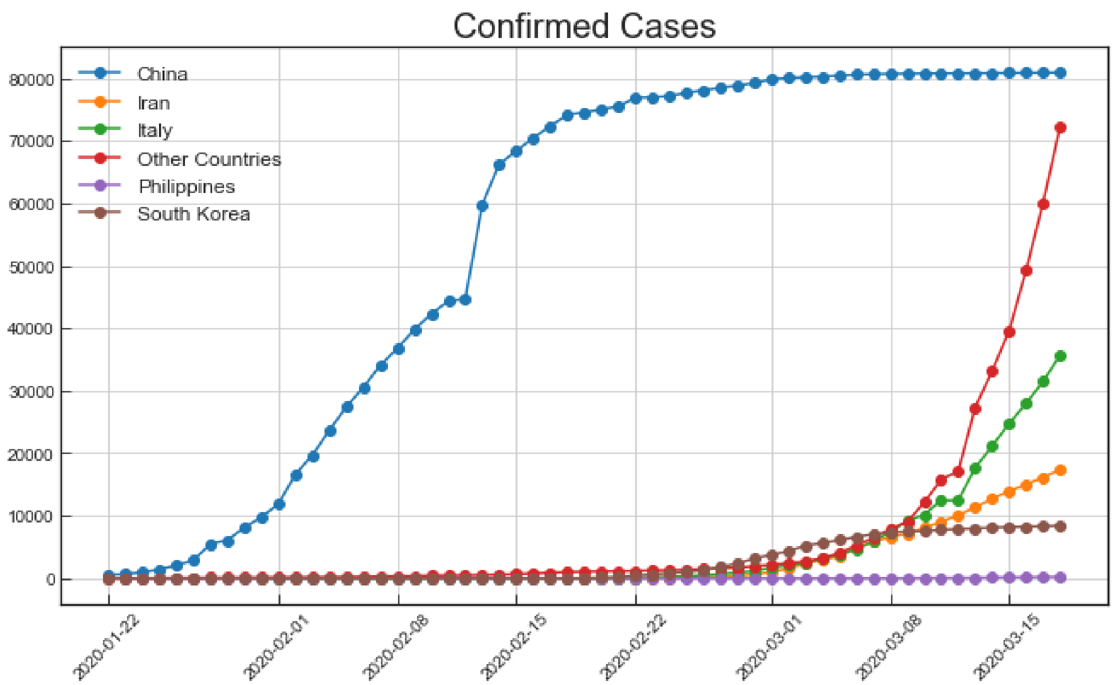


[15]:
```
_ = worldstat.sort_values('Deaths', ascending=False).head(15)
_ = _[_.Deaths>0]
plt.figure(figsize=(9,6))
sns.barplot(_.Deaths, _.index)
plt.title('Top Deaths cases World ex-China')
plt.yticks(fontsize=12)
plt.grid(axis='x')
plt.show()
```



0.2.4 Evolution of cases in Top Countries with Confirmed Cases and the Philippines

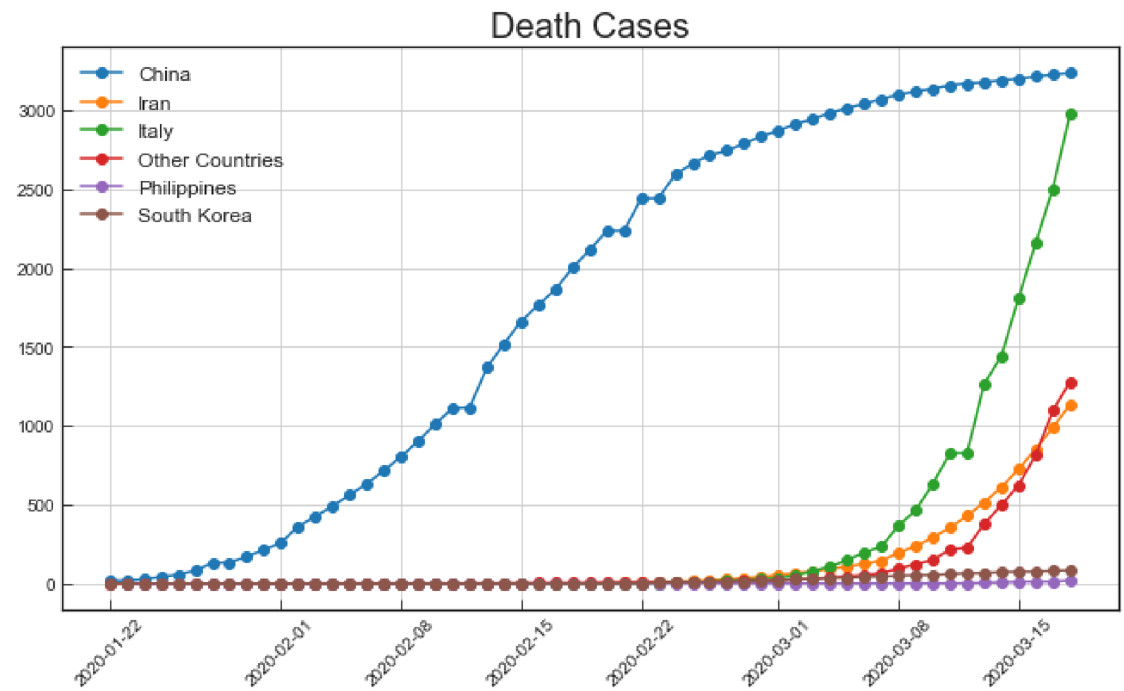
[16]:
```
confirm = pd.pivot_table(daily.dropna(subset=['Confirmed']), index='Date',
                         columns='segment', values='Confirmed', aggfunc=np.sum).
 ↪fillna(method='ffill')
plt.figure(figsize=(11,6))
plt.plot(confirm, marker='o')
plt.title('Confirmed Cases', size = 20)
plt.legend(confirm.columns, loc=2, fontsize=12)
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```



Findings:

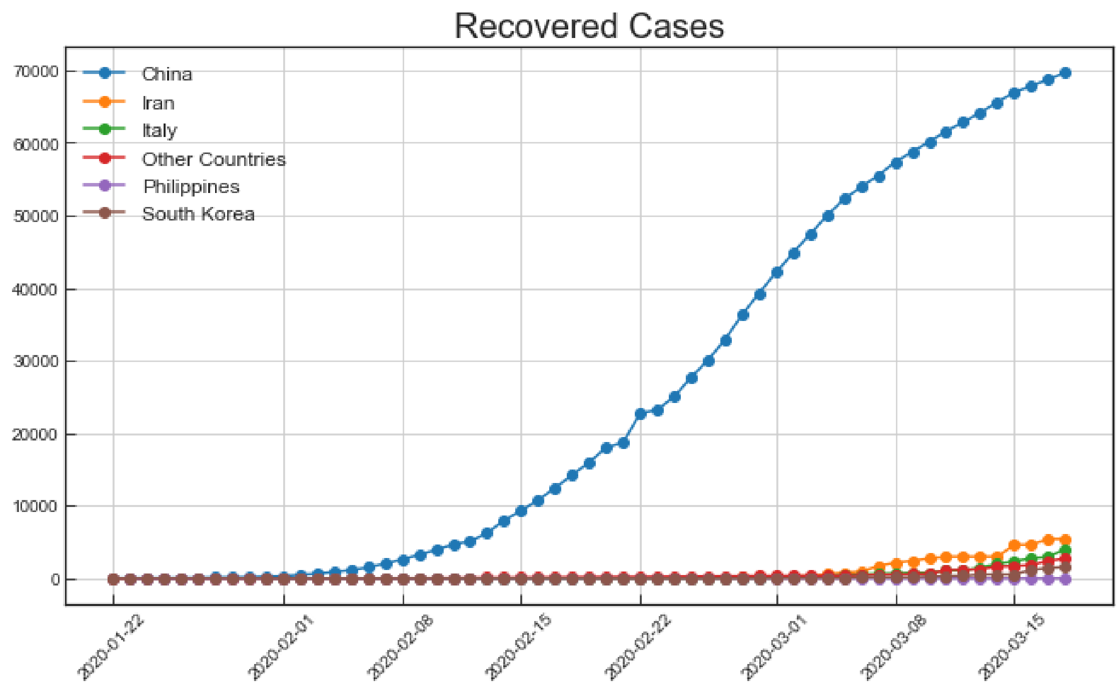
- There are much fewer new cases recently in China
- Cases outside China is growing exponentially

[17]:
```
death = pd.pivot_table(daily.dropna(subset=['Deaths']),
                       index='Date', columns='segment', values='Deaths',␣
 ↪aggfunc=np.sum).fillna(method = 'ffill')
plt.figure(figsize=(11,6))
plt.plot(death, marker='o')
plt.title('Death Cases', size = 20)
plt.legend(death.columns, loc=2, fontsize=12)
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```



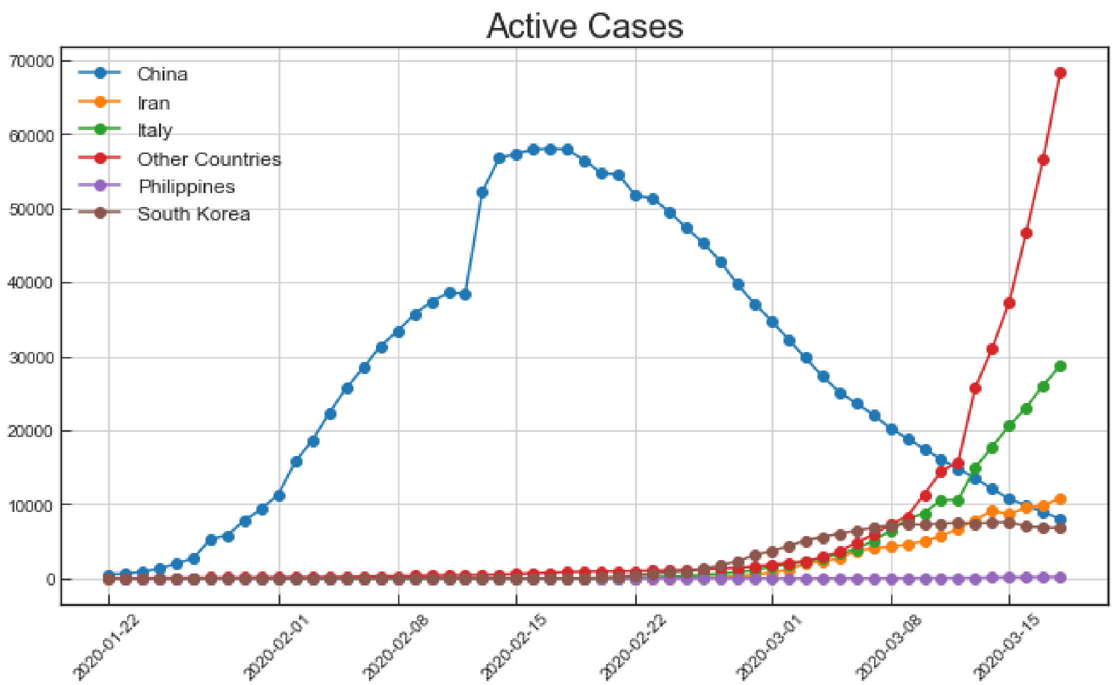
A worrying sign is world death cases showed an exponential rising trend.

```
[18]: good = pd.pivot_table(daily.dropna(subset=['Recovered']),
                               index='Date', columns='segment', values='Recovered',␣
       ↪aggfunc=np.sum).fillna(method = 'ffill')
      plt.figure(figsize=(11,6))
      plt.plot(good, marker='o')
      plt.title('Recovered Cases', size = 20)
      plt.legend(good.columns, loc=2, fontsize=12)
      plt.xticks(rotation=45)
      plt.grid(True)
      plt.show()
```



