



HANDS-ON

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

12TH DECEMBER 2018 | ANNE KÜSTERS & ALEXANDER PEYSER



Human Brain Project

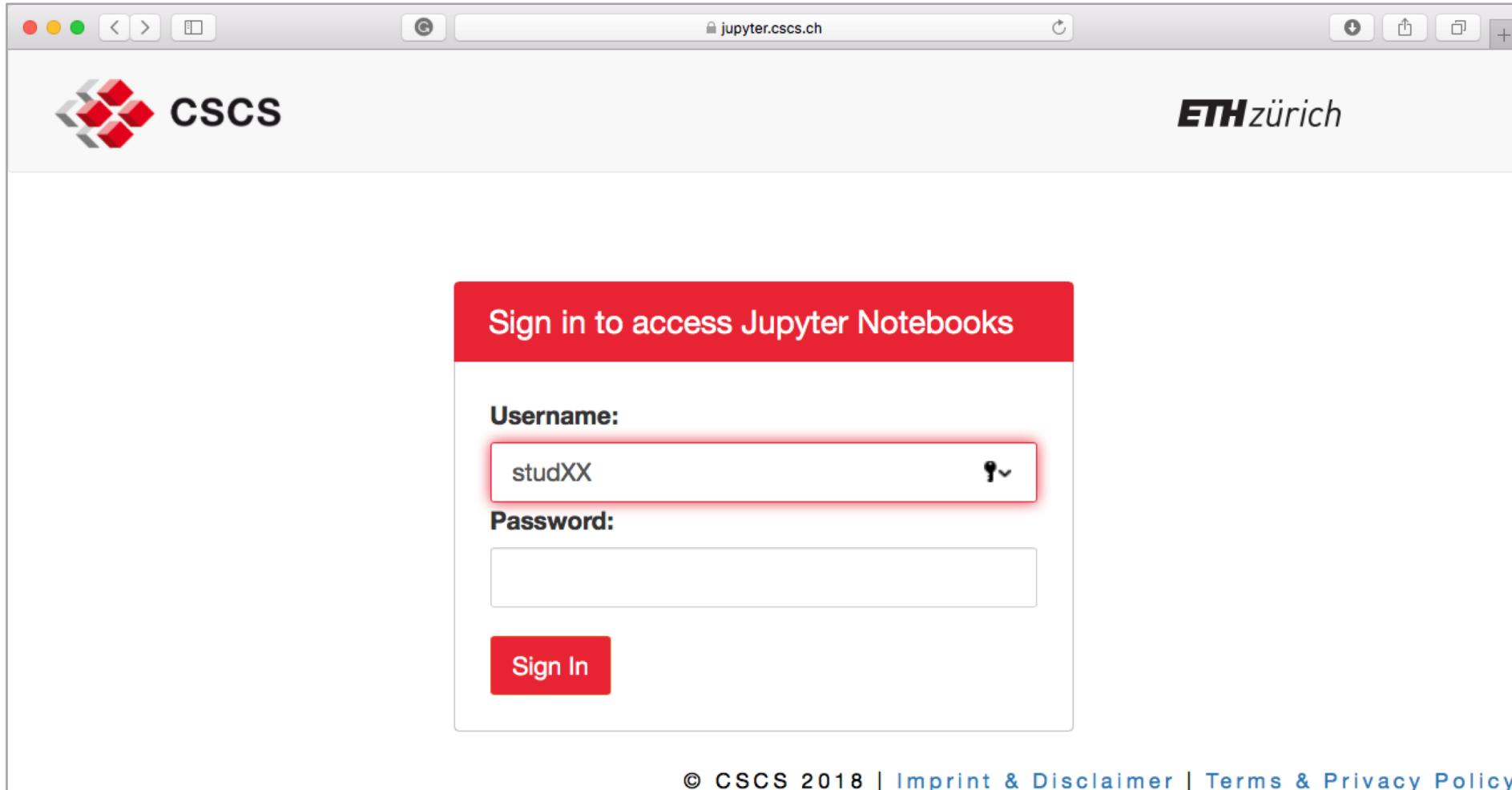


Co-funded by
the European Union



PREPARATION

Sign in to <https://jupyter.csccs.ch> with your training username and password



PREPARATION

Spawn a GPU with reservation of one node for one hour

The screenshot shows a web browser window with the URL `stud50.jupyter.cscs.ch`. The page is titled "Spawner Options". It features several dropdown menus and input fields:

- Piz Daint node type: gpu
- Reservation: Jupyterhub single node only
- Number of Nodes: 1
- Job duration: 1 hour
- Account (leave empty for default): (empty input field)

A large red "Spawn" button is centered at the bottom of the form.

At the bottom of the page, there is a footer with the text: © CSCS 2018 | [Imprint & Disclaimer](#) | [Terms & Privacy Policy](#).

ORGANIZATION AND WORK-FLOW

4 tasks, each dependent on the previous

Structure:

