Open-SIR

Release 1.0.0

José I. Alamos, Felipe Huerta, Sebastián Salata

Jul 12, 2020

CONTENTS:

Open-SIR is an Open Source Python project for modelling pandemics and infectious diseases using Compartmental Models, such as the widely used Susceptible-Infected-Removed (SIR) model

The current stage of the software is *Alpha*.

CHAPTER

ONE

FEATURES

- Model the dynamics of infectious diseases
- Parameter fitting
- Calculation of confidence intervals
- CLI for interfacing with non Python environments (Bash, Node.JS, Matlab, etc).

So far, Open-SIR provides an implementation of the SIR model and the novel SIR-X model, developed by Maier and Dirk from the Robert Koch Institut

1.1 Getting Started

1.1.1 Dependencies

- Python >= 3.2
- Numpy
- Sklearn
- Scipy

1.1.2 Installation

Open-SIR and all its dependencies can be installed from the repository using `pip`:

```
git clone https://github.com/open-sir/open-sir.git
cd open-sir
pip install .
```

In order to unintall Open-SIR simply execute:

```
pip uninstall opensir
```

1.1.3 Usage example

Command line interface

It's possible to run the model using the CLI:

```
usage: opensir-cli [-h] [-m {sir,sirx}] [-t TIME] [-f FILE] [-s] [-d DELIMITER]

Run SIR or SIR-X model given a set of initial conditions and model parameters.

optional arguments:
  -h, --help            show this help message and exit
  -m {sir,sirx}, --model {sir,sirx}
  -t TIME, --time TIME  Number of days for the simulation
  -f FILE, --file FILE  Name of the input file. If missing, the input file is read␣
↪from STDIN
  -s, --suppress_header
                        Suppress CSV header
  -d DELIMITER, --delimiter DELIMITER
                        CSV delimiter
```

The input file is a TOML file with the following format

```
[initial_conds]
<cond> = value

[parameters]
<param> = value
```

For example, You can use Open-SIR to create a 6 days prediction of the number of susceptible (S), infected (I) and removed (R) population. The initial conditions represent Ealing data as of 04/04/2020. The parameters provide a prediction in the hypothetical case that no lockdown would be taking place.

```
[initial_conds]
S0 = 341555
I0 = 445
R0 = 0

[parameters]
alpha = 0.95
beta = 0.38
```

Then, it's possible to run the model with `T=6` days:

```
opensir-cli --model sir --time 6 --file input_file.txt
```

Or reading the input file from STDIN:

```
cat input_file | opensir-cli --model sir --time 6
```

The output of opensir-cli is a .csv file with the output of the model.

Note: *Note*: On Windows, the CLI must be run from Powershell or any bash shell such as Git BASH

Python API

You can replicate the predictions of the CLI with the following python script:

```
from opensir.models import SIR
my_sir = SIR() # Initialize an empty SIR model
params = [0.95, 0.38] # Define model parameters (alpha, beta)
w0 = [341555, 445, 0] # Define initial conditions (S0, I0, R0)
my_sir.set_params(p=params, initial_conds=w0) # Set model parameters
n_days = 6 # Define the amount of days to predict
my_sir.solve(n_days, n_days+1) # Call model.solve functions
sol = my_sir.fetch() # Fetch model solution
```

Try the Jupyter Notebook

Open and run the Jupyter Notebook to:

- Get an overview of the SIR model
- Explore case studies

And learn how the API can be used to:

- Build compartmental models
- Fit parameters to existing data
- Predict susceptible, infected and removed population
- Calculate confidence intervals of the predictions

1.2 Open-SIR API

1.2.1 SIR Model

Most epidemic models share a common approach on modelling the spread of a disease. The susceptible-infectious-removed (SIR) model is a simple deterministic compartmental model to predict disease spread. An objective population is divided in three groups: the susceptible (S), the infected (I) and the recovered or removed (R). These quantities enter the model as fractions of the total population P.

$$S = \frac{\text{Number of susceptible individuals}}{\text{Population size}},$$

$$I = \frac{\text{Number of infected individuals}}{\text{Population size}},$$

$$R = \frac{\text{Number of recovered or removed individuals}}{\text{Population size}},$$

As a pandemics infects and kills much more quickly than human natural rates of birth and death, the population size is assumed constant except for the individuals that recover or die. Hence, $S + I + R = P/P = 1$. The pandemics dynamics is modelled as a system of ordinary differential equations which governs the rate of change at which the percentage of susceptible, infected and recovered/removed individuals in a population evolve.

The number of possible transmissions is proportional to the number of interactions between the susceptible and infected compartments, $S \times I$:

$$\frac{dS}{dt} = -\alpha SI,$$

Where α / [time]$^{-1}$ is the transmission rate of the process which quantifies how many of the interactions between susceptible and infected populations yield to new infections per day.

The population of infected individuals will increase with new infections and decrease with recovered or removed people.

$$\frac{dI}{dt} = \alpha SI - \beta I,$$

$$\frac{dR}{dt} = \beta I,$$

Where β is the percentage of the infected population that is removed from the transmission process per day.

The infectious period, T_I / [time] , is defined as the reciprocal of the removal rate:

$$T_I = \frac{1}{\beta}.$$

In early stages of the infection, the number of infected people is much lower than the susceptible population. Hence, $S \approx 1$ making dI/dt linear and the system has the analytical solution $I(t) = I_0 \exp(\alpha - \beta)t$.

class opensir.models.**SIR**

SIR model definition

exception InconsistentDimensionsError

Raised when the length of the days array is not equal to the dimension of the observed cases, or if the length of fit_index has a length different than the length of the parameter array self.p

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception InitializationError

Raised when a function executed violating the logical sequence of the Open-SIR pipeline

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception InvalidNumberOfParametersError

Raised when the number of initial parameters is not correct

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception InvalidParameterError

Raised when an initial parameter of a value is not correct

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

block_cv(*lags=1*, *min_sample=3*)

Calculates mean squared error of the predictions as a measure of model predictive performance using block cross validation.

The cross-validation mean squared error can be used to estimate a confidence interval of model predictions.

The model needs to be initialized and fitted prior calling block_cv.

Parameters

- **lags** (*int*) – Defines the number of days that will be forecasted to calculate the mean squared error. For example, for the prediction Xp(t) and the real value X(t), the mean squared error will be calculated as mse = *1/n_boots |Xp(t+lags)-X(t+lags)|*. This provides an estimate of the mean deviation of the predictions after "lags" days.
- **min_sample** (*int*) – Number of days that will be used in the train set to make the first prediction.

Returns

tuple containing:

- **mse_avg (float):** Simple average of the mean squared error between the model prediction for "lags" days and the real observed value.
- **mse_list (numpy.array):** List of the mean squared errors using (i) points to predict the X(i+lags) value, with i an iterator that goes from n_samples+1 to the end of t_obs index.
- **p_list (numpy.array):** List of the parameters sampled on the bootstrapping as a function of time. A common use of this list is to plot the mean squared error against time, to identify time periods where the model produces the best and worst fit to the data.

Return type tuple

ci_bootstrap(*alpha=0.95*, *n_iter=1000*, *r0_ci=True*)

Calculates the confidence interval of the parameters using the random sample bootstrap method.

The model needs to be initialized and fitted prior calling ci_bootstrap

Parameters

- **alpha** (*float*) – Percentile of the confidence interval required.
- **n_iter** (*int*) – Number of random samples that will be taken to fit the model and perform the bootstrapping. Use n_iter >= 1000
- **r0_ci** (*boolean*) – Set to True to also return the reproduction rate confidence interval.

Note: This traditional random sampling bootstrap is not a good way to bootstrap time-series data , baceuse the data because X(t+1) is correlated with X(t). In any case, it provides a reference case and it will can be an useful method for other types of models. When using this function, always compare the prediction error with the interval provided by the function ci_block_cv.

Returns

tuple containing:

- **ci (numpy.array):** list of lists that contain the lower and upper confidence intervals of each parameter.
- **p_bt (numpy.array):** list of the parameters sampled on the bootstrapping. The most common use of this list is to plot histograms to visualize and try to infer the probability density function of the parameters.

Return type tuple

export (*f*, *suppress_header=False*, *delimiter=','*)
Export the output of the model in CSV format.

Note: Calling this before solve() raises an exception.

Parameters

- **f** – file name or descriptor
- **suppress_header** (*boolean*) – Set to true to suppress the CSV header
- **delimiter** (*str*) – delimiter of the CSV file

fetch ()
Fetch the data from the model.

Returns An array with the data. The first column is the time.

Return type np.array

fit (*t_obs*, *n_obs*, *fit_index=None*)
Use the Levenberg-Marquardt algorithm to fit model parameters consistent with True entries in the fit_index list.

Parameters

- **t_obs** (*numpy.ndarray*) – Vector of days corresponding to the observations of number of infected people. Must be a non-decreasing array.
- **n_obs** (*numpy.nparray*) – Vector which contains the observed epidemiological variable to fit the model against. It must be consistent with t_obs and with the initial conditions defined when building the model and using the set_parameters and set_initial_conds function. The model fit_input attribute defines against which epidemiological variable the fitting will be performed.
- **fit_index** (*list of booleans , optional*) – this list must be of the same size of the number of parameters of the model. The parameter p[i] will be fitted if fit_index[i] = True. Otherwise, the parameter will be fixed. By default, fit will only fit the first parameter of p, p[0].

Returns Reference to self

Return type Model

predict (*n_days=7*, *n_I=None*, *n_R=None*)
Predict Susceptible, Infected and Removed

Parameters

- **n_days** (*int*) – number of days to predict
- **n_I** (*int*) – number of infected at the last
- **of available data. If no number of** (*day*) –
- **is provided, the value is taken** (*infected*) –
- **the last element of the number of** (*from*) –
- **array on which the model was** (*infected*) –
- **fitted.** –

- **n_R** (*int*) – number of removed at the last
- **of available data. If no number of** –
- **is provided, the value is set as** (*removed*) –
- **number of removed calculated by the** (*the*) –
- **model as a consequence of the parameter** (SIR) –
- **fitting.** –

Returns

Array with:

- T: days of the predictions, where T[0] represents the last day of the sample and T[1] onwards the predictions.
- S: Predicted number of susceptible
- I: Predicted number of infected
- R: Predicted number of removed

Return type np.array

property r0

Returns reproduction number

Returns

$$R_0 = \alpha/\beta$$

Return type float

set_initial_conds(*array=None*, *n_S0=None*, *n_I0=None*, *n_R0=None*)

Set SIR initial conditions

Parameters **array** (*list*) – List of initial conditions [n_S0, n_I0, n_R0]. If set, all other arguments are ignored.

- n_S0: Total number of susceptible to the infection
- n_I0: Total number of infected
- n_R0: Total number of removed

Note: n_S0 + n_I0 + n_R0 = Population

Note: Internally, the model initial conditions are the ratios

- S0 = n_S0/Population
- I0 = n_I0/Population
- R0 = n_R0/Population

which is consistent with the mathematical description of the SIR model.

Returns Reference to self

Return type *SIR*

set_parameters(*array=None*, *alpha=None*, *beta=None*)
Set SIR parameters

Parameters

- **array** (*list*) – list of parameters of the model ([alpha, beta]) If set, all other arguments are ignored. All these values should be in 1/day units.
- **alpha** (*float*) – Value of *alpha* in 1/day unit.
- **beta** (*float*) – Value of *beta* in 1/day unit.

Returns Reference to self

Return type *SIR*

set_params(*p*, *initial_conds*)
Set model parameters.

Parameters

- **p** (*dict or array*) – parameters of the model (alpha, beta). All these values should be in 1/day units. If a list is used, the order of parameters is [alpha, beta].
- **initial_conds** (*list*) – Initial conditions (n_S0, n_I0, n_R0), where:
  - n_S0: Total number of susceptible to the infection
  - n_I0: Toral number of infected
  - n_R0: Total number of removed

  Note n_S0 + n_I0 + n_R0 = Population

  Internally, the model initial conditions are the ratios
  - S0 = n_S0/Population
  - I0 = n_I0/Population
  - R0 = n_R0/Population

  which is consistent with the mathematical description of the SIR model.

  If a list is used, the order of initial conditions is [n_S0, n_I0, n_R0]

Deprecated: This function is deprecated and will be removed soon. Please use *set_parameters()* and *set_initial_conds()*

Returns reference to self

Return type *SIR*

solve(*tf_days=7*, *numpoints=7*)
Solve using children class model.

Parameters

- **tf_days** (*int*) – number of days to simulate
- **numpoints** (*int*) – number of points for the simulation.

Returns Reference to self

Return type Model

1.2.2 SIR-X Model

The SIR-X model extends the SIR model adding two parameters: the quarantine rate κ and the containement rate κ_0. This extension allows the model to capture the "decrease" of susceptible population owing containment and quarantine measures.

class opensir.models.**SIRX**
SIRX model definition

exception InconsistentDimensionsError
Raised when the length of the days array is not equal to the dimension of the observed cases, or if the length of fit_index has a length different than the length of the parameter array self.p

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception InitializationError
Raised when a function executed violating the logical sequence of the Open-SIR pipeline

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception InvalidNumberOfParametersError
Raised when the number of initial parameters is not correct

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception InvalidParameterError
Raised when an initial parameter of a value is not correct

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

block_cv(*lags=1*, *min_sample=3*)
Calculates mean squared error of the predictions as a measure of model predictive performance using block cross validation.

The cross-validation mean squared error can be used to estimate a confidence interval of model predictions.

