

# Appendix A

This appendix shows the imports that were required for the simulations in this report. Further utility functions, both adapted and custom made, are shown in Listing A.1, Listing A.2, Listing A.3 and Listing A.4.

## A.1 Utility and Utility Functions

This section includes listings that showcase the required imports, as well as utility functions that were built to be used together with the main simulation program. The main simulation program can be found in Appendix B. Listing A.1 shows the three types of imports that were required in this report.

**Listing A.1:** Required imports

```

1 # building and simulating battery models
2 import pybamm
3
4 # data processing
5 import numpy as np
6 import pandas as pd
7 import math
8
9 # memory packages
10 import tracemalloc
11 import gc

```

Listing A.2 shows how the data were obtained after the simulations were completed in this report. Note that there are multiple data types in PyBaMM, because of this three separate methods were constructed to obtain data for post-processing in MATLAB. The `save_data_to_csv` function saves one data point per cycle by using the simulation `solution` as an input, since the needed data are stored inside the `solution` object. The keyword argument `file_name_add_on` can be specified in the `save_data_to_csv` function such that CSV files are not overwritten.

**Listing A.2:** Processing and saving simulation data to a CSV file

```

1 def save_data_to_csv(N_per_slice, solution, file_name_add_on):
2     # reading the number of cycles which were simulated
3     N_cycles = solution.summary_variables["Cycle number"][-1]
4
5     # saving vars which depend on cycle number
6     dataID_names = ["Time [s]", "Terminal voltage [V]", "Battery voltage [V]",
7                    "Measured battery open circuit voltage [V]", "Resistance [Ohm]",
8                    "Current [A]", "C-rate", "Power [W]",
9                    "Discharge capacity [A.h]",
10                   "Total lithium lost [mol]", "LLI [%]",
11                   "Loss of lithium inventory, including electrolyte [%]",
12                   "LAM_ne [%]", "LAM_pe [%]",
13                   "Loss of active material in negative electrode [%]",
14                   "Loss of active material in positive electrode [%]",
15                   "Loss of capacity to SEI [A.h]",
16                   "Loss of lithium to SEI [mol]", "Discharge capacity [A.h]",
17                   "Loss of lithium to lithium plating [mol]",
18                   "Loss of capacity to lithium plating [A.h]",
19                   "Ambient temperature [K]"]
20     data_1Darray = [[], [], [], [], [], [], [], [], [], [], [],
21                    [], [], [], [], [], [], [], [], [], [], []]

```

```

22
23     i = 0
24     # saving data from each cycle
25     for entry_data1D_names in data1D_names:
26         # data from cycle 1 at the end of the first step.
27         # saved individually because the steps between the
28         # two first cycles which are saved, and the steps
29         # between the other cycles that are saved is different
30         data1D = solution.cycles[0].steps[0][entry_data1D_names]
31         # appending last value of the first step of cycle 1 to data1D_names
32         data_1Darray[i].append(data1D.data[-1])
33         for cycle_N in range(N_per_slice-1, N_cycles, N_per_slice):
34             # data from cycle N at the end of the first step
35             data1D = solution.cycles[cycle_N].steps[0][entry_data1D_names]
36             # appending last value of first step of cycle N to data1D_names
37             data_1Darray[i].append(data1D.data[-1])
38         i = i + 1
39
40     # saving the 1D variable data to a CSV file
41     df1 = pd.DataFrame(data_1Darray, index = data1D_names)
42     # disabled, saving in one file rather than three at the end of this function
43     # df1.to_csv(f"simX{str(file_name_add_on)}.csv", header = False, sep = ",")
44
45     # saving vars which depends on x and t
46     data2D_names = ["X-averaged SEI thickness [m]",
47                    "X-averaged lithium plating thickness [m]",
48                    "X-averaged Ohmic heating [W.m-3]",
49                    "X-averaged reversible heating [W.m-3]",
50                    "X-averaged irreversible electrochemical heating [W.m-3]",
51                    "X-averaged total heating [W.m-3]"]
52     data_2Darray = [[], [], [], [], [], []]
53
54     i = 0
55     # saving one averaged datapoint from each cycle for each data name
56     for data2D_name in data2D_names:
57         data2D = sum(solution.cycles[0][data2D_name].data)/len(solution.cycles[0][
58 data2D_name].data)
59         data_2Darray[i].append(data2D)
60         for cycle_N in range(N_per_slice-1, N_cycles, N_per_slice):
61             data2D = sum(solution.cycles[cycle_N][data2D_name].data)/len(solution.
62 cycles[cycle_N][data2D_name].data)
63             data_2Darray[i].append(data2D)
64         i = i + 1
65
66     # saving the 2D variable data to a CSV file
67     df2 = pd.DataFrame(data_2Darray, index = data2D_names)
68     # disabled, saving in one file rather than three at the end of this function
69     # df2.to_csv(f"simY{str(file_name_add_on)}.csv", header = False, sep = ",")
70
71     # saving summary variables
72     dataSummary_names = ["Cycle number", "Capacity [A.h]",
73                          "Local ECM resistance [Ohm]",
74                          "Loss of lithium inventory [%]",
75                          "Loss of lithium inventory, including electrolyte [%]",
76                          "Measured capacity [A.h]",
77                          "Negative electrode capacity [A.h]",
78                          "Positive electrode capacity [A.h]",
79                          "Total capacity lost to side reactions [A.h]",
80                          "Total lithium [mol]",
81                          "Total lithium in electrolyte [mol]",
82                          "Total lithium in negative electrode [mol]",
83                          "Total lithium in particles [mol]",
84                          "Total lithium in positive electrode [mol]",
85                          "Total lithium lost [mol]",

