WORKSHOP 2.3

Workflows and Parameter sweep with multi-sim

Contents

Introduction

In this workshop, you will look at some basic workflows that are available within the Modelon Impact user environment.

You will:

- Export a prepared Furuta Pendulum model as an FMU. The idea is to provide the model as a FMU to a third party who will do the control design using a different tool.
- Do a multistage simulation workflow, where one model is simulated into steady state, and some data is retrieved and written to file. This data is then used to parametrize a second model, that is analysed during dynamic conditions.
- Run a full factorial Design Of Experiment on a electric vehicle.
- Work in the package folder **TrainingPack.Day2.W3** (referred to as **W3** below)

FMU export

In this part of the workshop you will export a prepared model . It will also be shown how the compiler options can be leveraged to control variables exposed in the exported FMU.

TrainingPack has been prepared with a model that will be used during this exercise, **W3.WorkflowFMU.FurutaPendulum**. It is a physical model of a furuta pendulum with a prepared interface so that it can communicate with an external controller.

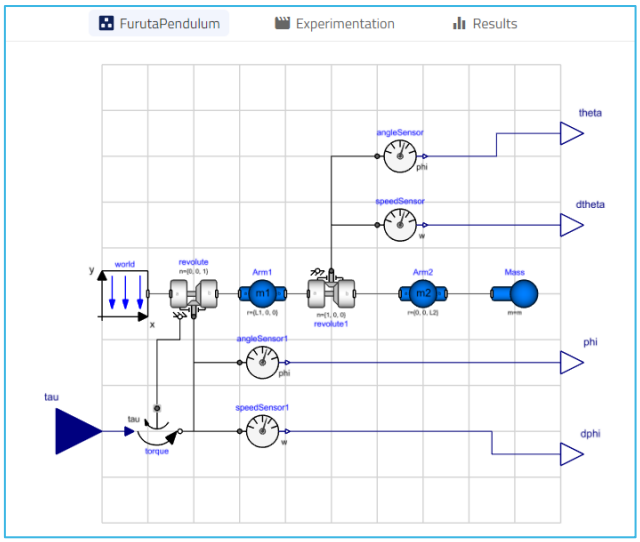


Figure 1

FMU export is initiated through the context manual in the library browser.

- Right-click **W3.WorkflowFMU.FurutaPendulum** in the library browser and export the FMU:

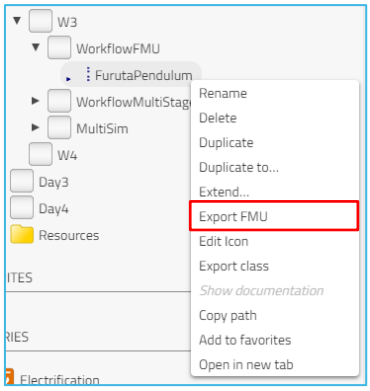


Figure 2

- A window will show up allowing to specifiy the basic properties of the FMU:

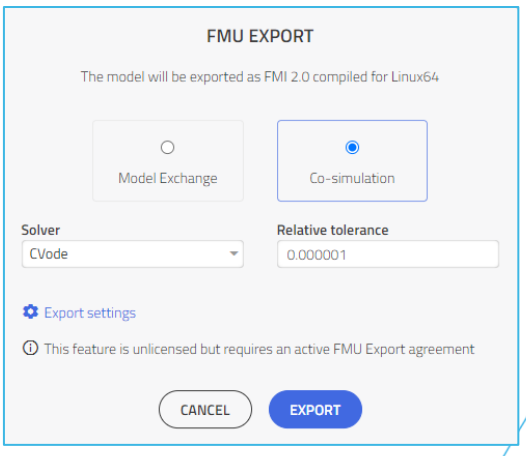
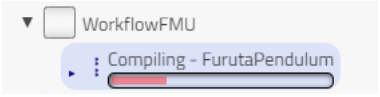


Figure 3

- Press the EXPORT BUTTON

A download will be initiated. The downloaded FMU is a zip file. Relevant information about the model is stored in modelDescription.xml file. Feel free to open and inspect the content of this file.



