Calibration*

Azadeh Kooshesh and Tom Krüger

MSc Informatics at Rostock University

1 Exercise

In Covasim, you can load demographic data and case data for a specific location, as demonstrated here. Find a case data set from Germany (plus corresponding demographic data). Calibrate the model to this data using mean squared error as a distance measure and optimization methods from the scipy package, as demonstrated here. Compare at least two different optimization methods regarding their performance (convergence, runtime). In your report, also explain how these methods work, and why you see this behavior. Then, re-run the what-if experiments and the sensitivity experiments you conducted with the pre-calibrated model during the previous milestones. How have the results changed?

2 Calibration

2.1 Nelder-Mead

The Nelder–Mead method, proposed by John Nelder and Roger Mead in 1965 [5], is a commonly used gradient-free algorithm for minimizing a target function. It uses a triangular shape, or simplex, to search for an optimal solution. Given n variables, the simplex is a special polytope of $n + 1$ vertices in n dimensions. The simplex shape flip flops towards its goal, growing, shrinking, and changing its shape according to a set of rules. The method approximates a local optimum of a problem.

2.2 Powell

Powell's method, strictly Powell's conjugate direction method, is an algorithm proposed by Michael J. D. Powell for finding a local minimum of a function. The function needs not be differentiable, and no derivatives are taken. The Powell method is a single-shot method that attempts to find the local fit-statistic minimum nearest to the starting point. Its principal advantage is that it is a robust direction-set method. A set of directions are defined; the method moves along one direction until a minimum is reached, then from there moves along the next direction until a minimum is reached, and so on, cycling through the whole set of directions until the fit statistic is minimized for a particular iteration. The set of directions is then updated and the algorithm proceeds [2].

* Data Driven Modeling and Simulation Project: Working with an Agent-based Model of COVID-19 dynamics and interventions

2.3 Calibration data sets

Covasim contains a script to scrape data from several sources. We downloaded the source from https://github.com/InstituteforDiseaseModeling/covasim/tree/master/data and executed the bash scipt *run_scrapers*. From this we investigated the Germany data[1]. As source for our calibration we decided to use the data from European Centre for Disease Prevention and Control Covid-19. It contains daily numbers of diagnoses and deaths between 31.12.2019 and 14.12.2020. In the year 2020 test-trace-quarantine was, alongside social distancing, the best countermeasure for the pandemic. To get a more suitable model we therefore decided to add real test data. The Robert-Koch-Institut publishes records of the antigen and PCR tests in an Excel format, see https://www.rki.de/DE/Content/InfAZ/N/Neuartiges_Coronavirus/Testzahl.html?nn=13490888. The Excel file *Testzahlen-gesamt.xlsx* contains a sheet named *1_Testzahlerfassung* with weekly number of tests.

The website https://www.worldometers.info/coronavirus/country/germany/#graph-cases-dailyfor09-2020 provides graphs with the number of active corona cases. The data is based on the number of positive tests (diagnoses). Since we believe that many infections were still undetected we multiplied the data by a factor of two. The number of care stations (hospital beds and intensive care) were the same as in our last milestone submissions.

The following code shows how we combined the data sources to our actual calibration data in Covasim readable format:

[1]:
```
import pandas as pd
import covasim as cv
cv.options.set('jupyter', verbose=0)
```

```
Covasim 3.1.1 (2021-12-06) | © 2021 by IDM
```

[2]:
```
# load data from European Centre for Disease Prevention and␣
↪Control
datafile = 'german_data/ecfdpc_Germany.csv'
pandemic_df = pd.read_csv(datafile)
pandemic_df.tail()
```

[2]:
```
     Unnamed: 0  day  new_diagnoses  new_deaths      key ␣
↪popData2019  \
345       21884  345          23679         440  Germany  ␣
↪83019213.0
346       21883  346          29875         598  Germany  ␣
↪83019213.0
```

[1] see https://git.informatik.uni-rostock.de/ak1543/data_driven/-/tree/main/Optimization/data/german_data

```
347       21882  347          28438         496  Germany  
 ↪83019213.0
348       21881  348          20200         321  Germany  
 ↪83019213.0
349       21880  349          16362         188  Germany  
 ↪83019213.0

    continentExp  cum per 100000 over 14 days  \
345       Europe  311.512228
346       Europe  320.027124
347       Europe  328.149341
348       Europe  334.881517
349       Europe  341.136696

           date
345  2020-12-10
346  2020-12-11
347  2020-12-12
348  2020-12-13
349  2020-12-14
```

[3]:
```
# load data from Robert-Koch-Institut
test_data_file = 'german_data/RKI_Testzahlen-gesamt.xlsx'
test_df = pd.read_excel(test_data_file,
 ↪sheet_name='1_Testzahlerfassung')
test_df.head()
```

[3]:
```
    Kalenderwoche  Anzahl Testungen  Positiv getestet  \
0  Bis KW10, 2020     69493  1722
1  11/2020            129291  7502
2  12/2020            374534  25886
3  13/2020            377599  33139
4  14/2020            417646  37649

   Positivenanteil (%)  Anzahl übermittelnder Labore
0                  NaN                           NaN
1             5.802415                         119.0
2             6.911522                         154.0
3             8.776241                         159.0
4             9.014572                         163.0
```