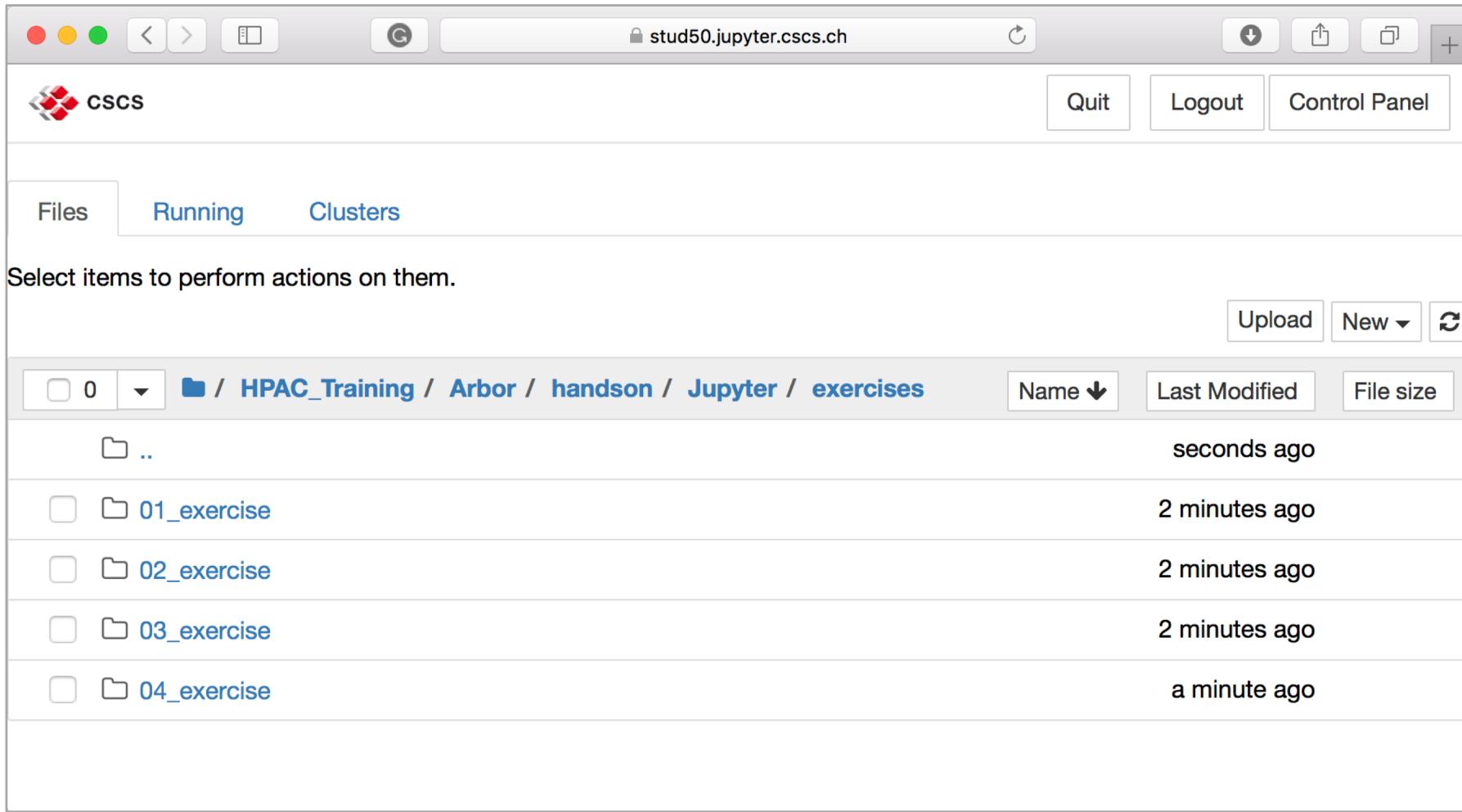
- On your training account, go to:
 - [HPAC_training](#) → Arbor
 - [handson](#) → Jupyter → exercises
 - [0X_exercise](#) (where X is the exercise number 1-4)
- Here, you find a python script with your `#TODO#`
- Each exercise builds up on the previous exercise
 - Work on your TODO's and save your solution 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