The model needs to be initialized and fitted prior calling block_cv.

Parameters

- **lags** (*int*) – Defines the number of days that will be forecasted to calculate the mean squared error. For example, for the prediction Xp(t) and the real value X(t), the mean squared error will be calculated as mse = *1/n_boots* $|Xp(t+lags)-X(t+lags)|$. This provides an estimate of the mean deviation of the predictions after "lags" days.
- **min_sample** (*int*) – Number of days that will be used in the train set to make the first prediction.

Returns

tuple containing:

- **mse_avg (float):** Simple average of the mean squared error between the model prediction for "lags" days and the real observed value.
- **mse_list (numpy.array):** List of the mean squared errors using (i) points to predict the X(i+lags) value, with i an iterator that goes from n_samples+1 to the end of t_obs index.
- **p_list (numpy.array):** List of the parameters sampled on the bootstrapping as a function of time. A common use of this list is to plot the mean squared error against time, to identify time periods where the model produces the best and worst fit to the data.

Return type tuple

ci_bootstrap(*alpha=0.95*, *n_iter=1000*, *r0_ci=True*)

Calculates the confidence interval of the parameters using the random sample bootstrap method.

The model needs to be initialized and fitted prior calling ci_bootstrap

Parameters

- **alpha** (*float*) – Percentile of the confidence interval required.
- **n_iter** (*int*) – Number of random samples that will be taken to fit the model and perform the bootstrapping. Use n_iter >= 1000
- **r0_ci** (*boolean*) – Set to True to also return the reproduction rate confidence interval.

Note: This traditional random sampling bootstrap is not a good way to bootstrap time-series data , baceuse the data because X(t+1) is correlated with X(t). In any case, it provides a reference case and it will can be an useful method for other types of models. When using this function, always compare the prediction error with the interval provided by the function ci_block_cv.

Returns

tuple containing:

- **ci (numpy.array):** list of lists that contain the lower and upper confidence intervals of each parameter.
- **p_bt (numpy.array):** list of the parameters sampled on the bootstrapping. The most common use of this list is to plot histograms to visualize and try to infer the probability density function of the parameters.

Return type tuple

export(*f*, *suppress_header=False*, *delimiter=','*)

Export the output of the model in CSV format.

Note: Calling this before solve() raises an exception.

Parameters

- **f** – file name or descriptor
- **suppress_header** (*boolean*) – Set to true to suppress the CSV header
- **delimiter** (*str*) – delimiter of the CSV file

fetch()
Fetch the data from the model.

Returns An array with the data. The first column is the time.

Return type np.array

fit(*t_obs*, *n_obs*, *fit_index=None*)
Use the Levenberg-Marquardt algorithm to fit model parameters consistent with True entries in the fit_index list.

Parameters

- **t_obs** (*numpy.ndarray*) – Vector of days corresponding to the observations of number of infected people. Must be a non-decreasing array.
- **n_obs** (*numpy.nparray*) – Vector which contains the observed epidemiological variable to fit the model against. It must be consistent with t_obs and with the initial conditions defined when building the model and using the set_parameters and set_initial_conds function. The model fit_input attribute defines against which epidemiological variable the fitting will be performed.
- **fit_index** (*list of booleans , optional*) – this list must be of the same size of the number of parameters of the model. The parameter p[i] will be fitted if fit_index[i] = True. Otherwise, the parameter will be fixed. By default, fit will only fit the first parameter of p, p[0].

Returns Reference to self

Return type Model

property pcl
Returns public containment leverage P

Returns

$$P = \frac{\kappa_0}{\kappa_0 + \kappa}$$

Return type float

predict(*n_days=7*, *n_X=None*, *n_R=None*)
Predicts Susceptible, Infected, Removed and Quarantined in the next n_days from the last day of the sample used to train the model.

Parameters

- **n_days** (*int*) – number of days to predict
- **n_X** (*int*) – number of confirmed cases at the last
- **of available data. If no number of** (*day*) –
- **cases is provided, the value is taken** (*confirmed*) –
- **the last element of the number of** (*from*) –
- **cases array on which the model was** (*confirmed*) –
- **fitted.** –
- **n_R** (*int*) – number of removed at the last
- **of available data. If no number of** –
- **is provided, the value is set as** (*removed*) –

- **number of removed calculated by the**(*the*) –
- **model as a consequence of the parameter**(*SIR-X*) –
- **fitting.** –

Returns

Array with:

- T: days of the predictions, where T[0] represents the last day of the sample and T[1] onwards the predictions.
- S: Predicted number of susceptible
- I: Predicted number of infected
- R: Predicted number of removed
- X: Predicted number of quarantined

Return type np.array

property q_prob

Returns quarantine probability Q

Returns

$$Q = \frac{\kappa_0 + \kappa}{\beta + \kappa_0 + \kappa}$$

Return type float

property r0

Returns reproduction number

Returns

$$R_0 = \alpha/\beta$$

Return type float

property r0_eff

Returns effective reproduction rate $R_{0,eff}$

Returns

$$R_{0,eff} = \alpha T_{I,eff}$$

Return type float

set_initial_conds(*array=None*, *n_S0=None*, *n_I0=None*, *n_R0=None*, *n_X0=None*)

Set SIR-X initial conditions

Parameters **array** (*list*) – List of initial conditions [n_S0, n_I0, n_R0, n_X0]. If set, all other arguments are ignored.

- n_S0: Total number of susceptible to the infection
- n_I0: Total number of infected
- n_R0: Total number of removed
- n_X0: Total number of quarantined

Note: n_S0 + n_I0 + n_R0 + n_X0 = Population

Note: Internally, the model initial conditions are the ratios

- S0 = n_S0/Population
- I0 = n_I0/Population
- R0 = n_R0/Population
- X0 = n_X0/Population

which is consistent with the mathematical description of the SIR-X model.

Returns Reference to self

Return type *SIRX*

set_parameters(*array=None*, *alpha=None*, *beta=None*, *kappa_0=None*, *kappa=None*, *inf_over_test=None*)

Set SIR-X parameters

Parameters

- **array** (*list*) – list of parameters of the model ([alpha, beta, kappa_0, kappa, inf_over_test]) If set, all other arguments are ignored. All these values should be in 1/day units.
- **alpha** (*float*) – Value of *alpha* in 1/day unit.
- **beta** (*float*) – Value of *beta* in 1/day unit.
- **kappa_0** (*float*) – Value of *kappa_0* in 1/day unit.
- **kappa** (*float*) – Value of *kappa* in 1/day unit.
- **inf_over_test** (*float*) – Value of infected/tested

Returns Reference to self

Return type *SIRX*

set_params(*p*, *initial_conds*)

Set model parameters.

Parameters

- **p** (*list*) – parameters of the model (alpha, beta, kappa_0, kappa, inf_over_test). All these values should be in 1/day units. If a list is used, the order of parameters is [alpha, beta, kappa_0, kappa, inf_over_test]
- **initial_conds** (*list*) – Initial conditions (n_S0, n_I0, n_R0, n_X0), where:
  - n_S0: Total number of susceptible to the infection
  - n_I0: Total number of infected
  - n_R0: Total number of removed
  - n_X0: Total number of quarantined

  Note: n_S0 + n_I0 + n_R0 + n_X0 = Population

  Internally, the model initial conditions are the ratios
  - S0 = n_S0/Population
  - I0 = n_I0/Population

- R0 = n_R0/Population
- X0 = n_X0/Population

which is consistent with the mathematical description of the SIR model.

If a list is used, the order of initial conditions is [n_S0, n_I0, n_R0, n_X0]

Deprecated: This function is deprecated and will be removed soon. Please use *set_parameters()* and *set_initial_conds()*

Returns Reference to self

Return type *SIRX*

solve (*tf_days=7*, *numpoints=7*)
Solve using children class model.

Parameters

- **tf_days** (*int*) – number of days to simulate
- **numpoints** (*int*) – number of points for the simulation.

Returns Reference to self

Return type Model

property t_inf_eff
Returns effective infectious period

Returns

$$T_{I,eff} = (\beta + \kappa + \kappa_0)^{-1}$$

Return type float

1.3 Tutorials

1.3.1 Modelling pandemics using compartmental models

Coronavirus COVID-19 is a pandemic that is spreading quickly worlwide. Up to the 29th of March, there are 666,211 cases confirmed, 30,864 deaths and 141,789 recovered people worldwide. Governments and citizens are taking quick decisions to limit the spread of the virus and minimize the number of infected and deaths. These decisions are taken based on the experts opinion, which justify their claims based in the results of predictive models.

This Jupyter Notebook is an effort to decrease the access barriers to state of the art yet simple models that can be used to take public policy decisions to limit disease spread and save lives.

SIR model

Most epidemic models share a common approach on modelling the spread of a disease. The SIR model is a simple deterministic compartmental model to predict disease spread. An objective population is divided in three groups: the susceptible (S), the infected (I) and the recovered or removed (R). These quantities enter the model as fractions of the total population P:

$$S = \frac{\text{Number of susceptible individuals}}{\text{Population size}}$$

$$I = \frac{\text{Number of infected individuals}}{\text{Population size}}$$

$$R = \frac{\text{Number of recovered or removed individuals}}{\text{Population size}}$$

As a pandemics infects and kills much more quickly than human natural rates of birth and death, the population size is assumed constant except for the individuals that recover or die. Hence, $S + I + R = P/P = 1$. The pandemics dynamics is modelled as a system of ordinary differential equations which governs the rate of change at which the percentage of susceptible, infected and recovered/removed individuals in a population evolve.

The number of possible transmissions is proportional to the number of interactions between the susceptible and infected populations, `$S \times I $`:

$$\frac{dS}{dt} = -\alpha SI.$$

Where α is the reproduction rate of the process which quantifies how many of the interactions between susceptible and infected populations yield to new infections per day.

The population of infected individuals will increase with new infections and decrease with recovered or removed people.

$$\frac{dI}{dt} = \alpha SI - \beta I,$$

$$\frac{dR}{dt} = \beta I.$$

Where `$ \beta $` is the percentage of the infected population that is removed from the transmission process per day.

In early stages of the infection, the number of infected people is much lower than the susceptible populations. Hence, $S \approx 1$ making dI/dt linear and the system has the analytical solution $I(t) = I_0 \exp(\alpha - \beta)t$.

Numerical implementation - SIR model

Three python packages are imported: numpy for numerical computing, matplotlib.pyplot for visualization and the numerical integration routine odeint from scipy.integrate:

[1]:
```
# Uncomment this cell for code formatting using nb_black
# %load_ext nb_black
```

[2]:
```
import numpy as np  # Numerical computing
import matplotlib.pyplot as plt  # Visualization
from scipy.integrate import odeint  # ODE system numerical integrator
from scipy.optimize import curve_fit  # Minimize squared errors using LM method
```

Implementing systems of ordinary differential equations (ODEs) in python is straightforward. First, a function is created to represent the system inputs and outputs. The inputs of the function are a vector of state variables $\vec{w}$, the independent variable t and a vector of parameters $\vec{p}$. The output of the function must be the right hand side of the ODE system as a list.

Following this approach, the SIR model can be implemented as it follows:

$$\vec{w} = [S, I, R]$$

$$\vec{p} = [\alpha, \beta]$$

And t enters directly. The function return will be the list of ODEs.

$$\vec{f} = \left[\frac{dS}{dt}, \frac{dI}{dt}, \frac{dR}{dt}\right]$$

So $\vec{f} = \text{sir}(\vec{w}, t, \vec{p})$.

The solution of this system is a vector field $\vec{w} = [S(t), I(t), R(t)]$. In day to day words, it gives the percentage of the population who are susceptible (S), infected (I) and recovered or removed R(t) as a function of time. There is no analytical solution for this system. However, a numerical solution can be obtained using a numerical integrator. In this implementation, the function scipy.odeint is used to integrate the differential system. The ODE system of the SIR model was implemented in the function sirx(t,w,p) on the module model.py. The solver is implemented in the function _solve on the module model.py.

SIR-X model

A new epidemic model based in SIR, SIRX, was developed by the Robert Koch Institut and is implemented in what follows. A full description of the model is available in the Robert Koch Institut SIRX model webiste.

The ODE system of the SIR-X model was implemented in the function sirx(t,w,p) on the module model.py

Usage example

Case study

The borough of Ealing, in London, is selected arbitrarly as one of the authors is living there at the moment. According to the UK office for National Statistics, the population of Ealing by mid-year 2018 is 342,000. The number of reported infections at 29/03/2020 is 241.

Model parameters

As an implementation examples, the parameter β is estimated from the methodology followed by the Robert Koch Institut SIRX model webiste. The institute estimated the a removal rate value $\beta = 0.38/d$ (mean infections time $T_I = 1/\beta = 2.6d$) based on one third of the reported average infections preioud of moderate cases in Mainland China.

The reproduction number is fixed $R_0 = \alpha/\beta = 2.5$ as a first approximation.

Please note that the predictions of this model shouldn't be taken in consideratin, as the SIR model doesn't consider dynamic variation of model parameters, which is observed in reality.

Solution and implementation

The aim of this API is to provide an user friendly approach to build a SIR model and fit it to a target dataset in order to make predictions in few lines of code.