[4]:
```
df = pandemic_df[['date', 'new_diagnoses', 'new_deaths']]
tests_col = [0]
for index, row in test_df.iterrows():
```

```
    tests = row['Anzahl Testungen']
    if index == 0:
        # first 10 x 7 - 2 = 68 days
        tests_col += [round(tests / 68) for i in range(68)]
    else:
        week = row['Kalenderwoche'].split('/')
        test_approx = round(tests / 7)
        if int(week[0]) == 51:
            # add the last monday with data (14.12)
            tests_col += [test_approx]
            break
        else:
            tests_col += [test_approx for i in range(7)]

df['new_tests'] = tests_col
df.to_csv('data_covasim_de_2020.csv.csv')
```

Demographic data For the demographics from 2020 we found data for man and female per five year bins at https://www.populationpyramid.net/api/pp/276/2020/?csv=true. The following code shows how we used it to check the existing demographic data in Covasim:

[1]:
```
import pandas as pd
import covasim as cv
df = pd.read_csv('german_data/Democraphics_source_Germany-2020.
 ↪csv')
df.head()
```

```
Covasim 3.1.1 (2021-12-06) | © 2021 by IDM
```

[1]:

	Age	M	F
0	0-4	2082342	1976336
1	5-9	1970345	1851880
2	10-14	1979340	1832305
3	15-19	2150221	1968750
4	20-24	2382496	2170940

[2]:
```
germany_pop = dict()
s,e = '', ''
current = 0
total = 0
for index, row in df.iterrows():
    pop = row['M'] + row['F']
    total += pop
    if index == 16:
```

```
        current = pop
    elif index > 16:
        current += pop
    else:
        age_bin = row['Age'].split('-')
        pop = row['M'] + row['F']
        if index % 2 == 0:
            current = pop
            s = age_bin[0]
        else:
            current += pop
            e = age_bin[1]
            germany_pop[f"{s}-{e}"] = current

germany_pop['80+'] = current
print(f"Total: {total}")
germany_pop
covasim_pop = cv.data.country_age_data.data['Germany']
for k in covasim_pop.keys():
    a = germany_pop[k]
    b = covasim_pop[k]
    print(f"{k} : {a} | {b} | {abs(a-b)}")
```

```
Total: 83783943
0-9 : 7880903 | 7880902 | 1
10-19 : 7930616 | 7930616 | 0
20-29 : 9377361 | 9377359 | 2
30-39 : 10872020 | 10872019 | 1
40-49 : 10243351 | 10243351 | 0
50-59 : 13488393 | 13488393 | 0
60-69 : 10644142 | 10644140 | 2
70-79 : 7471414 | 7471414 | 0
80+ : 5875743 | 5875748 | 5
```

So the data in Covasim contains the demographics of Germany in 2020. Therefore, we simply used the *location* keyword argument for the *Sim* class.

2.4 Compare Nelder-Mead vs Powell

The Nelder-Mead and Powell methods are based on the simplex method and both are free-derivative optimization methods. But the first one is based on a probability density function and it can be valued at any given sample whereas the Powell method is based on a fast algorithm to uniformly sample the space [1]. As you can see later the Nelder-Mead and Powell are zero-order(ZO) methods, which is solving the optimization problems similar to the gradient-based method but we don't need any gradient in ZO and only used function evaluations [6]. But the convergence of the Nelder-Mead method in this program is better than the Powell. Our Powell method had a long runtime without end because of parallel compute many dimensional optimizations and it caused a low convergence in this method. In these sample articles you can find the Powell method in a large sample doesn't have convergence and also has a large cost:

1. Calibration in OpenDA [7]
2. Differential evolution with sensitivity analysis and the Powell's method for crowd model calibration [3]
3. A Simulation and Optimization Based Method for Calibrating Agent-based Emergency Department Models Under Data Scarcity [4]

2.5 Calibration process

We calibrated the Covasim model to the given data from Germany on the two most important parameters which we found in the Sensitivity Analysis milestone:

1. Beta β: Basic disease transmission per symptomatic contact
2. rel_severe_prob: Scale factor for proportion of symptomatic cases that become severe

Since we wanted to include most of the data but do not possess a High-Performance-Computer we made a trade-off between the simulated population and simulation runtime. We tested several different settings. Using a low number of agents in the simulation (e.g. 10000) and applying dynamic scaling to the real number of people in Germany did not work because the number of people that died was extremely low in comparison to the real data [2]. Simulating a million agents takes a long time for a single run, which would end in several hours or even days of runtime for the complete calibration process.
This is why we scaled the given data to a population of one million people and simulated 200.000 agents for the time between 01.09.2020 and 14.12.2020. We feel that this period is important because at the end of the summer the numbers had risen sharply again. By using dynamic scaling from Covasim we were able to find a working balance and simulated one million people in Germany. Each simulation run took a few seconds.
We applied a 7-day window mean rolling to smooth the data. We included this after some prior tests because there were too many sharp ups and downs, especially for the test data. It is common to use a 7-day period by institutions such as the Robert-Koch-Institute to compensate for differences in the reporting of the data.