[19]:

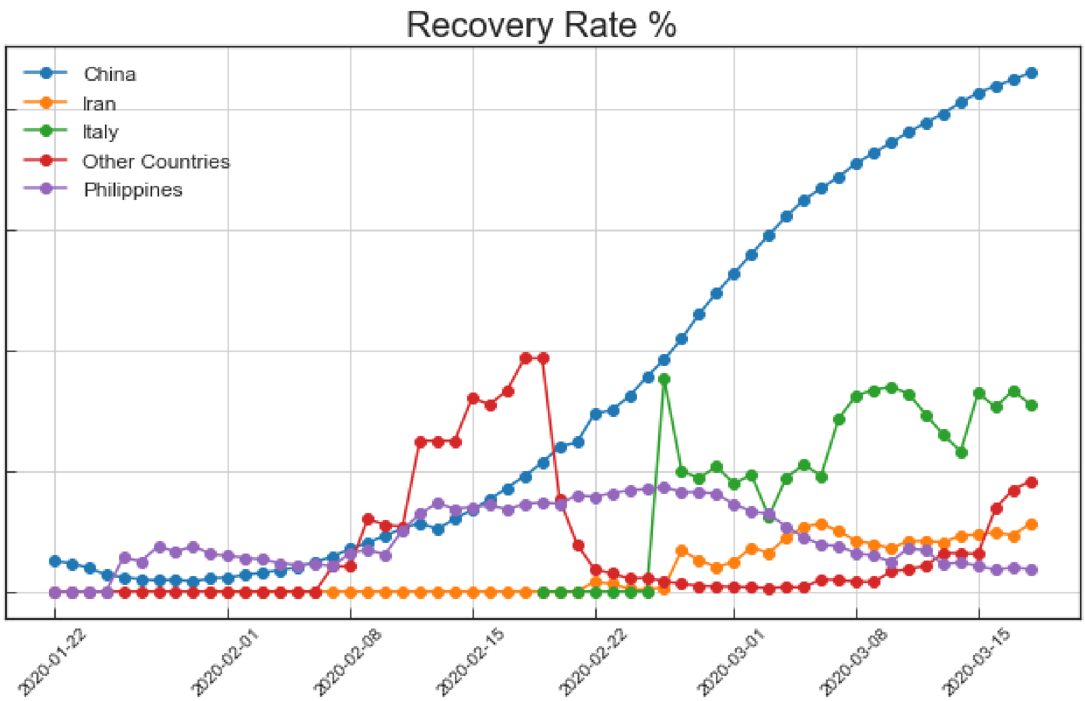
```
# Active case - confirmed minus deaths and recovered
daily['Active'] = daily['Confirmed'] - daily['Deaths'] - daily['Recovered']
active = pd.pivot_table(daily.dropna(subset=['Active']),
                              index='Date', columns='segment', values='Active',␣
 ↪aggfunc=np.sum).fillna(method = 'ffill')
plt.figure(figsize=(11,6))
plt.plot(active, marker='o')
plt.title('Active Cases', size = 20)
plt.legend(active.columns, loc=2, fontsize=12)
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```



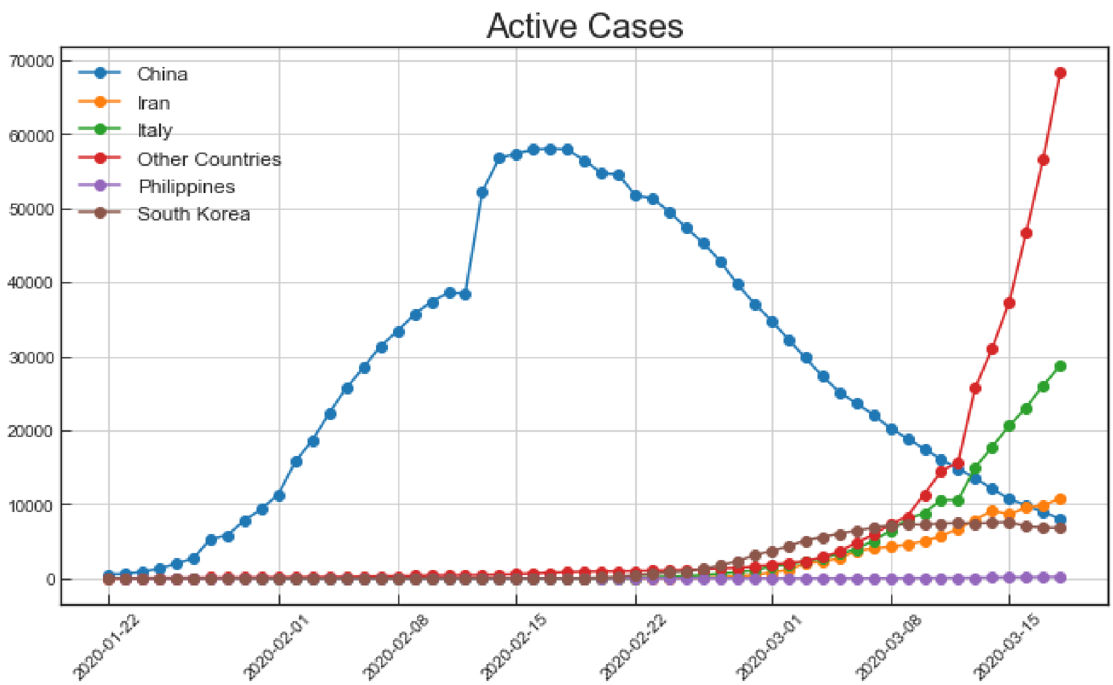
[20]:
```
df = confirm.join(death, lsuffix='_confirm', rsuffix='_death')
df = df.join(good.add_suffix('_recover'))
df['China_death_rate'] = df['China_death']/df['China_confirm']
df['Italy_death_rate'] = df['Italy_death']/df['Italy_confirm']
df['Iran_rate'] = df['Iran_death']/df['Iran_confirm']
df['South_Korea_rate'] = df['South Korea_death']/df['South Korea_confirm']
df['Other Countries_rate'] = df['Other Countries_death']/df['Other␣
↪Countries_confirm']
df['China_recover_rate'] = df['China_recover']/df['China_confirm']
df['Italy_recover_rate'] = df['Italy_recover']/df['Italy_confirm']
df['Iran_recover_rate'] = df['Iran_recover']/df['Iran_confirm']
df['South Korea_recover_rate'] = df['South Korea_recover']/df['South␣
↪Korea_confirm']
df['Other Countries_recover_rate'] = df['Other Countries_recover']/df['Other␣
↪Countries_confirm']
```

[21]:
```
recover_rate =␣
↪df[['China_recover_rate','Italy_recover_rate','Iran_recover_rate','South␣
↪Korea_recover_rate','Other Countries_recover_rate']]*100
plt.figure(figsize=(11,6))
plt.plot(recover_rate, marker='o')
plt.title('Recovery Rate %', size = 20)
plt.legend(good.columns, loc=2, fontsize=12)
plt.xticks(rotation=45)
```

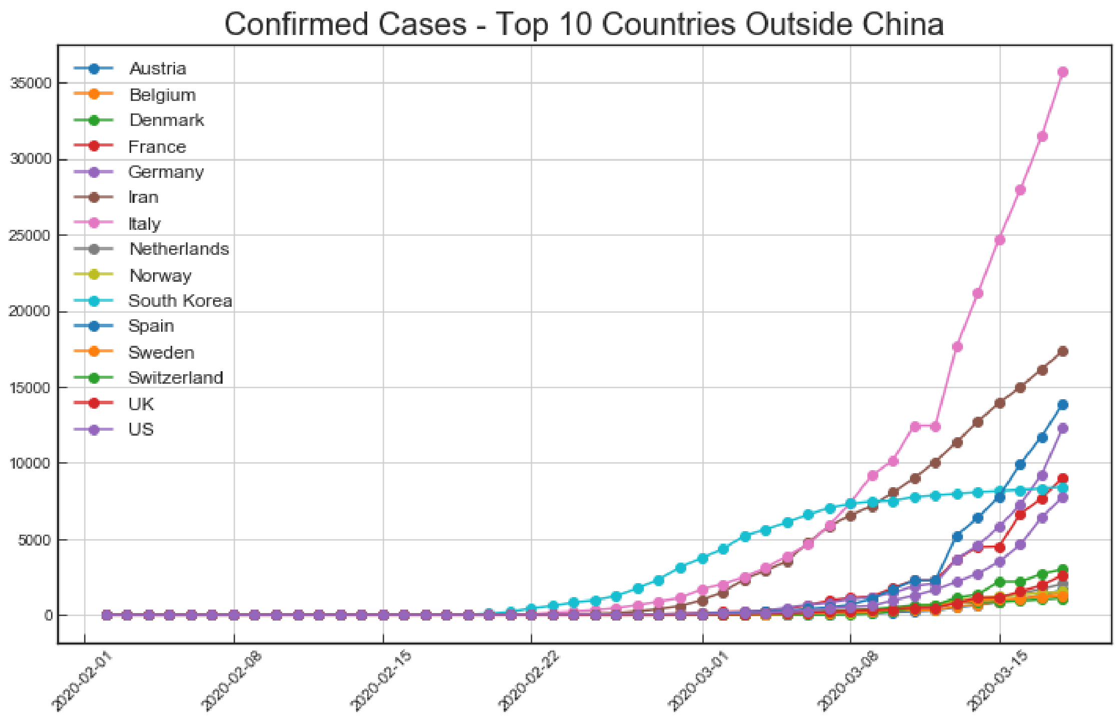
```
plt.grid(True)
plt.show()
```



[22]:
```
# Active case - confirmed minus deaths and recovered
daily['Active'] = daily['Confirmed'] - daily['Deaths'] - daily['Recovered']
active = pd.pivot_table(daily.dropna(subset=['Active']),
                        index='Date', columns='segment', values='Active',␣
 ↪aggfunc=np.sum).fillna(method = 'ffill')
plt.figure(figsize=(11,6))
plt.plot(active, marker='o')
plt.title('Active Cases', size = 20)
plt.legend(active.columns, loc=2, fontsize=12)
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```

