# Bayesian inference of COVID-19

***Release 0.2.0***

**Jonas Dehning, Johannes Zierenberg, F. Paul Spitzner, Michael W**

**Mar 11, 2021**

# CONTENTS

CHAPTER

ONE

# INSTALLATION

There exists three different possiblities to run the models:

1. Clone the repository, with the latest release:

```
git clone --branch v0.1.8 https://github.com/Priesemann-Group/covid19_inference
```

2. Install the module via pip

```
pip install git+https://github.com/Priesemann-Group/covid19_inference.git@v0.1.8
```

3. Run the notebooks directly in Google Colab. At the top of the notebooks files there should be a symbol which opens them directly in a Google Colab instance.

CHAPTER

TWO

# FIRST STEPS

To get started, we recommend to look at one of the currently two example notebooks:

1. **SIR model with one german state** This model is similar to the one discussed in our paper: Inferring COVID-19 spreading rates and potential change points for case number forecasts. The difference is that the delay between infection and report is now lognormal distributed and not fixed.
2. **Hierarchical model of the German states** This builds a hierarchical bayesian model of the states of Germany. Caution, seems to be currently broken!

We can for example recommend the following articles about bayesian modeling:

As a introduction to Bayesian statistics and the python package (PyMC3) that we use: https://docs.pymc.io/notebooks/api_quickstart.html

This is a good post about hierarchical Bayesian models in general: https://statmodeling.stat.columbia.edu/2014/01/21/everything-need-know-bayesian-statistics-learned-eight-schools/

CHAPTER

THREE

# EXAMPLES

We supply a number of examples which can be found in the scripts folder of the github repository.

These examples are given as python files and interactive ipython notebooks. The python files get automatically converted into ipython notebooks for easier use with google colab. The conversion is done by a slightly modified version of the python2jupyter module, which can be found here.

For starters the most useful examples are the non hierarchical one bundesland example and the hierarchical analysis of the bundeslaender.

CHAPTER

FOUR

# DISCLAIMER

We evaluate the data provided by the John Hopkins University link. We exclude any liability with regard to the quality and accuracy of the data used, and also with regard to the correctness of the statistical analysis. The evaluation of the different growth phases represents solely our personal opinion.

The number of cases reported may be significantly lower than the number of people actually infected. Also, we must point out that week-ends and changes in the test system may lead to fluctuations in reported cases that have no equivalent in actual case numbers.

Certainly, at this stage all statistical predictions are subject to great uncertainty because the general trends of the epidemic are not yet clear. In any case, the statistical trends that we interpret from the data are only suitable for predictions if the measures taken by the government and authorities to contain the pandemic remain in force and are being followed by the population. We must also point out that, even if the statistics indicate that the epidemic is under control, we may at any time see a resurgence of infection figures until the disease is eradicated worldwide.

CHAPTER

FIVE

# MODEL

If you are familiar with `pymc3`, then looking at the example below should explain how our model works. Otherwise, here is a quick overivew:

- First, we have to create an instance of the base class (that is derived from pymc3s model class). It has some convenient properties to get the range of the data, simulation length and so forth.
- We then add details that base model. They correspond to the actual (physical) model features, such as the change points, the reporting delay and the week modulation.
  - Every feature has it's own function that takes in arguments to set prior assumptions.
  - Sometimes they also take in input (data, reported cases ... ) but none of the function performs any actual modifactions on the data. They only tell pymc3 what it is supposed to do during the sampling.
  - None of our functions actually modifies any data. They rather define ways how pymc3 should modify data during the sampling.
  - Most of the feature functions add variables to the `pymc3.trace`, see the function arguments that start with `name_`.
- in pymc3 it is common to use a context, as we also do in the example. everything within the block `with cov19.model.Cov19Model(**params_model) as this_model:` automagically applies to `this_model`. Alternatively, you could provide a keyword to each function `model=this_model`.

