AdaptiveMD Documentation

Release 0.2.1+128.g942fff8.dirty

Jan-Hendrik Prinz, Frank Noé

Apr 28, 2017

Contents

1 Documentation 1

2 Indices and tables 117

CHAPTER 1

Documentation

installation

adaptive-sampling

A Python framework to run adaptive Markov state model (MSM) simulation on HPC resources

The generation of MSMs requires a huge amount of trajectory data to be analyzed. In most cases this leads to an enhanced understanding of the dynamics of the system which can be used to make decision about collection more data to achieve a desired accuracy or level of detail in the generated MSM. This alternating process between simulation/actively generating new observations and analysis is currently difficult and involves lots of human decision along the path.

This framework aim to automate this process with the following goals:

1. Ease of use: Simple system setup once an HPC has been added.
2. Flexibility: Modular setup, attach to multiple HPCs and different simulation engines
3. Automatism: Create an user-defined adaptive strategy that is executed
4. Compatibility: Build analysis tools and export to known formats

Prerequisites

There are a few things we need to install to make this work.

MongoDB

AdaptiveMD and RP both need access to a MongoDB. The FU has one that Allegro can access in place and you can use this for storing projects. If you want to store these locally you need to install MongoDB.

Just download your OS installer from MongoDB Community Edition and follow the installation instructions. This is very straight forward and should work without any problems. You only need to install MongoDB on your local machine from which you will connect to the cluster. No need to install it on the cluster.

```
curl -O https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-debian81-3.4.2.tgz
tar -zxvf mongodb-linux-x86_64-debian81-3.4.2.tgz

mkdir -p ~/mongodb
cp -R -n mongodb-linux-x86_64-debian81-3.4.2/ ~/mongodb

# add PATH to .bashrc
echo "export PATH=~/mongodb/bin:$PATH" >> ~/.bash_rc

# create directory for storage (everywhere you have space)
mkdir -p ~/mongodb/data/db

# run the deamon in the background
mongod --quiet --dbpath ~/mongodb/data/db &
```

Conda

Whereever you will run the actual tasks (local or a cluster) you probably use some python so we recommend to install the common set of conda packages. If you are remotely executing python then you can even use python 3 without problems. The RPC might also work with python 3 but that needs to be tested.

If you have not yet installed conda please do so using

```
# curl -O https://repo.continuum.io/miniconda/Miniconda2-latest-Linux-x86_64.sh
# bash Miniconda2-latest-Linux-x86_64.sh
```

or in analogy for python3

Add 2 useful channels

```
conda config --append channels conda-forge
conda config --append channels omnia
```

and `--append` will make sure that the regular conda packages are tried first and use `conda-forge` and `omnia` as a fallback.

Install required and necessary packages now

```
# for adaptivemd only
conda install ujson pyyaml pymongo=2.8 numpy

# for openmm, pyemma etc
conda install pyemma openmm mdtraj
```

Install *adaptiveMD*

Let's get adaptivemd from the github repo now.

```
# clone and install adaptivemd
git clone git@github.com:markovmodel/adaptivemd.git

```

```
# go to adativemd
cd adaptivemd/

# and install it
python setup.py develop

# see if it works
python -c "import adaptivemd" || echo 'FAILED'

# run a simple test
cd adaptive/tests/
python test_simple.py
```

All of this must also be installed on the cluster, where you want to run your simulations.

For allegro I suggest to use a miniconda installation. Note that you only need these packages if you want to use some of it on the cluster like run openmm or make computations using pyemma. Just for running, say `acemd` conda is not required!

That's it. Have fun running adaptive simulations.

Documentation

To compile the doc pages, clone this github repository, go into the `docs` folder and do

```
conda install sphinx sphinx_rtd_theme pandoc
make html
```

The HTML pages are in _build/html. Please note that the docs can only be compiled if all the above mentionend AdaptiveMD dependencies are available. If you are using conda environments, this means that your AdaptiveMD environment should be active.

You might want to start with the examples in `examples/tutorials`

Examples Notebooks

Example 1 - Setup

First we cover some basics about adaptive sampling to get you going.

We will briefly talk about

1. resources
2. files
3. generators
4. how to run a simple trajectory

Imports

```
In [1]: import sys, os
```

Alright, let's load the package and pick the `Project` since we want to start a project

```
In [2]: from adaptivemd import Project
```

Let’s open a project with a UNIQUE name. This will be the name used in the DB so make sure it is new and not too short. Opening a project will always create a non-existing project and reopen an exising one. You cannot chose between opening types as you would with a file. This is a precaution to not accidentally delete your project.

```
In [3]: # Use this to completely remove the example-worker project from the database.
        Project.delete('tutorial')
```

```
In [4]: project = Project('tutorial')
```

Now we have a handle for our project. First thing is to set it up to work on a resource.

The `Resource`

What is a resource?

A `Resource` specifies a shared filesystem with one or more clusteres attached to it. This can be your local machine or just a regular cluster or even a group of cluster that can access the same FS (like Titan, Eos and Rhea do).

Once you have chosen your place to store your results it is set for the project and can (at least should) not be altered since all file references are made to match this resource.

Let us pick a local resource on your laptop or desktop machine for now. No cluster / HPC involved for now.

```
In [6]: from adaptivemd import LocalResource
```

We now create the Resource object

```
In [7]: resource = LocalResource()
```

Since this object defines the path where all files will be placed, let’s get the path to the shared folder. The one that can be accessed from all workers. On your local machine this is trivially the case.

```
In [8]: resource.shared_path
```

```
Out[8]: '$HOME/adaptivemd/'
```

Okay, files will be placed in `$HOME/adaptivemd/`. You can change this using an option when creating the `Resource`

```
LocalCluster(shared_path='$HOME/my/adaptive/folder/')
```

If you are interested in more information about `Resource` setup consult the documentation about `Resource`

Last, we save our configured `Resource` and initialize our empty prohect with it. This is done once for a project and should not be altered.

```
In [17]: project.initialize(resource)
```

Files

```
In [18]: from adaptivemd import File, Directory
```

First we define a `File` object. Instead of just a string, these are used to represent files anywhere, on the cluster or your local application. There are some subclasses or *extensions* of `File` that have additional meta information like `Trajectory` or `Frame`. The underlying base object of a `File` is called a `Location`.

We start with a first PDB file that is located on this machine at a relative path

```
In [21]: pdb_file = File('file://../files/alanine/alanine.pdb')
```

File like any complex object in adaptivemd can have a .name attribute that makes them easier to find later. You can either set the .name property after creation, or use a little helper method .named() to get a one-liner. This function will set .name and return itself.

For more information about the possibilities to specify filelocation consult the documentation for File

```
In [ ]: pdb_file.name = 'initial_pdb'
```

The .load() at the end is important. It causes the File object to load the content of the file and if you save the File object, the actual file is stored with it. This way it can simply be rewritten on the cluster or anywhere else.

```
In [ ]: pdb_file.load()
```

Generators

TaskGenerators are instances whose purpose is to create tasks to be executed. This is similar to the way Kernels work. A TaskGenerator will generate Task objects for you which will be translated into a ComputeUnitDescription and executed. In simple terms:

The task generator creates the bash scripts for you that run a simulation or run pyemma.

A task generator will be initialized with all parameters needed to make it work and it will now what needs to be staged to be used.

```
In [48]: from adaptivemd.engine.openmm import OpenMMEngine
```

A task generator that will create jobs to run simulations. Currently it uses a little python script that will excute OpenMM. It requires conda to be added to the PATH variable or at least openmm to be installed on the cluster. If you setup your resource correctly then this should all happen automatically.

So let's do an example for an OpenMM engine. This is simply a small python script that makes OpenMM look like a executable. It run a simulation by providing an initial frame, OpenMM specific system.xml and integrator.xml files and some additional parameters like the platform name, how often to store simulation frames, etc.

```
In [49]: engine = OpenMMEngine(
             pdb_file=pdb_file,
             system_file=File('file://../files/alanine/system.xml').load(),
             integrator_file=File('file://../files/alanine/integrator.xml').load(),
             args='-r --report-interval 1 -p CPU'
         ).named('openmm')
```

We have now an OpenMMEngine which uses the previously made pdb File object and uses the location defined in there. The same for the OpenMM XML files and some args to run using the CPU kernel, etc.

Last we name the engine openmm to find it later.

```
In [50]: engine.name
```

```
Out[50]: 'openmm'
```

Next, we need to set the output types we want the engine to generate. We chose a stride of 10 for the master trajectory without selection and a second trajectory with only protein atoms and native stride.

Note that the stride and all frame number ALWAYS refer to the native steps used in the engine. In out example the engine uses 2fs time steps. So master stores every 20fs and protein every 2fs

```
In [51]: engine.add_output_type('master', 'master.dcd', stride=10)
         engine.add_output_type('protein', 'protein.dcd', stride=1, selection='protein')
```

```
In [52]: from adaptivemd.analysis.pyemma import PyEMMAAnalysis
```

The instance to compute an MSM model of existing trajectories that you pass it. It is initialized with a .pdb file that is used to create features between the c_α atoms. This implementaton requires a PDB but in general this is not necessay. It is specific to my PyEMMAAnalysis show case.

```
In [53]: modeller = PyEMMAAnalysis(
             engine=engine,
             outtype='protein',
             features={'add_inverse_distances': {'select_Backbone': None}}
         ).named('pyemma')
```

Again we name it `pyemma` for later reference.

The other two option chose which output type from the engine we want to analyse. We chose the protein trajectories since these are faster to load and have better time resolution.

The features dict expresses which features to use. In our case use all inverse distances between backbone c_alpha atoms.

Next step is to add these to the project for later usage. We pick the `.generators` store and just add it. Consider a store to work like a `set()` in python. It contains objects only once and is not ordered. Therefore we need a name to find the objects later. Of course you can always iterate over all objects, but the order is not given.

To be precise there is an order in the time of creation of the object, but it is only accurate to seconds and it really is the time it was created and not stored.

```
In [54]: project.generators.add(engine)
         project.generators.add(modeller)
```

Note, that you cannot add the same engine twice. But if you create a new engine it will be considered different and hence you can store it again.

Create one initial trajectory

Finally we are ready to run a first trajectory that we will store as a point of reference in the project. Also it is nice to see how it works in general.

We are using a *Worker* approach. This means simply that someone (in our case the user from inside a script or a notebook) creates a list of tasks to be done and some other instance (the worker) will actually do the work.

First we create the parameters for the engine to run the simulation. Since it seemed appropriate we use a `Trajectory` object (a special `File` with initial frame and length) as the input. You could of course pass these things separately, but this way, we can actualy reference the no yet existing trajectory and do stuff with it.