As distance measure we applied mean squared error by adding the appropriate keyword arguments to method *compute_fit* at *Sim* instance. These arguments are then passed to Covasim's *Fit* class and used in *compute_gof* (https://docs.idmod.org/projects/covasim/en/latest/covasim.misc.html#covasim.misc.compute_

[2] See for instance https://git.informatik.uni-rostock.de/ak1543/data_driven/-/blob/main/Optimization/Calibration/pre_tests/Calibration_2020_597_death.ipynb

gof). It is important to note that there was a bug before Covasim version 3.1.0, which led to an error in this parameter passing.
For the calibration we tested the two methods *nelder-mead* and *powell*, which are implemented in *Scipy*. We applied the methods without any other arguments except the printing to console. The reason for this was to get a good comparison of the defaults. We will now show the Jupyter Notebook and discuss the results of the calibration afterward.

[1]:
```
import covasim as cv
import numpy as np
import pandas as pd
from scipy import optimize

cv.options.set('jupyter', verbose=0)
# needs Covasim 3.1.0 fix for gof kwargs
```

```
Covasim 3.1.1 (2021-12-06) | © 2021 by IDM
```

[2]:
```
# real german data
pop_size_total = 83780000
number_infected = 30000  # estimation "Dunkelziffer" 2 x https:/
 ↪/www.worldometers.info/coronavirus/country/germany/
 ↪#graph-cases-daily for 09-2020
datafile = 'data_covasim_de_2020.csv'
df = pd.read_csv(datafile)
df.tail()
```

[2]:

Unnamed: 0	date	new_diagnoses	new_deaths	new_tests
345	2020-12-10	23679	440	216577
346	2020-12-11	29875	598	216577
347	2020-12-12	28438	496	216577
348	2020-12-13	20200	321	216577
349	2020-12-14	16362	188	238862

[3]:
```
# simulation data
n_agents = 200000
n_beds_hosp = round(8 * n_agents / 1000)  # https://
 ↪tradingeconomics.com/germany/hospital-beds - 8 per 1000␣
 ↪people
n_beds_icu = round(620 * n_agents / 100000)  # https://
 ↪tradingeconomics.com/germany/icu-beds - 620 per 100.000␣
 ↪people
pop_infected = round(n_agents * (number_infected /␣
 ↪pop_size_total))
```

```
pop_size_sim = 1000000  # scaled simulated people
# scale the original data accordingly with 7 day rolling mean␣
↪window
for col in ['new_diagnoses', 'new_deaths', 'new_tests']:
    df[col] = df[col] / pop_size_total * pop_size_sim
    df[col] = np.round(df[col].rolling(7).mean())

# select dates
start_day = '2020-09-01'
end_day = '2020-12-14'
df['date'] = pd.to_datetime(df['date'])
mask = (df['date'] >= start_day) & (df['date'] <= end_day)
df = df.loc[mask]
df.tail()
```

[3]:

	Unnamed: 0	date	new_diagnoses	new_deaths	new_tests
345	345	2020-12-10	231.0	5.0	2497.0
346	346	2020-12-11	242.0	5.0	2526.0
347	347	2020-12-12	251.0	5.0	2556.0
348	348	2020-12-13	255.0	5.0	2585.0
349	349	2020-12-14	262.0	5.0	2623.0

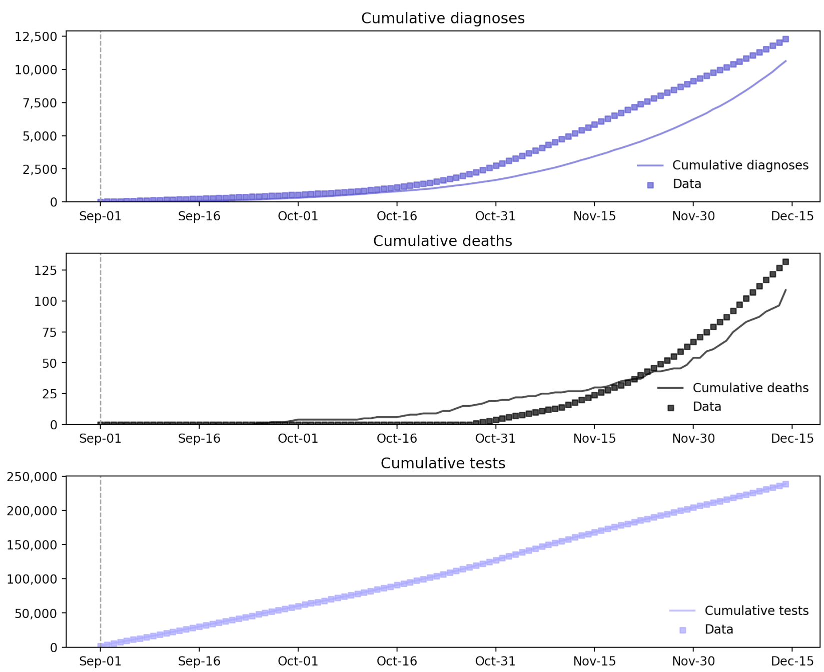
[4]:

```
# Initial guess of parameters - beta and relative severe␣
↪probability
guess = [0.0155, 1.1]
# Define baseline parameters
baseline_pars = dict(
    verbose=0,
    beta=guess[0],  # guess from previous runs
    rel_severe_prob=guess[1],  # guess from previous runs
    start_day=start_day,
    end_day=end_day,
    pop_type='hybrid',
    n_agents=n_agents,
    rescale=True,
    scaled_pop=pop_size_sim,  # references the scaled data
    pop_infected=pop_infected,
    location='Germany',  # use demographic data from 2020
    n_beds_hosp=n_beds_hosp,
    n_beds_icu=n_beds_icu
)

intervention_test = cv.test_num(daily_tests='data')
```

```
intervention_trace = cv.contact_tracing(trace_probs=dict(h=0.5, s=0.25, w=0.25, c=0.25))
interventions = [intervention_test, intervention_trace]
```