```

```

84         "Total lithium lost from electrolyte [mol]",
85         "Total lithium lost from particles [mol]",
86         "Total lithium lost to side reactions [mol]"
87     data_Summaryarray = [[], [], [], [], [], [], [], [], [],
88                          [], [], [], [], [], [], [], [], [], []]
89
90     i = 0
91     # saving data from each summary variable
92     for dataSummary_name in dataSummary_names:
93         save_data_typeSummary = solution.summary_variables[dataSummary_name][0]
94         data_Summaryarray[i].append(save_data_typeSummary)
95         for cycle_N in range(N_per_slice-1, N_cycles, N_per_slice):
96             save_data_typeSummary = solution.summary_variables[dataSummary_name][
97                 cycle_N]
98             data_Summaryarray[i].append(save_data_typeSummary)
99             i = i + 1
100
101     # saving the summary variables to a CSV file
102     df3 = pd.DataFrame(data_Summaryarray, index = dataSummary_names)
103     # disabled, saving in one file rather than three at the end of this function
104     # df3.to_csv(f"simZ{str(file_name_add_on)}.csv", header = False, sep = ",")
105
106     # collecting all variable values and names into respective arrays
107     save_data_to_csvFile = data_1Darray + data_2Darray + data_Summaryarray
108     varData_names = data1D_names + data2D_names + dataSummary_names
109
110     # concatenating the dataframes and aligning them along the row (index) axis
111     df = pd.concat([df1, df2, df3], axis = 0)
112     df = pd.DataFrame(save_data_to_csvFile, index = varData_names)
113     # saving the data to a CSV file for further post-processing
114     df.to_csv(f"Sim{str(file_name_add_on)}.csv", header = False, sep = ",")
115     return

```

The memory tracking function, `display_memory_usage`, were utilised together with the garbage collector module, `gc`, from Listing A.1, to map memory usage and memory clearing. Together, they were used to pinpoint memory usage by the different program components to a reasonable accuracy. The `display_memory_usage` in Listing A.3 is directly adopted from the `tracemalloc` documentation [141].

**Listing A.3:** Memory utilization tracking function

```

1  # https://docs.python.org/3/library/tracemalloc.html
2  def display_memory_usage(snapshot, key_type = 'lineno'):
3      snapshot = snapshot.filter_traces((
4          tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
5          tracemalloc.Filter(False, "<unknown>"),
6      ))
7      top_stats = snapshot.statistics(key_type)
8      total = sum(stat.size for stat in top_stats)
9      print("Total allocated size: %.1f GiB" % (total/(1024*10**6)))
10     return

```

Listing A.4 presents a custom memory clearing function. The `RAM_clearing_routine` replicates the functionality provided by the `save_at_cycles` keyword argument in the `PyBaMM solve` method. The function were however not based on any source code. To see that the `RAM_clearing_routine` worked as intended, it was checked against the functionality which `save_at_cycles` provides. After some iterations of the `RAM_clearing_routine` it provided the same functionality, but it can be used for any cycling scheme involving a loop, unlike `save_at_cycles` which were not designed for this.