## 5.1 Example

```
import datetime

import pymc3 as pm
import numpy as np
import covid19_inference as cov19

# limit the data range
bd = datetime.datetime(2020, 3, 2)

# download data
jhu = cov19.data_retrieval.JHU(auto_download=True)
new_cases = jhu.get_new(country="Germany", data_begin=bd)

# set model parameters
params_model = dict(
    new_cases_obs=new_cases,
    data_begin=bd,
```

(continues on next page)

(continued from previous page)

```
    fcast_len=28,
    diff_data_sim=16,
    N_population=83e6,
)

# change points like in the paper
change_points = [
    dict(pr_mean_date_transient=datetime.datetime(2020, 3, 9)),
    dict(pr_mean_date_transient=datetime.datetime(2020, 3, 16)),
    dict(pr_mean_date_transient=datetime.datetime(2020, 3, 23)),
]

# create model instance and add details
with cov19.model.Cov19Model(**params_model) as this_model:
    # apply change points, lambda is in log scale
    lambda_t_log = cov19.model.lambda_t_with_sigmoids(
        pr_median_lambda_0=0.4,
        pr_sigma_lambda_0=0.5,
        change_points_list=change_points,
    )

    # prior for the recovery rate
    mu = pm.Lognormal(name="mu", mu=np.log(1 / 8), sigma=0.2)

    # new Infected day over day are determined from the SIR model
    new_I_t = cov19.model.SIR(lambda_t_log, mu)

    # model the reporting delay, our prior is ten days
    new_cases_inferred_raw = cov19.model.delay_cases(
        cases=new_I_t,
        pr_mean_of_median=10,
    )

    # apply a weekly modulation, fewer reports during weekends
    new_cases_inferred = cov19.model.week_modulation(new_cases_inferred_raw)

    # set the likeliehood
    cov19.model.student_t_likelihood(new_cases_inferred)

# run the sampling
trace = pm.sample(model=this_model, tune=50, draws=10, init="advi+adapt_diag")
```

**Table of Contents**

## 5.2 Model Base Class

## 5.3 Compartmental models

### 5.3.1 SIR — susceptible-infected-recovered

### 5.3.2 More Details

$$
\begin{aligned}
I_{new}(t) &= \lambda_t I(t-1)\frac{S(t-1)}{N} \\
S(t) &= S(t-1) - I_{new}(t) \\
I(t) &= I(t-1) + I_{new}(t) - \mu I(t)
\end{aligned}
$$

The prior distributions of the recovery rate $\mu$ and $I(0)$ are set to

$$
\begin{aligned}
\mu &\sim \text{LogNormal}\left[\log(\text{pr\_median\_mu}), \text{pr\_sigma\_mu}\right] \\
I(0) &\sim \text{HalfCauchy}\left[\text{pr\_beta\_I\_begin}\right]
\end{aligned}
$$

---

### 5.3.3 SEIR-like — susceptible-exposed-infected-recovered

### 5.3.4 More Details

$$
\begin{aligned}
E_{\text{new}}(t) &= \lambda_t I(t-1)\frac{S(t)}{N} \\
S(t) &= S(t-1) - E_{\text{new}}(t) \\
I_{\text{new}}(t) &= \sum_{k=1}^{10} \beta(k) E_{\text{new}}(t-k) \\
I(t) &= I(t-1) + I_{\text{new}}(t) - \mu I(t) \\
\beta(k) &= P(k) \sim \text{LogNormal}\left[\log(d_{\text{incubation}}), \text{sigma\_incubation}\right]
\end{aligned}
$$

The recovery rate $\mu$ and the incubation period is the same for all regions and follow respectively:

$$
\begin{aligned}
P(\mu) &\sim \text{LogNormal}\left[\text{log(pr\_median\_mu), pr\_sigma\_mu}\right] \\
P(d_{\text{incubation}}) &\sim \text{Normal}\left[\text{pr\_mean\_median\_incubation, pr\_sigma\_median\_incubation}\right]
\end{aligned}
$$