[3]:
```
# Use Ealing as an example to determine model initial conditions
# Input data must be np.array
Ealing_data = np.array(
    [8, 18, 20, 28, 31, 42, 53, 54, 80, 97, 106, 123, 136, 165, 209, 241]
)  # N_of infected

P_Ealing = 342000  # Ealing population ONS 2018 mid year
I_Ealing = 8  # Infected people at 14/03/2020
R_Ealing = 0  # Recovered people at 29/03/2020
n_days = len(Ealing_data)

# Input parameters
beta = 0.38  # Per day
alpha = 2.5 * beta  # WHO estimate
```

Calculate model parameters and initial conditions

[4]:
```
# Calculate initial conditions in terms of total number of individuals
S0 = P_Ealing - I_Ealing
I0 = I_Ealing
R0 = R_Ealing  # Recovered people

# Construct vector of parameters
params = [alpha, beta]

# Construct vector of initial conditions
w0 = [S0, I0, R0]
```

Build the model with the default parameters and predict the number of susceptible, infected and recovered people in the Ealing borough.

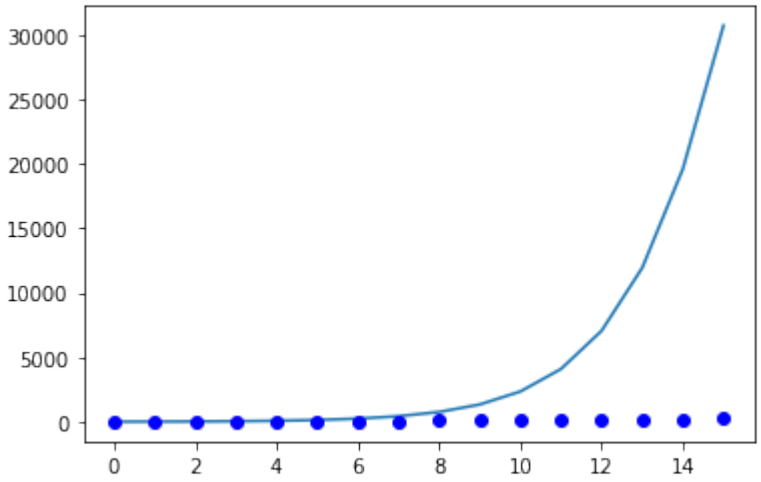
[5]:
```
# These lines are required only if opensir wasn't installed using pip install, or if␣
↪opensir is running in the pipenv virtual environment
import sys

path_opensir = "../../"
sys.path.append(path_opensir)

# Import SIR and SIRX models
from opensir.models import SIR, SIRX
```

[6]:
```
# Initialize an emplty SIR model
my_SIR = SIR()
# Set model parameters
my_SIR.set_params(p=params, initial_conds=w0)

# Call model.solve functions with the time in days and the number of points
```

(continues on next page)

(continued from previous page)

```
# as the number of days
my_SIR.solve(n_days - 1, n_days)
# Unpack the numerical solution using the model.fetch() method
sol = my_SIR.fetch()
# Unpack the numerical solution for the susceptible (S), infected (I) and recovered␣
↪or removed (R)
S_sir = sol[:, 1]
I_sir = sol[:, 2]
R_sir = sol[:, 3]
# Plot the results
# Define array of days. Note that the initial day is the day "zero", so
# the final day is the number of days minus one. This is divided in n_days
# intervals to be consistent with the input
days_list = np.linspace(0, n_days - 1, n_days)
plt.plot(days_list, I_sir)
plt.plot(days_list, Ealing_data, "bo")
plt.show()
my_SIR.r0
```



[6]: 2.5

If the default parameters are used, the results are completely unreliable. Indeed, the model predicts more than 150 times more people infected. This is why a model shouldn't be used blindly, and always the parameters must be checked. In UK, Social distancing started voluntarily on the 16th of February, and the lockdown started on the 23rd of February. The effect of this policy change in terms of our model, is a decrease in the reproduction rate $R_0 = \alpha/\beta$. As the national health system (NHS) of UK didn't reach full capacity on the period between the 15th and the 29th of March, it is reasonable to assume that the main change occured owing to a decrease in the transmission rate α.

To obtain a more realistic approximation, the parameter can be modified to better reproduce the observed data. This process is named **parameter fitting** and it is widely used not only on epidemiology, but in any discipline which uses mathematical models to make prediction.

The function model.fit() enables to fit the desired parameters to a certain dataset. The parameter fitting is straightforward using open-sir:

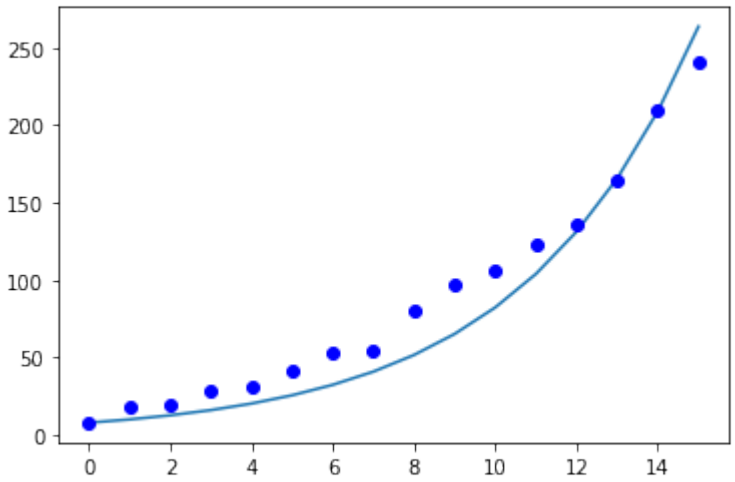
Parameter Fitting

Fitting R_0 through α keeping β constant

In the following case study, R_0 will be fitted to minimize the mean squared error between the model predictions and UK historical data on the Ealing borough in the time period between the 15th and the 29th of March of 2020.

[7]:
```
# Create SIR with default parameters
my_SIR_fitted = SIR()
my_SIR_fitted.set_params(params, w0)

# Fit parameters
w = my_SIR_fitted.fit(days_list, Ealing_data, fit_index=[True, False])
# Print the fitted reproduction rate
print("Fitted reproduction rate R_0 = %.2f" % my_SIR_fitted.r0)
# Build the new solution
my_SIR_fitted.solve(n_days - 1, n_days)
# Extract solution
sol = my_SIR_fitted.fetch()
# Plot the results

plt.plot(days_list, sol[:, 2])
plt.plot(days_list, Ealing_data, "bo")
plt.show()
```

```
Fitted reproduction rate R_0 = 1.61
```



[8]:
```
Ealing_data
```

[8]:
```
array([  8,  18,  20,  28,  31,  42,  53,  54,  80,  97, 106, 123, 136,
       165, 209, 241])
```

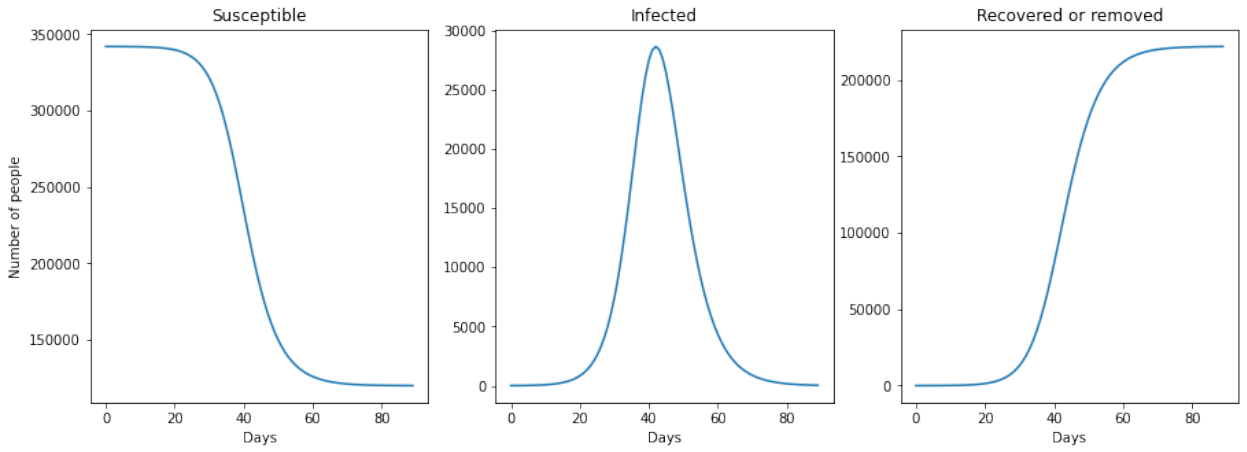
DANGER ZONE

This is extremely dangerous as R_0 is extremely likely to change with time. However we have seen many people taking decisions in this kind of analysis. Use it at your own risk and with a metric ton of salt.

Example: predict the total number of infections and the time where the number of infected people is maximum

[9]:
```
long_term_days = 90
# Convert into seconds
tf_long = long_term_days - 1
sol_long = my_SIR_fitted.solve(tf_long, long_term_days).fetch()
N_S_long = sol_long[:, 1]
N_I_long = sol_long[:, 2]
N_R_long = sol_long[:, 3]
```

[10]:
```
# Plot the number of susceptible, infected and recovered in a two months period
tspan_long = np.linspace(0, tf_long, long_term_days)
plt.figure(figsize=[15, 5])
plt.subplot(1, 3, 1)
plt.plot(tspan_long, N_S_long)
plt.xlabel("Days")
plt.ylabel("Number of people")
plt.title("Susceptible")
plt.subplot(1, 3, 2)
plt.plot(tspan_long, N_I_long)
plt.xlabel("Days")
plt.title("Infected")
plt.subplot(1, 3, 3)
plt.plot(tspan_long, N_R_long)
plt.title("Recovered or removed")
plt.xlabel("Days")
plt.show()
```



It can be observed that the SIR model reproduces the all familiar infection bell, as well as the evolution of susceptible and recovered population. It is interesting to observe that if no measures are taken in a $R_0 = 1.47$ scenario, 65% of the Ealing population would be infected in three months.

Sensitivity to R_0

A known weakness of all pandemics prediction model is the sensitivity to their parameters. In the following case study, R_0 will be fitted to minimize the mean squared error between the model predictions and UK historical data on the Ealing borough in the time period between the 15th and the 29th of March of 2020.

[11]:
```
def compare_infections(model, tf, numpoints, alpha_list=2.5, abserr=1e-8, relerr=1e-
↪6):
    """ compare_infections compare SIR model predictions against
    a list of alpha values

    Inputs:
    w0: Initial conditions
    t: Time vector /
    alpha_list: list or numpy array of values of alpha to be tested

    Outputs:
    S_list: List of predictions for the fraction of susceptible population for each␣
↪alpha
    I_list: List of predictions for the fraction of infected population for each alpha
    R_list: List of predictions for the fraction of recovered/removed population for␣
↪each alpha
    """
    S_list = []
    I_list = []
    R_list = []

    for i in alpha_list:
        # Update parameter list
        model.p[0] = i
        wsol = model.solve(tf, numpoints).fetch()
        S_list.append(wsol[:, 1])
        I_list.append(wsol[:, 2])
        R_list.append(wsol[:, 3])
    return S_list, I_list, R_list
```

Generate predictions for each alpha

[12]:
```
alpha_list = beta * np.array([1.5, 1.6, 1.7])
S_list, I_list, R_list = compare_infections(my_SIR, tf_long, long_term_days, alpha_
↪list)
```

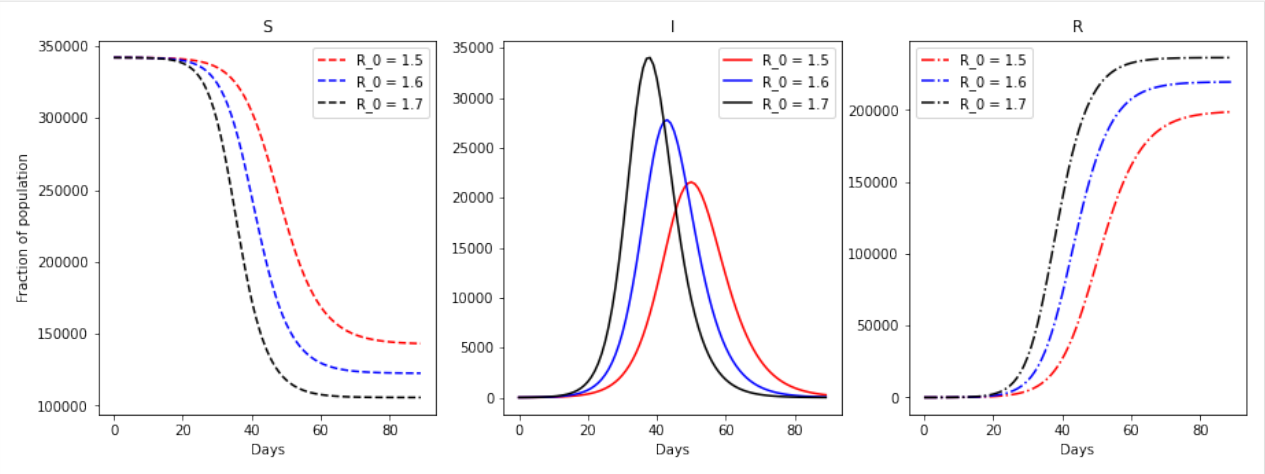
[13]:
```
col = ["r", "b", "k"]
plt.figure(figsize=[15, 5])
for i in range(len(S_list)):
    plt.subplot(1, 3, 1)
    plt.plot(tspan_long, S_list[i], col[i] + "--")
    plt.legend(["R_0 = 1.5", "R_0 = 1.6", "R_0 = 1.7"])
    plt.xlabel("Days")
    plt.ylabel("Fraction of population")
    plt.title("S")
    plt.subplot(1, 3, 2)
    plt.plot(tspan_long, I_list[i], col[i])
    plt.legend(["R_0 = 1.5", "R_0 = 1.6", "R_0 = 1.7"])
    plt.xlabel("Days")
```