[5]:
```
def run_sim_and_plot(pars: dict):
    sim = cv.Sim(pars=pars, datafile=df, interventions=interventions)
    sim.run()
    return sim.plot(to_plot=['cum_diagnoses', 'cum_deaths', 'cum_tests'])


run_sim_and_plot(baseline_pars)
```

[5]:



[6]:
```
# use mean squared error as goodness of fit
# see https://docs.idmod.org/projects/covasim/en/latest/covasim.misc.html?highlight=compute_gof#covasim.misc.compute_gof
compute_gof_kwargs = {'normalize': False, 'use_squared': True, 'as_scalar': 'mean'}
```

```
def objective(x, n_runs=10):
    print(f'Running sim for beta={x[0]},␣
↪rel_severe_prob={x[1]}')
    opt_pars = dict(
        beta=x[0],
        rel_severe_prob=x[1],
    )
    pars = {**baseline_pars, **opt_pars}
    sim = cv.Sim(pars=pars, datafile=df,␣
↪interventions=interventions)
    msim = cv.MultiSim(sim)
    msim.run(n_runs=n_runs)
    mismatches = [sim.compute_fit(**compute_gof_kwargs).
↪mismatch for sim in msim.sims]
    return np.mean(mismatches)
```

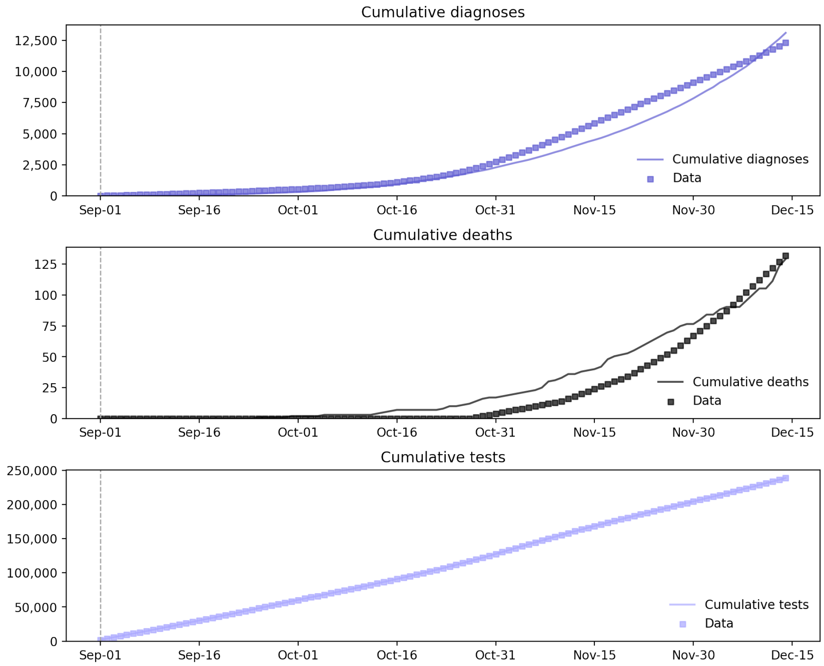
[7]:
```
opt_results_nelder = optimize.minimize(objective, x0=guess,␣
↪method='nelder-mead', options={'disp': True})
```

```
Running sim for beta=0.0155, rel_severe_prob=1.1
Running sim for beta=0.016275, rel_severe_prob=1.1
Running sim for beta=0.0155, rel_severe_prob=1.1550000000000002
Running sim for beta=0.016274999999999998, rel_severe_prob=1.045
Running sim for beta=0.016662499999999997, rel_severe_prob=0.
↪9899999999999998
Running sim for beta=0.017437499999999995, rel_severe_prob=0.
↪9899999999999998
Running sim for beta=0.015984375, rel_severe_prob=1.0725
Running sim for beta=0.016371874999999999, rel_severe_prob=0.
↪9624999999999999
Running sim for beta=0.017049999999999992, rel_severe_prob=0.
↪8799999999999997
Running sim for beta=0.01625078125, rel_severe_prob=1.024375
Running sim for beta=0.016541406250000005, rel_severe_prob=1.
↪051875
Running sim for beta=0.016414257812499992, rel_severe_prob=0.
↪9848437499999999
Running sim for beta=0.016456640624999998, rel_severe_prob=1.
↪0071875
Running sim for beta=0.016311328124999996, rel_severe_prob=0.
↪9934375
Running sim for beta=0.01639609375, rel_severe_prob=1.
↪0381249999999997
Running sim for beta=0.016190234375, rel_severe_prob=1.0553125
```

```
Running sim for beta=0.016044921875, rel_severe_prob=1.
↪0415625000000006
Running sim for beta=0.01630830078125, rel_severe_prob=1.038984375
Running sim for beta=0.0162205078125, rel_severe_prob=1.
↪0398437500000002
Running sim for beta=0.0163234375, rel_severe_prob=1.03125
..... SNIPPED FOR BREVITY ......
Running sim for beta=0.016251274884595476, rel_severe_prob=1.
↪026029709522846
Running sim for beta=0.016251275245437864, rel_severe_prob=1.
↪0260296110343194
Running sim for beta=0.016251275068828546, rel_severe_prob=1.
↪0260287997557322
Running sim for beta=0.01625127543729468, rel_severe_prob=1.
↪0260269802215047
Running sim for beta=0.016251275389330473, rel_severe_prob=1.
↪0260276379247084
Optimization terminated successfully.
         Current function value: 2819220.772609
         Iterations: 36
         Function evaluations: 93
```