When the model is compiled, the default behavior is to make all variables visible in the output FMU with the structured Modelica naming intact. In Impact, the information about the Modelica source code that can be derived from the exported FMU can be reduced by:

a) Limiting what variables are exposed in the modelDescription.xml file
b) Automatically rename variables in the modelDescription.xml file

We will show how both these options work.

First we will create a “grey-box” FMU with only selected variables and required variables (states/derivatives/solver-related variables) exposed in the FMU interface. This can be used to hide sensitive data. This is done via a compiler option called **exclude_internal_variables**. Some hidden variables can then be selectively exposed by using the compiler option **include_internal_variables**. In this exercise we will hide all internal variables except for revolute.tau.

- Go to Settings, Export tab
- Add two new compiler options, they can be found in the drop down list of options:
    - exclude_internal_variables
    - include_internal_variables
- Set exclude_internal_variables to: *
- Set include_internal_variables to: revolute.tau

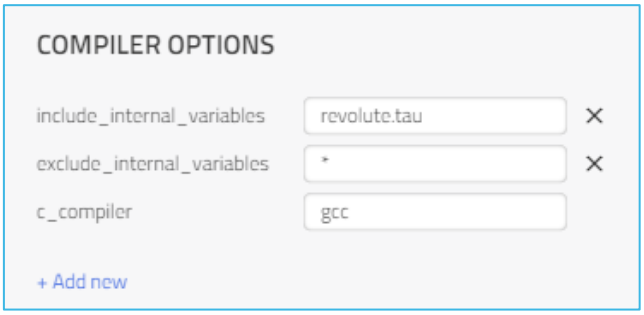


Figure 4

Export the model as an FMU again and inspect the modelDescription.xml again. All internal variables should now be gone. Note that a variable is an internal variable if it is not an input, output, or state variable or a runtime option.

Now we will obfuscate the names of all the exposed variables except for the some specially selected ones, in this workshop we choose this to be revolute.tau again. This is done with the compiler options **obfuscate_variables** and **keep_variables**.

- Remove the two compiler options from before (exclude_internal_variables and include_internal_variables)

- Set two new compiler options:
    - obfuscate_variables : *
    - keep_variables : revolute.phi

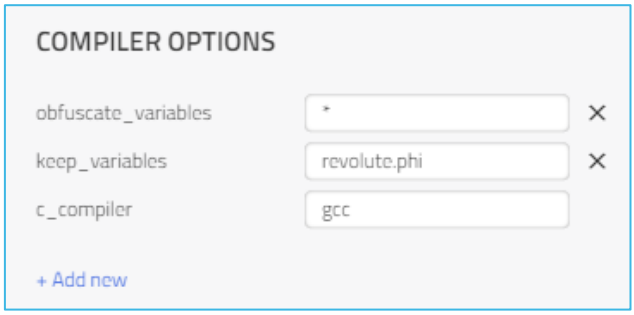


Figure 5

Export the model as an FMU and inspect the new modelDescription.xml. All internal variables should now have generic names, except for the one we selected to keep.

Multistage simulation workflow

In this part of the workshop we are considering a simple oscillating mass, connected to a spring damper. We will first calculate the spring configuration, to achieve a predefined target preload. We will then use this calculated configuration to initialize a second dynamic simulation.

It will show:

- How to read and write data from a file.
- An example of how to transfer data from one experiment to the next