(continues on next page)

(continued from previous page)

```
    plt.title("I")
    plt.subplot(1, 3, 3)
    plt.plot(tspan_long, R_list[i], col[i] + "-.")
    plt.legend(["R_0 = 1.5", "R_0 = 1.6", "R_0 = 1.7"])
    plt.xlabel("Days")
    plt.title("R")

plt.show()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/open-sir/envs/latest/lib/python3.7/
↪site-packages/ipykernel_launcher.py:4: MatplotlibDeprecationWarning: Adding an axes␣
↪using the same arguments as a previous axes currently reuses the earlier instance. ␣
↪In a future version, a new instance will always be created and returned.  Meanwhile,
↪ this warning can be suppressed, and the future behavior ensured, by passing a␣
↪unique label to each axes instance.
  after removing the cwd from sys.path.
/home/docs/checkouts/readthedocs.org/user_builds/open-sir/envs/latest/lib/python3.7/
↪site-packages/ipykernel_launcher.py:10: MatplotlibDeprecationWarning: Adding an␣
↪axes using the same arguments as a previous axes currently reuses the earlier␣
↪instance.  In a future version, a new instance will always be created and returned.␣
↪ Meanwhile, this warning can be suppressed, and the future behavior ensured, by␣
↪passing a unique label to each axes instance.
  # Remove the CWD from sys.path while we load stuff.
/home/docs/checkouts/readthedocs.org/user_builds/open-sir/envs/latest/lib/python3.7/
↪site-packages/ipykernel_launcher.py:15: MatplotlibDeprecationWarning: Adding an␣
↪axes using the same arguments as a previous axes currently reuses the earlier␣
↪instance.  In a future version, a new instance will always be created and returned.␣
↪ Meanwhile, this warning can be suppressed, and the future behavior ensured, by␣
↪passing a unique label to each axes instance.
  from ipykernel import kernelapp as app
/home/docs/checkouts/readthedocs.org/user_builds/open-sir/envs/latest/lib/python3.7/
↪site-packages/ipykernel_launcher.py:4: MatplotlibDeprecationWarning: Adding an axes␣
↪using the same arguments as a previous axes currently reuses the earlier instance. ␣
↪In a future version, a new instance will always be created and returned.  Meanwhile,
↪ this warning can be suppressed, and the future behavior ensured, by passing a␣
↪unique label to each axes instance.
  after removing the cwd from sys.path.
/home/docs/checkouts/readthedocs.org/user_builds/open-sir/envs/latest/lib/python3.7/
↪site-packages/ipykernel_launcher.py:10: MatplotlibDeprecationWarning: Adding an␣
↪axes using the same arguments as a previous axes currently reuses the earlier␣
↪instance.  In a future version, a new instance will always be created and returned.␣
↪ Meanwhile, this warning can be suppressed, and the future behavior ensured, by␣
↪passing a unique label to each axes instance.
  # Remove the CWD from sys.path while we load stuff.
/home/docs/checkouts/readthedocs.org/user_builds/open-sir/envs/latest/lib/python3.7/
↪site-packages/ipykernel_launcher.py:15: MatplotlibDeprecationWarning: Adding an␣
↪axes using the same arguments as a previous axes currently reuses the earlier␣
↪instance.  In a future version, a new instance will always be created and returned.␣
↪ Meanwhile, this warning can be suppressed, and the future behavior ensured, by␣
↪passing a unique label to each axes instance.
  from ipykernel import kernelapp as app
```

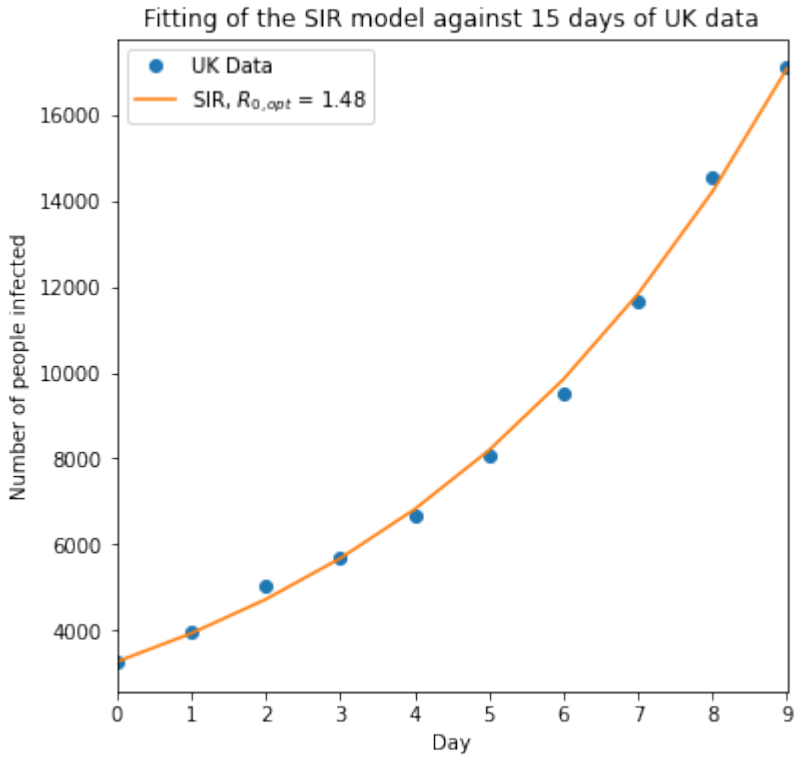


We observe that a change as little as 6% in the reproduction rate, can change dramatically the dynamic of the pandemic

Example 4: Fit R_0 for UK values

sourced from UK Arcgis

[14]:
```
P_UK = 67886011
# Data up to 28th of March
I_UK= np.array([3269, 3983, 5018, 5683, 6650, 8077, 9529, 11658, 14543, 17089])
n_days = len(I_UK) # Final day
t_d = np.linspace(0,n_days-1,n_days)

n_S0 = P_UK-I_UK[0]
n_I0 = I_UK[0]
n_R0 = 0
n0_UK = [n_S0, n_I0, n_R0]
p = [alpha,beta]

# Create empty model
SIR_UK = SIR()
SIR_UK.set_params(p,n0_UK)
# Train model
SIR_UK.fit(t_d, I_UK)
# Build numerical solution
I_opt = SIR_UK.solve(n_days-1,n_days).fetch()[:,2]
# lag = 6

R_opt = SIR_UK.r0 #

plt.figure(figsize=[6,6])
plt.plot(t_d,I_UK,'o')
plt.plot(t_d, I_opt)
plt.legend(["UK Data","SIR, $R_{0,opt}$ = %.2f"%R_opt])
plt.title("Fitting of the SIR model against 15 days of UK data")
plt.ylabel("Number of people infected")
plt.xlabel("Day")
plt.xlim([min(t_d),max(t_d)])
plt.show()
```



[15]:
```
MSE = sum(np.sqrt((I_opt - I_UK) ** 2)) / len(I_UK)
print("Mean squared error on the model in the train dataset %.2f" % MSE)
```

```
Mean squared error on the model in the train dataset 147.51
```

The mean squared error calculated above indicates the average error difference between the model fitting and the train data. It is a measure of wellness of fit, but it doesn't provide information about how accurately the model predicts the number of infected.

The error in the future predictions can be estimating through confidence intervals.

Making out of sample predictions using the `model.predict` function

The `model.predict` function allows out of sample predictions. It recieves one mandatory parameter, n_days, and two optional parameters. The two optional parameters are the observed number of infected (`n_I`) and the number of recovered (`n_R`) individuals. If `n_I` is not provided, the last value of the train set is used, while if `n_R` is not provided it is estimated from the fitted SIR model.

[16]:
```
# Obtain the results 7 days after the train data ends
pred_7 = SIR_UK.predict(7)
print("T n_S \t   n_I\t n_R")
for i in pred_7:
    print(*i.astype(int))
```

```
T n_S      n_I   n_R
0 67840348 17089 28573
1 67829779 20529 35701
2 67817087 24659 44263
```

(continues on next page)

(continued from previous page)

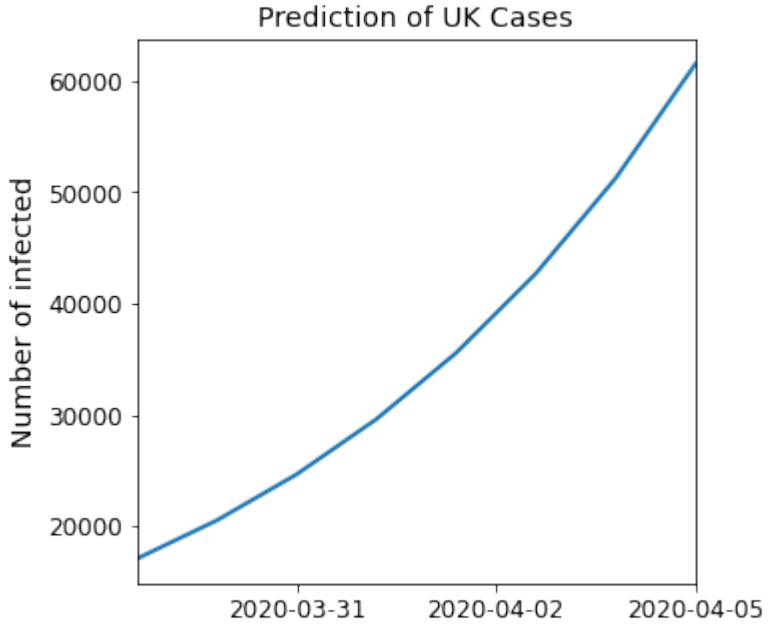
```
3 67801844 29618 54548
4 67783543 35568 66899
5 67761573 42706 81730
6 67735206 51267 99536
7 67703572 61529 120909
```

Visualize predictions

Predict the next seven days of the spread of COVID-19 in the UK, considering the 29th of March as the last day of the sample data on which the SIR model was fitted.

[17]:
```
import datetime  # Import datetime module from the standard library

# Create a date time range based on the number of rows of the prediction
numdays = pred_7.shape[0]
day_zero = datetime.datetime(2020, 3, 29)
date_list = [day_zero + datetime.timedelta(days=x) for x in range(numdays)]
```

[18]:
```
# Extract figure and axes
fig, ax = plt.subplots(figsize=[5, 5])
plt.plot(date_list, pred_7[:, 2], linewidth=2)
plt.title("Prediction of UK Cases", size=14)
plt.ylabel("Number of infected", size=14)
# Remove trailing space
plt.xlim(date_list[0], date_list[-1])
# Limit the amount of data displayed
ax.xaxis.set_major_locator(plt.MaxNLocator(3))
# Increase the size of the ticks
ax.tick_params(labelsize=12)
plt.show()
```



Calculate confidence intervals

[19]:
```
# Get the confidence interval through random bootstrap
# Define bootstrap options
options = {"alpha": 0.95, "n_iter": 1000, "r0_ci": True}
# Call bootstrap
par_ci, par_list = SIR_UK.ci_bootstrap(**options)
```

[20]:
```
print("Confidence intervals of alpha, beta and R_0")
print(par_ci)
```

```
Confidence intervals of alpha, beta and R_0
[[0.55231672 0.56552158]
 [0.38       0.38      ]
 [1.45346506 1.48821468]]
```

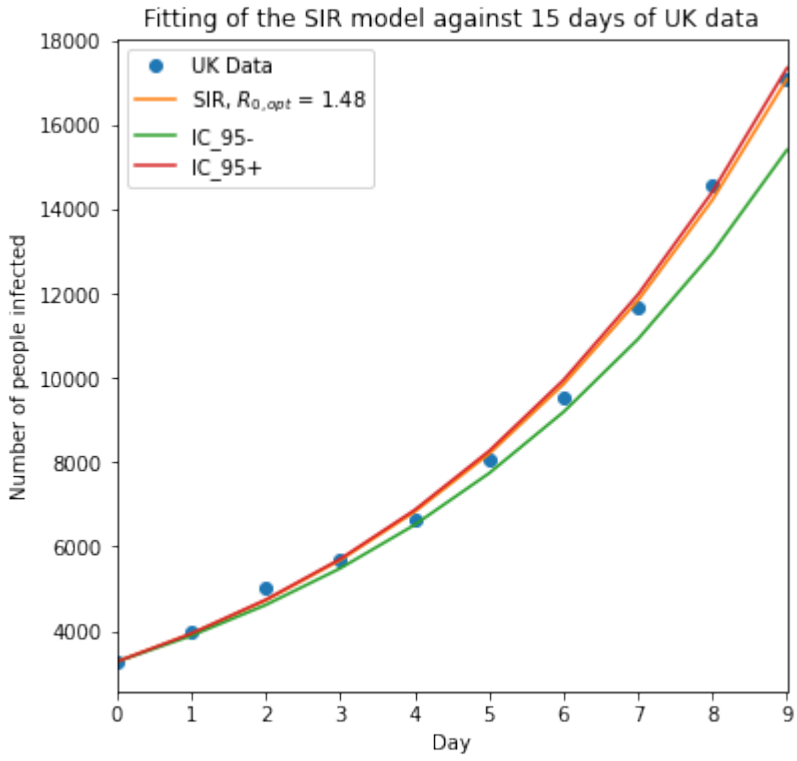
[21]:
```
alpha_min = par_ci[0][0]
alpha_max = par_ci[0][1]
# Explore the confidence intervals
print("IC 95% for alpha:", par_ci[0])
print("IC 95% for beta:", par_ci[1])
print("IC 95% for r0:", par_ci[2])
```

```
IC 95% for alpha: [0.55231672 0.56552158]
IC 95% for beta: [0.38 0.38]
IC 95% for r0: [1.45346506 1.48821468]
```

Visualization of confidence intervals

After 1000 of random sampling of the train data, it is possible to visualize the range of predictions produced within the 95% confidence intervals.

