Modelon Impact Cloud: Tools and Add-ons

Presented by Modelon

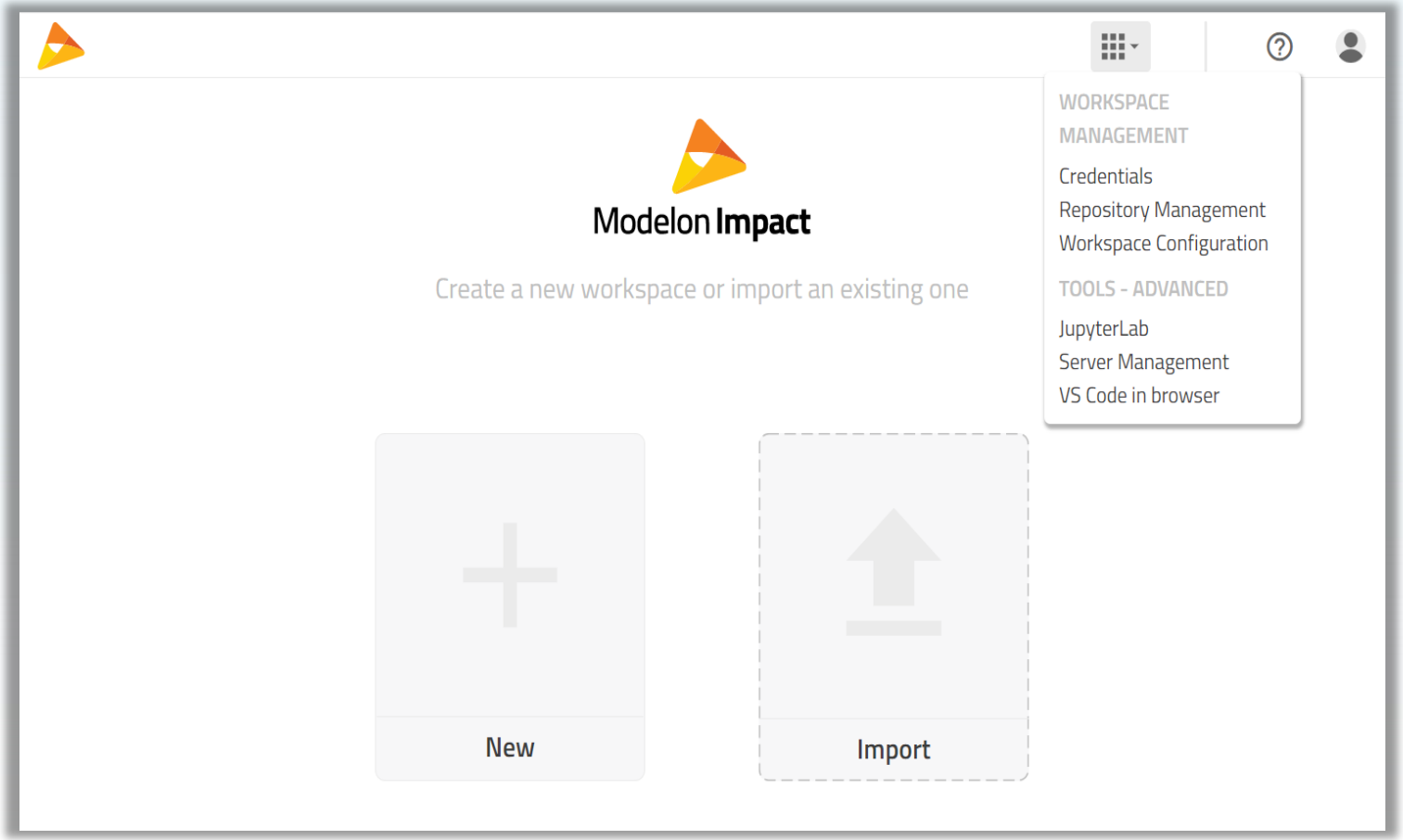
Outline

- Introduction
  - Impact Tools and Add-Ons
  - Why Scripting?
  - Scripting with Modelon Impact
- Hands-On workshop
  - Jupyter Notebook Handout
  - Follow along with Instructor