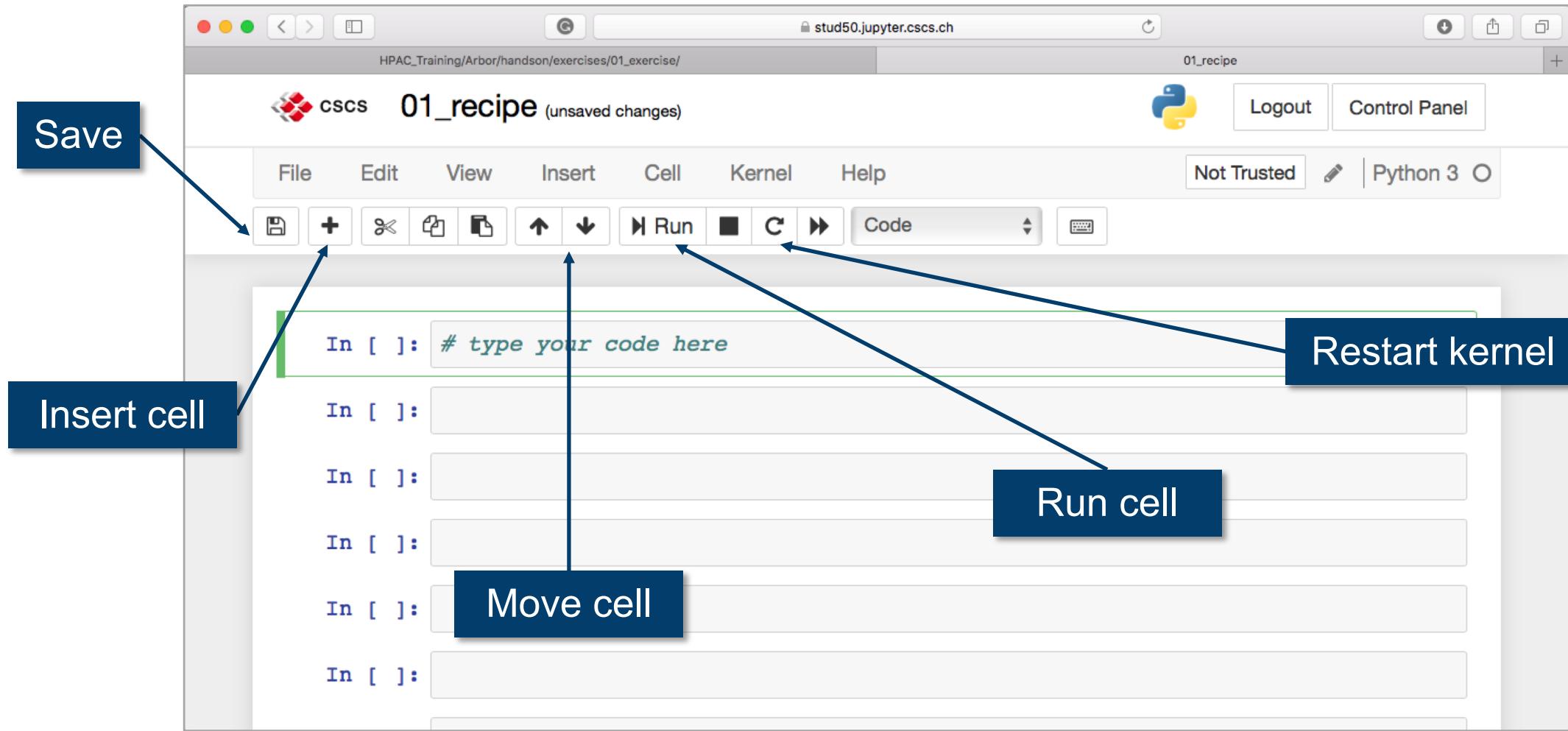
ORGANIZATION AND WORK-FLOW

4 tasks, each dependent on the previous



SHORT INTRODUCTION OF JUPYTER NOTEBOOKS

Easy to use

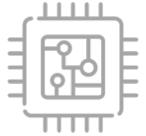


OVERVIEW OF EXERCISES

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



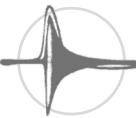
1. Describe neuron model by defining a **recipe**



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 **contexts** 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.

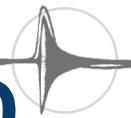
The workflow to build a simulation is, thus, to first generate a domain decomposition, then initiate the simulation that is the executable form of a model. A simulation is initiated from a recipe, the decomposition and the context. Then the simulation is used to update and monitor the model state.



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

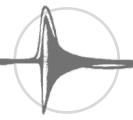
For measuring memory (and energy) consumption Arbor's **meter manager** can be used. First the meter manager needs to be initiated, then the metering started and checkpoints set, wherever the manager should report the meters. The measurement starts from the start to the first checkpoint and then in between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

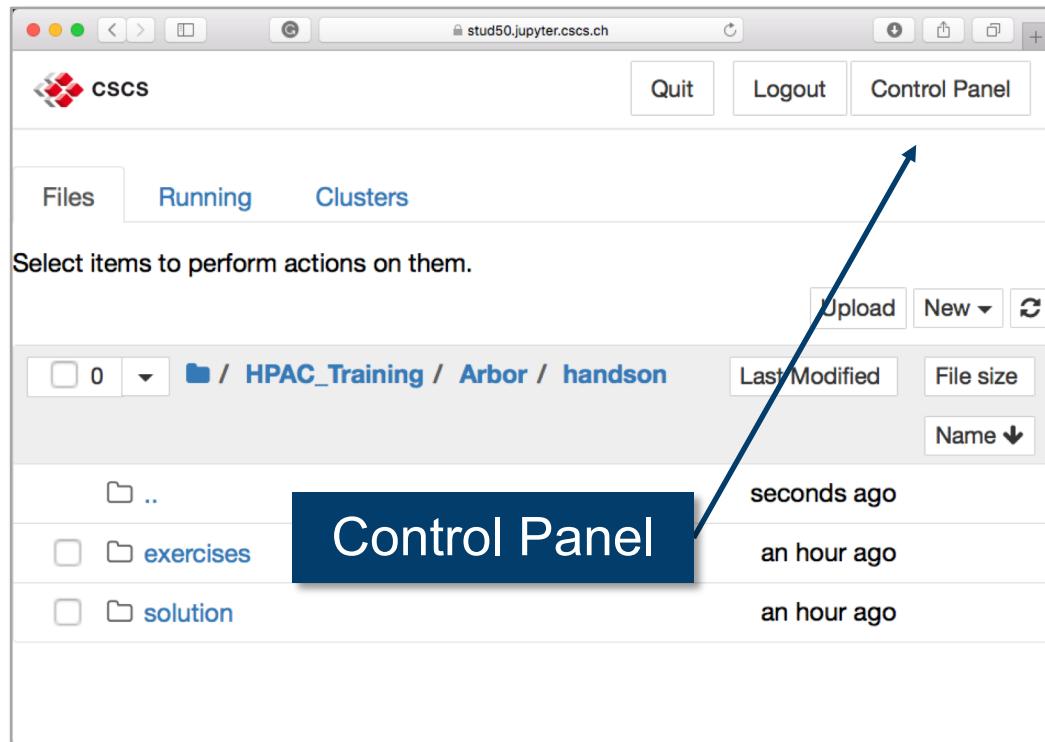
We learned how ...

- 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.