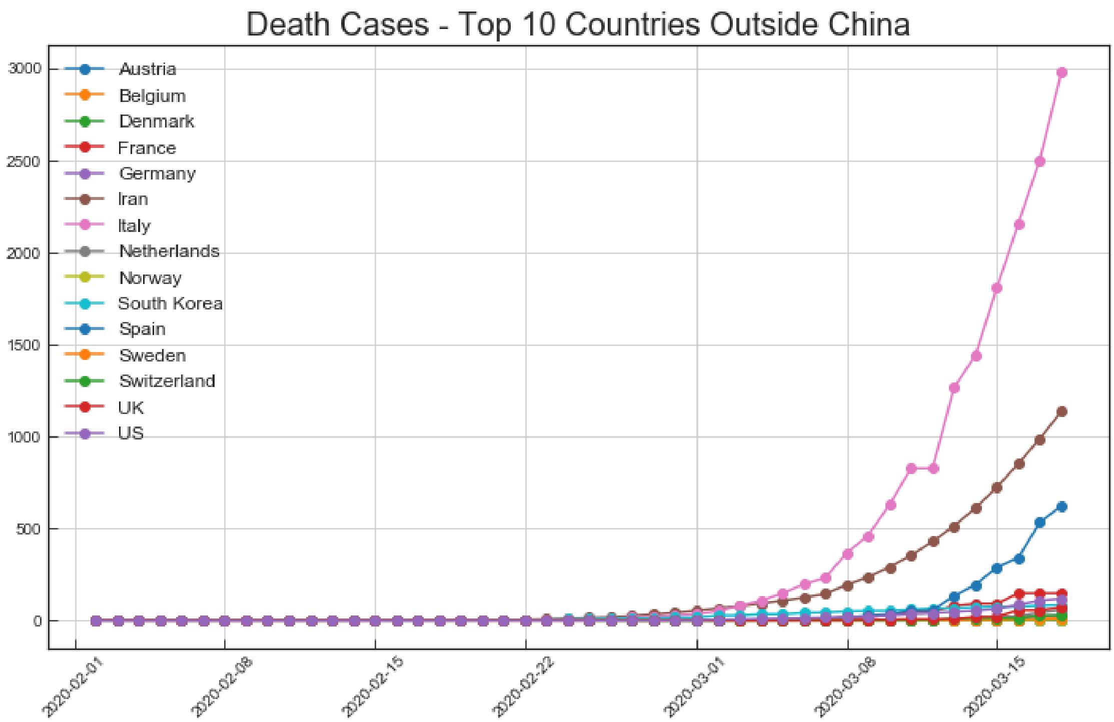


[23]:
```
# Global ex-China - Top 10 countries
# Confirmed cases
c10cases = daily[daily['Country'].isin(c10)]
confirm_w = pd.pivot_table(c10cases.dropna(subset=['Confirmed']), index='Date',
                          columns='Country', values='Confirmed', aggfunc=np.sum).
 ↪fillna(method='ffill')
plt.figure(figsize=(12,7))
plt.plot(confirm_w[confirm_w.index>'2020-02-01'], marker='o')
plt.title('Confirmed Cases - Top 10 Countries Outside China', size = 20)
plt.legend(confirm_w.columns, loc=2, fontsize=12)
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```



Italy and Iran are countries that see exponential increase in cases. South Korea cases started to slow down.

[24]:
```
death_w = pd.pivot_table(c10cases.dropna(subset=['Deaths']), index='Date',
                         columns='Country', values='Deaths', aggfunc=np.sum).
 ↪fillna(method='ffill')
plt.figure(figsize=(12,7))
plt.plot(death_w[death_w.index>'2020-02-01'], marker='o')
plt.title('Death Cases - Top 10 Countries Outside China', size = 20)
plt.legend(death_w.columns, loc=2, fontsize=12)
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```



---

[25]:
```
from datetime import datetime
time_format = "%d%b%Y %H:%H"
datetime.now().strftime(time_format)
```

[25]: '20Mar2020 21:21'

0.2.5 COVID-19 Growth Factor

Growth factor can be used to estimate the lifespan of the growth. When the growth factor for the spread is 1.0, this can be a sign that it hit the inflection point. Inflection point is where the growth starts to slow down moving forward. The growth factor just tells us the curvature of the data. If we take our data and take the 2nd derivative, basically all it is telling us is whether the cases are growing at an accelerating or decelerating rate. The inflection point is where the curve changes concavity.

This being said, we compare both the growth factor and the 2nd derivative since, maybe one or the other can be used to measure how far we are from an inflection point.

The **growth factor** on day N is the number of confirmed cases on day N minus confirmed cases on day N-1 divided by the number of confirmed cases on day N-1 minus confirmed cases on day N-2.

*Source: https://www.youtube.com/watch?v=Kas0tIxDvrg

[83]:

```
global_data = pd.read_csv("covid_19_data.csv")
```

[84]:

```
# This is a function which plots (for in input country) the active, confirmed,␣
↪and recovered cases, deaths, and the growth factor.
def plot_country_active_confirmed_recovered(country):
    # Plots Active, Confirmed, and Recovered Cases. Also plots deaths.
    country_data = global_data[global_data['Country/Region']==country]
    table = country_data.drop(['SNo','Province/State', 'Last Update'], axis=1)
    table['ActiveCases'] = table['Confirmed'] - table['Recovered'] -␣
↪table['Deaths']
    table2 = pd.pivot_table(table, values=['ActiveCases','Confirmed',␣
↪'Recovered','Deaths'], index=['ObservationDate'], aggfunc=np.sum)
    table3 = table2.drop(['Deaths'], axis=1)

    # Growth Factor

    table2['Confirmed[i]'] = table2['Confirmed']
    table2['Confirmed[i-1]'] = table2['Confirmed[i]'].shift(1, axis=0)
    table2['Confirmed[i-2]'] = table2['Confirmed[i-1]'].shift(1, axis=0)
    table2['GrowthFactor'] = (table2['Confirmed[i]'] -␣
↪table2['Confirmed[i-1]'])/(table2['Confirmed[i-1]'] -␣
↪table2['Confirmed[i-2]'])
    table2['2nd_Derivative'] = np.gradient(np.gradient(table2['Confirmed']))␣
↪#2nd derivative

      # We will map the growth factor on day i to the average of the growth␣
↪factors on days i-1, i, i+1 (this is a smoothing technique)
    table2['GrowthFactor[i]'] = table2['GrowthFactor']
    table2['GrowthFactor[i-1]'] = table2['GrowthFactor[i]'].shift(1, axis=0)
    table2['GrowthFactor[i-2]'] = table2['GrowthFactor[i-1]'].shift(1, axis=0)

    #Plot Growth Factor
    table4=table2.drop(['ActiveCases','Confirmed','Recovered','Deaths',␣
↪'Confirmed[i]','Confirmed[i-1]','Confirmed[i-2]',␣
↪'2nd_Derivative','GrowthFactor'], axis=1)
    #Smoothed GrowthFactor data
    #w=0.75
    #table4['GrowthFactor'] = (table2['GrowthFactor[i]'] +␣
↪w*table2['GrowthFactor[i-1]'] + w*table2['GrowthFactor[i-2]'])/(1+2*w)
    table4['GrowthFactor'] = table4.mean(axis=1)
    table4['GrowthFactor'] = table4['GrowthFactor'].shift(-1, axis=0)
    table4 = table4.
↪drop(['GrowthFactor[i]','GrowthFactor[i-1]','GrowthFactor[i-2]'], axis=1)

    #Plot second derivative
```

```
    table5=table2.
↪drop(['ActiveCases','Confirmed','Recovered','Deaths','GrowthFactor',␣
↪'Confirmed[i]','Confirmed[i-1]','Confirmed[i-2]',␣
↪'GrowthFactor[i]','GrowthFactor[i-1]','GrowthFactor[i-2]'], axis=1) #2nd␣
↪derivative

    #Plot confirmed[i]/confirmed[i-1]
    table6 = table2
    table6['GrowthRatio'] = table6['Confirmed[i]']/table6['Confirmed[i-1]']
    table6 = table6.
↪drop(['ActiveCases','Confirmed','Recovered','Deaths','GrowthFactor',␣
↪'Confirmed[i]','Confirmed[i-1]','Confirmed[i-2]',␣
↪'GrowthFactor[i]','GrowthFactor[i-1]','GrowthFactor[i-2]',␣
↪'2nd_Derivative'], axis=1)

    # horizontal line at growth rate 1.0 for reference
    x_coordinates = [1, 100]
    y_coordinates = [1, 1]

    return table2['Deaths'].plot(), table3.plot(), table4.plot(), plt.
↪plot(x_coordinates, y_coordinates), table5.plot(), table6.plot(), plt.
↪plot(x_coordinates, y_coordinates)
```

[85]:
```
table = global_data.drop(['SNo','Province/State', 'Last Update'], axis=1)
table['ActiveCases'] = table['Confirmed'] - table['Recovered']
table.head()
```

[85]:

	ObservationDate	Country/Region	Confirmed	Deaths	Recovered	ActiveCases
0	01/22/2020	Mainland China	1.0	0.0	0.0	1.0
1	01/22/2020	Mainland China	14.0	0.0	0.0	14.0
2	01/22/2020	Mainland China	6.0	0.0	0.0	6.0
3	01/22/2020	Mainland China	1.0	0.0	0.0	1.0
4	01/22/2020	Mainland China	0.0	0.0	0.0	0.0

[86]:
```
table2 = pd.pivot_table(table, values=['ActiveCases','Confirmed',␣
↪'Recovered','Deaths'], index=['ObservationDate'], aggfunc=np.sum)
table2.head()
```

[86]:

ObservationDate	ActiveCases	Confirmed	Deaths	Recovered
01/22/2020	527.0	555.0	17.0	28.0
01/23/2020	623.0	653.0	18.0	30.0
01/24/2020	905.0	941.0	26.0	36.0
01/25/2020	1399.0	1438.0	42.0	39.0
01/26/2020	2066.0	2118.0	56.0	52.0

[87]:
```
# Growth Factor

table2['Confirmed[i]'] = table2['Confirmed']
table2['Confirmed[i-1]'] = table2['Confirmed[i]'].shift(1, axis=0)
table2['Confirmed[i-2]'] = table2['Confirmed[i-1]'].shift(1, axis=0)
table2['GrowthFactor'] = (table2['Confirmed[i]'] - table2['Confirmed[i-1]'])/
↪(table2['Confirmed[i-1]'] - table2['Confirmed[i-2]'])
# We will map the growth factor on day i to the average of the growth factors␣
↪on days i-1, i, i+1 (this is a smoothing technique)
table2['GrowthFactor[i]'] = table2['GrowthFactor']
table2['GrowthFactor[i-1]'] = table2['GrowthFactor[i]'].shift(1, axis=0)
table2['GrowthFactor[i-2]'] = table2['GrowthFactor[i-1]'].shift(1, axis=0)

table3=table2.drop(['ActiveCases','Confirmed','Recovered','Deaths',␣
↪'Confirmed[i]','Confirmed[i-1]','Confirmed[i-2]', 'GrowthFactor'], axis=1)

#Smoothed GrowthFactor data
w=0.8
table3['GrowthFactor'] = (table2['GrowthFactor[i]'] +␣
↪w*table2['GrowthFactor[i-1]'] + w*table2['GrowthFactor[i-2]'])/(1+2*w)
#table3['GrowthFactor'] = table3.mean(axis=1)
table3['GrowthFactor'] = table3['GrowthFactor'].shift(-1, axis=0)
table3.head()
```