Go to **W3.WorkflowMultiStageSim.Step1_Setup.**

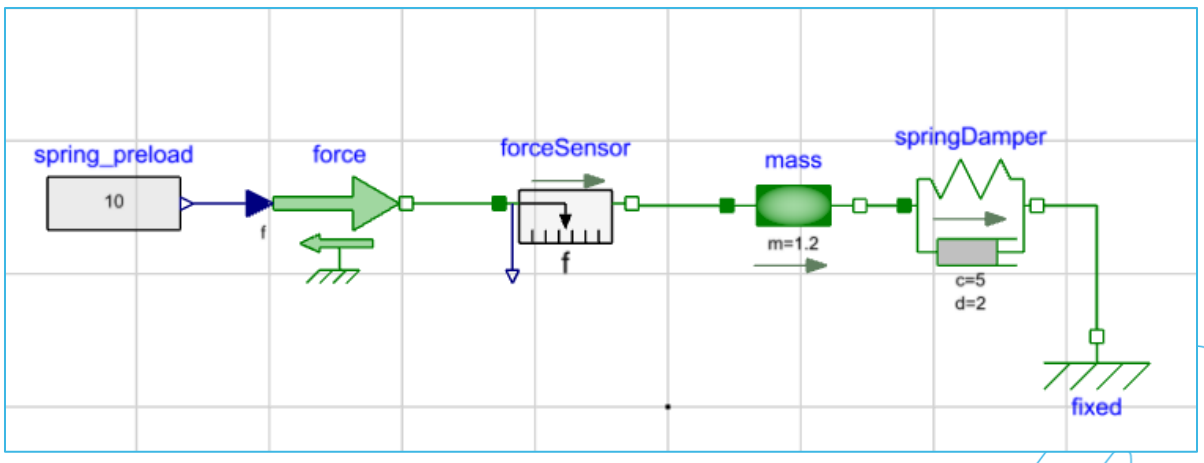


Figure 6

Simulate the model for 20s, and plot the mass position **s**.

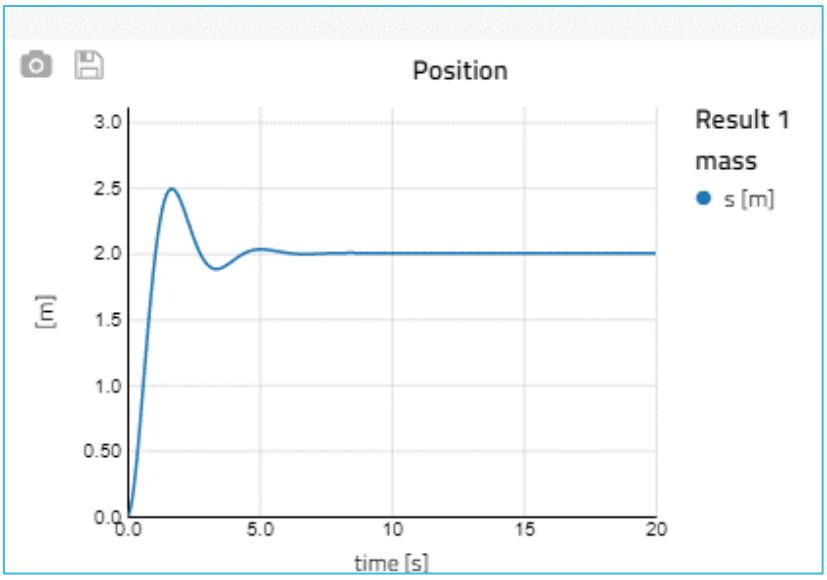


Figure 7

In the next step we will use an external xml file with data, to parametrize the model, and we want to write the final position of the mass, back into the file.

The example.xml file can be found in the Resources folder of TrainingPack.

The content of the file looks like this:

```
<?xml version="1.0"?>
<data_access>
    <m>1.2</m>
    <c>50</c>
    <d>2</d>
    <distance>0</distance>
</data_access>
```

Figure 8

Tip! You can upload your own files to the Resources folder by:

- Drag & drop it on the Resources folder
- Using the upload buttons:
- Right click on the folder and upload file/folder

Next, in **Step1_Setup**, add a position sensor (**Modelica.Mechanics.Translational.Sensors)** to the mass.