A Trajectory should have a unique name and so there is a project function to get you one. It uses numbers and makes sure that this number has not been used yet in the project.

```
In [56]: trajectory = project.new_trajectory(engine['pdb_file'], 100, engine)
         trajectory
```

```
Out[56]: Trajectory('alanine.pdb' >> [0..100])
```

This says, initial is `alanine.pdb` run for 100 frames and is named `xxxxxxxx.dcd`.

You might wonder why a `Trajectory` object is necessary. You could just build a function that will take these parameters and run a simulation. At the end it will return the trajectory object. The same object we created just now.

The main reason is to familiarize you with the general concept of asyncronous execution and so-called *Promises*. The trajectory object we built is similar to a *Promise* so what is that exactly?

A *Promise* is a value (or an object) that represents the result of a function at some point in the future. In our case it represents a trajectory at some point in the future. Normal promises have specific functions do deal with the unknown result, for us this is a little different but the general concept stands. We create an object that represents the specifications of a `Trajectory` and so, regardless of the existence, we can use the trajectory as if it would exists:

Get the length

```
In [61]: print trajectory.length
```

```
100
```

and since the length is fixed, we know how many frames there are and can access them

```
In [64]: print trajectory[20]
```

```
Frame(sandbox:////00000001/[20])
```

ask for a way to extend the trajectory

```
In [65]: print trajectory.extend(100)
```

```
<adaptivemd.engine.engine.TrajectoryExtensionTask object at 0x110e6e210>
```

ask for a way to run the trajectory

```
In [66]: print trajectory.run()
```

```
<adaptivemd.engine.engine.TrajectoryGenerationTask object at 0x110dd46d0>
```

We can ask to extend it, we can save it. We can reference specific frames in it before running a simulation. You could even build a whole set of related simulations this way without running a single frame. You might understand that this is pretty powerful especially in the context of running asynchronous simulations.

Last, we did not answer why we have two separate steps: Create the trajectory first and then a task from it. The main reason is educational: **> It needs to be clear that a ‘‘Trajectory‘‘ *can exist* before running some engine or creating a task for it. The ‘‘Trajectory‘‘ *is not* a result of a simulation action.**

Now, we want that this trajectory actually exists so we have to make it. This requires a `Task` object that *knows* to describe a simulation. Since `Task` objects are very flexible and can be complex there are helper functions (i.e. factories) to get these in an easy manner, like the ones we already created just before. Let's use the openmm engine to create an openmm task now.

```
In [57]: task = engine.run(trajectory)
```

As an alternative you can directly use the trajectory (which knows its engine) and call `.run()`

```
In [58]: task = trajectory.run()
```

That's it, just take a trajectory description and turn it into a task that contains the shell commands and needed files, etc.

Finally we need to add this task to the things we want to be done. This is easy and only requires saving the task to the project. This is done to the `project.tasks` bundle and once it has been stored it can be picked up by any worker to execute it.

```
In [32]: project.queue(task)  # shortcut for project.tasks.add(task)
```

That is all we can do from here. To execute the tasks you need to run a worker using

```
adaptivemdworker -l tutorial --verbose
```

Once this is done, come back here and check your results. If you want you can execute the next cell which will block until the task has been completed.

```
In [33]: print project.files
         print project.trajectories
```

```
<StoredBundle for with 6 file(s) @ 0x111fa1150>
<ViewBundle for with 0 file(s) @ 0x111fa1450>
```

and close the project.

```
In [27]: project.close()
```

The final project.close() will close the DB connection.

```
In [ ]:
```

Example 2 - Run

Example 2 - The Tasks

Imports

```
In [1]: import sys, os
```

```
In [2]: from adaptivemd import Project, Event, FunctionalEvent, Trajectory
```

Let's open our `test` project by its name. If you completed the previous example this should all work out of the box.

```
In [3]: project = Project('tutorial')
```

Open all connections to the `MongoDB` and `Session` so we can get started.

Let's see where we are. These numbers will depend on whether you run this notebook for the first time or just continue again. Unless you delete your project it will accumulate models and files over time, as is our ultimate goal.

```
In [4]: print project.tasks

        print project.trajectories
        print project.models
```

```
<StoredBundle for with 1 file(s) @ 0x10949f5d0>
<ViewBundle for with 0 file(s) @ 0x10949f710>
<StoredBundle for with 0 file(s) @ 0x10949f510>
```

Now restore our old ways to generate tasks by loading the previously used generators.

```
In [5]: engine = project.generators['openmm']
        modeller = project.generators['pyemma']
        pdb_file = project.files['initial_pdb']
```

Remember that we stored some files in the database and of course you can look at them again, should that be important.

```
In [7]: print pdb_file.get_file()[:1000] + ' [...]'
```

```
REMARK   1 CREATED WITH MDTraj 1.8.0, 2016-12-22
CRYST1   26.063   26.063   26.063  90.00  90.00  90.00 P 1           1
MODEL        0
ATOM      1  H1  ACE A   1      -1.900   1.555  26.235  1.00  0.00           H
ATOM      2  CH3 ACE A   1      -1.101   2.011  25.651  1.00  0.00           C
ATOM      3  H2  ACE A   1      -0.850   2.954  26.137  1.00  0.00           H
ATOM      4  H3  ACE A   1      -1.365   2.132  24.600  1.00  0.00           H
ATOM      5  C   ACE A   1       0.182   1.186  25.767  1.00  0.00           C
ATOM      6  O   ACE A   1       1.089   1.407  26.645  1.00  0.00           O
ATOM      7  N   ALA A   2       0.302   0.256  24.807  1.00  0.00           N
ATOM      8  H   ALA A   2      -0.588   0.102  24.354  1.00  0.00           H
ATOM      9  CA  ALA A   2       1.498  -0.651  24.567  1.00  0.00           C
ATOM     10  HA  ALA A   2       1.810  -0.944  25.570  1.00  0.00           H
ATOM     11  CB  ALA A   2       1.054  -1.959  23.852 [...]
```

The `Trajectory` object

Before we talk about adaptivity, let's have a look at possibilities to generate trajectories.

We assume that you successfully ran a first trajectory using a worker. Next, we talk about lot's of ways to generate new trajectories.

Trajectories from a pdb

You will do this in the beginning. Remember we already have a PDB stored from setting up the engine. if you want to start from this configuration do as before

1. create the `Trajectory` object you want
2. make a task
3. submit the task to craft the object into existance on the HPC

A trajectory contains all necessary information to make itself. It has

1. a (hopefully unique) location: This will we the folder where all the files that belong to the trajectory go.
2. an initial frame: the initial configuration to be used to tell the MD simulation package where to start
3. a length in frames to run
4. the `Engine`: the actual engine I want to use to create the trajectory.

Note, the `Engine` is technically not required unless you want to use `.run()` but it makes sense, because the engine contains information about the topology and, more importantly information about which output files are generated. This is the essential information you will need for analysis, e.g. what is the filename of the trajectory file that contains the protein structure and what is its stride?

Let's first build a `Trajectory` from scratch

```
In [8]: file_name = next(project.traj_name)              # get a unique new filename

        trajectory = Trajectory(
            location=file_name,                          # this creates a new filename
            frame=pdb_file,                              # initial frame is the PDB
            length=100,                                  # length is 100 frames
            engine=engine                                # the engine to be used
        )
```

Since this is tedious to write there is a shortcut

```
In [9]: trajectory = project.new_trajectory(
            frame=pdb_file,
            length=100,
            engine=engine,
            number=1          # if more then one you get a list of trajectories
        )
```

Like in the first example, now that we have the parameters of the `Trajectory` we can create the task to do that.

The `Task` object

First, an example

```
In [13]: task_run = engine.run(trajectory)
```

This was easy, but we can do some interesting stuff. Since we know the trajectory will exist now we can also extend by some frames. Remember, the trajectory does not really exist yet (not until we submit it and a worker executes it), but we can pretend that it does, since it's relevant propertier are set.

```
In [14]: task_extend = engine.extend(trajectory, 50)
```

The only problem is to make sure the tasks are run in the correct order. This would not be a problem if the worker will run tasks in the order they are place in the queue, but that defeats the purpose of parallel runs. Therefore an extended tasks knows that is depends on the existance of the source trajectory. The worker will hence only run a trajectory, once the source exists.

A queueing system ?

We might wonder at this point how we manage to construct the dependency graph between all tasks and how this is handled and optimized, etc...

Well, we don't. There is no dependency graph, at least not explicitely. All we do, is to check at a time among all task that *should* be run, which of there *can* be run. And this is easy to check, all dependent tasks need to be completed and must have succeeded. Then we can rely on their (the dependencies) results to exists and it makes sense to continue.

A real dependency graph would go even further and know about all future relations and you could identify bottleneck tasks which are necessary to allow other tasks to be run. We don't do that (yet). It could improve performance in the sense that you will run at optimal load balance and keep all workers as busy as possible. Consider our a attempt a first order dependency graph.

```
In [15]: project.queue(task_run, task_extend)
```

A not on simulation length

Remember that we allow an engine to output multiple trajectory types with freely chosen strides? This could leave to trouble. Imagine this (unrealistic) situation:

We have 1. full trajectory with `stride=10` 2. a reduced protein-only trajectory with `stride=7`

Now run a trajectory of `length=300`. We get

1. 30+1 full (+1 for the initial frame) and
2. 42+1 protein frames

That per se is no problem, but if you want to extend we only have a restart file for the very last frame and while this works for the full trajectory, for the protein trajectory you are 6 frames short. Just continuing and concatenating the two leads to a gap of 6+7=13 frames instead of 7. A small big potentially significant source of error.

So, compute the least common multiple of all strides using

```
In [16]: engine.native_stride
```

```
Out[16]: 10
```

simpler function calls

There is also a shorter way of writing this

```
In [17]: # task = trajectory.run().extend(50)
```

This will create two tasks that first runs the trajectory and then extend it by 50 frames (in native engine frames)

If you want to do that several times, you can pass a list of ints which is a shortcut for calling `.extend(l1).extend(l2). ...`

```
In [18]: # task = trajectory.run().extend([10] * 10)
```

This will create 10! tasks that eacht will extend the previous one. Each of the task requires the previous one to finish, this way the dependency is preserved. You can use this to mimick using several restarts in between and it also means that you have no idea which worker will actually start and which worker will continue or finish a trajectory.

Checking the results

For a seconds let's see if everything went fine.

```
In [60]: for t in project.trajectories:
             print t.short, t.length
```

```
sandbox:////00000000/ 150
sandbox:////00000003/ 100
sandbox:////00000005/ 100
sandbox:////00000006/ 100
sandbox:////00000007/ 100
sandbox:////00000008/ 100
sandbox:////00000009/ 100
```

If this works, then you should see one 100 frame trajectory from the setup (first example) and a second 150 length trajectory that we just generated by running 100 frames and extending it by another 50.