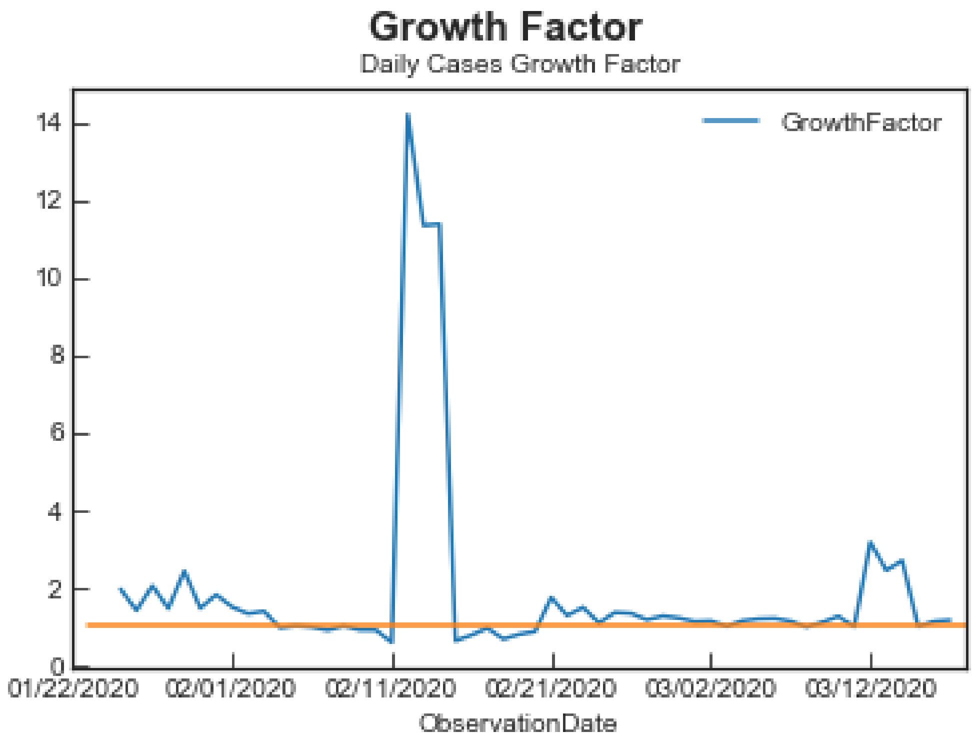
[87]:
```
                 GrowthFactor[i]  GrowthFactor[i-1]  GrowthFactor[i-2]  \
ObservationDate
01/22/2020                   NaN                NaN                NaN
01/23/2020                   NaN                NaN                NaN
01/24/2020              2.938776                NaN                NaN
01/25/2020              1.725694           2.938776                NaN
01/26/2020              1.368209           1.725694           2.938776

                 GrowthFactor
ObservationDate
01/22/2020                NaN
01/23/2020                NaN
01/24/2020                NaN
01/25/2020           1.961456
01/26/2020           1.409550
```

[88]:
```
table3 = table3.
↪drop(['GrowthFactor[i]','GrowthFactor[i-1]','GrowthFactor[i-2]'], axis=1)
```

[94]:
```
table3.plot()
plt.suptitle('Growth Factor', size = 15, weight = "bold")
plt.title('Daily Cases Growth Factor', size = 10)
# horizontal line at growth rate 1.0 for reference
```

```
x_coordinates = [1, 100]
y_coordinates = [1, 1]
plt.plot(x_coordinates, y_coordinates)
```

[94]: [<matplotlib.lines.Line2D at 0x11f430ef0>]



0.3 Part 2: Simulating COVID-19 Spread and its Basic Reproduction Number

0.3.1 Prediction with SIR-D Model

SIR-D model is a simple mathematical model to understand outbreak of infectious diseases. In this work, we will focus on the members of the population that are either susceptible, infected, recovered, or dead.

Because we can measure the number of fatal cases and recovered cases separately, we can use two variables ("Recovered" and "Deaths") instead of "Recovered + Deaths" in the mathematical model.

0.3.2 What is SIR-D model?

- S: Susceptible – not immune and capable of contracting the pathogen
- I: Infected – currently infected with the pathogen
- R: Recovered – immune from the pathogen (via vaccine or post exposure)

- D: Fatal – deaths from the pathogen

Model:
$S + I \xrightarrow{\beta} 2I$
$I \xrightarrow{\gamma} R$
$I \xrightarrow{\alpha} D$

α: Mortality rate [1/min]
β: Effective contact rate [1/min]
γ: Recovery rate [1/min]

Ordinary Differential Equation (ODE):
$\frac{\mathrm{d}S}{\mathrm{d}T} = -N^{-1}\beta SI$
$\frac{\mathrm{d}I}{\mathrm{d}T} = N^{-1}\beta SI - (\gamma + \alpha)I$
$\frac{\mathrm{d}R}{\mathrm{d}T} = \gamma I$
$\frac{\mathrm{d}D}{\mathrm{d}T} = \alpha I$

Where $N = S + I + R + D$ is the total population, T is the elapsed time from the start date.

The SIR model is governed by the differential equations of the infection rate of the pathogen and the recovery rate. Together these two values give the basic reproduction number R0: the average number of secondary infections caused by an infected host.

Basic reproduction number, Non-dimentional parameter, is defined as
$R_0 = \rho\sigma^{-1} = \beta\gamma^{-1}$

Estimated Mean Values of R_0:
R_0 means "the average number of secondary infections caused by an infected host" (Infection Modeling — Part 1).
(Secondary data: Van den Driessche, P., & Watmough, J. (2002).)
2.06: Zika in South America, 2015-2016
1.51: Ebola in Guinea, 2014
1.33: H1N1 influenza in South Africa, 2009
3.5 : SARS in 2002-2003
1.68: H2N2 influenza in US, 1957
3.8 : Fall wave of 1918 Spanish influenza in Genova
1.5 : Spring wave of 1918 Spanish influenza in Genova
***

0.3.3 Setting up the parameters, hyperparameters, and functions for the model

0.3.4 Total population

Total population value in 2020 was retrieved from PopulationPyramid.net WORLD 2020. This is by PopulationPyramid.net and licenced under Creative Commons license CC BY 3.0.

Global
China
Japan
South Korea (Republic of Korea)

Italy
Iran (Islamic Republic of)

Philippines

[26]:
```
population_date = "14Mar2020"
_dict = {
    "Global": "7,794,798,729",
    "China": "1,439,323,774",
    "Japan": "126,476,458",
    "South Korea": "51,269,182",
    "Italy": "60,461,827",
    "Iran": "83,992,953",
    "Philippines":"109,581,084"
}
population_dict = {k: int(v.replace(",", "")) for (k, v) in _dict.items()}
df = pd.io.json.json_normalize(population_dict)
df.index = [f"Total population on {population_date}"]
df
```

[26]:
```
                                 Global       China      Japan  South Korea  \
Total population on 14Mar2020  7794798729  1439323774  126476458     51269182

                                  Italy      Iran  Philippines
Total population on 14Mar2020  60461827  83992953    109581084
```

0.3.5 Dataset

[27]:
```
import os
for dirname, _, filenames in os.walk("R Notebook"):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

0.3.6 Functions

We define the functions to use in this notebook.

[28]:
```
def line_plot(df, title, ylabel="Cases", h=None, v=None, xlim=(None, None),␣
 ↪ylim=(0, None), math_scale=True):
    """
    Show chlonological change of the data.
    """
    ax = df.plot()
    if math_scale:
        ax.yaxis.set_major_formatter(ScalarFormatter(useMathText=True))
        ax.ticklabel_format(style="sci",  axis="y",scilimits=(0, 0))
```

```
    ax.set_title(title)
    ax.set_xlabel(None)
    ax.set_ylabel(ylabel)
    ax.set_xlim(*xlim)
    ax.set_ylim(*ylim)
    ax.legend(bbox_to_anchor=(1.02, 0), loc="lower left", borderaxespad=0)
    if h is not None:
        ax.axhline(y=h, color="black", linestyle="--")
    if v is not None:
        ax.axvline(x=v, color="black", linestyle="--")
    plt.tight_layout()
    plt.show()
```

0.3.7 Trend analysis

[29]:
```
def select_area(ncov_df, places=None, excluded_places=None):
    """
    Select the records of the palces.
    @ncov_df <pd.DataFrame>: the clean data
    @places <list[tuple(<str/None>, <str/None>)]: the list of places
        - if the list is None, all data will be used
        - (str, str): both of country and province are specified
        - (str, None): only country is specified
        - (None, str) or (None, None): Error
    @excluded_places <list[tuple(<str/None>, <str/None>)]: the list of excluded␣
 ↪places
        - if the list is None, all data in the "places" will be used
        - (str, str): both of country and province are specified
        - (str, None): only country is specified
        - (None, str) or (None, None): Error
    @return <pd.DataFrame>: index and columns are as same as @ncov_df
    """
    # Select the target records
    df = ncov_df.copy()
    c_series = ncov_df["Country"]
    p_series = ncov_df["Province"]
    if places is not None:
        df = pd.DataFrame(columns=ncov_df.columns)
        for (c, p) in places:
            if c is None:
                raise Exception("places: Country must be specified!")
            if p is None:
                new_df = ncov_df.loc[c_series == c, :]
            else:
                new_df = ncov_df.loc[(c_series == c) & (p_series == p), :]
            df = pd.concat([df, new_df], axis=0)
```

```
    if excluded_places is not None:
        for (c, p) in excluded_places:
            if c is None:
                raise Exception("excluded_places: Country must be specified!")
            if p is None:
                df = df.loc[c_series != c, :]
            else:
                c_df = df.loc[(c_series == c) & (p_series != p), :]
                other_df = df.loc[c_series != c, :]
                df = pd.concat([c_df, other_df], axis=0)
    df = df.groupby("Date").sum().reset_index()
    return df
```

[64]:

```
def show_trend(ncov_df, variable="Confirmed", n_changepoints=2, places=None,␣
 ↪excluded_places=None):
    """
    Show trend of log10(@variable) using fbprophet package.
    @ncov_df <pd.DataFrame>: the clean data
    @variable <str>: variable name to analyse
        - if Confirmed, use Infected + Recovered + Deaths
    @n_changepoints <int>: max number of change points
    @places <list[tuple(<str/None>, <str/None>)]: the list of places
        - if the list is None, all data will be used
        - (str, str): both of country and province are specified
        - (str, None): only country is specified
        - (None, str) or (None, None): Error
    @excluded_places <list[tuple(<str/None>, <str/None>)]: the list of excluded␣
 ↪places
        - if the list is None, all data in the "places" will be used
        - (str, str): both of country and province are specified
        - (str, None): only country is specified
        - (None, str) or (None, None): Error
    """
    # Data arrangement
    df = select_area(ncov_df, places=places, excluded_places=excluded_places)
    if variable == "Confirmed":
        df["Confirmed"] = df[["Infected", "Recovered", "Deaths"]].sum(axis=1)
    df = df.loc[:, ["Date", variable]]
    df.columns = ["ds", "y"]
    # Log10(x)
    warnings.resetwarnings()
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        df["y"] = np.log10(df["y"]).replace([np.inf, -np.inf], 0)
    # fbprophet
    model = Prophet(growth="linear", daily_seasonality=False,␣
 ↪n_changepoints=n_changepoints)
```

```
    model.fit(df)
    future = model.make_future_dataframe(periods=0)
    forecast = model.predict(future)
    # Create figure
    fig = model.plot(forecast)
    _ = add_changepoints_to_plot(fig.gca(), model, forecast)
    plt.title(f"log10({variable}) over time and change points")
    plt.ylabel(f"log10(the number of cases)")
    plt.xlabel("")
```

0.3.8 Dataset arrangement

[31]:
```
def create_target_df(ncov_df, total_population, places=None,
                     excluded_places=None, start_date=None,␣
 ↪date_format="%d%b%Y"):
    """
    Select the records of the palces, calculate the number of susceptible␣
 ↪people,
     and calculate the elapsed time [day] from the start date of the target␣
 ↪dataframe.
    @ncov_df <pd.DataFrame>: the clean data
    @total_population <int>: total population in the places
    @places <list[tuple(<str/None>, <str/None>)]: the list of places
        - if the list is None, all data will be used
        - (str, str): both of country and province are specified
        - (str, None): only country is specified
        - (None, str) or (None, None): Error
    @excluded_places <list[tuple(<str/None>, <str/None>)]: the list of excluded␣
 ↪places
        - if the list is None, all data in the "places" will be used
        - (str, str): both of country and province are specified
        - (str, None): only country is specified
        - (None, str) or (None, None): Error
    @start_date <str>: the start date or None
    @date_format <str>: format of @start_date
    @return <tuple(2 objects)>:
        - 1. start_date <pd.Timestamp>: the start date of the selected records
        - 2. target_df <pd.DataFrame>:
            - column T: elapsed time [min] from the start date of the dataset
            - column Susceptible: the number of patients who are in the palces␣
 ↪but not infected/recovered/died
            - column Infected: the number of infected cases
            - column Recovered: the number of recovered cases
            - column Deaths: the number of death cases
    """
```