Then drag in the object **W3.WorkflowMultiStageSim.DataAccess.DataAccessBlock.**

The DataAccessBlock component contains code that opens the file, and reads and writes data. Inspect the component by right-clicking it and select “Open class” review the Modelica code using the code-editor:

```
model DataAccessBlock
    constant String filename = Modelica.Utilities.Files.loadResource("modelica://TrainingPack/Resources/example.xml");
    parameter .Modelon.DataAccess.XmlFile access(filename = filename) annotation(...);
    parameter Real m = access.getReal("m");
    parameter Real c = access.getReal("c");
    parameter Real d = access.getReal("d");
    parameter Real t_write = 0.9;
    parameter Real t_store = 1;
    .Modelica.Blocks.Interfaces.RealInput value annotation(...);
equation
    when (time >= t_write) then
        access.putReal("distance",value);
    end when;
    when (time >= t_store) then
        access.saveAsFile(filename);
    end when;
    annotation(...);
end DataAccessBlock;

```

Figure 9

You can switch back to the model “Step1_Setup” by clicking “back” in your browser:

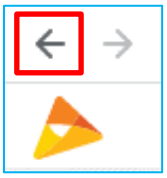


Figure 10

Connect the components as shown below:

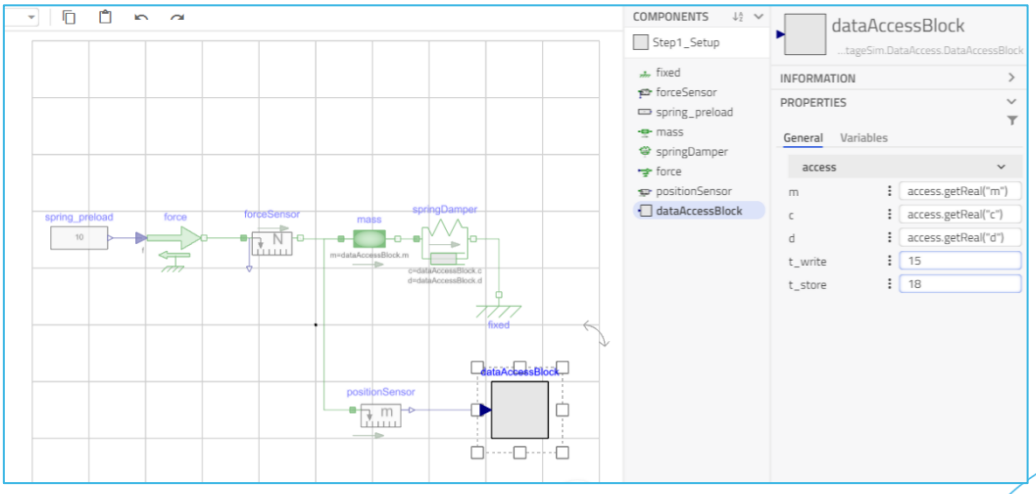


Figure 11

Inside the datablock we have the refrences m, c, d from the file, so lets use those to parameterize the system. This can be done by component referencing and the dot notation. Also note the variables t_write=15 and t_store=18. This defines when the data will be captured, and stored to file.

Open the parameter window for the **mass**, and start typing data… you will get auto completion of whats available. Choose **dataAccessBlock.m**

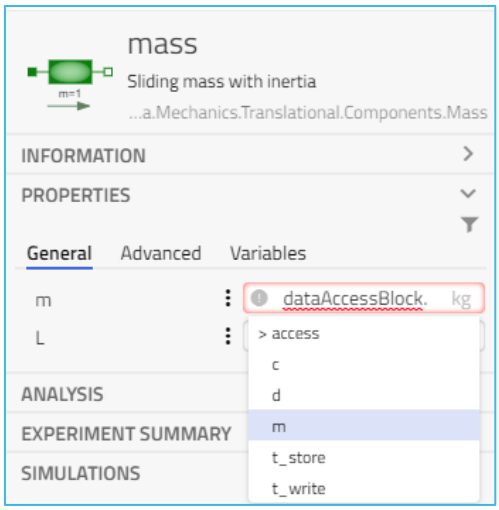


Figure 12

Do the same for c as **dataAccessBlock.c** and d as **dataAccessBlock.d** in the spring component (springDamper).

Now we need to feed the information of the final position of the mass. The Feed of the sensor signal into the data block will write the sensor data to the xml file, **distance** value.

Now run the simulation for 20s.

Download the XML file again, and see that the value of distance has changed to 0.2. This is because the dataAccessBlock wrote new data to the examples.xml file.