If not, there might be a problem or (more likely) the tasks are not finished yet. Just try the above cell again and see if it changes to the expected output.

`project.trajectories` will show you *only* existing trajectories. Not ones, that are planned or have been extended. If you want to see all the ones already in the project, you can look at `project.files`. Which is a bundle and bundles can be filtered. But first all files

```
In [63]: for f in project.files:
             print f
```

```
file:///Users/jan-hendrikprinz/Studium/git/adaptivemd/adaptivemd/scripts/_run_.py
file:///Users/jan-hendrikprinz/Studium/git/adaptivemd/adaptivemd/engine/openmm/openmmrun.py
file:///Users/jan-hendrikprinz/Studium/git/adaptivemd/examples/files/alanine/alanine.pdb
file:///Users/jan-hendrikprinz/Studium/git/adaptivemd/examples/files/alanine/system.xml
file:///Users/jan-hendrikprinz/Studium/git/adaptivemd/examples/files/alanine/integrator.xml
sandbox:///projects/tutorial/trajs/00000000/
sandbox:///projects/tutorial/trajs/00000000/
sandbox:///projects/tutorial/trajs/00000001/
sandbox:///projects/tutorial/trajs/00000002/
sandbox:///projects/tutorial/trajs/00000003/
sandbox:///projects/tutorial/trajs/00000005/
file:///Users/jan-hendrikprinz/Studium/git/adaptivemd/examples/tutorial/_rpc_input_0x43a0c8dc148311e7
file:///Users/jan-hendrikprinz/Studium/git/adaptivemd/adaptivemd/scripts/_run_.py
file:///Users/jan-hendrikprinz/Studium/git/adaptivemd/examples/tutorial/_rpc_output_0x43a0c8dc148311e
project:///models/model.0x43a0c8dc148311e7acff0000000001a0L.json
sandbox:///projects/tutorial/trajs/00000006/
sandbox:///projects/tutorial/trajs/00000007/
sandbox:///projects/tutorial/trajs/00000008/
sandbox:///projects/tutorial/trajs/00000009/
```

Now all files filtered by [c]lass `Trajectory`. `DT` is a little helper to convert time stamps into something readable.

```
In [66]: from adaptivemd import DT
```

```
In [75]: for t in project.files.c(Trajectory):
             print t.short, t.length,
             if t.created:
                 if t.created > 0:
                     print 'created  @ %s' % DT(t.created)
                 else:
                     print 'modified @ %s' % DT(-t.created)
             else:
                 print 'not existent'
```

```
sandbox:////00000000/ 100 modified @ 2017-03-29 15:57:38
sandbox:////00000000/ 150 created  @ 2017-03-29 15:57:38
sandbox:////00000001/ 100 not existent
sandbox:////00000002/ 100 not existent
sandbox:////00000003/ 100 created  @ 2017-03-29 15:58:55
sandbox:////00000005/ 100 created  @ 2017-03-29 15:59:59
sandbox:////00000006/ 100 created  @ 2017-03-29 16:01:27
sandbox:////00000007/ 100 created  @ 2017-03-29 16:01:33
sandbox:////00000008/ 100 created  @ 2017-03-29 16:01:39
sandbox:////00000009/ 100 created  @ 2017-03-29 16:01:45
```

You see, that the extended trajecory appears twice once with length 100 and once with length 150. This is correct, because at the idea of a 100 frame trajectory was used and hence is saved. But why does this one not appear in the list of trajectories. It was created first and had a timestamp of creation written to `.created`. This is the time when the worker finishes and was successful.

At the same time, all files that are overwritten, are marked as modified by setting a negative timestamp. So if

1. `.created is None`, the file does not exist nor has it.
2. `.created > 0`, the file exists
3. `.created < 0`, the file existed but has been overwritten

Finally, all `project.trajectories` are files of type `Trajectory` with positive `created` index.

Dealing with errors

Let's do something stupid and produce an error by using a wrong initial pdb file.

```
In [26]: trajectory = project.new_trajectory(engine['system_file'], 100)
         task = engine.run(trajectory)
         project.queue(task)
```

Well, nothing changed obviously and we expect it to fail. So let's inspect what happened.

```
In [32]: task.state
```

```
Out[32]: u'fail'
```

You might need to execute this cell several times. It will first become `queued`, then `running` and finally `fail` and stop there.

It failed, well, we kind of knew that. No suprise here, but why? Let's look at the stdout and stderr

```
In [33]: print task.stdout
```

```
15:58:14 [worker:3] stdout from running task
GO...
Reading PDB
```

```
In [34]: print task.stderr
```

```
15:58:14 [worker:3] stderr from running task
Traceback (most recent call last):
  File "openmmrun.py", line 169, in <module>
    pdb = PDBFile(args.topology_pdb)
  File "/Users/jan-hendrikprinz/anaconda/lib/python2.7/site-packages/simtk/openmm/app/pdbfile.py", li
    self.positions = self._positions[0]
IndexError: list index out of range
```

We see, what we expect. In `openmmrun.py` the openmm executable it could not load the pdb file.

> *NOTE* If your worker dies for some reason, it will not set a STDOUT or STDERR. If you think that your task should be able to execute, then you can do `task.state = 'created'` and reset it to be accessible to workers. This is NOT recommended, just to explain how this works. Of course you need a new worker anyway.

What else

If you have a `Trajectory` object and create the real trajectory file, you can also put the `Trajectory` directly into the queue. This is equivalent to call `.run` on the trajectory and submit the resulting `Task` to the queue. The only downside is that you do not see the task object and cannot directly work with it, check it's status, etc...

```
In [76]: # project.queue(project.new_trajectory(pdb_file, 100, engine).run()) can be called as
         project.queue(project.new_trajectory(pdb_file, 100, engine))
```

Trajectories from other trajectories

This will be the most common case. At least in any remote kind of adaptivity you will not start always from the same position or extend. You want to pick any exisiting frame and continue from there. So, let's do that.

First we get a trajectory. Every `Bundle` in the project (e.g. `.trajectories`, `.models`, `.files`, `.tasks`) acts like an enhanced set. You can iterate over all entries as we did before, and you can get one element, which usually is the first stored, but not always. If you are interested in `Bundles` see the documentation. For now that is enough to know, that a bundle (as a set) has a `.one` function which is short for getting the first object if you iterate. As if you would call `next(project.trajectories)`. Note, that the iterator does not ensure a fixed order. You literally might get any object, if there is at least one.

```
In [36]: trajectory = project.trajectories.one
```

Good, at least 100 frames. We pick, say, frame at index 28 (which is the 29th frame, we start counting at zero) using the way you pick an element from a python list (which is almost what a `Trajectory` represents, a list of frames)

```
In [38]: frame = trajectory[28]
         print frame, frame.exists
```

```
Frame(sandbox:////00000000/[28]) False
```

```
In [39]: frame = trajectory[30]
         print frame, frame.exists
```

```
Frame(sandbox:////00000000/[30]) True
```

This part is important! We are running only one full atom trajectory with stride larger than one, so if we want to pick a frame from this trajectory you can pick in theory every frame, but only some of these really exist. If you want to restart from a frame this needs to be the case. Otherwise you run into trouble.

To run a trajectory just use the frame as the initial frame.

```
In [40]: frame = trajectory[28]
         task = project.new_trajectory(frame, 100, engine).run()
         print task
```

```
None
```

```
In [41]: frame = trajectory[30]
         task = project.new_trajectory(frame, 100, engine).run()
         print task
```

```
<adaptivemd.engine.engine.TrajectoryGenerationTask object at 0x10360f4d0>
```

```
In [42]: print task.description
```

```
Task: TrajectoryGenerationTask(OpenMMEngine) [created]

Sources
- staging:///integrator.xml
- staging:///system.xml
- staging:///alanine.pdb
- staging:///openmmrun.py
- sandbox:////00000000/ [exists]
Targets
- sandbox:////00000005/
Modified

<pretask>
Link('staging:///alanine.pdb' > 'worker://initial.pdb)
Link('staging:///system.xml' > 'worker://system.xml)
Link('staging:///integrator.xml' > 'worker://integrator.xml)
Link('staging:///openmmrun.py' > 'worker://openmmrun.py)
Link('sandbox:////00000000/' > 'worker://source/)
mdconvert -o worker://input.pdb -i 3 -t worker://initial.pdb worker://source/master.dcd
Touch('worker://traj/')
python openmmrun.py -r --report-interval 1 -p CPU --types="'protein':'stride':1,'selection':'protein
Move('worker://traj/' > 'sandbox:////00000005/)
<posttask>
```

See, how the actual frame picked in the `mdconvert` line is `-i 3` meaning index 3 which represents frame 30 with stride 10.

Now, run the task.

```
In [43]: project.queue(task)
```

Btw, you can wait until something happens using `project.wait_until(condition)`. This is not so useful in notebooks, but in scripts it does. `condition` here is a function that evaluates to `True` or `False`. it will be tested in regular intervals and once it is `True` the function returns.

```
In [44]: project.wait_until(task.is_done)
```

```
In [45]: task.state
```

```
Out[45]: u'success'
```

Each `Task` has a function `is_done` that you can use. It will return once a task is done. That means it either failed or succeeded or was cancelled. Basically when it is not queued anymore.

If you want to run adaptively, *all you need to do* is to figure out where to start new simulations from and use the methods provided to run these.

`Model` tasks

There are of course other things you can do besides creating new trajectories

```
In [46]: from adaptivemd.analysis.pyemma import PyEMMAAnalysis
```

The instance to compute an MSM model of existing trajectories that you pass it. It is initialized with a `.pdb` file that is used to create features between the c_α atoms. This implementaton requires a PDB but in general this is not necessay. It is specific to my PyEMMAAnalysis show case.

```
In [47]: modeller = PyEMMAAnalysis(
             engine=engine,
             outtype='protein',
             features={'add_inverse_distances': {'select_Backbone': None}}
         ).named('pyemma')
```

Again we name it pyemma for later reference.

The other two option chose which output type from the engine we want to analyse. We chose the protein trajectories since these are faster to load and have better time resolution.

The features dict expresses which features to use. In our case use all inverse distances between backbone c_alpha atoms.