Modelon

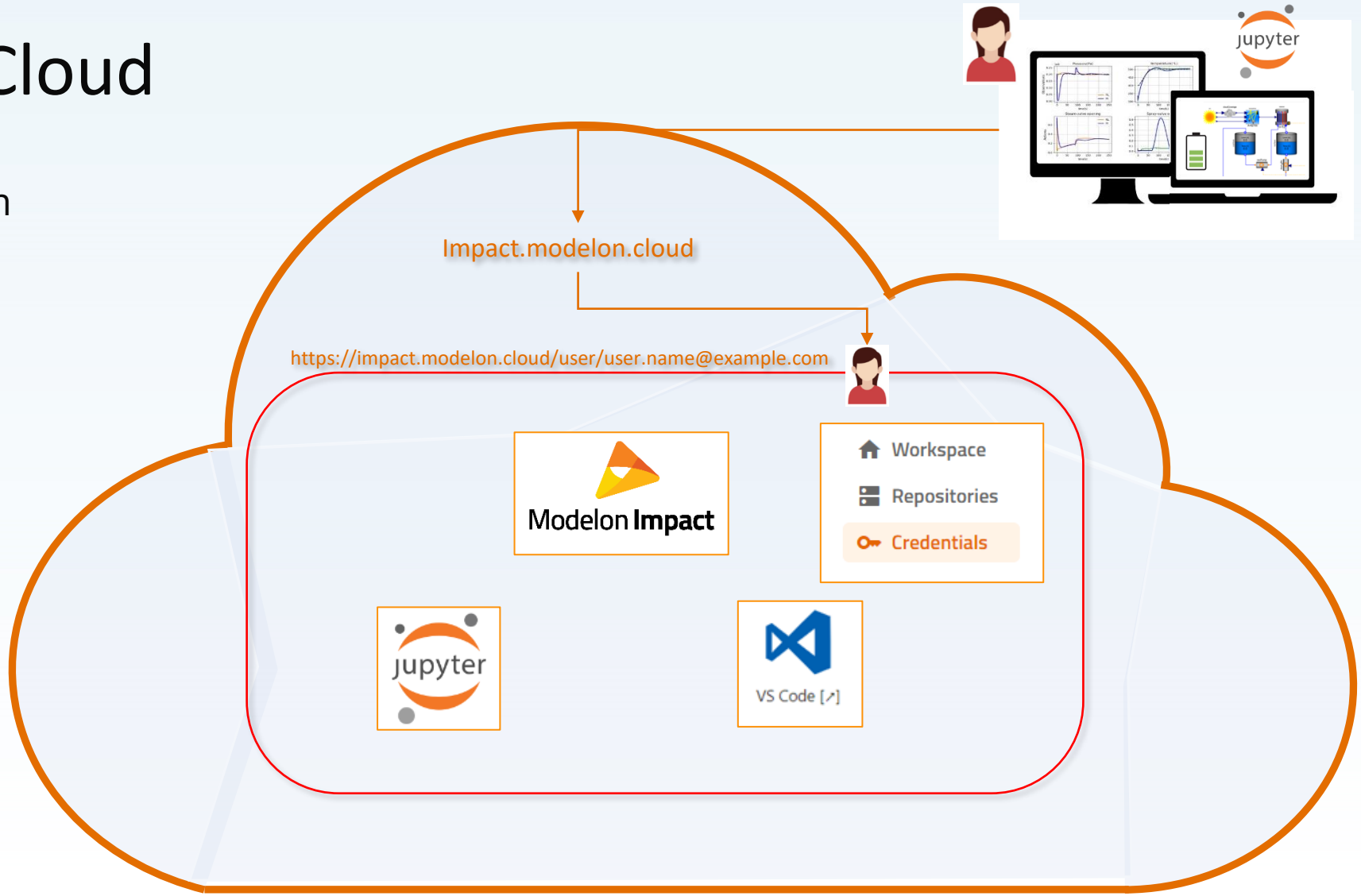
Impact Tools and Add-ons

- Productivity tools included
  - Compare vs. Desktop
- Scripting/Programming
- Version control



Modelon Impact Cloud

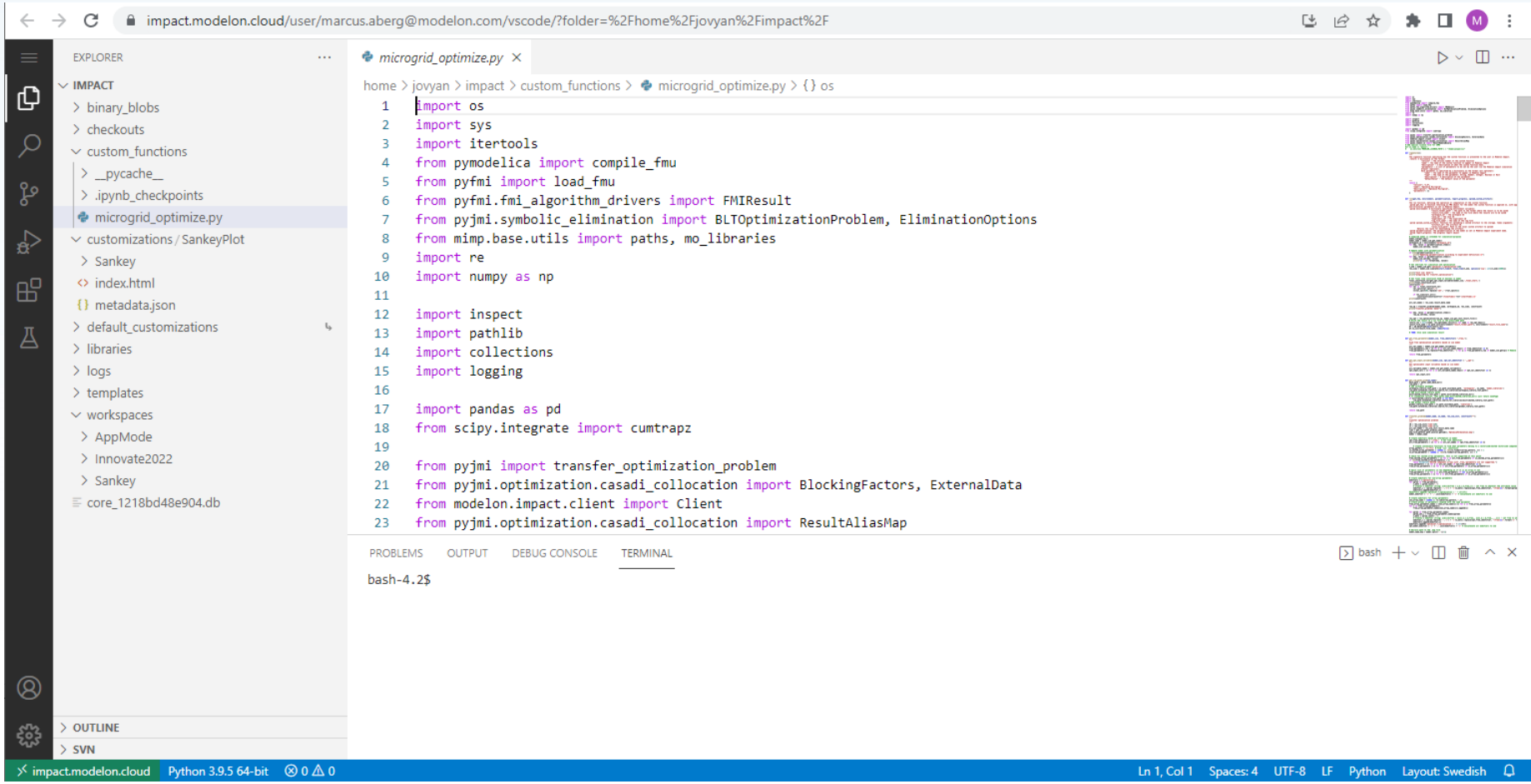
- Linux based operative system
- File Storage
- Set of pre-installed Tools
- Impact Tools:
  - Server management
  - Workspace management
- Advanced Tools:
  - VS Code
  - JupyterLab



VS Code

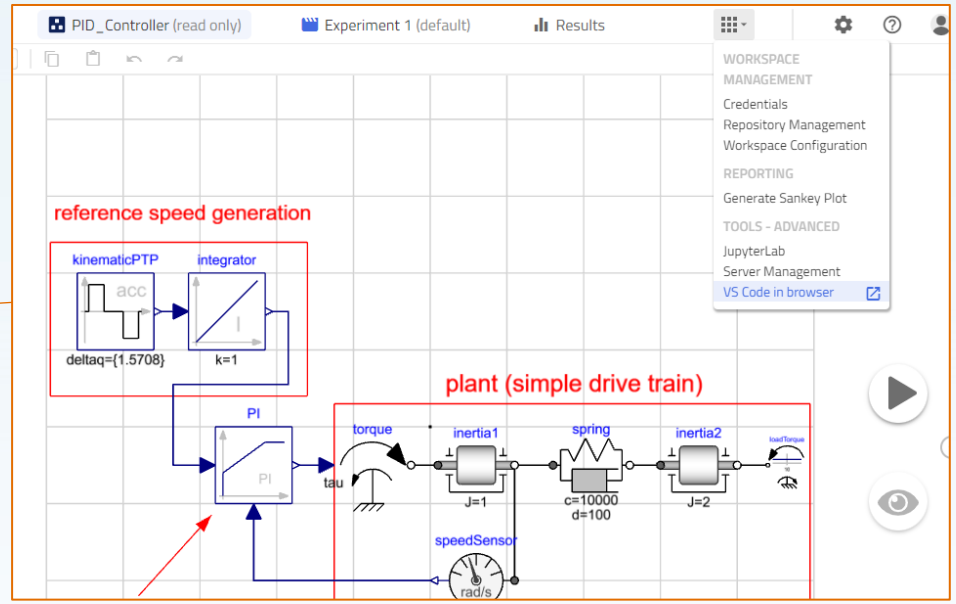
Web Version of Visual Studio Code

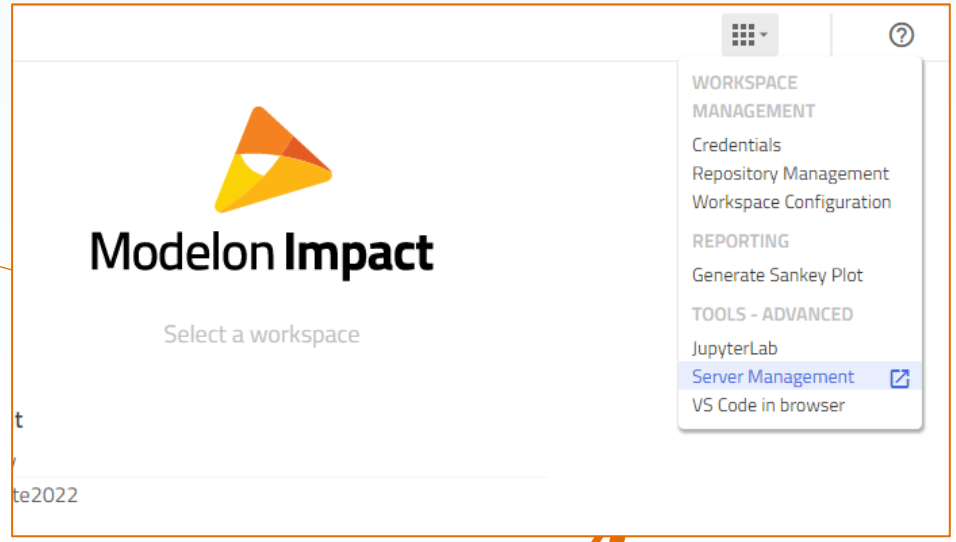
- File Explorer
- Coding IDE
- Version Control
- Debugger
- Terminal



VS Code: Opening context

- Opening from the Workspace-Context
  - Creates a relevant VS Code working environment
    - Current workspace folder
    - All version-controlled checkouts
      - More in version control workshop!
- Opening from Landing screen
  - Opens the root folder of Impact
  - /home/jovyan/impact
- Change context from within VS Code

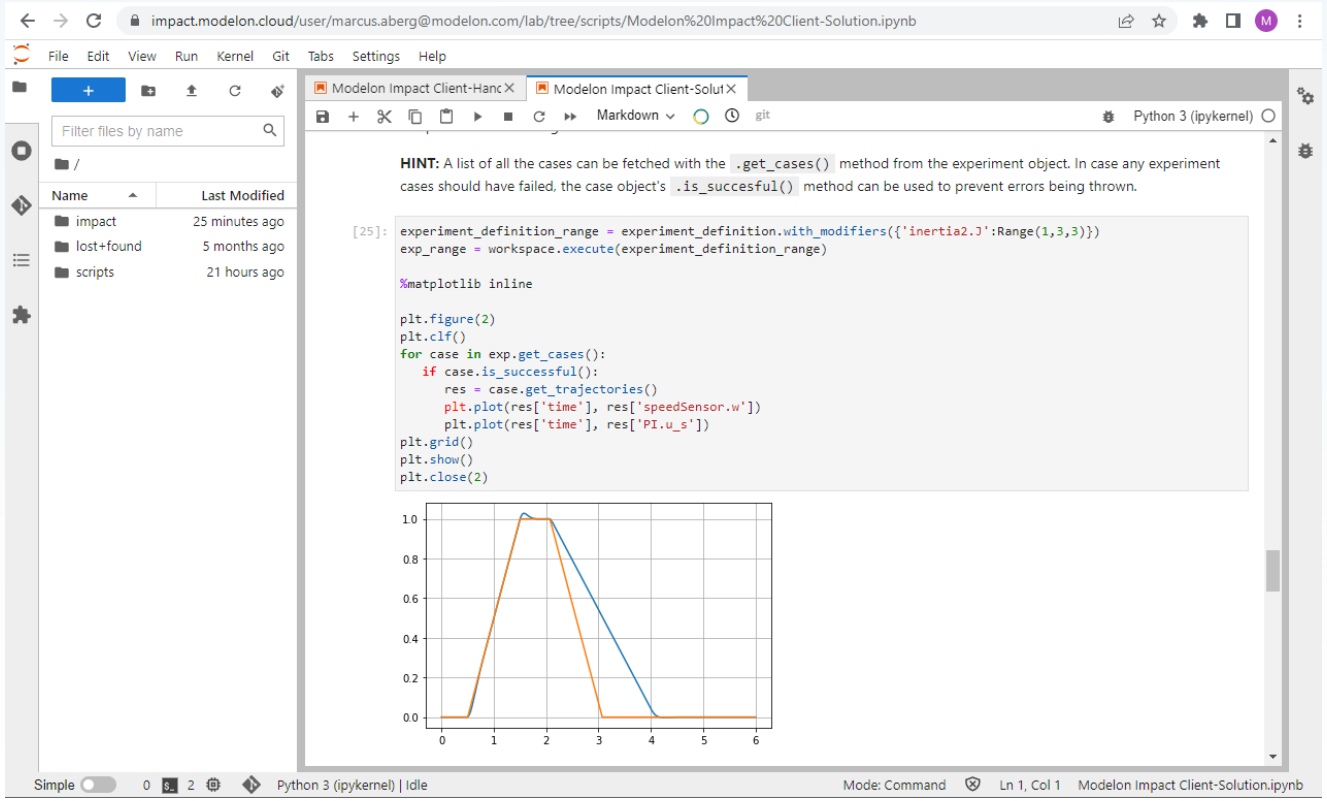




JupyterLab

Interactive development environment

- File Explorer
- Notebook Editor
- File Editor
- Version Control
- Terminal
- Python Shell



JupyterLab: Notebooks

Consists of Cells:

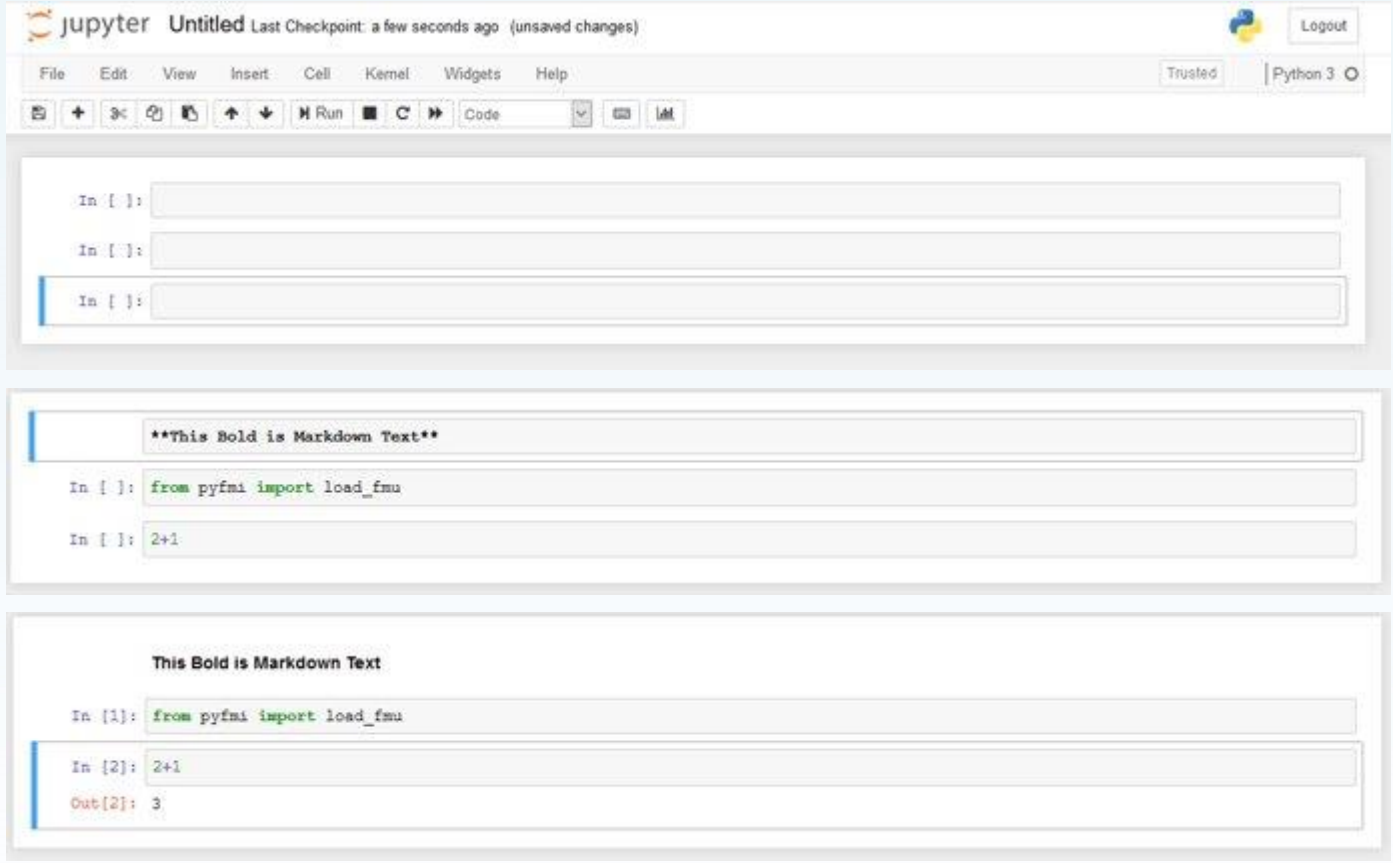
- Either Code or Markdown
- Cells executed in arbitrary order
- Default language is Python

Ideal for:

- Scripting and data visualization

Widgets:

- Simplify user interaction
- Makes Notebooks more “Appy”

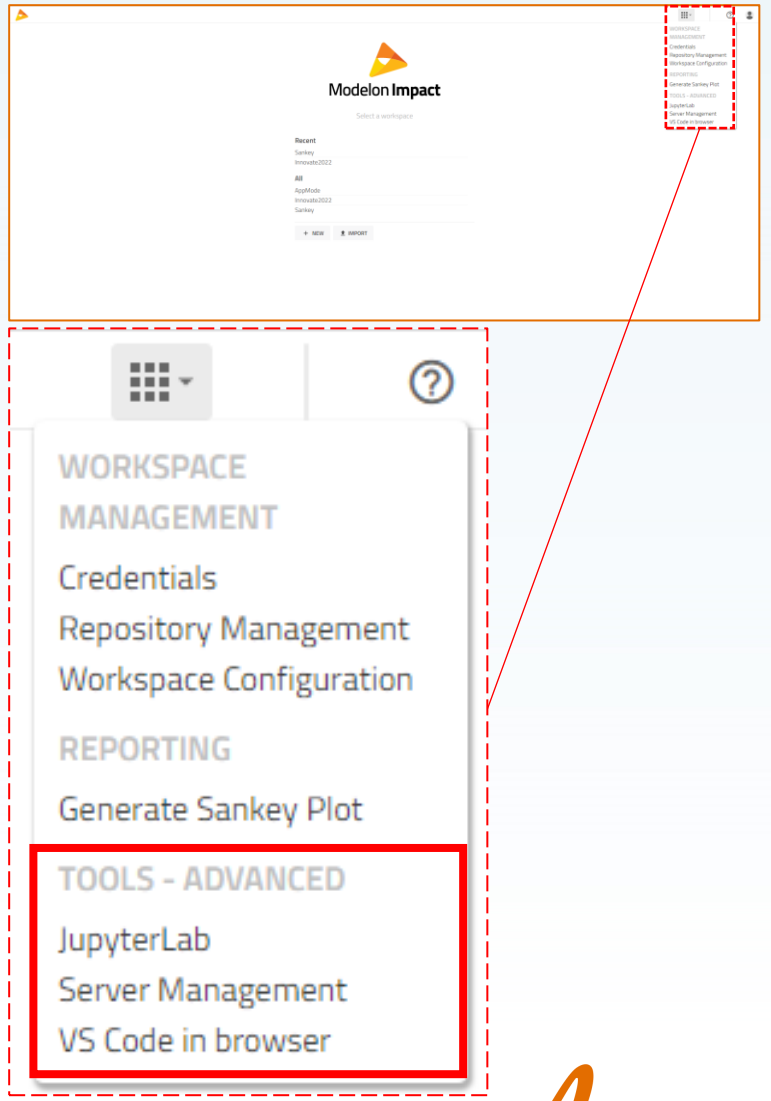


Access to Tools and Add-Ons

Access through apps-dropdown

- Welcome screen
- Inside Workspace

All tools have specific URLs (Bookmark)



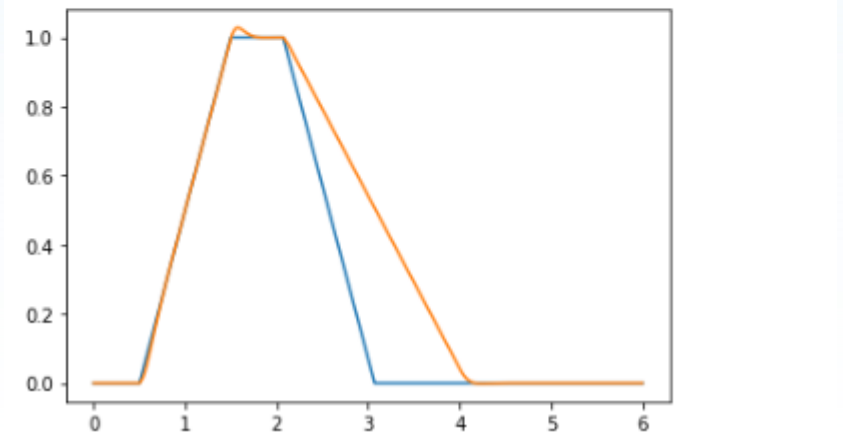
Why scripting?

For most MBSE workflows, automatization and scripting functionality is essential.

- Pre-processing
- Post-processing
- Scale up of simulations/compilation:
- Model calibration
- Model verification

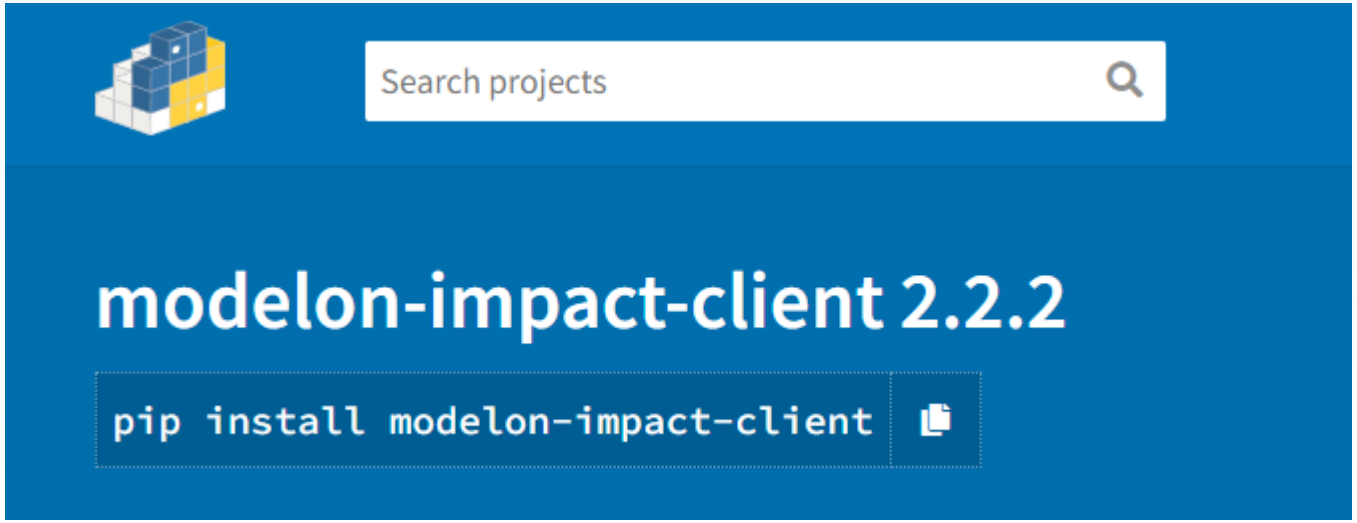
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(1)
case = exp.get_case('case_1')
res = case.get_trajectories()
plt.plot(res['time'], res['PI.u_s'])
plt.plot(res['time'], res['speedSensor.w'])
plt.show()
plt.close(1)
```

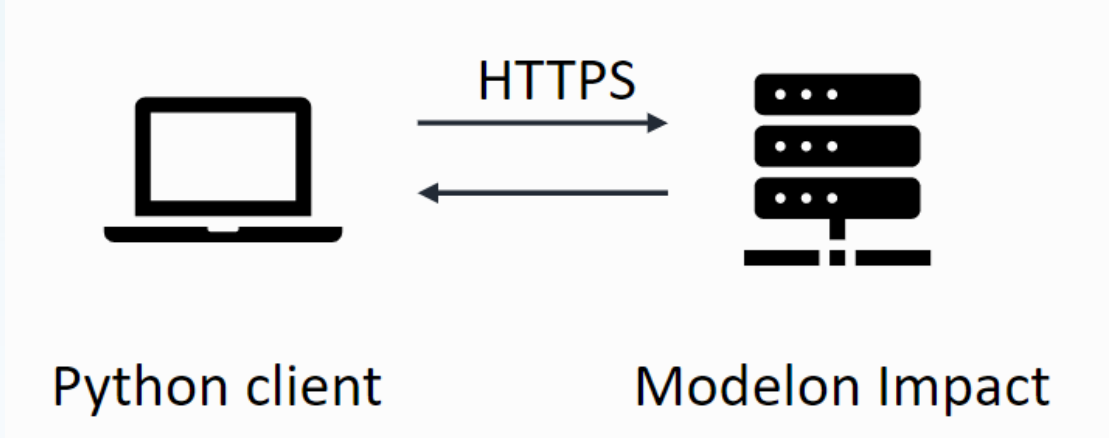


How to script with Modelon Impact?

- Python is the recommended scripting language for Modelon Impact
- Modelon develops and maintains a Python Module:
  - **Modelon Impact Client**
- Installation through PIP



Modelon Impact Python Client

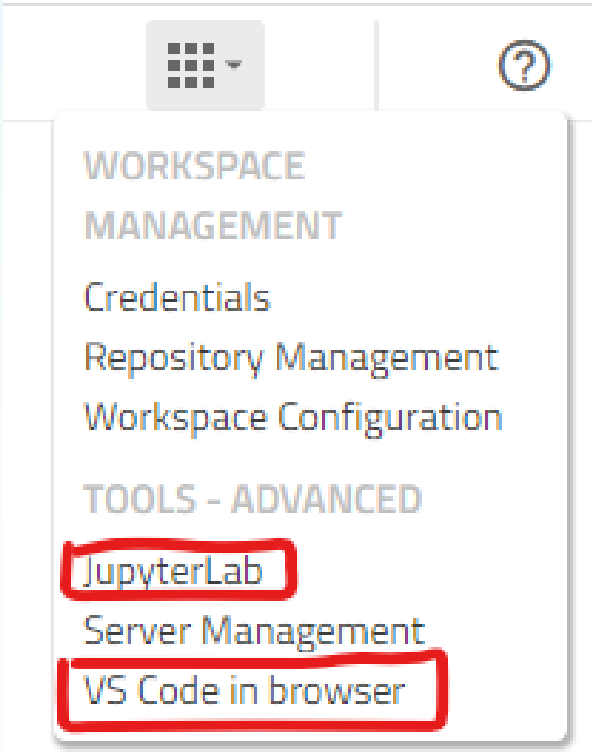


Client Features:

- Authentication
- Adding and extracting artifacts from Modelon Impact
  - Results, Models, Libraries etc.
- Perform Experiments on the Server
- Let Modelon Impact do the Simulation work, you worry about analysis
- Experiment definition format to set up batch simulations.
  - Highly flexible, yet intuitive API
  - Modelon Impact handles parallelization
- WEB API – communicates over the WEB with the Modelon Impact Server

Scripting in the Cloud

- Modelon Impact Cloud comes with apps that enable scripting:
  - JupyterLab
  - VS Code
- **Modelon Impact Client** is pre-installed into these environments
- In the workshop mainly **JupyterLab** environment will be used



Workshop

Getting started with scripting in Modelon Impact

Modelon

Workshop Logistics

- Open JupyterLab from the Apps-dropdown
- Download the workshop notebook from https://help.modelon.com
- Upload it to JupyterLab
- Follow along in the Notebook and add/edit your own code as we go along

Detailed Reference Slides

General Workflow

When executing experiments on the Impact server with the Modelon Impact Client library for python, a general workflow could be outlined which will be covered in the coming slides:

- Generate API-key using the API key manager
- Initialize the client object
- Set up the workspace
- Defining your analysis
- Set up model for analysis
- Analysis
    - FMU based workflow
    - Class name based workflow
- Setting up simulation series
    - Operators
    - Experiment extensions
- Executing
- Fetching/visualizing the result

Modelon

Generate API Keys

Impact Key:

- https://impact.modelon.cloud/user-redirect/impact/admin/keys

Jhub key:

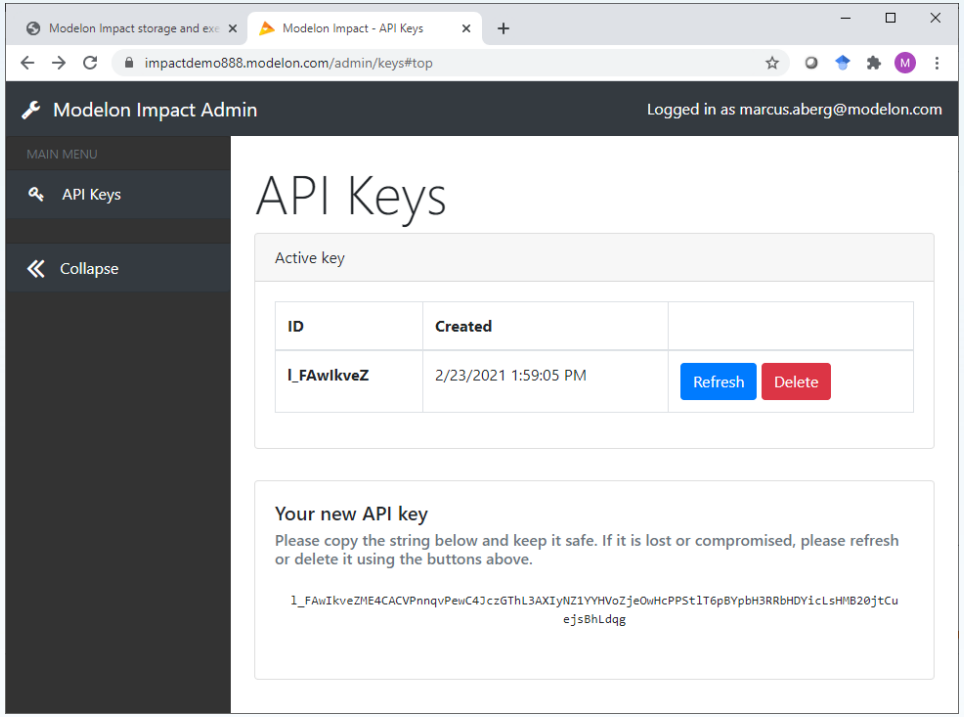
- https://impact.modelon.cloud/hub/token

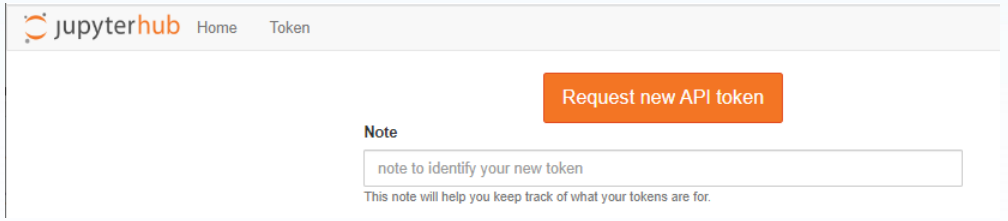
Requires the user to be logged in on the relevant Impact instance.

The keys are linked to the current user

- This means when using the client, you can access workspaces, models etc. stored in your user.

Key is only showed once, if lost it needs to be refreshed (old key becomes inactive)





The Client Object

- Initializes the connection between the client and the Impact domain
  - Initializes connection with Impact server hosted at the given Impact URL
  - Authenticates the session with API-key
- Key is stored in .impact folder in the user's home folder for future access
  - The user only need to give the api keys once on the same machine

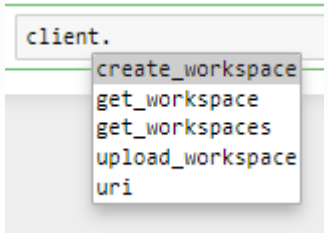
```
from modelon.impact.client import Client

client = Client(url="http://localhost:8080/", interactive=True)
```

```
WARNING:modelon.impact.client.client:The provided API key is not valid, please enter a new key

Enter API key:
```

Methods include:



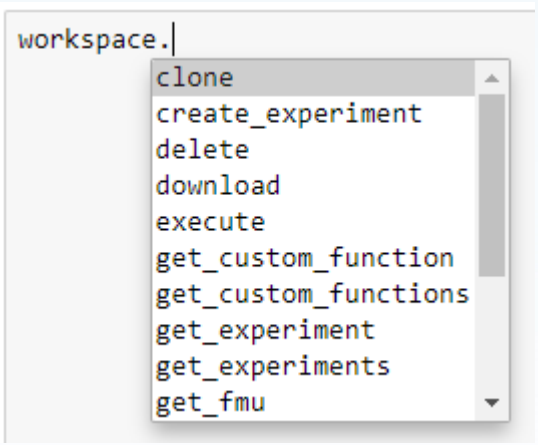
- Workspace handlers

Set up the workspace

- All operations on models needs to be done in scope of a workspace
- Instantiate a workspace object from the client object:

```
workspace = client.create_workspace("ws_name")
```

- Methods include:



- Workspace operations
- Getters for workspace resources
- Experiment execution

Modelon

Define analysis

- The workspace has several different analyses it can run on the models
  - Referred to as “Custom Functions”

```
dynamic = workspace.get_custom_function('dynamic')
```

- Each Custom Function has a set of recommended options for compilation and simulation:

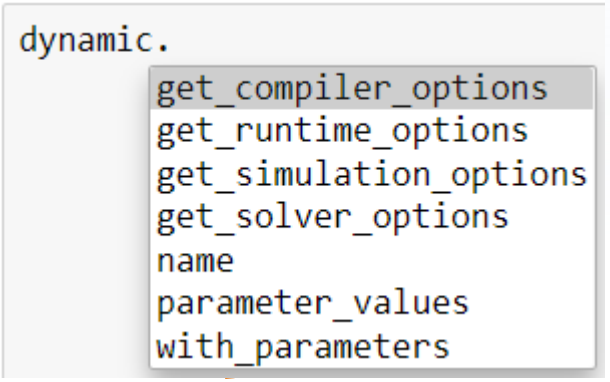
```
compiler_options = dynamic.get_compiler_options()
runtime_options = dynamic.get_runtime_options()
```

- Modification is possible:

```
compiler_options_modified = compiler_options.with_values(c_compiler='gcc')
```

```
compiler_options_modified = {'c_compiler':'gcc'}
```

Methods available:



Set up model for analysis

- Declare model from workspace to run analysis on:

```
model = workspace.get_model("Modelica.Blocks.Examples.PID_Controller")
```

- If the model exists only on the local machine, we can upload it first:

```
workspace.upload_model_library("PID_Controller.mo")
model = workspace.get_model("PID_Controller")
```


Modelon

Analysis

With the model set up we can start the experimentation/analysis

There are currently two workflows for this supported:

- FMU based workflow:
  - Requires a first compilation step
  - Experiment is defined on the compiled FMU
  - Preferred when control over re-compilations are prioritized
- Class name-based workflow:
  - Uses the model object directly to set up the experiment definition
  - Allows for changing structural parameters (requires re-compilation)
  - Preferred when re-compilation is needed in the experiment

Modelon

Analysis: Fmu based workflow - Compilation

- Compile the model using compiler options created earlier:

```
compiler_options = dynamic.get_compiler_options()
runtime_options = dynamic.get_runtime_options()
```

```
compiler_options_modified = compiler_options.with_values(c_compiler='gcc')
```

```
fmu = model.compile(compiler_options=compiler_options_modified,runtime_options=runtime_options).wait()
```

- Returns reference to the FMU (runnable model) on the server side

Optional arguments:

```
model.compile()
```

```
Signature:
model.compile(
    compiler_options,
    runtime_options=None,
    compiler_log_level='warning',
    fmi_target='me',
    fmi_version='2.0',
    platform='auto',
)
```

Analysis: Asynchronous programming - .wait()

```
fmu = model.compile(compiler_options=compiler_options_modified,runtime_options=runtime_options).wait()
```

?

- .compile is an asynchronous call to the server api.
- .wait() method is used to ensure that the compilation process reaches completion
- If wait() is not called Operation object is returned:
    - is_complete() can be used to check the status of the compilation
- .wait() method returns a ModelExecutable object which represents the compiled model (FMU) on the server side

To illustrate how the wait() method functions we can use an Operation object instead to check the status of the operation.
When it is done the ModelExecutable object is created from the Operation object with the data() method.