```
    # Select the target records
    df = select_area(ncov_df, places=places, excluded_places=excluded_places)
    if start_date is not None:
        df = df.loc[df["Date"] > datetime.strptime(start_date, date_format), :]
    start_date = df.loc[df.index[0], "Date"]
    # column T
    df["T"] = ((df["Date"] - start_date).dt.total_seconds() / 60).astype(int)
    # coluns except T
    df["Susceptible"] = total_population - df["Infected"] - df["Recovered"] -␣
↪df["Deaths"]
    response_variables = ["Susceptible", "Infected", "Recovered", "Deaths"]
    # Return
    target_df = df.loc[:, ["T", *response_variables]]
    return (start_date, target_df)
```

0.3.9 Numerical simulation

We will perform numerical analysis to solve the ODE using scipy.integrate.solve_ivp function.

[32]:
```
def simulation(model, initials, step_n, **params):
    """
    Solve ODE of the model.
    @model <ModelBase>: the model
    @initials <tuple[float]>: the initial values
    @step_n <int>: the number of steps
    """
    tstart, dt, tend = 0, 1, step_n
    sol = solve_ivp(
        fun=model(**params),
        # Implicit Runge-Kutta method of the Radau IIA family of order 5
        # method="Radau",
        t_span=[tstart, tend],
        y0=np.array(initials, dtype=np.float64),
        t_eval=np.arange(tstart, tend + dt, dt),
        dense_output=True
    )
    t_df = pd.Series(data=sol["t"], name="t")
    y_df = pd.DataFrame(data=sol["y"].T.copy(), columns=model.VARIABLES)
    sim_df = pd.concat([t_df, y_df], axis=1)
    return sim_df
```

0.3.10 Parameter Estimation using Optuna

[33]:
```
class Estimater(object):
    def __init__(self, model, ncov_df, total_population, name=None, places=None,
                 excluded_places=None, start_date=None, date_format="%d%b%Y"):
        """
        Set training data.
        @model <ModelBase>: the model
        @name <str>: name of the area
        @the other params: See the function named create_target_df()
        """
        dataset = model.create_dataset(
            ncov_df, total_population, places=places,␣
↪excluded_places=excluded_places,
            start_date=start_date, date_format=date_format
        )
        self.start_time, self.initials, self.Tend, self.train_df = dataset
        self.total_population = total_population
        self.name = name
        self.model = model
        self.param_dict = dict()
        self.study = None
        self.optimize_df = None

    def run(self, n_trials=500):
        """
        Try estimation (optimization of parameters and tau).
        @n_trials <int>: the number of trials
        """
        if self.study is None:
            self.study = optuna.create_study(direction="minimize")
        self.study.optimize(
            lambda x: self.objective(x),
            n_trials=n_trials,
            n_jobs=-1
        )
        param_dict = self.study.best_params.copy()
        param_dict["R0"] = self.calc_r0()
        param_dict["score"] = self.score()
        param_dict.update(self.calc_days_dict())
        self.param_dict = param_dict.copy()
        return param_dict

    def history_df(self):
        """
        Return the hsitory of optimization.
        @return <pd.DataFrame>
```

```
        """
        optimize_df = self.study.trials_dataframe()
        optimize_df["time[s]"] = optimize_df["datetime_complete"] -␣
↪optimize_df["datetime_start"]
        optimize_df["time[s]"] = optimize_df["time[s]"].dt.total_seconds()
        self.optimize_df = optimize_df.drop(["datetime_complete",␣
↪"datetime_start"], axis=1)
        return self.optimize_df.sort_values("value", ascending=True)

    def history_graph(self):
        """
        Show the history of parameter search using pair-plot.
        """
        if self.optimize_df is None:
            self.history_df()
        df = self.optimize_df.copy()
        sns.pairplot(df.loc[:, df.columns.str.startswith("params_")],␣
↪diag_kind="kde", markers="+")
        plt.show()

    def objective(self, trial):
        # Time
        tau = trial.suggest_int("tau", 1, 1440)
        # Apply adjusted Exponential Moving Average on the training data
        #.set_index("T").ewm(span=7, adjust=True).mean().reset_index()
        train_df_divided = self.train_df.copy()
        train_df_divided["t"] = (train_df_divided["T"] / tau).astype(int)
        # Parameters
        p_dict = dict()
        for (name, info) in self.model.param_dict(train_df_divided).items():
            if info[0] == "float":
                param = trial.suggest_uniform(name, info[1], info[2])
            else:
                param = trial.suggest_int(name, info[1], info[2])
            p_dict[name] = param
        # Simulation
        t_end = train_df_divided.loc[train_df_divided.index[-1], "t"]
        sim_df = simulation(self.model, self.initials, step_n=t_end, **p_dict)
        return self.error_f(train_df_divided, sim_df)

    def error_f(self, train_df_divided, sim_df):
        """
        We need to minimize the difference of the observed values and estimated␣
↪values.
        This function calculate the difference of the estimated value and␣
↪obeserved value.
        """
```

```
        df = pd.merge(train_df_divided, sim_df, on="t", suffixes=("_observed",␣
↪"_estimated"))
        diffs = [
            # Weighted Average: the recent data is more important
            p * np.average(
                abs(df[f"{v}_observed"] - df[f"{v}_estimated"]) /␣
↪(df[f"{v}_observed"] * self.total_population + 1),
                weights=df["t"]
            )
            for (p, v) in zip(self.model.PRIORITIES, self.model.VARIABLES)
        ]
        return sum(diffs) * (self.total_population ** 2)

    def compare_df(self):
        """
        Show the taining data and simulated data in one dataframe.

        """
        est_dict = self.study.best_params.copy()
        tau = est_dict["tau"]
        est_dict.pop("tau")
        observed_df = self.train_df.drop("T", axis=1)
        observed_df["t"] = (self.train_df["T"] / tau).astype(int)
        t_end = observed_df.loc[observed_df.index[-1], "t"]
        sim_df = simulation(self.model, self.initials, step_n=t_end, **est_dict)
        df = pd.merge(observed_df, sim_df, on="t", suffixes=("_observed",␣
↪"_estimated"))
        df = df.set_index("t")
        return df

    def compare_graph(self):
        """
        Compare obsereved and estimated values in graphs.
        """
        df = self.compare_df()
        val_len = len(self.model.VARIABLES)
        fig, axes = plt.subplots(ncols=1, nrows=val_len, figsize=(9, 6 *␣
↪val_len / 2))
        for (ax, v) in zip(axes.ravel()[1:], self.model.VARIABLES[1:]):
            df[[f"{v}_observed", f"{v}_estimated"]].plot.line(
                ax=ax, ylim=(0, None), sharex=True,
                title=f"{self.model.NAME}: Comparison of observed/estimated␣
↪{v}(t)"
            )
            ax.yaxis.set_major_formatter(ScalarFormatter(useMathText=True))
            ax.ticklabel_format(style="sci",  axis="y",scilimits=(0, 0))
```

```
            ax.legend(bbox_to_anchor=(1.02, 0), loc="lower left",␣
↪borderaxespad=0)
        for v in self.model.VARIABLES[1:]:
            df[f"{v}_diff"] = df[f"{v}_observed"] - df[f"{v}_estimated"]
            df[f"{v}_diff"].plot.line(
                ax=axes.ravel()[0], sharex=True,
                title=f"{self.model.NAME}: observed - estimated"
            )
        axes.ravel()[0].axhline(y=0, color="black", linestyle="--")
        axes.ravel()[0].yaxis.
↪set_major_formatter(ScalarFormatter(useMathText=True))
        axes.ravel()[0].ticklabel_format(style="sci",  axis="y",scilimits=(0,␣
↪0))
        axes.ravel()[0].legend(bbox_to_anchor=(1.02, 0), loc="lower left",␣
↪borderaxespad=0)
        fig.tight_layout()
        fig.show()

    def calc_r0(self):
        """
        Calculate R0.
        """
        est_dict = self.study.best_params.copy()
        est_dict.pop("tau")
        model_instance = self.model(**est_dict)
        return model_instance.calc_r0()

    def calc_days_dict(self):
        """
        Calculate 1/beta etc.
        """
        est_dict = self.study.best_params.copy()
        tau = est_dict["tau"]
        est_dict.pop("tau")
        model_instance = self.model(**est_dict)
        return model_instance.calc_days_dict(tau)

    def predict_df(self, step_n):
        """
        Predict the values in the future.
        @step_n <int>: the number of steps
        @return <pd.DataFrame>: predicted data for measurable variables.
        """
        est_dict = self.study.best_params.copy()
        tau = est_dict["tau"]
        est_dict.pop("tau")
        df = simulation(self.model, self.initials, step_n=step_n, **est_dict)
```

```
        df["Time"] = (df["t"] * tau).apply(lambda x: timedelta(minutes=x)) +␣
↪self.start_time
        df = df.set_index("Time").drop("t", axis=1)
        df = (df * self.total_population).astype(int)
        upper_cols = [n.upper() for n in df.columns]
        df.columns = upper_cols
        df = self.model.calc_variables_reverse(df).drop(upper_cols, axis=1)
        return df

    def predict_graph(self, step_n, name=None, excluded_cols=None):
        """
        Predict the values in the future and create a figure.
        @step_n <int>: the number of steps
        @name <str>: name of the area
        @excluded_cols <list[str]>: the excluded columns in the figure
        """
        if self.name is not None:
            name = self.name
        else:
            name = str() if name is None else name
        df = self.predict_df(step_n=step_n)
        if excluded_cols is not None:
            df = df.drop(excluded_cols, axis=1)
        r0 = self.param_dict["R0"]
        title = f"Prediction in {name} with {self.model.NAME} model: R0 = {r0}"
        line_plot(df, title, v= datetime.today(), h=self.total_population)

    def score(self):
        """
        Return the sum of differences of observed and estimated values devided␣
↪by the number of steps.
        """
        variables = self.model.VARIABLES[:]
        compare_df = self.compare_df()
        score = 0
        for v in variables:
            score += abs(compare_df[f"{v}_observed"] -␣
↪compare_df[f"{v}_estimated"]).sum()
        score = score / len(compare_df)
        return score

    def info(self):
        """
        Return Estimater information.
        @return <tupple[object]>:
            - <ModelBase>: model
            - <dict[str]=str>: name, total_population, start_time, tau
```

```
            - <dict[str]=float>: values of parameters of model
        """
        param_dict = self.study.best_params.copy()
        info_dict = {
            "name": self.name,
            "total_population": self.total_population,
            "start_time": self.start_time,
            "tau": param_dict["tau"],
            "initials": self.initials
        }
        param_dict.pop("tau")
        return (self.model, info_dict, param_dict)
```

0.3.11 Description of Math Model

[34]:

```
class ModelBase(object):
    NAME = "Model"
    VARIABLES = ["x"]
    PRIORITIES = np.array([1])
    QUANTILE_RANGE = [0.3, 0.7]

    @classmethod
    def param_dict(cls, train_df_divided=None, q_range=None):
        """
        Define parameters without tau. This function should be overwritten.
        @train_df_divided <pd.DataFrame>:
            - column: t and non-dimensional variables
        @q_range <list[float, float]>: quantile rage of the parameters␣
↪calculated by the data
        @return <dict[name]=(type, min, max):
            @type <str>: "float" or "int"
            @min <float/int>: min value
            @max <float/int>: max value
        """
        param_dict = dict()
        return param_dict

    @staticmethod
    def calc_variables(df):
        """
        Calculate the variables of the model.
        This function should be overwritten.
        @df <pd.DataFrame>
        @return <pd.DataFrame>
        """
        return df
```