A model generating task work similar to trajectories. You create the generator with options (so far, this will become more complex in the future) and then you create a Task from passing it a list of trajectories to be analyzed.

```
In [48]: task = modeller.execute(list(project.trajectories))
         project.queue(task)
```

```
In [49]: project.wait_until(task.is_done)
```

```
In [52]: for m in project.models:
             print m
```

```
<adaptivemd.model.Model object at 0x1036bff50>
```

So we generated one model. The Model objects contain (in the base version) only a .data attribute which is a dictionary of information about the generated model.

```
In [53]: model = project.models.last
```

```
In [54]: print model['msm']['P']
```

```
[[ 0.84615385  0.          0.          0.07701397  0.07683217]
 [ 0.          0.94936708  0.02307278  0.02756013  0.        ]
 [ 0.          0.02591964  0.94047619  0.00963989  0.02396427]
 [ 0.01328607  0.05096999  0.01586998  0.89333333  0.02654062]
 [ 0.01433636  0.          0.04267144  0.02870648  0.91428572]]
```

Pick frames automatically

The last thing that is implemented is a function that can utilize models to decide which frames are better to start from. The simplest one will use the counts per state, take the inverse and use this as a distribution.

```
In [55]: project.find_ml_next_frame(4)
```

```
Out[55]: [Frame(sandbox:///{}/00000003/[40]),
          Frame(sandbox:///{}/00000003/[20]),
          Frame(sandbox:///{}/00000005/[50]),
          Frame(sandbox:///{}/00000003/[20])]
```

So you can pick states according to the newest (last) model. (This will be moved to the Brain). And since we want trajectories with these frames as starting points there is also a function for that

```
In [56]: trajectories = project.new_ml_trajectory(length=100, number=4, engine=engine)
         trajectories
```

```
Out[56]: [Trajectory(Frame(sandbox:///{}/00000000/[10]) >> [0..100]),
          Trajectory(Frame(sandbox:///{}/00000003/[50]) >> [0..100]),
          Trajectory(Frame(sandbox:///{}/00000000/[10]) >> [0..100]),
          Trajectory(Frame(sandbox:///{}/00000003/[20]) >> [0..100])]
```

Let’s submit these before we finish this notebook with a quick discussion of workers

```
In [57]: project.queue(trajectories)
```

That’s it.

The `Worker` objects

Worker are the instances that execute tasks for you. If you did not stop the worker in the command line it will still be running and you can check its state

```
In [59]: project.trigger()
         for w in project.workers:
             if w.state == 'running':
                 print '[%s:%s] %s:%s' % (w.state, DT(w.seen).time, w.hostname, w.cwd)
```

```
[running:16:01:47] Stevie.fritz.box:/Users/jan-hendrikprinz/Studium/git/adaptivemd
```

Okay, the worker is running, was last reporting its heartbeat at ... and has a hostname and current working directory (where it was executed from). The generators specify which tasks from some generators are executed. If it is `None` then the worker runs all tasks it finds. You can use this to run specific workers for models and some for trajectory generation.

You can also control it remotely by sending it a command. `shutdown` will shut it down for you.

```
In [77]: # project.workers.last.command = 'shutdown'
```

Afterwards you need to restart you worker to continue with this examples.

If you want to control `Worker` objects look at the documentation.

```
In [101]: project.close()
```

```
In [ ]:
```

Example 3 - Adaptive

AdaptiveMD

Example 3 - Running an adaptive loop

0. Imports

```
In [1]: import sys, os
```

```
In [2]: from adaptivemd import (
            Project,
            Event, FunctionalEvent,
            File
        )

        # We need this to be part of the imports. You can only restore known objects
        # Once these are imported you can load these objects.
        from adaptivemd.engine.openmm import OpenMMEngine
        from adaptivemd.analysis.pyemma import PyEMMAAnalysis
```

Let's open our `test` project by its name. If you completed the first examples this should all work out of the box.

```
In [3]: project = Project('tutorial')
```

Open all connections to the `MongoDB` and `Session` so we can get started.

> An interesting thing to note here is, that since we use a DB in the back, data is synced between notebooks. If you want to see how this works, just run some tasks in the last example, go back here and check on the change of the contents of the project.

Let's see where we are. These numbers will depend on whether you run this notebook for the first time or just continue again. Unless you delete your project it will accumulate models and files over time, as is our ultimate goal.

```
In [4]: print project.files
        print project.generators
        print project.models
```

```
<StoredBundle for with 222 file(s) @ 0x11443ab50>
<StoredBundle for with 2 file(s) @ 0x11443ab10>
<StoredBundle for with 34 file(s) @ 0x11443aad0>
```

Now restore our old ways to generate tasks by loading the previously used generators.

```
In [5]: engine = project.generators['openmm']
        modeller = project.generators['pyemma']
        pdb_file = project.files['initial_pdb']
```

Run simulations

You are free to conduct your simulations from a notebook but normally you will use a script. The main point about adaptivity is to make decision about tasks along the way.

To make this happen we need `Conditions` which are functions that evaluate to `True` or `False` and once they are `True` they cannot change anymore back to `False`. Like a one time on switch.

These are used to describe the happening of an event. We will now deal with some types of events.

Functional Events

We want to first look into a way to run python code asynchroneously in the project. For this, we write a function that should be executed. Inside you will create tasks and submit them.

If the function should pause, write `yield {condition_to_continue}`. This will interrupt your script until the function you return will return `True` when called. An example

```
In [6]: def strategy(loops=10, trajs_per_loop=4, length=100):
            for loop in range(loops):
                # submit some trajectory tasks
                trajectories = project.new_ml_trajectory(length, trajs_per_loop)
                tasks = map(engine.task_run_trajectory, trajectories)
                project.queue(tasks)

                # continue if ALL of the tasks are done (can be failed)
                yield [task.is_done for task in tasks]

                # submit a model job
                task = modeller.execute(list(project.trajectories))
                project.queue(task)

                # when it is done do next loop
                yield task.is_done
```

and add the event to the project (these cannot be stored yet!)

```
In [7]: project.add_event(strategy(loops=2))
```

```
Out[7]: <adaptivemd.event.FunctionalEvent at 0x10d615050>
```

What is missing now? The adding of the event triggered the first part of the code. But to recheck if we should continue needs to be done manually.

RP has threads in the background and these can call the trigger whenever something changed or finished.

Still that is no problem, we can do that easily and watch what is happening

Let's see how our project is growing. TODO: Add threading.Timer to auto trigger.

```
In [8]: import time
        from IPython.display import clear_output
```

```
In [ ]: try:
            while project._events:
                clear_output(wait=True)
                print '# of files  %8d : %s' % (len(project.trajectories), '#' * len(project.traject
                print '# of models %8d : %s' % (len(project.models), '#' * len(project.models))
                sys.stdout.flush()
                time.sleep(2)
                project.trigger()

        except KeyboardInterrupt:
            pass
```

```
# of files        74 : ##########################################################################
# of models       33 : #################################
```

Let's do another round with more loops

```
In [10]: project.add_event(strategy(loops=2))
```

```
Out[10]: <adaptivemd.event.FunctionalEvent at 0x10d633850>
```

And some analysis (might have better functions for that)

```
In [11]: # find, which frames from which trajectories have been chosen
         trajs = project.trajectories
         q = {}
         ins = {}
         for f in trajs:
             source = f.frame if isinstance(f.frame, File) else f.frame.trajectory
             ind = 0 if isinstance(f.frame, File) else f.frame.index
             ins[source] = ins.get(source, []) + [ind]

         for a,b in ins.iteritems():
             print a.short, ':', b
```

```
file:///alanine.pdb : [0, 0, 0]
sandbox:////00000005/ : [95, 92, 67, 92]
sandbox:////00000007/ : [11]
sandbox:////00000011/ : [55]
sandbox:////00000000/ : [28, 89, 72]
sandbox:////00000002/ : [106]
sandbox:////00000004/ : [31, 25, 60]
```

Event

And do this with multiple events in parallel.

```
In [12]: def strategy2():
             for loop in range(10):
                 num = len(project.trajectories)
                 task = modeller.execute(list(project.trajectories))
                 project.queue(task)
                 yield task.is_done
```

```
                # continue only when there are at least 2 more trajectories
                yield project.on_ntraj(num + 2)
```

```
In [13]: project.add_event(strategy(loops=10, trajs_per_loop=2))
         project.add_event(strategy2())
```

```
Out[13]: <adaptivemd.event.FunctionalEvent at 0x107744c90>
```

And now wait until all events are finished.

```
In [6]: project.wait_until(project.events_done)
```

See, that we again reused our strategy.

```
In [18]: project.close()
```

Example 4 - Tasks

AdaptiveMD

Example 4 - Custom `Task` objects

0. Imports

```
In [1]: import sys, os
```

```
In [2]: from adaptivemd import (
            Project, Task, File, PythonTask
        )
```

Let's open our `test` project by its name. If you completed the first examples this should all work out of the box.

```
In [3]: project = Project('tutorial')
```

Open all connections to the `MongoDB` and `Session` so we can get started.

Let's see again where we are. These numbers will depend on whether you run this notebook for the first time or just continue again. Unless you delete your project it will accumulate models and files over time, as is our ultimate goal.

```
In [4]: print project.files
        print project.generators
        print project.models
```

```
<StoredBundle for with 222 file(s) @ 0x10b9ada10>
<StoredBundle for with 2 file(s) @ 0x10b9ad9d0>
<StoredBundle for with 34 file(s) @ 0x10b9ad990>
```

Now restore our old ways to generate tasks by loading the previously used generators.

```
In [5]: engine = project.generators['openmm']
        modeller = project.generators['pyemma']
        pdb_file = project.files['initial_pdb']
```

A simple task

A task is in essence a bash script-like description of what should be executed by the worker. It has details about files to be linked to the working directory, bash commands to be executed and some meta information about what should happen in case we succeed or fail.

Let's first explain briefly how a task is executed and what its components are. This was originally build so that it is compatible with radical.pilot and still is. So, if you are familiar with it, all of the following information should sould very familiar.

A task is executed from within a unique directory that only exists for this particular task. These are located in `adaptivemd/workers/` and look like

```
worker.0x5dcccd05097611e7829b000000000072L/
```

the long number is a hex representation of the UUID of the task. Just if you are curious type

```
print hex(my_task.__uuid__)
```

Then we change directory to this folder write a `running.sh` bash script and execute it. This script is created from the task definition and also depends on your resource setting (which basically only contain the path to the workers directory, etc)

The script is divided into 1 or 3 parts depending on which `Task` class you use. The main `Task` uses a single list of commands, while `PrePostTask` has the following structure

1. **Pre-Exec**: Things to happen before the main command (optional)
2. **Main**: the main commands are executed
3. **Post-Exec**: Things to happen after the main command (optional)

Okay, lots of theory, now some real code for running a task that generated a trajectory

```
In [6]: task = engine.task_run_trajectory(project.new_trajectory(pdb_file, 100))
```

```
In [7]: task.script
```

```
Out[7]: [Link('staging:///alanine.pdb' > 'worker://initial.pdb),
         Link('staging:///system.xml' > 'worker://system.xml),
         Link('staging:///integrator.xml' > 'worker://integrator.xml),
         Link('staging:///openmmrun.py' > 'worker://openmmrun.py),
         Touch('worker://traj/'),
         'python openmmrun.py -r --report-interval 1 -p CPU --store-interval 1 -t worker://initial.p
         Move('worker://traj/' > 'sandbox:///{}/00000076/)]
```

We are linking a lot of files to the worker directory and change the name for the .pdb in the process. Then call the actual `python` script that runs openmm. And finally move the `output.dcd` and the restart file back tp the trajectory folder.