Now go to the next simulation step, **W3.WorkflowMultiStageSim.Step2_DynamicAnalysis**

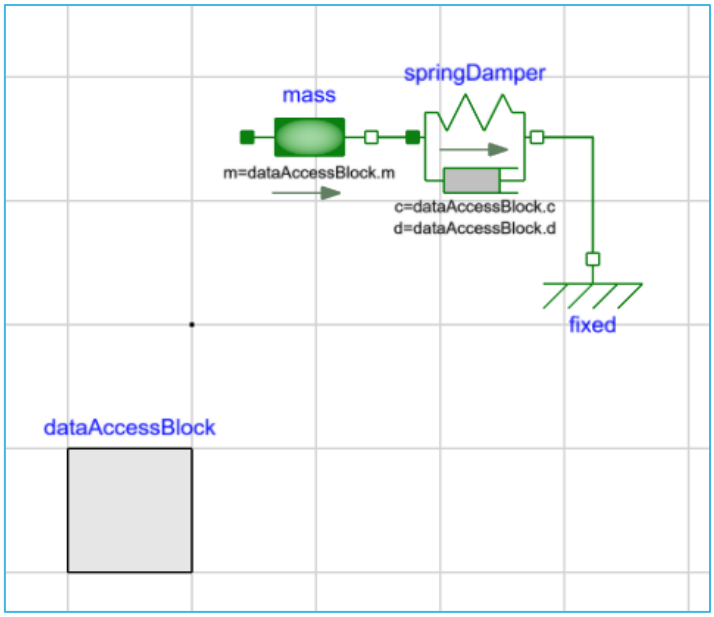


Figure 13

This model is already preconfigured with a dataAccessBlock, and parametrized.

Now we will initialize the mass using the preload distance we determined in step 1 (using the model **Step1_Setup**).

Open parameters for the mass component and find **s.** Press the three dot, to find the initial start attribute. Type the path to the **distance** value. (**dataAccessBlock.distance**). Click the checkbox to set fixed=true.

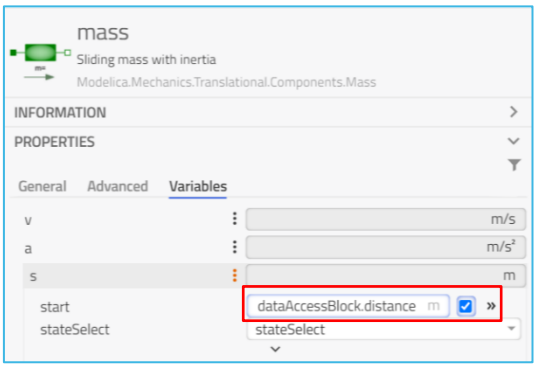


Figure 14

Run the simulation and plot the mass position.

Verify that the system initialized at the correctly obtained position.

