BPL_TEST2_Batch_calibration - demo

This notebook shows the possibilities for calibration of the model BPL_TEST2_Batch using scipy.optimize.minimize() routine. There are several different methods to choose between.

The text-book model of batch cultivation we simulate is the following where S is substrate, X is cell concentration, and V is volume of the broth

$$\frac{d(VS)}{dt} = -q_S(S) \cdot VX$$
$$\frac{d(VX)}{dt} = \mu(S) \cdot VX$$

and where specific cell growth rate μ and substrate uptake rate q_S are

$$\mu(S) = Y \cdot q_S(S)$$

$$q_S(S) = q_S^{max} \frac{S}{K_s + S}$$

where Y is the yield, q_S^{max} is the maximal specific substrate uptake rate and K_s is the corresponding saturation constant.

The parameter estimation is done with optimization methods that only require evaluation of the missmatch between simulation with given parameters and data. At start the allowed range for each parameter is given. The method used for optimization is SLQSP but can easily be changed.

In the near future the FMU may provide first derivative gradient informaion, that will make it possible to choose corresponding method of minimize() for improved performance. This possibility is related to the upgrade to the FMI-standard ver 3.0 for the Modelica compiler.

The Python package PyFMI that is the base for FMU-explore has a simplified built-in functionality for parameter estimation that also use scipy.optimize.minimize(). However, there is no possibility to include parameter changes to the compiled model that should not be estimated and the purpose seems to only address smaller examples. Therefore we here define a Python function evaluation() that facilitate the formulation of the parameter estimation and bring flexibility to choice of optimization method.

In [1]:
```
run -i BPL_TEST2_Batch_explore.py
```

```
Windows - run FMU pre-compiled JModelica 2.14

Model for bioreactor has been setup. Key commands:
 - par()       - change of parameters and initial values
 - init()      - change initial values only
 - simu()      - simulate and plot
 - newplot()   - make a new plot
 - show()      - show plot from previous simulation
 - disp()      - display parameters and initial values from the last simulation
 - describe()  - describe culture, broth, parameters, variables with values / units

Note that both disp() and describe() takes values from the last simulation
```

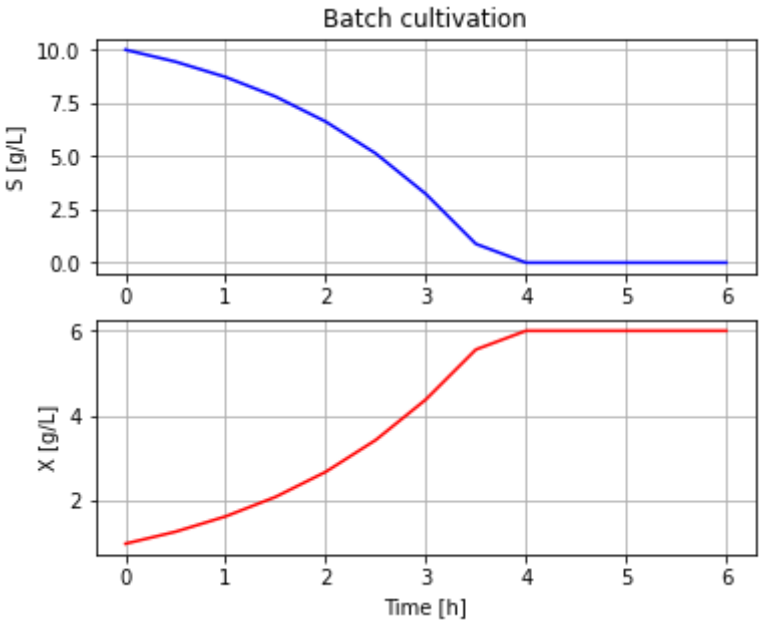
```
Brief information about a command by help(), eg help(simu)
Key system information is listed with the command system_info()
```

In [2]:
```
# Adjust the size of diagrams
plt.rcParams['figure.figsize'] = [15/2.54, 12/2.54]
```