There is a way to list lot's of things about tasks and we will use it a lot to see our modifications.

```
In [8]: print task.description
```

```
Task: TrajectoryGenerationTask(OpenMMEngine) [created]

Sources
- staging:///integrator.xml
- staging:///alanine.pdb
- staging:///openmmrun.py
- staging:///system.xml
Targets
- sandbox:////00000076/
Modified

<pretask>
Link('staging:///alanine.pdb' > 'worker://initial.pdb)
Link('staging:///system.xml' > 'worker://system.xml)
Link('staging:///integrator.xml' > 'worker://integrator.xml)
Link('staging:///openmmrun.py' > 'worker://openmmrun.py)
Touch('worker://traj/')
python openmmrun.py -r --report-interval 1 -p CPU --store-interval 1 -t worker://initial.pdb --length
Move('worker://traj/' > 'sandbox:////00000076/)
<posttask>
```

Modify a task

As long as a task is not saved and hence placed in the queue, it can be altered in any way. All of the 3 / 5 phases can be changed separately. You can add things to the staging phases or bash phases or change the command. So, let’s do that now

Add a bash line

First, a `Task` is very similar to a list of bash commands and you can simply append (or prepend) a command. A text line will be interpreted as a bash command.

```
In [9]: task.append('echo "This new line is pointless"')
```

```
In [10]: print task.description
```

```
Task: TrajectoryGenerationTask(OpenMMEngine) [created]

Sources
- staging:///integrator.xml
- staging:///alanine.pdb
- staging:///openmmrun.py
- staging:///system.xml
Targets
- sandbox:////00000076/
Modified

<pretask>
Link('staging:///alanine.pdb' > 'worker://initial.pdb)
Link('staging:///system.xml' > 'worker://system.xml)
Link('staging:///integrator.xml' > 'worker://integrator.xml)
Link('staging:///openmmrun.py' > 'worker://openmmrun.py)
Touch('worker://traj/')
python openmmrun.py -r --report-interval 1 -p CPU --store-interval 1 -t worker://initial.pdb --length
Move('worker://traj/' > 'sandbox:////00000076/)
echo "This new line is pointless"
<posttask>
```

As expected this line was added to the end of the script.

Add staging actions

To set staging is more difficult. The reason is, that you normally have no idea where files are located and hence writing a copy or move is impossible. This is why the staging commands are not bash lines but objects that hold information about the actual file transaction to be done. There are some task methods that help you move files but also files itself can generate this commands for you.

Let’s move one trajectory (directory) around a little more as an example

```
In [11]: traj = project.trajectories.one
In [12]: transaction = traj.copy()
         print transaction
```

```
Copy('sandbox:////00000010/' > 'worker://)
```

This looks like in the script. The default for a copy is to move a file or folder to the worker directory under the same name, but you can give it another name/location if you use that as an argument. Note that since trajectories are a directory you need to give a directory name (which end in a /)

```
In [13]: transaction = traj.copy('new_traj/')
         print transaction
```

```
Copy('sandbox:////00000010/' > 'worker://new_traj/)
```

If you want to move it not to the worker directory you have to specify the location and you can do so with the prefixes (`shared://`, `sandbox://`, `staging://` as explained in the previous examples)

```
In [14]: transaction = traj.copy('staging:///cached_trajs/')
         print transaction
```

```
Copy('sandbox:////00000010/' > 'staging:///cached_trajs/)
```

Besides `.copy` you can also `.move` or `.link` files.

```
In [15]: transaction = pdb_file.copy('staging:///delete.pdb')
         print transaction
         transaction = pdb_file.move('staging:///delete.pdb')
         print transaction
         transaction = pdb_file.link('staging:///delete.pdb')
         print transaction
```

```
Copy('file:///alanine.pdb' > 'staging:///delete.pdb)
Move('file:///alanine.pdb' > 'staging:///delete.pdb)
Link('file:///alanine.pdb' > 'staging:///delete.pdb)
```

Local files

Let's mention these because they require special treatment. We cannot (like RP can) copy files to the HPC, we need to store them in the DB first.

```
In [16]: new_pdb = File('file://../files/ntl9/ntl9.pdb').load()
```

Make sure you use `file://` to indicate that you are using a local file. The above example uses a relative path which will be replaced by an absolute one, otherwise we ran into trouble once we open the project at a different directory.

```
In [17]: print new_pdb.location
```

```
file:///Users/jan-hendrikprinz/Studium/git/adaptivemd/examples/files/ntl9/ntl9.pdb
```

Note that now there are 3 `/` in the filename, two from the `://` and one from the root directory of your machine

The `load()` at the end really loads the file and when you save this `File` now it will contain the content of the file. You can access this content as seen in the previous example.

```
In [18]: print new_pdb.get_file()[:300]
```

```
CRYST1   50.000   50.000   50.000  90.00  90.00  90.00 P 1
ATOM      1  N   MET     1      33.720  28.790  34.120  0.00  0.00           N
ATOM      2  H1  MET     1      33.620  29.790  33.900  0.00  0.00           H
ATOM      3  H2  MET     1      33.770  28.750  35.120  0.00  0.00
```

For local files you normally use `.transfer`, but `copy`, `move` or `link` work as well. Still, there is no difference since the file only exists in the DB now and copying from the DB to a place on the HPC results in a simple file creation.

Now, we want to add a command to the staging and see what happens.

```
In [19]: transaction = new_pdb.transfer()
         print transaction
```

```
Transfer('file:///ntl9.pdb' > 'worker://ntl9.pdb)
```

```
In [20]: task.append(transaction)
```

```
In [21]: print task.description
```

```
Task: TrajectoryGenerationTask(OpenMMEngine) [created]

Sources
- staging:///integrator.xml
- staging:///alanine.pdb
- staging:///openmmrun.py
- file:///ntl9.pdb [exists]
- staging:///system.xml
Targets
- sandbox:////00000076/
Modified

<pretask>
Link('staging:///alanine.pdb' > 'worker://initial.pdb)
Link('staging:///system.xml' > 'worker://system.xml)
Link('staging:///integrator.xml' > 'worker://integrator.xml)
Link('staging:///openmmrun.py' > 'worker://openmmrun.py)
Touch('worker://traj/')
python openmmrun.py -r --report-interval 1 -p CPU --store-interval 1 -t worker://initial.pdb --length
Move('worker://traj/' > 'sandbox:////00000076/)
echo "This new line is pointless"
Transfer('file:///ntl9.pdb' > 'worker://ntl9.pdb)
<posttask>
```

We now have one more transfer command. But something else has changed. There is one more files listed as required. So, the task can only run, if that file exists, but since we loaded it into the DB, it exists (for us). For example the newly created trajectory `25.dcd` does not exist yet. Would that be a requirement the task would fail. But let's check that it exists.

```
In [22]: new_pdb.exists
```

```
Out[22]: True
```

Okay, we have now the PDB file staged and so any real bash commands could work with a file `ntl9.pdb`. Alright, so let's output its stats.

```
In [23]: task.append('stat ntl9.pdb')
```

Note that usually you place these stage commands at the top or your script.

Now we could run this task, as before and see, if it works. (Make sure you still have a worker running)

```
In [24]: project.queue(task)
```

And check, that the task is running

```
In [33]: task.state
```

```
Out[33]: u'success'
```

If we did not screw up the task, it should have succeeded and we can look at the STDOUT.

```
In [34]: print task.stdout
```

```
13:11:19 [worker:3] stdout from running task
GO...
Reading PDB
```

```
Done
Initialize Simulation
Done.
('# platform used:', 'CPU')
('# temperature:', Quantity(value=300.0, unit=kelvin))
START SIMULATION
DONE
Written to directory `traj/`
This new line is pointless
16777220 97338745 -rw-r--r-- 1 jan-hendrikprinz staff 0 1142279 "Mar 21 13:11:18 2017" "Mar 21 13:11
```

Well, great, we have the pointless output and the stats of the newly staged file `ntl9.pdb`

How does a real script look like

Just for fun let's create the same scheduler that the `adaptivemdworker` uses, but from inside this notebook.

```
In [35]: from adaptivemd import WorkerScheduler
```

```
In [36]: sc = WorkerScheduler(project.resource)
```

If you really wanted to use the worker you need to initialize it and it will create directories and stage files for the generators, etc. For that you need to call `sc.enter(project)`, but since we only want it to parse our tasks, we only set the project without invoking initialization. You should normally not do that.

```
In [37]: sc.project = project
```

Now we can use a function `.task_to_script` that will parse a task into a bash script. So this is really what would be run on your machine now.

```
In [38]: print '\n'.join(sc.task_to_script(task))
```

```
set -e
# This is part of the adaptivemd tutorial
ln -s ../staging_area/alanine.pdb initial.pdb
ln -s ../staging_area/system.xml system.xml
ln -s ../staging_area/integrator.xml integrator.xml
ln -s ../staging_area/openmmrun.py openmmrun.py
mkdir -p traj/
python openmmrun.py -r --report-interval 1 -p CPU --store-interval 1 -t initial.pdb --length 100 tra
mkdir -p ../../projects/tutorial/trajs/00000076/
mv traj/* ../../projects/tutorial/trajs/00000076/
rm -r traj/
echo "This new line is pointless"
# write file `ntl9.pdb` from DB
stat ntl9.pdb
```

Now you see that all file paths have been properly interpreted to work. See that there is a comment about a temporary file from the DB that is then renamed. This is a little trick to be compatible with RPs way of handling files. (TODO: We might change this to just write to the target file. Need to check if that is still consistent)

A note on file locations

One problem with bash scripts is that when you create the tasks you have no concept on where the files actually are located. To get around this the created bash script will be scanned for paths, that contain prefixed like we are used to and are interpreted in the context of the worker / scheduler. The worker is the only instance to know all that is necessary so this is the place to fix that problem.

