Design aspects of ‘FMU-explore’ – a Python module to complement PyFMI

Jan Peter Axelsson

E-mail: jan.peter.axelsson@vascaia.se

Vascaia AB, Stockholm, Sweden

OpenModelica workshop 2022-01-31

Outline

- Background
  - Command line vs GUI
  - Command line vs Python scripting
- Examples
- Five commands made of Python functions
- Workspace dictionaries
- Handling of model state
- Conclusion: eat the cake and keep it!

Background

- Modelica – embraced GUI from start
- Process “view”
  - Facilitate configuration: components, connectors… OO
  - Facilitate “configuration” of diagrams to use
- Diagram “view”
  - Focus attention to standard diagrams
  - Cf fligth-simulator – “instrument panel”
- Teaching operators

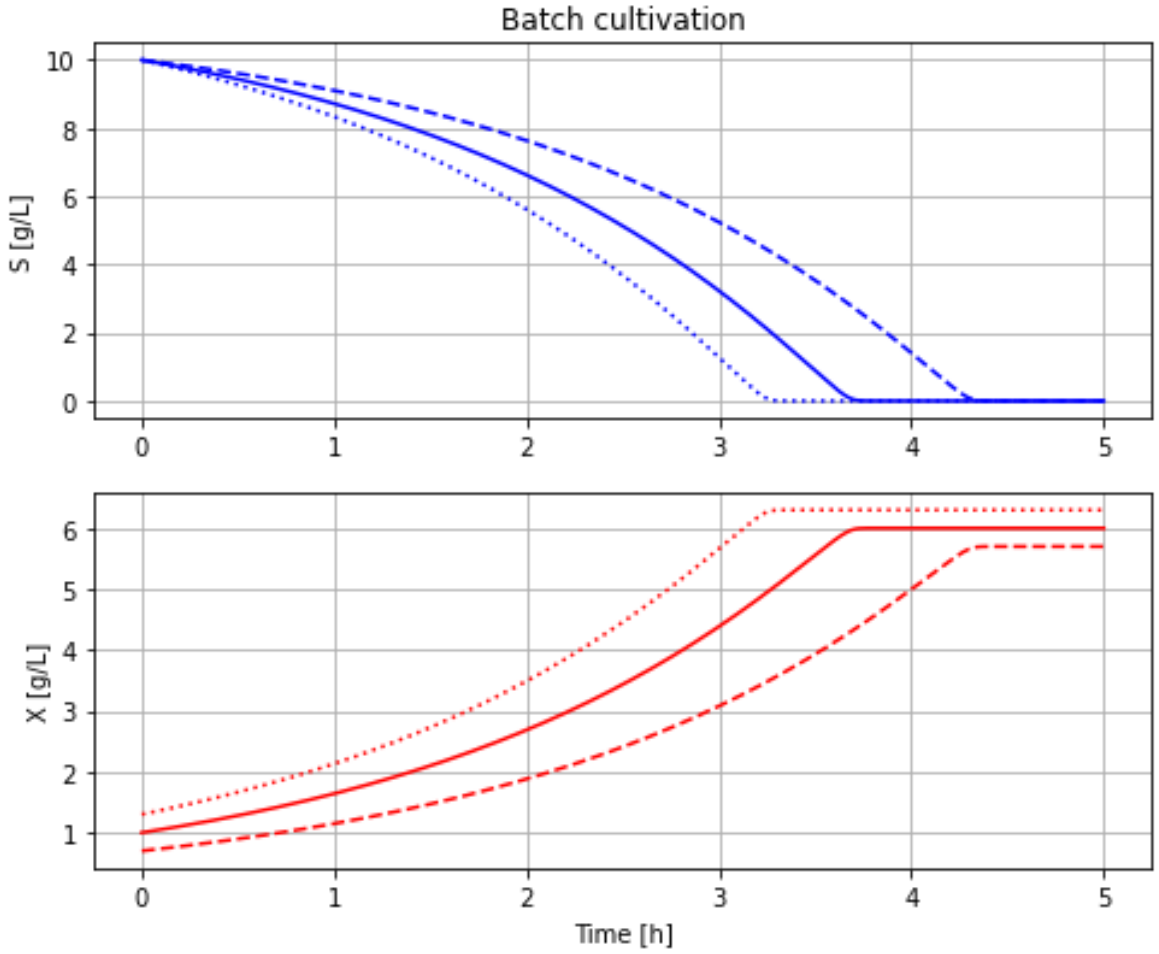
Command line tools for Python

Compile Modelica to get an FMU to interact with

- PyFMI – Modelon
- OMPython - OpenModelica
- FMPy – Dassault

Focus talk on Python and PyFMI
but likely relevant also for FMPy and OMPython

A simple example



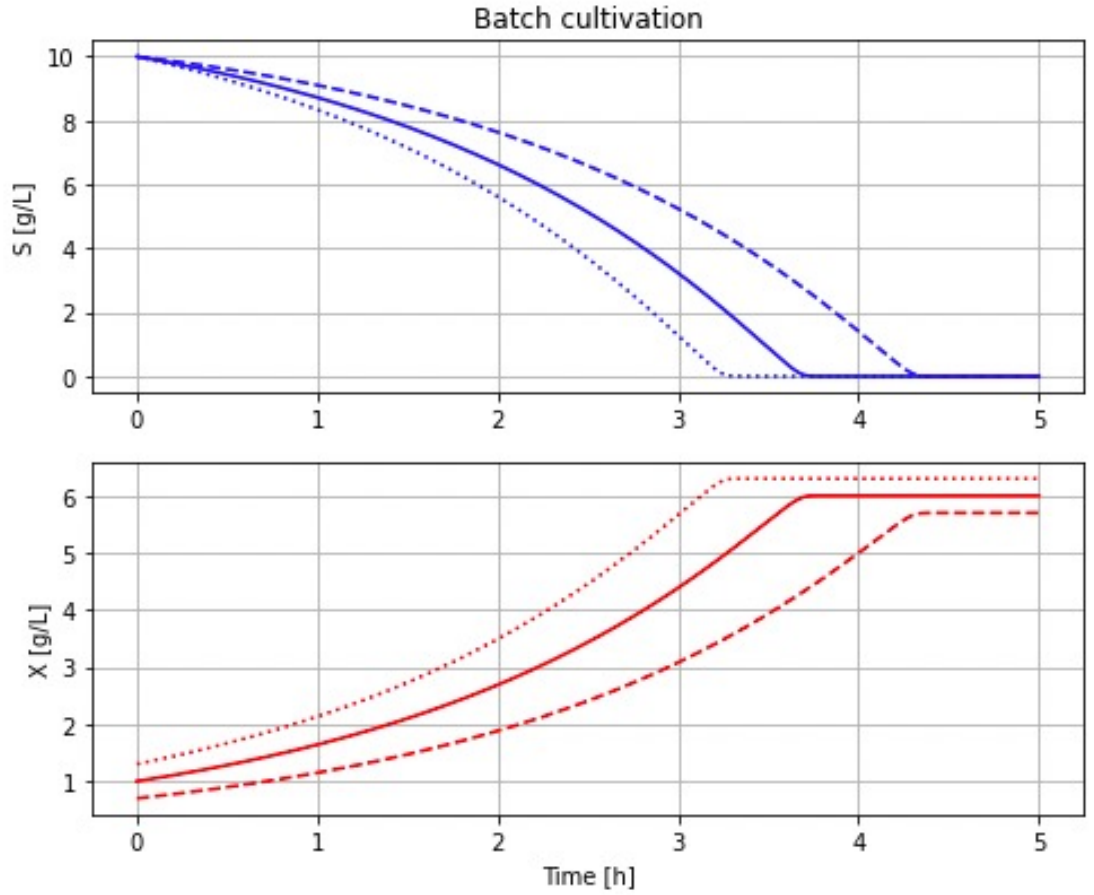
Scripting using PyFMI

In [9]:
```
# Setup-script defines fmu_model, parDict[], parLocation[] and some more

# - newplot()
plt.figure()
ax1=plt.subplot(2,1,1); ax1.grid(); ax1.set_ylabel('S [g/L]')
ax2=plt.subplot(2,1,2); ax2.grid(); ax2.set_ylabel('X [g/L]'); ax2.set_xlabel('Time [h]')
lines=['-','--',':']; linecycler = cycle(lines)

for value in [1.0, 0.7, 1.3]:
    # - init(VX_0=value)
    parDict['VX_0'] = value

    # - simu(5)
    model = load_fmu(fmu_model)
    for key in parDict.keys(): model.set(parLocation[key], parDict[key])
    sim_res = model.simulate(0, 5, options=opts)

    linetype = next(linecycler)
    ax1.plot(sim_res['time'], sim_res['bioreactor.c[2]'], color='b', linestyle=linetype)
    ax2.plot(sim_res['time'], sim_res['bioreactor.c[1]'], color='r', linestyle=linetype)
```

Much to write at the command line
Flexibility comes at a price
Use diagram “canvas” to collect results

FMU-explore with PyFMI

In [4]:
```
newplot(plotType='Demo_1')
for value in [1,0.7,1.3]: init(VX_0=value); simu(5)
```

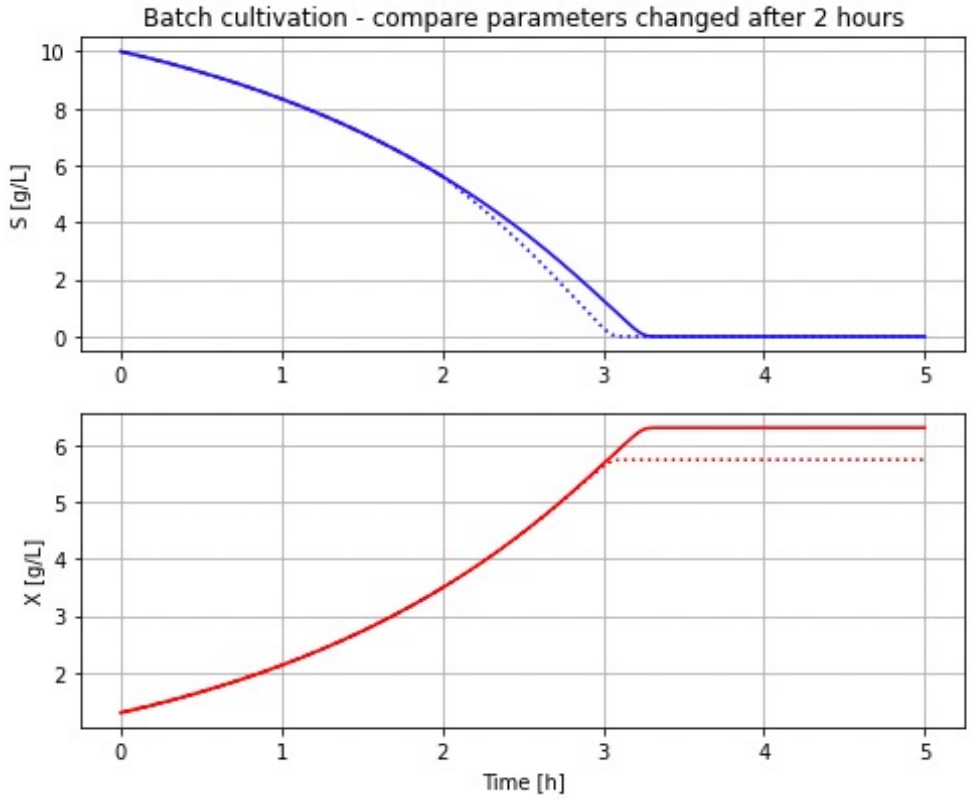


In [5]:
```
describe('bioreactor.V')
```

```
Reactor broth volume 1.0 [ L ]
```

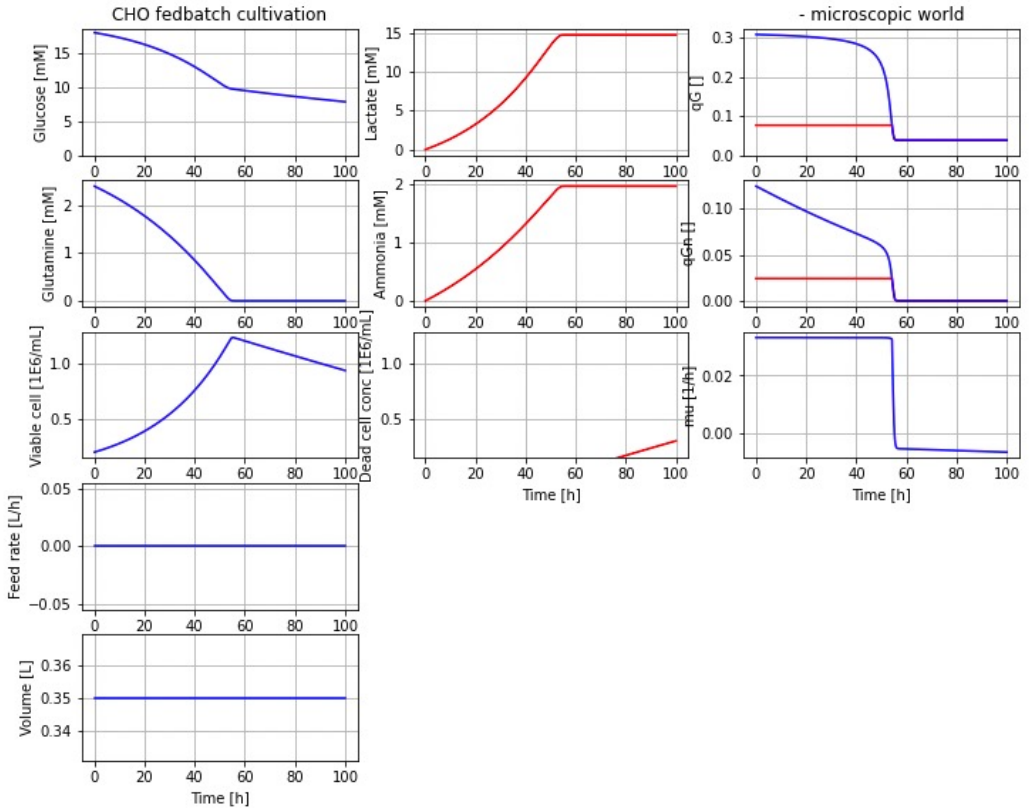
FMU-explore with PyFMI
- parameter change and continue

```
In [7]: newplot(title='Batch cultivation - compare parameters changed after 2 hours', plotType='Demo_1')
        par(Y=0.5, qSmax=1.0); simu(5)
        simu(2); par(Y=0.4, qSmax=1.25); simu(3, 'cont')
```

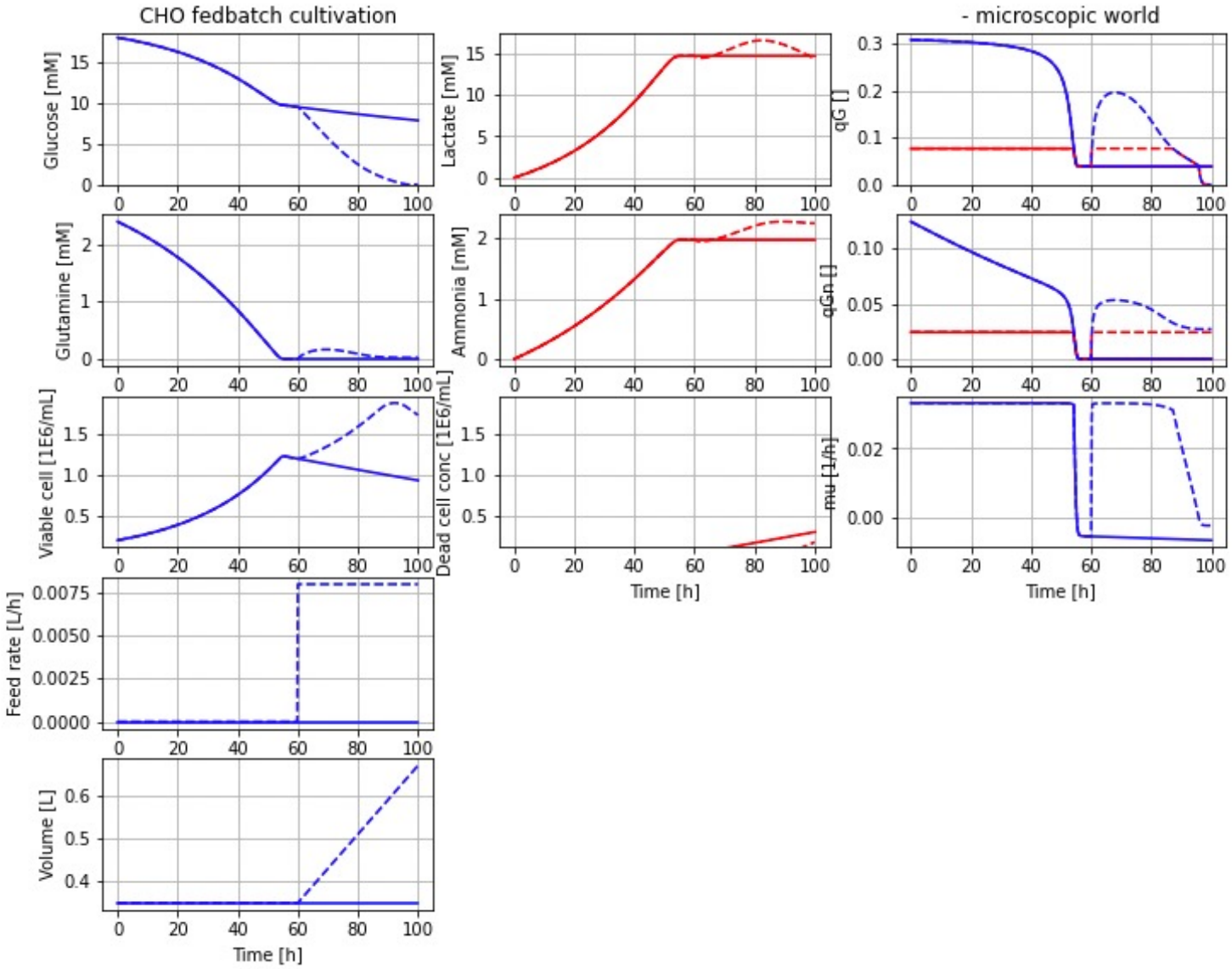


FMU-explore with PyFMI - larger example

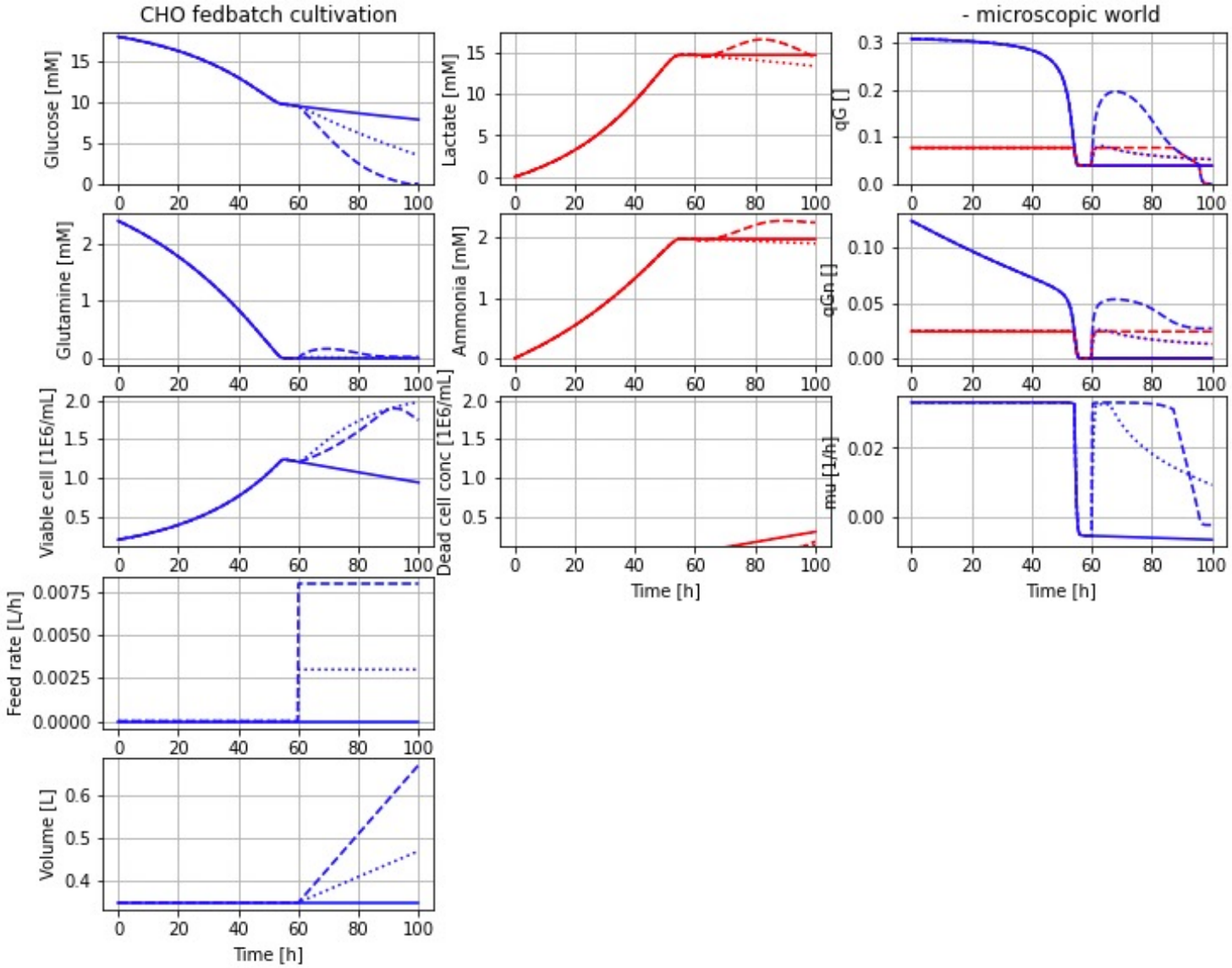
```
In [6]: # Slide 1
newplot('CHO fedbatch cultivation', plotType='Textbook_2')
par(G_in=0, Gn_in=4.0)
par(t1=60, F1=0.0);  simu(100)
```



```
In [7]: # Slide 2
newplot('CHO fedbatch cultivation', plotType='Textbook_2')
par(G_in=0, Gn_in=4.0)
for value in [0, 0.008]: par(t1=60, F1=value); simu(100)
```

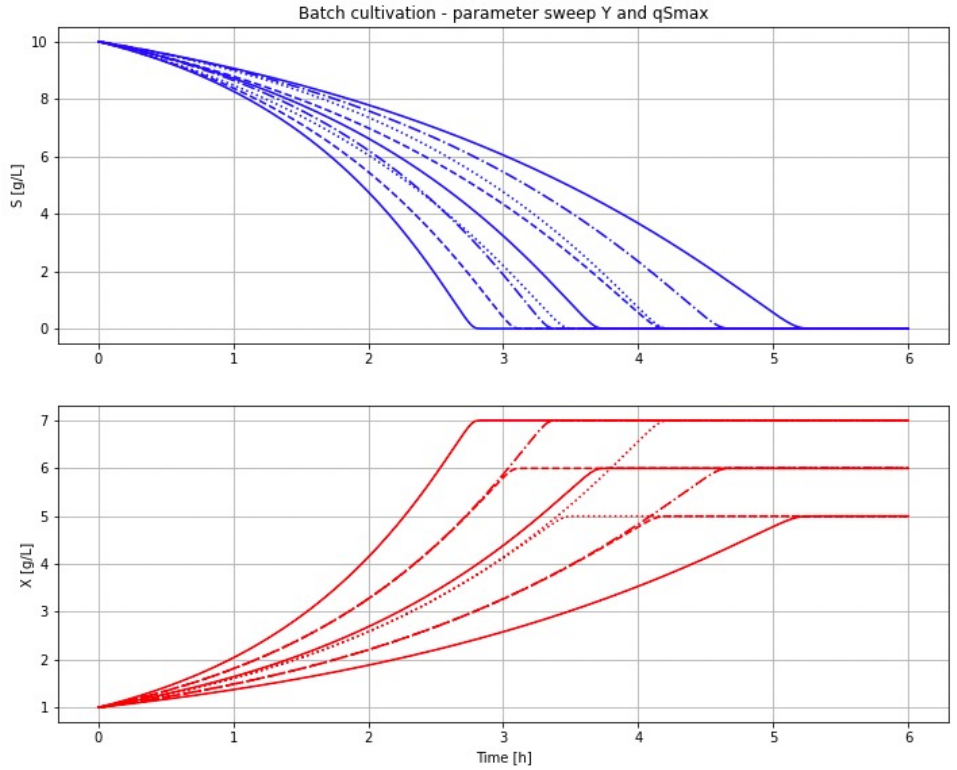


In [8]:
```
# Slide 3
newplot('CHO fedbatch cultivation', plotType='Textbook_2')
par(G_in=0, Gn_in=4.0)
for value in [0, 0.008, 0.003]: par(t1=60, F1=value); simu(100)
```



FMU-explore with PyFMI - automated exploration - GSA

```
In [5]: # Overview of the impact of parameter variations
        newplot(title='Batch cultivation - parameter sweep Y and qSmax', plotType='Demo_1')
        for Y_value in [0.4,0.5,0.6]:
            for qSmax_value in [0.8, 1.0, 1.2]:
                par(Y=Y_value, qSmax=qSmax_value)
                simu(6)
```



Outline

- Background
  - Command line vs GUI
  - Command line vs Python scripting
- Examples
- **Five commands made of Python functions**
- Workspace dictionaries
- Handling of model state
- Conclusion: eat the cake and keep it!

Handfull commands
- general code independent of app

- newplot()
- par(), init()
- simu()
- disp(), describe()

Cf *Simnon* (H. Elmqvist, 1975)

Benefits
- Less to enter
- Readability
- Incremental changes…

Implementation: Workspace dictionaries and lists

Information to provide for the application

- parDict[] – short name for parameter and value
- parLocation[] – short name > modelica model name
- stateDict[] – states to handle simu(mode='cont')
- diagrams[] – tailored standard diagram for results
  (type list of "command line" strings)

- sim_res[] – all simulation results stored
  (type FMIResult - extract numpy.ndarray)

Global variables

Global variables?

Workspace benefit from “global variables”

- Less to type
- Facilitate study of incremental changes…
  - i.e. you WANT side effects

Little written about proper use of global variables?

parDict[] and parLocation[]

parDict – short names and values

parLocation – short names -> modelica code names

```
for key in parDict.keys(): model.set(parLocation[key], parDict[key])
```

Focus/restrict interaction to certain parameters

- disp() – displays only these

  pyfmi: `model.get_model_variables().keys()` – displays all

Good for

- Teaching situation – avoid unnecessary information
- Consultancy – a step towards protect IP… (needs to do more)

Handling of model state

simu(mode='cont') need to set initial values

- Continuous time state
- Discrete time sampled systems (i.e. regulators)

- Continuous time – pyfmi: model.get_states_list()
- Discrete time? – today configure manually

  Interested in an automatic solution!

  Possibly pyfmi: model.get_fmu_state() …

Diagrams as a list-of-commands

```
diagrams.clear()
diagrams.append(“ax1.plot(t, sim_res[‘reactor.c[2]’], color=‘b’)
diagrams.append(“ax2.plot(t, sim_res[‘reactor.c[1]’], color=‘b’)
…
def simu(..., diagrams=diagrams,…)
     …
     sim_res = model.simulate(…)
     …
     for command in diagrams: eval(command)
```

Application setup-file

The setup needed:

- fmu_model – name of FMU
- parDict[] – short names -> values
- parLocation[] – short names -> modelica address
- timeDiscreteStates[] – used to make stateDict[]

- newplot() – used to make different lists diagrams[]
- describe() – extend with information not in code

FMU-explore with PyFMI - automated exploration - GSA

Full factor parameter sweep easy to express

```
for A in […]:
    for B in […]:  par(A=A, B=B); simu()
```

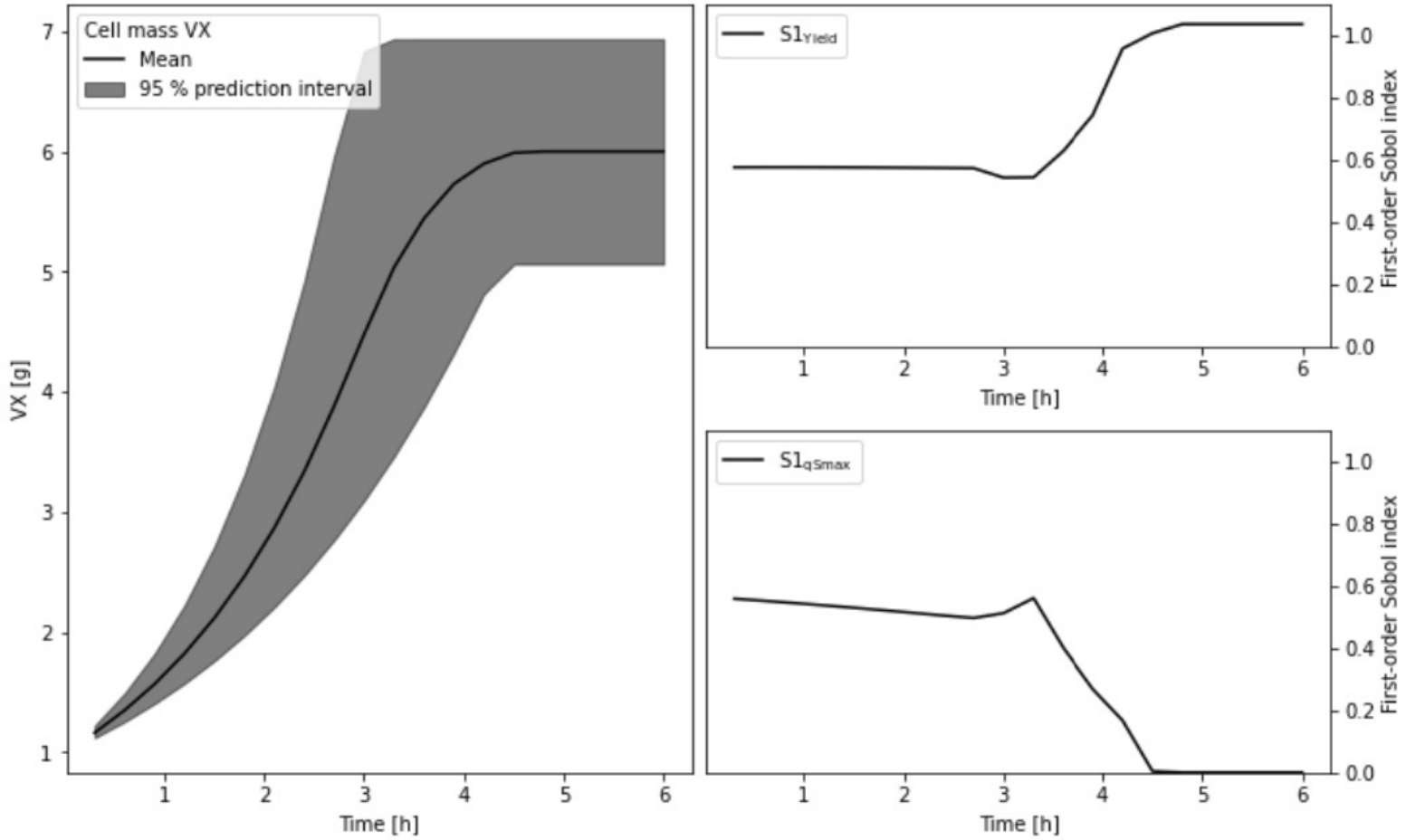
Reduced factor or “sampled” parameter space (module SALib)

```
par_values = saltelli.sample(problem, …)
out = np.array([simulation(*AB) for AB in par_values])
```

```
def simulation(A,B):
     par(A=A, B=B)
     simu()
     return sim_res
```

```
def simulation(A,B):
     model = load_fmu()
     model.set(..) // initial states
     model.set(..) // default parameters
     model.set(['A','B'], [A, B])
     sim_res = model.simulate()
     return sim_res
```

Result (using sobol_indices) - automated exploration - GSA



Conclusion

- Revival of command line interaction!?
  - Jupyter notebook
  - Easier to document procedures than GUI?
- FMU-explore facilitate interactive use of PyFMI
  - Extend to OMPython, FMPy?
- FMU-explore used side by side with the richer PyFMI-commands – **“eat the cake and keep it”**

- Next steps
  - FMI 3.0
  - FMPy?
  - Github?

Demo of FMU-explore with a batch model

This Jupyter notebook shows the possibilities with using FMU-explore to facilitate investigation of a batch cell cultivation model. In the first sections the ease of command-line interaction is shown. The final section is an example of more advanced use for global sensitivity analysis and the FMU-explore environment work well also here.

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

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

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

The first step is to run a setup-file that make the FMU-explore environment accesible together with the application dependent workpace dictionaries and the compiled FMU for the model taken from Bioprocess Library. This library was presented at OpenModelica workshop 2021.

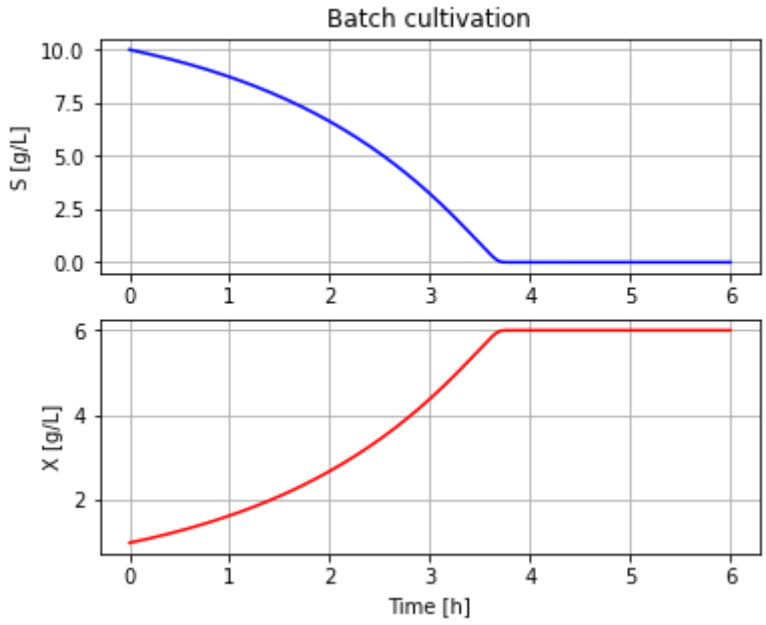
In [1]:
```
run -i BPL_TEST2_Batch_explore.py
```

```
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]
```

A first simulation

In [3]:
```
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')
simu(6)
```



Access information about the model

Here we see the use of FMU-explore functions describe() and disp() to bring up information about the model we simulate.

In [4]:
```
describe('parts')
```

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

In [5]:
```
disp('culture')
```

```
Y : 0.5
qSmax : 1.0
Ks : 0.1
```

In [6]:
```
disp('culture', mode='long')
```

```
bioreactor.culture.Y : Y : 0.5
bioreactor.culture.qSmax : qSmax : 1.0
bioreactor.culture.Ks : Ks : 0.1
```

In [7]:
```
describe('bioreactor.culture.Y')
```

```
Yield of cells from substrate 0.5 [ g/g ]
```

In [8]:
```
disp('bioreactor', mode='long')
```

```
bioreactor.V_0 : V_0 : 1.0
bioreactor.m_0[1] : VX_0 : 1.0
bioreactor.m_0[2] : VS_0 : 10.0
bioreactor.culture.Y : Y : 0.5
bioreactor.culture.qSmax : qSmax : 1.0
bioreactor.culture.Ks : Ks : 0.1
```

Note that describe handle synonyms in the same way

In [9]:
```
describe('VX_0')
```

```
Initial substance mass 1.0 [ g ]
```

In [10]:
```
describe('bioreactor.m_0[1]')
```

```
Initial substance mass 1.0 [ g ]
```

Note that with describe() we can get information about all variables and parameters and not only those shown by disp(). The disp() command only access parameters that can be changed by par() or init().

In [11]:
```
describe('bioreactor.m[1]')
```

```
Substance mass 6.0 [ g ]
```

In [12]:
```
describe('culture')
```

```
Simplified text book model - only substrate S and cell concentration X
```

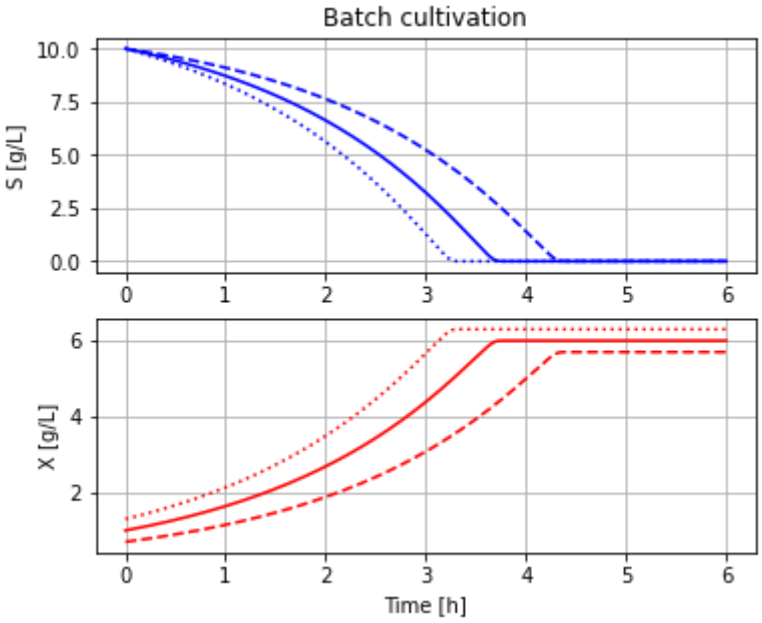
In [13]:
```
describe('liquidphase')
```

```
Reactor broth substances included in the model

Cells     index =  1 molecular weight =  24.6 Da
Substrate index =  2 molecular weight =  180.0 Da
```

Explore the model interactively

The model is simulated interactively with ease using the FMU-explore envrionment.

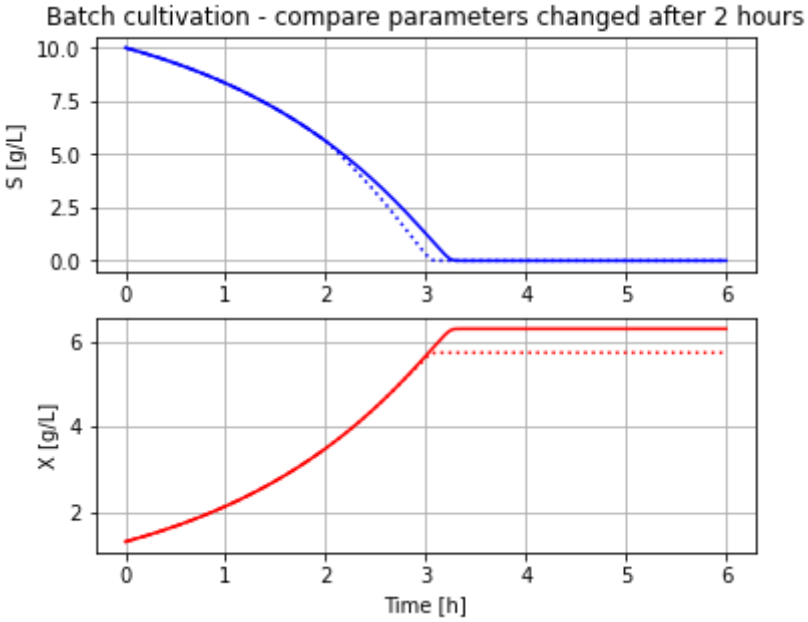
In [14]:
```
newplot(plotType='Demo_1')
for value in [1,0.7,1.3]: init(VX_0=value); simu(6)
```



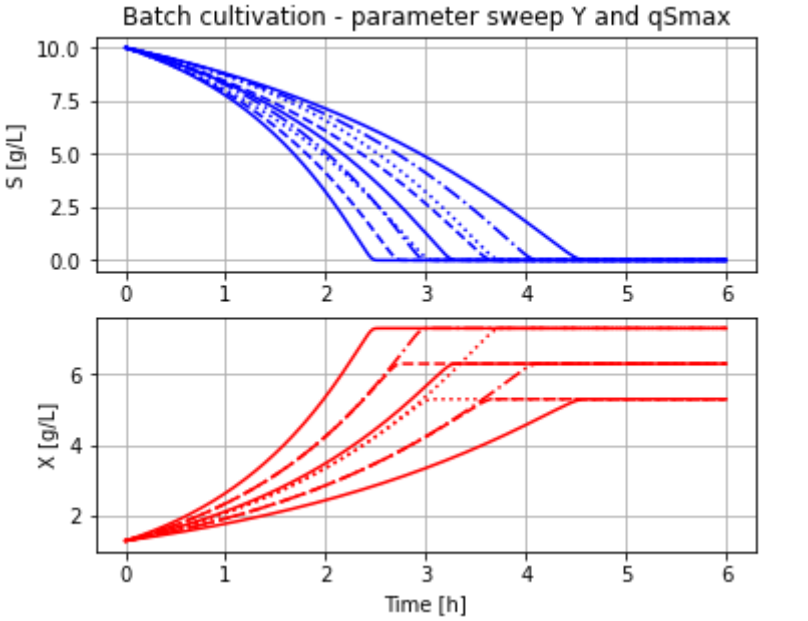
In [15]:
```
describe('bioreactor.V')
```

```
Reactor broth volume 1.0 [ L ]
```

In [16]:
```
newplot(title='Batch cultivation - compare parameters changed after 2 hours', plotTy
par(Y=0.5, qSmax=1.0); simu(6)
simu(2); par(Y=0.4, qSmax=1.25); simu(4, 'cont')
```



In [17]:
```
# Overview of the impact of parameter variation of Y and qSmax
newplot(title='Batch cultivation - parameter sweep Y and qSmax', plotType='Demo_1')
for Y_value in [0.4,0.5,0.6]:
    for qSmax_value in [0.8, 1.0, 1.2]:
        par(Y=Y_value, qSmax=qSmax_value)
        simu(6)
```



Exploration automated

The FMU-explore environment is here used together with a module for sensitivity analysis SALib. The advantage to continue to use FMU-explore here is mainly to easily within the same notebook switch between ad-hoc command-line interaction (as above) and more focused script-based analysis. More about the module SALib you find here.

https://salib.readthedocs.io/en/latest/getting-started.html#installing-salib

In [18]:
```
from SALib.sample import saltelli
from SALib.analyze import sobol
```

In [19]:
```
# Define the problem for SALib
problem = { 'num_vars': 2,
    'names': ['Yield', 'qSmax'],
    'bounds': [[0.4, 0.6],
               [0.8, 1.2]]}
```

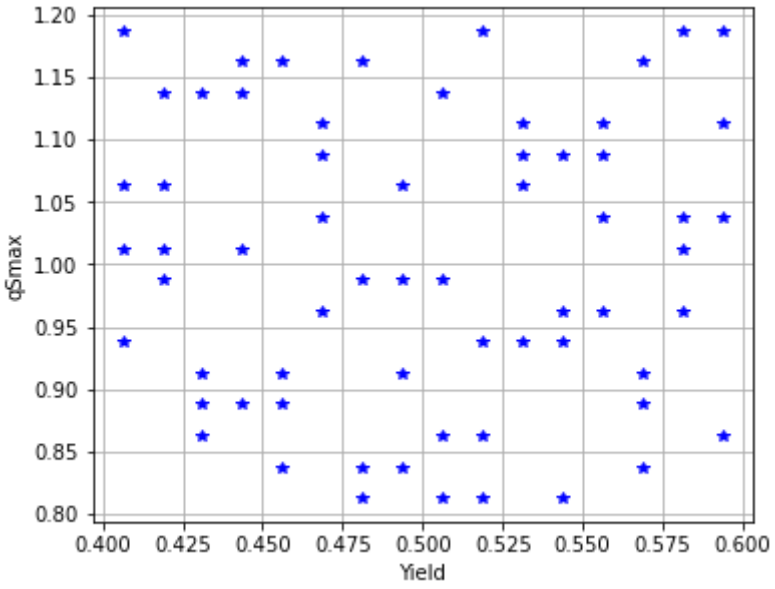
The sample size is chosen to give reasonably good statistics in the end.

In [20]:
```
# Sample
par_values = saltelli.sample(problem, 2**4, calc_second_order=False)
```

In [21]:
```
# Check size of par_values
len(par_values)
```

Out[21]: 64

In [22]:
```
# Take a look at the sample of parameter values obatained
plt.figure()
plt.plot(par_values[:,0], par_values[:,1],'*b')
plt.xlabel('Yield'); plt.ylabel('qSmax')
plt.grid()
plt.show()
```



The next step is do define a function that evaluates a given parameter setting and we call that function simulation(). It combines the tasks of the FMU-explore functions par() and simu() and also deliver an output for later post-processing.

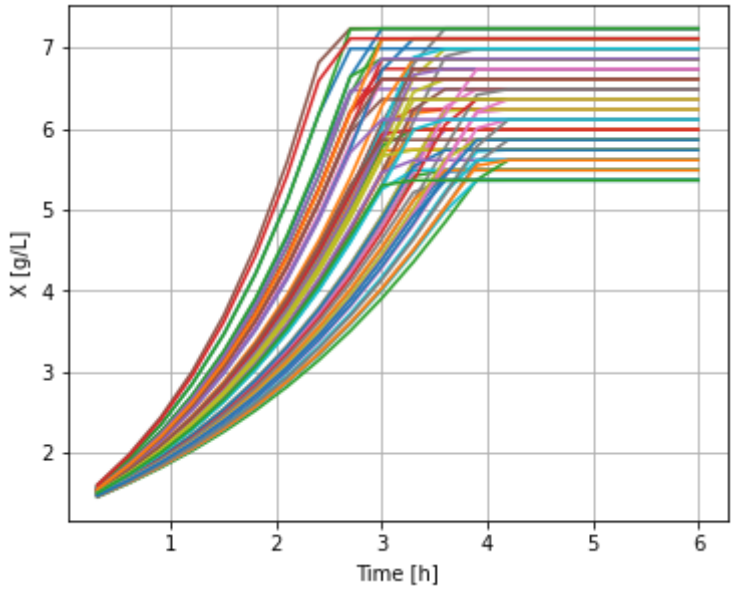
We choose here to make use of FMU-explore par() and simu() and the implicit workspace variables when we define simulation(). This function could alternatively be defined using PyFMI functions instead and avoid using the workspace variables and the code would be somewhat longer, but slightly faster.

In [23]:
```
# Define evaluation in terms of simulation
# - note that the first data point is the initial value
#   and eliminated since not influenced by the parameter variatioin

def simulation(Y_value, qSmax_value):
    par(Y=Y_value, qSmax=qSmax_value)
    opts['ncp'] = 20
    simu(6.0)
    return sim_res['bioreactor.c[1]'][1:]
```

The evaluation of all par_values can now be done effectively and stored in ndarray out.

In [24]:
```
# Evaluate all the different par_values
out = np.array([simulation(*p) for p in par_values])
```

In [25]:
```
# Plot the different cell concentration curves stored in out
plt.figure()
for i in range(len(out)): plt.plot(sim_res['time'][1:], out[i,:])
plt.ylabel('X [g/L]'); plt.xlabel('Time [h]'); plt.grid()
```

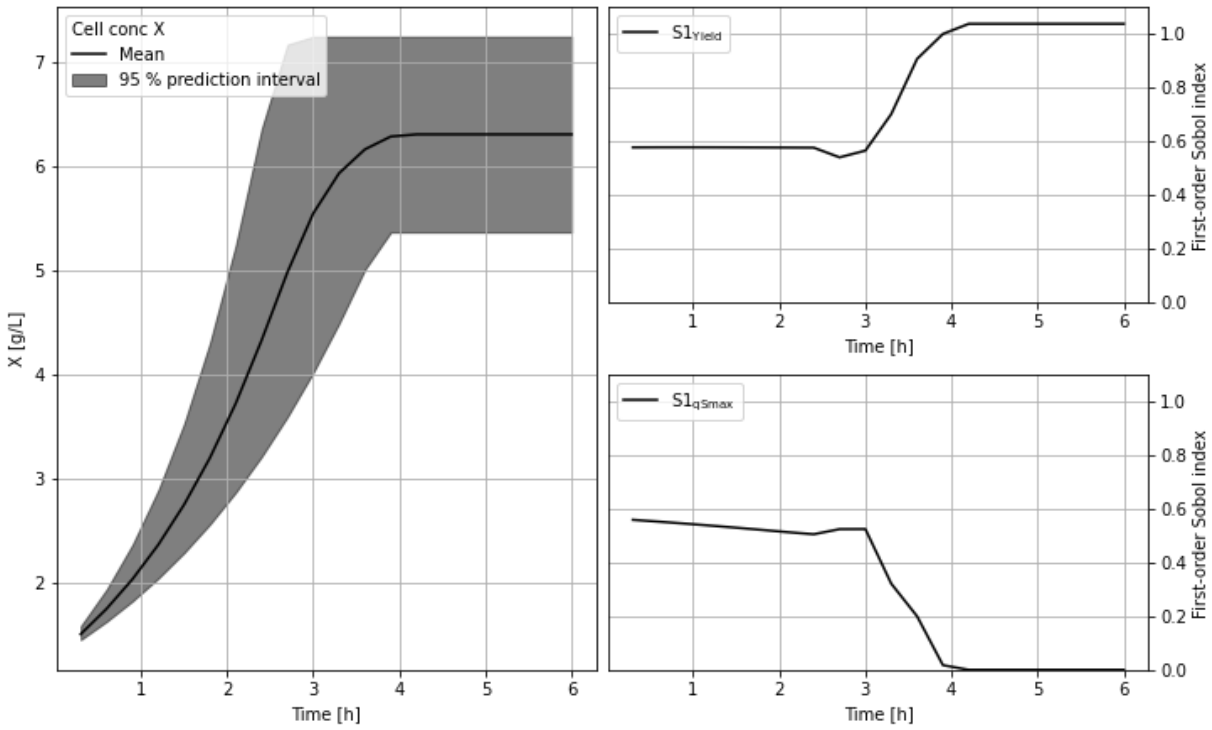


Note, the X values increase monotonically with time which can be seen by model inspection.

In [26]:
```
# Analyse using sobol indices
sobol_indices=[sobol.analyze(problem,OUT,calc_second_order=False) for OUT in out.T]
```

In [27]:
```
# Standard plot - adjusted for this dynamical batch example
t = sim_res['time'][1:]
#np.shape(out.T)

S1s = np.array([s['S1'] for s in sobol_indices])
fig = plt.figure(figsize=(10, 6), constrained_layout=True)
gs = fig.add_gridspec(2, 2)

ax0 = fig.add_subplot(gs[:, 0])
ax1 = fig.add_subplot(gs[0, 1])
ax2 = fig.add_subplot(gs[1, 1])

for i, ax in enumerate([ax1, ax2]):
    ax.plot(t, S1s[:, i],
        label=r'S1$_\mathregular{{{}}}$'.format(problem["names"][i]),
```

```
        color='black')
    ax.set_xlabel("Time [h]")
    ax.set_ylabel("First-order Sobol index")
    ax.set_ylim(0, 1.1)

    ax.yaxis.set_label_position("right")
    ax.yaxis.tick_right()

    ax.legend(loc='upper left')
    ax.grid()

ax0.plot(t, np.mean(out, axis=0), label="Mean", color='black')

# in percent
prediction_interval = 95

ax0.fill_between(t,
    np.percentile(out, 50 - prediction_interval/2., axis=0),
    np.percentile(out, 50 + prediction_interval/2., axis=0),
    alpha=0.5, color='black',
    label=f"{prediction_interval} % prediction interval")

ax0.set_xlabel("Time [h]")
ax0.set_ylabel("X [g/L]")
ax0.legend(title=r"Cell conc X", loc='upper left')._legend_box.align = "left"
ax0.grid()
plt.show()
```



Concluding remarks

We have seen how FMU-explore brings an environment with functions and global variables to the work space that facilitate interactive simulation of a compiled Modelica model in the form of an FMU.

In the last section we see that the FMU-explore environment is also useful in a more advanced scripting environment. An example of global sensitivity analysis using SALib was shown.

In [28]:
```
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
 -Description: Bioprocess Library version 2.0.9 - beta
 -Interaction: FMU-explore ver 0.8.9
```