Listing A.4: Memory clearing function

```

1 def RAM_clearing_routine(cycle_N, N_per_slice, sol):
2     # calculating which slice to clear solutions from
3     clearing_N = math.ceil(cycle_N/N_per_slice)
4     # If first slice, data needs to be treated differently,
5     # since the first cycle should be saved
6     if clearing_N == 1:
7         # looping through the cycles of the first slice
8         for N in range(len(sol.cycles)): # N number of cycles in the first slice
9             if N == 0:
10                # saving the first cycle
11                continue
12            elif (N+1)%N_per_slice:
13                # clearing RAM by replacing sol with None
14                sol.cycles[N] = None
15            else:
16                # saving the ones specified by N_per_slice
17                continue
18     # all the slices after the first slice is treated similarly
19     # since the first cycle of these slices should not be saved
20     else:
21         if cycle_N < N_per_slice*clearing_N:
22             # calculating the number of cycles which were
23             # not completed within the slice
24             subN = N_per_slice*clearing_N - len(sol.cycles)
25             # looping through the cycles of the partially completed slice
26             for N in range((clearing_N-1)*N_per_slice, clearing_N*N_per_slice-subN):
27                 # clearing RAM by replacing sol with None
28                 sol.cycles[N] = None
29         else:
30             # looping through the cycles of a fully completed slice
31             for N in range((clearing_N-1)*N_per_slice, clearing_N*N_per_slice):
32                 if (N+1)%N_per_slice:
33                     # clearing RAM by replacing sol with None
34                     sol.cycles[N] = None
35                 else:
36                     # saving the ones specified by N_per_slice
37                     continue

```

# Appendix B

This appendix includes Listing B.1 which shows how the simulations of this report were customized. Listing B.2 further shows how these settings were implemented into simulations, and in the end simulated.

## B.1 Customizing and Conducting the SoC Window Simulation

Listing B.1 shows how the discussed settings from Chapter 4, are implemented into a simulation, and how it is solved. The base parameter set is set, further, Listing B.1 shows the imported parameters, submodel options, solver options, and mesh customisation. At the end of Listing B.1 all the listed choices are implemented into a custom simulation function, which was utilised to cycle between two defined SoC values. This function can be found in Listing B.2. To vary any given parameter three lines of code are simply changed in Listing B.1. The program has been divided into sections to ease the readability. The sections are defined as presented in the following list.

- Simulation settings
  - Lines 6-33
- Setting submodel options
  - Lines 36-47
- Parameter settings
  - Lines 49-126
- Solver settings
  - Lines 129-131
- Mesh settings
  - Lines 135-147
- Solving the specified battery model
  - Lines 150-173
- Vary another parameter
  - Lines 16, 168 and 170

**Listing B.1:** Customizing the simulation

```

1  ### UTILITY SECTION ###
2  # setting the logging level to notice for visualisation of simulation progress:
3  pybamm.set_logging_level("NOTICE")
4
5
6  ### SIMULATION SETTINGS SECTION ###
7  # defining important variables for the type of experiment which is conducted:
8  # how many cycles for each experiment + which SoC ranges to test in the
9  # experiments, C-rates for charge and discharge are also set.
10
11 # Note that for every array element a corresponding datafile