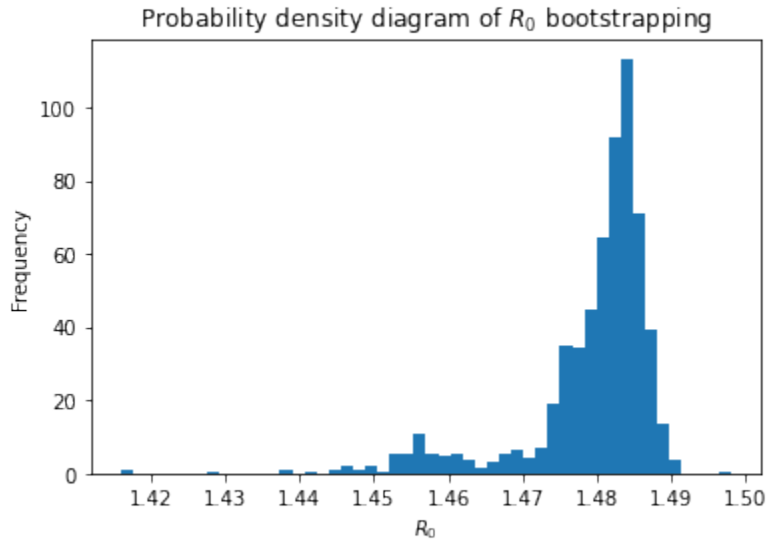
[22]:
```
# Build numerical solution
# I_opt = SIR_UK.solve(n_days-1, n_days).fetch()[:,2]
beta_0 = SIR_UK.p[1]
SIR_minus = SIR().set_params([alpha_min, beta_0], n0_UK)
SIR_plus = SIR().set_params([alpha_max, beta_0], n0_UK)
I_minus = SIR_minus.solve(n_days - 1, n_days).fetch()[:, 2]
I_plus = SIR_plus.solve(n_days - 1, n_days).fetch()[:, 2]

# lag = 6

R_opt = SIR_UK.r0  #

plt.figure(figsize=[6, 6])
plt.plot(t_d, I_UK, "o")
plt.plot(t_d, I_opt)
plt.plot(t_d, I_minus)
plt.plot(t_d, I_plus)
plt.legend(["UK Data", "SIR, $R_{0,opt}$ = %.2f" % R_opt, "IC_95-", "IC_95+"])
plt.title("Fitting of the SIR model against 15 days of UK data")
plt.ylabel("Number of people infected")
plt.xlabel("Day")
plt.xlim([min(t_d), max(t_d)])
plt.show()
```



An extremely asymmetrical confidence interval for R_0 using simple random bootstrap is observed. This occurs most likely because of neglecting the temporal structure of the exponential.

To further investigate this phenomena, we can observe the distribution of the R_0 parameter on the parameter list

[23]:
```
plt.hist(par_list[:, 0] / par_list[:, 1], bins=50, density=True, stacked=True)
plt.xlabel("$R_0$")
plt.ylabel("Frequency")
plt.title("Probability density diagram of $R_0$ bootstrapping")
plt.show()
```



It is interesting to observe that the spread is assymetrical towards lower R_0 values. This asymmetry is expected owing to the effect of lockdowns and public policies to promote social distancing. A strong assumption of the SIR model is that the spread rate α and removal rate β are constant, which is not the case in reality specially when strong public policies to limit the spread of a virus take place.

Evaluate model performance through block cross validation

A reliable approach to evaluate the predictive accuracy of a model which variables are time-dependent is to use block cross validation. In Open-SIR, it is implemented through the `model.block_cv` function. The inputs of the model is the minimum sample to use to perform the cross validation. The outputs of the model are lists with the average mean squared error, rolling mean squared error, evolution of the fitted parameters and a `PredictionResults` dataclass.

[24]:
```
# We previously imported ci_block_cv which provides a better prediction of the mean␣
↪squared error of the predictions
n_lags = 1
MSE_avg, MSE_list, p_list, pred_data = SIR_UK.block_cv(lags=n_lags, min_sample=3)
```

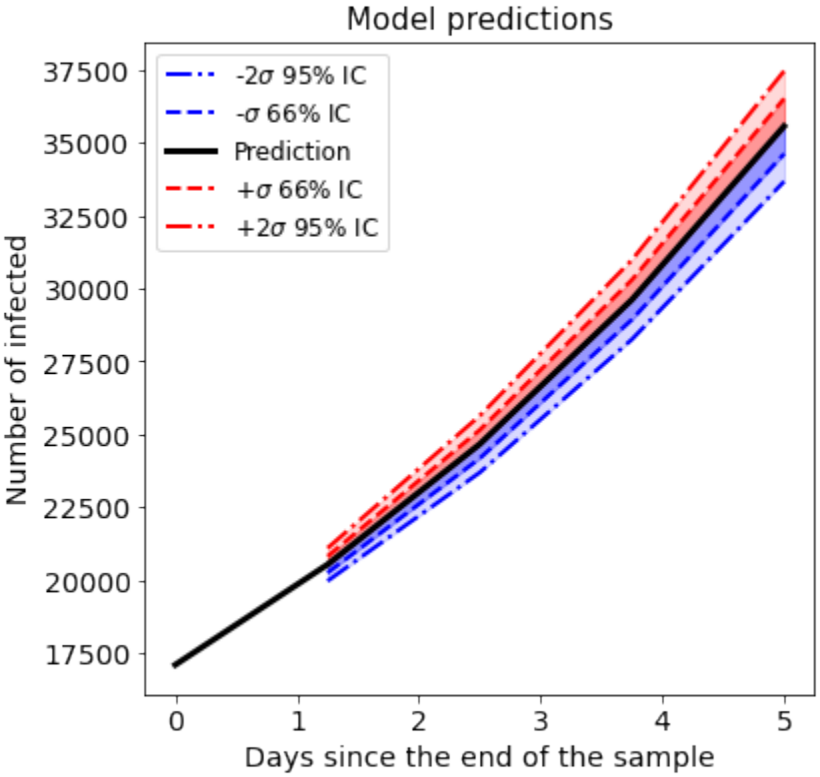
This `pred_data` instance of the `PredictionResults` dataclass offers a simplified syntax to access details of the cross-validation test on model predictions. For instance, the member function `.print_mse()` prints a summary of the mean-squared errors for different forecasts horizons. The number of forecasts horizons provided is by default the length of the observations minus the min_sample parameter.

[25]:
```
pred_data.print_mse()
```

```
Average MSE for 0-day predictions = 281.77, MSE sample size = 7
Average MSE for 1-day predictions = 486.87, MSE sample size = 6
Average MSE for 2-day predictions = 681.41, MSE sample size = 5
Average MSE for 3-day predictions = 947.02, MSE sample size = 4
Average MSE for 4-day predictions = 1181.14, MSE sample size = 3
Average MSE for 5-day predictions = 2247.42, MSE sample size = 2
Average MSE for 6-day predictions = 4896.45, MSE sample size = 1
```

If the residuals between the model and the observed data are normally distributed , the mean squared error is an estimator of the error variance. The member function `.plot_predictions(n_days)` offers a convenient ways to visualize short term predictions with an estimations of the 66% and 95% confidence intervals.

[26]:
```
pred_data.plot_pred_ci(4)
```

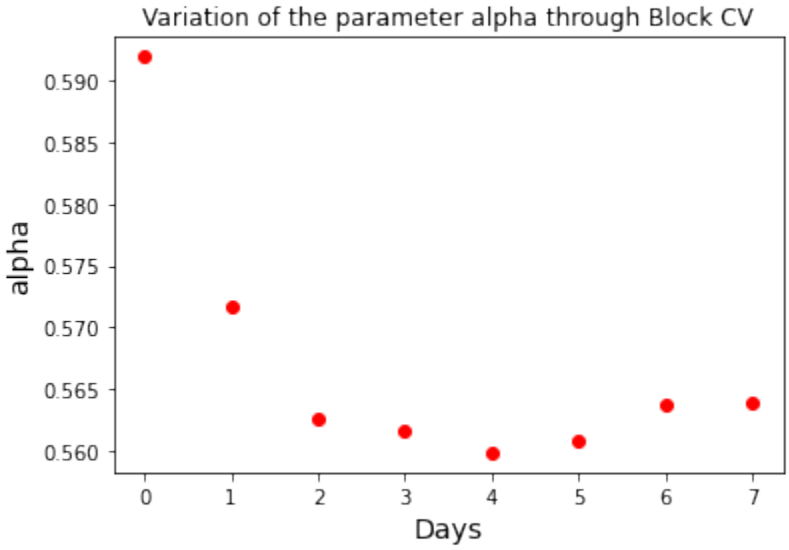


The robustness of the model fitting can be explored plotting the change of the parameters during the block cross validation. As the model is fitted with more and more recent data, a measure of robustness is the convergence of the model parameters to a particular value. The list `p_list` contains the information of all model parameters, which can be plotted to assess parameter convergence.

[27]:
```
print("Block cross validation parametric range")
plt.plot(p_list[:, 0], "ro")
plt.title("Variation of the parameter alpha through Block CV")
plt.xlabel("Days", size=14)
plt.ylabel("alpha", size=14)
```

```
Block cross validation parametric range
```

[27]:
```
Text(0, 0.5, 'alpha')
```



It is clear that the α parameter is converging to a value between 0.56 and 0.57 as time progresses. This results have to be reassessed as new data appears, as some fundamental change in the disease epidemiology or social distance may occur suddenly.

The `MSE_avg` list contains the mean squared errors for a forecast of the day $i + 1$ since the fitting data ends. For example, the average mean squared error for one day predictions can be accessed on `MSE_avg[0]`

[28]:
```
print(
    """
    The average mean squared error on the time
    block cross validation is: %.3f"""
    % MSE_avg[0]
)
```

```
    The average mean squared error on the time
    block cross validation is: 281.774
```

Another way to visualize the results of the block cross-validation, is to observe the variation of the reproduction rate R_0 and the mean squared error when a subset of the days is taken. By default, `block_cv` starts with the data of three days, fit the model on that data, predicts the number of infected in the next period, calculate the mean squared error between the prediction and the test dataset, and stores it into two arrays. Afterwards, it computes the MSE of 2,3 and up to `len(n_obs)-min_sample` days to be forecasted. It repeats this until it uses the data of $(n - 1)$ intervals to predict the $n - th$ latest observation of infections.

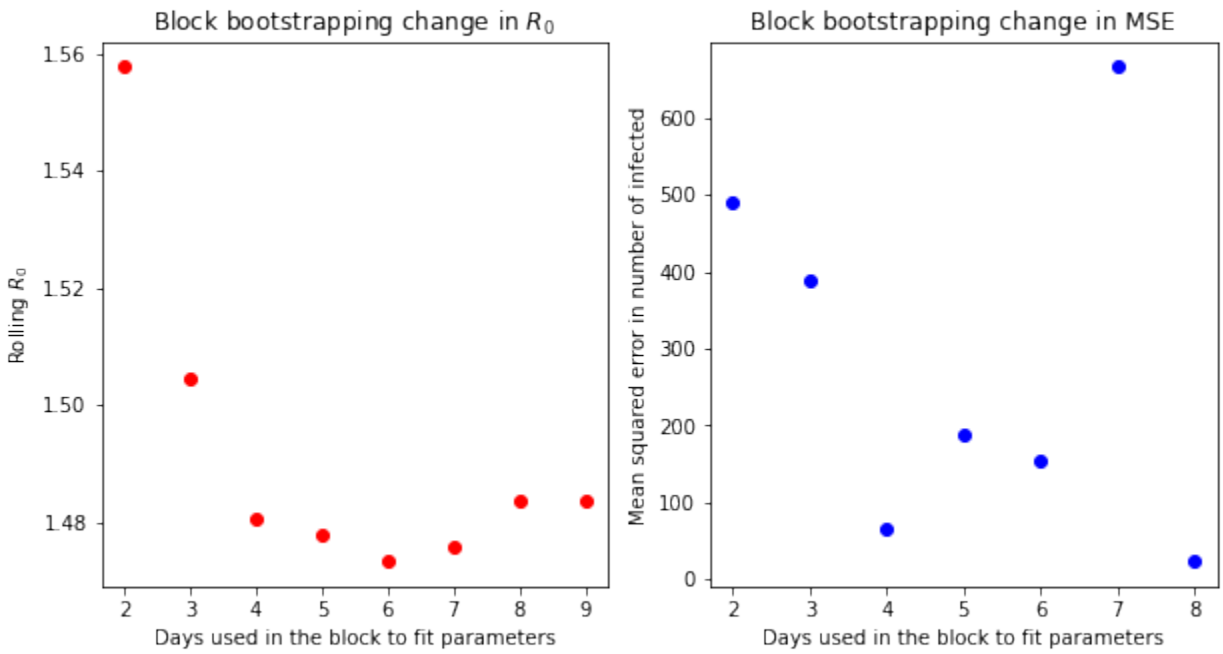
In the next example, the list of the mean-squared errors for 1-day prediction is visualized

[29]:
```
r0_roll = p_list[:, 0] / p_list[:, 1]

plt.figure(figsize=[10, 5])
plt.subplot(1, 2, 1)
plt.plot(t_d[2:], r0_roll, "ro")
plt.xlabel("Days used in the block to fit parameters")
plt.ylabel("Rolling $R_0$")
plt.title("Block bootstrapping change in $R_0$")
plt.subplot(1, 2, 2)
plt.plot(t_d[(2):-1], MSE_list[0], "bo")
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("Days used in the block to fit parameters")
plt.ylabel("Mean squared error in number of infected")
plt.title("Block bootstrapping change in MSE")
plt.show()
```



Interestingly, it is hard to see any convergence on the change on mean-squared error.