[8]:
```python
opt_pars_nelder = dict(
    beta=opt_results_nelder.x[0],
    rel_severe_prob=opt_results_nelder.x[1],
)
run_sim_and_plot({**baseline_pars, **opt_pars_nelder})
```

[8]:



[9]:
```python
opt_results_powell = optimize.minimize(objective, x0=guess,␣
 ↪method='Powell', options={'disp': True})
# interrupted after 215 iterations because it seems to cycle...
```

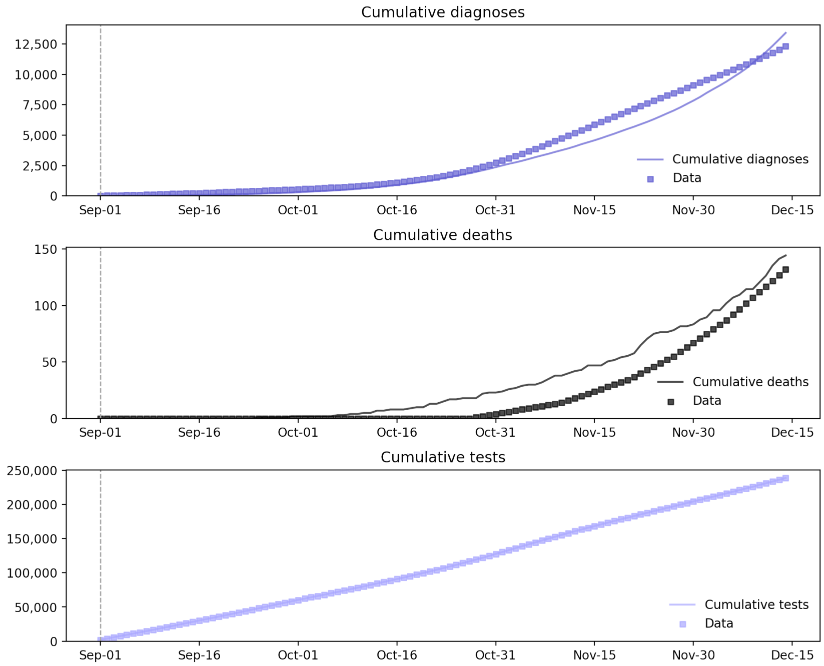
```
Running sim for beta=0.0155, rel_severe_prob=1.1
Running sim for beta=0.0155, rel_severe_prob=1.1
Running sim for beta=1.0155, rel_severe_prob=1.1
Running sim for beta=-1.602534, rel_severe_prob=1.1
Running sim for beta=0.0155, rel_severe_prob=1.1
Running sim for beta=-0.6025339748440001, rel_severe_prob=1.1
Running sim for beta=0.39746600000000004, rel_severe_prob=1.1
Running sim for beta=-0.2928479919568678, rel_severe_prob=1.1
Running sim for beta=0.16139802515600005, rel_severe_prob=1.1
Running sim for beta=-0.10227844909579697, rel_severe_prob=1.1
Running sim for beta=0.07122808507673671, rel_severe_prob=1.1
Running sim for beta=-0.02948736308732519, rel_severe_prob=1.1
Running sim for beta=0.03678623374442082, rel_severe_prob=1.1
Running sim for beta=-0.001683643129013257, rel_severe_prob=1.1
Running sim for beta=0.02363061755842144, rel_severe_prob=1.1
Running sim for beta=0.008936432568583322, rel_severe_prob=1.1
Running sim for beta=0.018605619466320004, rel_severe_prob=1.1
```

```
Running sim for beta=0.012992940402491496, rel_severe_prob=1.1
..... SNIPPED FOR BREVITY ......
Running sim for beta=0.01614969695800642, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=-0.6018842778859936, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=0.39811569695800647, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=-0.29211207943699047, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=0.16204772211400645, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=-0.10159582072448499, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=0.07187778203474313, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=-0.028825087449104097, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=0.037435930702427234, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=-0.0010291415428399578, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=0.02428031451642786, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=0.009587964731192132, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=0.019255316424326423, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=0.013643338346259071, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=0.017335938003078807, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=0.015846623230175744, rel_severe_prob=1.
 ↪0744559560590403
Running sim for beta=0.0163836532364136, rel_severe_prob=1.
 ↪0744559560590403
..... SNIPPED FOR BREVITY ......
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
 ↪0744559577462554
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
 ↪0744559571017966
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
 ↪074455956703499
```

```
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
↪0744559564573377
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
↪0744559563052016
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
↪0744559562111764
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
↪0744559561530656
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
↪0744559561171512
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
↪0744559560949547
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
↪0744559560812368
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
↪0744559560710147
Running sim for beta=0.016152786789737108, rel_severe_prob=1.
↪0744559560829892
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
↪0744559560710147
Running sim for beta=1.0161512418738718, rel_severe_prob=1.
↪0744559560710147
Running sim for beta=-1.6018827581261281, rel_severe_prob=1.
↪0744559560710147
Running sim for beta=0.016151241873871763, rel_severe_prob=1.
↪0744559560710147

Traceback (most recent call last):
..... SNIPPED FOR BREVITY ......
"/home/tom/.local/share/virtualenvs/data_driven-S9i1bRE7/lib/
↪python3.9/site-
packages/covasim/population.py", line 314, in make_hybrid_contacts
    contacts_dict['c'] = make_random_contacts(pop_size,␣
↪contacts['c'])
KeyboardInterrupt
```

