arbor

HANDS-ON

A morphologically-detailed neural network simulation library for contemporary high performance computing architectures

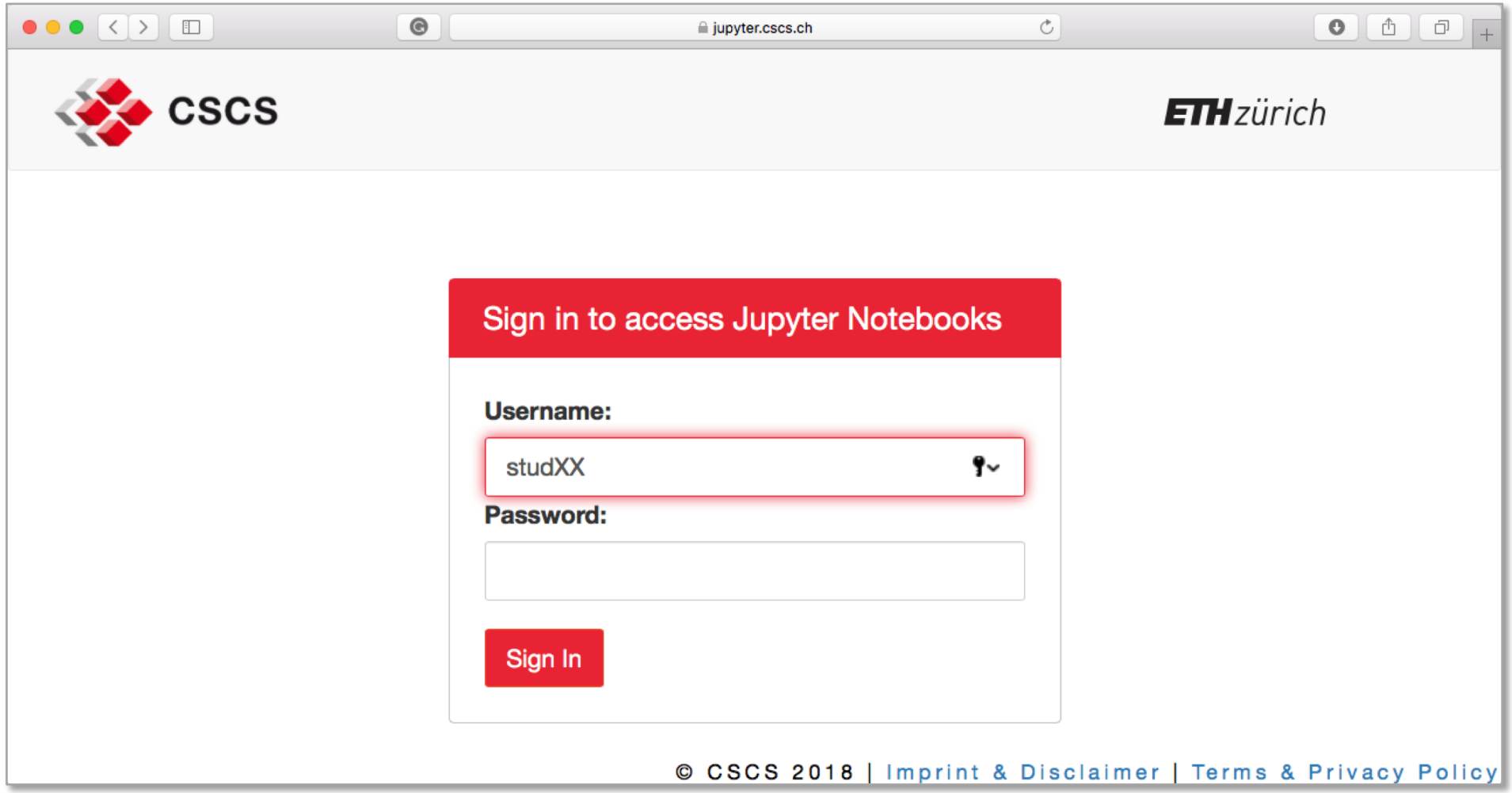
12TH DECEMBER 2018 I ANNE KÜSTERS & ALEXANDER PEYSER

Human Brain Project

Co-funded by the European Union

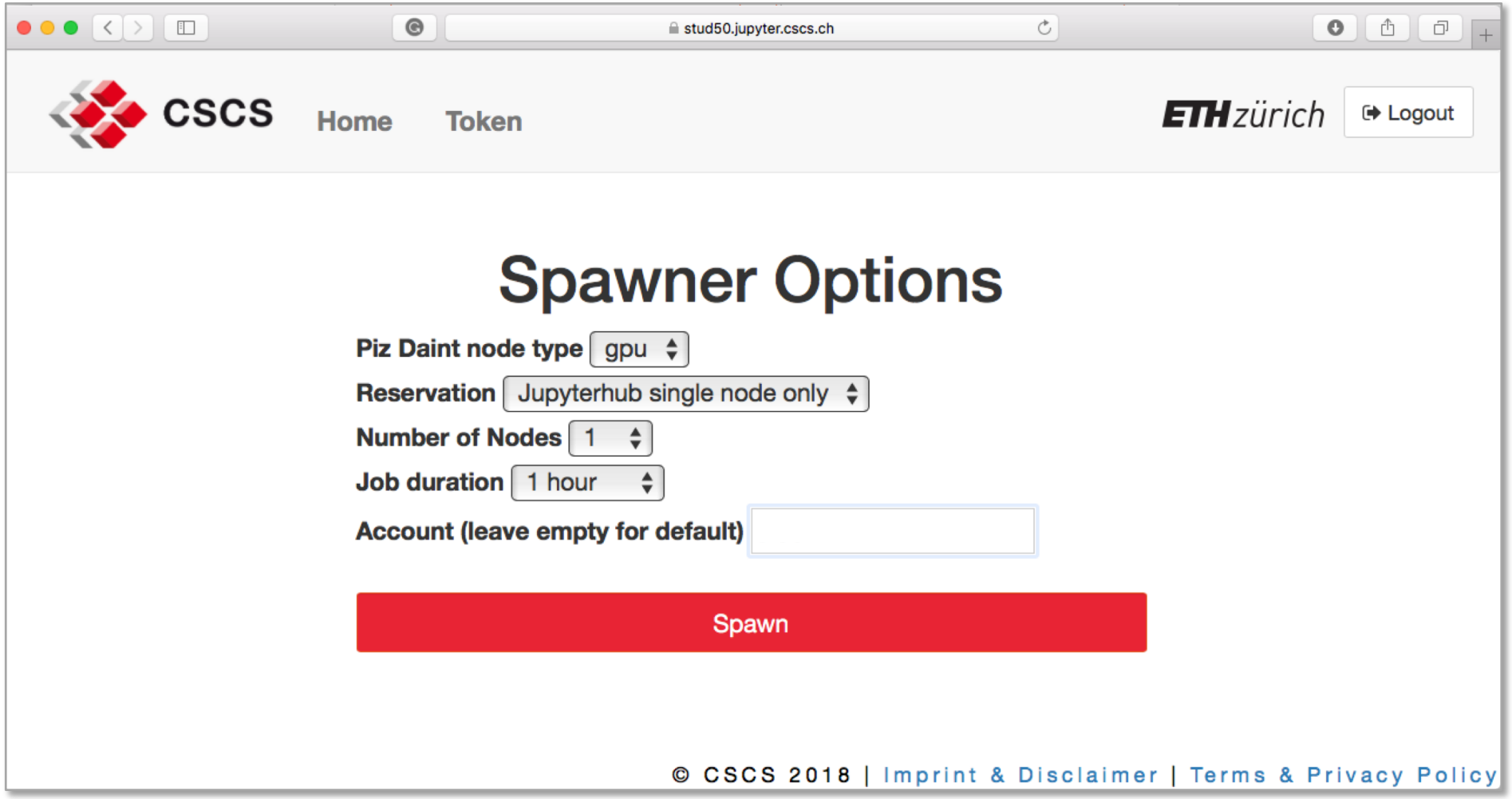
PREPARATION

Sign in to `https://jupyter.cscs.ch` with your training username and password



PREPARATION

Spawn a GPU with reservation of one node for one hour



ORGANIZATION AND WORK-FLOW

4 tasks, each dependent on the previous

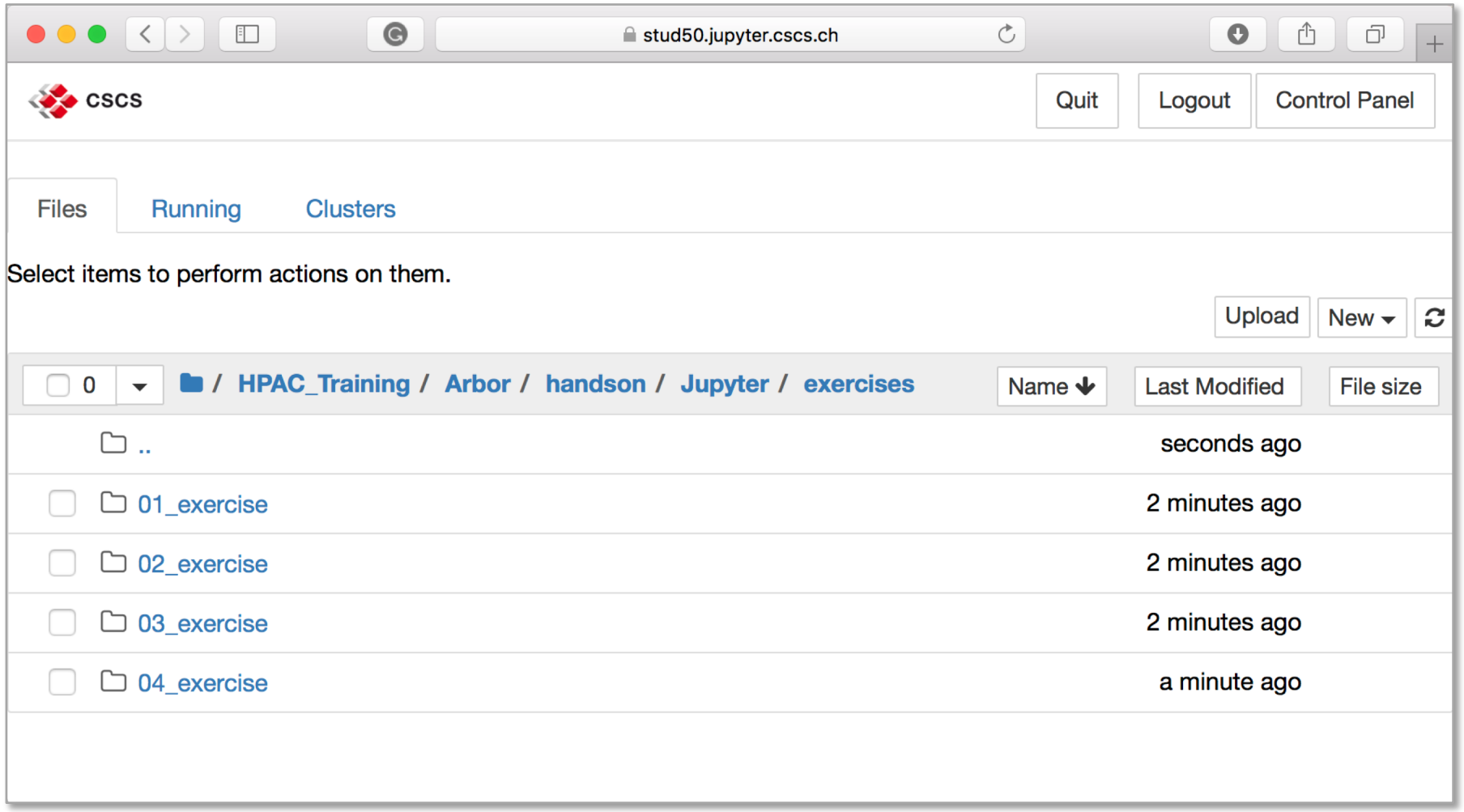
Structure:

- On your training account, go to:
  - `HPAC_training` → `Arbor`
  - `handson` → `Jupyter` → `exercises`
  - `0X_exercise` (where X is the exercise number 1-4)
- Here, you find a python script with your `#TODO#`
- Each exercise builds up on the previous exercise
  - Work on your TODO`s and save your solution to the next exercise folder
  - In case your solution does not work (after really trying and asking tutors), use provided script in next exercise

Source of picture: flaticon.com

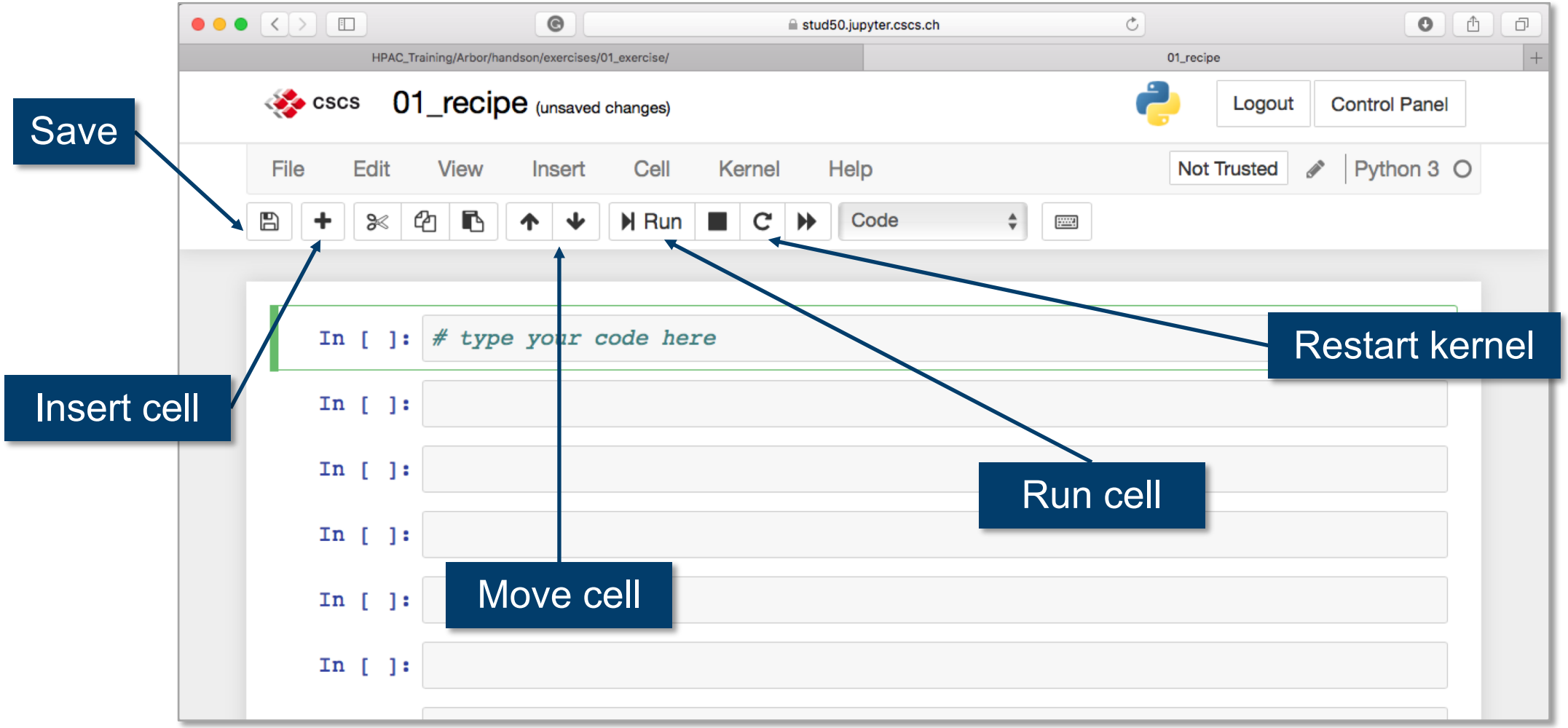
ORGANIZATION AND WORK-FLOW

4 tasks, each dependent on the previous



SHORT INTRODUCTION OF JUPYTER NOTEBOOKS

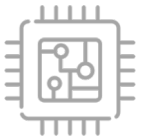
Easy to use



OVERVIEW OF EXERCISES

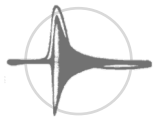
Exercises follow steps to setup and run a simulation with Arbor`s python frontend

1. Describe neuron model by defining a **recipe**



2. Get the **resources**, create the parallel **execution context** and part the work into segments by partitioning the **load balance**

3. **Initiate the simulation** over the distributed system and **run** the simulation



4. (OPTIONAL:
Set up measurement meters, get spike **recorder and its spikes** to print meter report and spike times of the cells)

Source of pictures: flaticon.com

RECIPE

Distinction between the description of a model, and the execution of a model (simulation)

A `recipe` is a description of a model. The recipe is queried during the model building phase to provide cell information, such as:

- the *number of cells* in the model as input argument (`num_cells`)
- the *type* of a cell (`kind`)
- a *description* of a cell with soma, synapses, detectors, stimuli (`cell_description`)

- optionally, e.g.:
  - the number of spike *targets* (`num_targets`)
  - the number of spike *sources* (`num_sources`)
  - incoming network *connections* on a cell (`connections_on`)

TASK 1: CREATE A RECIPE FOR RING NETWORK

a) Make a soma cell with Arbor`s `make_soma_cell()`

b) Add a synapse to the cell with the cells`s function `add_synapse()` which takes a location, here at segment 0 position 0.5, using Arbor`s `segment_location(segment,position)`

c) Add a detector to the cell with the cells`s function `add_detector(location,threshold)` with a threshold of 20 mV

d) Add a stimulus to the cell with `gid 0` at t_0=0 ms for a period of 20 ms with weight 0.01 nA using the cells`s function `add_stimulus(location,start,duration,weight)`

e) Define the source as the previous cell with `gid-1`, baring in mind that the cell with `gid 0` has the last cell in the ring network with `num_cells` as source

HARDWARE RESOURCES

Arbor supports running on systems from laptops and workstations to large distributed HPC clusters

Therefore, Arbor uses distributed **context**s to

- Describe the computer system that a simulation is to be distributed over and run on.
- Perform collective operations over the distributed system.
- Query information about the distributed system.

The global context is determined at run time taking the **computational resources** as input. Further, the user can choose between using a non-distributed (local) context, or a distributed MPI context (if available).

An execution context is created by a user before building and running a simulation. This context is then used to perform domain decomposition and initialize the simulation.

DOMAIN DECOMPOSITION AND LOAD BALANCING

Arbor is performance portable

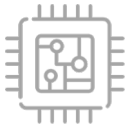
A **domain decomposition** is a description of the distribution of the model over the available computational resources. The description partitions the cells in the model as follows:

- Group the cells into cell groups of the same kind of cell.
- Assign each cell group to either a CPU core or GPU on a specific MPI rank.

A **load balancer** is a distributed algorithm that

- Determines the domain decomposition to balance the work over threads (and GPU accelerators if available).
- Uses the model recipe and a description of the available computational resources as inputs.

TASK 2: CREATE PARALLEL EXECUTION CONTEXT



a) Get all available local hardware resources by using Arbor`s `proc_allocation()`.

b) Create a context by using Arbor`s `context()` that uses the local resources, and an MPI communicator for distributed communication.

c) Initiate the `recipe()` defined in task 1 with 100 cells.

d) Partition the simulation over the distributed system by using Arbor`s `partition_load_balance()` that uses the recipe and the context handle.

FROM RECIPE TO SIMULATION

A simulation needs a recipe, a context and a domain decomposition

To build a **simulation** the following are needed:

- A recipe that describes the cells and connections in the model.
- A context used to execute the simulation.
- Thereof, a domain decomposition describing the distribution of the model over the local and distributed hardware resources.

The workflow to build a simulation is, thus, to first generate a domain decomposition, then initiate the simulation that is the executable form of a model. A simulation is initiated from a recipe, the decomposition and the context. Then the simulation is used to update and monitor the model state.

TASK 3: BUILD AND RUN THE SIMULATION

a) Initiate the simulation by using Arbor`s `simulation()` which takes the recipe, the domain decomposition and the context.

b) Run the simulation using the simulation`s command `run()` for a simulated time of 2000 ms with a time stepping size of 0.025 ms.

OPTIONAL: METER MEASUREMENT AND SPIKE RECORD

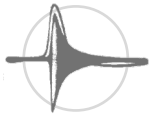
To measure memory and (if available) energy consumption

For measuring memory (and energy) consumption Arbor`s **meter manager** can be used.
First the meter manager needs to be initiated, then the metering started and checkpoints set, wherever the manager should report the meters. The measurement starts from the start to the first checkpoint and then in between checkpoints.

Checkpoints are defined by a string describing the process to be measured. The **meter report** finally collects all meters.

A **spike recorder** monitors all spikes recorded by a detector on the cells during the simulation.

OPTIONAL TASK 4: MEASURE



a) Initiate Arbor`s `meter_manager()` and start measuring by using the meters` `start()` function that takes the context as input.

b) Set checkpoints using the meters` `checkpoint()` function which takes a string and the context as input

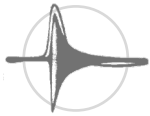
   i. „recipe create“ after initiating the recipe

   ii. „load balance“ after partitioning the simulation

   iii. „simulation init“ after initiating the simulation

   iv. „simulation run“ after running the simulation

OPTIONAL TASK 4: GET RESULTS



c) Between initiating and running the simulation, build the spike recorder using Arbor‘s `make_spike_recorder()` which takes the simulation handle as input

d) Make and print a meter report by using Arbor`s `make_meter_report()` function that takes the meters and the context as input.

e) Get the recorder`s spikes

f) Play around with parameters and see what happens.

SUMMARY

Using Arbor`s python frontend

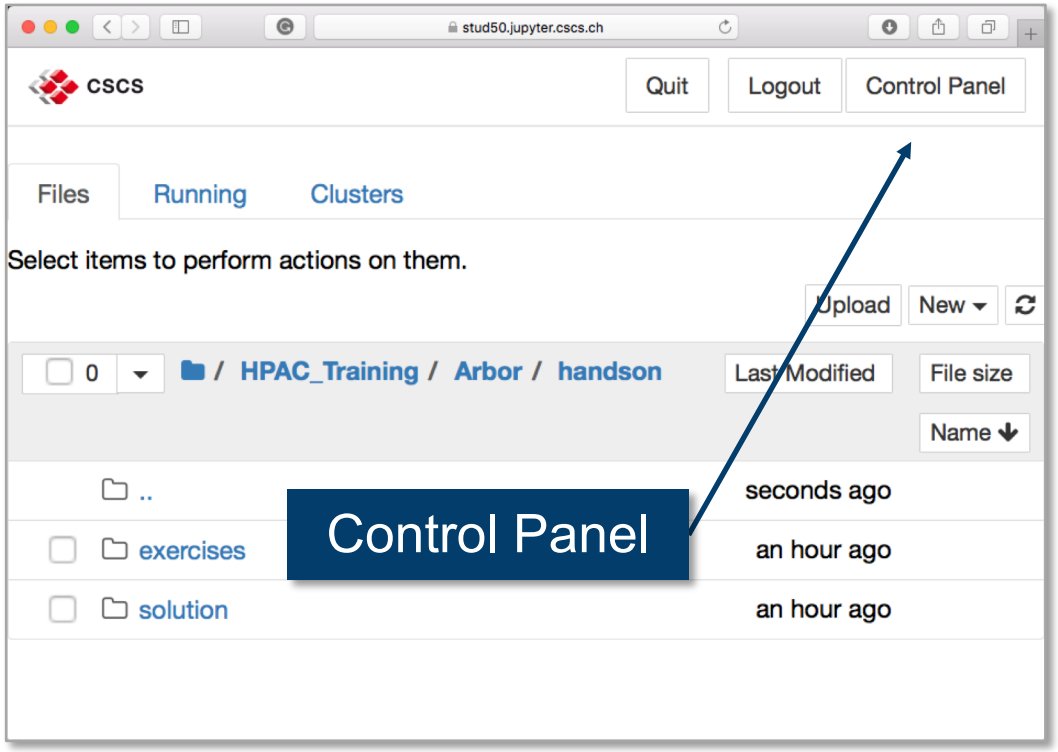
We learned how …

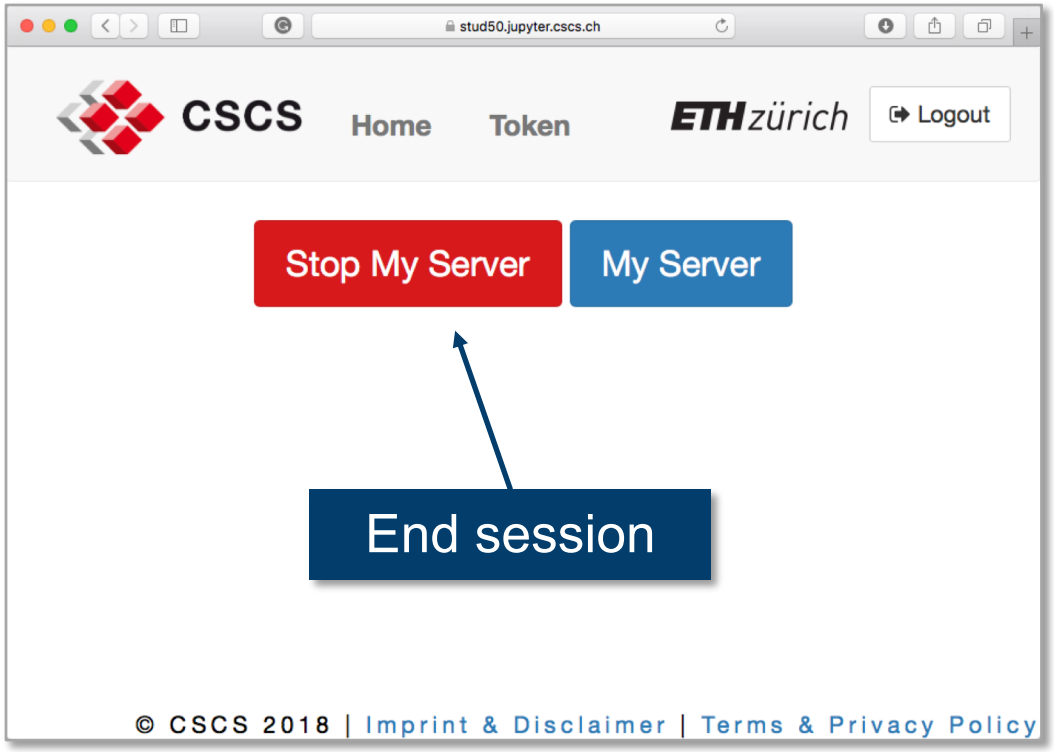
- to describe a neuron model **using a recipe;**
- to get **resources**, create a **parallel execution context** and partition the **load balance;**
- **initiate the simulation** over the distributed system and run the simulation;
- set up **measurement meters**, get spikes **recorded** and print a meter report along with the spiking times of the cells.

Further, we learned how to use and create Jupyter notebooks on CSCS` JupyterHub.

CLOSE JUPYTER NOTEBOOKS AND END SESSION

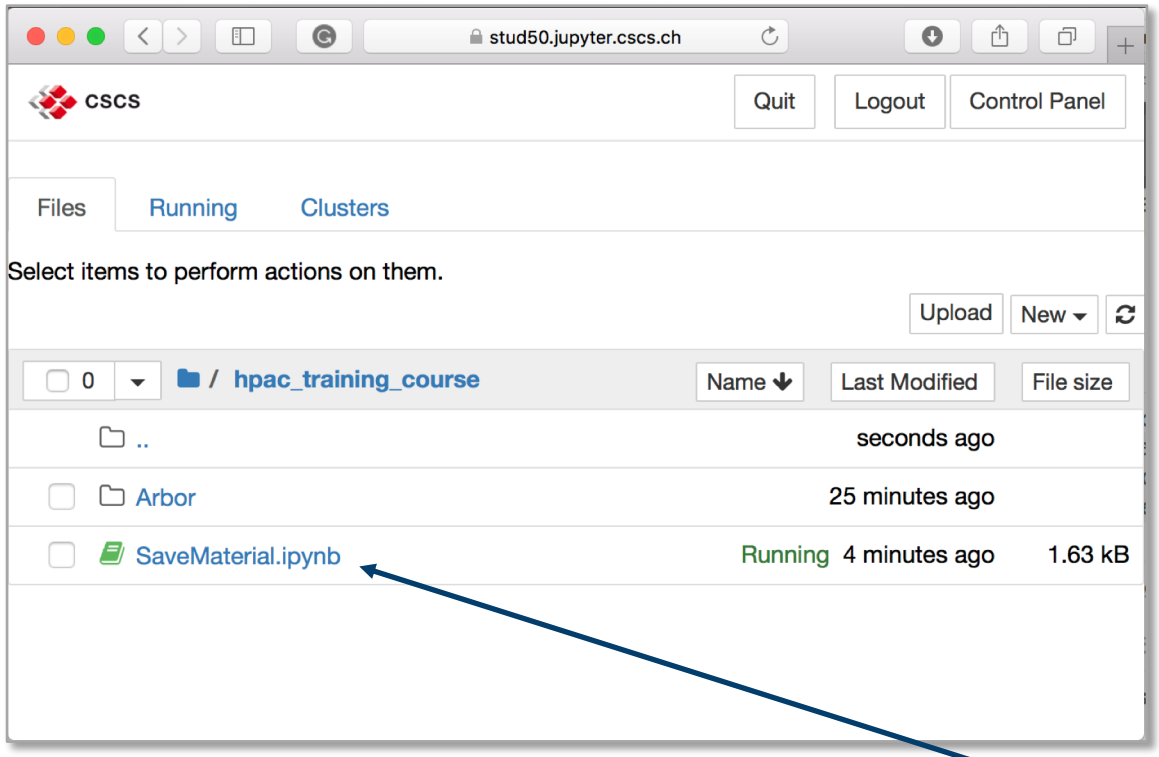
End your session via “Control Panel – Stop My Server” after completing the tasks

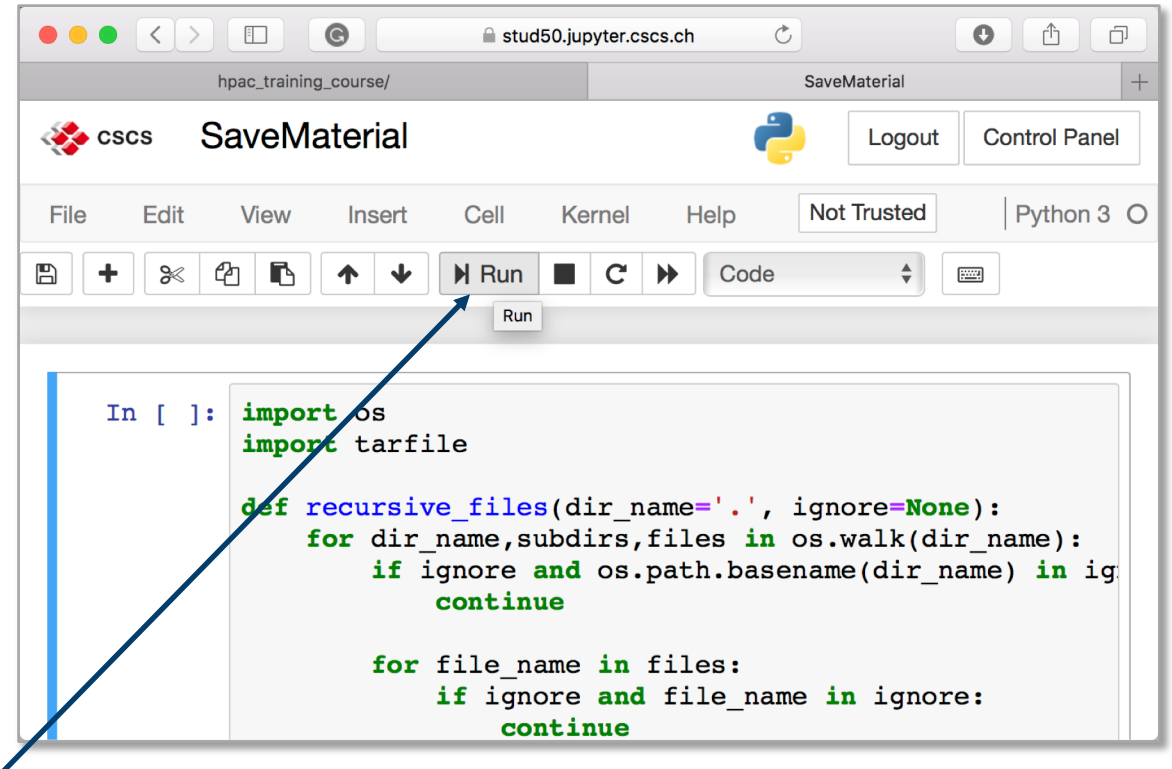




DOWNLOAD THE MATERIAL

First, run the notebook SaveMaterial.ipynb

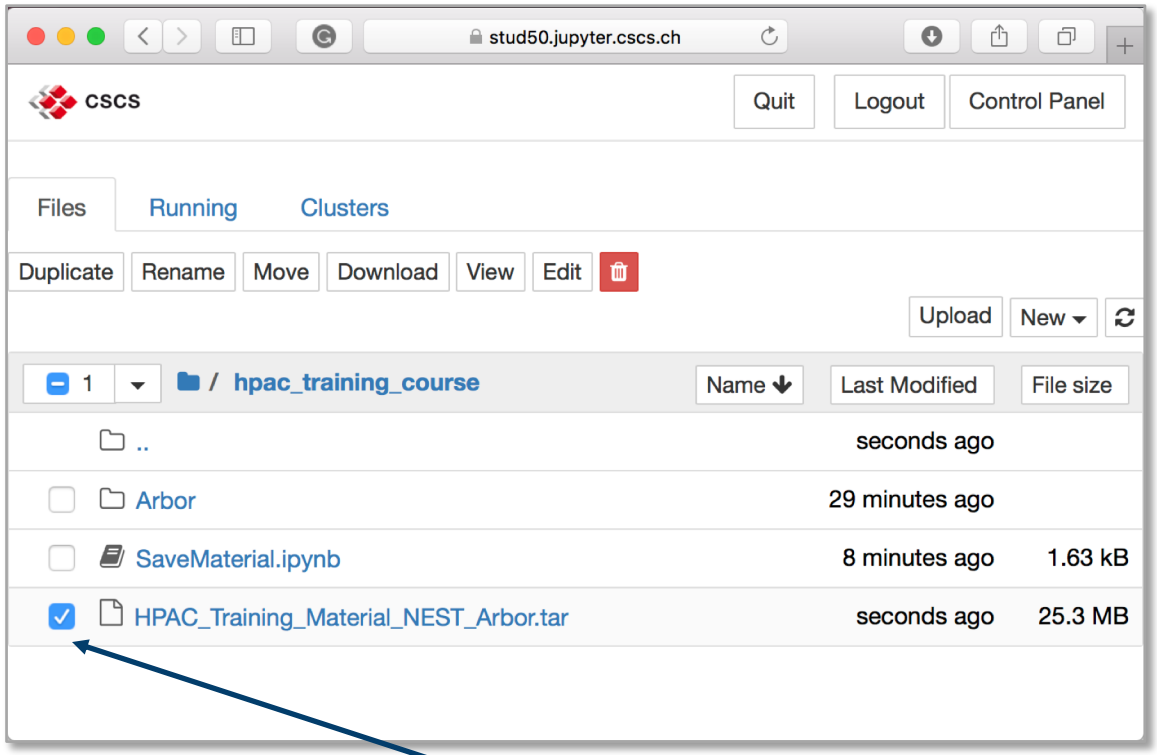


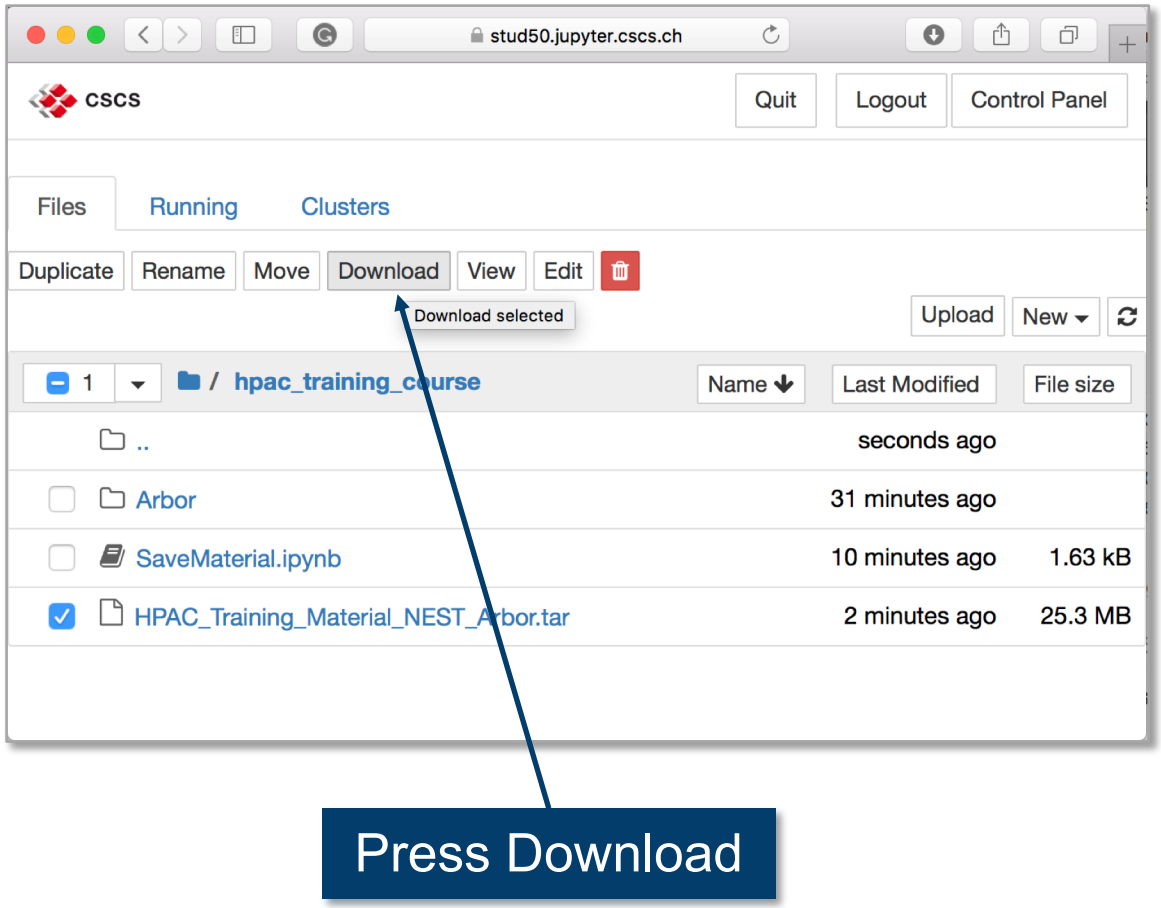


Run notebook

DOWNLOAD THE MATERIAL

Then, download zipped file





LITERATURE AND LINKS

- *Arbor – a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures*, High Performance Computing for Neuroscience, PDP2019
- https://github.com/arbor-sim/arbor
- https://arbor.readthedocs.io

BACKUP – INSTALLATION GUIDE

BUILD AND INSTALL ARBOR FROM SOURCE CODE

Requirements

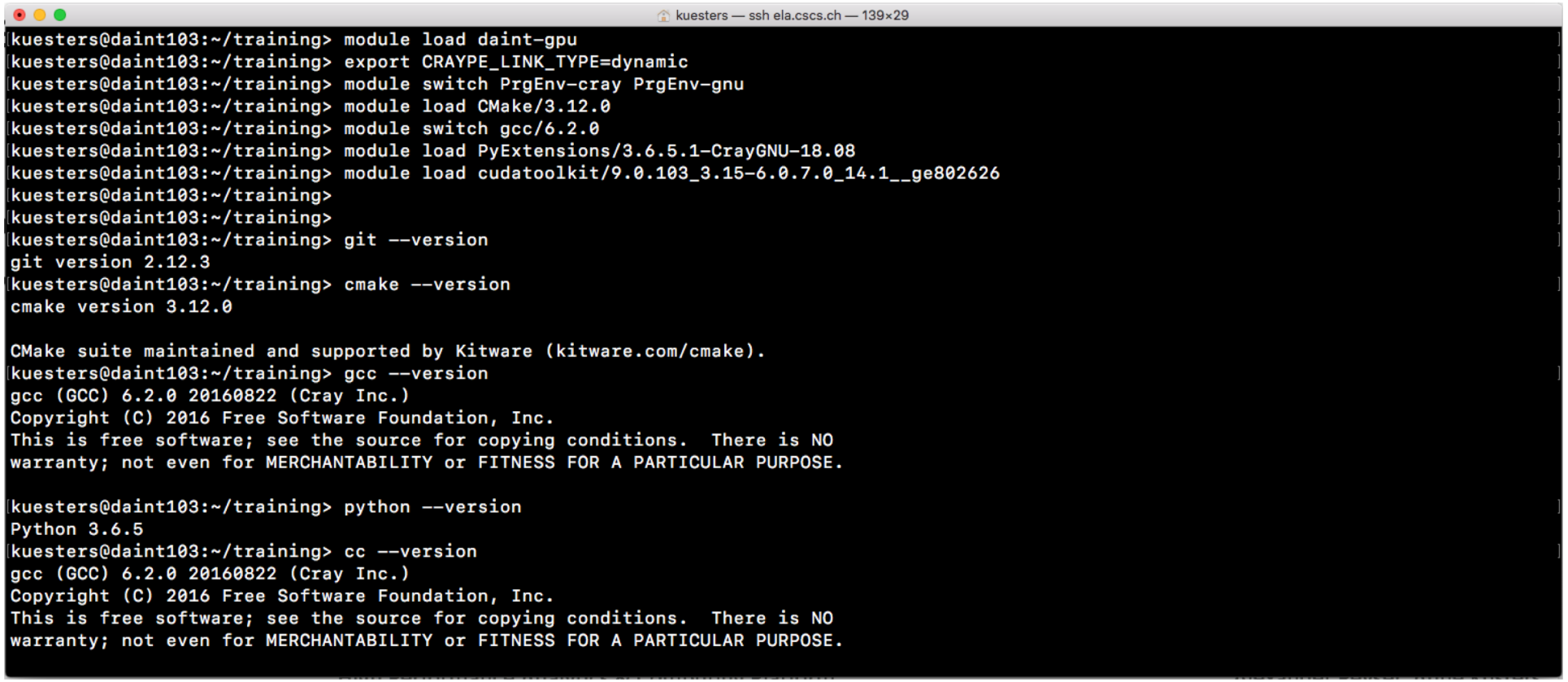
Follow steps to use Arbor`s python frontend:

1. Minimal requirements:
   - git
   - cmake
   - gcc 6/ clang 4/ Apple clang 9
   - python
   - mpi

BUILD AND INSTALL ARBOR FROM SOURCE CODE

Load modules

1. Load modules on Piz Daint



BUILD AND INSTALL ARBOR FROM SOURCE CODE

Clone code

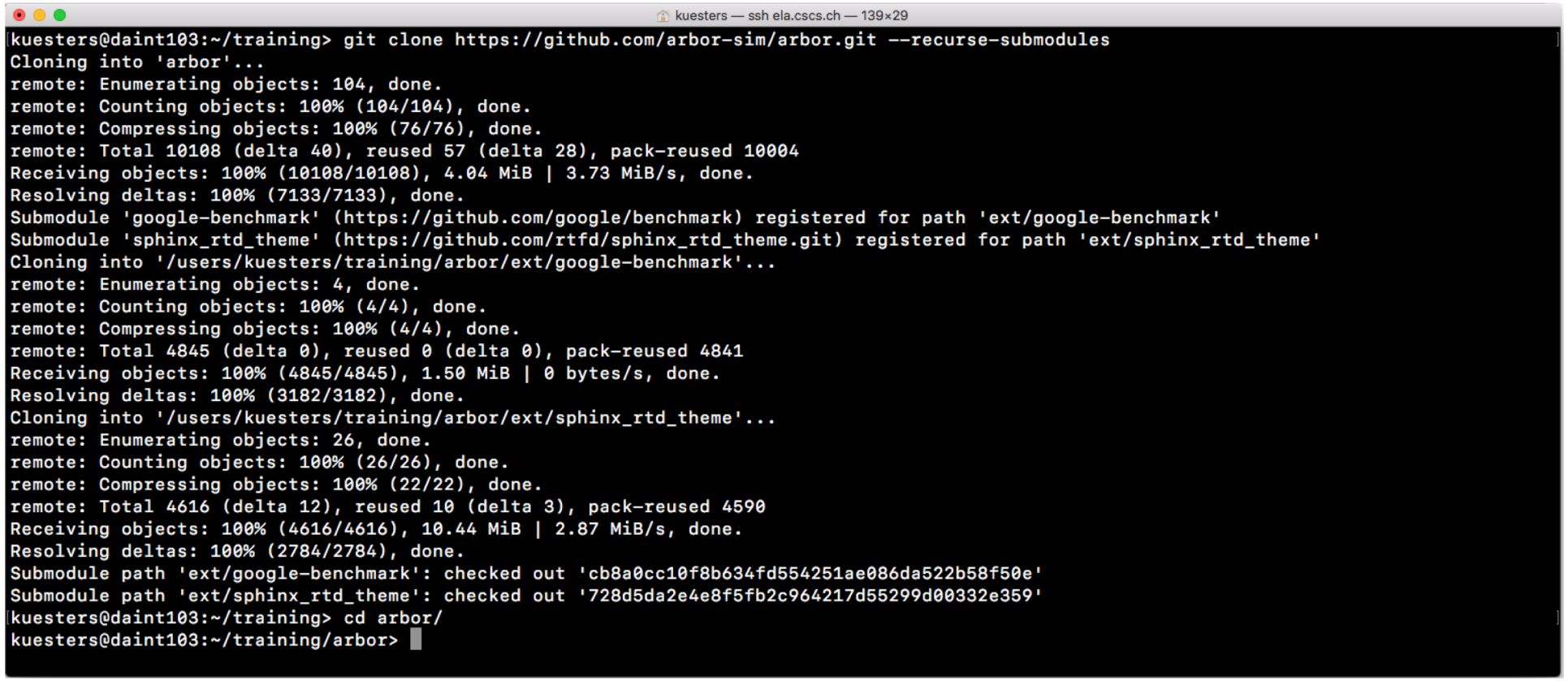
Steps to use Arbor`s python frontend:

1. Minimal requirements
2. Clone Arbor`s source code
   - `git clone https://github.com/arbor-sim/arbor.git --recurse-submodules`
   - `cd arbor`

BUILD AND INSTALL ARBOR FROM SOURCE CODE

Clone code

2. Clone Arbor`s source code



BUILD AND INSTALL ARBOR FROM SOURCE CODE

Checkout python branch

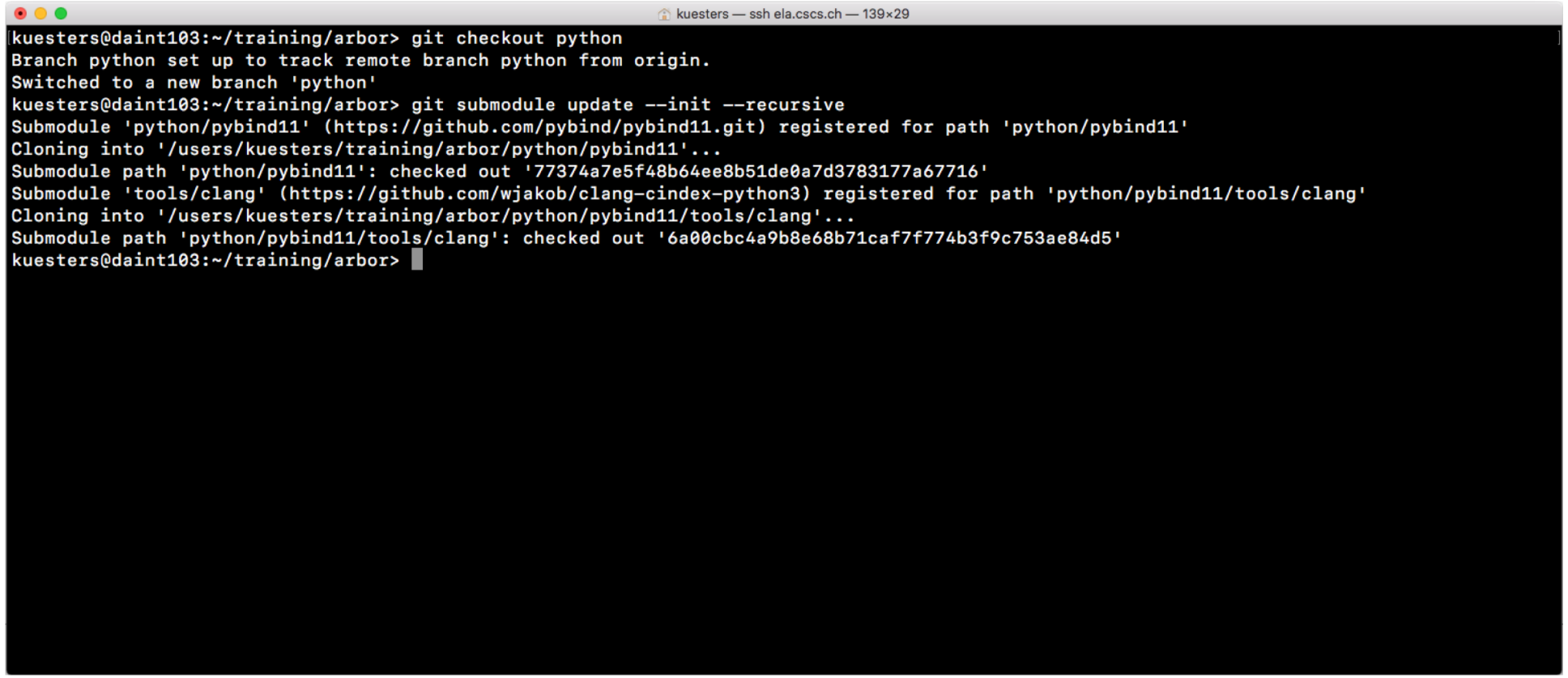
Steps to use Arbor`s python frontend:

1. Minimal requirements
2. Clone Arbor`s source code
3. Checkout python branch
   - `git checkout python`
   - `git submodule update --init --recursive`

BUILD AND INSTALL ARBOR FROM SOURCE CODE

Checkout python branch

3. Switch to python branch and update submodules



BUILD AND INSTALL ARBOR FROM SOURCE CODE

Prepare environment

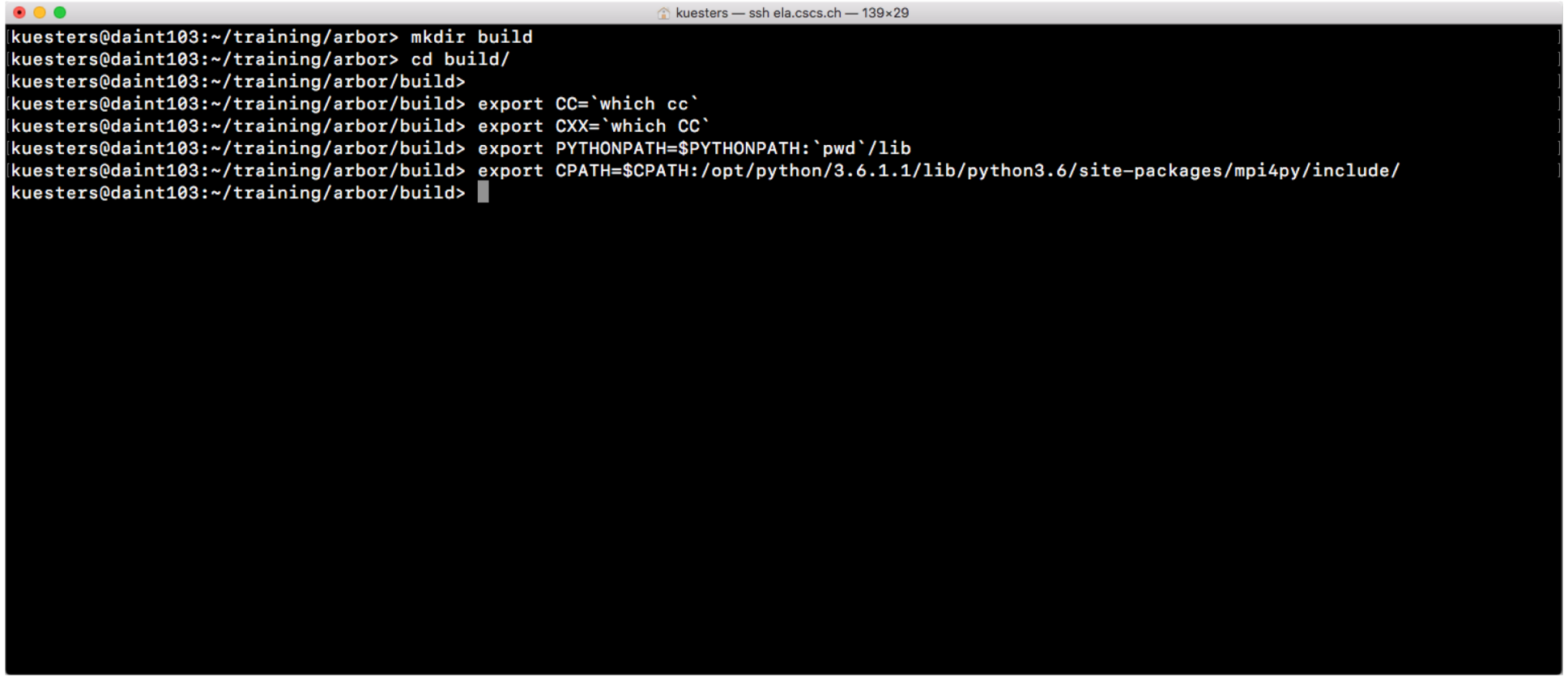
Steps to use Arbor`s python frontend:

1. Minimal requirements
2. Clone Arbor`s source code
3. Checkout python branch
4. Prepare environment
   - `mkdir build; cd build`
   - set `CC` and `CXX`
   - set `PYTHONPATH`
   - set `CPATH`

BUILD AND INSTALL ARBOR FROM SOURCE CODE

Prepare environment

4. Prepare environment



BUILD AND INSTALL ARBOR FROM SOURCE CODE

Build

Steps to use Arbor`s python frontend:

1. Minimal requirements
2. Clone Arbor`s source code
3. Checkout python branch
4. Prepare environment
5. Build

- ```
  cmake .. \
  -DARB_WITH_MPI=ON \
  -DARB_WITH_PYTHON=ON \
  -DARB_WITH_GPU=ON \
  -DCMAKE_INSTALL_PREFIX=$HOME/training/installation/arbor
  ```
- `make -j8`
  ```

BUILD AND INSTALL ARBOR FROM SOURCE CODE

Build

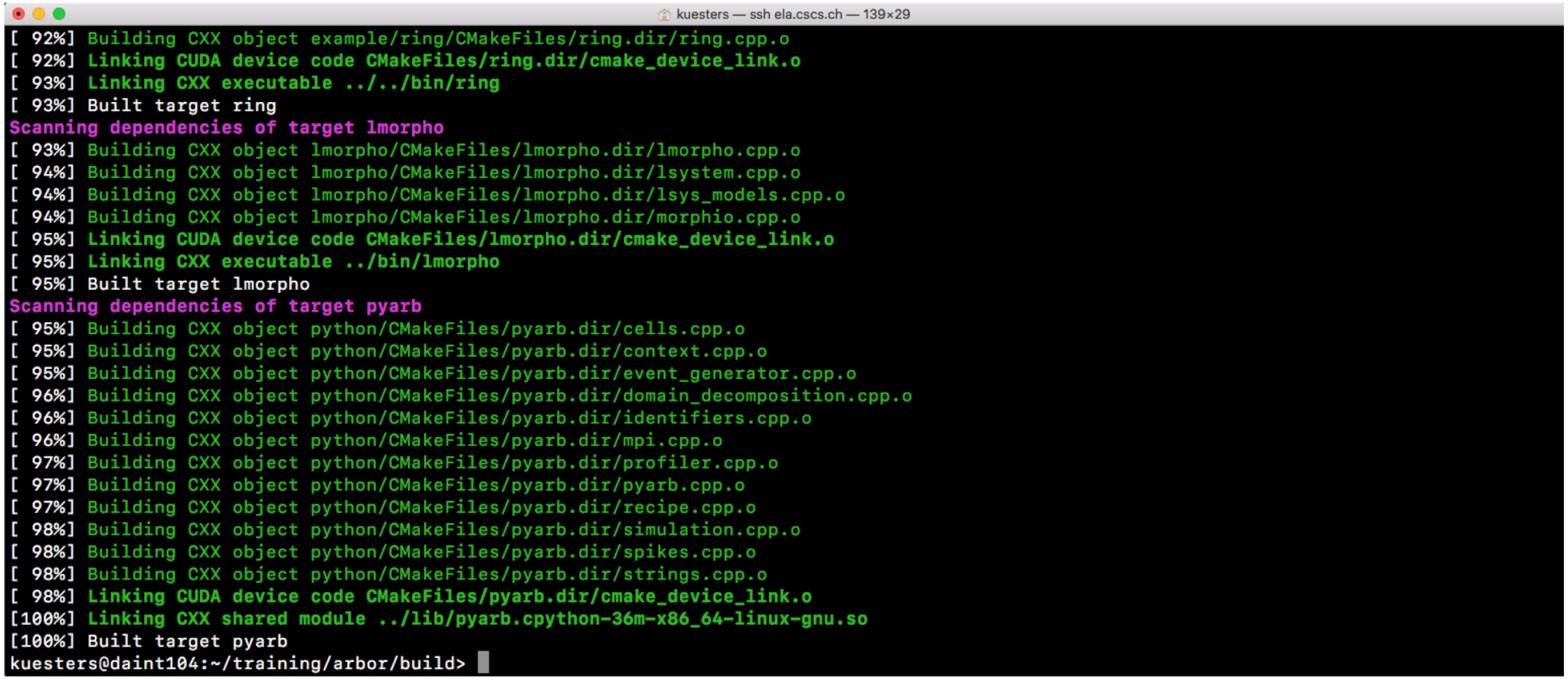
5. Build Arbor's python wrapper

```
kuesters — ssh ela.cscs.ch — 139×29
kuesters@daint104:~/training/arbor/build> cmake .. -DARB_WITH_MPI=ON -DARB_WITH_PYTHON=ON -DARB_WITH_GPU=ON -DCMAKE_INSTALL_PREFIX=$HOME/tr
aining/installation/arbor; make
-- The C compiler identification is GNU 6.2.0
-- The CXX compiler identification is GNU 6.2.0
-- Cray Programming Environment 2.5.15 C
-- Check for working C compiler: /opt/cray/pe/craype/2.5.15/bin/cc
-- Check for working C compiler: /opt/cray/pe/craype/2.5.15/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Cray Programming Environment 2.5.15 CXX
-- Check for working CXX compiler: /opt/cray/pe/craype/2.5.15/bin/CC
-- Check for working CXX compiler: /opt/cray/pe/craype/2.5.15/bin/CC -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- The CUDA compiler identification is NVIDIA 9.0.102
-- Check for working CUDA compiler: /opt/nvidia/cudatoolkit9.0/9.0.103_3.15-6.0.7.0_14.1__ge802626/bin/nvcc
-- Check for working CUDA compiler: /opt/nvidia/cudatoolkit9.0/9.0.103_3.15-6.0.7.0_14.1__ge802626/bin/nvcc -- works
-- Detecting CUDA compiler ABI info
-- Detecting CUDA compiler ABI info - done
-- Could NOT find PY_mpi4py (missing: PY_MPI4PY)
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - found
-- Found Threads: TRUE
```

BUILD AND INSTALL ARBOR FROM SOURCE CODE

Build

5. Build Arbor's python wrapper



BUILD AND INSTALL ARBOR FROM SOURCE CODE

Install

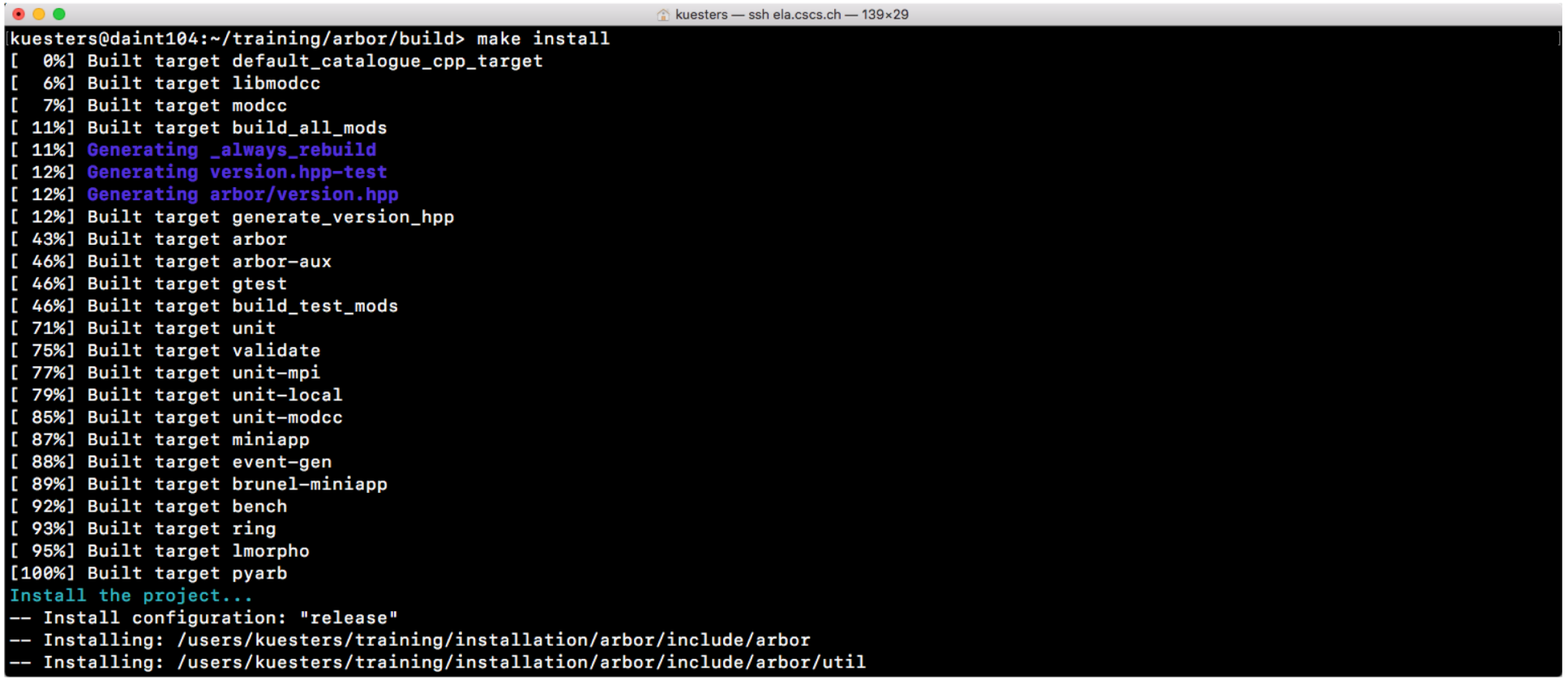
Steps to use Arbor`s python frontend:

1. Minimal requirements
2. Clone Arbor`s source code
3. Checkout python branch
4. Prepare environment
5. Build
6. Install
   - `make install`

BUILD AND INSTALL ARBOR FROM SOURCE CODE

Install

6. Install



BUILD AND INSTALL ARBOR FROM SOURCE CODE

Test

Steps to use Arbor`s python frontend:

1. Minimal requirements
2. Clone Arbor`s source code
3. Checkout python branch
4. Prepare environment
5. Build
6. Install
7. Test

- `salloc -Cgpu -N 4 -t 60`
- `srun -n 4 ./bin/unit`

BUILD AND INSTALL ARBOR FROM SOURCE CODE

Test

7. Test – get allocation (e.g. GPU, 4 nodes) and start unit test