Generate data that later used for parameter estimation

In [3]:
```
import pandas as pd
```

In [4]:
```
# Data generated
par(Y=0.5, qSmax=1.0, Ks=0.1)
init(V_0=1.0, VS_0=10, VX_0=1.0)
newplot(plotType='Demo_1')
opts['ncp'] = 12
simu(6)
```



In [5]:
```
# Store data in a DataFrame for later use
data = pd.DataFrame(data={'time':sim_res['time'], 'X':sim_res['bioreactor.c[1]'], 'S
data
```

Out[5]:

	time	X	S
0	0.0	1.000000	1.000000e+01
1	0.5	1.280773	9.438453e+00
2	1.0	1.640079	8.719842e+00
3	1.5	2.099615	7.800770e+00
4	2.0	2.686770	6.626459e+00
5	2.5	3.435479	5.129043e+00
6	3.0	4.385325	3.229350e+00
7	3.5	5.559252	8.814967e-01

	time	X	S
8	4.0	6.000000	1.048375e-08
9	4.5	6.000000	-1.936268e-10
10	5.0	6.000000	2.156125e-12
11	5.5	6.000000	9.975889e-14
12	6.0	6.000000	4.189854e-15

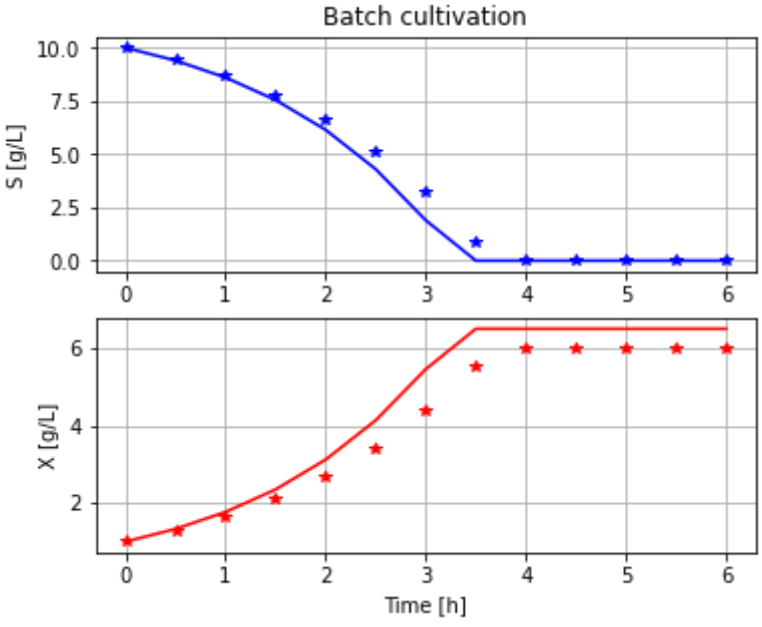
Simulation with initial guess of parameters compared with data

Here we define the parameters that should be estimated and specify allowed ranges. Nominal parameters are chosen as the mid-point of the allowed parameter range.

Simulation with these nominal parameter set and compare with data give an idea of who well the model fit data.

