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. In this notebook we work with simulated data.

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 SLSQP but can easily be changed [1].

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 [2] 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 estimatation functionaly but the purpose seems to only address smaller examples. There is for instance no support to handle models that takes sub-models from libraries and necesssary changes of default parameters not to be estimated. Therefore we here define a Python function evaluation() that facilitate the formulation of the parameter estimation and also 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
```

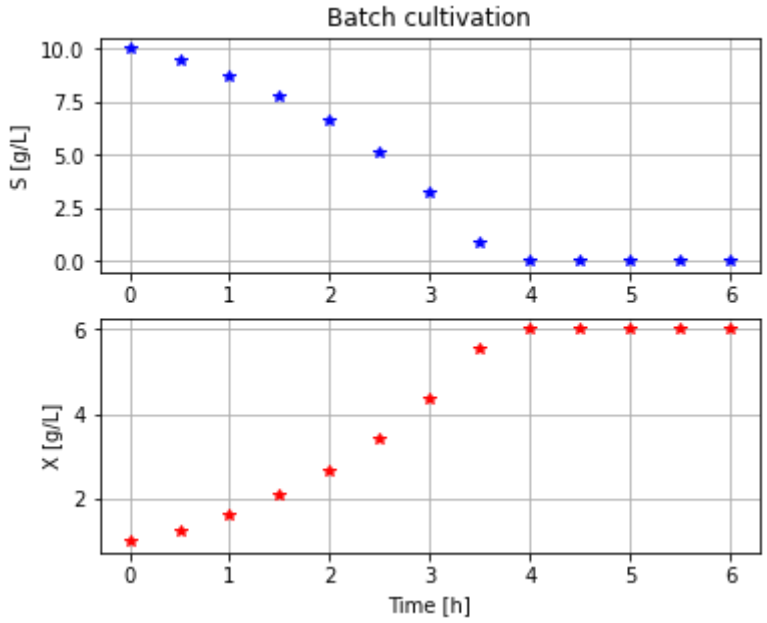
```
- disp()      - display parameters and initial values from the last simulation
- describe()  - describe culture, broth, parameters, variables with values / units

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]
```

1 Generate data later used for parameter estimation

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

In [4]:
```
# Data generated
simulationTime = 6.0
par(Y=0.50, qSmax=1.00, Ks=0.1)
init(V_0=1.0, VS_0=10, VX_0=1.0)
newplot(plotType='Demo_2')
opts['ncp'] = 12
simu(simulationTime)
```



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

	time	X	S
5	2.5	3.435479	5.129043e+00
6	3.0	4.385325	3.229350e+00
7	3.5	5.559252	8.814967e-01
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

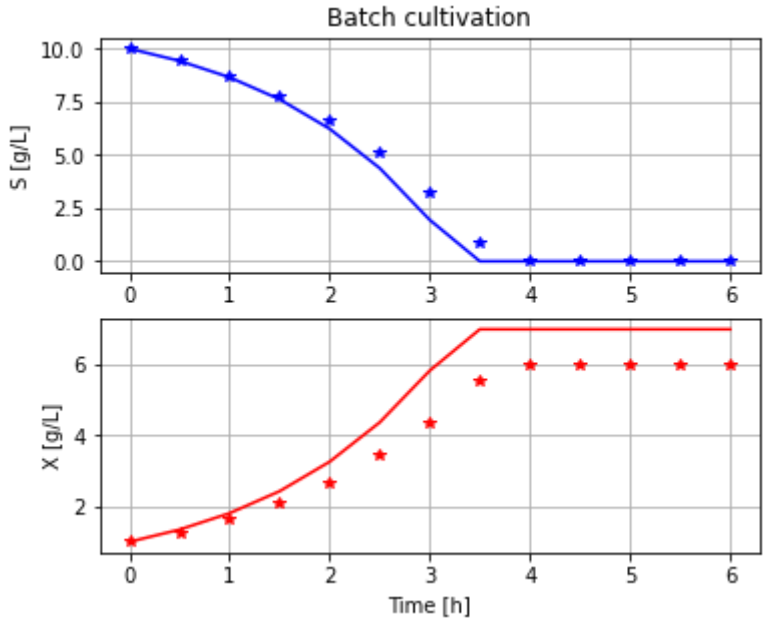
2 Simulation with initial guess of parameters compared with data

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.

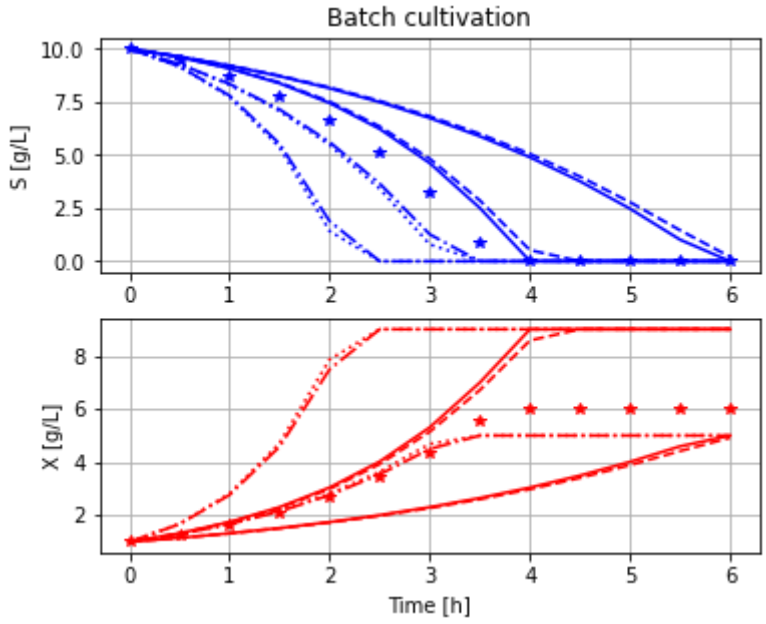
In [6]:
```
# Parameters to be estimated using parDict names and their bounds
parEstim = ['Y', 'qSmax', 'Ks']
parBounds = [(0.4, 0.8), (0.7, 1.3), (0.05, 0.20)]
parEstim_0 = [np.mean(parBounds[k]) for k in range(len(parBounds))]
```

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

# 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(simulationTime)

# 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 we have good hope to find a parameter combination that fits data well.

3 Parameter estimation

Here we use the scipy.optimize.minimize() procedure which contain a family of different methods [1]. 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.

Note that we in the definition of evaluation() make use of PyFMI-functions to administrate the simulation parameters as well as running it, instead of using the simplified simu() function we are used to.

In [9]:
```
# Optimization routine import
import scipy.optimize
```

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

In [11]:
```
# Modified evaluation function tailored for Python optimization algorithms
def evaluation(x, parEstim, data=data, fmu_model=fmu_model, simulationTime=simulatio
               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 and parEstim is a list of the names of these parameters."""

