Aconity Control Documentation

Release 0.0

Ricardo Dominguez-Olmedo

Jul 11, 2019

CONTENTS:

CHAPTER

ONE

OVERVIEW

This software package offers real-time modeling and optimisation of a laser powder bed fusion process (AconityMINI). To do so, three scripts are executed simultenously: *aconity.py*, *machine.py* and *cluster.py*, the two former executed locally in the AconityMINI computer and the latter executed in a remote server for enhanced run-time performance.

- *aconity.py*: Makes use of the API provided by Aconity to automatically start, pause and resume a build, and to change individual part parameters in real-time.
- *machine.py*: Reads the raw sensory data outputted by the aconity machine, processes it into a low-dimensional state vector and uploads it a remote server for parameter optimisation.
- *cluster.py*: Computes optimal process parameters, at each layer, given feedback obtained from the machine sensors. Based on the deep reinforcement learning algorithm Probability Ensembles with Trajectory Sampling.

1.1 Program flow

- Layer is started by *performLayer()* in *aconity.py*
- Pyrometer data is read and processed in real-time by *getStates()* in *machine.py*
- When the layer is completed and all data has been read, the low-dimensional processed states are sent to the remote server by *sendStates()* in *machine.py*
- The states are received at the remote server by *getStates()* in *cluster.py*
- A new control action is computed (build parameters are optimised) according to the received feedback by *computeAction* in *cluster.py*
- The computed actions are saved to the remote server by *sendAction()* in *cluster.py*
- The computed actions are downloaded locally by *getActions()* in *machine.py*
- A new layer is built using the updated parameters by *performLayer()* in *aconity.py*

The Aconity API software package provided by Aconity3D must be installed in the computer connected to the Aconity machine according to Aconity's guidelines. The two files containing the bulk of the functionality of the API are *AconitySTUDIO_client.py* and *AconitySTUDIO_utils.py*.

CHAPTER

TWO

INSTALLING, RUNNING AND ENHANCING THE SOFTWARE

2.1 Installing required dependencies

The simplest way to install all required software packages is using *conda*.

The modeling and optimisation software requires TensorFlow. This package can be run on CPU or GPU (if one is available), the latter offering up to 100x faster run time performance.

For a CPU installation use:

```
conda create -n tf-cpu python=3.5
conda activate tf-cpu
conda install tensorflow==1.10
pip install dotmap scipy gpflow gym==0.9.4 pytest tqdm sklearn scikit-optimize
```

For a GPU installation use:

```
conda create -n tf-gpu python=3.5
conda activate tf-gpu
conda install tensorflow-gpu==1.10
pip install dotmap scipy gpflow gym==0.9.4 pytest tqdm sklearn scikit-optimize
```

If there are dependencies missing, these can be installed using *pip* or *conda* in the typical Python fashion.

2.2 Running the software

First, one must set the desired configuration for the build. The configuration files are:

- config_cluster.py
- config_windows.py
- config_dmbrl.py

Details regarding the available configurations can be found under the *Configuration* section.

After setting the desired configuration, one must:

- Run *aconity.py* in the AconityComputer (i.e. using the Python IDLE), and wait until the command line displays *"Waiting for actions…"*
- Open MobaXterm
  - Log into *USERNAME@scentrohpc.shef.ac.uk* and provide the pertinent password.
  - Run *source activate tf-cpu* (or whichever conda environment has the required dependencies)

- Run *cd software-path* where *software-path* is the location of the software package on the remote server
- Run *python cluster.py*
- Wait until the command line displays "Waiting for states…"

- Run *machine.py* (i.e. using the Python IDLE)

2.3 Enhancing the software

- To implement a different control strategy, modify the function *computeAction()* in *cluster.py*.
- To make changes to the current control strategy, modify the relevant files within *dmbrl/*
- To change how the pyrometer measurements are converted into the low-dimensional features used for modeling and control, change the function *getStates()* from *machine.py*.

2.3.1 Adding another sensor

To add another sensor one could change the function *getStates()* from *machine.py* to resemble:

```
states = np.zeros((n_parts, M+N)) # Initialise state vector
for part in range(n_parts): # Read information for all parts being monitored
  # Load raw data from sensors
  data_sensor1 = loadSensor1(file_path_to_part_sensor1)
  data_sensor2 = loadSensor2(file_path_to_part_sensor2)

  # Process raw data from sensors
  state_sensor1 = processDataSensor1(data_sensor1) # vector with shape (N,)
  state_sensor2 = processDataSensor2(data_sensor2) # vector with shape (M,)

  # Combine
  state = np.concatenate((state_sensor1, state_sensor2)) # shape (N+M,)
  states[part] = state
return states
```

One would also need to ensure that the new state representation is suitable for modeling the system with sufficient accuracy. To do so, convert the build data of interest into the state representation to be tested, train the model with the given state representation and check its accuracy in making predictions over previously unseen data (R2, RMSE…). Take the following script for reference:

```
import numpy as np
import tensorflow as tf
from dotmap import DotMap
import matplotlib.pyplot as plt
from dmbrl.modeling.models import BNN
from dmbrl.modeling.layers import FC

states = np.load(states_file) # Dimension (n_samples, n_states)
actions = np.load(actions_file) # Dimension (n_samples-1, n_actions)
XU = np.concatenate((X[:-1], U), axis=1) # inputs to the model
Yd = X[1:] # training targets

# Split data into train and test sets
test_ratio = 0.2
num_test = int(X.shape[0] * test_ratio)
```

(continues on next page)

(continued from previous page)

```
permutation = np.random.permutation(X.shape[0])

train_x, test_x = XU[permutation[num_test:]], XU[permutation[:num_test]]
train_y, test_y = Yd[permutation[num_test:]], Yd[permutation[:num_test]]

# Before this, define the model parameters model_in, model_out, n_layers, n_neurons,␣
↪l_rate, wd_in, wd_hid, wd_out, num_networks
sess = tf.Session()
params = DotMap(name="model1", num_networks=num_networks, sess=sess)
model = BNN(params)
self.model.add(FC(n_neurons, input_dim=model_in, activation="swish", weight_decay=wd_
↪in))
for i in range(n_layers): self.model.add(FC(n_neurons, activation="swish", weight_
↪decay=wd_hid))
model.add(FC(model_out, weight_decay=wd_out))
model.finalize(tf.train.AdamOptimizer, {"learning_rate": l_rate})

# Train and make the predictions
model.train(train_x, train_y, batch_size, training_epochs, rescale=True)
predicted_y, var_y = model.predict(test_x) # Essentially the mean and variance of the␣
↪prediction, as it is a probabilistic model

# Compute metrics (R2 and RMSE should be previously define functions)
r2_metric = R2(test_y, predicted_y)
rmse_metric = RMSE(test_y, predicted_y)
```

When incorporating a new sensor, you most likely want to change how the scaling of the data is done, so that the magnitude of all your state elements is similar. Otherwise model performance will be degraded. To change the scaler functionality, make changes to *dmbrl/modeling/utils/ModelScaler.py* as needed.