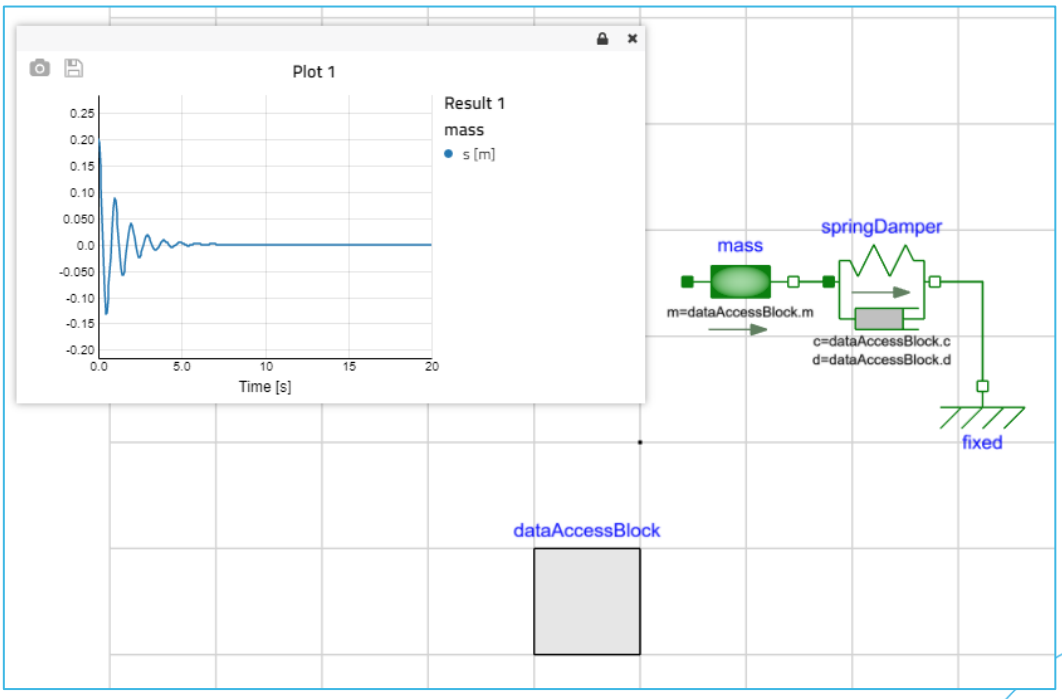


Figure 15

Parameter sweep

In this exercise we will test to set up a model with an multi-execution parameter sweep of two variables.

The model used for this workshop is **W3.MultiSim.SeriesDriveCycle**. This is a variant of the Electrification Library example, **Electrification.Examples.SeriesDriveCycle**. The example represents

the electric propulsion system of a series hybrid electric car. The system has a battery pack and two electric machines. One machine is used for accelerating the vehicle. The other is used as a generator, providing energy to the electric system from a fixed speed combustion engine. that has been pre-configured and saved in the TrainingPack.

W3.MultiSim.SeriesDriveCycle has been configured to start the simulation with 90 % Battery-SOC level, and to simulate until this reaches 10%. The driver in this example accelerates the car up to 70km/h (19.444 m/s) and holds that speed throughout the simulation.

The idea in this exercise is to sweep the battery capacity and ambient temperature, and to look at how this affects the "range", i.e. the distance that the car can drive before the simulation terminates due to the SOC falling below 10%. To set up the view for the experiment we have to add a sticky for the two parameters that are to be swept.