[30]:
```
print("MSE_lastday = %.2f" % MSE_list[0][-1])
```

```
MSE_lastday = 22.76
```

Note that the mean squared error is provided in absolute terms. However, the number of infected cases is increasing with time.

The percentage error of the predictions can be calculated directly from the mean squared error.

$$\epsilon(t) = \frac{\mathrm{MSE}(t)}{I(t)}$$

For example, in the last entry of I_UK, they were 17089 infected, while the MSE was 666.45. Then, the percentage deviation would be:

$$\epsilon(10) = \frac{22.76}{17089} = 0.13\%$$

However, this takes model prediction over the accumulated number of cases. Another way to quantify the deviation of the model predictions is to calculate the percentage error over the new infected cases:

$$\epsilon_{\mathrm{new}}(t) = \frac{\mathrm{MSE}(t)}{I(t) - I(t-1)}$$

If this metric is used, the error naturally will be higher.

[31]:
```
e_new = MSE_list[0][-1] / (I_UK[-1] - I_UK[-2])
print(
    "The percentage error of the SIR model over the last day reported cases is %.1f%%"
    % (100 * e_new)
)
```

```
The percentage error of the SIR model over the last day reported cases is 0.9%
```

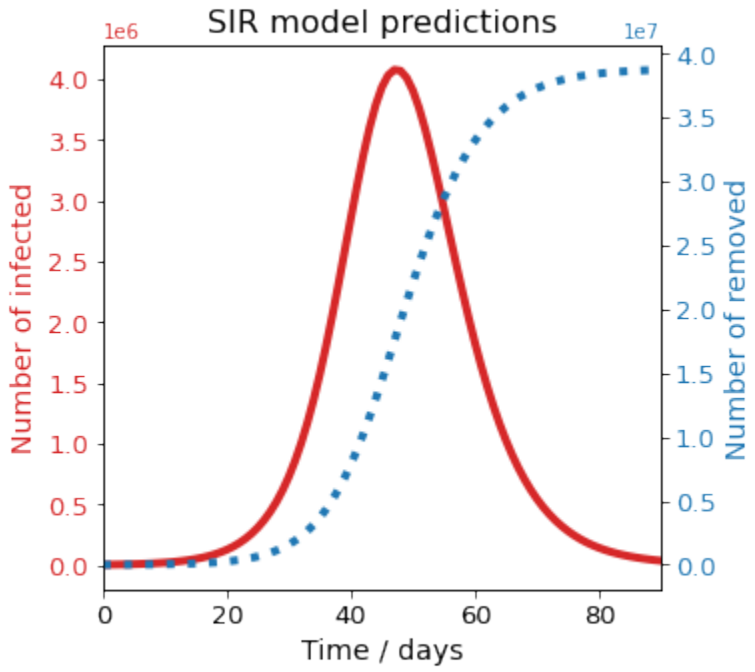
It must be noted that in the last day studied, the error is extremely low owing to an exceptionally good agreement on the last point of the data. Hence, a better estimate of the error in the predictions is to take the average percentage error on the cross validation subset, which considers from day 3 onwards.

[32]:
```
eps_avg = np.mean(MSE_list[0] / I_UK[-7:]) * 100
print("The average percentage deviation on the number of infected is %.1f%%" % eps_
↪avg)
```

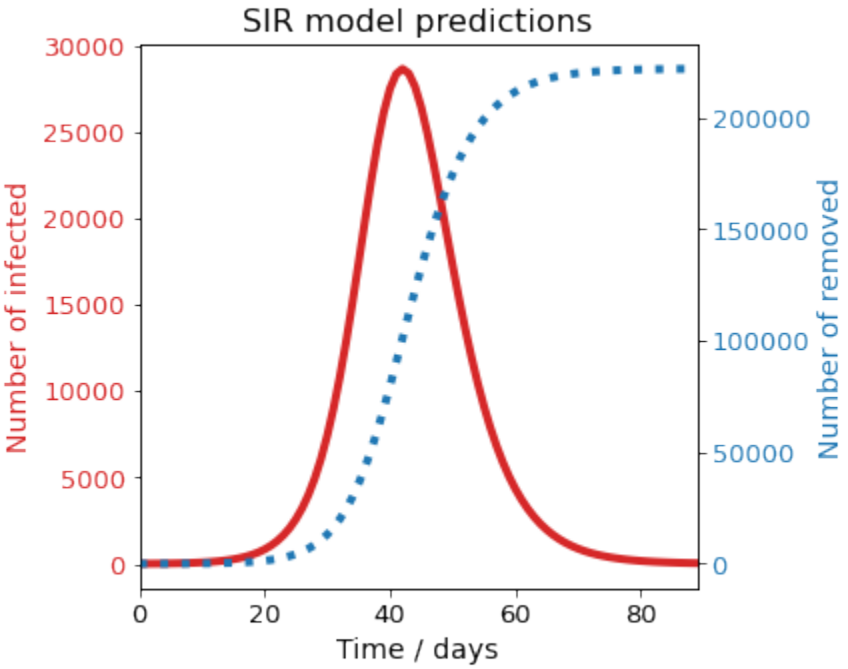
```
The average percentage deviation on the number of infected is 3.3%
```

Visualize long term predictions

[33]:
```
long_time = 90
SIR_UK.solve(long_time, long_time + 1)
SIR_UK.plot()
```



[34]:
```
my_SIR_fitted.plot()
```



1.3.2 SIR-X

This notebook exemplifies how Open-SIR can be used to fit the SIR-X model by Maier and Dirk (2020) to existing data and make predictions. The SIR-X model is a standard generalization of the Susceptible-Infectious-Removed (SIR) model, which includes the influence of exogenous factors such as policy changes, lockdown of the whole population and quarantine of the infectious individuals.

The Open-SIR implementation of the SIR-X model will be validated reproducing the parameter fitting published in the supplementary material of the original article published by Maier and Brockmann (2020). For simplicity, the validation will be performed only for the city of Guangdong, China.

Import modules

[1]:
```
# Uncomment this cell to activate black code formatter in the notebook
# %load_ext nb_black
```

[2]:
```
# Import packages
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

Data sourcing

We will source data from the repository of the [John Hopkins University COVID-19 dashboard] (https://coronavirus.jhu.edu/map.html) published formally as a correspondence in The Lancet. This time series data contains the number of reported cases $C(t)$ per day for a number of cities.

[3]:
```
# Source data from John Hokpins university reposotiry
# jhu_link = "https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/who_
↪covid_19_situation_reports/who_covid_19_sit_rep_time_series/who_covid_19_sit_rep_
↪time_series.csv"
jhu_link = "https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/csse_
↪covid_19_data/csse_covid_19_time_series/time_series_covid19_confirmed_global.csv"
jhu_df = pd.read_csv(jhu_link)
# Explore the dataset
jhu_df.head(10)
```

[3]:

	Province/State	Country/Region	Lat	Long	1/22/20	1/23/20	1/24/20	1/25/20	1/26/20	1/27/20	...	7/2/20	7/3/20	7/4/20	7/5/20	7/6/20	7/7/20	7/8/20	7/9/20	7/10/20	7/11/20
0	NaN	Afghanistan	33.0000	65.0000	0	0	0	0	0	0	...	32022	32324	32672	32951	33190	33384	33594	33908	34194	34366
1	NaN	Albania	41.1533	20.1683	0	0	0	0	0	0	...	2662	2752	2819	2893	2964	3038	3106	3188	3278	3371
2	NaN	Algeria	28.0339	1.6596	0	0	0	0	0	0	...	14657	15070	15500	15941	16404	16879	17348	17808	18242	18712
3	NaN	Andorra	42.5063	1.5218	0	0	0	0	0	0	...	855	855	855	855	855	855	855	855	855	855
4	NaN	Angola	-11.2027	17.8739	0	0	0	0	0	0	...	315	328	346	346	346	386	386	396	458	462
5	NaN	Antigua and Barbuda	17.0608	-61.7964	0	0	0	0	0	0	...	69	68	68	68	70	70	70	73	74	74
6	NaN	Argentina	-38.4161	-63.6167	0	0	0	0	0	0	...	69941	72786	75376	77815	80447	83426	87030	90693	94060	97509
7	NaN	Armenia	40.0691	45.0382	0	0	0	0	0	0	...	26658	27320	27900	28606	28936	29285	29820	30346	30903	31392
8	Australian Capital Territory	Australia	-35.4735	149.0124	0	0	0	0	0	0	...	108	108	108	108	108	111	112	113	113	113
9	New South Wales	Australia	-33.8688	151.2093	0	0	0	0	3	4	...	3211	3405	3419	3429	3433	3440	3453	3467	3474	3478

[10 rows x 176 columns]

It is observed that the column "Province/States" contains the name of the cities, and since the forth column a time series stamp (or index) is provided to record daily data of reported cases. Additionally, there are many days without recorded data for a number of chinese cities. This won't be an issue for parameter fitting as Open-SIR doesn't require uniform spacement of the observed data.

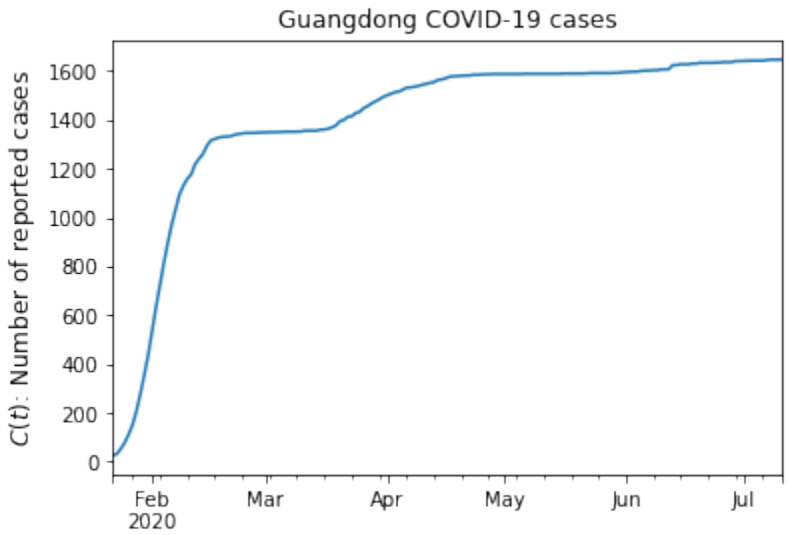
Data preparation

In the following lines, the time series for Guangdong reported cases $C(t)$ is extracted from the original dataframe. Thereafter, the columns are converted to a pandas date time index in order to perform further data preparation steps.

[4]:
```
China = jhu_df[jhu_df[jhu_df.columns[1]] == "China"]
city_name = "Guangdong"
city = China[China["Province/State"] == city_name]
city = city.drop(columns=["Province/State", "Country/Region", "Lat", "Long"])
time_index = pd.to_datetime(city.columns)
data = city.values
# Visualize the time
ts = pd.Series(data=city.values[0], index=time_index)
```

Using the function ts.plot() a quick visualization of the dataset is obtained:

[5]:
```
ts.plot()
plt.title("Guangdong COVID-19 cases")
plt.ylabel("$C(t)$: Number of reported cases", size=12)
plt.show()
```



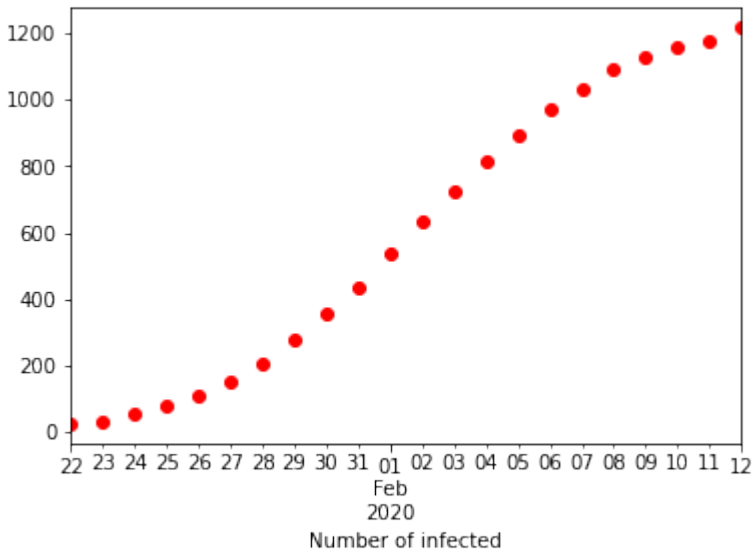
Data cleaning

[6]:
```
ts_clean = ts.dropna()
# Extract data
ts_fit = ts_clean["2020-01-21":"2020-02-12"]
# Convert index to numeric
ts_num = pd.to_numeric(ts_fit.index)
t0 = ts_num[0]
# Convert datetime to days
t_days = (ts_num - t0) / (10 ** 9 * 86400)
t_days = t_days.astype(int).values
# t_days is an input for SIR
```

[7]:
```
# Define the X number
nX = ts_fit.values  # Number of infected
N = 104.3e6  # Population size of Guangdong
```

Exploration of the dataset

[8]:
```
ts_fit.plot(style="ro")
plt.xlabel("Number of infected")
plt.show()
```



[ ]:
```
```

Setting up SIR and SIR-X models

The population N of the city is a necessary input for the model. In this notebook, this was hardocded, but it can be sourced directly from a web source.

Note that whilst the SIR model estimates directly the number of infected people, $NI(t)$, SIR-X estimates the number of infected people based on the number of tested cases that are in quarantine or in an hospital $NX(t)$

[9]:
```
# These lines are required only if opensir wasn't installed using pip install, or if␣
↪opensir is running in the pipenv virtual environment
import sys

path_opensir = "../../"
sys.path.append(path_opensir)