The initial number of infected and newly exposed differ for each region and follow prior `HalfCauchy` distributions:

$$
\begin{aligned}
E(t) &\sim \text{HalfCauchy}\left[\text{pr\_beta\_E\_begin}\right] \quad \text{for } t \in -9, -8, ..., 0 \\
I(0) &\sim \text{HalfCauchy}\left[\text{pr\_beta\_I\_begin}\right].
\end{aligned}
$$

### 5.3.5 References

-
-

## 5.4 Likelihood

## 5.5 Spreading Rate

## 5.6 Delay

### 5.6.1 More Details

$$y_{\text{delayed}}(t) = \sum_{\tau=0}^{T} y_{\text{input}}(\tau)\text{LogNormal}\left[\log(\text{delay}), \text{pr\_median\_scale\_delay}\right](t-\tau)$$
$$\log(\text{delay}) = \text{Normal}\left[\log(\text{pr\_sigma\_delay}), \text{pr\_sigma\_delay}\right]$$

The *LogNormal* distribution is a function evaluated at $t - \tau$.

If the model is 2-dimensional (hierarchical), the $\log(\text{delay})$ is hierarchically modelled with the `hierarchical_normal()` function using the default parameters except that the prior *sigma* of *delay_L2* is HalfNormal distributed (`error_cauchy=False`).

## 5.7 Week modulation

## 5.8 Utility

CHAPTER

SIX

# DATA RETRIEVAL

**Table of Contents**

## 6.1 Utility

## 6.2 Johns Hops University

## 6.3 Robert Koch Institute

## 6.4 Robert Koch Institute situation reports

## 6.5 Google

## 6.6 Our World in Data

## 6.7 Financial times

## 6.8 Oxford COVID-19 Government Response Tracker

## 6.9 Base Retrieval Class

CHAPTER

SEVEN

# PLOTTING

We provide a lot of plotting functions which can be used to recreate our plots or create completely new visualizations. If you are familiar with `matplotlib` it should be no problem to use them extensively.

We provide three different types of functions here:

- *High level functions* These can be used create figures similar to our paper Dehning et al. arXiv:2004.01105. The are neat little one liners which create a good looking plot from our model, but do not have a lot of customization options.
- *Low level functions* These extend the normal `matplotlib` plotting functions and can be used to plot arbitrary data. They have a lot of customization options, it could take some time to get nicely looking plots with these functions though.
- *Helper functions* These are mainly functions that manipulate data or retrieve data from our model. These do not have to be used most of the time and are only documented here for completeness.

If one just wants to recreate our figures with a different color. The easiest was is to change the default rc parameters.

## 7.1 High level functions

## 7.2 Low level functions

### 7.2.1 Example

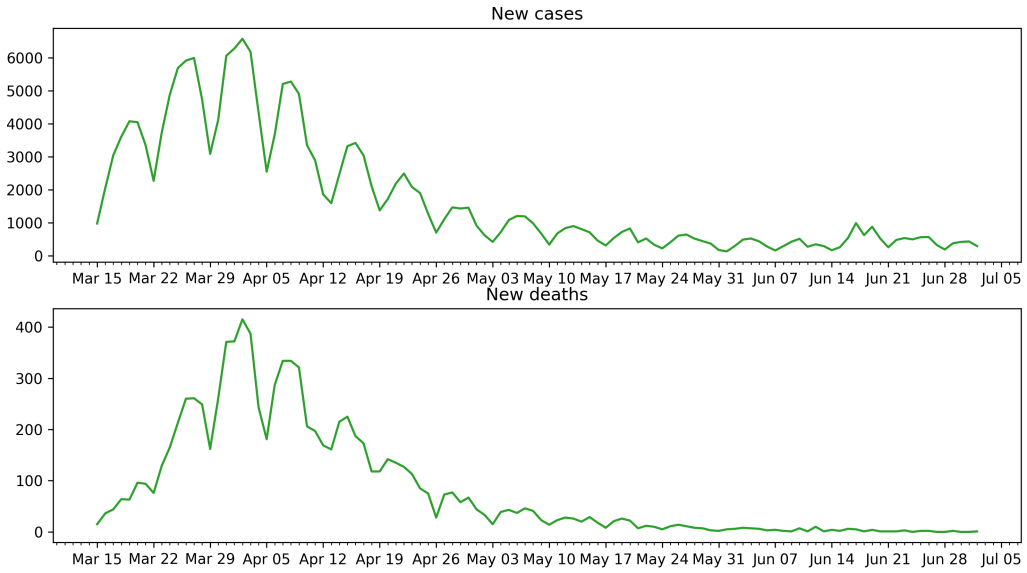
In this example we want to use the low level time series function to plot the new daily cases and deaths reported by the Robert Koch institute.

```
import datetime
import matplotlib.pyplot as plt
import covid19_inference as cov19

# Data retrieval i.e. download new data from RobertKochInstitue
rki = cov19.data_retrieval.RKI()
rki.download_all_available_data()

new_deaths = rki.get_new(
    value = "deaths",
    data_begin=datetime.datetime(2020,3,15), #arbitrary data
    data_end=datetime.datetime.today())

new_cases = rki.get_new(
```