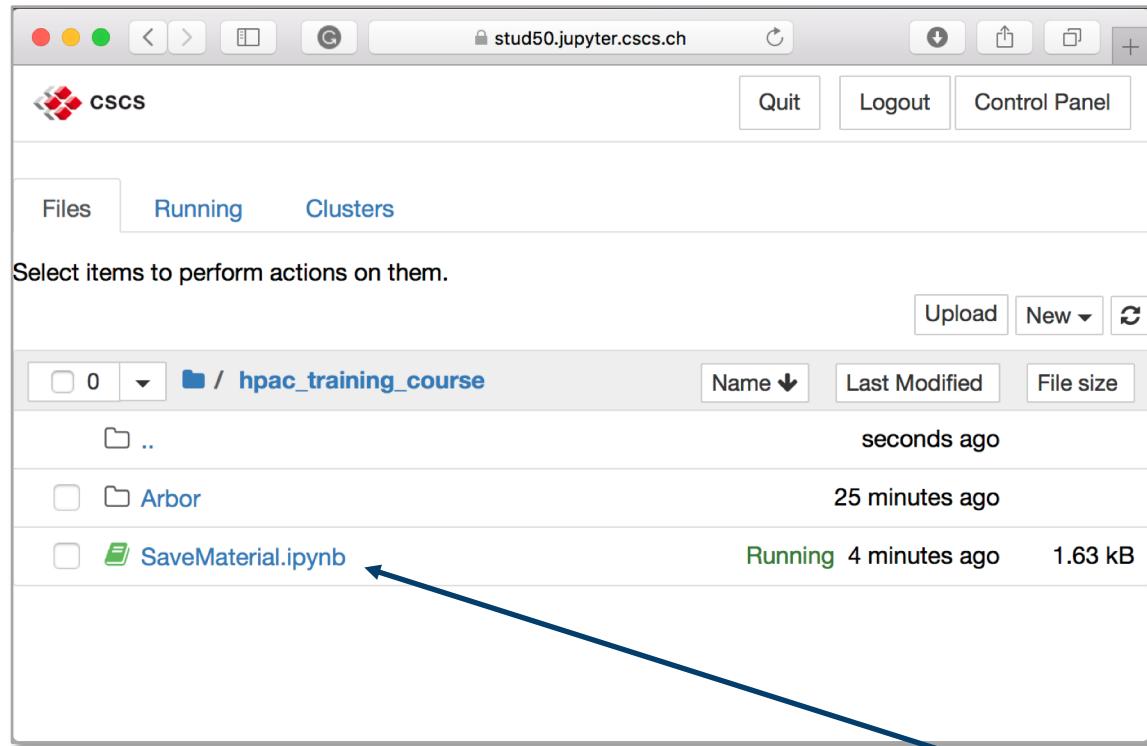
CLOSE JUPYTER NOTEBOOKS AND END SESSION

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



DOWNLOAD THE MATERIAL

First, run the notebook **SaveMaterial.ipynb**



The screenshot shows a Jupyter Notebook titled 'SaveMaterial'. The code cell contains the following Python script:

```
In [ ]: import os
import tarfile

def recursive_files(dir_name='.', ignore=None):
    for dir_name, subdirs, files in os.walk(dir_name):
        if ignore and os.path.basename(dir_name) in ignore:
            continue

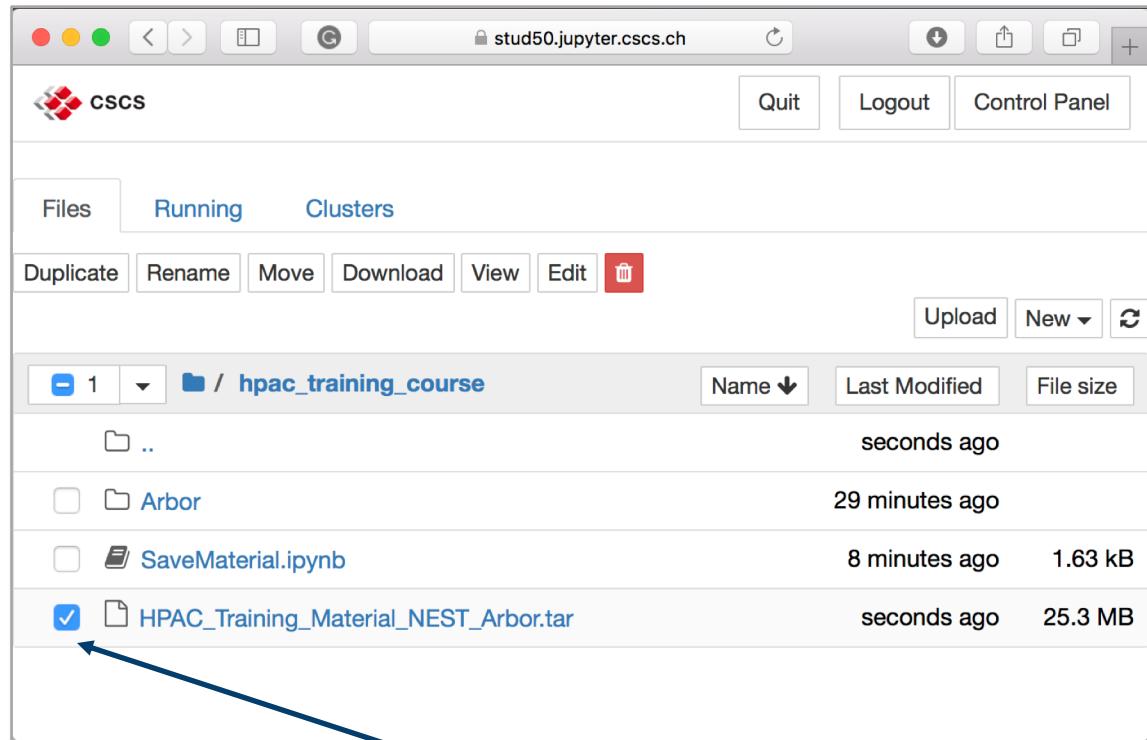
        for file_name in files:
            if ignore and file_name in ignore:
                continue
```

A blue arrow points from the 'Run' button in the toolbar to the 'Run' button in the code cell's status bar.