# Import SIR and SIRX models
from opensir.models import SIR, SIRX

nX = ts_fit.values  # Number of observed infections of the time series
N = 104.3e6  # Population size of Guangdong
params = [0.95, 0.38]
w0 = (N - nX[0], nX[0], 0)

G_sir = SIR()
G_sir.set_params(p=params, initial_conds=w0)
G_sir.fit_input = 2
G_sir.fit(t_days, nX)
G_sir.solve(t_days[-1], t_days[-1] + 1)
```

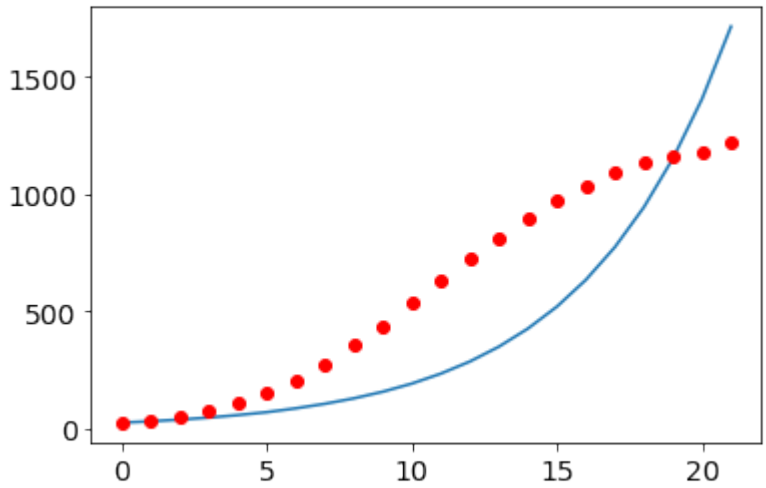
(continues on next page)

(continued from previous page)

```
t_SIR = G_sir.fetch()[:, 0]
I_SIR = G_sir.fetch()[:, 2]
```

Try to fit a SIR model to Guangdong data

[10]:
```
ax = plt.axes()
ax.tick_params(axis="both", which="major", labelsize=14)
plt.plot(t_SIR, I_SIR)
plt.plot(t_days, nX, "ro")
plt.show()
```



The SIR model is clearly not appropriate to fit this data, as it cannot resolve the effect of exogeneous containment efforts such as quarantines or lockdown. We will repeat the process with a SIR-X model.

Fit SIR-X to Guangdong Data

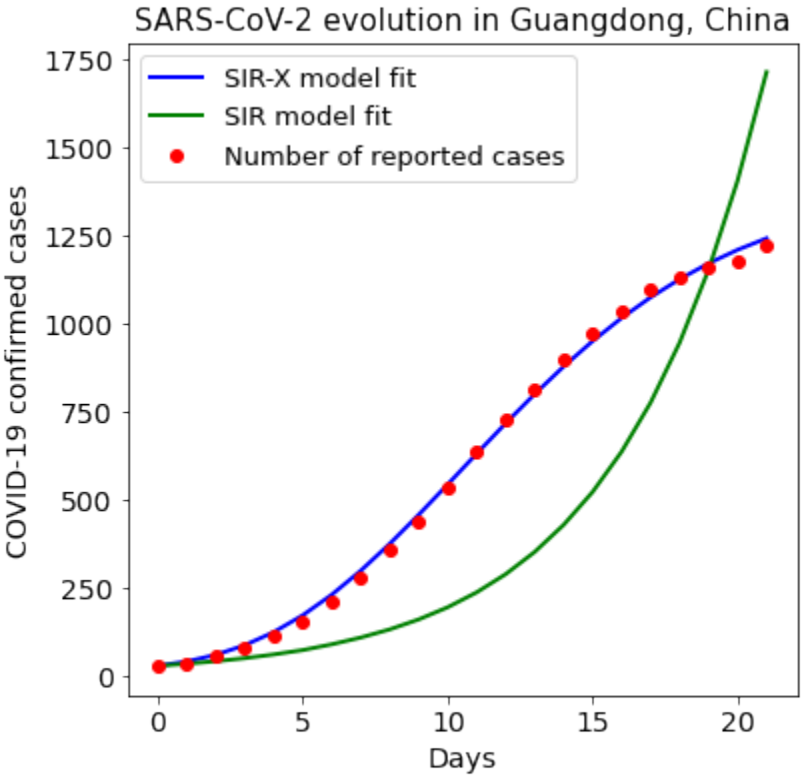
[11]:
```
g_sirx = SIRX()
params = [6.2 / 8, 1 / 8, 0.05, 0.05, 5]
# X_0 can be directly ontained from the statistics
n_x0 = nX[0]  # Number of people tested positive
n_i0 = nX[0]

w0 = (N - n_x0 - n_i0, n_i0, 0, n_x0)
g_sirx.set_params(p=params, initial_conds=w0)
# Fit all parameters
fit_index = [False, False, True, True, True]
g_sirx.fit(t_days, nX, fit_index=fit_index)
g_sirx.solve(t_days[-1], t_days[-1] + 1)
t_sirx = g_sirx.fetch()[:, 0]
inf_sirx = g_sirx.fetch()[:, 4]
```

[12]:
```
plt.figure(figsize=[6, 6])
ax = plt.axes()
plt.plot(t_sirx, inf_sirx, "b-", linewidth=2)
```

(continues on next page)

(continued from previous page)

```
plt.plot(t_SIR, I_SIR, "g-", linewidth=2)
plt.plot(t_days, nX, "ro")
plt.legend(
    ["SIR-X model fit", "SIR model fit", "Number of reported cases"], fontsize=13
)
plt.title("SARS-CoV-2 evolution in Guangdong, China", size=15)
plt.xlabel("Days", fontsize=14)
plt.ylabel("COVID-19 confirmed cases", fontsize=14)
ax.tick_params(axis="both", which="major", labelsize=14)
plt.show()
```



After fitting the parameters, the effective infectious period $T_{I,eff}$ and the effective reproduction rate $R_{0,eff}$ can be obtained from the model properties

$$T_{I,eff} = (\beta + \kappa + \kappa_0)^{-1}$$

$$R_{0,eff} = \alpha T_{I,eff}$$

Aditionally, the Public containment leverage P and the quarantine probability Q can be calculated through:

$$P = \frac{\kappa_0}{\kappa_0 + \kappa}$$

$$Q = \frac{\kappa_0 + \kappa}{\beta + \kappa_0 + \kappa}$$

[13]:
```
print("Effective infectious period T_I_eff =  %.2f days " % g_sirx.t_inf_eff)
print(
```

(continues on next page)

(continued from previous page)

```
    "Effective reproduction rate R_0_eff =  %.2f, Maier and Brockmann = %.2f"
    % (g_sirx.r0_eff, 3.02)
)
print(
    "Public containment leverage =  %.2f, Maier and Brockmann = %.2f"
    % (g_sirx.pcl, 0.75)
)
print(
    "Quarantine probability =  %.2f, Maier and Brockmann = %.2f" % (g_sirx.q_prob, 0.
↪51)
)
```

```
Effective infectious period T_I_eff =  3.89 days
Effective reproduction rate R_0_eff =  3.01, Maier and Brockmann = 3.02
Public containment leverage =  0.80, Maier and Brockmann = 0.75
Quarantine probability =  0.51, Maier and Brockmann = 0.51
```

Make predictions using `model.predict`

[14]:
```
# Make predictions and visualize
# Obtain the results 14 days after the train data ends
sirx_pred = g_sirx.predict(14)
print("T n_S \t   n_I \tn_R \tn_X")
for i in sirx_pred:
    print(*i.astype(int))
```

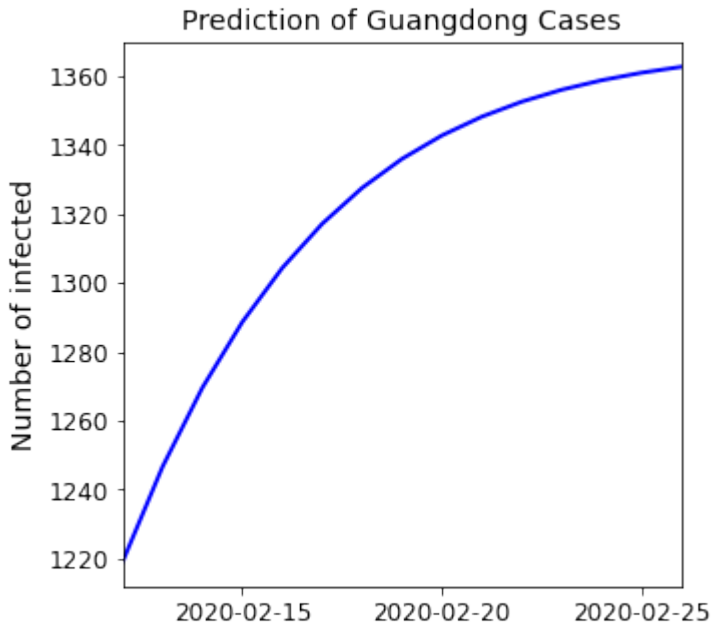
```
T n_S     n_I         n_R    n_X
0 11450984 227 92847569 1219
1 10307571 190 93990991 1246
2 9278326 158 95020245 1269
3 8351856 130 95946723 1288
4 7517894 107 96780694 1304
5 6767209 87 97531385 1317
6 6091483 70 98207117 1327
7 5483230 57 98815376 1336
8 4935707 45 99362903 1342
9 4442871 36 99855743 1348
10 3999251 29 100299366 1352
11 3599916 23 100698704 1356
12 3240456 18 101058166 1358
13 2916888 14 101381735 1361
14 2625630 11 101672995 1362
```

Prepare date time index to plot predictions

[15]:
```
# Import datetime module from the standard library
import datetime

# Obtain the last day from the data used to train the model
last_time = ts_fit.index[-1]
# Create a date time range based on the number of rows of the prediction
numdays = sirx_pred.shape[0]
day_zero = datetime.datetime(last_time.year, last_time.month, last_time.day)
date_list = [day_zero + datetime.timedelta(days=x) for x in range(numdays)]
```

Plot predictions

[16]:
```
# Extract figure and axes
fig, ax = plt.subplots(figsize=[5, 5])
# Create core plot attributes
plt.plot(date_list, sirx_pred[:, 4], color="blue", linewidth=2)
plt.title("Prediction of Guangdong Cases", size=14)
plt.ylabel("Number of infected", size=14)
# Remove trailing space
plt.xlim(date_list[0], date_list[-1])
# Limit the amount of data displayed
ax.xaxis.set_major_locator(plt.MaxNLocator(3))
# Increase the size of the ticks
ax.tick_params(labelsize=12)
plt.show()
```



Calculation of predictive confidence intervals

The confidence intervals on the predictions of the SIR-X model can be calculated using a block cross validation. This technique is widely used in Time Series Analysis. In the open-sir API, the function `model.ci_block_cv` calculates the average mean squared error of the predictions, a list of the rolling mean squared errors and the list of parameters which shows how much each parameter changes taking different number of days for making predictions.

The three first parameters are the same as the fit function, while the last two parameters are the `lags` and the `min_sample`. The `lags` parameter indicates how many periods in the future will be forecasted in order to calculate the mean squared error of the model prediction. The `min_sample` parameter indicates the initial number of observations and days that will be taken to perform the block cross validation.

In the following example, `model.ci_block_cv` is used to estimate the average mean squared error of *1-day* predictions taking *6* observations as the starting point of the cross validation. For Guangdong, a `min_sample=6` higher than the default 3 is required to handle well the missing data. This way, both the data on the four first days, and two days after the data starts again, are considered for cross validation.

[17]:
```
# Calculate confidence intervals
mse_avg, mse_list, p_list, pred_data = g_sirx.block_cv(lags=1, min_sample=6)
```

If it is assumed that the residuals distribute normally, then a good estimation of a 95% confidence interval on the one-day prediction of the number of confirmed cases is

$$\sigma \sim \text{MSE} \rightarrow n_{X,t+1} \sim \hat{n}_{X,t+1} \pm 2\sigma$$

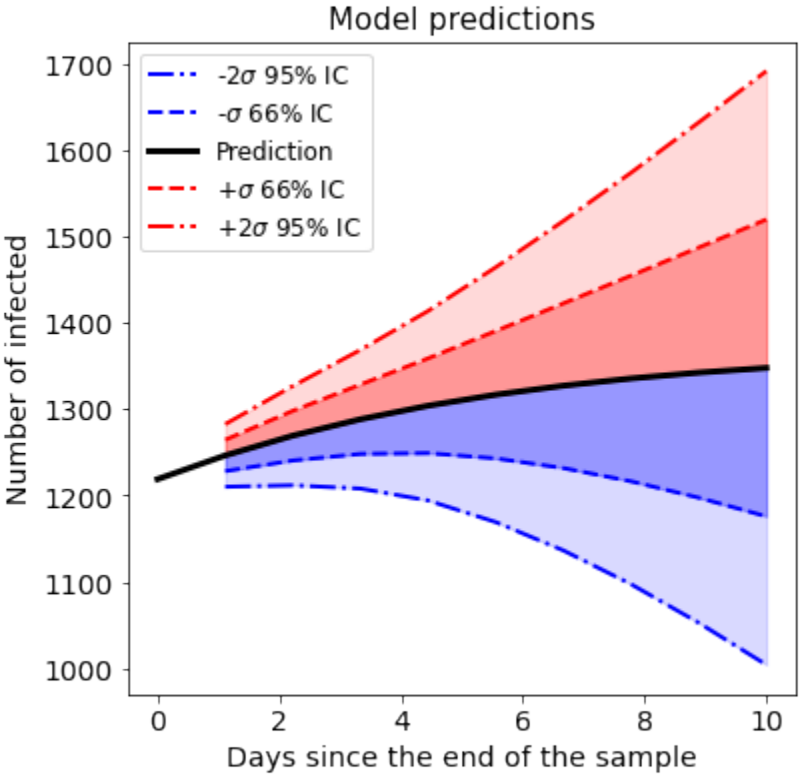
Where $n_{X,t+1}$ is the real number of confirmed cases in the next day, and $\hat{n}_{X,t+1}$ is the estimation using the SIR-X model using cross validation. We can use the `PredictionResults` instance `pred_data` functionality to explore the mean-squared errors and the predictions confidence intervals:

[18]:
```
pred_data.print_mse()
```