```
    @staticmethod
    def calc_variables_reverse(df):
        """
        Calculate measurable variables using the variables of the model.
        This function should be overwritten.
        @df <pd.DataFrame>
        @return <pd.DataFrame>
        """
        return df

    @classmethod
    def create_dataset(cls, ncov_df, total_population, places=None,
                       excluded_places=None, start_date=None,␣
↪date_format="%d%b%Y"):
        """
        Create dataset with the model-specific varibles.
        The variables will be divided by total population.
        The column names (not include T) will be lower letters.
        @params: See the function named create_target_df()
        @return <tuple(objects)>:
            - start_date <pd.Timestamp>
            - initials <tuple(float)>: the initial values
            - Tend <int>: the last value of T
            - df <pd.DataFrame>: the dataset
        """
        start_date, target_df = create_target_df(
            ncov_df, total_population, places=places,␣
↪excluded_places=excluded_places,
            start_date=start_date, date_format=date_format
        )
        df = cls.calc_variables(target_df).set_index("T") / total_population
        df.columns = [n.lower() for n in df.columns]
        initials = df.iloc[0, :].values
        df = df.reset_index()
        Tend = df.iloc[-1, 0]
        return (start_date, initials, Tend, df)

    def calc_r0(self):
        """
        Calculate R0. This function should be overwritten.
        """
        return None

    def calc_days_dict(self, tau):
        """
        Calculate 1/beta [day] etc.
```

```
        This function should be overwritten.
        @param tau <int>: tau value [hour]
        """
        return dict()
```

0.3.12 Creating a function for SIR Model before the SIR-D

[35]:
```
class SIR(ModelBase):
    NAME = "SIR"
    VARIABLES = ["x", "y", "z"]
    PRIORITIES = np.array([1, 1, 1])

    def __init__(self, rho, sigma):
        super().__init__()
        self.rho = float(rho)
        self.sigma = float(sigma)

    def __call__(self, t, X):
        # x, y, z = [X[i] for i in range(len(self.VARIABLES))]
        # dxdt = - self.rho * x * y
        # dydt = self.rho * x * y - self.sigma * y
        # dzdt = self.sigma * y
        dxdt = - self.rho * X[0] * X[1]
        dydt = self.rho * X[0] * X[1] - self.sigma * X[1]
        dzdt = self.sigma * X[1]
        return np.array([dxdt, dydt, dzdt])

    @classmethod
    def param_dict(cls, train_df_divided=None, q_range=None):
        param_dict = super().param_dict()
        q_range = super().QUANTILE_RANGE[:] if q_range is None else q_range
        if train_df_divided is None:
            param_dict["rho"] = ("float", 0, 1)
            param_dict["sigma"] = ("float", 0, 1)
        else:
            df = train_df_divided.copy()
            # rho = - (dx/dt) / x / y
            rho_series = 0 - df["x"].diff() / df["t"].diff() / df["x"] / df["y"]
            param_dict["rho"] = ("float", *rho_series.quantile(q_range))
            # sigma = (dz/dt) / y
            sigma_series = df["z"].diff() / df["t"].diff() / df["y"]
            param_dict["sigma"] = ("float", *sigma_series.quantile(q_range))
        return param_dict

    @staticmethod
    def calc_variables(df):
```

```
        df["X"] = df["Susceptible"]
        df["Y"] = df["Infected"]
        df["Z"] = df["Recovered"] + df["Deaths"]
        return df.loc[:, ["T", "X", "Y", "Z"]]

    @staticmethod
    def calc_variables_reverse(df):
        df["Susceptible"] = df["X"]
        df["Infected"] = df["Y"]
        df["Recovered/Deaths"] = df["Z"]
        return df

    def calc_r0(self):
        if self.sigma == 0:
            return np.nan
        r0 = self.rho / self.sigma
        return round(r0, 2)

    def calc_days_dict(self, tau):
        _dict = dict()
        _dict["1/beta [day]"] = int(tau / 24 / 60 / self.rho)
        _dict["1/gamma [day]"] = int(tau / 24 / 60 / self.sigma)
        return _dict
```

0.3.13 Creating the function for SIR-D Model

[36]:
```
class SIRD(ModelBase):
    NAME = "SIR-D"
    VARIABLES = ["x", "y", "z", "w"]
    PRIORITIES = np.array([1, 10, 10, 2])

    def __init__(self, kappa, rho, sigma):
        super().__init__()
        self.kappa = float(kappa)
        self.rho = float(rho)
        self.sigma = float(sigma)

    def __call__(self, t, X):
        # x, y, z, w = [X[i] for i in range(len(self.VARIABLES))]
        # dxdt = - self.rho * x * y
        # dydt = self.rho * x * y - (self.sigma + self.kappa) * y
        # dzdt = self.sigma * y
        # dwdt = self.kappa * y
        dxdt = - self.rho * X[0] * X[1]
        dydt = self.rho * X[0] * X[1] - (self.sigma + self.kappa) * X[1]
        dzdt = self.sigma * X[1]
```

```
        dwdt = self.kappa * X[1]
        return np.array([dxdt, dydt, dzdt, dwdt])

    @classmethod
    def param_dict(cls, train_df_divided=None, q_range=None):
        param_dict = super().param_dict()
        q_range = super().QUANTILE_RANGE[:] if q_range is None else q_range
        if train_df_divided is None:
            param_dict["kappa"] = ("float", 0, 1)
            param_dict["rho"] = ("float", 0, 1)
            param_dict["sigma"] = ("float", 0, 1)
        else:
            df = train_df_divided.copy()
            # kappa = (dw/dt) / y
            kappa_series = df["w"].diff() / df["t"].diff() / df["y"]
            param_dict["kappa"] = ("float", *kappa_series.quantile(q_range))
            # rho = - (dx/dt) / x / y
            rho_series = 0 - df["x"].diff() / df["t"].diff() / df["x"] / df["y"]
            param_dict["rho"] = ("float", *rho_series.quantile(q_range))
            # sigma = (dz/dt) / y
            sigma_series = df["z"].diff() / df["t"].diff() / df["y"]
            param_dict["sigma"] = ("float", *sigma_series.quantile(q_range))
        return param_dict

    @staticmethod
    def calc_variables(df):
        df["X"] = df["Susceptible"]
        df["Y"] = df["Infected"]
        df["Z"] = df["Recovered"]
        df["W"] = df["Deaths"]
        return df.loc[:, ["T", "X", "Y", "Z", "W"]]

    @staticmethod
    def calc_variables_reverse(df):
        df["Susceptible"] = df["X"]
        df["Infected"] = df["Y"]
        df["Recovered"] = df["Z"]
        df["Deaths"] = df["W"]
        return df

    def calc_r0(self):
        try:
            r0 = self.rho / (self.sigma + self.kappa)
        except ZeroDivisionError:
            return np.nan
        return round(r0, 2)
```

```
    def calc_days_dict(self, tau):
        _dict = dict()
        if self.kappa == 0:
            _dict["1/alpha2 [day]"] = 0
        else:
            _dict["1/alpha2 [day]"] = int(tau / 24 / 60 / self.kappa)
        _dict["1/beta [day]"] = int(tau / 24 / 60 / self.rho)
        if self.sigma == 0:
            _dict["1/gamma [day]"] = 0
        else:
            _dict["1/gamma [day]"] = int(tau / 24 / 60 / self.sigma)
        return _dict
```

0.3.14 Reading and inspecting the data

[37]:
```
raw = pd.read_csv("covid_19_data.csv")
raw.tail()
```

[37]:
```
       SNo ObservationDate Province/State         Country/Region  \
6717  6718      03/18/2020            NaN               Guernsey
6718  6719      03/18/2020            NaN                 Jersey
6719  6720      03/18/2020            NaN            Puerto Rico
6720  6721      03/18/2020            NaN  Republic of the Congo
6721  6722      03/18/2020            NaN             The Gambia

              Last Update  Confirmed  Deaths  Recovered
6717  2020-03-17T18:33:03        0.0     0.0        0.0
6718  2020-03-17T18:33:03        0.0     0.0        0.0
6719  2020-03-17T16:13:14        0.0     0.0        0.0
6720  2020-03-17T21:33:03        0.0     0.0        0.0
6721  2020-03-18T14:13:56        0.0     0.0        0.0
```

[38]:
```
raw.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6722 entries, 0 to 6721
Data columns (total 8 columns):
SNo                6722 non-null int64
ObservationDate    6722 non-null object
Province/State     3956 non-null object
Country/Region     6722 non-null object
Last Update        6722 non-null object
Confirmed          6722 non-null float64
Deaths             6722 non-null float64
Recovered          6722 non-null float64
```

```
dtypes: float64(3), int64(1), object(4)
memory usage: 420.2+ KB
```

[39]:
```python
raw.describe()
```

[39]:

	SNo	Confirmed	Deaths	Recovered
count	6722.000000	6722.000000	6722.000000	6722.000000
mean	3361.500000	601.195924	19.855846	226.341267
std	1940.618587	4896.332140	204.486922	2556.035202
min	1.000000	0.000000	0.000000	0.000000
25%	1681.250000	2.000000	0.000000	0.000000
50%	3361.500000	13.000000	0.000000	0.000000
75%	5041.750000	108.000000	1.000000	11.000000
max	6722.000000	67800.000000	3122.000000	56927.000000

[40]:
```python
pd.DataFrame(raw.isnull().sum()).T
```

[40]:
```
   SNo  ObservationDate  Province/State  Country/Region  Last Update  \
0    0                0            2766               0            0

   Confirmed  Deaths  Recovered
0          0       0          0
```

[41]:
```python
", ".join(raw["Country/Region"].unique().tolist())
```

[41]:
```
"Mainland China, Hong Kong, Macau, Taiwan, US, Japan, Thailand, South Korea,
Singapore, Philippines, Malaysia, Vietnam, Australia, Mexico, Brazil, Colombia,
France, Nepal, Canada, Cambodia, Sri Lanka, Ivory Coast, Germany, Finland,
United Arab Emirates, India, Italy, UK, Russia, Sweden, Spain, Belgium, Others,
Egypt, Iran, Israel, Lebanon, Iraq, Oman, Afghanistan, Bahrain, Kuwait, Austria,
Algeria, Croatia, Switzerland, Pakistan, Georgia, Greece, North Macedonia,
Norway, Romania, Denmark, Estonia, Netherlands, San Marino,  Azerbaijan,
Belarus, Iceland, Lithuania, New Zealand, Nigeria, North Ireland, Ireland,
Luxembourg, Monaco, Qatar, Ecuador, Azerbaijan, Czech Republic, Armenia,
Dominican Republic, Indonesia, Portugal, Andorra, Latvia, Morocco, Saudi Arabia,
Senegal, Argentina, Chile, Jordan, Ukraine, Saint Barthelemy, Hungary, Faroe
Islands, Gibraltar, Liechtenstein, Poland, Tunisia, Palestine, Bosnia and
Herzegovina, Slovenia, South Africa, Bhutan, Cameroon, Costa Rica, Peru, Serbia,
Slovakia, Togo, Vatican City, French Guiana, Malta, Martinique, Republic of
Ireland, Bulgaria, Maldives, Bangladesh, Moldova, Paraguay, Albania, Cyprus, St.
Martin, Brunei, occupied Palestinian territory, ('St. Martin',), Burkina Faso,
Channel Islands, Holy See, Mongolia, Panama, Bolivia, Honduras, Congo
(Kinshasa), Jamaica, Reunion, Turkey, Cuba, Guyana, Kazakhstan, Cayman Islands,
Guadeloupe, Ethiopia, Sudan, Guinea, Antigua and Barbuda, Aruba, Kenya, Uruguay,
Ghana, Jersey, Namibia, Seychelles, Trinidad and Tobago, Venezuela, Curacao,
Eswatini, Gabon, Guatemala, Guernsey, Mauritania, Rwanda, Saint Lucia, Saint
Vincent and the Grenadines, Suriname, Kosovo, Central African Republic, Congo
```