- Add stickies for **battery.core.capacity.Q_cap_cell_nom** and **ambient.T**

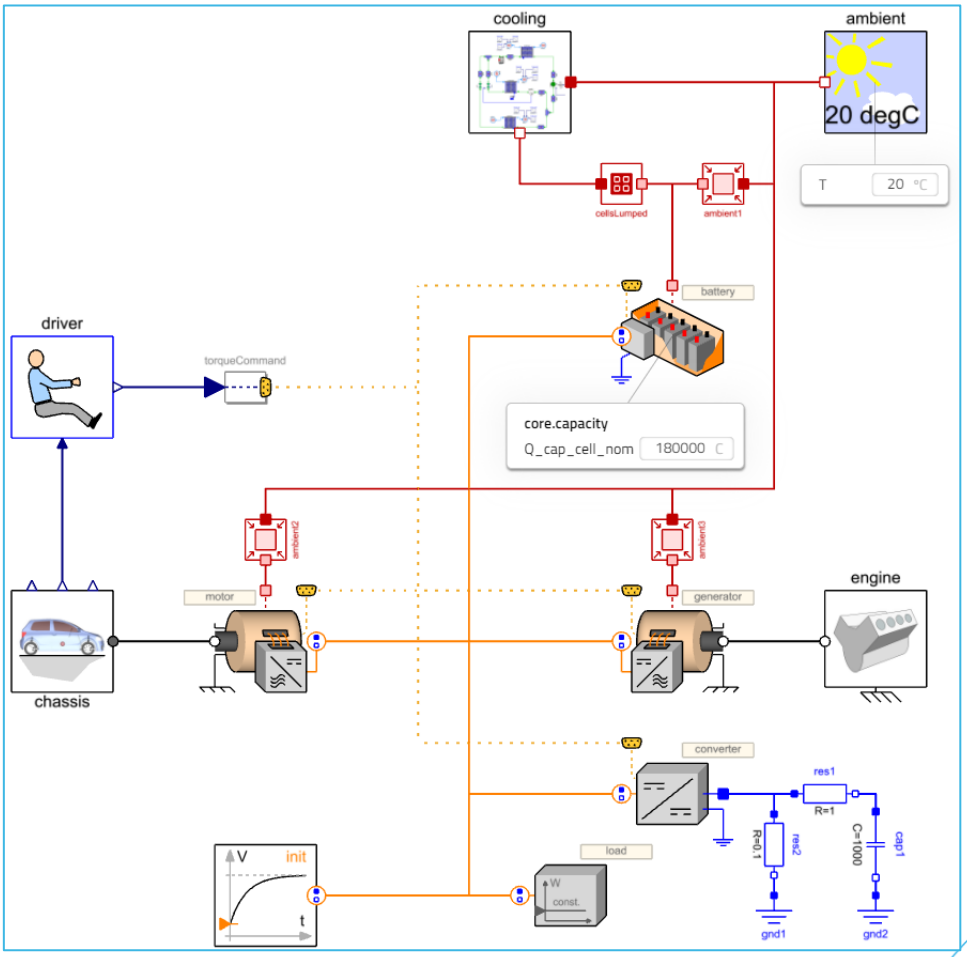


Figure 16 Stickies added for the parameters to be swept

Now we are ready to activate the **experiment mode** which is needed to define parameter ranges for the sweep.

- Click the **Experimentation** tab on top of the modelling canvas.

Parameter sweeping ranges are defined with a range-operator inside Impact. It follows the syntax: **range(start_value, stop_value, nbr_of_values)**.

- To sweep the two parameters **from the stickies on the canvas**, simply input the range operators in the parameter fields:
    - Q_cap_cell_nom: range(1e5, 2e5, 5)
    - ambient.T: range(-20,30,5)

If everything was done correctly, a summary of the experiment should be visible from the experiment tab in the control panel now. See figure below.