2.3.2 Dividing a part into multiple "subparts"

This approach aims to improve *intra*-layer temperature homogeneity. There are two sides to this problem. First, one probably wants to model the entire part as one. To do so, one should combine the sensory information obtained for all subparts into a single state vector, in a similar manner to explanation above (but instead of combining sensors, combining parts).

Secondly, you must configure the MPC class to output more parameters. For instance, if you have 7 sub-parts and want to select distinct laser power and scan configurations in each of them, then your action vector should have dimension 7 * 2 = 14. To change the dimension of the vector, simply ensure all *ac_lb*, *ac_up* and *constrains* variables fed to the *MPC* class have the correct dimension, i.e check that ac_lb.shape==14, ac_up.shape==14 and same with *constrains*.

Then, you must make changes to the function *performLayer()* within *aconity.py* so that the correct parts are addressed when changing the laser power and scan speed (must be able to handle action inputs with secondary dimension > 2).

CHAPTER

THREE

CONFIGURATION PARAMETERS

The desired configuration for the build is set on the following files:

- *config_windows.py*: General configuration parameters concerning the build, such as number of parts, build parameters, . . .
- *config_dmbrl.py*: Low-level control specific configuration. Generally one would not need to change this file, but rather *config_cluster.py*
- *config_cluster.py*: Control configuration, divided into 'pretrained' (model trained using data collected previously) and *unfamiliar* (model learned in real-time).

3.1 config_windows.py

- LASER_ON (bool): Laser is enabled when True.
- JOB_NAME (str): Job name as displayed in the AconitySTUDIO web application.
- LAYERS (array of int): Layer range to be built, as [layer_min, layer_max].
- N_PARTS (int): Number of parts to be built (not regarding ignored parts).
- N_STATES (int): Number of low-dimensional states used for the processing of the raw pyrometer data.
- TEMPERATURE_TARGET (float): Temperature target in mV.
- N_PARTS_IGNORED (int): Number of additional parts to be built on top of *N_PARTS* (pyrometer may not record data for the first few parts).
- IGNORED_PARTS_SPEED (float): Scan speed used for parts being "ignored".
- IGNORED_PARTS_POWER (float): Laser power used for parts being "ignored".
- N_PARTS_FIXED_PARAMS (int): Number of parts built using fixed build parameters.
- FIXED_PARAMS (array): Parameters to be used for those parts being built with fixed build parameters, as [speed (m/s), power (W)]
- SLEEP_TIME_READING_FILES (float): Time between a sensor data file being first detected and attempting to read it. Prevents errors emerging from opening the file while it is still being written.
- PART_DELTA (int): Parts of interest may increase 1 by 1, or 3 by 3 (refer to the AconitySTUDIO web application).

3.2 config_dmbrl.py and config_cluster.py

- ctrl_cfg: Configuration parameters for the control algorithm.

  -dO: dimensionality of observations -dU: dimensionality of control inputs - per: How often the action sequence will be optimized, i.e, for per=1 it is reoptimized at every call to *MPC.act()*. - constrains: [[np.array([min v, min q]), np.array([max v, max q])], [min q/v, max q/v], [min q/sqrt(v), max q/sqrt(v)]] - prop_cfg: Configuration parameters for modeling and uncertainty propagation.

  - model_pretrained: *True* if model used for MPC has been trained on previous data, *False* otherwise.
  - model_init_cfg: Configuration parameters for model initialisation.
    * ensemble_size: Number of models within the ensemble.
    * load_model: *True* for a pretrained model to be loaded upon initialisation.
    * model_dir: Directory in which the model files (.mat, .nns) are located.
    * model_name: Name of the model files (model_dir/model_name.mat or model_dir/model_name.nns)
  - model_train_cfg: Configuration parameters for model training optimisation
    * batch_size: Batch size.
    * epochs: Number of training epochs.
    * hide_progress: If 'True', additional information regarding model training is printed.
  - npart: Number of particles used for uncertainty propagation.
  - model_in: Number of inputs to the model.
  - model_out: Number of outputs to the model.
  - n_layers: Number of hidden layers.
  - n_neurons: Number of neurons per hidden layer.
  - learning_rate: Learning rate.
  - wd_in: Weight decay for the input layer neurons.
  - wd_hid: Weight decay for the hidden layer neurons.
  - wd_out: Weight decay for the output layer neurons.
  - opt_cfg: Configuration parameters for optimisation.
    * mode: Uncertainty propagation method.
    * plan_hor: Planning horizon for the model predictive control algorithm.
    * cfg
      · popsize: Number of cost evaluations per iteration.
      · max_iters: Maximum number of optimisation iterations.
      · num_elites: Number of elites.
      · alpha: Alpha parametero of the CEM optimisation algorithm.
      · eps: Epsilon parameter of the CEM optimisation algorithm.
    * prop_cfg

      · mode: Uncertainty propagation method, ie "TSinf"

- change_target: True if multiple setpoints used, i.e. 980 and 1010
- n_parts_targets: Number of parts to be built for each target
- targets: Different temperature setpoints to be used (must be of same length as *n_parts_targets*)
- force: Configuration parameters to periodically overwrite ("force") predefined build parameters
  * on: Force functionality enabled if True
  * start_part: First part where functionality is enabled (disregarding the first few ignored parts)
  * n_parts: Number of parts for which the functionality is enabled
  * n_repeats: Number of consecutive layers for which inputs are forced. For [1,2], n_parts will be forced only once (periodically), while a further n_parts will be forced two times consecutively (periodically)
  * init_buffer: Initial number of layers for which parameters are not forced
  * upper_init: Upper bound is initialised to this.
  * upper_delta: Upper bound increases by this. For instance, for upper_init=105 and upper_delta=5, the upper bound sequence will be 105, 110, 115...
  * lower_init: Lower bound is initialised to this.
  * lower_delta: Lower bound is increased by this. For instance, for lower_init=65 and lower_delta=-5, the lower bound sequence will be 60, 55, 50...
  * fixed_speed: For the forced parameters, power will be adjusted but mark speed will be kept fixed to this value.