```
(Brazzaville), Equatorial Guinea, Uzbekistan, Guam, Puerto Rico, Benin,
Greenland, Liberia, Mayotte, Republic of the Congo, Somalia, Tanzania, The
Bahamas, Barbados, Montenegro, The Gambia, Kyrgyzstan, Mauritius, Zambia,
Djibouti, Gambia, The"
```

[42]:
```
pprint(raw.loc[raw["Country/Region"] == "Others", "Province/State"].unique().
↪tolist(), compact=True)
```

```
['Cruise Ship', 'Diamond Princess cruise ship']
```

0.3.15 Data Cleaning

Note: “Infected” = ” Confirmed“-”Deaths

[43]:
```
data_cols = ["Infected", "Deaths", "Recovered"]
rate_cols = ["Fatal per Confirmed", "Recovered per Confirmed", "Fatal per␣
↪(Fatal or Recovered)"]
variable_dict = {"Susceptible": "S", "Infected": "I", "Recovered": "R",␣
↪"Deaths": "D"}
```

[44]:
```
covid_df = raw.rename({"ObservationDate": "Date", "Province/State":␣
↪"Province"}, axis=1)
covid_df["Date"] = pd.to_datetime(covid_df["Date"])
covid_df["Country"] = covid_df["Country/Region"].replace(
    {
        "Mainland China": "China",
        "Hong Kong SAR": "Hong Kong",
        "Taipei and environs": "Taiwan",
        "Iran (Islamic Republic of)": "Iran",
        "Republic of Korea": "South Korea",
        "Republic of Ireland": "Ireland",
        "Macao SAR": "Macau",
        "Russian Federation": "Russia",
        "Republic of Moldova": "Moldova",
        "Taiwan*": "Taiwan",
        "Cruise Ship": "Others",
        "United Kingdom": "UK",
        "Viet Nam": "Vietnam",
        "Czechia": "Czech Republic",
        "St. Martin": "Saint Martin",
        "Cote d'Ivoire": "Ivory Coast",
        "('St. Martin',)": "Saint Martin",
        "Congo (Kinshasa)": "Congo",
    }
)
covid_df["Province"] = covid_df["Province"].fillna("-").replace(
    {
```

```
        "Cruise Ship": "Diamond Princess cruise ship",
        "Diamond Princess": "Diamond Princess cruise ship"
    }
)
covid_df["Infected"] = covid_df["Confirmed"] - covid_df["Deaths"] -␣
 ↪covid_df["Recovered"]
covid_df[data_cols] = covid_df[data_cols].astype(int)
covid_df = covid_df.loc[:, ["Date", "Country", "Province", *data_cols]]
covid_df.tail()
```

[44]:

	Date	Country	Province	Infected	Deaths	Recovered
6717	2020-03-18	Guernsey	-	0	0	0
6718	2020-03-18	Jersey	-	0	0	0
6719	2020-03-18	Puerto Rico	-	0	0	0
6720	2020-03-18	Republic of the Congo	-	0	0	0
6721	2020-03-18	The Gambia	-	0	0	0

[45]:

```
covid_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6722 entries, 0 to 6721
Data columns (total 6 columns):
Date         6722 non-null datetime64[ns]
Country      6722 non-null object
Province     6722 non-null object
Infected     6722 non-null int64
Deaths       6722 non-null int64
Recovered    6722 non-null int64
dtypes: datetime64[ns](1), int64(3), object(2)
memory usage: 315.2+ KB
```

[46]:

```
covid_df.describe(include="all").fillna("-")
```

[46]:

	Date	Country	Province	Infected	Deaths	Recovered
count	6722	6722	6722	6722	6722	6722
unique	57	177	276	-	-	-
top	2020-03-18 00:00:00	China	-	-	-	-
freq	284	1765	2766	-	-	-
first	2020-01-22 00:00:00	-	-	-	-	-
last	2020-03-18 00:00:00	-	-	-	-	-
mean	-	-	-	354.999	19.8558	226.341
std	-	-	-	2830.26	204.487	2556.04
min	-	-	-	-105	0	0
25%	-	-	-	1	0	0
50%	-	-	-	7	0	0
75%	-	-	-	51	1	11
max	-	-	-	50633	3122	56927

[47]:
```
pd.DataFrame(covid_df.isnull().sum()).T
```

[47]:
```
   Date  Country  Province  Infected  Deaths  Recovered
0     0        0         0         0       0          0
```

[48]:
```
", ".join(covid_df["Country"].unique().tolist())
```

[48]:
```
'China, Hong Kong, Macau, Taiwan, US, Japan, Thailand, South Korea, Singapore,
Philippines, Malaysia, Vietnam, Australia, Mexico, Brazil, Colombia, France,
Nepal, Canada, Cambodia, Sri Lanka, Ivory Coast, Germany, Finland, United Arab
Emirates, India, Italy, UK, Russia, Sweden, Spain, Belgium, Others, Egypt, Iran,
Israel, Lebanon, Iraq, Oman, Afghanistan, Bahrain, Kuwait, Austria, Algeria,
Croatia, Switzerland, Pakistan, Georgia, Greece, North Macedonia, Norway,
Romania, Denmark, Estonia, Netherlands, San Marino,  Azerbaijan, Belarus,
Iceland, Lithuania, New Zealand, Nigeria, North Ireland, Ireland, Luxembourg,
Monaco, Qatar, Ecuador, Azerbaijan, Czech Republic, Armenia, Dominican Republic,
Indonesia, Portugal, Andorra, Latvia, Morocco, Saudi Arabia, Senegal, Argentina,
Chile, Jordan, Ukraine, Saint Barthelemy, Hungary, Faroe Islands, Gibraltar,
Liechtenstein, Poland, Tunisia, Palestine, Bosnia and Herzegovina, Slovenia,
South Africa, Bhutan, Cameroon, Costa Rica, Peru, Serbia, Slovakia, Togo,
Vatican City, French Guiana, Malta, Martinique, Bulgaria, Maldives, Bangladesh,
Moldova, Paraguay, Albania, Cyprus, Saint Martin, Brunei, occupied Palestinian
territory, Burkina Faso, Channel Islands, Holy See, Mongolia, Panama, Bolivia,
Honduras, Congo, Jamaica, Reunion, Turkey, Cuba, Guyana, Kazakhstan, Cayman
Islands, Guadeloupe, Ethiopia, Sudan, Guinea, Antigua and Barbuda, Aruba, Kenya,
Uruguay, Ghana, Jersey, Namibia, Seychelles, Trinidad and Tobago, Venezuela,
Curacao, Eswatini, Gabon, Guatemala, Guernsey, Mauritania, Rwanda, Saint Lucia,
Saint Vincent and the Grenadines, Suriname, Kosovo, Central African Republic,
Congo (Brazzaville), Equatorial Guinea, Uzbekistan, Guam, Puerto Rico, Benin,
Greenland, Liberia, Mayotte, Republic of the Congo, Somalia, Tanzania, The
Bahamas, Barbados, Montenegro, The Gambia, Kyrgyzstan, Mauritius, Zambia,
Djibouti, Gambia, The'
```

0.3.16 Analyze COVID Indicators Worldwide

[49]:
```
total_df = covid_df.groupby("Date").sum()
total_df[rate_cols[0]] = total_df["Deaths"] / total_df[data_cols].sum(axis=1)
total_df[rate_cols[1]] = total_df["Recovered"] / total_df[data_cols].sum(axis=1)
total_df[rate_cols[2]] = total_df["Deaths"] / (total_df["Deaths"] +␣
 ↪total_df["Recovered"])
total_df.tail()
```

[49]:
```
            Infected  Deaths  Recovered  Fatal per Confirmed  \
Date
2020-03-14     77656    5819      72624             0.037278
2020-03-15     84973    6440      76034             0.038460
```

```
2020-03-16      96332    7126      78088              0.039252
2020-03-17     108423    7905      80840              0.040093
2020-03-18     122869    8733      83313              0.040635

            Recovered per Confirmed  Fatal per (Fatal or Recovered)
Date
2020-03-14                 0.465243                        0.074181
2020-03-15                 0.454078                        0.078085
2020-03-16                 0.430128                        0.083625
2020-03-17                 0.410006                        0.089075
2020-03-18                 0.387656                        0.094876
```

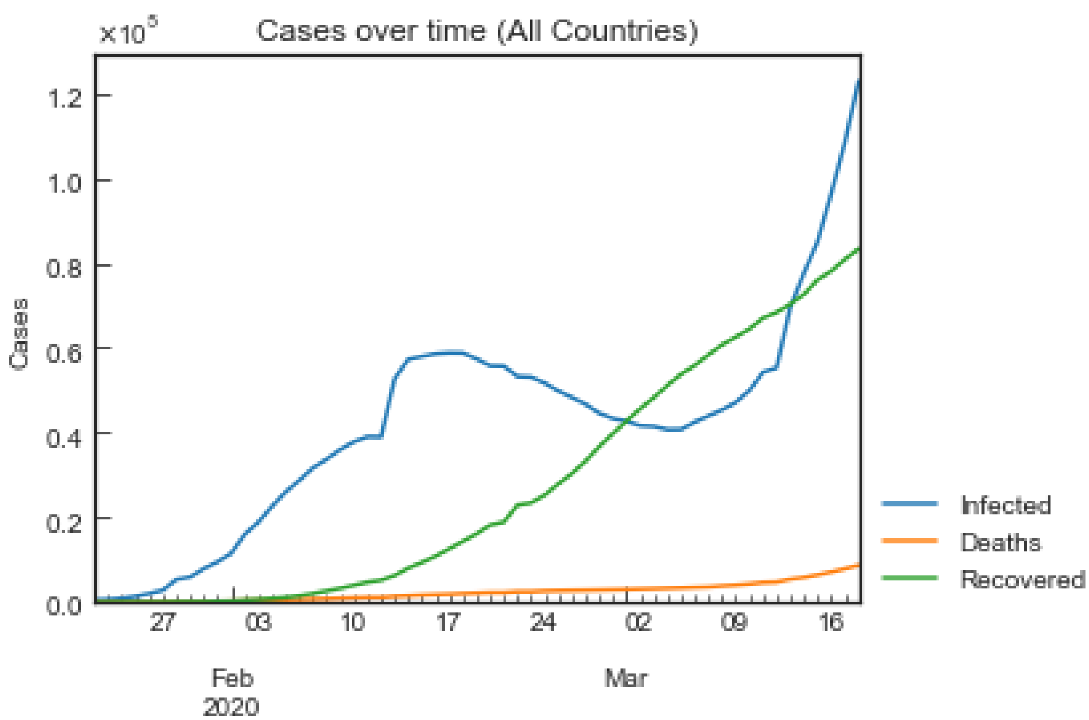
[50]:

```
f"{(total_df.index.max() - total_df.index.min()).days} days have passed from␣
↪the start date."
```

[50]: '56 days have passed from the start date.'

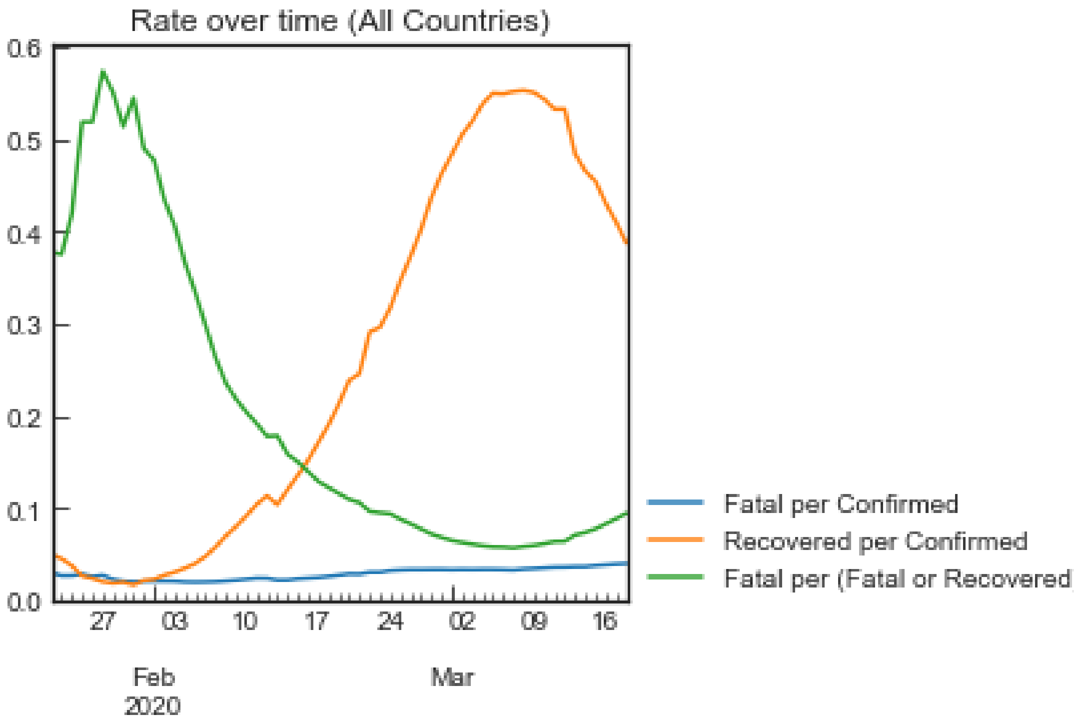
[51]:

```
line_plot(total_df[data_cols], "Cases over time (All Countries)")
```



[52]:

```
line_plot(total_df[rate_cols], "Rate over time (All Countries)", ylabel="",␣
↪math_scale=False)
```



[53]:
```
total_df[rate_cols].describe().T
```

[53]:
```
                                count      mean       std       min       25%  \
Fatal per Confirmed              57.0  0.029310  0.006153  0.020408  0.023485
Recovered per Confirmed          57.0  0.255284  0.203967  0.017365  0.048251
Fatal per (Fatal or Recovered)   57.0  0.203621  0.167785  0.057464  0.072556

                                     50%       75%       max
Fatal per Confirmed             0.029293  0.034160  0.040635
Recovered per Confirmed         0.213125  0.462511  0.552602
Fatal per (Fatal or Recovered)  0.116319  0.334123  0.573427
```

0.3.17 Hyperparameter Optimization

Using Optuna package, $(\kappa, \rho, \sigma, \tau)$ will be estimated by model fitting.

[54]:
```
%%time
sir_estimater = Estimater(
    SIR, covid_df, population_dict["Global"],
    name="All Countries")
sir_dict = sir_estimater.run()
```

```
CPU times: user 1min 5s, sys: 6.02 s, total: 1min 11s
Wall time: 1min 6s
```

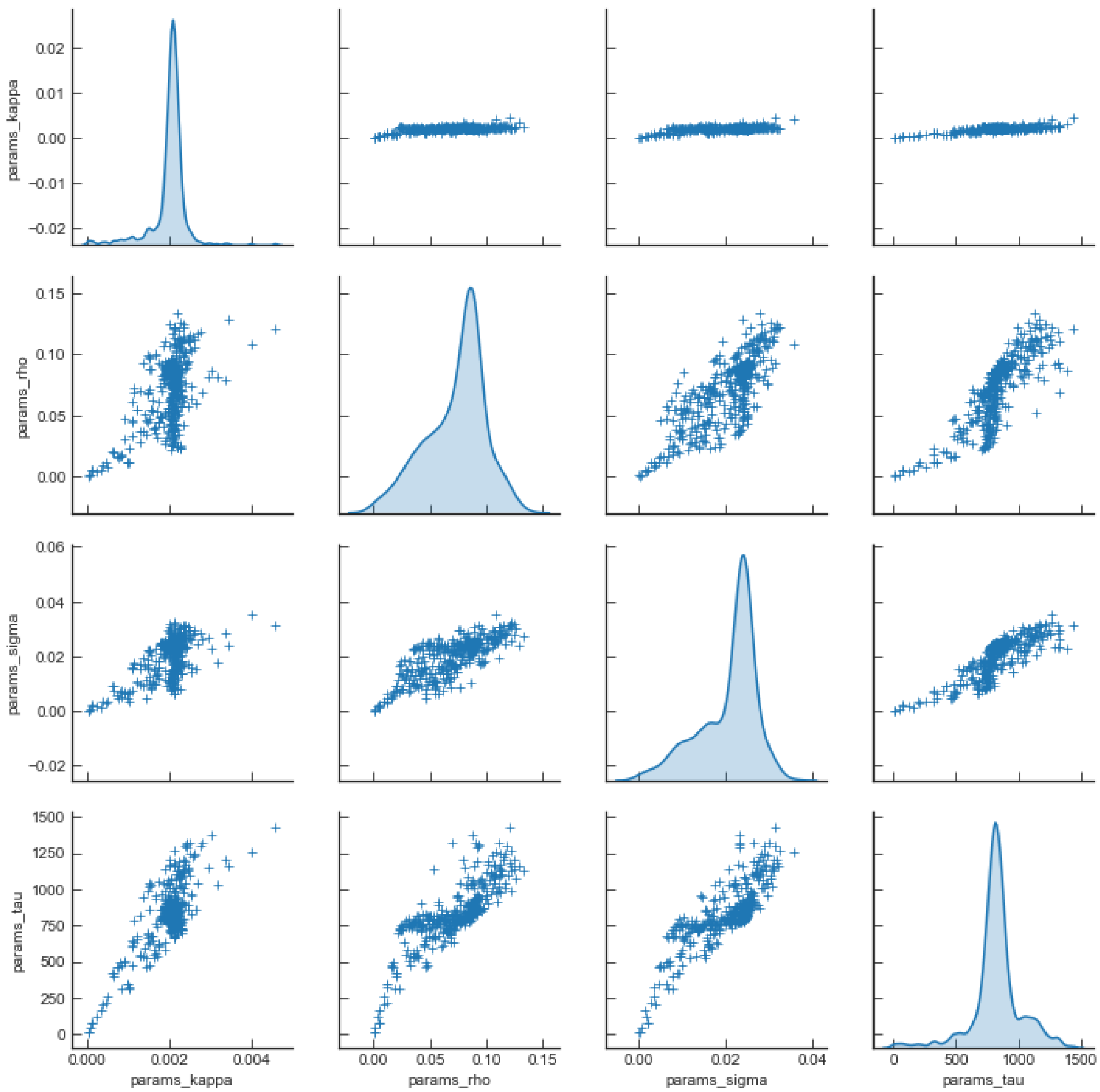
[55]:

```
%%time
sird_estimater = Estimater(
SIRD, covid_df, population_dict["Global"],
name="All Countries")
sird_dict = sird_estimater.run()
```

```
CPU times: user 1min 16s, sys: 8.13 s, total: 1min 24s
Wall time: 1min 18s
```

[56]:

```
sird_estimater.history_graph()
```



0.3.18 Computing the Basic Reproduction Number (R0) of COVID-19

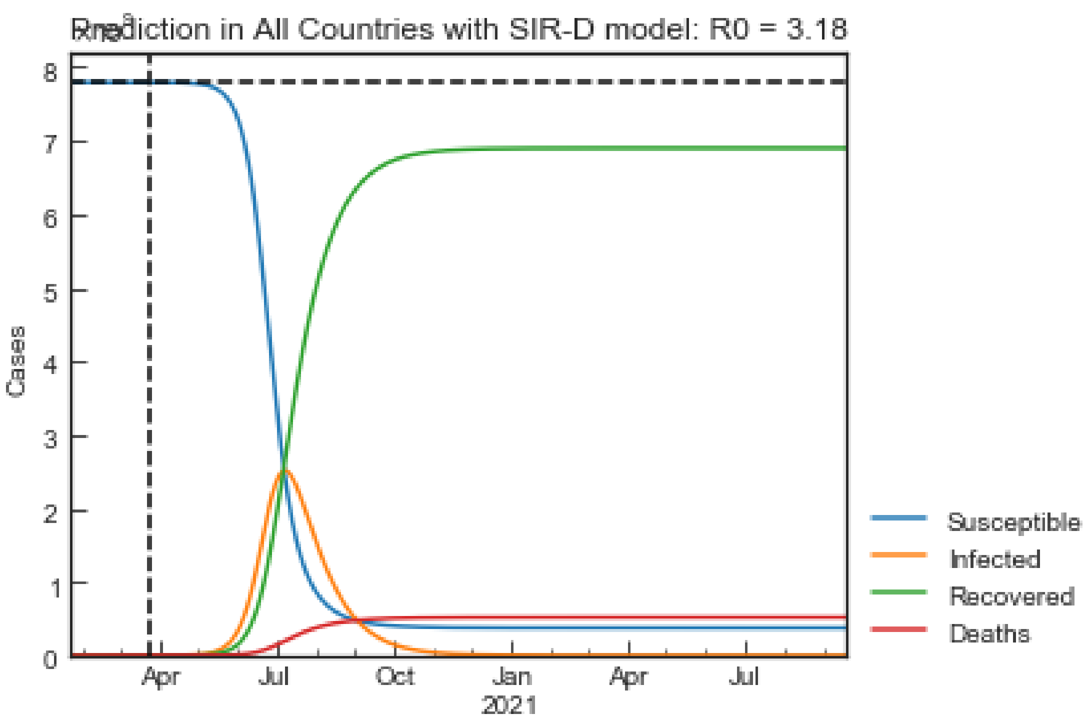
[57]:
```
pd.DataFrame.from_dict({"SIR": sir_dict, "SIR-D": sird_dict}, orient="index").
 ↪fillna("-")
```

[57]:

	tau	rho	sigma	R0	score	1/beta [day]	1/gamma [day]	kappa	1/alpha2 [day]
SIR	960	0.100004	0.032359	3.09	0.000010	6	20	-	-
SIR-D	872	0.094526	0.027665	3.18	0.000009	6	21	0.00208633	290

0.3.19 Prediction in All Countries with SIR-D Model: R0 = 3.18 (in ten millions)

[96]:
```
sird_estimater.predict_graph(step_n=1000)
```



Interpretation: We plotted the I and D trajectories of the SIRD compartment. The Infected and Dead trajectories tell us the number of individuals in those compartments over time. It can be seen from the plot that the maximum number of Infected individuals is around 255 million globally on the first week of July. Thereafter, the number of Infected individuals dropped to under 10 million around mid August, under 1 million on Oct 2020, and under 10,000 at the month of Nov 2020.

Although the SIRD model is a numerical simulation, the numbers provide us a degree to which the COVID-19 cases can surge to. These trajectories could serve as a means for governments, businesses, and individuals to plan and mitigate for such a spike in Infected cases. Everyone should work towards blunting the curve and stopping the spread as per instructions set out by their governments on personal hygiene, control measures and refraining from mass gatherings.