[10]:
```
# insert values from stdout
opt_pars_powell = dict(
    beta=0.016151241873871763,
    rel_severe_prob=1.0744559560710147
)
run_sim_and_plot({**baseline_pars, **opt_pars_powell})
```

[10]:



2.6 Calibration evaluation

We started the calibration with an initial guess that did not fit the curve very accurately but at least a bit. The number of diagnoses and deaths were underestimated. The number of tests fits perfectly since we used this data as input. From the calibration, one would anticipate that beta and rel_death_prob should increase a bit.

The Nelder-mead method took 93 function evaluations and converged to reasonable values. The simulation fits better than before in both the number of diagnoses and deaths. However, as one can see from the output, Nelder-mead only searched locally around the initial guess. It changed both parameters in each iteration. Note that we snipped parts of the output because they were very long. See the Notebook and Powell_calibration_stdout.txt in GitLab for the complete outputs. The Powell method did not converge within 215 function evaluations. The method holds one parameter value fixed for several iterations in which it alters the other parameter. What one can see from the outputs is that Powell searches in a more global fashion because it sometimes alters a value quite strongly. However, due to this altering, it sometimes jumped to strange values even though it seemed to converge before. We think that this is the reason it did not converge in a reasonable amount of time. Nevertheless, it could be that many iterations would have been necessary as it looked as if the optimization sometimes sort of cycled. For example, the values at evaluation 54 were beta=0.0161492717497295, rel_severe_prob=1.0772808611211189 and within evaluations 156 to 210 beta was fixed at 0.016151241873871763, while rel_severe_prob was converging to 1.0744559560710147. But then beta jumped to 1.0161512418738718 and it started all over. We used the values before the last jump for reference.

An improvement in the overall runtime and convergence would probably have resulted from a higher tolerance threshold for the acceptance of a value. Since this run already lasted several hours, we decided not to repeat it with other thresholds.

We conclude, that Nelder-mead converges faster to a local minimum around the initial guess while Powell takes longer but has the ability to find a better local minimum. Putting the values at the end into a simulation run shows that both methods optimize the simulation so that the line fits better to the data. The parameters from the Nelder-mead method seem to fit a bit better. We will therefore use these for the rest of this work.

3 Re-running the last milestones

3.1 What-if-Experiments

Since we fixed our Covasim baseline parameters in our *helpers* file it was straightforward to include the calibration results. We only needed to set both parameters in the Python dictionary. We actually used the correct demographics all along in the last milestones. With the calibrated parameters the outcomes of our intervention scenarios were almost identical. We would have derived the same conclusions.

The Jupyter Notebooks with the graphs are in our gitlab repository [3]. Here, we only show the outcome for the intervention of social distancing which was supposed to represent the proper timing of interventions in the intervention milestone. The updated version of our helpers file is appended A.

[1]:
```
import covasim as cv

from helpers.what_if_helpers import run_base_and_intervention,␣
↪baseline_pars, optimized_base_pars

cv.options.set(dpi=100, show=False, close=True, verbose=0)  #␣
↪Standard options for Jupyter notebook
```

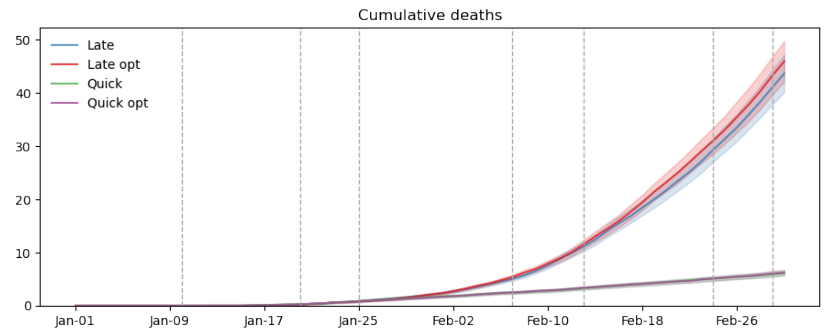
```
Covasim 3.0.7 (2021-06-29) | © 2021 by IDM
```