CHAPTER FOUR

ACONITY API

4.1 Creating and configuring the client

The client is created as follows:

```
from AconitySTUDIO_client import AconitySTUDIOPythonClient

login_data = {
  'rest_url' : 'http://192.168.1.1:9000',
  'ws_url' : 'ws://192.168.1.1:9000',
  'email' : 'admin@yourcompany.com',
  'password' : '<password>'
}

client = await AconitySTUDIOPythonClient.create(login_data)
```

Each job has a unique identifier which must be known in order to interact with said job. To automatically gather and set the correct job id for the Python Client use:

```
await client.get_job_id('TestJob')
```

This will automatically create an attribute *job_id*. From now on, if any method of the Python Client would require a job id, you can omit this argument in the function call. If you chose to explicitly fill in this parameter in a function call, the clients own attribute (if it exists at all) will be ignored.

For normal operation of the Python Client, identifiers of the configuration and the machine itself must be known aswell:

```
await client.get_machine_id('my_unique_machine_name')
await client.get_config_id('my_unique_config_name')
```

If multiple machines, configurations or jobs exist with the same name, they need to be looked up in the browser url field and given to the Python Client manually:

```
client.job_id = '5c4bg4h21a00005a00581012'
client.machine_id = 'your_machine_id_gathered_from_browser_url_bar'
client.config_id = 'your_config_id_gathered_from_browser_url_bar'
```

4.2 Script execution

Use the *execute()* coroutine. For instance:

```
light_on = '$m.on($c[light])'
await client.execute(channel='manual', script=light_on)
movement = '$m.move_rel($c[slider], -180)'
await client.execute(channel='manual_move', script=movement)
```

These commands get executed on different channels. If a channel is occupied, any command sent to that channel will be ignored. The execute coroutine takes care of this because if you await it, it will only yield control to its caller once the channel is free again. This could be bypassed by commenting out some of the source code.

4.3 Job management

Job management comprises the starting, pausing, resuming and stopping of jobs.

For starting a job, we need to specify the job id, an execution script, and which layers shall be built with which parts. If we have set the attribute job_id and all parts should be built, a job can be started like this:

```
layers = [1,3] #build layer 1,2,3

execution_script = \
'''layer = function(){
for(p:$p){
  $m.expose(p[next;$h],$c[scanner_1])
}
$m.add_layer($g)
}
repeat(layer)'''

await start_job(layers, execution_script)
```

This does not take care of starting a config or importing parameters from the config into a job. This needs to be done in the GUI beforehand. Of course, it is always possible to do the basic job configuration via the REST API in the Python Client, but no convenience functions exist to simplify these tasks.

After a job is paused (*await client.pause_job()*‘), one can change parameters. For instance, subpart *001_s1_vs* shall be exposed with a different laser power:

```
part_id = 1 #part_id of 001_s1_vs. See next section `Documentation of all functions`.
param = 'laser_power'
new_laser_power = 123
await client.change_part_parameter(part_id, param, new_value)
```

Changing a global parameter can be done via:

```
param = 'supply_factor'
new_value = 2.2
await client.change_global_parameter(param, new_value)
```

4.4 How pyrometer data is saved

Pyrometer data is automatically saved by the AconityMINI as follows:

```
log
  |-session_2019_03_08_16_2etc - date
    |-config_1_etc -
      |-job_N_id
        |-sensors
          |-2Pyrometer
            |-pyrometer2
              |-1 - often missing files
              |-4
                |- 0.03.pcd
                |- 0.06.pcd
                |- 0.09.pcd
                |-...
              |-7
              |-...
```

The session directory is created upon starting the AconitySTUDIO web application. The config directory is created upon starting the *Unheated 3D Monitoring* functionality. The job folder is created upon starting script execution.

CHAPTER

FIVE

CODE DOCUMENTATION

5.1 aconity module

5.2 machine module

class machine.**Machine**(*shared_cfg*, *machine_cfg*)

Bases: object

Reads the raw sensory data outputted by the aconity machine, processes it into a low-dimensional state vector and uploads it a remote server for parameter optimisation.

Parameters

- **shared_cfg** (*dotmap*) –
  - **n_ignore** (*int*): Number of additional parts to be built on top of *env.n_parts* (pyrometer may not record data for the first few parts).
  - **env.nS** (*int*): Dimensionality of the state vector.
  - **comms** (*dotmap*): Configuration parameters for server communication.
- **machine_cfg** (*dotmap*) –
  - **aconity.layers** (*array of int*): Layer range to be built, as [layer_min, layer_max].
  - **aconity.open_loop** (*np.array*): Parameters used to build the parts built using fixed parameters, np.array with shape (*n_fixed_parts*, 2).
  - **aconity.n_parts** (*int*): Number of parts to be built, excluding ignored parts.
  - **process.sess_dir** (*str*): Folder where pyrometer data is stored by the Aconity machine.
  - **process.sleep_t** (*float*): Time between a sensor data file being first detected and attempting to read it. Prevents errors emerging from opening the file while it is still being written.

getActions()

Download locally the action file outputted by the remote server.

getFileName(*layer*, *piece*)

Returns the pyrometer data file path for a given layer and part number.

This function accounts for the parts being ignored. The layer thickness is 0.03 mm.

Parameters

- **layer** (*int*) – Layer number.
- **piece** (*int*) – Part number.

Returns File path.

Return type str

getStates()

Read the raw data outputted from the pyrometer and processes it into low-dimensional state vectors.

For every part that must be observed, the raw data is red from the file outputted by the Aconity machine, cold lines are removed, the data pertaining to the object manufactured is kept, and it is discretised into a number of discrete regions in which the mean sensor value is computed.

Returns State vectors with shape (*n_parts*, *nS*)

Return type np.array

initProcessing()

Obtains the folder in which data will be written by the pyrometer sensor.

This function automatically detects the latest session and job folders.

log(*states*)

loop()

Iteratively obtain next layer's parameters from the remote server, read and process raw pyrometer data, and upload the low-dimensional states to the remote server to compute the next set of optimal parameters.

Allows the class functionality to be conveniently used as follows:

```
machine = Machine(s_cfg, m_cfg)
machine.loop()
```

pieceNumber(*piece_indx*, *buffer*)

Returns the index given by AconityStudio to each individual part.

For instance, if the first part should be ignored, and part numbers increase three by three, then *return int((piece_indx+1)*3+1)* should be used, thus 0 -> 4, 1 -> 7, 2 -> 10, etc.