    # 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=simulationTime, 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]:
```
import time
```

In [13]:
```
# Run minimize()
start_time = time.time()
result = scipy.optimize.minimize(evaluation, x0=parEstim_0, args=extra_args,
                                 method='SLSQP', bounds=parBounds, options={"disp":T
print('CPU-time =', time.time()-start_time)
```

```
Optimization terminated successfully    (Exit mode 0)
            Current function value: 2.1979389561914015e-05
```

```
            Iterations: 40
            Function evaluations: 200
            Gradient evaluations: 40
CPU-time = 1.0532145500183105
```

In [14]:
```
result
```

Out[14]:
```
     fun: 2.1979389561914015e-05
     jac: array([-37.65308896, -31.98953381,   4.33273867])
 message: 'Optimization terminated successfully'
    nfev: 200
     nit: 40
    njev: 40
  status: 0
 success: True
       x: array([0.50000013, 1.00000072, 0.10000807])
```

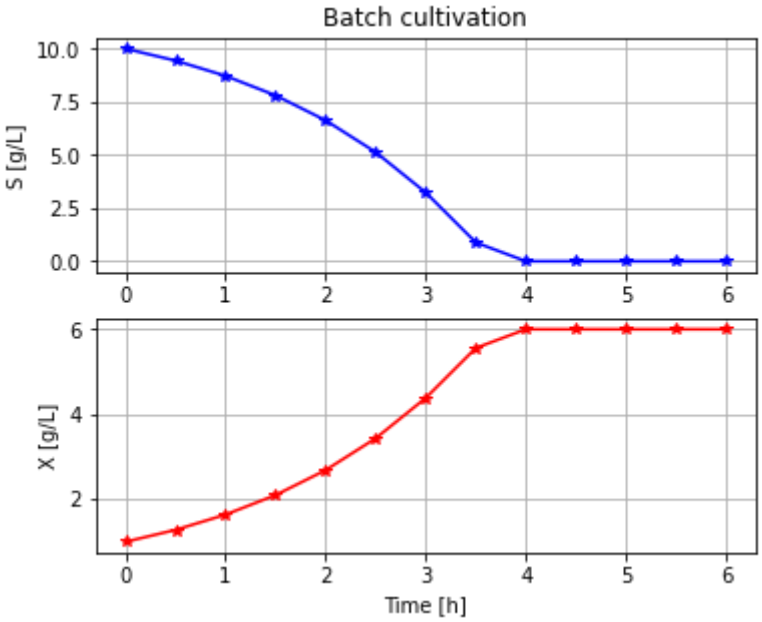
The estimated parameters result.x are very close to the original values and no surprise.

Test of the three methods available that handle parameter bounds: TNC, L-BFGS-B and SLSQP. It turns out that SLSQP is by far the fastest. It is 3 times faster than L-BFGS-B which is faster than TNC. Can be that SLSQP is less robust though. The nit (number of iterations does not differ that much though: 24 vs 30. The nfev (number of function evaluations) is perhaps more important 127 vs 256. A more precise timer function is likely timeit for this short times.

The Nelder-Mead algorithm has a good reputation to be very robust, but more slow, and with this method we cannot have bounds on the parameters.