Let's see that in a little example, where we create an empty file in the staging area.

```
In [39]: task = Task()
         task.append('touch staging:///my_file.txt')
```

```
In [40]: print '\n'.join(sc.task_to_script(task))
```

```
set -e
# This is part of the adaptivemd tutorial
touch ../staging_area/my_file.txt
```

And voila, the path has changed to a relative path from the working directory of the worker. Note that you see here the line we added in the very beginning of example 1 to our resource!

A Task from scratch

If you want to start a new task you can begin with

```
In [41]: task = Task()
```

as we did before.

Just start adding staging and bash commands and you are done. When you create a task you can assign it a generator, then the system will assume that this task was generated by that generator, so don't do it for you custom tasks, unless you generated them in a generator. Setting this allows you to tell a worker only to run tasks of certain types.

The Python RPC Task

The tasks so far a very powerful, but they lack the possibility to call a python function. Since we are using python here, it would be great to really pretend to call a python function from here and not taking the detour of writing a python bash executable with arguments, etc... An example for this is the PyEmma generator which uses this capability.

Let's do an example of this as well. Assume we have a python function in a file (you need to have your code in a file so far so that we can copy the file to the HPC if necessary). Let's create the `.py` file now.

```
In [42]: %%file my_rpc_function.py

         def my_func(f):
             import os
             print f
             return os.path.getsize(f)
```

```
Overwriting my_rpc_function.py
```

Now create a PythonTask instead

```
In [43]: task = PythonTask()
```

and the call function has changed. Note that also now you can still add all the bash and stage commands as before. A PythonTask is also a subclass of `PrePostTask` so we have a `.pre` and `.post` phase available.

```
In [44]: from my_rpc_function import my_func
```

We call the function `my_func` with one argument

```
In [45]: task.call(my_func, f=project.trajectories.one)
```

```
In [46]: print task.description
```

```
Task: PythonTask(NoneType) [created]

Sources
- staging:///_run_.py
- file:///_rpc_input_0x71bdd2d10e2f11e7a0f00000000002eaL.json
- file:///my_rpc_function.py [exists]
Targets
- file:///_rpc_output_0x71bdd2d10e2f11e7a0f00000000002eaL.json
Modified

<pretask>
Transfer('file:///_rpc_input_0x71bdd2d10e2f11e7a0f00000000002eaL.json' > 'worker://input.json)
Link('staging:///_run_.py' > 'worker://_run_.py)
Transfer('file:///my_rpc_function.py' > 'worker://my_rpc_function.py)
python _run_.py
Transfer('worker://output.json' > 'file:///_rpc_output_0x71bdd2d10e2f11e7a0f00000000002eaL.json)
<posttask>
```

Well, interesting. What this actually does is to write the input arguments to the function into a temporary .json file on the worker, (in RP on the local machine and then transfers it to remote), rename it to input.json and read it in the _run_.py. This is still a little clumsy, but needs to be this way to be RP compatible which only works with files! Look at the actual script.

You see, that we really copy the .py file that contains the source code to the worker directory. All that is done automatically. A little caution on this. You can either write a function in a single file or use any installed package, but in this case the same package needs to be installed on the remote machine as well!

Let's run it and see what happens.

```
In [47]: project.queue(task)
```

And wait until the task is done

```
In [48]: project.wait_until(task.is_done)
```

The default settings will automatically save the content from the resulting output.json in the DB an you can access the data that was returned from the task at .output. In our example the result was just the size of a the file in bytes

```
In [49]: task.output
```

```
Out[49]: 136
```

And you can use this information in an adaptive script to make decisions.

The last thing we did not talk about is the possibility to also call a function with the returned data automatically on successful execution. Since this function is executed on the worker we (so far) only support function calls with the following restrictions.

1. you can call a function of the related generator class. for this you need to create the task using PythonTask(generator)
2. the function name you want to call is stored in task.then_func_name. So you can write a generator class with several possible outcomes and chose the function for each task.
3. The Generator needs to be part of adaptivemd

So in the case of modeller.execute we create a PythonTask that references the following functions

```
In [50]: task = modeller.execute(project.trajectories)
```

```
In [51]: task.then_func_name
```

```
Out[51]: 'then_func'
```

So we will call the default then_func of modeller or the class modeller is of.

```
In [52]: help(modeller.then_func)
```

```
Help on function then_func in module adaptivemd.analysis.pyemma.emma:
```

```
then_func(project, task, model, inputs)
```

These callbacks are called with the current project, the resulting data (which is in the modeller case a `Model` object) and array of initial inputs.

This is the actual code of the callback

```
@staticmethod
def then_func(project, task, model, inputs):
    # add the input arguments for later reference
    model.data['input']['trajectories'] = inputs['kwargs']['files']
    model.data['input']['pdb'] = inputs['kwargs']['topfile']
    project.models.add(model)
```

All it does is to add some of the input parameters to the model for later reference and then store the model in the project. You are free to define all sorts of actions here, even queue new tasks.

Next, we will talk about the factories for `Task` objects, called `generators`. There we will actually write a new class that does some stuff with the results.

```
In [53]: project.close()
```

```
In [ ]:
```

Example 5 - Generators

Custom `Generator` objects

This example should guide you to build your own simple generator.

```
In [ ]: from adaptivemd import (
            Project, Task, File, PythonTask
        )

        project = Project('tutorial')

        engine = project.generators['openmm']
        modeller = project.generators['pyemma']
        pdb_file = project.files['initial_pdb']
```

Basic knowledge

We assume that you have completed at least some of the previous examples and have a general idea of how adaptiveMD works. Still, let’s recapitulate what we think is the typical way of a simulation.

How to execute something

To execute something you need

1. a description of the task to be done. This is the `Task` object. Once you have this you can,
2. use it in a `Scheduler` which will interpret the `Task` into some code that the computer understands. It handles all the little things you expect from the task, like registering generated file, etc... And to do so, the `Scheduler` needs
3. your `Resource` description which acts like a config for the scheduler

When you have a Scheduler (with Resource) you let it execute Task objects. If you know how to build these you are done. That is all you need.

What are Generators?

Build a task can be cumbersome and often repetative, and a factory for Task objects is extremely useful. These are called Generators (maybe TaskFactory) is a better name?!?

In your final scheme where you observe all generated objects and want to build new tasks accordingly you will (almost) never build a Task yourself. You use a generator.

A typical example is an Engine. It will generate tasks, that simulate new trajectories, extend existing ones, etc... Basic stuff. The second big class is Analysis. It will use trajectories to generate models or properties of interest to guide your decisions for new trajectories.

In this example we will build a simple generator for a task, that uses the mdtraj package to compute some features and store these in the database and in a file.

The MDTrajFeaturizer generator

First, we think about how this featurizer works if we would not use adaptivemd. The reason is, that we have basically two choices for designing a Task (see example 4 about Task objects).

1. A task that calls bash commands for you
2. A task that calls a python function for you

Since we want to call mdtraj functions we use the 2nd and start with a skeleton for this type and store it under my_generator.py

```
In [ ]: %% file my_generator.py
        # This is an example for building your own generator
        # This file must be added to the project so that it is loaded
        # when you import `adaptivemd`. Otherwise your worker don't know
        # about the class!

        from adaptivemd import Generator

        class MDTrajFeaturizer(Generator):
            def __init__(self, {things we always need}):
                super(PyEMMAAnalysis, self).__init__()

                # stage file you want to reuse (optional)
                # self['pdb_file'] = pdb_file
                # stage = pdb_file.transfer('staging:///')
                # self['pdb_file_stage'] = stage.target
                # self.initial_staging.append(stage)

            @staticmethod
            def then_func(project, task, data, inputs):
                # add the output for later reference
                project.data.add(data)

            def execute(self, {options per task}):

                t = PythonTask(self)

                # get your staged files (optional)
                # input_pdb = t.link(self['pdb_file_stage'], 'input.pdb')

                # add the python function call to your script (there can be only one!)
                t.call(
                    my_script,
                    param1,
                    param2,
                    ...
                )

                return t

        def my_script(param1, param2, ...):
            return {"whatever you want to return"}
```

What input does our generator always need?

Mdtraj needs a topology unless it is already present. Interestingly, our `Trajectory` objects know about their topology so we could access these, if our function is to process a `Trajectory`. This requires the `Trajectory` to be the input. If we want to process any file, then we might need a topology.

The decision if we want the generator to work for a fixed topology is yours. To show how this would work, we do this here. We use a fixed topology per generator that applies to `File` objects.

Second is the feature we want to compute. This is tricky and so we hard code this now. You can think of a better way to represent this. But let's pick the tertiary stucture prediction

```
In [ ]: def __init__(self, pdb_file=None):
            super(PyEMMAAnalysis, self).__init__()

            # if we provide a pdb_file it should be used
            if pdb_file is not None:
                # stage file you want to reuse (optional)

                # give the file an internal name
                self['pdb_file'] = pdb_file
                # create the transfer from local to staging:
                stage = pdb_file.transfer('staging:///')
                # give the staged file an internal name
                self['pdb_file_stage'] = stage.target
                # append the transfer action to the initial staging action list
                self.initial_staging.append(stage)
```

The task building

```
In [ ]: def execute(self, file_to_analyze):

            assert(isinstance(file_to_analyze, File))

            t = PythonTask(self)

            # get your staged files (optional)
            if self.get('pdb_file_stage'):
                input_pdb = t.link(self['pdb_file_stage'], 'input.pdb')
            else:
                input_pdb = None
            # add the python function call to your script (there can be only one!)
            t.call(
                my_script,
                file_to_analyze,
                input_pdb
            )

            return t
```

The actual script

This script is executed on the HPC for you. And requires mdtraj to be installed on it.

```
In [ ]: def my_script(file_to_analyze, input_pdb):
            import mdtraj as md

            traj = md.load(file_to_analyze, top=input_pdb)
            features = traj.compute_xyz()

            return features
```

That's it. At least in the simplest form. When you use this to create a `Task`

```
In [ ]: my_generator = MDTrajFeaturizer(pdb_file)
        task = my_generator.execute(traj.file('master.dcd'))
        project.queue(task)
```

We wait and then the `Task` object has a `.output` property which now contains the returned result.

This can now be used in your execution plans...

```
In [ ]: def strategy():
            # generate some structures...
            # yield wait ...
            # get a traj object
            task = my_generator.execute(traj.outputs('master'))
            # wait until the task is done
            yield task.is_done
            # print the output
            output = task.output
            # do something with the result, store in the DB, etc...
```

Next, we look at improvements

Better storing of results

Often you want to save the output from your function in the DB in some form or another. Though the output is stored, it is not conviniently accessed unless you know the task that was used.