On the other hand, if the first three parts should be ignored, and part numbers increase one by one, then *return int((piece_indx+3)+1)* should be used, thus 0 -> 4, 1 -> 5, 2 -> 6, etc.

Parameters

- **piece_indx** (`int`) – Input index, starting from 0.
- **n_ignore** (`int`) – Number of initial parts that should be ignored.

Returns Output index as used by AconityStudio.

Return type int

sendStates(*states*)

Uploads to the the remote server the input state vector.

Parameters **states** (`np.array`) – Processed state vector.

5.3 cluster module

class cluster.**Cluster**(*shared_cfg*, *pretrained_cfg*, *learned_cfg*)

Bases: `object`

Computes optimal process parameters, at each layer, given feedback obtained from the machine sensors.

Parameters

- **shared_cfg** (*dotmap*) –
  - **env.n_parts** (*int*): Total number of parts built under feedback control.
  - **env.horizon** (*int*): Markov Decision Process horizon (here number of layers).
  - **env.nS** (*int*): Dimension of the state vector.
  - **comms** (*dotmap*): Parameters for communication with other classes.
- **pretrained_cfg** (*dotmap*) –
  - **n_parts** (*dotmap*): Number of parts built under this control scheme.
  - **ctrl_cfg** (*dotmap*): Configuration parameters passed to the MPC class.
- **learned_cfg** (*dotmap*) –
  - **n_parts** (*dotmap*): Number of parts built under this control scheme.
  - **ctrl_cfg** (*dotmap*): Configuration parameters passed to the MPC class.

clearComms()

computeAction(*states*)

Computes the control actions given the observed system states.

Parameters **states** (*np.array*) – Observed states, shape (*n_parts*, *nS*)

Returns Computed actions, with shape (*n_parts*, *nU*)

Return type np.array

getStates()

Load state vectors uploaded to the server by the *Machine* class.

This function waits for the *comms.dir/comms.state.rdy_name* folder to be created by the *Machine* class, before reading the file where the states are located, *comms.dir/comms.state.f_name*

Returns State vector with shape (*n_parts*, *nS*)

Return type np.array

initAction()

Returns the initial action vector.

This function is required because an initial layer must be built before any feedback is available.

Returns Initial action vector with shape (*n_parts*, *nU*)

Return type np.array

log()

Logs the state and action trajectories, as well as the predicted cost, which may be of interest to tune some algorithmic parameters.

loop()

While within the time horizon, read the states provided by the *Machine* class, and compute and save the corresponding actions.

Allows the class functionality to be conveniently used as follows:

```
cluster = Cluster(s_cfg, cp_cfg, cl_cfg)
cluster.loop()
```

sendAction (*actions*)
Saves the computed actions.

Signals the *Machine* class that actions are ready to be downloaded by locally creating the *comms.dir/comms.action.rdy_name* folder

Parameters **actions** (`np.array`) – Action vector with shape (*n_parts*, *nU*)

5.4 controllers package

5.4.1 Submodules

5.4.2 controllers.Controller module

class `controllers.Controller.`**Controller** (**args*, ***kwargs*)
Bases: `object`

Framework of a controller class

act (*obs*, *t*, *get_pred_cost=False*)
Performs an action.

dump_logs (*primary_logdir*, *iter_logdir*)
Dumps logs into primary log directory and per-train iteration log directory.

reset ()
Resets this controller.

train (*obs_trajs*, *acs_trajs*, *rews_trajs*)
Trains this controller using lists of trajectories.

5.4.3 controllers.MPC module

class `controllers.MPC.`**MPC** (*params*)
Bases: `controllers.Controller.Controller`

Parameters **params** (`dotmap`) – Configuration parameters. .env (gym.env):

Environment for which this controller will be used.

.update_fns (list<func>): A list of functions that will be invoked (possibly with a tensorflow session) every time this controller is reset.

.ac_ub (np.ndarray): (optional) An array of action upper bounds. Defaults to environment action upper bounds.

.ac_lb (np.ndarray): (optional) An array of action lower bounds. Defaults to environment action lower bounds.

.per (int): (optional) Determines how often the action sequence will be optimized. Defaults to 1 (reoptimizes at every call to act()).

.prop_cfg

.model_init_cfg (DotMap): A DotMap of initialization parameters for the model. .model_constructor (func):

A function which constructs an instance of this model, given model_init_cfg.

.model_train_cfg (dict): (optional) A DotMap of training parameters that will be passed into the model every time is is trained. Defaults to an empty dict.

.model_pretrained (bool): (optional) If True, assumes that the model has been trained upon construction.

.mode (str): Propagation method. Choose between [E, DS, TSinf, TS1, MM]. See https://arxiv.org/abs/1805.12114 for details.

.npart (int): Number of particles used for DS, TSinf, TS1, and MM propagation methods.

.ign_var (bool): (optional) Determines whether or not variance output of the model will be ignored. Defaults to False unless deterministic propagation is being used.

.obs_preproc (func): (optional) A function which modifies observations (in a 2D matrix) before they are passed into the model. Defaults to lambda obs: obs. Note: Must be able to process both NumPy and Tensorflow arrays.

.obs_postproc (func): (optional) A function which returns vectors calculated from the previous observations and model predictions, which will then be passed into the provided cost function on observations. Defaults to lambda obs, model_out: model_out. Note: Must be able to process both NumPy and Tensorflow arrays.

.obs_postproc2 (func): (optional) A function which takes the vectors returned by obs_postproc and (possibly) modifies it into the predicted observations for the next time step. Defaults to lambda obs: obs. Note: Must be able to process both NumPy and Tensorflow arrays.

.targ_proc (func): (optional) A function which takes current observations and next observations and returns the array of targets (so that the model learns the mapping obs -> targ_proc(obs, next_obs)). Defaults to lambda obs, next_obs: next_obs. Note: Only needs to process NumPy arrays.

.opt_cfg

.mode (str): Internal optimizer that will be used. Choose between [CEM, Random].

.cfg (DotMap): A map of optimizer initializer parameters.

.plan_hor (int): The planning horizon that will be used in optimization.

.obs_cost_fn (func): A function which computes the cost of every observation in a 2D matrix. Note: Must be able to process both NumPy and Tensorflow arrays.

.ac_cost_fn (func): A function which computes the cost of every action in a 2D matrix.

.constrains (np.array): An array with the optimisation constrains = [[lb, ub], [lc1, uc1], [lc2, uc2]] so that if u = [v, q], lb <= u <= ub, lc1 <= q/v <= uc2, lc2 <= q/sqrt(v) <= uc2. Overwrites ac_lb and ac_ub is constrains[0] is not None.