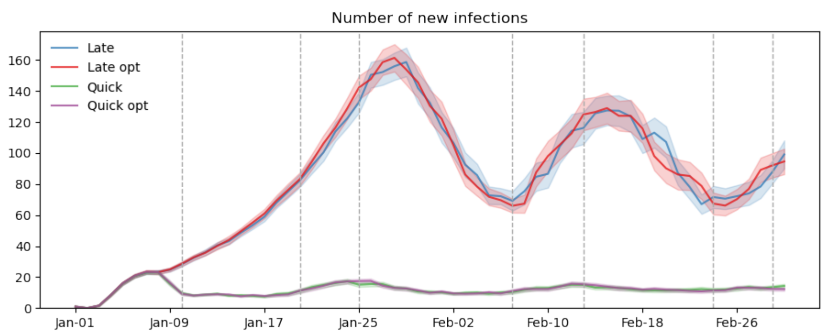
[3] https://git.informatik.uni-rostock.de/ak1543/data_driven/-/tree/main/Optimization/what-if-scenarios

[2]:

```python
from helpers.what_if_helpers import␣
↪init_intervention_for_inf_thresh, inf_thresh_callback

changes = [1 / 4, 1.0]  # remove 75 % of contacts

ce_late = cv.clip_edges(days=inf_thresh_callback(1000),␣
↪changes=changes)
ce_quick = cv.clip_edges(days=inf_thresh_callback(100),␣
↪changes=changes)

init_intervention_for_inf_thresh(ce_late)
init_intervention_for_inf_thresh(ce_quick)

base_sim = cv.Sim(baseline_pars, interventions=ce_late,␣
↪label='Late')  # late lockdown
base_sim_opt = cv.Sim(optimized_base_pars,␣
↪interventions=ce_late, label='Late opt')

lockdown_sim = cv.Sim(baseline_pars, interventions=ce_quick,␣
↪label='Quick')
lockdown_sim_opt = cv.Sim(optimized_base_pars,␣
↪interventions=ce_quick, label='Quick opt')

multi_sim = run_base_and_intervention(base_sim, base_sim_opt,␣
↪lockdown_sim, lockdown_sim_opt)

multi_sim.plot(to_plot=["cum_deaths", "new_infections"])
```

[2]:





According to this current output, we can notice with these calibrated parameters the final intervention's result scenarios were almost identical. We combined the graphs with the new version and we can see that changes in them are very imperceptible. Therefore, we would have derived to the same conclusions.

3.2 Sensitivity Analysis

For SA we needed to adjust the parameter ranges to our calibrated parameters. While doing the calibration we noticed that there is a rather high sensitivity to small changes on the values of the parameters. For instance, setting β to 0.03 gave a completely different scenario that did not fit the data at all. Therefore, we decided to adjust the boundaries in the SALib problem specification to more reasonable ranges. This means that we did the SA again on parameter ranges which fit more to the data from Germany. The boundaries in our problem specification changed as follows:

parameter	before	after
beta	[0.005, 0.03]	[0.013, 0.018]
asymp_factor	[0.5, 3.0]	[0.9, 1.0]
rel_severe_prob	[0.75, 4.5]	[0.9, 1.15]
rel_death_prob	[0.75, 4.5]	[0.9, 1.15]

The notebooks with the results of our changed problem specification are in our GitLab repository https://git.informatik.uni-rostock.de/ak1543/data_driven/-/tree/main/Optimization/sa. Here we only present Figure 1 with the final normalized bar plot on the indices of the SA methods.

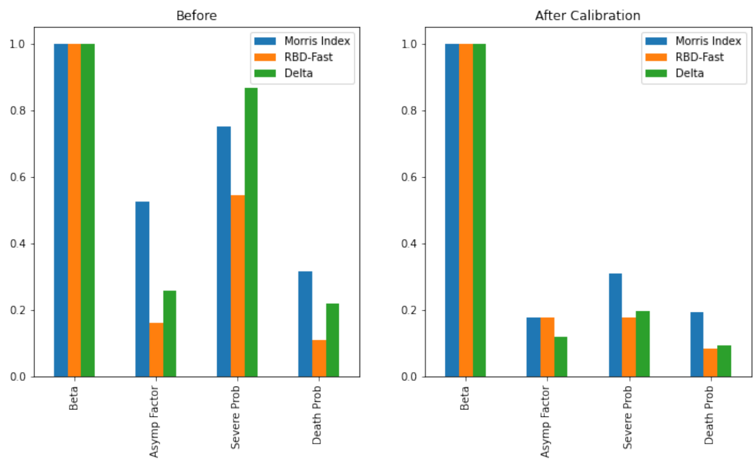


Fig. 1. Matplotlib bar plots of the normalized indices for the three applied sensitivity analysis methods before and after calibration.

The order of parameter importance regarding effects on the number of death did not change. What changed are the relative differences in the indices of the three SA methods. Beta seems to be way more important now, however, the relation in the first-order effects between the other three parameters seems to be the same as before. We conclude, that the range of parameters needs to be chosen well thought and has implications for the meaningfulness of SA. Nevertheless, we would have come to the same conclusion in that the virus transmission parameter is very important and the relative probability of transferring to the severe state is more important than the relative probability of transferring to the death state.