4 Simulation with estimated parameters compared with data

In [15]:
```
newplot(plotType='Demo_1')
par(Y=result.x[0], qSmax=result.x[1], Ks=result.x[2])
simu(simulationTime)

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



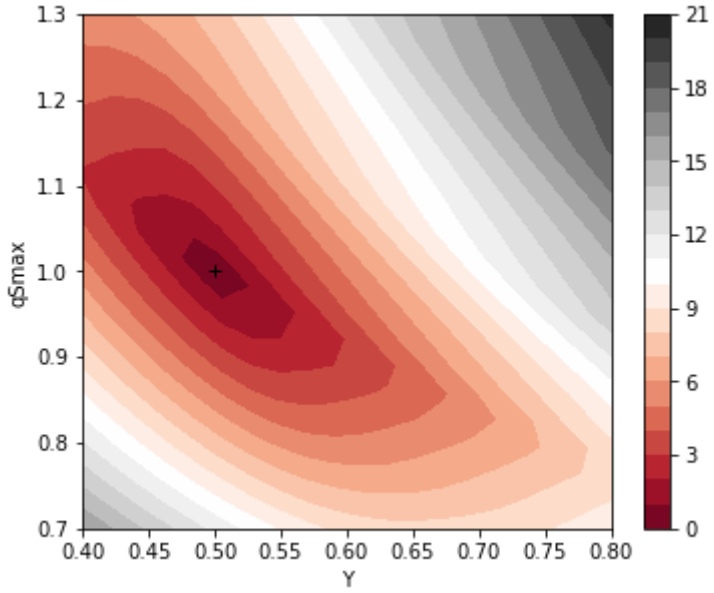
In [16]:
```
# The estimated parameters are
for i in range(len(parEstim)): print(parEstim[i],':', result.x[i])
```

```
Y : 0.5000001288023594
qSmax : 1.0000007200190635
Ks : 0.10000807243333859
```

5 Analysis of the loss function

The problem is small and analysis of the loss function brings some insight. From the diagram above showing parameter sweep over combinations min- and max-parameters we see that the parameter K_s has little influence. Let use set that a fixed value and then plot the loss function in the parameters Y and $qSmax$. We do this by go through all the parametera combinations and evaluate each of them.

In [17]:
```
# Sweep through Y and qSmax variation and store the value of the loss-function for e
nY = 20
nqSmax = 20
V = np.zeros((nY, nqSmax))

Y = np.linspace(parBounds[0][0],parBounds[0][1],nY)
qSmax = np.linspace(parBounds[1][0],parBounds[1][1],nqSmax)

for j in range(nY):
    for k in range(nqSmax):
        V[k,j] = evaluation([Y[j], qSmax[k], 0.1], parEstim)

# Contour plot
plt.figure()
plt.clf
plt.subplot(1,1,1)
plt.contourf(Y, qSmax, V, 20, cmap='RdGy')
plt.plot(result.x[0], result.x[1],'k+')
plt.colorbar()
plt.ylabel('qSmax')
plt.xlabel('Y')
plt.show()
```



We see the following in the contour diagram of the loss function simplified:

- The minima is unique in the range of parmaters we study. This is good news.
- The contour plot is ellipsoid and rather narrow. The more narrow the ellipsoid the more difficult and more time it takes to converge to the minima.
- The direction of the ellipsoid axis indicate the correlation you may get between the two parameters during the minimization process.

Note that the form of the contour plot change with the parameters (and initial values) of the actual proces. You can see the impact by changing the parameters in "cell # 4" where data is generated and then just choose to run that cell and the cells below. No need to restart the notebook.

6 Summary

A choice was made to work with allowed ranges of parameters to be estimated and a start value was defined as the center point in this parameter space. There are only three methods available in optimize.minimize() that can handle bounds on parameters.

An evaluation() function was created that define how the difference beween simulation and data is measured. The function is rather transparent and easy to modify and you may want to change weight on the loss in S and X, for instance. Here they have so far equal weight.

The FMU-explore workspace dictionaries partDict[] and parLocation[] are useful also here and simplify the code for the evaluation() function. But we also use the detailed PyFMI-functions to administrate and set parameters of the actual simulation.

The call optimize.minimize() has several parameters and can easily be modified, for instance change of method.

The estimated parameters were close to perfect!

The contour plot of the simplified loss function shows that the minima is unique and should not be difficult too difficut to obtain.

7 References

[1] Scipy Reference guide on optimize.minimize() here

[2] Andersson, C., Åkesson, J., Fuhrer C. : "PyFMI: A Python package for simulation of coupled dynamic models with the functional mock-up interface", Centre for Mathematical Sciences, Lund University, Report LUTFNA-5008-2016, 2016.

Appendix

In [18]:
```
describe('parts')
```

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

In [19]:
```
describe('MSL')
```

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

In [20]:
```
system_info()
```

```
System information
 -OS: Windows
 -Python: 3.9.5
 -Scipy: 1.7.1
 -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 0.9.4
```

In [ ]:
```

```