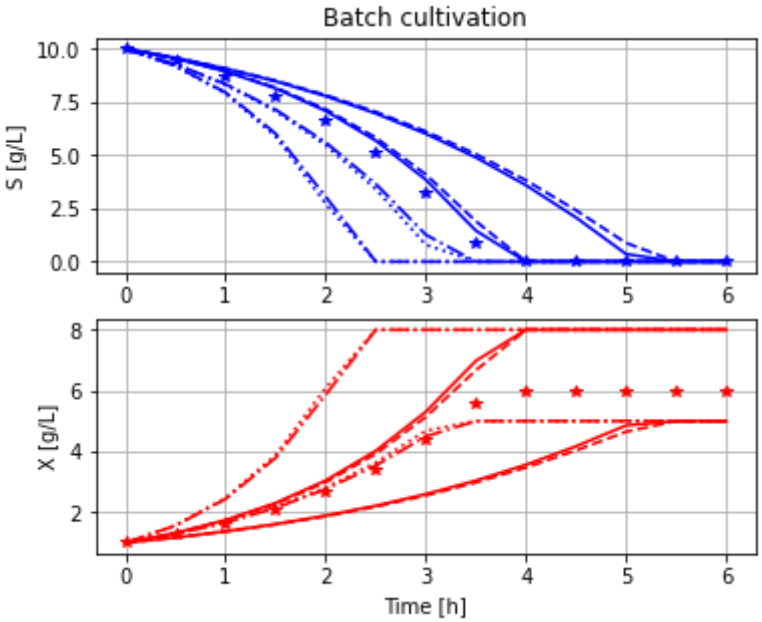
In [6]:
```
# Parameters to be estimated using parDict names and their bounds
parEstim = ['Y', 'qSmax', 'Ks']
parBounds = [(0.4, 0.7), (0.8, 1.3), (0.05, 0.20)]
parEstim_0 = [np.mean(parBounds[k]) for k in range(len(parBounds))]
```

In [7]:
```
# Simulation with nominal parameters
newplot(plotType='Demo_1')
par(Y=parEstim_0[0], qSmax=parEstim_0[1], Ks=parEstim_0[2])
simu(6)

# Show data
ax1.plot(data['time'], data['S'],'b*')
ax2.plot(data['time'], data['X'],'r*')
plt.show()
```



In [8]:
```
# Simulation over the parameter ranges given
newplot(plotType='Demo_1')
```

```
for Y_value in parBounds [0]:
    for qSmax_value in parBounds[1]:
        for Ks_value in parBounds[2]:
            par(Y=Y_value, qSmax=qSmax_value, Ks=Ks_value)
            simu(6)

# Show data
ax1.plot(data['time'], data['S'],'b*')
ax2.plot(data['time'], data['X'],'r*')
plt.show()
```



Simulation over the diferent parameter combinations of the parameter bounds shows that data is "covered" and it should be possible to find a parameter combination that describes the data well.

Parameter estimation

Here we use the scipy.optimize.minimize() procedure which contain a family of differenet methods. Since we has chosen to work with bounds on the parameters to be estimated there are only three methods to choose between. Here the method Sequential Least SQuares Programming SLSQP is chosen.

In [9]:
```
# For clarity of code here we import again
import scipy.optimize
```

In [10]:
```
# Parameters to be estimated using parDict names and their bounds
extra_args = (parEstim, data, fmu_model , parDict, parLocation)
```

In [11]:
```
# Modified evaluation function tailored for Python optimization algorithms
def evaluation(x, parEstim, data=data, fmu_model=fmu_model,
               parDict=parDict, parLocation=parLocation):
    """The parameter list is tailored for scipy optimization algorithms interface,
       where the first parameter x is an array with parameters that are tuned
       and evalauted."""

    # Load model
    global model
```

```
    if model is None:
        model = load_fmu(fmu_model)
    model.reset()

    # Change parameters and initial values from default
    for i, p in enumerate(parEstim): model.set(parLocation[p], x[i])
    for p in set(parDict)-set(parEstim): model.set(parLocation[p], parDict[p])

    # Simulation options
    opts = model.simulate_options()
    opts['ncp'] = 12
    opts['result_handling'] = 'memory'
    opts['silent_mode'] = True

    # Simulate
    sim_res = model.simulate(start_time=0.0, final_time=6.0, options=opts)

    # Calculate loss function V
    V={}
    V['X'] = np.linalg.norm(data['X'] - np.interp(data['time'], sim_res['time'], sim
    V['S'] = np.linalg.norm(data['S'] - np.interp(data['time'], sim_res['time'], sim

    return V['X'] + V['S']
```

In [12]:

```
# Run minimize()
result = scipy.optimize.minimize(evaluation, x0=parEstim_0, args=extra_args,
                                 method='SLSQP', bounds=parBounds, options={"disp":T
```

```
Optimization terminated successfully    (Exit mode 0)
            Current function value: 0.0001843492608832435
            Iterations: 24
            Function evaluations: 127
            Gradient evaluations: 24
```

In [13]:

```
result
```

Out[13]:

```
     fun: 0.0001843492608832435
     jac: array([ 2.03835198, -0.46493559, -1.85094705])
 message: 'Optimization terminated successfully'
    nfev: 127
     nit: 24
    njev: 24
  status: 0
 success: True
       x: array([0.50000025, 1.00002488, 0.10014957])
```

The estimated parameters x are very close to the original values.