```

```

12 # will be created. It can be ID'd by its value.
13 # Which SoC ranges to cycle between and compare:
14 # The different ambient temperatures to compare ageing
15 # between in degrees C:
16 T_amb = [0, 25, 50]
17 # default value: 25 degrees C (298.15 K)
18
19 # Number of cycles per temperature setting:
20 N_cycles_per_param = 80
21 # Number of cycles before clearing RAM:
22 # (Must be smaller than or to N_cycles_per_param)
23 cycles_per_slice = 40
24 # Done to reduce RAM usage, clears RAM after
25 # every slice of N cycles is run through
26
27 # C-rates:
28 charging_C_rate = 2
29 discharging_C_rate = 1
30
31 # Which initial SoC's to cycle between:
32 lower_SoC = 0.3
33 upper_SoC = 0.7
34
35
36 ### MODEL SECTION ###
37 # setting submodel options:
38 LAM_li_plt_SEI_thermal_options = {
39     "particle mechanics": "swelling and cracking",
40     "loss of active material" : "stress-driven",
41     "lithium plating" : "irreversible",
42     "lithium plating porosity change" : "true",
43     "SEI" : "solvent-diffusion limited",
44     "SEI film resistance" : "distributed",
45     "SEI porosity change" : "true",
46     "thermal" : "lumped"
47 }
48
49 ### PARAMETER SECTION ###
50 # choosing Chen2020 as base parameter set and editing it:
51 chen2020_params = pybamm.parameter_sets.Chen2020
52
53 # to run with Oregon2021 as the base parameter set
54 # https://github.com/pybamm-team/PyBaMM/blob/develop/pybamm/input/parameters/
55 # lithium_ion/seis/example/parameters.csv
56 # chen2020_params["sei"] = "example"
57
58 # Lithium plating parameters inserted from Okane2020 parameter set:
59 # https://github.com/pybamm-team/PyBaMM/tree/develop/pybamm/input/parameters/
60 # lithium_ion/lithium_platings/okane2020_Li_plating
61 chen2020_params["lithium plating"] = "okane2020_Li_plating"
62
63 # ParameterValues set such that missing parameters can be inserted:
64 params = pybamm.ParameterValues(chemistry = chen2020_params)
65
66 # Negative electrode parameters inserted from Ai2020 parameter set:
67 # https://github.com/pybamm-team/PyBaMM/blob/develop/pybamm/input/parameters/
68 # lithium_ion/negative_electrodes/graphite_Ai2020/parameters.csv
69
70 # Negative electrode, mechanical properties:
71 params.update({"Negative electrode Poisson's ratio" : 0.3}, check_already_exists =
72             False)
73 params.update({"Negative electrode Young's modulus [Pa]" : 15e9},
74             check_already_exists = False)
75 params.update({"Negative electrode reference concentration for free of deformation [

```

```

    mol.m-3]" : 0}, check_already_exists = false)
74 params.update({"Negative electrode partial molar volume [m3.mol-1]" : 3.1e-6},
    check_already_exists = false)
75 # Parameter defined as a function, as shown in Table 4.6
76 params.update({"Negative electrode volume change" : graphite_volume_change_Ai2020},
    check_already_exists = false)
77
78 # Negative electrode, crack model:
79 params.update({"Negative electrode initial crack length [m]" : 20e-9},
    check_already_exists = false)
80 params.update({"Negative electrode initial crack width [m]" : 15e-9},
    check_already_exists = false)
81 params.update({"Negative electrode number of cracks per unit area [m-2]" : 3.18e15},
    check_already_exists = false)
82 params.update({"Negative electrode Paris' law constant b" : 1.12},
    check_already_exists = false)
83 params.update({"Negative electrode Paris' law constant m" : 2.2},
    check_already_exists = false)
84 # Parameter defined as a function, as shown in Table 4.6
85 params.update({"Negative electrode cracking rate" : graphite_cracking_rate_Ai2020},
    check_already_exists = false)
86 params.update({"Negative electrode activation energy for cracking rate [J.mol-1]" :
    0}, check_already_exists = false)
87
88 # Negative electrode, loss of active materials (LAM) model:
89 params.update({"Negative electrode LAM constant proportional term [s-1]" : 0},
    check_already_exists = false)
90 params.update({"Negative electrode LAM constant exponential term" : 2},
    check_already_exists = false)
91 params.update({"Negative electrode critical stress [Pa]" : 60e6},
    check_already_exists = false)
92
93 # Positive electrode parameters inserted from Ai2020 parameter set:
94 # https://github.com/pybamm-team/PyBaMM/blob/develop/pybamm/input/parameters/
95 # lithium_ion/positive_electrodes/lico2_Ai2020/parameters.csv
96
97 # Positive electrode, mechanical properties:
98 params.update({"Positive electrode Poisson's ratio" : 0.2}, check_already_exists =
    false)
99 params.update({"Positive electrode Young's modulus [Pa]" : 375e9},
    check_already_exists = false)
100 params.update({"Positive electrode reference concentration for free of deformation [
    mol.m-3]" : 0}, check_already_exists = false)
101 params.update({"Positive electrode partial molar volume [m3.mol-1]" : -7.28e-7},
    check_already_exists = false)
102 # Parameter defined as a function, as shown in Table 4.6
103 params.update({"Positive electrode volume change" : lico2_volume_change_Ai2020},
    check_already_exists = false)
104
105 # Positive electrode, crack model:
106 params.update({"Positive electrode initial crack length [m]" : 20e-9},
    check_already_exists = false)
107 params.update({"Positive electrode initial crack width [m]" : 15e-9},
    check_already_exists = false)
108 params.update({"Positive electrode number of cracks per unit area [m-2]" : 3.18e15},
    check_already_exists = false)
109 params.update({"Positive electrode Paris' law constant b" : 1.12},
    check_already_exists = false)
110 params.update({"Positive electrode Paris' law constant m" : 2.2},
    check_already_exists = false)
111 # Parameter defined as a function, as shown in Table 4.6
112 params.update({"Positive electrode cracking rate" : lico2_cracking_rate_Ai2020},
    check_already_exists = false)
113 params.update({"Positive electrode activation energy for cracking rate [J.mol-1]" :