(continues on next page)

(continued from previous page)

```
    value = "confirmed",
    data_begin=datetime.datetime(2020,3,15),
    data_end=datetime.datetime.today())

# Create a multiplot
fig, axes = plt.subplots(2,1, figsize=(12,6))

# Plot the new cases onto axes[0]
cov19.plot._timeseries(
    x=new_cases.index,
    y=new_cases,
    ax=axes[0],
    what="model", #We define model here to get a line instead of data points
)

# Plot the new deaths onto axes[1]
cov19.plot._timeseries(
    x=new_deaths.index,
    y=new_deaths,
    ax=axes[1],
    what="model", #We define model here to get a line instead of data points
)

# Label the plots

axes[0].set_title("New cases")

axes[1].set_title("New deaths")

# Show the figure
fig.show()
```



## 7.3 Helper functions

CHAPTER

# EIGHT

# VARIABLES SAVED IN THE TRACE

The trace by default contains the following parameters in the SIR/SEIR hierarchical model. XXX denotes a number.

| Name in trace | Dimensions | Created by function |
|---|---|---|
| lambda_XXX_L1 | samples | lambda_t_with_sigmoids/make_change_point_RVs |
| lambda_XXX_L2 | samples x regions | lambda_t_with_sigmoids/make_change_point_RVs |
| sigma_lambda_XXX_L2 | samples | lambda_t_with_sigmoids/make_change_point_RVs |
| transient_day_XXX_L1 | samples | lambda_t_with_sigmoids/make_change_point_RVs |
| transient_day_XXX_L2 | samples x regions | lambda_t_with_sigmoids/make_change_point_RVs |
| sigma_transient_day_XXX_L2 | samples | lambda_t_with_sigmoids/make_change_point_RVs |
| transient_len_XXX_L1 | samples | lambda_t_with_sigmoids/make_change_point_RVs |
| transient_len_XXX_L2 | samples x regions | lambda_t_with_sigmoids/make_change_point_RVs |
| sigma_transient_len_XXX_L2 | samples | lambda_t_with_sigmoids/make_change_point_RVs |
| delay_L1 | samples | delay_cases |
| delay_L2 | samples x regions | delay_cases |
| sigma_delay_L2 | samples | delay_cases |
| weekend_factor_L1 | samples | week_modulation |
| weekend_factor_L2 | samples x regions | week_modulation |
| sigma_weekend_factor_L2 | samples | week_modulation |
| offset_modulation | samples | week_modulation |
| new_cases_raw | samples x time x regions | week_modulation |
| mu | samples | SIR/SEIR |
| I_begin | samples x regions | SIR/SEIR |
| new_cases | samples x time x regions | SIR/SEIR |
| sigma_obs | samples x regions | SIR/SEIR |
| new_E_begin | samples x 11 x regions | SEIR |
| median_incubation_L1 | samples | SEIR |
| median_incubation_L2 | samples x regions | SEIR |
| sigma_median_incubation_L2 | samples | SEIR |

For the non-hierchical model, variables with _L2 suffixes are missing, and _L1 suffixes are removed from the name.