.log_cfg

.save_all_models (bool): (optional) If True, saves models at every iteration. Defaults to False (only most recent model is saved). Warning: Can be very memory-intensive.

.log_traj_preds (bool): (optional) If True, saves the mean and variance of predicted particle trajectories. Defaults to False.

.log_particles (bool) (optional) If True, saves all predicted particles trajectories. Defaults to False. Note: Takes precedence over log_traj_preds. Warning: Can be very memory-intensive

act (*obs*, *t*, *get_pred_cost=False*)
Returns the action that this controller would take at time t given observation obs.

Parameters

- **obs** – The current observation
- **t** – The current timestep
- **get_pred_cost** – If True, returns the predicted cost for the action sequence found by the internal optimizer.

Returns: An action (and possibly the predicted cost)

changePlanHor (*T*, *freq=None*, *change_over=False*)
Dynamically changes the planning horizon of the MPC algorithm.

Parameters **T** (*int*) – New planning horizon.

changeTargetCost (*target*)

dump_logs (*primary_logdir*, *iter_logdir*)
Saves logs to either a primary log directory or another iteration-specific directory. See __init__ documentation to see what is being logged.

Parameters

- **primary_logdir** (*str*) – A directory path. This controller assumes that this directory does not change every iteration.
- **iter_logdir** (*str*) – A directory path. This controller assumes that this directory changes every time dump_logs is called.

Returns: None

optimizers = {'CEM': <class 'dmbrl.misc.optimizers.cem.CEMOptimizer'>, 'Random': <cla

reset ()
Resets this controller (clears previous solution, calls all update functions).

Returns: None

train (*obs_trajs*, *obs_prime_trajs*, *acs_trajs*)
Trains the internal model of this controller. Once trained, this controller switches from applying random actions to using MPC.

Parameters

- **obs_trajs** – (N, nS)
- **obs_prime_trajs** – (N, nS) observations at next time step
- **acs_trajs** – (N, nU)

Returns: None.

5.4.4 Module contents

5.5 layers package

5.5.1 Submodules

5.5.2 layers.FC module

class `layers.FC.`**FC**(*output_dim*, *input_dim=None*, *activation=None*, *weight_decay=None*, *ensemble_size=1*)

Bases: `object`

Represents a fully-connected layer in a network.

Parameters

- **output_dim** – (int) The dimensionality of the output of this layer.
- **input_dim** – (int/None) The dimensionality of the input of this layer.
- **activation** – (str/None) The activation function applied on the outputs. See FC._activations to see the list of allowed strings. None applies the identity function.
- **weight_decay** – (float) The rate of weight decay applied to the weights of this layer.
- **ensemble_size** – (int) The number of networks in the ensemble within which this layer will be used.

compute_output_tensor(*input_tensor*)

Returns the resulting tensor when all operations of this layer are applied to input_tensor.

If input_tensor is 2D, this method returns a 3D tensor representing the output of each layer in the ensemble on the input_tensor. Otherwise, if the input_tensor is 3D, the output is also 3D, where output[i] = layer_ensemble[i](input[i]).

Parameters **input_tensor** – (tf.Tensor) The input to the layer.

Returns: The output of the layer, as described above.

construct_vars()

Constructs the variables of this fully-connected layer.

Returns: None

copy(*sess=None*)

Returns a Layer object with the same parameters as this layer.

Parameters

- **sess** – (tf.Session/None) session containing the current values of the variables to be copied. Must be passed in to copy values.
- **copy_vals** – (bool) Indicates whether variable values will be copied over. Ignored if the variables of this layer has not yet been constructed.

Returns: The copied layer.

get_activation(*as_func=True*)

Returns the current activation function for this layer.

Parameters **as_func** – (bool) Determines whether the returned value is the string corresponding to the activation function or the activation function itself.

Returns: The activation function (string/function, see as_func argument for details).

get_decays()
Returns the list of losses corresponding to the weight decay imposed on each weight of the network.

Returns: the list of weight decay losses.

get_ensemble_size()
Returns the ensemble size.

Returns: int

get_input_dim()
Returns the dimension of the input.

Returns: The dimension of the input

get_output_dim()
Returns the dimension of the output.

Returns: The dimension of the output.

get_vars()
Returns the variables of this layer.

get_weight_decay()
Returns the current rate of weight decay set for this layer.

Returns: The weight decay rate.

set_activation(*activation*)
Sets the activation function for this layer.

Parameters **activation** – (str) The activation function to be used.

Returns: None.

set_ensemble_size(*ensemble_size*)
Sets the ensemble size.

Parameters **ensemble_size** (*int*) –

Returns: None

set_input_dim(*input_dim*)
Sets the dimension of the input.

Parameters **input_dim** – (int) The dimension of the input.

Returns: None

set_output_dim(*output_dim*)
Sets the dimension of the output.

Parameters **output_dim** – (int) The dimension of the output.

Returns: None.

set_weight_decay(*weight_decay*)
Sets the current weight decay rate for this layer.

Returns: None

unset_activation()
Removes the currently set activation function for this layer.

Returns: None

unset_weight_decay()

Removes weight decay from this layer.

Returns: None

5.5.3 Module contents

5.6 misc package

5.6.1 Subpackages

misc.optimizers package

Submodules

misc.optimizers.cem module

class misc.optimizers.cem.**CEMOptimizer**(*sol_dim*, *max_iters*, *popsize*, *num_elites*, *constrains*, *tf_session=None*, *epsilon=0.001*, *alpha=0.25*, *max_resamples=10*)

Bases: *misc.optimizers.optimizer.Optimizer*

A Tensorflow-compatible CEM optimizer.

Parameters

- **sol_dim** (*int*) – The dimensionality of the problem space
- **max_iters** (*int*) – The maximum number of iterations to perform during optimization
- **popsize** (*int*) – The number of candidate solutions to be sampled at every iteration
- **num_elites** (*int*) – The number of top solutions that will be used to obtain the distribution at the next iteration.
- **constrains** (*array*) – [[np.array([min v, min q]), np.array([max v, max q])], [min q/v, max q/v], [min q/sqrt(v), max q/sqrt(v)]]
- **tf_session** (*tf.Session*) – (optional) Session to be used for this optimizer. Defaults to None, in which case any functions passed in cannot be tf.Tensor-valued.
- **epsilon** (*float*) – A minimum variance. If the maximum variance drops below epsilon, optimization is stopped.
- **alpha** (*float*) – Controls how much of the previous mean and variance is used for the next iteration. next_mean = alpha * old_mean + (1 - alpha) * elite_mean, and similarly for variance.

changeSolDim(*sol_dim*)

Change the dimension of the CEM optimisation solution.