```
from time import sleep

operation = model.compile(compiler_options=compiler_options_modified,runtime_options=runtime_options)
while not operation.is_complete():
    print(operation.status())
    sleep(0.5)

print(operation.status())
fmu = operation.data()
```

```
Status.RUNNING
Status.RUNNING
Status.RUNNING
Status.DONE
```


Modelon

Analysis: Fmu based workflow – Experiment Definition

In the impact client experiments are declared using so called “experiment definition” objects. They contain all necessary information for Impact to run a specific analysis on a specific model.

- From the generated model executable object, a new experiment definition object is created:

```
experiment_definition = fmu.new_experiment_definition(dynamic)
```

- Custom Function parameters could be changed with .with_parameters():
  - For example, for ‘dynamic’ these are ‘start_time’ and ‘final_time’

```
experiment_definition = fmu.new_experiment_definition(dynamic.with_parameters(start_time=0.0, final_time=6.0))
```

- Modifiers to the model used in the experiment definition can be applied with .with_modifiers():

```
experiment_definition = experiment_definition.with_modifiers({'inertia1.J': 2.0})
```

Optional arguments:

```
fmu.new_experiment_definition()
```

```
Signature:
fmu.new_experiment_definition(
    custom_function,
    solver_options=None,
    simulation_options=None,
    simulation_log_level='WARNING',
)
```


Modelon

Analysis: Class name-based workflow – Experiment Definition

- Similar call as in FMU based workflow (but on the model object):

```
model = workspace.get_model("PID_Controller")
model.new_experiment_definition(dynamic)
```

- When used like this, the experiment definition can take arguments related to compilation:

```
Signature:
model.new_experiment_definition(
    custom_function,
    *,
    compiler_options=None,
    fmi_target='me',
    fmi_version='2.0',
    platform='auto',
    compiler_log_level='warning',
    runtime_options=None,
    solver_options=None,
    simulation_options=None,
    simulation_log_level='WARNING',
)
```


Modelon

Setting up simulation series

There are two methods of setting up series of simulations:

- Operators:
  - Uses an operator to declare parameter values
  - Uses a full factorial combination of the parameter values
- Extensions
  - Highly flexible and parametrizable way of declaring simulation series
  - Every simulation case can be tailored to use specific options and parameter values

Modelon

Setting up simulation series: Operators

Instead of declaring an explicit parameter value for the modifier, we can instead use a range of values.

```
from modelon.impact.client import Range

experiment_definition = experiment_definition.with_modifiers({'inertia1.J': 2,'PI.k': Range(10.0, 100.0, 3)})
```

Currently supported operators are:

- Range: `Range(start_value, end_value, no_of_steps)`
- Choices: `Choices(*values)`

Setting up simulation series: Extensions

Extensions provide a highly customizable interface of declaring simulation series.

Simple approach:
The simplest approach is to use the experiment definitions .with_cases() method with the modifiers as input:

```
experiment_definition = experiment_definition.with_cases([{'PI.k': 20}, {'PI.k': 30}])
```

Advanced:

For more advanced schemes, extensions can be declared by instantiating SimpleExperimentExtension objects. These can be parameterized with:

- custom function parameters
- solver and simulation options
- simulation log level

```
from modelon.impact.client import SimpleExperimentExtension

experiment_extension_1 = SimpleExperimentExtension(
   parameter_modifiers={'final_time': 2.0},
   solver_options={'atol': 1e-9},
   simulation_options=dynamic.get_simulation_options().with_values(ncp=1500),
)
experiment_extension_2 = SimpleExperimentExtension(
   parameter_modifiers={'final_time': 5.0},
   solver_options={'atol': 1e-10},
   simulation_options=dynamic.get_simulation_options().with_values(ncp=1200),
)
```

Add variable modifiers for each of these cases using the .with_modifiers() method:

```
experiment_extension_1 = experiment_extension_1.with_modifiers({'PI.k': 25})
experiment_extension_2 = experiment_extension_2.with_modifiers({'PI.Ti': 5})
```

The extensions are given as a list argument to a “base”-experiment definition to create a new experiment definition including all the extensions:

```
experiment_definition = experiment_definition.with_extensions(
   [experiment_extension_1, experiment_extension_2]
)
```


Modelon

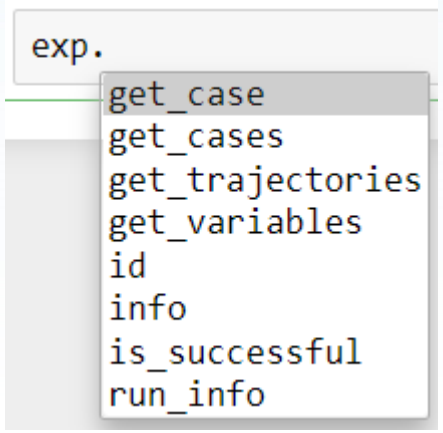
Executing the experiment

When the experiment definition is finalized, it is passed to workspace objects .execute() method:

```
exp = workspace.execute(experiment_definition).wait()
```

.wait() is used for a similar purpose as in the compilation case.

Upon completion an Experiment object is returned:



- Info on how the execution went
- References to the results of the experiment cases

Fetching/Visualizing the results

To fetch the case trajectories for a given experiment the .get_cases() method can be called on the experiment.

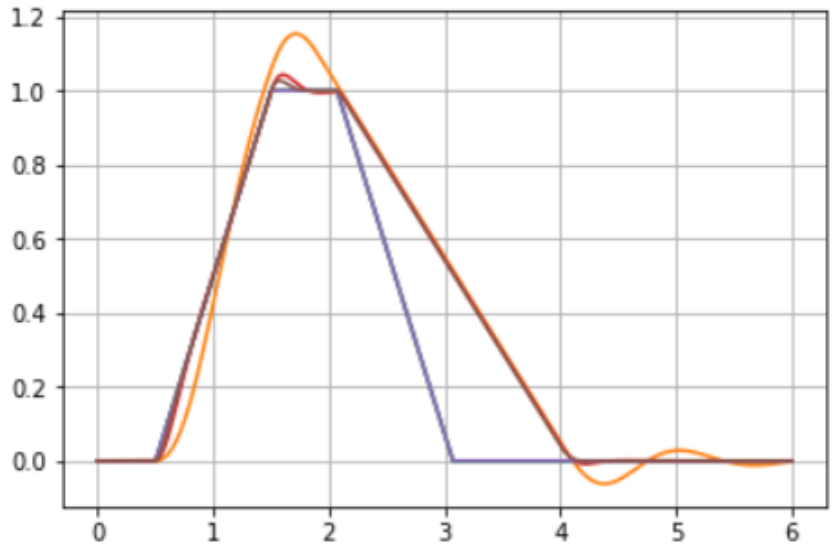
It is a good idea to check if the cases did simulate successfully by calling the .is_successful() method on the case.

The .get_trajectories() function is called on the individual case objects to fetch the Result class object for that specific case.

To get a specific variable, the variable names of interest are passed as index variables on the Result class object, for example: res['time']

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(2)
plt.clf()
for case in exp.get_cases():
    if case.is_successful():
        res = case.get_trajectories()
        plt.plot(res['time'], res['integrator.y'])
        plt.plot(res['time'], res['inertia1.w'])
plt.grid()
plt.show()
plt.close(2)
```



Accurate Simulations. Better Decisions.