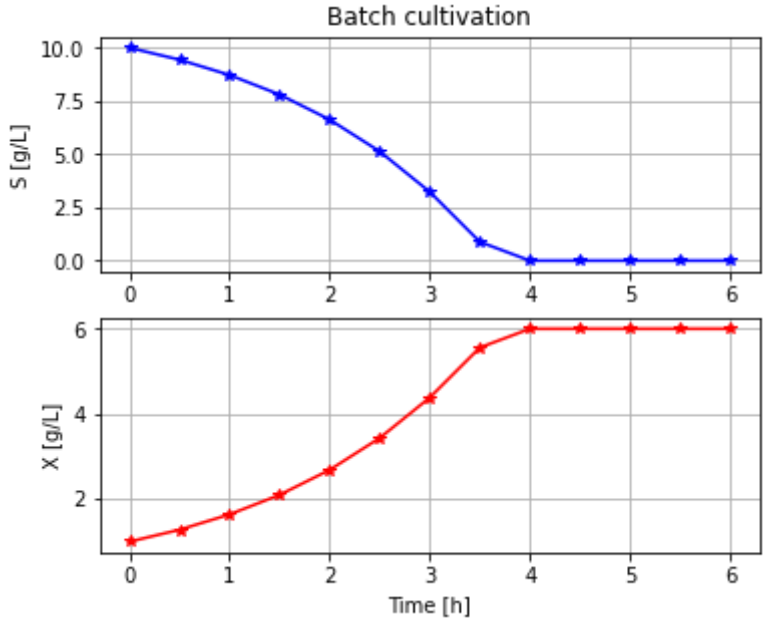
But if a use method 'Nelder-Mead' instead of the default 'BFGS' then I get success but needs double amount of function evaluations.

Simulation with estimated parameters compared with data

In [14]:

```
newplot(plotType='Demo_1')
par(Y=result.x[0], qSmax=result.x[1], Ks=result.x[2])
simu(6)

# Show data
```

```
ax1.plot(data['time'], data['S'],'b*')
ax2.plot(data['time'], data['X'],'r*')
plt.show()
```



Concluding remarks

- It is rather easy to make scipy function evaluations with flexibility to use compiled FMUs and provide parameters different fram default and that should not be tuned.
- FMU-explore with workspace dictionaries parDict[] and parLocation[] etc are useful also in this scipy-optimization context. We see also that the broad optimization family scipy.minimization() used in pyfmi for model.estimate() does work.

Appendix

In [15]:
```
describe('parts')
```

```
['bioreactor', 'bioreactor.culture', 'liquidphase', 'MSL']
```

In [16]:
```
describe('MSL')
```

```
MSL: 3.2.2 build 3 - used components:
```

In [17]:
```
system_info()
```

```
System information
 -OS: Windows
 -Python: 3.9.5
 -PyFMI: 2.9.5
 -FMU by: JModelica.org
 -FMI: 2.0
 -Type: FMUModelCS2
 -Name: BPL_TEST2.Batch
 -Generated: 2022-09-13T11:19:04
 -MSL: 3.2.2 build 3
 -Description: Bioprocess Library version 2.1.0 beta
 -Interaction: FMU-explore ver 0.9.3
```

In [18]:
```
optse = model.estimate_options()
```

In [19]:
```
optse
```

Out[19]:
```
{'tolerance': 1e-06,
 'result_file_name': '',
 'filter': None,
 'method': 'Nelder-Mead',
 'scaling': 'Default',
 'simulate_options': 'Default'}
```

In [ ]:
```

```