For this reason there is a callback function you can set, that can take care of doing a custom handling of the output. The function to be called needs to be a method of the generator and you can give the task the name of the method. The name (str) of the funtion can be set using the `then()` command. An the default name is `then_func`.

```
In [ ]: def execute(self, ...):
            t = PythonTask(self)
            t.then('handle_my_output')

        @staticmethod
        def handle_my_output(project, task, data, inputs):
            print 'Saving data from task', task, 'into model'
            m = Model(data)
            project.model.add(m)
```

The function takes exactly 4 parameters

1. `project`: the project in which the task was run. Is used to access the database, etc
2. `task`: the actual task object that produced the output
3. `data`: the output returned by the function
4. `inputs`: the input to the python function call (internally). The data actually transmitted to the worker to run

Like in the above example you can do whatever you want with your data, store it, alter it, write it to a file, etc. In case you do not want to additionally save the output (`data`) in the DB as an object, you can tell the trask not to by setting

```
In [ ]: def execute(self, ...):
            t = PythonTask(self)
            t.then('handle_my_output')
            t.store_output = False  # default is `True`
```

in that case `.output` will stay `None` even after execution

Working with `Trajectory` files and get their properties

Note that you always have to write file generation and file analysis/reading that matches. We only store some very general properties of objects with them, e.g. a stride for trajectories. This means you cannot arbitrarily mix code for these.

Now we want that this works

```
In [ ]: my_generator.execute(traj)
```

This is rather simple: All you need to do is to extract the actual files from the trajectory object.

```
In [ ]: def __init__(self, outtype, pdb_file=None):
            super(PyEMMAAnalysis, self).__init__()

            # we store a str that holds the name of the outputtype
            # this must match the definition
            self.outtype = outtype

            # ...

        def execute(self, traj, *args, **kwargs):
            t = PythonTask(self)
            # ...
            file_location = traj.outputs(self.outtype)  # get the trajectory file matching outtype
            # use the file_location.

            # ...
```

Import! You have no access to the `Trajectory` object in our remove function. These will be converted to a real path relative to the working directory. This makes sure that you will not have to deal with prefixes, etc. This might change in the future, but. The scripts are considered independent of adaptivemd!

Problem with saving your generator to the DB

This is not complicated but you need to briefly learn about the mechanism to store complex Python objects in the DB. The general way to Store an instance of a class requires you to subclass from `adaptivemd.mongodb.StorableMixin`. This provides the class with a `__uuid__` attribute that is a unique number for each storable object that is given at creation time. (If we would just store objects using pymongo we would get a number like this, but later). Secondly, it add two functions

1. `to_dict()`: this converts the (immutable) state of the object into a dictionary that is simple enough that it can be stored. Simple enought means, that you can have Python primitives, things like numpy arrays or even other storable objects, but not arbitrary objects in it, like lambda constructs (these are possible but need special treatment)
2. `from_dict()`: The reverse. It takes the dictionary from `to_dict` and must return an equivalent object!

So, you can do

```
clone = obj.__class__.from_dict(obj.to_dict())
```

and get an equal object in that it has the same attributes. You could also say a deep copy.

This is not always trivial and there exists a default implementation, which will make an additional assumption:

All necessary attributes have the same parameters in `__init__`. So, this would correspond to this rule

```
In [ ]: class MyStorableObject(StorableMixin):
            def __init__(self, state):
                self.state = state
```

while this would not work

```
In [ ]: class MyStorableObject(StorableMixin):
            def __init__(self, initial_state):
                self.state = initial_state
```

In the second case you need to overwrite the default function. All of these will work

```
In [ ]: # fix `to_dict` to match default `from_dict`
        class MyStorableObject(StorableMixin):
            def __init__(self, initial_state):
                self.state = initial_state

            def to_dict(self):
                return {
                    'initial_state': self.state
                }
```

```
In [ ]: # fix `from_dict` to match default `to_dict`
        class MyStorableObject(StorableMixin):
            def __init__(self, initial_state):
                self.state = initial_state

            @classmethod
            def from_dict(cls, dct):
                return cls(initial_state=dct['state'])
```

```
In [ ]: # fix both `from_dict` and `to_dict`
        class MyStorableObject(StorableMixin):
            def __init__(self, initial_state):
                self.state = initial_state

            def to_dict(self):
                return {
                    'my_state': self.state
                }

            @classmethod
            def from_dict(cls, dct):
                return cls(initial_state=dct['my_state'])
```

If you do that, make sure that you really capture all variables. Especially if you subclass from an existing one. You can use super to access the result from the parent class

```
In [ ]: class MyStorableObject(StorableMixin):
            @classmethod
            def from_dict(cls, dct):
                obj = super(MyStorableObject, cls).from_dict(dct)
                obj.missing_attr1 = dct['missing_attr_key1']
                return obj

            def to_dict(self):
                dct = super(MyStorableObject, self).to_dict(self)
                dct.update({
                    'missing_attr_key1': self.missing_attr1
                })
                return dct
```

This is the recommended way to build your custom functions. For completeness we show here what the base `TaskGenerator` class will do

```
In [ ]: @classmethod
        def from_dict(cls, dct):
            obj = cls.__new__(cls)
            StorableMixin.__init__(obj)
            obj._items = dct['_items']
            obj.initial_staging = dct['initial_staging']
            return obj

        def to_dict(self):
            return {
                '_items': self._items,
                'initial_staging': self.initial_staging
            }
```

The only unfamiliar part is the

```
obj = cls.__new__(cls)
StorableMixin.__init__(obj)
```

which needs a little explanation.

In most `__init__` functions for a `TaskGenerator` you will construct the `initial_staging` attribute with some functions. If you would reconstruct by just calling the constructor with the same parameters again, this would result in an equal object as expected and that would work, but not in all regards as expected: The problem is that if you generate objects that can be stored, these will get new UUIDs and hence are considered different from the ones that you wanted to store. In short, the construction in the `__init__` prevents you from getting the real old object back, you always construct something new.

This can be solved by not using `__init__` but creating an empty object using `__new__` and then fixing all attributes to the original state. This is very similar to `__setstate__` which we do not use in general to still allow using `__init__` which makes sense in most cases where not storable objects are generated.

In the following we discuss an existing generator

A simple generator

In [ ]:

A word about this example. While a `Task` can be created and configured a new class in `adaptivemd` needs to be part of the project. So we will write discuss the essential parts of the existing code.

A generator is in essence a factory to create `Task` objects with a single command. A generator can be initialized with certain files that the created tasks will always need, like an engine will need a topology for each task, etc. It also (as explained briefly before in Example 4) knows about certain callback behaviour of their tasks. Last, a generator allows you to assign a worker only to tasks that were created by a generator.

Let's look at the code of the PyEMMAAnalysis

```
class PyEMMAAnalysis(Analysis):
    def __init__(self, pdb_file):
        super(PyEMMAAnalysis, self).__init__()

        self['pdb_file'] = pdb_file
        stage = pdb_file.transfer('staging:///')

        self['pdb_file_stage'] = stage.target
        self.initial_staging.append(stage)

    @staticmethod
    def then_func(project, task, model, inputs):
        # add the input arguments for later reference
        model.data['input']['trajectories'] = inputs['files']
        model.data['input']['pdb'] = inputs['topfile']
        project.models.add(model)

    def execute(
            self,
            trajectories,
            tica_lag=2,
            tica_dim=2,
            msm_states=5,
            msm_lag=2,
            stride=1):

        t = PythonTask(self)

        input_pdb = t.link(self['pdb_file_stage'], 'input.pdb')
        t.call(
            remote_analysis,
            trajectories=list(trajectories),
            topfile=input_pdb,
            tica_lag=tica_lag,
            tica_dim=tica_dim,
            msm_states=msm_states,
            msm_lag=msm_lag,
            stride=stride
        )

        return t
```

```
def __init__(self, pdb_file):
    # don't forget to call super
    super(PyEMMAAnalysis, self).__init__()
    # a generator also acts like a dictionary for files
    # this way you can later access certain files you might need

    # save the pdb_file under the same name
    self['pdb_file'] = pdb_file

    # this creates a transfer action like it is used in tasks
    # and moves the passed pdb_file (usually on the local machein)
    # to the staging_area root directory
    stage = pdb_file.transfer('staging:///')

    # and the new target file (which is also like the original)
    # on the staging_area is saved unter `pdb_file_stage`
    # so, we can access both files if we wanted to
    # note that the original file most likely is in the DB
    # so we could just skip the stage transfer completely
    self['pdb_file_stage'] = stage.target

    # last we add this transfer to the initial_staging which
    # is done only once per used generator
    self.initial_staging.append(stage)
```

```
# the kwargs is to keep the exmaple short, you should use explicit
# parameters and add appropriate docs
def execute(self, trajectories, **kwargs):
    # create the task and set the generator to self, our new generator
    t = PythonTask(self)

    # we want to copy the staged file to the worker directory
    # and name it `input.pdb`
    input_pdb = t.link(self['pdb_file_stage'], 'input.pdb')

    # if you chose not to use the staging file and copy it directly you
    # would use in analogy
    # input_pdb = t.link(self['pdb_file'], 'input.pdb')

    # finally we use `.call` and want to call the `remote_analysis` function
    # which we imported earlier from somewhere
    t.call(
        remote_analysis,
        trajectories=list(trajectories),
        **kwargs
    )

    return t
```

And finally a call_back function. The name `then_func` is the default function name to be called.

```
# we use a static method, but you can of course write a normal method
@staticmethod
# the call_backs take these arguments in this order
# the second parameter is actually a `Model` object in this case
# which has a `.data` attribute
def then_func(project, task, model, inputs):
    # add the input arguments for later reference to the model
    model.data['input']['trajectories'] = inputs['kwargs']['files']
    model.data['input']['pdb'] = inputs['kwargs']['topfile']
    # and save the model in the project
    project.models.add(model)
```

A brief summary and things you need to set to make your generator work

```
class MyGenerator(Analysis):
    def __init__(self, {things your generator always needs}):
        super(MyGenerator, self).__init__()

        # Add input files to self
        self['file1'] = file1

        # stage all files to the staging area of you want to keep these
        # files on the HPC
        for fn in ['file1', 'file2', ...]:
            stage = self[fn].transfer('staging:///')
            self[fn + '_stage'] = stage.target
            self.initial_staging.append(stage)

    @staticmethod
    def then_func(project, task, outputs, inputs):
        # do something with input and outputs
        # store something in your project

    def task_using_python_rpc(
            self,
            {arguments}):

        t = PythonTask(self)

        # set any task dependencies if you need
        t.dependencies = []

        input1 = t.link(self['file1'], 'alternative_name1')
        input2 = t.link(self['file2'], 'alternative_name2')
        ...

        # add whatever bash stuff you need BEFORE the function call
        t.append('some bash command')
        ...

        # use input1, etc in your function call if you like. It will
        # be converted to a regular file location you can use
        t.call(
            {my_remote_python_function},
            files=list(files),
        )

        # add whatever bash stuff you need AFTER the function call
        t.append('some bash command')
        ...

        return t

    def task_using_bash_argument_call(
            self,
            {arguments}):
```

```
        t = Task(self)

        # set any task dependencies if you need
        t.dependencies = []

        input1 = t.link(self['file1'], 'alternative_name1')
        input2 = t.link(self['file2'], 'alternative_name2')
        ...
        # add more staging
        t.append({action})
        ...

        # add whatever bash stuff you want to do
        t.append('some bash command')
        ...

        # add whatever staging stuff you need AFTER the function call
        t.append({action})
        ...

        return t
```