CHAPTER

NINE

# CONTRIBUTING

We always welcome contributions. Here we gather some guidelines to make the process as smooth as possible.

## 9.1 Beginning

To see where help is needed, go to the issues page on Github. If you want to begin on an issue, make a comment below and begin a draft pull request: https://github.blog/2019-02-14-introducing-draft-pull-requests/ You can link the pull request on the right side of the commit to it.

When you have finished working on the issue, change it to a regular pull request. Check that there are no conflicts to the current master (https://www.digitalocean.com/community/tutorials/how-to-rebase-and-update-a-pull-request)

## 9.2 Code formatting

We use black https://github.com/psf/black as automatic code formatter. Please run your code through it before you open a pull request.

We do not check for formatting in the testing (travis) but have a config in the repository that uses black as a pre-commit hook.

This snippet should get you up and running:

```
conda install -c conda-forge black
conda install -c conda-forge pre-commit
pre-commit install
```

Try to stick to PEP 8. You can use type annotations if you want, but it is not necessary or encouraged.

## 9.3 Testing

We use travis and pytest. To check your changes locally:

```
python -m pytest --log-level=INFO --log-cli-level=INFO
```

It would be great if anything that is added to the code-base has an according test in the `tests` folder. We are not there yet, but it is on the todo. Be encouraged to add tests :)

## 9.4 Documentation

The documentation is built using Sphinx from the docstrings. To test it before submitting, navigate with a terminal to the docs/ directory. Install if necessary the packages listed in `piprequirements.txt` run `make html`. The documentation can then be accessed in `docs/_build/html/index.html`. As an example you can look at the documentation of `covid19_inference.model.SIR()`

CHAPTER

TEN

# DEBUGGING

This is some pointer to help debugging models and sampling issues

## 10.1 General approach for nans/infs during sampling

The idea of this approach is to sample from the prior and then run the model. If the log likelihood is then -inf, there is a problem, and the output of the theano functions is inspected.

Sample from prior:

```
from pymc3.util import (
    get_untransformed_name,
    is_transformed_name)

varnames = list(map(str, model.vars))

for name in varnames:
    if is_transformed_name(name):
        varnames.append(get_untransformed_name(name))

with model:
    points = pm.sample_prior_predictive(var_names = varnames)
    points_list = []
    for i in range(len(next(iter(points.values())))):
        point_dict = {}
        for name, val in points.items():
                point_dict[name] = val[i]
        points_list.append(point_dict)
```

points_list is a list of the starting points for the model, sampled from the prior. Then to run the model and print the log-likelihood:

```
fn = model.fn(model.logpt)

for point in points_list[:]:
    print(fn(point))
```

To monitor the output and save it in a file (for use in ipython). Learned from: http://deeplearning.net/software/theano/tutorial/debug_faq.html#how-do-i-step-through-a-compiled-function

```
%%capture cap --no-stderr
def inspect_inputs(i, node, fn):
    print(i, node, "input(s) value(s):", [input[0] for input in fn.inputs],
```

(continues on next page)

(continued from previous page)

```
          end='')

def inspect_outputs(i, node, fn):
    print(" output(s) value(s):", [output[0] for output in fn.outputs])

fn_monitor = model.fn(model.logpt,
                      mode=theano.compile.MonitorMode(
                           pre_func=inspect_inputs,
                           post_func=inspect_outputs).excluding(
                                 'local_elemwise_fusion', 'inplace'))

fn = model.fn(model.logpt)

for point in points_list[:]:
    if fn(point) < -1e10:
        print(fn_monitor(point))
        break
```

In a new cell:

```
with open('output.txt', 'w') as f:
    f.write(cap.stdout)
```

Then one can open output.txt in a text editor, and follow from where infs or nans come from by following the inputs and outputs up through the graph