A What-if-Scenario Helper Functions

[ ]:
```
from functools import partial
import covasim as cv
import numpy as np
import scipy.stats as st

def get_current_infected_ratio():
    # Returns the current ratio of infected people in germany
```

```
    number_infected = 651500  # https://www.deutschland.de/de/
↪topic/politik/corona-in-deutschland-zahlen-und-fakten
    number_total = 83100000  # https://www.destatis.de/DE/
↪Themen/Gesellschaft-Umwelt/Bevoelkerung/Bevoelkerungsstand/
↪_inhalt.html
    infected_ratio = number_infected / number_total
    return infected_ratio

delta_variant = cv.variant('delta', days=0)  # delta was the␣
↪dominant variant in germany

# Define baseline parameters
baseline_pars = dict(
    start_day='2022-01-01',
    n_days=60,
    pop_type='hybrid',
    pop_size=10_000,
    pop_infected=int(get_current_infected_ratio() * 10000),
    location='Germany',
    use_waning=True,  # use dynamically calculated immunity
    n_beds_hosp=80,  # https://tradingeconomics.com/germany/
↪hospital-beds - 8 per 1000 people
    n_beds_icu=62,  # https://tradingeconomics.com/germany/
↪icu-beds - 620 per 100.000 people
    variants=[delta_variant],
)

optimized_base_pars = {**baseline_pars, **dict(
    beta=0.016251275389330473,
    rel_severe_prob=1.0260276379247084)
                       }

def run_simulations(sim: cv.Sim, n_runs: int, confidence_level:␣
↪float, method: str = "t") -> cv.MultiSim:
    msim = cv.MultiSim(sim)
    msim.run(n_runs=n_runs)
    if method == "t":  # use t-distribution
        bounds = st.t.interval(alpha=confidence_level,␣
↪df=n_runs - 1)[1]
    else:  # use normal distribution
        bounds = st.norm.interval(alpha=confidence_level)[1]
    bounds = bounds / np.sqrt(n_runs)
    msim.mean(bounds=bounds)
    return msim
```

```
def run_base_and_intervention(base_sim: cv.Sim, base_sim_opt:␣
↪cv.Sim, intervention_sim: cv.Sim,
                              intervention_sim_opt: cv.Sim,␣
↪n_runs: int = 100,
                              confidence_level: float = 0.9) ->␣
↪cv.MultiSim:
    sims = [base_sim, base_sim_opt, intervention_sim,␣
↪intervention_sim_opt]
    mean_sims = [run_simulations(s, n_runs, confidence_level).
↪base_sim for s in sims]
    return cv.MultiSim(mean_sims)

def _inf_thresh(self: cv.Intervention, sim: cv.Sim, thresh:␣
↪int):
    ''' Dynamically define on and off days with respect to the␣
↪number of infected people.
    See https://docs.idmod.org/projects/covasim/en/latest/
↪tutorials/tut_interventions.html#Dynamic-triggering'''

    if sim.people.infectious.sum() > thresh:
        if not self.active:
            self.active = True
            self.t_on = sim.t
            self.plot_days.append(self.t_on)
    else:
        if self.active:
            self.active = False
            self.t_off = sim.t
            self.plot_days.append(self.t_off)

    return [self.t_on, self.t_off]

def inf_thresh_callback(thresh: int = 500):
    return partial(_inf_thresh, thresh=thresh)

def init_intervention_for_inf_thresh(c: cv.Intervention):
    """Setup attributes for `inf_thresh_callback`"""
    c.t_on = np.nan
    c.t_off = np.nan
    c.active = False
    c.plot_days = []
    return c
```

References

1. Authors: A. Koscianski, M.L.: Globalization and Parallelization of Nelder-Mead and Powell Optimization Methods. Springer Netherlands (2008)
2. Birkinshaw, M.: Sherpa optimization methods. https://asc.harvard.edu/sherpa3.4/documents/manuals/html/refmethods.html#SECTION00690000000000000000 (1998)
3. Jinghui Zhong, W.C.: Differential evolution with sensitivity analysis and the Powell's method for crowd model calibration. Science Direct pp. 26–32 (07 2015). https://doi.org/10.1016/j.jocs.2015.04.013, https://www.sciencedirect.com/science/article/pii/S1877750315000514?via%3Dihub
4. Luquea, Z.L.D.R.F.E.E.: A Simulation and Optimization Based Method for Calibrating Agent-based Emergency Department Models Under Data Scarcity. Science Direct pp. 300–309 (01 2017). https://doi.org/10.1016/j.cie.2016.11.036, https://www.mcs.anl.gov/~zcliu/file/abm-calibration-zhengchun-liu.pdf
5. Nelder, J.A., Mead, R.: A Simplex Method for Function Minimization. The Computer Journal **7**(4), 308–313 (01 1965). https://doi.org/10.1093/comjnl/7.4.308, https://doi.org/10.1093/comjnl/7.4.308
6. Sijia Liu Pin-Yu Chen, B.K.P.K.V.: A Primer on Zeroth-Order Optimization in Signal Processing and Machine Learning. IEEE pp. 43 – 54 (06 2020). https://doi.org/10.1109/MSP.2020.3003837, https://ieeexplore.ieee.org/document/9186148
7. Verlaan, M.: Calibration in openda. http://www.openda.org/course/calibration.pdf