Parameters **sol_dim** (*int*) – New dimension of the CEM optimisation solution.

obtain_solution(*init_mean*, *init_var*)

Optimizes the cost function using the provided initial candidate distribution

Parameters

- **init_mean** (*np.ndarray*) – The mean of the initial candidate distribution.

- **init_var** (*np.ndarray*) – The variance of the initial candidate distribution.

reset()
Blank function for compatibility with optimisation class framework.

setup(*cost_function*, *tf_compatible*)
Sets up this optimizer using a given cost function.

Parameters

- **cost_function** (*func*) – A function for computing costs over a batch of candidate solutions.
- **tf_compatible** (*bool*) – True if the cost function provided is tf.Tensor-valued.

Returns: None

misc.optimizers.optimizer module

class misc.optimizers.optimizer.**Optimizer**(**args*, ***kwargs*)
Bases: `object`

Framework for Optimizer subclasses

obtain_solution(**args*, ***kwargs*)
Compute optimisation problem solution

reset()
Function iteratively called at MPC.act()

setup(*cost_function*, *tf_compatible*)
Function called upon initialisation of the MPC class.

misc.optimizers.random module

class misc.optimizers.random.**RandomOptimizer**(*sol_dim*, *popsize*, *tf_session*, *upper_bound=None*, *lower_bound=None*)
Bases: `misc.optimizers.optimizer.Optimizer`

Random shooting optimisation.

Parameters

- **sol_dim** (*int*) – The dimensionality of the problem space
- **popsize** (*int*) – The number of candidate solutions to be sampled at every iteration
- **num_elites** (*int*) – The number of top solutions that will be used to obtain the distribution at the next iteration.
- **tf_session** (*tf.Session*) – (optional) Session to be used for this optimizer. Defaults to None, in which case any functions passed in cannot be tf.Tensor-valued.
- **upper_bound** (*np.array*) – An array of upper bounds
- **lower_bound** (*np.array*) – An array of lower bounds

obtain_solution(**args*, ***kwargs*)
Optimizes the cost function provided in setup().

Parameters

- **init_mean** (*np.ndarray*) – The mean of the initial candidate distribution.

- **init_var** (*np.ndarray*) – The variance of the initial candidate distribution.

reset()
Function iteratively called at MPC.act()

setup(*cost_function*, *tf_compatible*)
Sets up this optimizer using a given cost function.

Parameters

- **cost_function** (*func*) – A function for computing costs over a batch of candidate solutions.
- **tf_compatible** (*bool*) – True if the cost function provided is tf.Tensor-valued.

Returns: None

Module contents

5.6.2 Submodules

5.6.3 misc.DotmapUtils module

misc.DotmapUtils.**get_required_argument**(*dotmap*, *key*, *message*, *default=None*)
Returns an argument from a dotmap object, raises and error if it does not exist.

Parameters

- **dotmap** (*dotmap*) –
- **key** (*str*) –
- **message** (*str*) – Error message to be raised.

5.6.4 Module contents

5.7 utils package

5.7.1 Submodules

5.7.2 utils.ModelScaler module

class utils.ModelScaler.**ModelScaler**(*xdim=None*)
Bases: object

Normalise the inputs and outputs to the NN model.

fit(*inputs*, *targets*)
Fits the scaler to the model inputs and targets.

Parameters

- **inputs** (*np.array or tf.Tensor*) – shape N x 16 + 2
- **targets** (*np.array or tf.Tensor*) – shape N x 16

get_vars()
Returns the tf.variables of the scaler objects used

inverse_transformInput (*input*)
Returns the inverse transform of the inputs

Parameters **input** (*np.array or tf.Tensor*) – shape N x nS + nU or M x N x nS + nU

Returns shape N x nS + nU or M x N x nS + nU

Return type np.array or tf.Tensor

inverse_transformOutput (*mean*, *variance*)
Normalises the inverse transform of the targets to the NN model.

Parameters

- **mean** (*np.array or tf.Tensor*) – shape N x nS
- **variance** (*np.array or tf.Tensor*) – shape N x nS

Returns shape N x nS

Return type np.array or tf.Tensor

transformInput (*input*)
Normalises the inputs to the NN model.

Parameters **input** (*np.array or tf.Tensor*) – shape N x nS + nU or M x N x nS + nU

Returns shape N x nS + nU or M x N x nS + nU

Return type np.array or tf.Tensor

transformTarget (*targets*)
Normalises the targets to the NN model.

Parameters **targets** (*np.array or tf.Tensor*) – shape N x nS

Returns shape N x nS

Return type np.array or tf.Tensor

5.7.3 utils.TensorStandardScaler module

class utils.TensorStandardScaler.**TensorStandardScaler** (*x_dim*, *name=0*)
Bases: object

Helper class for automatically normalizing inputs into the network.

cache ()
Caches current values of this scaler.

Returns: None.

fit (*data*)
Runs two ops, one for assigning the mean of the data to the internal mean, and another for assigning the standard deviation of the data to the internal standard deviation. This function must be called within a 'with <session>.as_default()' block.

Arguments: data (np.ndarray): A numpy array containing the input

Returns: None.

get_vars ()
Returns a list of variables managed by this object.

Returns: (list<tf.Variable>) The list of variables.

inverse_transform(*data*)
Undoes the transformation performed by this scaler.

Arguments: data (np.array): A numpy array containing the points to be transformed.

Returns: (np.array) The transformed dataset.

load_cache()
Loads values from the cache

Returns: None.

transform(*data*)
Transforms the input matrix data using the parameters of this scaler.

Arguments: data (np.array): A numpy array containing the points to be transformed.

Returns: (np.array) The transformed dataset.

class utils.TensorStandardScaler.**TensorStandardScaler1D**(*name=0*)
Bases: *utils.TensorStandardScaler.TensorStandardScaler*

Helper class for automatically normalizing inputs into the network.

fit(*data*)
Runs two ops, one for assigning the mean of the data to the internal mean, and another for assigning the standard deviation of the data to the internal standard deviation. This function must be called within a 'with <session>.as_default()' block.

Arguments: data (np.ndarray): A numpy array containing the input

Returns: None.

5.7.4 Module contents

5.8 AconitySTUDIO_client module

class AconitySTUDIO_client.**AconitySTUDIO_client**(*login_data*)
Bases: object

The AconitySTUDIO Python Client. Allows for easy automation and job management.