## 10.2 Sampler: MCMC (Nuts)

### 10.2.1 Divergences

During sampling, a significant fraction of divergences are a sign that the sampler doesn't sample the whole posterior. In this case the model should be reparametrized. See this tutorial for a typical example: https://docs.pymc.io/notebooks/Diagnosing_biased_Inference_with_Divergences.html

And these papers include some more details: https://pdfs.semanticscholar.org/7b85/fb48a077c679c325433fbe13b87560e12886.pdf https://arxiv.org/pdf/1312.0906.pdf

### 10.2.2 Bad initial energy

This typically occurs when some distribution in the model can't be evaluated at the starting point of chain. Run this to see which distribution throws nans or infs:

```
for RV in model.basic_RVs:
    print(RV.name, RV.logp(model.test_point))
```

However, this is only evaluates the test_point. When PyMC3 starts sampling, it adds some jitter around this test_point, which then could lead to nans. Run this to add jitter and then evaluate the logp:

```
chains=4
for RV in model.basic_RVs:
    print(RV.name)
```

(continues on next page)

(continued from previous page)

```
    for _ in range(chains):
        mean = {var: val.copy() for var, val in model.test_point.items()}
        for val in mean.values():
            val[...] += 2 * np.random.rand(*val.shape) - 1
        print(RV.logp(mean))
```

This code could potentially change in newer versions of PyMC3 (this is tested in 3.8). Read the source code, to know which random jitter PyMC3 currently adds at beginning.

### 10.2.3 Nans occur during sampling

Run the sampler with the debug mode of Theano.

```
from theano.compile.nanguardmode import NanGuardMode
mode = NanGuardMode(nan_is_error=True, inf_is_error=False, big_is_error=False,
                    optimizer='o1')
trace = pm.sample(mode=mode)
```

However this doesn't lead to helpful messages if nans occur during gradient evaluations.

## 10.3 Sampler: Variational Inference

There exist some ways to track parameters during sampling. An example:

```
with model:
    advi = pm.ADVI()
    print(advi.approx.group)

    print(advi.approx.mean.eval())
    print(advi.approx.std.eval())

    tracker = pm.callbacks.Tracker(
        mean=advi.approx.mean.eval,  # callable that returns mean
        std=advi.approx.std.eval  # callable that returns std
    )

    approx = advi.fit(100000, callbacks=[tracker],
                      obj_optimizer=pm.adagrad_window(learning_rate=1e-3),)
                     #total_grad_norm_constraint=10) #constrains maximal gradient,␣
↪could help


    print(approx.groups[0].bij.rmap(approx.params[0].eval()))

    plt.plot(tracker['mean'])
    plt.plot(tracker['std'])
```

For the tracker, the order of the parameters is saved in:

```
approx.ordering.by_name
```

and the indices encoded there in the slc field. To plot the mean value of a given parameter name, run:

```
plt.plot(np.array(tracker['mean'])[:, approx.ordering.by_name['name'].slc]
```

The debug mode is set with the following parameter:

```
from theano.compile.nanguardmode import NanGuardMode
mode = NanGuardMode(nan_is_error=True, inf_is_error=False, big_is_error=False,
                    optimizer='o1')
approx = advi.fit(100000, callbacks=[tracker],
             fn_kwargs={'mode':mode})
```

CHAPTER

ELEVEN

# INDICES AND TABLES

- genindex
- modindex
- search

# BIBLIOGRAPHY

[Nishiura2020] Nishiura, H.; Linton, N. M.; Akhmetzhanov, A. R. Serial Interval of Novel Coronavirus (COVID-19) Infections. Int. J. Infect. Dis. 2020, 93, 284–286. https://doi.org/10.1016/j.ijid.2020.02.060.

[Lauer2020] Lauer, S. A.; Grantz, K. H.; Bi, Q.; Jones, F. K.; Zheng, Q.; Meredith, H. R.; Azman, A. S.; Reich, N. G.; Lessler, J. The Incubation Period of Coronavirus Disease 2019 (COVID-19) From Publicly Reported Confirmed Cases: Estimation and Application. Ann Intern Med 2020. https://doi.org/10.7326/M20-0504.