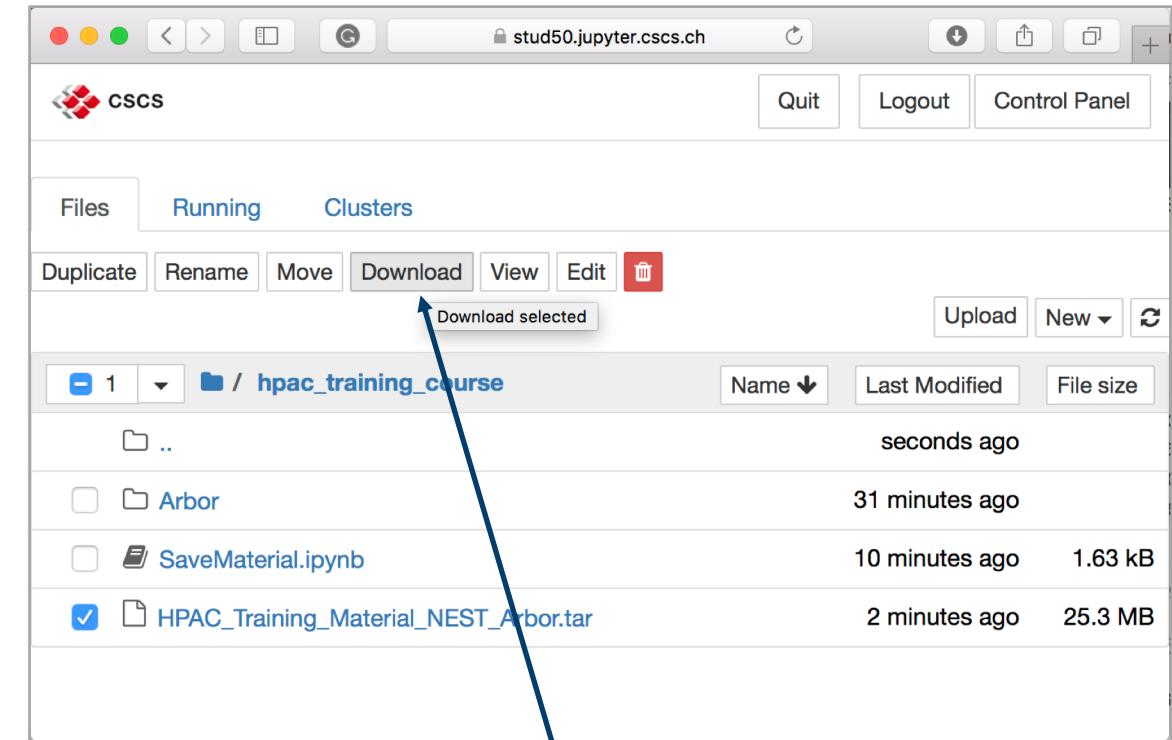
Run notebook

DOWNLOAD THE MATERIAL

Then, download zipped file



Check box



Press Download

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

```
kuesters@daint103:~/training> module load daint-gpu
kuesters@daint103:~/training> export CRAYPE_LINK_TYPE=dynamic
kuesters@daint103:~/training> module switch PrgEnv-cray PrgEnv-gnu
kuesters@daint103:~/training> module load CMake/3.12.0
kuesters@daint103:~/training> module switch gcc/6.2.0
kuesters@daint103:~/training> module load PyExtensions/3.6.5.1-CrayGNU-18.08
kuesters@daint103:~/training> module load cudatoolkit/9.0.103_3.15-6.0.7.0_14.1__ge802626
kuesters@daint103:~/training>
kuesters@daint103:~/training>
kuesters@daint103:~/training> git --version
git version 2.12.3
kuesters@daint103:~/training> cmake --version
cmake version 3.12.0

CMake suite maintained and supported by Kitware (kitware.com/cmake).
kuesters@daint103:~/training> gcc --version
gcc (GCC) 6.2.0 20160822 (Cray Inc.)
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

kuesters@daint103:~/training> python --version
Python 3.6.5
kuesters@daint103:~/training> cc --version
gcc (GCC) 6.2.0 20160822 (Cray Inc.)
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

BUILD AND INSTALL ARBOR FROM SOURCE CODE

Clone code

Steps to use Arbor's python frontend:

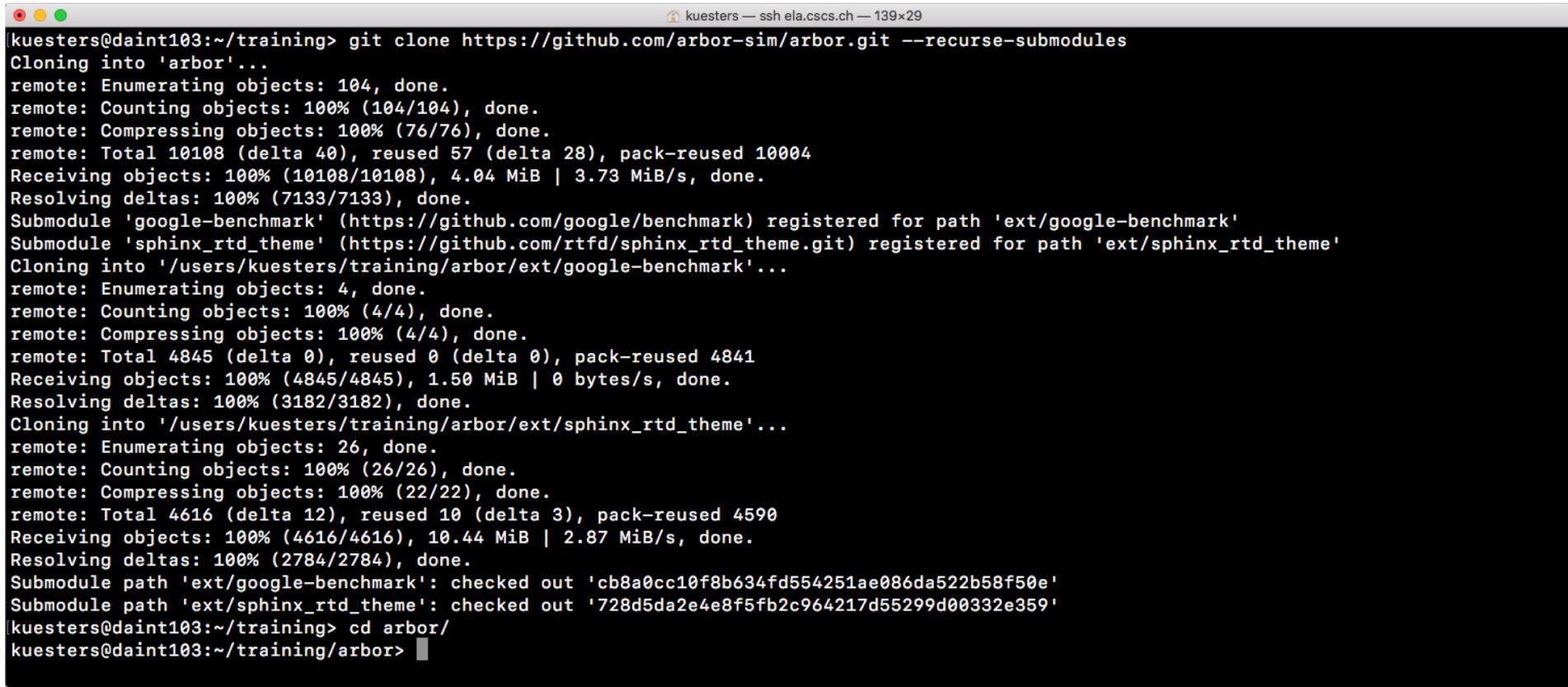
1. Minimal requirements
2. Clone Arbor's source code

- `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