```

```

    0}, check_already_exists = false)
114
115 # Positive electrode, loss of active materials (LAM) model:
116 params.update({"Positive electrode LAM constant proportional term [s-1]" : 2.78e-13},
    check_already_exists = false)
117 params.update({"Positive electrode LAM constant exponential term" : 2},
    check_already_exists = false)
118 params.update({"Positive electrode critical stress [Pa]" : 375e6},
    check_already_exists = false)
119
120 # Cell parameter inserted from Ai2020:
121 # https://github.com/pybamm-team/PyBaMM/blob/develop/pybamm/input/parameters/
122 # lithium_ion/cells/Enertech_Ai2020/parameters.csv
123 params.update({"Cell thermal expansion coefficient [m.K-1]" : 1.1e-6},
    check_already_exists = false)
124
125 # Lowering the cut-off voltage:
126 # params.update({"Lower voltage cut-off [V]" : 2.5}) # default value: 2.8 V
127
128
129 ### CHOSING SOVLER SECTION ###
130 # Using the CasadiSolver with custom settings to solve the model
131 solver = pybamm.CasadiSolver(atol = 1e-6, rtol = 1e-6, mode = "fast with events")
132
133
134
135 ### MESH SECTION ###
136 # making the mesh finer to avoid SolverErrors
137 var_pts = {
138     "x_n" : 50, # x-direction, length, negative electrode
139     "x_s" : 50, # x-direction, length, separator
140     "x_p" : 50, # x-direction, length, positive electrode
141     "r_n" : 50, # number of volumes in the radial direction, negative particle
142     "r_p" : 50, # number of volumes in the radial direction, positive particle
143     "y" : 10, # y-direction, depth (kept at default value)
144     "z" : 10, # z-direction, height (kept at default value)
145     "R_n" : 30, # negative particle radius (kept at default value)
146     "R_p" : 30 # positive particle radius (kept at default value)
147 }
148
149
150 ### ACQUIRING INITIAL CELL CAPACITY SECTION ###
151 # running an experiment to get the initial battery cell capacity
152 mapping_experiment = pybamm.Experiment([
153     ("Rest for 1 minute")]
154 )
155
156 # extracting initial capacity of a fresh LiB cell by solving
157 # the model and reading the "Capacity [A.h]" summary variable.
158 dfn = pybamm.lithium_ion.DFN()
159 mapping_sim = pybamm.Simulation(dfn, parameter_values = params, solver = solver,
    experiment = mapping_experiment, var_pts = var_pts)
160 # start at 0.3 SoC to avoid minimum/maximum voltage error
161 mapping_sol = mapping_sim.solve(initial_soc = 0.3)
162 mapping_solution = mapping_sim.solution
163 initial_cap = mapping_solution.summary_variables["Capacity [A.h]"][0] # Ah
164
165
166 ### SOLVING THE SPECIFIED SIMULATION SECTION ###
167 # SIM start, one for each parameter value
168 for T_amb in T_ambs:
169     # updating the param which is investigated
170     params.update({"Ambient temperature [K]" : 273.15 + T_amb})
171     conductSim = sim_N_cycles_individually(N_cycles = N_cycles_per_param,

```

```

N_per_slice = cycles_per_slice, solution = mapping_solution, lower_SoC =
SoC_range[0], upper_SoC = SoC_range[1], options = LAM_li_plt_SEI_thermal_options,
params = params, solver = solver, var_pts = var_pts, C_charge = charging_C_rate
, C_discharge = discharging_C_rate)
172 # saving specified data
173 save_data_to_csv(N_per_slice = cycles_per_slice, solution = conductSim[0],
file_name_add_on = str(T_amb))

```

Listing B.2 shows how the SoC cycling method was performed after all settings were set, as shown in Listing B.1. The `sim_N_cycles_individually` function accepts information on amount of cycles, when to clear memory, starting solution to use, lower and upper SoC, C-rates, submodel options, parameter values, solver and mesh. It makes sure that the simulation starts on the specified lower SoC, as seen in line 25. The next cycles are, however, set to start on the previous cycle's simulation `sol`, as seen in line 31. Several checks are implemented from line 26 to 45 to deal with the minimum and maximum voltage issue, for instance, as discussed in Section 3.4.3. The memory clearing routine is also called every `N_per_slice` cycles in line 42, to counteract the general relation of rising memory utilisation as a function of number of cycles. Even though it copied the only memory saving technique presented in the PyBaMM documentation, it only helped to alleviate some memory usage.

**Listing B.2:** SoC window simulation function

```

1 def sim_N_cycles_individually(N_cycles, N_per_slice, solution, lower_SoC, upper_SoC,
2 options, params, solver, var_pts, C_charge, C_discharge):
3     sim_stopped_progression = False
4     # set to false by default since the sim is not stopped by default
5     # cycling through every cycle one by one
6     for cycle_N in range(1, N_cycles+1):
7         # obtaining remaining cap
8         remaining_cap = solution.summary_variables["Capacity [A.h]"][-1]
9         initial_lower_Q = lower_SoC * remaining_cap # Ah
10        initial_upper_Q = upper_SoC * remaining_cap # Ah
11        charging_current = C_charge * remaining_cap # A
12        discharging_current = C_discharge * remaining_cap # A
13        charge_time = (initial_upper_Q - initial_lower_Q)/charging_current # h
14        discharge_time = (initial_upper_Q - initial_lower_Q)/discharging_current # h
15
16        # defining experiment, model and simulation
17        experiment_N_1 = pybamm.Experiment([
18            (f"Charge at {charging_current} A for {charge_time} hours",
19             f"Discharge at {discharging_current} A for {discharge_time} hours")]
20        )
21        dfn = pybamm.lithium_ion.DFN(options = options)
22        sim = pybamm.Simulation(dfn, parameter_values = params, solver = solver,
23                               experiment = experiment_N_1, var_pts = var_pts)
24
25        if cycle_N == 1:
26            # solving sim, first cycle starting on lower SoC
27            sol = sim.solve(initial_soc = lower_SoC)
28            try: # checking if the first cycle was successfully solved
29                len(sol.cycles)
30            except AttributeError:
31                raise Exception('The first cycle did not complete. Use pybamm.
32                set_logging_level("NOTICE") to see what went wrong.')
33            else:
34                # solving sim, starting on sol
35                sol = sim.solve(starting_solution = sol)
36            # obtaining information for starting point for next cycle
37            solution = sim.solution

```

```
35
36     # check if last cycle were successfully solved
37     if len(sol.cycles) != cycle_N:
38         sim_stopped_progression = True
39
40     # RAM clearing routine enabled every N_per_slice cycles
41     if cycle_N%N_per_slice == 0 or sim_stopped_progression == True:
42         RAM_clearing_routine(cycle_N, N_per_slice, sol)
43         # loop is cut short if the sim is not progressing
44         if sim_stopped_progression:
45             return [solution, sol]
46     return [solution, sol]
```

# Appendix C

This appendix is included to highlight a proposed SoC pathfinder algorithm. The basic working principle is explained, and the program is presented in Listing C.1.

## C.1 Proposal for an Optimal SoC Window Pathfinder Algorithm

A pathfinder theorem has been conjectured, and it has been shown to increase SoH as a function of time and cycle number in simulations. The use of the program presented in Listing C.1 has been shown to increase capacity retention compared to cycling at static SoC windows, with the method shown in Listing B.2. Note that the  $\Delta$ SoC stays the same between the comparisons. The program finds the best static SoC range for a set amount of cycles defined by `loop_n_cycles`. Then a new best SoC range is found for the next `loop_n_cycles`. This is repeated until the number of desired cycles are completed. Note that the code, as presented in Listing C.1, checks with 2% SoC steps and excludes the 20% upper and 20% lower states of charge. This is done to decrease compute time. Accuracy decreases when using 2% steps compared to 1%, and lower. It was however not practical to do the simulations with any lower resolution due to both computational constraints, and the fact that SoC cannot be measured directly in the real world. The upper and lower states of charge were also eliminated, and thus not checked, in order not to waste any time checking the areas which are traditionally known as the worst for battery health.

**Listing C.1:** Proposed pathfinder algorithm

```

1  ### SIMULATION SETTINGS SECTION ###
2  # How many cycles to find the best SoC range for:
3  N_cycles = 1000
4
5  # Which C-rates to use throughout the experiment:
6  charging_C_rate = 2
7  discharging_C_rate = 1
8
9  # How large should the operating SoC range be (given as X %):
10 # Script searches from min SoC of 20% to max SoC of 80%
11 # A range_SoC of 30 cycles between 30%-60% for example
12 range_SoC = 40
13 # lowest SoC value to try
14 loop_from_lower_soc = 20
15 # when to check for new optimal SoC range
16 loop_n_cycles = 50
17 # Max upper SoC, set to 80%
18 max_upper_loop_SoC = round(100-19-range_SoC, 2)
19
20
21
22
23 ### SOLVING THE SPECIFIED SIMULATION SECTION ###
24 # making empty lists such that information can be appended at the
25 # end of each experiment
26 caps, sols, actual_lowerSoC_maxs, actual_upperSoC_maxs = [], [], [], []
27 N_split = N_cycles//loop_n_cycles-1
28 cap_max = 0
29 # doing "N_split #1"
30 for lowerSoC in range(loop_from_lower_soc, int(max_upper_loop_SoC), 2): # 20-80%, 2%
31     actual_lowerSoC = lowerSoC/100
32     actual_upperSoC = (lowerSoC + range_SoC)/100
33     upperSoC = lowerSoC + range_SoC

```

```

34 lowerQ = lowerSoC * initial_cap/100
35 upperQ = upperSoC * initial_cap/100
36 charging_current = charging_C_rate * initial_cap
37 discharging_current = discharging_C_rate * initial_cap
38 charge_time = (upperQ - lowerQ)/charging_current
39 if charge_time <= 0.1: # minimum 6 min. charging
40     continue
41 discharge_time = (upperQ - lowerQ)/discharging_current
42
43 first_cycle_experiment = pybamm.Experiment([
44     (f"Charge at {charging_current} A for {charge_time} hours",
45     f"Discharge at {discharging_current} A for {discharge_time} hours")]
46 )
47
48 sim = pybamm.Simulation(dfn, parameter_values = params, solver = solver,
49 experiment = first_cycle_experiment, var_pts = var_pts)
50 sol = sim.solve(initial_soc = actual_lowerSoC)
51 solution = sim.solution # completed the first cycle
52
53 exceptionBool = False
54 try: # trying next n-1 cycles (try is used to counter voltage exception)
55     for cycle_N in range(1, loop_n_cycles): # looping through slice
56         remaining_cap = solution.summary_variables["Capacity [A.h]"][-1] # Ah
57         charging_current = charging_C_rate * remaining_cap # A
58         discharging_current = discharging_C_rate * remaining_cap # A
59         charge_time = (upperQ - lowerQ)/charging_current # h
60         discharge_time = (upperQ - lowerQ)/discharging_current # h
61
62         experiment = pybamm.Experiment([
63             (f"Charge at {charging_current} A for {charge_time} hours",
64             f"Discharge at {discharging_current} A for {discharge_time} hours")
65         ])
66
67         sim = pybamm.Simulation(dfn, parameter_values = params, solver = solver,
68 experiment = experiment, var_pts = var_pts)
69 sol = sim.solve(starting_solution = sol)
70 solution = sim.solution
71
72 if len(sol.cycles) != cycle_N + 1:
73     print("Jump to outer SoC loop (Min/max voltage exception)")
74     exceptionBool = True
75     raise Exception()
76
77 if exceptionBool == False:
78     if cycle_N + 1 == loop_n_cycles:
79         cap = solution.summary_variables["Capacity [A.h]"][-1]
80         print(f"Last completed cycles: 1-{loop_n_cycles} with SoC: ",
81 actual_lowerSoC, "-", actual_upperSoC, "and Capacity: ", cap)
82         # saving the best static SoC for cycle 1-N
83         if cap > cap_max:
84             print("New max cap: ", cap)
85             sol_max = sol
86             cap_max = cap
87             solution_max = solution
88             actual_lower_SoC_max = actual_lowerSoC
89             actual_upper_SoC_max = actual_upperSoC
90
91 except Exception:
92     exceptionBool = False
93     continue
94
95 # first 10 cycles done, saving vars
96 caps.append(cap_max)
97 sols.append(sol_max)
98 actual_lowerSoC_maxs.append(actual_lower_SoC_max)

```

```

94 actual_upperSoC_maxs.append(actual_upper_SoC_max)
95 print("First slice of cycles are completed. Best SoC saved:", actual_lowerSoC_maxs,
    actual_upperSoC_maxs, "Capacity: ", caps)
96
97 # Simulating the next cycle slices
98 for j in range(1, N_split+1):
99     actual_completed_cycle_count = j*loop_n_cycles
100     cap_max = 0
101     previous_lower_Q = actual_lowerSoC_maxs[-1]
102     previous_max_sol = sols[-1] # saving previous_max_sol to use in loop
103     remaining_cap = caps[-1] # saving previous max capacity
104     for lowerSoC in range(loop_from_lower_soc, int(max_upper_loop_SoC), 2):
105         actual_lowerSoC = lowerSoC/100
106         actual_upperSoC = (lowerSoC + range_SoC)/100
107         upperSoC = lowerSoC + range_SoC
108         lowerQ = lowerSoC * remaining_cap/100
109         upperQ = upperSoC * remaining_cap/100
110         charging_current = charging_C_rate * remaining_cap
111         discharging_current = discharging_C_rate * remaining_cap
112
113         # start on previous SoC
114         charge_time = (upperQ - previous_lower_Q)/charging_current
115         if charge_time <= 0.1: # minimum 6 min. charging
116             continue
117         discharge_time = (upperQ - lowerQ)/discharging_current
118
119         experiment = pybamm.Experiment([
120             (f"Charge at {charging_current} A for {charge_time} hours",
121              f"Discharge at {discharging_current} A for {discharge_time} hours")]
122
123         sim = pybamm.Simulation(dfn, parameter_values = params, solver = solver,
124                                experiment = experiment, var_pts = var_pts)
125         # starting on previous best sol
126         sol = sim.solve(starting_solution = previous_max_sol) # previous best
127         # using sol for the next N-1 cycles
128         solution = sim.solution
129
130         if len(sol.cycles) != actual_completed_cycle_count+1:
131             print("Jump to SoC loop start (Min/max voltage exception)")
132             continue
133
134         exceptionBool = False
135         try: # trying to do the next slice
136             for cycle_N in range(2, loop_n_cycles + 1):
137                 remaining_cap = solution.summary_variables["Capacity [A.h]"][-1]
138                 charging_current = charging_C_rate * remaining_cap # A
139                 discharging_current = discharging_C_rate * remaining_cap # A
140                 new_lowerQ = actual_lowerSoC * remaining_cap # Ah
141                 new_upperQ = actual_upperSoC * remaining_cap # Ah
142                 charge_time = (new_upperQ - new_lowerQ)/charging_current # h
143                 discharge_time = (new_upperQ - new_lowerQ)/discharging_current # h
144
145                 experiment = pybamm.Experiment([
146                     (f"Charge at {charging_current} A for {charge_time} hours",
147                      f"Discharge at {discharging_current} A for {discharge_time}
148 hours")]
149
150                 sim = pybamm.Simulation(dfn, parameter_values = params, solver =
151                                solver, experiment = experiment, var_pts = var_pts)
152                 sol = sim.solve(starting_solution = sol)
153                 solution = sim.solution
154
155                 if len(sol.cycles) != actual_completed_cycle_count + cycle_N:
156                     print("Jump to outer SoC loop (Min/max voltage exception)")

```

```
154         exceptionBool = True
155         raise Exception()
156
157     if exceptionBool == False:
158         if cycle_N == loop_n_cycles:
159             cap = solution.summary_variables["Capacity [A.h]"][-1]
160             print(f"Completed cycles: {actual_completed_cycle_count}-{
actual_completed_cycle_count+10} with SoC: ", actual_lowerSoC,"-",
actual_upperSoC,"and Capacity: ", cap)
161
162             if cap > cap_max: # saving the best static SoC for cycle 1-N
163                 print("New max cap: ", cap)
164                 sol_max = sol
165                 cap_max = cap
166                 solution_max = solution
167                 actual_lowerSoC_max = actual_lowerSoC
168                 actual_upperSoC_max = actual_upperSoC
169
170     except Exception:
171         exceptionBool = False
172         continue
173
174     # saving vars for the next loop_n_cycles cycles
175     caps.append(cap_max)
176     sols.append(sol_max)
177     actual_lowerSoC_maxs.append(actual_lowerSoC_max)
178     actual_upperSoC_maxs.append(actual_upperSoC_max)
179     print("Next slice of cycles are completed. Best SoC saved:",
actual_lowerSoC_maxs, actual_upperSoC_maxs, "Capacity: ", caps)
180 print("All cycles completed.")
```