For example usages, please consult the examples folder in the root directory from this repository.

To create the client call the *classmethod* create.

change_global_parameter(*param*, *value*, *check_boundaries=True*)
Change a global parameter in the locally saved job and synchronizes this change with the Server Database.

If the parameter may only have values confined in a certain range, the new value will be changed to fit these requirements. (Example: The parameter must lie in the interval [1, 10]. If the attempted change is to set the value to 12 the function sets it to 10.)

Parameters

- **param** (*string*) – The parameter to be changed. Example: 'supply_factor'
- **value** (*int/float/bool*) – The new value of the parameter to be changed
- **check_boundaries** (*bool*) – Ignore min and max values of a parameter.

Note: Calling this function does not mean that a running job will be paused and resumed with the updated value.

change_part_parameter(*part_id*, *param*, *value*, *laser='*'*, *check_boundaries=True*)
Change a part parameter in the locally saved job and synchronizes this change with the Server Database.

If the parameter may only have values confined in a certain range, the new value will be changed to fit these requirements. (Example: The parameter must lie in the interval [1, 10]. If the attempted change is to set the value to 12 the function sets it to 10.)

Note: Calling this function does not mean that a running job will be paused and resumed with the updated value.

Parameters

- **part_id** (*int*) – The part id to be changed. For example, this number can be seen in the GUI inside the jobs view -> clicking on a part -> expanding the part -> a number within "[ ]" is appearing. Other possibility: In the Script tab -> Init/Resume there are lines like "$p.add(4,2,_modelsection_002_s1_vs)". part_id -> 4.
- **param** (*string*) – The parameter to be changed. Example: 'laser_power'
- **value** (*int/float/bool*) – The new value of the parameter to be changed
- **laser** (*int*) – Used to select the scanner. Either '*' (->"Scanner All") or 1, 2, 3, 4 etc …
- **check_boundaries** (*bool*) – Ignore min and max values of a parameter.

config_exists(*config_name=None*, *config_id=None*)
Checks if a config exists.

One can *either* use the config_name or the config_id as a search parameter (XOR). If this is not done, raises a ValueError.

Parameters

- **config_name** (*str*) – Name of the config
- **config_id** (*str*) – Id of the config

Return type bool

config_has_component(*component*, *config_id=None*)
Checks if a config has a certain component.

Parameters

- **component** (*string*) – The component to be checked.
- **config_id** (*string*) – Config Id. If *config_id == None*, the client uses its own attribute config_id.
- **config_name** (*string*) – Config Name.

Return type bool

config_state(*config_id=None*)
Returns the current state of the config

Parameters **config_id** (*str*) – Id of the config. If none is given, the client uses its own attribute *config_id*.

Returns 'operational', 'inactive', or 'initialized'

Return type string

classmethod create(*login_data*)
Factory class method to initialize a client. Convenient as this function takes care of logging in and creating a websocket connection. It will also set set up a ping, to ensure the connection will not be lost.

Parameters **login_data** (*dictionary*) – required keys are *rest_url*, *ws_url*, *password* and *email*.

Usage:

```
login_data = {
    'rest_url' : 'http://192.168.1.1:2000',
    'ws_url' : 'ws://192.168.1.1:2000',
    'email' : 'admin@yourcompany.com',
    'password' : '<password>'
}
client = await AconitySTUDIO_client.create(login_data)
```

download_chunkwise(*url*, *save_to*, *chunk_size=1024*)

enable_pymongo_database(*name='database_test'*, *keep_last=120*)
Setup for the Mongodatabase

Parameters

- **mongodatabase** (*string*) – name of the database
- **keep_last** (*float*) – If larger that zero, automatically delete entries older than keep_last seconds

execute(*channel*, *script*, *machine_id=None*)
Sends scripts (commands) to the WebSocket Server.

Parameters

- **machine_id** (*string*) – Machine ID
- **channel** (*string*) – currently only "manual" is supported
- **script** (*string*) – The command(s) that the Server executes

get(*url*, *log_level='debug'*, *logger=True*, *headers={}*, *verbose=False*, *timeout=300*)
The client sends a get request to the Server. If the response status is != 200, raises a http Exception. If the response status is 200, returns the body of the return json.

Parameters **url** (*string*) – request url, which will get added to self.rest_url. For example, to call the route http://192.168.1.123:9000/machines/functions the url is "machines/functions".

get_config_id(*config_name*)
Returns the config_id of the config with the given name.

If it is not unique or no config with the given name is found, raises a ValueError. In this case, start the Browser based GUI AconitySTUDIO and copy the id from the URL and manually set the attribute config_id.

Saves the config_id into self.config_id. Saves the name of the operational config into self.config_name.

Returns Config ID

Return type string

get_job_id(*job_name*)
Get the job id for a given jobname. If the job_name is unique, sets and returns the attribute job_id. If it is not unique or no job with the given name is found, raises a ValueError. In this case, start the Browser based GUI AconitySTUDIO and copy the id from the URL and manually set the attribute machine_id.

Parameters **job_name** (*string*) – jobname

Returns Job ID

Return type string

get_lasers()

Returns a list with all lasers.

If no config_id is set, raises an AttributeError

get_lasers_off_cmds()

Returns the command to turn the laser off.

get_last_built_layer()

When a job is running, a websockets receives information about how many addLayerCommands have been executed. This information is used to calculate the current layer number by adding it to the starting layer which was specified when a job was started.

Returns current layer number during a job

Return type int

get_machine_id(*machine_name*)

Get the machine_id from a given Machine Name.

If no or multiple machines with the given name are given, raises ValueErrors. In this case, start the Browser based GUI AconitySTUDIO and copy the id from the URL and manually set the attribute machine_id.

If successfull, returns the machine_id and saves it to self.machine_id.

Parameters **machine_name** (*string*) – Name of Machine

Returns Machine ID

Return type string

get_session_id(*n=-1*)

Get all session ids. If successfull, saves the session ID in self.session_id

Parameters **n** (*int*) – With the default parameter *n=-1*, the most recent session id gets saved to self.session_id (second last session, use n=-2 etc).

Returns Session ID

Return type string

get_workunit_and_channel_id(*result=None*)

Retrieves workunit_id and channel_id. If successfull, saves them in self.channel_id and self.workunit_id and returns them

If not successfull, raises a ValueError.