```
kuesters@daint103:~/training> git clone https://github.com/arbor-sim/arbor.git --recurse-submodules
Cloning into 'arbor'...
remote: Enumerating objects: 104, done.
remote: Counting objects: 100% (104/104), done.
remote: Compressing objects: 100% (76/76), done.
remote: Total 10108 (delta 40), reused 57 (delta 28), pack-reused 10004
Receiving objects: 100% (10108/10108), 4.04 MiB | 3.73 MiB/s, done.
Resolving deltas: 100% (7133/7133), done.
Submodule 'google-benchmark' (https://github.com/google/benchmark) registered for path 'ext/google-benchmark'
Submodule 'sphinx_rtd_theme' (https://github.com/rtd/sphinx_rtd_theme.git) registered for path 'ext/sphinx_rtd_theme'
Cloning into '/users/kuesters/training/arbor/ext/google-benchmark'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4845 (delta 0), reused 0 (delta 0), pack-reused 4841
Receiving objects: 100% (4845/4845), 1.50 MiB | 0 bytes/s, done.
Resolving deltas: 100% (3182/3182), done.
Cloning into '/users/kuesters/training/arbor/ext/sphinx_rtd_theme'...
remote: Enumerating objects: 26, done.
remote: Counting objects: 100% (26/26), done.
remote: Compressing objects: 100% (22/22), done.
remote: Total 4616 (delta 12), reused 10 (delta 3), pack-reused 4590
Receiving objects: 100% (4616/4616), 10.44 MiB | 2.87 MiB/s, done.
Resolving deltas: 100% (2784/2784), done.
Submodule path 'ext/google-benchmark': checked out 'cb8a0cc10f8b634fd554251ae086da522b58f50e'
Submodule path 'ext/sphinx_rtd_theme': checked out '728d5da2e4e8f5fb2c964217d55299d00332e359'
kuesters@daint103:~/training> cd arbor/
kuesters@daint103:~/training/arbor>
```

BUILD AND INSTALL ARBOR FROM SOURCE CODE

Checkout python branch

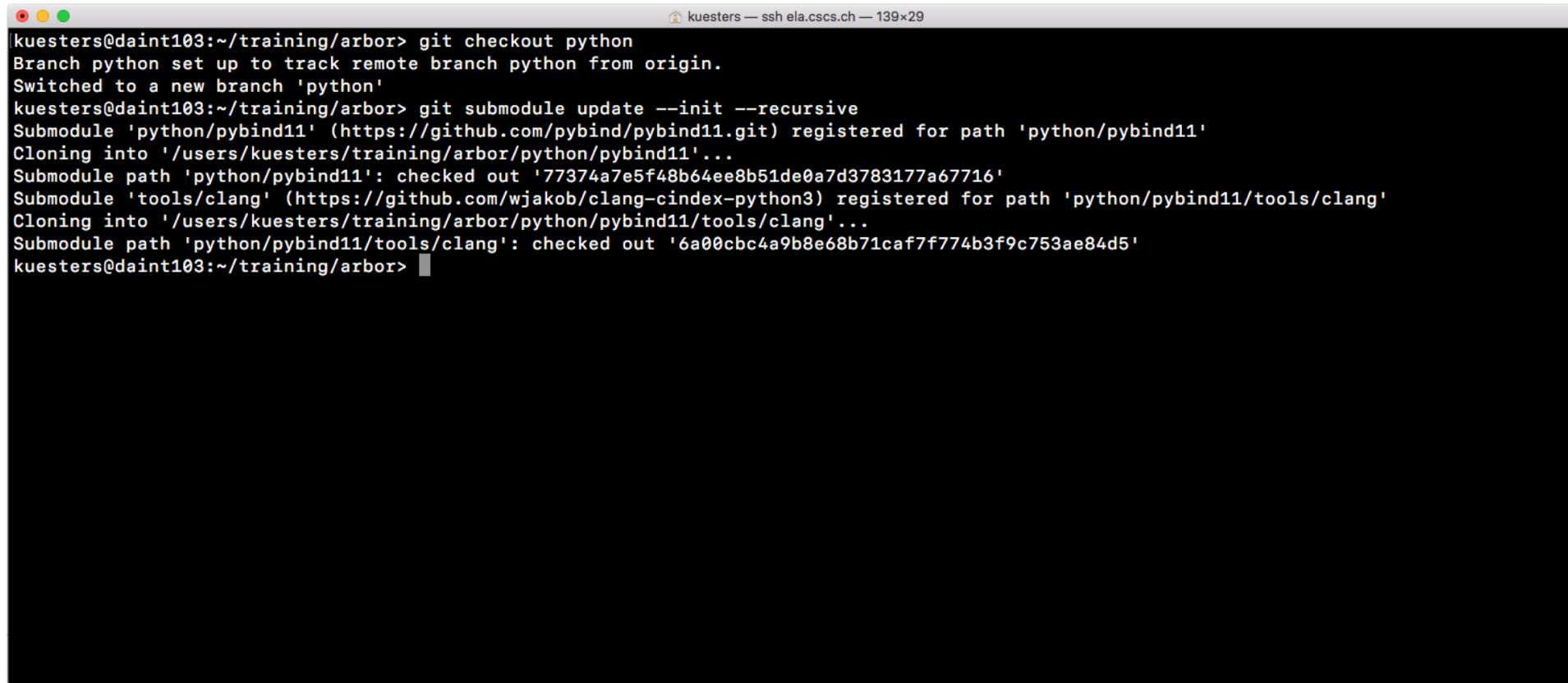
Steps to use Arbor's python frontend:

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



```
[kuesters@daint103:~/training/arbor> git checkout python
Branch python set up to track remote branch python from origin.
Switched to a new branch 'python'
kuesters@daint103:~/training/arbor> git submodule update --init --recursive
Submodule 'python/pybind11' (https://github.com/pybind/pybind11.git) registered for path 'python/pybind11'
Cloning into '/users/kuesters/training/arbor/python/pybind11'...
Submodule path 'python/pybind11': checked out '77374a7e5f48b64ee8b51de0a7d3783177a67716'
Submodule 'tools/clang' (https://github.com/wjakob/clang-cindex-python3) registered for path 'python/pybind11/tools/clang'
Cloning into '/users/kuesters/training/arbor/python/pybind11/tools/clang'...
Submodule path 'python/pybind11/tools/clang': checked out '6a00cbc4a9b8e68b71caf7f774b3f9c753ae84d5'
kuesters@daint103:~/training/arbor>
```

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

```
kuesters@daint103:~/training/arbor> mkdir build
kuesters@daint103:~/training/arbor> cd build/
kuesters@daint103:~/training/arbor/build>
kuesters@daint103:~/training/arbor/build> export CC=`which cc`
kuesters@daint103:~/training/arbor/build> export CXX=`which CC`
kuesters@daint103:~/training/arbor/build> export PYTHONPATH=$PYTHONPATH:`pwd`/lib
kuesters@daint103:~/training/arbor/build> export CPATH=$CPATH:/opt/python/3.6.1.1/lib/python3.6/site-packages/mpi4py/include/
kuesters@daint103:~/training/arbor/build>
```

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@daint104:~/training/arbor/build> cmake .. -DARB_WITH_MPI=ON -DARB_WITH_PYTHON=ON -DARB_WITH_GPU=ON -DCMAKE_INSTALL_PREFIX=$HOME/training/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

```
[ 92%] Building CXX object example/ring/CMakeFiles/ring.dir/ring.cpp.o
[ 92%] Linking CUDA device code CMakeFiles/ring.dir/cmake_device_link.o
[ 93%] Linking CXX executable ../../bin/ring
[ 93%] Built target ring
Scanning dependencies of target lmorpho
[ 93%] Building CXX object lmorpho/CMakeFiles/lmorpho.dir/lmorpho.cpp.o
[ 94%] Building CXX object lmorpho/CMakeFiles/lmorpho.dir/lstem.cpp.o
[ 94%] Building CXX object lmorpho/CMakeFiles/lmorpho.dir/lsys_models.cpp.o
[ 94%] Building CXX object lmorpho/CMakeFiles/lmorpho.dir/morphio.cpp.o
[ 95%] Linking CUDA device code CMakeFiles/lmorpho.dir/cmake_device_link.o
[ 95%] Linking CXX executable ../../bin/lmorpho
[ 95%] Built target lmorpho
Scanning dependencies of target pyarb
[ 95%] Building CXX object python/CMakeFiles/pyarb.dir/cells.cpp.o
[ 95%] Building CXX object python/CMakeFiles/pyarb.dir/context.cpp.o
[ 95%] Building CXX object python/CMakeFiles/pyarb.dir/event_generator.cpp.o
[ 96%] Building CXX object python/CMakeFiles/pyarb.dir/domain_decomposition.cpp.o
[ 96%] Building CXX object python/CMakeFiles/pyarb.dir/identifiers.cpp.o
[ 96%] Building CXX object python/CMakeFiles/pyarb.dir/mpi.cpp.o
[ 97%] Building CXX object python/CMakeFiles/pyarb.dir/profiler.cpp.o
[ 97%] Building CXX object python/CMakeFiles/pyarb.dir/pyarb.cpp.o
[ 97%] Building CXX object python/CMakeFiles/pyarb.dir/recipe.cpp.o
[ 98%] Building CXX object python/CMakeFiles/pyarb.dir/simulation.cpp.o
[ 98%] Building CXX object python/CMakeFiles/pyarb.dir/spikes.cpp.o
[ 98%] Building CXX object python/CMakeFiles/pyarb.dir/strings.cpp.o
[ 98%] Linking CUDA device code CMakeFiles/pyarb.dir/cmake_device_link.o
[100%] Linking CXX shared module ../../lib/pyarb.cpython-36m-x86_64-linux-gnu.so
[100%] Built target pyarb
kuesters@daint104:~/training/arbor/build> █
```

BUILD AND INSTALL ARBOR FROM SOURCE CODE

Install

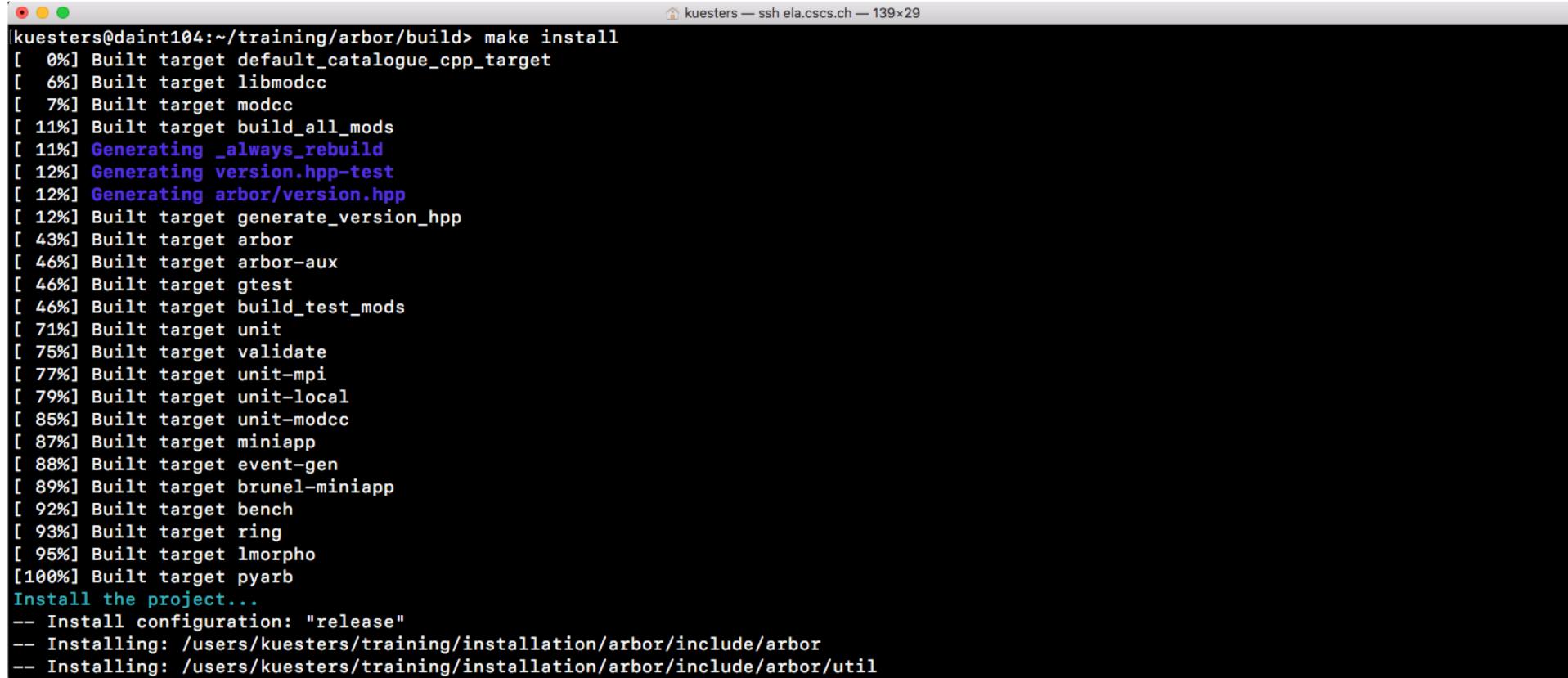
Steps to use Arbor's python frontend:

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



```
kuesters@daint104:~/training/arbor/build> make install
[  0%] Built target default_catalogue_cpp_target
[  6%] Built target libmodcc
[  7%] Built target modcc
[ 11%] Built target build_all_mods
[ 11%] Generating _always_rebuild
[ 12%] Generating version.hpp-test
[ 12%] Generating arbor/version.hpp
[ 12%] Built target generate_version_hpp
[ 43%] Built target arbor
[ 46%] Built target arbor-aux
[ 46%] Built target gtest
[ 46%] Built target build_test_mods
[ 71%] Built target unit
[ 75%] Built target validate
[ 77%] Built target unit-mpi
[ 79%] Built target unit-local
[ 85%] Built target unit-modcc
[ 87%] Built target miniapp
[ 88%] Built target event-gen
[ 89%] Built target brunel-miniapp
[ 92%] Built target bench
[ 93%] Built target ring
[ 95%] Built target lmorpho
[100%] Built target pyarb
Install the project...
-- Install configuration: "release"
-- Installing: /users/kuesters/training/installation/arbor/include/arbor
-- Installing: /users/kuesters/training/installation/arbor/include/arbor/util
```

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

```
kuesters@daint104:~/training/arbor/build> salloc -Cgpu -N 4 -t 60
salloc: Pending job allocation 10233673
salloc: job 10233673 queued and waiting for resources
salloc: job 10233673 has been allocated resources
salloc: Granted job allocation 10233673
kuesters@daint104:~/training/arbor/build> srun -n 4 ./bin/unit
[=====] Running 665 tests from 115 test cases.
[-----] Global test environment set-up.
[-----] 14 tests from algorithms
[ RUN    ] algorithmms.sum
[ OK     ] algorithmms.sum (0 ms)
[ RUN    ] algorithmms.make_index
[ OK     ] algorithmms.make_index (0 ms)
[ RUN    ] algorithmms.minimal_degree
[ OK     ] algorithmms.minimal_degree (0 ms)
[ RUN    ] algorithmms.is_strictly_monotonic_increasing
[ OK     ] algorithmms.is_strictly_monotonic_increasing (0 ms)
[ RUN    ] algorithmms.is_strictly_monotonic_decreasing
[ OK     ] algorithmms.is_strictly_monotonic_decreasing (0 ms)
[ RUN    ] algorithmms.all_positive
[ OK     ] algorithmms.all_positive (0 ms)
[ RUN    ] algorithmms.all_negative
[ OK     ] algorithmms.all_negative (0 ms)
[ RUN    ] algorithmms.has_contiguous_compartments
[ OK     ] algorithmms.has_contiguous_compartments (0 ms)
[ RUN    ] algorithmms.is_unique
[ OK     ] algorithmms.is_unique (0 ms)
[ RUN    ] algorithmms.child_count
[ OK     ] algorithmms.child_count (0 ms)
```