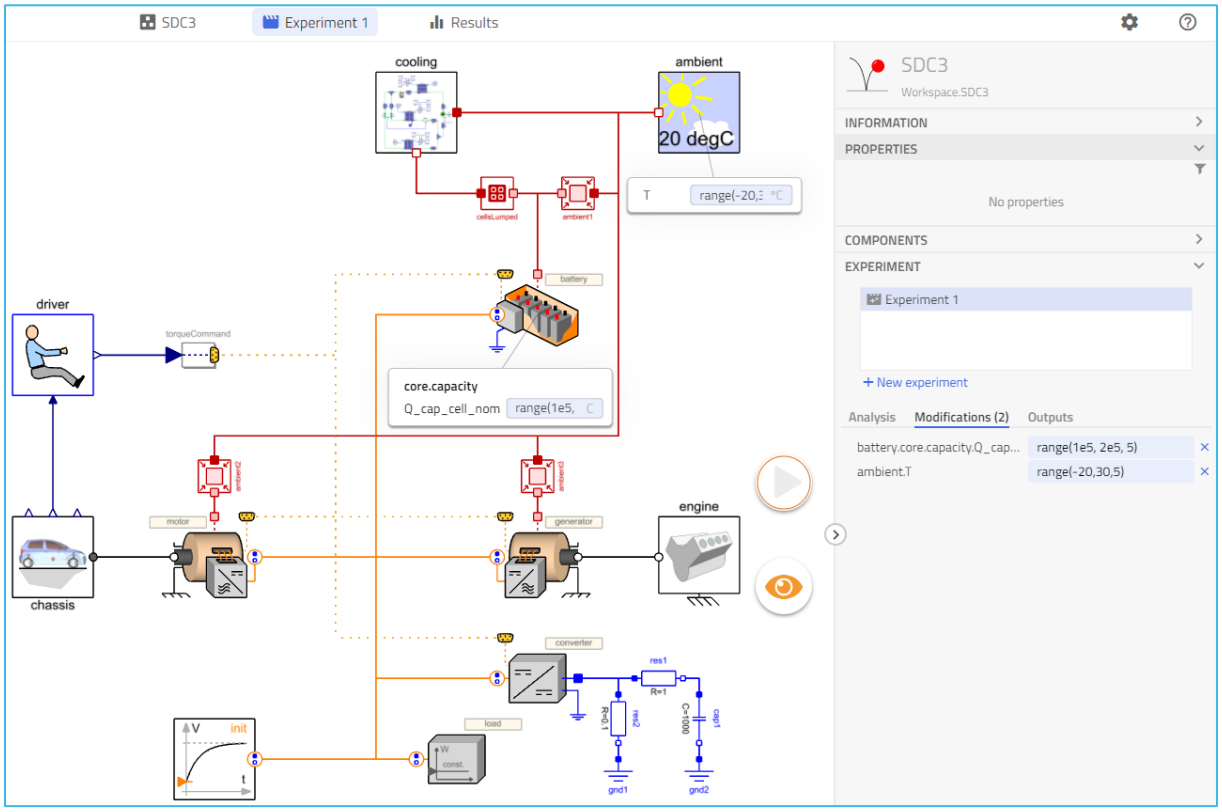


Figure 17 Stickies are used to define the experiment with range()-operators. A summary of the experiment is displayed in the Experiment tab

- Make sure that the simulation stop time is large enough for the simulation not to stop before the battery SoC criteria is met, 100000s should be enough. Also **increase the interval size to 1000 to avoid having too many result points!**
- Hit the simulation button. The simulations runs in parallell in the background, and you are given feedback how many simulations have finished.
- Plot the "range" of the vehicle by plotting the variable *chassis.position*.
- Activate the **3D** option on the plot window of the chassis.position in order to display the dependency of the range as a function of battery capacity and ambient temperature!

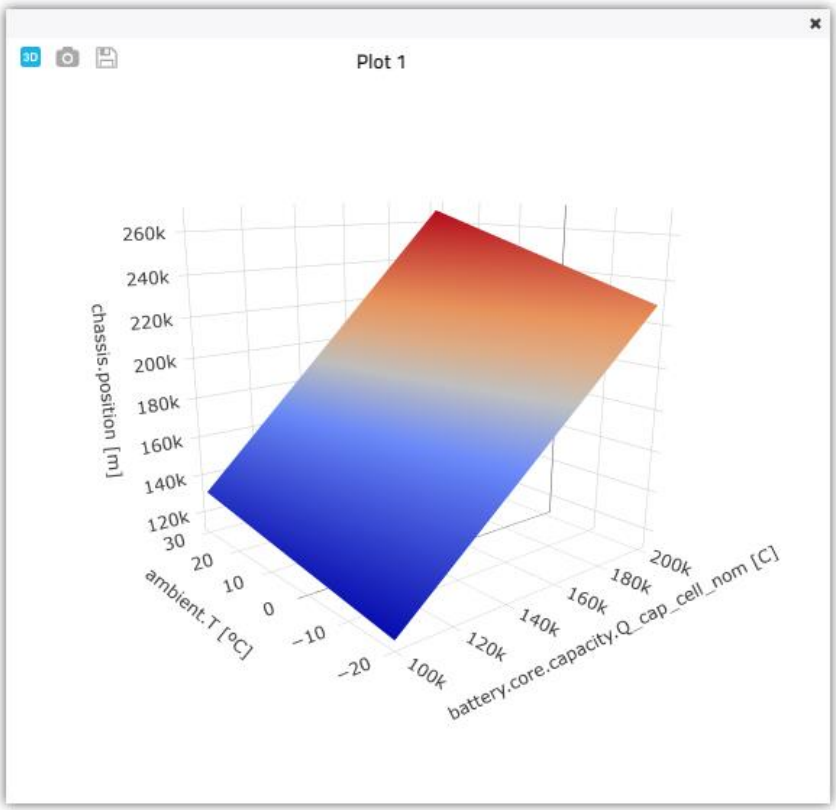


Figure 18: 3D plot of vehicle range as a function of ambient temperature and battery cell capacity.

This concludes workshop 2.3. Well done!