Returns workunit_id, channel_id

Return type tuple

pause_job(*workunit_id=None*, *channel_id='run0'*)

Pauses the running script on the given channel and workunit

Parameters

- **workunit_id** (*string*) – the route GET /script yields information about the current workunit_id
- **channel** – the route GET /script yields information about the current workunit_id

post (*url*, *data=None*, *files=None*, *headers={}*, *timeout=300*)
The client sends a post request to the Server. If the response status is 200, returns the body of the return json, else a http exception is raised.

Parameters

- **url** (*string*) – request url, will get added to self.rest_url (for details see get())
- **data** (*dict*) – data to be posted

post_script (*init_script=''*, *execution_script=''*, *job_id=None*, *channel_id='run0'*, *file_path_init_script=None*, *file_path_execution_script=None*)
The client posts execution and init/resume scripts to the Server.

If the response status is != 200, raises Exception. Returns the body of the return json.

It is recommended that the API function *start_job* is used instead of this function, because *start_job* generates the init_script automatically.

Parameters

- **data** (*dict*) – data to be posted
- **job_id** (*string*) – job_id of the job
- **channel_id** (*string*) – channel_id of the job, for instance "run0".
- **execution_script** (*string*) – execution script
- **init_script** (*string*) – init script
- **file_path_execution_script** – If != None, gets interpreted as a filepath. The file will be read and any string given to parameter execution_script is ignored.
- **file_path_execution_script** – string
- **file_path_init_script** – If != None, gets interpreted as a filepath. The file will be read and any string given to parameter init_script is ignored.
- **file_path_init_script** – string

Returns Returns the body of the return json from the request.

Return type dict

put (*url*, *data=None*, *files=None*, *headers={}*)
The client sends a put request to the Server. If the response status is 200, returns the body of the return json, else a http exception is raised.

Parameters

- **url** (*string*) – request url, will get added to self.rest_url (for details see get())
- **data** (*dict*) – data to be posted

restart_config ()
The attribute "config_id" must be set. Restarts the config with that id.

If no `config_id` is set, raises a ValueError.

resume_job (*layers=None*, *parts='all'*, *workunit_id=None*, *channel_id='run0'*)
Resumes the running job on the given channel and workunit.

Parameters

- **init_resume_script** (*string*) – the init/resume script.

- **workunit_id** (*string*) – the route GET /script yields information about the current workunit_id
- **channel** – the route GET /script yields information about the current workunit_id

resume_script (*init_resume_script*, *workunit_id=None*, *channel_id='run0'*, *file_path_given=False*)

Resumes the running script on the given channel and workunit.

Parameters

- **init_resume_script** (*string*) – the init/resume script.
- **workunit_id** (*string*) – the route GET /script yields information about the current workunit_id. If workunit_id = None, the client attempts to use its own attribute workunit_id. If that fails, raises ValueError.
- **channel** – the route GET /script yields information about the current workunit_id

save_data_to_pymongo_db ()

Continually saves the output of the WebSocket Server by saving it into a Mongo database Call enable_pymongo_database() before calling this function

start_job (*layers*, *execution_script*, *job_id=None*, *channel_id='run0'*, *parts='all'*)

Starts a job. The init/resume script will be generated automatically from the current job.

Parameters

- **execution_script** (*string*) – The execution script which shall be executed.
- **job_id** (*string*) – Id of the Job. Get it by calling get_job_id().
- **channel_id** (*string*) – 'run0'.
- **layers** (*list*) – Specify the layers which shall be built. Must be given as list with 2 integer entries
- **parts** (*list/string*) – Specify the parts which shall be built. Can either be a list of integers or the string 'all'.

stop_channel (*channel='manual_move'*)

Stops the running execution on the given channel.

Parameters **channel** (*string*) – Example: 'manual_move'

stop_job (*workunit_id=None*, *channel='run0'*)

Stops the running script on the given channel and workunit

Parameters

- **workunit_id** (*string*) – the route GET /script yields information about the current workunit_id
- **channel** (*string*) – the route GET /script yields information about the current channel

subscribe_report (*name*)

Subscribes to reports via the WebServer.

To get information about the reports use the route configurations/{client.config_id}/topics).

Parameters **name** (*string*) – name of report, example reports: 'state', 'task'.

subscribe_topic (*name*)

Subscribes to reports via the WebServer.

To get information about the topics use the route configurations/{client.config_id}/topics).

Parameters **name** (*string*) – name of topic. Examples are 'State', 'Sensor','cmds' and 'Positioning'.

5.9 AconitySTUDIO_utils module

class AconitySTUDIO_utils.**JobHandler**(*job*, *logger*, *studio_version*)

Bases: object

The Python Client uses this Class to modify a job locally (so it can later be uploaded to the Server database). Additionally, it uses the locally saved job to create init and init_resume scripts The user of the Python Client never needs to use this class directly.

change_global_parameter(*param*, *new_value*, *check_boundaries=True*)
Function to change global build parameters

change_part_parameter(*part_id*, *param*, *new_value*, *laser='*'*, *check_boundaries=True*)
Function to build parameters for individual parts

convert_to_string(*data=None*)
Convert input data type into a sring.

create_addParts()
Returns the commands to add parts to the job

create_init_resume_script(*layers*, *parts='all'*)
Message to resume the execution of a script

create_init_script(*layers*, *parts='all'*)
Message to start the execution of a script

create_laser_beam_sources(*lasers*)
Creates and returns a dictionary containing the laser beam sources

create_preStartParams()
Returns the parameters set before the start of the build

create_preStartSelection(*layers*, *parts*)
Select the parts to be used

get_lasers()
Returns the available lasers

get_mapping_parts_to_index()
Returns the indices of the job parts

set(*job*)
Set the internal job variable to the input job

to_json()
Convert current job to json

AconitySTUDIO_utils.**customTime**(**args*)
Converts time to the local machine timezone.

AconitySTUDIO_utils.**filter_out_keys**(*data, allowed=['name', 'type', 'value']*)
Loop through input dictionary and only retain the 'name', 'type', 'value' keys

AconitySTUDIO_utils.**fix_ws_msg**(*msg*, *replace_value=-1*)

AconitySTUDIO_utils.**get_adress**(*args*)
Return the address of the machine

`AconitySTUDIO_utils.`**`get_time_string`**`(`*raw_time_stamp*, *format='%b %d %H:%M:%S'*`)`

Return a string with the current time

`AconitySTUDIO_utils.`**`log_setup`**`(`*filename*, *directory_path=''*`)`

Initialise the logging functionality

`AconitySTUDIO_utils.`**`track_layer_number`**`(`*client*, *msg*`)`

Update the current layer class attribute

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

A

C

D

E

F

G

I

J

L

M

O

P

R

S

T

U