```
Average MSE for 0-day predictions = 18.16, MSE sample size = 16
Average MSE for 1-day predictions = 28.81, MSE sample size = 15
Average MSE for 2-day predictions = 40.23, MSE sample size = 14
Average MSE for 3-day predictions = 54.97, MSE sample size = 13
Average MSE for 4-day predictions = 73.92, MSE sample size = 12
Average MSE for 5-day predictions = 95.56, MSE sample size = 11
Average MSE for 6-day predictions = 119.34, MSE sample size = 10
Average MSE for 7-day predictions = 145.10, MSE sample size = 9
Average MSE for 8-day predictions = 172.19, MSE sample size = 8
Average MSE for 9-day predictions = 198.89, MSE sample size = 7
Average MSE for 10-day predictions = 223.80, MSE sample size = 6
Average MSE for 11-day predictions = 245.12, MSE sample size = 5
Average MSE for 12-day predictions = 283.18, MSE sample size = 4
Average MSE for 13-day predictions = 266.35, MSE sample size = 3
Average MSE for 14-day predictions = 92.48, MSE sample size = 2
Average MSE for 15-day predictions = 196.84, MSE sample size = 1
```

The predictive accuracy of the model is quite impressive, even for 9-day predictions. Let's take advantage of the relatively low mean squared error to forecast a 10 days horizon with confidence intervals using `pred_data.plot_predictions(n_days=9)`

[19]:
```
pred_data.plot_pred_ci(n_days=9)
```



If it is assumed that the residuals distribute normally, then a good estimation of a 95% confidence interval on the one-day prediction of the number of confirmed cases is

$$\sigma \sim \text{MSE} \rightarrow n_{X,t+1} \sim \hat{n}_{X,t+1} \pm 2\sigma$$

Where $n_{X,t+1}$ is the real number of confirmed cases in the next day, and $\hat{n}_{X,t+1}$ is the estimation using the SIR-X model using cross validation. We use solve to make a 1-day prediction and append the 95% confidence interval.

[20]:
```
# Predict
g_sirx.solve(t_days[-1] + 1, t_days[-1] + 2)
n_X_tplusone = g_sirx.fetch()[-1, 4]
print("Estimation of n_X_{t+1} = %.0f +- %.0f " % (n_X_tplusone, 2 * mse_avg[0]))
```

```
Estimation of n_X_{t+1} = 1268 +- 36
```

[21]:
```
# Transform parameter list into a DataFrame
par_block_cv = pd.DataFrame(p_list)
# Rename dataframe columns based on SIR-X parameter names
par_block_cv.columns = g_sirx.PARAMS
# Add the day. Note that we take the days from min_sample until the end of the array,␣
↪as days
# 0,1,2 are used for the first sampling in the block cross-validation
par_block_cv["Day"] = t_days[5:]
# Explore formatted dataframe for parametric analysis
par_block_cv.head(len(p_list))
```

[21]:

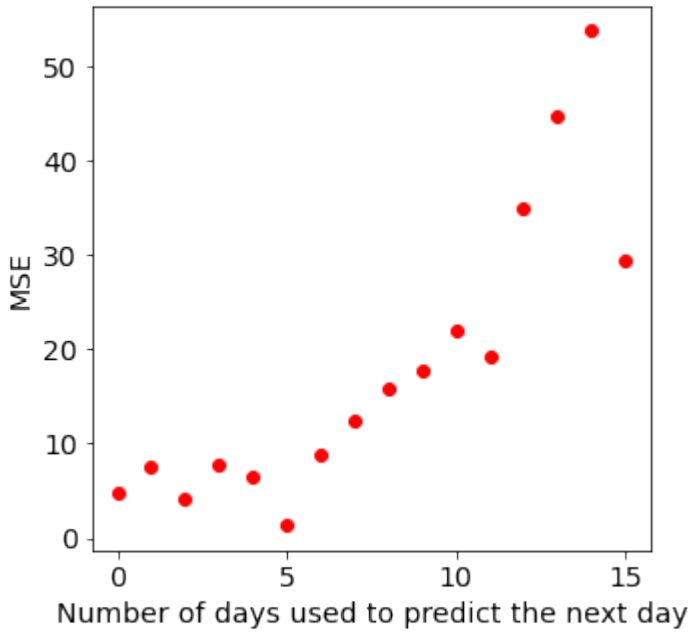
	alpha	beta	kappa_0	kappa	inf_over_test	Day
0	0.775	0.125	0.115899	1.020877e-10	2.705780	5
1	0.775	0.125	0.109540	4.023231e-13	2.774667	6
2	0.775	0.125	0.068467	1.099828e-01	1.818714	7

(continues on next page)

(continued from previous page)

```
3   0.775  0.125  0.082939  6.627930e-02       2.100077    8
4   0.775  0.125  0.104995  6.901938e-11       2.822296    9
5   0.775  0.125  0.090598  4.458775e-02       2.285169   10
6   0.775  0.125  0.092793  3.733649e-02       2.355333   11
7   0.775  0.125  0.104239  1.903129e-09       2.824648   12
8   0.775  0.125  0.105289  3.024668e-09       2.843301   13
9   0.775  0.125  0.106338  7.624121e-11       2.866025   14
10  0.775  0.125  0.107283  1.264505e-10       2.889912   15
11  0.775  0.125  0.108250  3.590963e-14       2.917548   16
12  0.775  0.125  0.108956  1.935729e-15       2.939909   17
13  0.775  0.125  0.110043  8.197801e-11       2.977366   18
14  0.775  0.125  0.109884  4.608366e-03       2.954701   19
15  0.775  0.125  0.106441  2.069015e-02       2.797521   20
16  0.775  0.125  0.104914  2.805141e-02       2.739572   21
```

[22]:
```
plt.figure(figsize=[5, 5])
ax = plt.axes()
ax.tick_params(axis="both", which="major", labelsize=14)
plt.plot(mse_list[0], "ro")
plt.xlabel("Number of days used to predict the next day", size=14)
plt.ylabel("MSE", size=14)
plt.show()
```



There is an outlier on day 1, as this is when the missing date starts. A more reliable approach would be to take the last 8 values of the mean squared error to calculate a new average assuming that there will be no more missing data.

Variation of fitted parameters

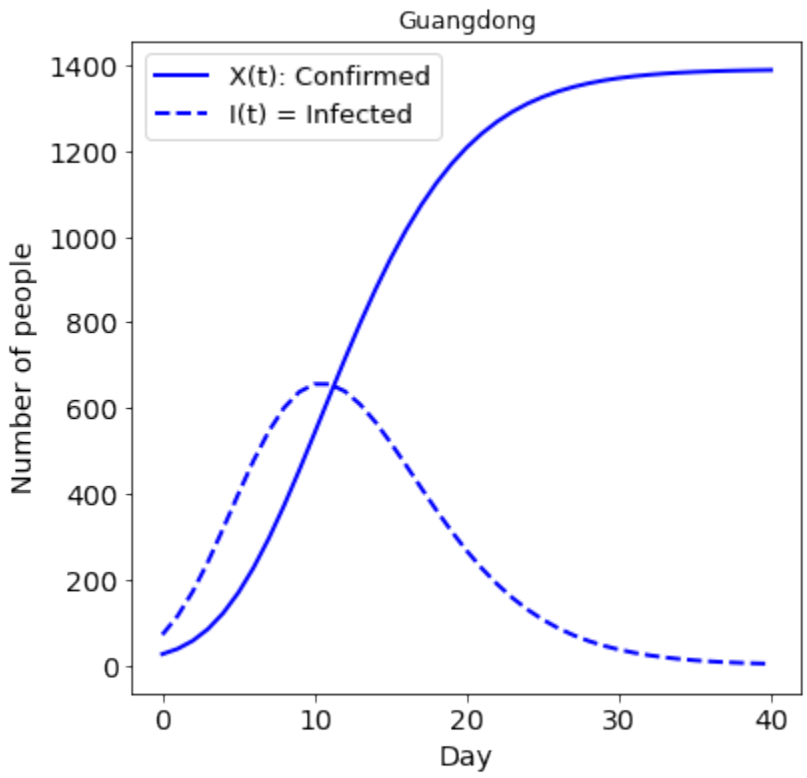
Finally, it is possible to observe how the model parameters change as more days and number of confirmed cases are introduced in the block cross validation.

It is clear to observe that after day 15 all parameters except kappa begin to converge. Therefore, care must be taken when performing inference over the parameter kappa.

Long term prediction

Now we can use the model to predict when the peak will occur and what will be the maximum number of infected

[23]:
```
# Predict
plt.figure(figsize=[6, 6])
ax = plt.axes()
ax.tick_params(axis="both", which="major", labelsize=14)
g_sirx.solve(40, 41)
# Plot
plt.plot(g_sirx.fetch()[:, 4], "b-", linewidth=2)  # X(t)
plt.plot(g_sirx.fetch()[:, 2], "b--", linewidth=2)  # I(t)
plt.xlabel("Day", size=14)
plt.ylabel("Number of people", size=14)
plt.legend(["X(t): Confirmed", "I(t) = Infected"], fontsize=13)
plt.title(city_name)
plt.show()
```



The model was trained with a limited amount of data. It is clear to observe that since the measures took place in Guangdong, at least 6 weeks of quarantine were necessary to control the pandemics. Note that a limitation of this model is that it predicts an equilibrium where the number of infected, denoted by the yellow line in the figure above,

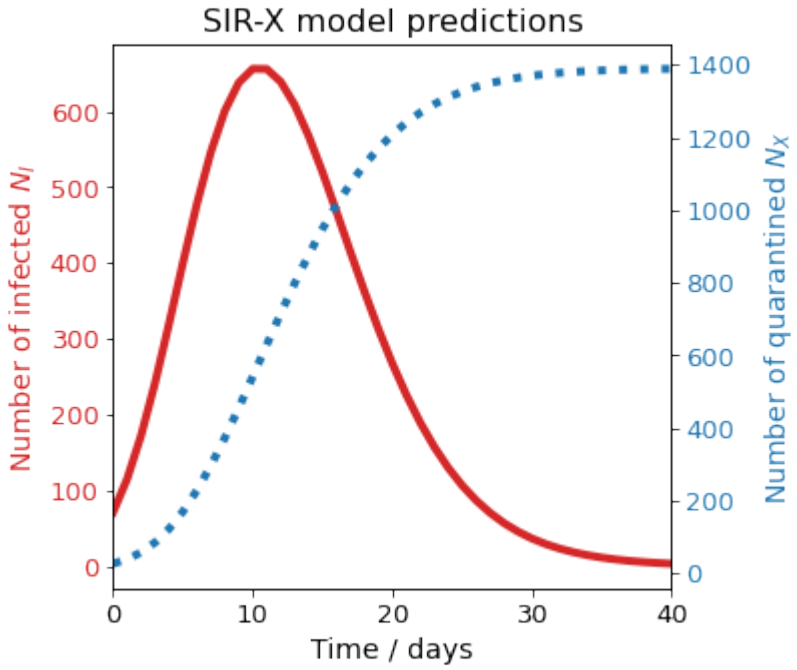
is 0 after a short time. In reality, this amount will decrease to a small number.

After the peak of infections is reached, it is necessary to keep the quarantine and effective contact tracing for at least 30 days more.

Validate long term plot using model.plot()

The function `model.plot()` offers a handy way to visualize model fitting and predictions. Custom visualizations can be validated against the `model.plot()` function.

[24]:
```
g_sirx.plot()
```



CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX