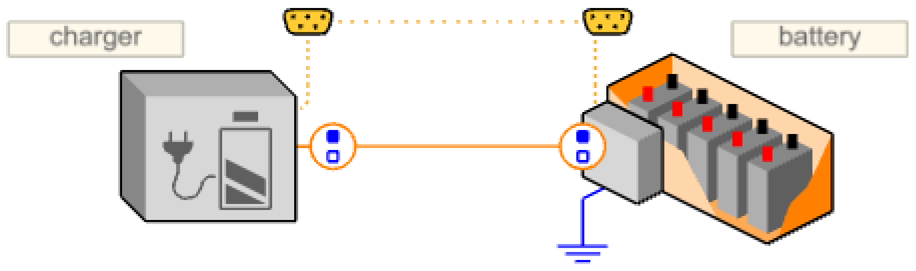
Monte Carlo simulation with stochastic parameters

Application: Battery charging with cell imbalances

This notebook demonstrates an analysis of imbalances between cells in a battery pack during charging. This involves using stochastic parameters for modeling statistical variations and uncertainties in the model, and analysing these with a Monte Carlo approach over a large batch of simulations.

This analysis is based on the *CellBalancing* model from the *Electrification Library*. More information about this model can be found in the library documentation via the *Modelon Impact* GUI (path: Electrification.Batteries.Experiments.Imbalances.CellBalancing).



Configure a workspace

Before running this notebook:

1. Create and configure a workspace that includes the Electrification Library.
2. **Edit** the *workspace_name* below to the name of your workspace.

```
In [1]: # EDIT THIS!

        workspace_name = "el2024p1"
```

(No more changes to the code are needed after this point)

Libraries and routines

We need to import the *Modelon Impact* **Client**, which we use to compile and simulate models with the *Modelon Impact* server. The **Range** functionality is also needed for defining batch simulations. We also import additional Python dependencies for data processing and plotting.

A routine is also defined for providing us with status about ongoing operations on the Modelon Impact server.

In [2]:
```
# Import Python libraries
from modelon.impact.client import Client
from modelon.impact.client import Range
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sys import stdout
from time import sleep

# Define a status routine
def monitor_operation(operation):
    tic = 0
    while not operation.is_complete():
        stdout.write(str(operation.status) + " (" + str(tic) + ")\r")
        tic = tic + 1
        sleep(1)
    print(str(operation.status) + " (" + str(tic) + ")             ")
    return operation.data()
```

Connect to *Modelon Impact* and get the workspace and model

Start by connecting to the *Modelon Impact* server and get the specific workspace and model.

In [3]:
```
# Create a client object and authenticate
client = Client()

# Get the workspace
workspace = client.get_workspace(workspace_name)

# Get the model
model = workspace.get_model("Electrification.Batteries.Experiments.Imbalances.CellB
```

Define a dynamic simulation experiment

We define an experiment for a dynamic simulation, that we will use throughout the notebook. This defines the length of the simulation and the result output frequency. We also define an output filter to store only the variables that we are interested in. This reduces the ammount of data to store (extra useful when doing a large number of simulations).

In [4]:
```
T_END = 2400.0 # Simulation end time (seconds)
NCP = 500 # Number of Communication Points (simulation output points)
filter = "[\"battery.summary*\",\"battery.core*.summary*\"]"

# Define a dynamic simulation and options
dynamic = workspace.get_custom_function('dynamic')
```

```
simulation_options = dynamic.get_simulation_options().with_values(
    filter=filter,
    ncp=NCP)
solver_options = {'store_event_points':True}

# Create a simulation experiment object
experiment_definition = model.new_experiment_definition(
    dynamic.with_parameters(start_time=0.0, final_time=T_END),
    simulation_options=simulation_options,
    solver_options=solver_options)

# Set model parameters (disable random parameters)
NS = 6 # number of cells in series
NP = 3 # number of cells in parallel
model_parameters = {
    'battery.ns': NS,
    'battery.np': NP}
experiment_definition = experiment_definition.with_modifiers(model_parameters)
```

Compile the model

We compile the model to a binary (FMU) that we can simulate. Note that we only need to do this once. Since the model allows us to change parameter values after compilation, we can re-use it for several simulations.

In [5]:
```
# Compile the model into a binary fmu, with default options
operation = model.compile(
    compiler_options=dynamic.get_compiler_options(),
    runtime_options=dynamic.get_runtime_options())
fmu = monitor_operation(operation)
```

```
Status.DONE (0)
```

Simulation case 1: A nominal battery

In the first case, we simulate a battery pack where all the cells are have identical parameters (in this case, it would have been sufficient to simulate a single cell and scale the result to represent all of them).

We configure this by disabling the random parameter functionality in the model. This means that the stochastic parameters in the model will not be generated randomly, but instead use their nominal (expected) value.

In [6]:
```
# Set model parameters (disable random parameters)
model_parameters['globalRandom.random'] = False
experiment_definition = experiment_definition.with_modifiers(model_parameters)

# Simulate
operation = workspace.execute(experiment_definition)
exp = monitor_operation(operation)
```

```
# Get result trajectories
case1 = exp.get_case('case_1')
res1 = case1.get_trajectories()
```

```
Status.DONE (1)
```

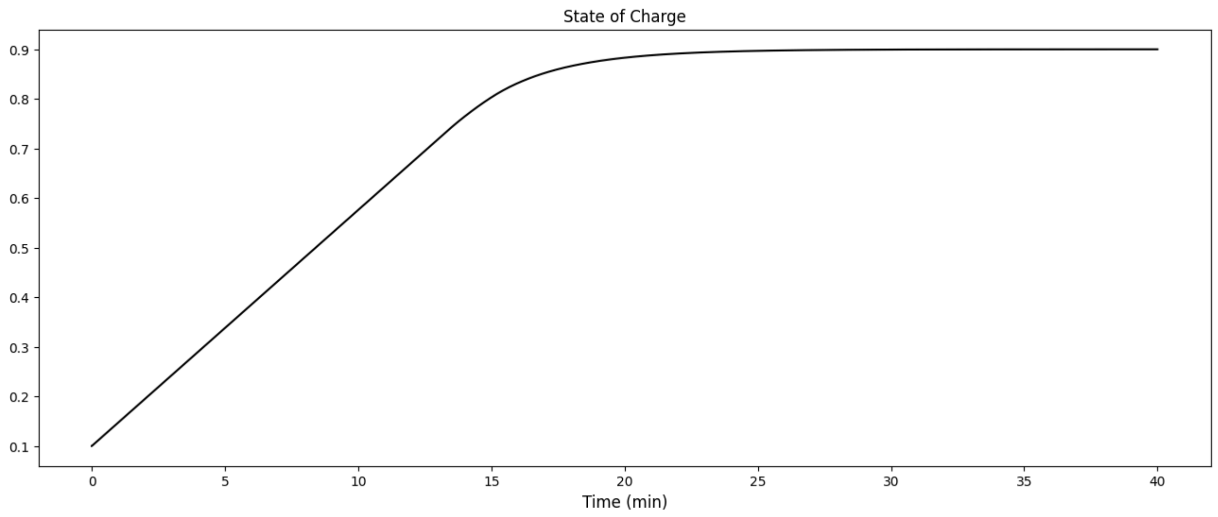
Plot results

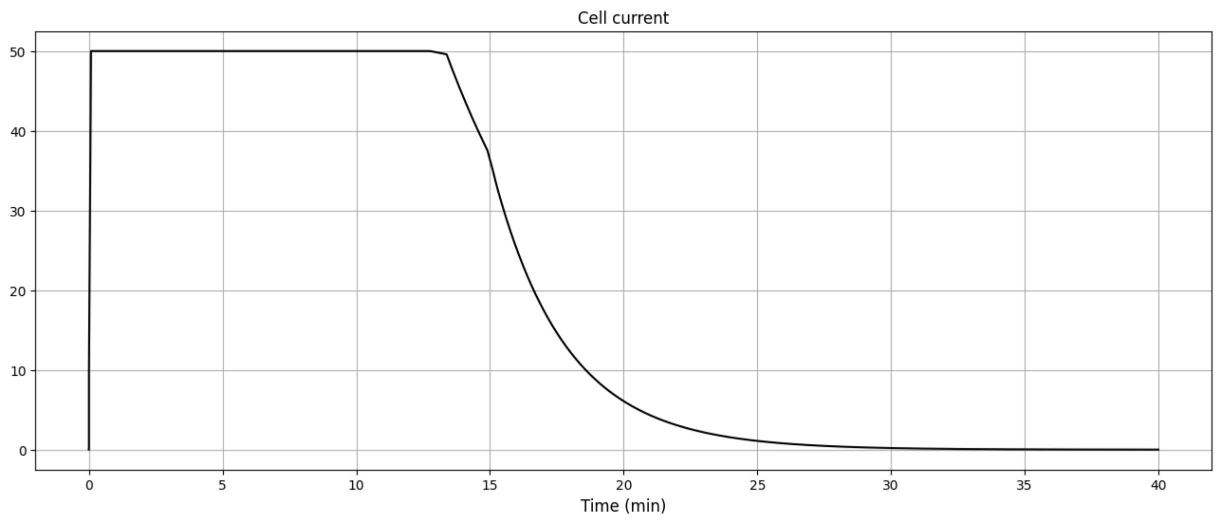
We can see that the battery is charged up from an initial 10 % SoC (State of Charge) to the specified limit of 90 %. We can also observe the cell current reaches the specified 50 A limit during the first part of the charging process, and then drops of towards the second part.

In [7]:
```
# Time vector (in minutes)
res1TimeMin = np.array(res1['time'])/60

# Plot SoC
plt.figure(11,figsize=(16, 6))
plt.plot(res1TimeMin, res1['battery.summary.SoC'],'k')
plt.xlabel('Time (min)', fontsize=12)
plt.title('State of Charge', fontsize=12)
plt.show()
plt.close(11)

# Plot currents
plt.figure(12,figsize=(16, 6))
plt.plot(res1TimeMin, res1['battery.summary.i_cell_max'],'k')
plt.xlabel('Time (min)', fontsize=12)
plt.title('Cell current', fontsize=12)
plt.grid()
plt.show()
plt.close(12)
```





Simulation case 2: Imbalances

Let us now consider that there are uncertainties parameters of the battery cells. The model parameters for charge capacity and resistance have been defined as statistical distributions. We can use this to randomly select a set of cells with parameters that vary accordingly.

We perform a new simulation of the same experiment. But first we enable random initialization of these parameters. We also disable the cell balancing functionality of the battery pack (this will be enabled later).

In [8]:
```
# Change model parameters
model_parameters['globalRandom.random'] = True # Enable random parameter initializa
model_parameters['globalRandom.seed'] = 123 # Initial seed for the random number ge
model_parameters['battery.controller.balancing.socDiff'] = 1.0 # No cell balancing
experiment_definition = experiment_definition.with_modifiers(model_parameters)

# Simulate
operation = workspace.execute(experiment_definition)
exp = monitor_operation(operation)

# Get results
case2 = exp.get_case('case_1')
res2 = case2.get_trajectories()
```

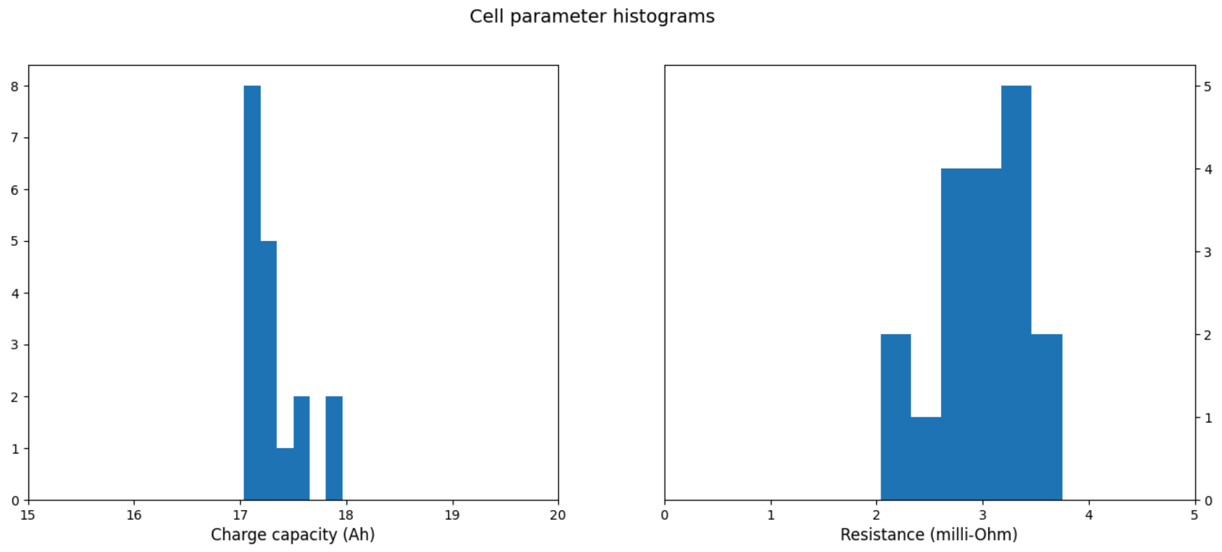
```
Status.DONE (1)
```

Parameter variations

We will now plot histograms of the parameter values for the cells in the pack. We can see that the charge capacity varies between 17 and 18 Ah among the cells. And the cell resistances are somewhat normally distributed with a mean of 0.003 Ohm.

In [9]:
```
cellResMilli = np.zeros(NS*NP)
cellCapAh = np.zeros(NS*NP)
for xs in range(0,NS,1):
    for xp in range(0,NP,1):
        cellResMilli[xs+xp*NS] = np.array(res2[f'battery.core[{(1+xs):},{(1+xp):}].
        cellCapAh[xs+xp*NS] = np.array(res2[f'battery.core[{(1+xs):},{(1+xp):}].sum

gridsize = (1, 2)
fig = plt.figure(20,figsize=(16, 6))
ax1 = plt.subplot2grid(gridsize, (0, 0))
ax2 = plt.subplot2grid(gridsize, (0, 1))
ax1.hist(cellCapAh, bins='auto')
ax2.hist(cellResMilli, bins='auto')
ax1.set_xlim([15,20])
ax2.set_xlim([0,5])
ax1.set_xlabel('Charge capacity (Ah)', fontsize=12)
ax2.set_xlabel('Resistance (milli-Ohm)', fontsize=12)
ax2.yaxis.tick_right()
fig.suptitle('Cell parameter histograms', fontsize=14)
plt.show()
plt.close(20)
```



Plot results

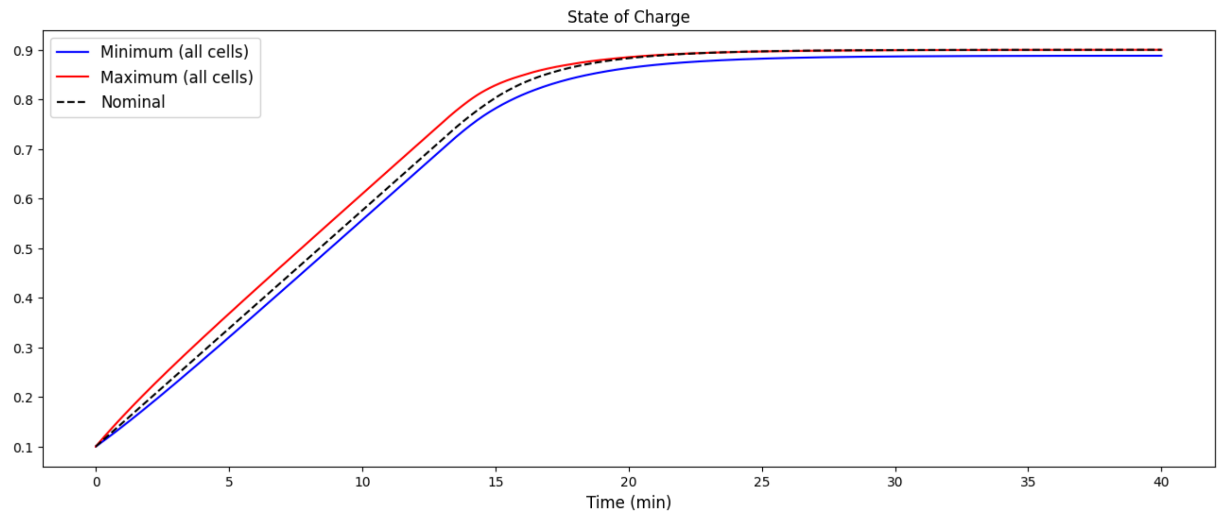
We now look at the simulation result compared to the nominal scenario where all cells have identical parameters.

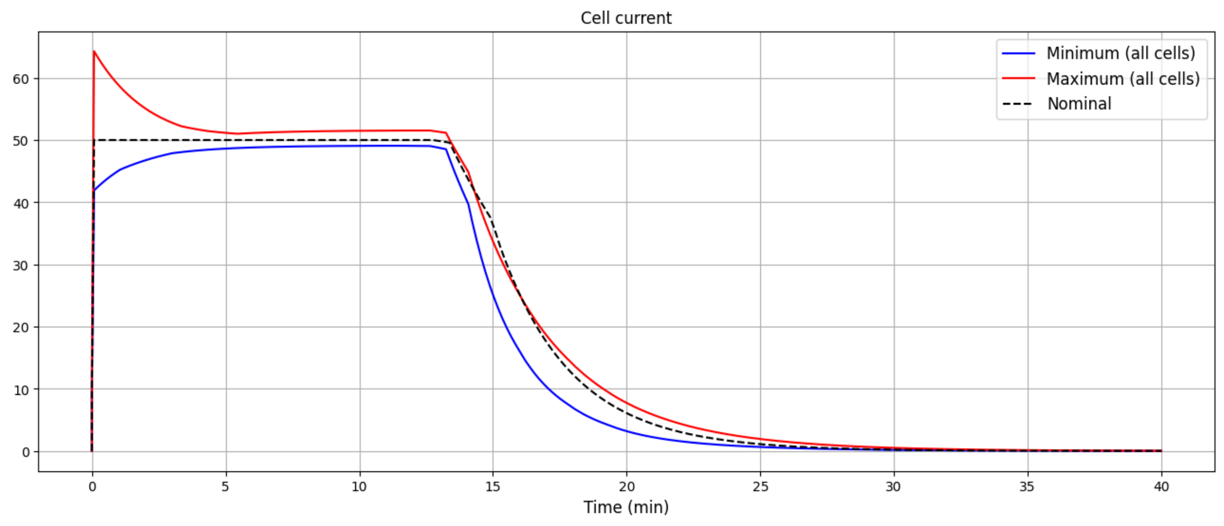
The plots below show both the maximum and minimum values for the SoC and current among all of the cells. This is also compared to the nominal results from the first simulation. We note that some of the cells reach the target 90 % SoC, but not all of them. The charging has stopped as the highest charged cells have reached the limit. We also note that the current through some cells reach much higher than the 50 A limit.

In [10]:
```
# Prepare
res2TimeMin = np.array(res2['time'])/60

# Plot
plt.figure(21,figsize=(16, 6))
plt.plot(res2TimeMin, res2['battery.summary.SoC_min'],'b')
plt.plot(res2TimeMin, res2['battery.summary.SoC_max'],'r')
plt.plot(res1TimeMin, res1['battery.summary.SoC'],'k--')
plt.legend([
    'Minimum (all cells)',
    'Maximum (all cells)',
    'Nominal'], fontsize=12)
plt.xlabel('Time (min)', fontsize=12)
plt.title('State of Charge', fontsize=12)
plt.show()
plt.close(21)

plt.figure(22,figsize=(16, 6))
plt.plot(res2TimeMin, res2['battery.summary.i_cell_min'],'b')
plt.plot(res2TimeMin, res2['battery.summary.i_cell_max'],'r')
plt.plot(res1TimeMin, res1['battery.summary.i_cell_max'],'k--')
plt.legend([
    'Minimum (all cells)',
    'Maximum (all cells)',
    'Nominal'], fontsize=12)
plt.xlabel('Time (min)', fontsize=12)
plt.title('Cell current', fontsize=12)
plt.grid()
plt.show()
plt.close(22)
```





KPIs

We can define two KPIs for the cell SoC and current, to quantify the deviations from the nominal results as we continue the analysis.

In [11]:
```
socDelta = res2['battery.summary.SoC_max'][-1]-res2['battery.summary.SoC_min'][-1]
iMax = max(res2['battery.summary.i_cell_max'])
print(f'{socDelta:.1%} delta SOC after {T_END/60:.0f} minutes')
print(f'{iMax:.1f} A maximum cell current')
```

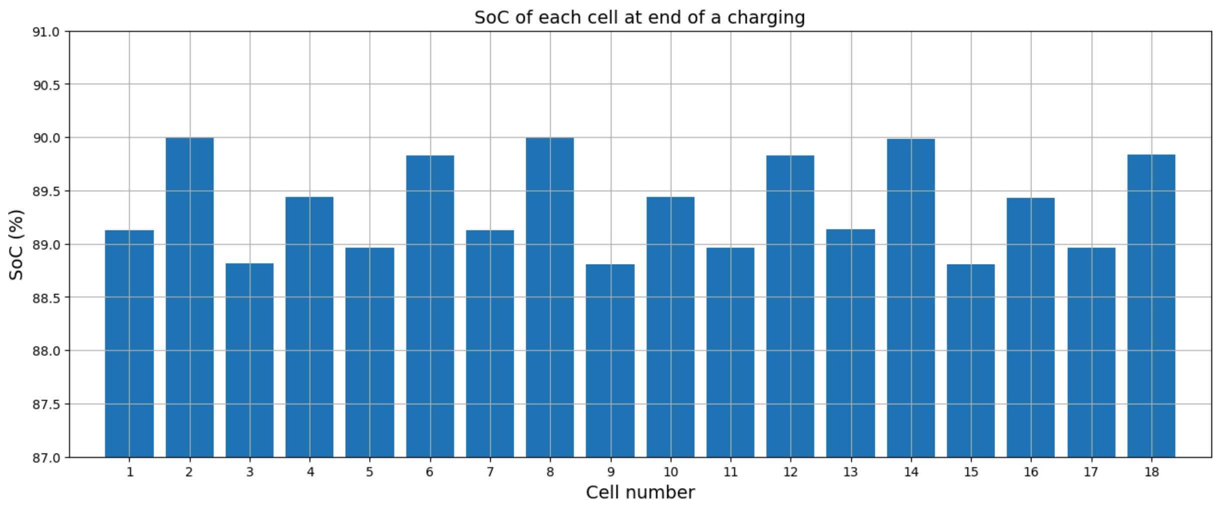
```
1.2% delta SOC after 40 minutes
64.3 A maximum cell current
```

SoC of each cell

The plot below shows the SoC of each cell at the end of the simulation. We can see that some of the cells have reached 90 %, while some are much lower.

In [12]:
```
socCell = np.zeros(NS*NP)
for xs in range(0,NS,1):
    for xp in range(0,NP,1):
        socCell[xs+xp*NS] = res2[f'battery.core[{(1+xs):},{(1+xp):}].summary.SoC'][

plt.figure(23,figsize=(16, 6))
plt.bar(range(1,NS*NP+1,1),100*np.array(socCell))
plt.ylim([87,91])
#plt.ylim([89.7,90])
plt.xlim([0,NS*NP+1])
plt.xticks(range(1,NS*NP+1,1))
plt.xlabel('Cell number', fontsize=14)
plt.ylabel('SoC (%)', fontsize=14)
plt.title(f'SoC of each cell at end of a charging', fontsize=14)
plt.grid()
plt.show()
plt.close(23)
```



Simulation case 3: Cell balancing

Now let us enable the cell balancing functionality of this battery pack, and perform the same simulation again. We configure the balancing controller to be active until the difference in SoC between the cells is less than 0.2 %.

Note that we will get the same parameters values as in the previous simulation, because we are specifying the same *seed* for the random generator. This also allows us to reproduce the results if we need to.

This simulation may be a bit slower, since the balancing control introduce a lot of dynamics in the system (as the results will show).

In [13]:
```
# Model parameters
model_parameters['battery.controller.balancing.socDiff'] = 0.002 # 0.2 % SoC differ
experiment_definition = experiment_definition.with_modifiers(model_parameters)

# Simulate
operation = workspace.execute(experiment_definition)
exp = monitor_operation(operation)

# Get results
case3 = exp.get_case('case_1')
res3 = case3.get_trajectories()
```

```
Status.DONE (1)
```

Plot results

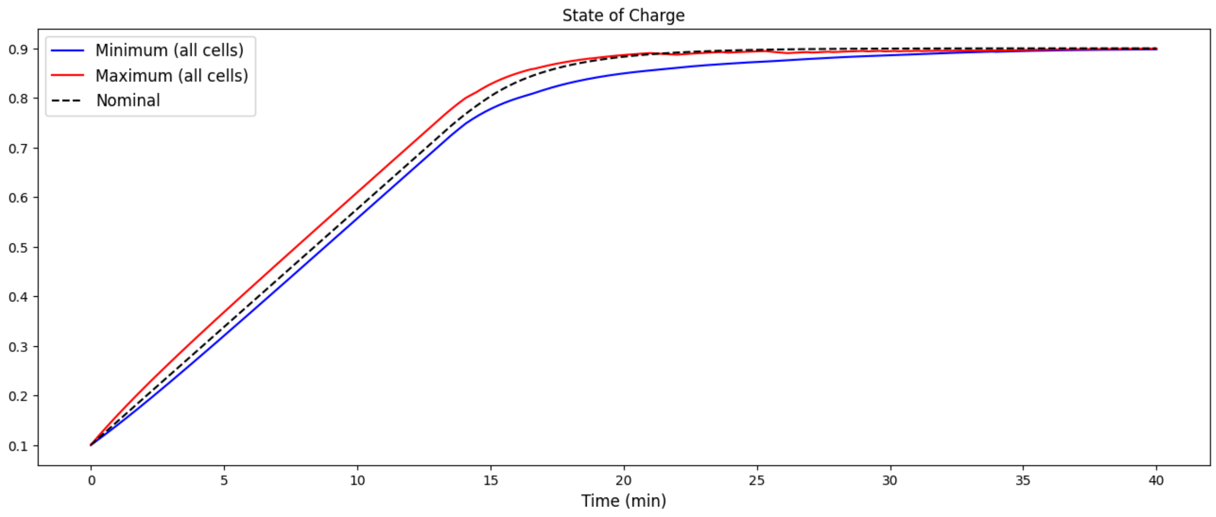
When we now plot the SoC of the cells, we can see that they all reach 90 % at the end of the simulation.

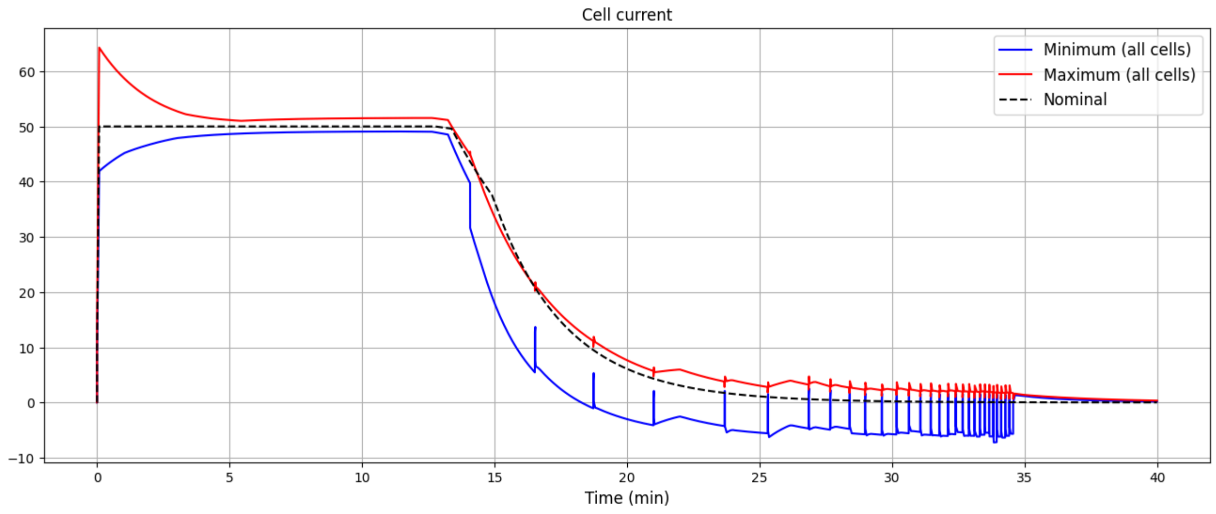
And if when we plot the cell currents, we can see that there is a lot of dynamics during the second part of the charging process, when the balancing is active, "bleeding" energy from the cells that are charged the most, to allow the lowest charged cells to catch up.

In [14]:
```
# Prepare
res3TimeMin = np.array(res3['time'])/60

# Plot
plt.figure(31,figsize=(16, 6))
plt.plot(res3TimeMin, res3['battery.summary.SoC_min'],'b')
plt.plot(res3TimeMin, res3['battery.summary.SoC_max'],'r')
plt.plot(res1TimeMin, res1['battery.summary.SoC'],'k--')
plt.legend([
    'Minimum (all cells)',
    'Maximum (all cells)',
    'Nominal'], fontsize=12)
plt.xlabel('Time (min)', fontsize=12)
plt.title('State of Charge', fontsize=12)
plt.show()
plt.close(31)

plt.figure(32,figsize=(16, 6))
plt.plot(res3TimeMin, res3['battery.summary.i_cell_min'],'b')
plt.plot(res3TimeMin, res3['battery.summary.i_cell_max'],'r')
plt.plot(res1TimeMin, res1['battery.summary.i_cell_max'],'k--')
plt.legend([
    'Minimum (all cells)',
    'Maximum (all cells)',
    'Nominal'], fontsize=12)
plt.xlabel('Time (min)', fontsize=12)
plt.title('Cell current', fontsize=12)
plt.grid()
plt.show()
plt.close(32)
```





KPIs

The SoC delta has now decreased to the specified 0.2 %. Also note that the maximum cell current has not changed, since we used the same parameters as the previous simulation (we used the same initial seed for the random generator).

In [15]:
```
socDelta = res3['battery.summary.SoC_max'][-1]-res3['battery.summary.SoC_min'][-1]
iMax = max(res3['battery.summary.i_cell_max'])
print(f'{socDelta:.1%} delta SOC after {T_END/60:.0f} minutes')
print(f'{iMax:.1f} A maximum cell current')
```

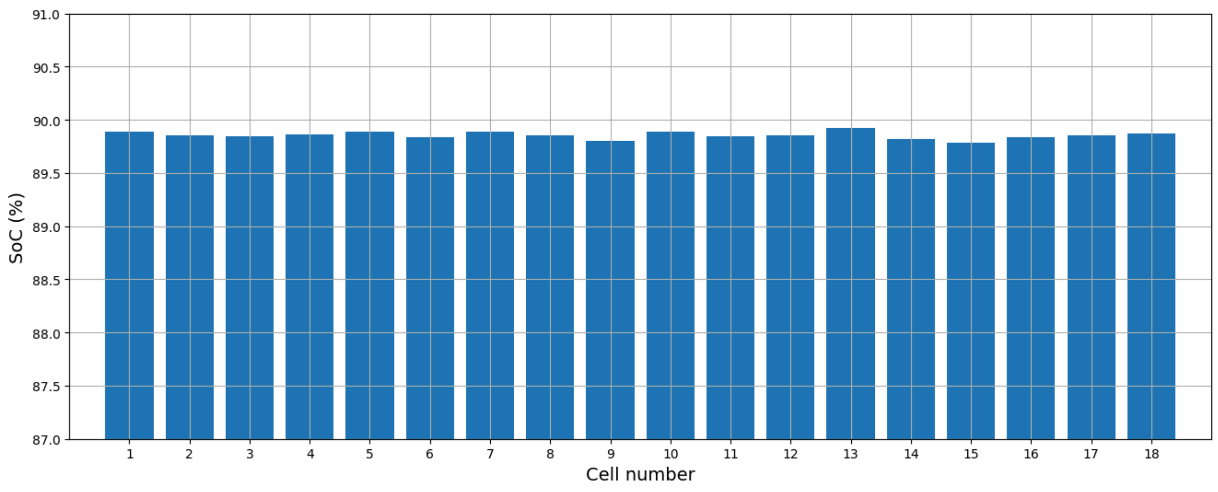
```
0.1% delta SOC after 40 minutes
64.3 A maximum cell current
```

SoC of each cell

We also plot the SoC of all cells at the end of the simulation.

In [16]:
```
socCell = np.zeros(NS*NP)
for xs in range(0,NS,1):
    for xp in range(0,NP,1):
        socCell[xs+xp*NS] = res3[f'battery.core[{(1+xs):},{(1+xp):}].summary.SoC'][

plt.figure(23,figsize=(16, 6))
plt.bar(range(1,NS*NP+1,1),100*np.array(socCell))
plt.ylim([87,91])
#plt.ylim([89.7,90])
plt.xlim([0,NS*NP+1])
plt.xticks(range(1,NS*NP+1,1))
plt.xlabel('Cell number', fontsize=14)
plt.ylabel('SoC (%)', fontsize=14)
plt.grid()
plt.show()
plt.close(23)
```



Simulation case 4: Lots of different individuals (Monte Carlo)

Finally, let's evaluate how much the results vary between different battery packs, that all have the same cell parameter distributions. The results may be different for a different set of cell samples. If we have a large enough number of simulations, we get an approximation of the statistical result for a general battery pack with these specific parameter uncertainties.

We can create such a *Monte Carlo* experiment by performing the same simulation a lot of times, where each simulation has a different seed for the random number generator. This gives us a unique collection of cells for each simulation. We do this by defining the *seed* parameter with the **Range** operator instead of a scalar value.

In [17]:
```
# Define the experiment
NUM_CASES = 500

# Adjust experiment to provide the same number of data points
experiment_definition = model.new_experiment_definition(
    dynamic.with_parameters(start_time=0.0, final_time=T_END),
    simulation_options=simulation_options,
    solver_options={'store_event_points':False})

# Model parameters
model_parameters['globalRandom.seed'] = Range(1, NUM_CASES, NUM_CASES)
experiment_definition = experiment_definition.with_modifiers(model_parameters)

# Simulate
operation = workspace.execute(experiment_definition)
exp = monitor_operation(operation)
```

```
Status.DONE (98)
```

Post-processing

The following code calculates the KPI statistics for all simulation cases, and generates min/max results for the collection of all results.

In [18]:
```
stdout.write('Loading results ...\r\n')
stdout.flush()

res = exp.get_trajectories([
    'time',
    'battery.summary.SoC_max',
    'battery.summary.SoC_min',
    'battery.summary.i_cell_max',
    'battery.summary.i_cell_min'])

nSamples = len(res['case_1']['time'])
timeMin = np.array(res['case_1']['time'])/60

socDeltaPctM = []
iMaxM = []
socMaxMaxM = -1e20*np.ones(nSamples)
socMaxMinM =  1e20*np.ones(nSamples)
socMinMaxM = -1e20*np.ones(nSamples)
socMinMinM =  1e20*np.ones(nSamples)
iMaxMaxM = -1e20*np.ones(nSamples)
iMaxMinM =  1e20*np.ones(nSamples)
iMinMaxM = -1e20*np.ones(nSamples)
iMinMinM =  1e20*np.ones(nSamples)
```

```
cR = 0
for cN in res.keys():
    case = res[cN]
    socDeltaPct = 100*(case['battery.summary.SoC_max'][-1]-case['battery.summary.So
    iMax = max(case['battery.summary.i_cell_max'])
    socDeltaPctM.append(socDeltaPct)
    iMaxM.append(iMax)
    socMaxMaxM = np.maximum(socMaxMaxM,case['battery.summary.SoC_max'])
    socMaxMinM = np.minimum(socMaxMinM,case['battery.summary.SoC_max'])
    socMinMaxM = np.maximum(socMinMaxM,case['battery.summary.SoC_min'])
    socMinMinM = np.minimum(socMinMinM,case['battery.summary.SoC_min'])
    iMaxMaxM = np.maximum(iMaxMaxM,case['battery.summary.i_cell_max'])
    iMaxMinM = np.minimum(iMaxMinM,case['battery.summary.i_cell_max'])
    iMinMaxM = np.maximum(iMinMaxM,case['battery.summary.i_cell_min'])
    iMinMinM = np.minimum(iMinMinM,case['battery.summary.i_cell_min'])
    cR = cR + 1
    stdout.write(f'Calculating statistics ... {cR:}/{len(res):}\r')
    stdout.flush()
```

```
Loading results ...
Calculating statistics ... 500/500
```

Plot results

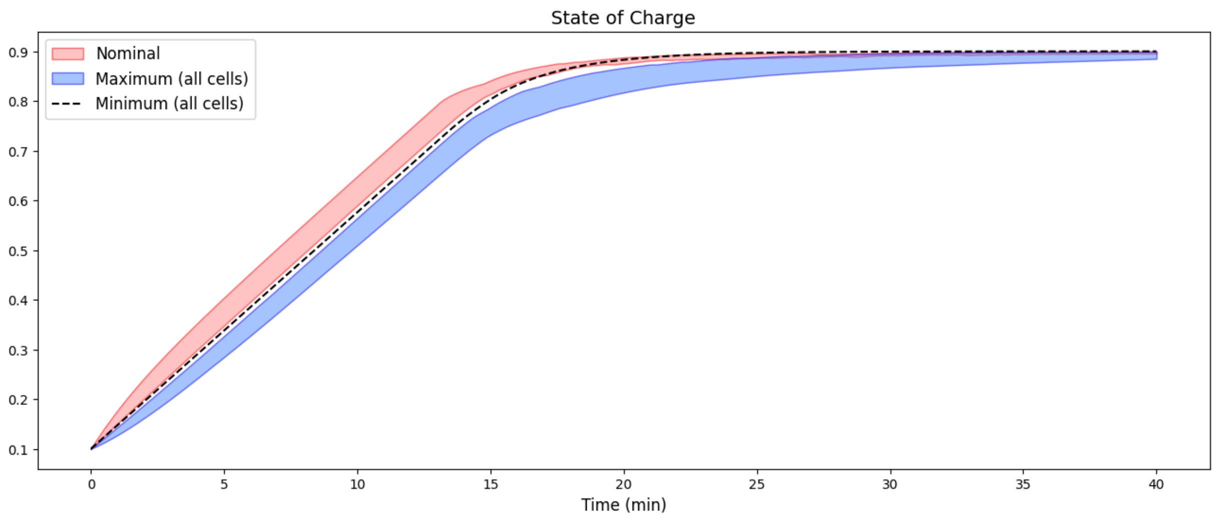
Instead of plotting the trajectories for all of the simulations, we plot an area that show the bounds for each variable over all simulations.

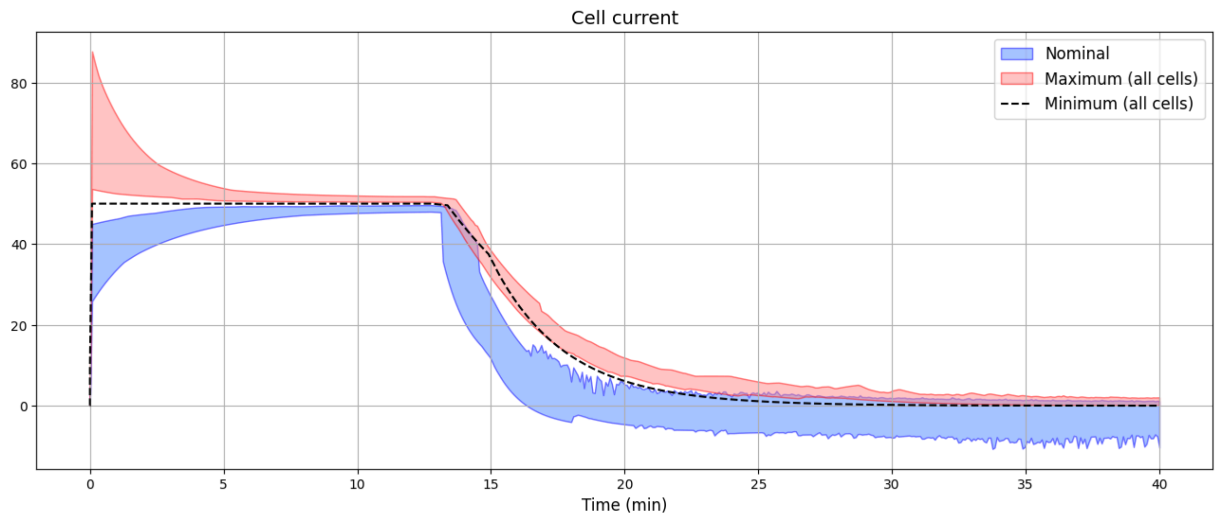
When we look at the SoC, we can see that there is some variation in the results. And we especially note that the minimum SoC no longer reach all the way up to the 90 % target. There is some spread between the simulation cases here. Some cases reach the target, while others fall short of it.

We also see a spread in the results when we look at the maximum and minimum cell currents. We conclude that there are cases that are even worse than the 65 A we saw in the previous single simulation case.

In [19]:
```
# Plot
plt.figure(51,figsize=(16, 6))
plt.fill_between(timeMin,socMaxMinM, socMaxMaxM,
    alpha=0.5, edgecolor='#FF0000', facecolor='#FF8888',linewidth=1, antialiased=Tr
plt.fill_between(timeMin,socMinMinM, socMinMaxM,
    alpha=0.5, edgecolor='#0000FF', facecolor='#5588FF',linewidth=1, antialiased=Tr
plt.plot(res1TimeMin, res1['battery.summary.SoC'],'k--')
plt.title('State of Charge', fontsize=14)
plt.legend([
    'Nominal',
    'Maximum (all cells)',
    'Minimum (all cells)'], fontsize=12)
plt.xlabel('Time (min)', fontsize=12)
plt.show()
plt.close(51)

plt.figure(52,figsize=(16, 6))
plt.fill_between(timeMin,iMinMinM, iMinMaxM,
    alpha=0.5, edgecolor='#0000FF', facecolor='#5588FF',linewidth=1, antialiased=Tr
plt.fill_between(timeMin,iMaxMinM, iMaxMaxM,
    alpha=0.5, edgecolor='#FF0000', facecolor='#FF8888',linewidth=1, antialiased=Tr
plt.plot(res1TimeMin, res1['battery.summary.i_cell_max'],'k--')
plt.title('Cell current', fontsize=14)
plt.legend([
    'Nominal',
    'Maximum (all cells)',
    'Minimum (all cells)'], fontsize=12)
plt.xlabel('Time (min)', fontsize=12)
plt.grid()
plt.show()
plt.close(52)
```



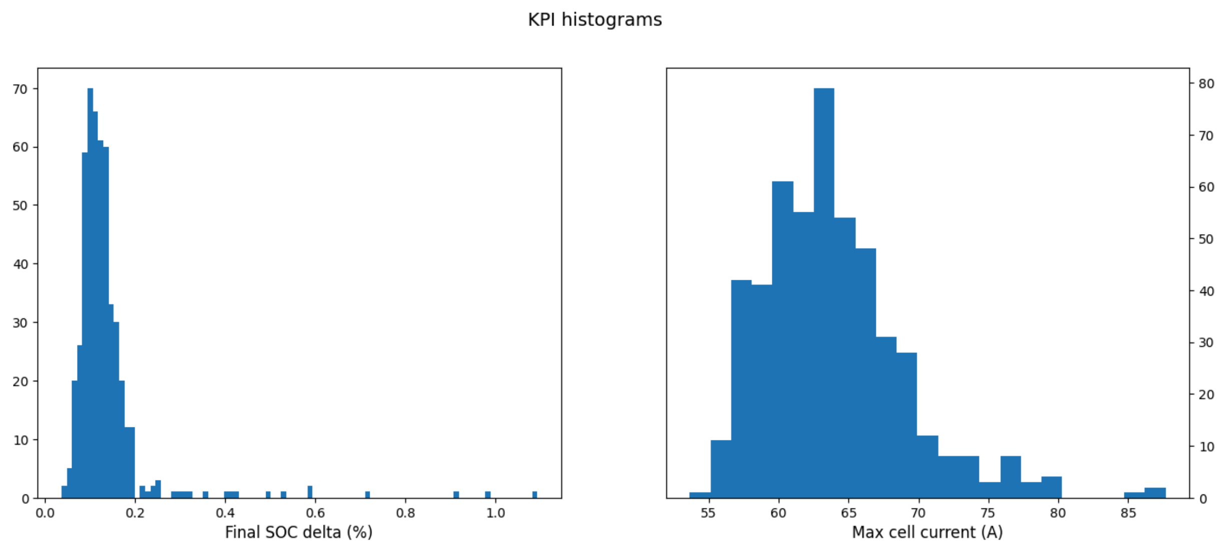


KPI histograms

We also plot histograms for the KPIs for all of the battery packs we simulated. This provides an approximations of the statistical distributions for the SoC delta and maximum cell current, which allows us to determine how common the more extreme results are. We can conclude that there are cases where the cell balancing is not successful, but in moste cases it is. And we also see that most cases are below 70 A maximum cell current.

In [20]:
```
# KPI histograms

gridsize = (1, 2)
fig = plt.figure(53,figsize=(16, 6))
ax1 = plt.subplot2grid(gridsize, (0, 0))
ax2 = plt.subplot2grid(gridsize, (0, 1))
ax1.hist(socDeltaPctM, bins='auto')
ax2.hist(iMaxM, bins='auto')
#ax1.set_title('Histogram: Final SOC delta (%)')
#ax2.set_title('Histogram: Max cell current')
ax1.set_xlabel('Final SOC delta (%)', fontsize=12)
ax2.set_xlabel('Max cell current (A)', fontsize=12)
ax2.yaxis.tick_right()
fig.suptitle('KPI histograms', fontsize=14)
plt.show()
plt.close(53)
```


KPI histograms
Final SOC delta (%)
Max cell current (A)