The simplified code for the OpenMMEngine

```
class OpenMMEngine(Engine):
    trajectory_ext = 'dcd'

    def __init__(self, system_file, integrator_file, pdb_file, args=None):
        super(OpenMMEngine, self).__init__()

        self['pdb_file'] = pdb_file
        self['system_file'] = system_file
        self['integrator_file'] = integrator_file
        self['_executable_file'] = exec_file

        for fn in self.files:
            stage = self[fn].transfer(Location('staging:///'))
            self[name + '_stage'] = stage.target
            self.initial_staging.append(stage)

        if args is None:
            args = '-p CPU --store-interval 1'

        self.args = args

    # this one only works if you start from a file
    def task_run_trajectory_from_file(self, target):
        # we create a special Task, that has some additional functionality
        t = TrajectoryGenerationTask(self, target)

        # link all the files we require
        initial_pdb = t.link(self['pdb_file_stage'], Location('initial.pdb'))
        t.link(self['system_file_stage'])
        t.link(self['integrator_file_stage'])
        t.link(self['_executable_file_stage'])

        # use the initial PDB to be used
        input_pdb = t.get(target.frame, 'coordinates.pdb')
        # this represents our output trajectory
        output = Trajectory('traj/', target.frame, length=target.length, engine=self)

        # create the directory so openmmrun can write to it
        t.touch(output)

        # build the actual bash command
        cmd = 'python openmmrun.py {args} -t {pdb} --length {length} {output}'.format(
            pdb=input_pdb,
            length=target.length,
            output=output,
            args=self.args,
        )
        t.append(cmd)

        # copy the resulting trajectory directory back to the staging area
        t.put(output, target)

        return t
```

```
In [ ]: project.close()
```

Example 6 - Multiple Output Types

AdaptiveMD

Example 6 - Multi-traj

0. Imports

```
In [1]: import sys, os
```

Alright, let's load the package and pick the `Project` since we want to start a project

```
In [2]: from adaptivemd import Project
```

Let's open a project with a UNIQUE name. This will be the name used in the DB so make sure it is new and not too short. Opening a project will always create a non-existing project and reopen an exising one. You cannot chose between opening types as you would with a file. This is a precaution to not accidentally delete your project.

```
In [3]: # Use this to completely remove the example-worker project from the database.
        Project.delete('tutorial-multi')
```

```
In [4]: project = Project('tutorial-multi')
```

Now we have a handle for our project. First thing is to set it up to work on a resource.

1. Set the resource

What is a resource? A `Resource` specifies a shared filesystem with one or more clusteres attached to it. This can be your local machine or just a regular cluster or even a group of cluster that can access the same FS (like Titan, Eos and Rhea do).

Once you have chosen your place to store your results this way it is set for the project and can (at least should) not be altered since all file references are made to match this resource. Currently you can use the Fu Berlin Allegro Cluster or run locally. There are two specific local adaptations that include already the path to your conda installation. This simplifies the use of `openmm` or `pyemma`.

Let us pick a local resource on a laptop for now.

```
In [5]: from adaptivemd import LocalCluster, AllegroCluster
```

first pick your resource – where you want to run your simulation. Local or on Allegro

```
In [6]: resource = LocalCluster()
```

```
In [7]: project.initialize(resource)
```

2. Add `TaskGenerators`

TaskGenerators are instances whose purpose is to create tasks to be executed. This is similar to the way Kernels work. A TaskGenerator will generate `Task` objects for you which will be translated into a `ComputeUnitDescription` and executed. In simple terms:

The task generator creates the bash scripts for you that run a simulation or run pyemma.

A task generator will be initialized with all parameters needed to make it work and it will now what needs to be staged to be used.

```
In [8]: from adaptivemd.engine.openmm import OpenMMEngine
        from adaptivemd import File, Directory
```

The engine

```
In [9]: pdb_file = File('file://../files/alanine/alanine.pdb').named('initial_pdb').load()
```

```
In [10]: engine = OpenMMEngine(
             pdb_file=pdb_file,
             system_file=File('file://../files/alanine/system.xml').load(),
             integrator_file=File('file://../files/alanine/integrator.xml').load(),
             args='-r --report-interval 1 -p CPU'
         ).named('openmm')
```

```
In [11]: engine.add_output_type('master', 'master.dcd', 10)
         engine.add_output_type('protein', 'protein.dcd', 1)
```

```
In [12]: engine.types
```

```
Out[12]: {'master': <adaptivemd.engine.engine.OutputTypeDescription at 0x10f7254d0>,
          'protein': <adaptivemd.engine.engine.OutputTypeDescription at 0x10f725510>}
```

```
In [13]: project.generators.add(engine)
```

```
In [14]: s = engine._create_output_str()
         print s
```

```
--types="'protein':'stride':1,'filename':'protein.dcd','master':'stride':10,'filename':'master.dcd'"
```

```
In [15]: task = project.new_trajectory(pdb_file, 100, engine=engine).run()
```

3. Create one intial trajectory

Create a `Trajectory` object

```
In [16]: project.queue(task)  # shortcut for project.tasks.add(task)
```

That is all we can do from here. To execute the tasks you need to run a worker using

```
adaptivemdworker -l tutorial --verbose
```

```
In [17]: print project.tasks
```

```
<StoredBundle for with 2 file(s) @ 0x10f6e3e90>
```

```
In [18]: task.trajectory
```

```
Out[18]: Trajectory('alanine.pdb' >> [0..100])
```

```
In [21]: task.state
```

```
Out[21]: u'success'
```

```
In [22]: t = project.trajectories.one
```

```
In [24]: t.types['protein']
```

```
Out[24]: <adaptivemd.engine.engine.OutputTypeDescription at 0x10f725510>
```

Once this is done, come back here and check your results. If you want you can execute the next cell which will block until the task has been completed.

```
In [25]: print project.files
         print project.trajectories
```

```
<StoredBundle for with 5 file(s) @ 0x10f6e3e50>
<ViewBundle for with 1 file(s) @ 0x10f6e3e10>
```

and close the project.

```
In [25]: project.close()
```

The final project.close() will close the DB connection.

```
In [ ]:
```

Projects

file_structure

Classes

Project(name)	A simulation project

adaptivemd.Project

class adaptivemd.**Project**(*name*)

A simulation project

Notes

You will later create *Scheduler* objects that explicitly correspond to a specific cue on a specific cluster that is accessible from within this shared FS resource.

Variables

- **name** (*str*) – a short descriptive name for the project. This name will be used in the database creation also.
- **resource** (*Resource*) – a resource to run the project on. The resource specifies the memory storage location. Not necessarily which cluster is used. An example is, if at an institute several clusters (CPU, GPU) share the same shared FS. If clusters use the same FS you can run simulations across clusters without problems and so so this resource is the most top-level limitation.
- **files** (*Bundle*) – a set of file objects that are available in the project and are believed to be available within the resource as long as the project lives
- **trajectories** (*ViewBundle*) – all *File* object that are of *Trajectory* type and which have a positive *created* attribute. This means the file was really created and has not been altered yet.
- **workers** (*Bundle*) – a set of all registered *Worker* instanced in the project
- **files** – a set of file objects that are available in the project and are believed to be available within the resource as long as the project lives
- **models** (*Bundle*) – a set of stored models in the DB
- **tasks** (*Bundle*) – a set of all queued ‘Task‘s in the project
- **logs** (*Bundle*) – a set of all stored log entries
- ***data*** (*Bundle*) – a set of *DataDict* objects that represent completely stored files in the database of arbitrary size
- **schedulers** (set of *Scheduler*) – a set of attached schedulers with controlled shutdown and reference
- **storage** (*MongoDBStorage*) – the mongodb storage wrapper to access the database of the project
- **_worker_dead_time** (*int*) – the time after which an unresponsive worker is considered dead. Its tasks will be assigned the state set in `_set_task_state_from_dead_workers`. Default is 60s. Make sure that the heartbeat of a worker is much less that this.
- **_set_task_state_from_dead_workers** (*str*) – if a worker is dead then its tasks are assigned this state. Default is `created` which means the task will be restarted by another worker. You can also chose `halt` or `cancelled`. See *Task* for details

See also:

Task

__init__(*name*)

Methods

`__init__`(name)	
`add_event`(event)	Attach an event to the project
`close`()	Close the project and all related sessions and DB connections
`close_rp`()	Close the RP session

Continued on next page

Table 1.2 – continued from previous page

delete(name)	Delete a complete project
events_done()	Check if all events are done
find_ml_next_frame([n_pick])	Find initial frames picked by inverse equilibrium distribution
get_scheduler([name])	**param name** name of the scheduler class provided by the *Resource* used in
initialize(resource)	Initialize a project with a specific resource.
list()	List all projects in the DB
new_ml_trajectory(engine, length, number)	Find trajectories that have initial points picked by inverse eq dist
new_trajectory(frame, length[, engine, number])	Convenience function to create a new *Trajectory* object
on_nmodel(numbers)	Return a condition representing the reach of a certain number of models
on_ntraj(numbers)	Return a condition that is true as soon a the project has n trajectories
queue(*tasks)	Submit jobs to the worker queue
reconnect()	Reconnect the DB
run()	Starts observing events in the project
stop()	Stop observing events
trigger()	Trigger a check of state changes that leads to task execution
wait_until(condition)	Block until the given condition evaluates to true

initialize(*resource*)

Initialize a project with a specific resource.

Notes

This should only be called to setup the project and only the very first time.

Parameters **resource** (*Resource*) – the resource used in this project

reconnect()

Reconnect the DB

close_rp()

Close the RP session

Before using RP you need to re-open and then you will run in a new session.

classmethod list()

List all projects in the DB

Returns a list of all project names

Return type list of str

classmethod delete(*name*)

Delete a complete project

Notes

Attention!!!! This cannot be undone!!!!

Parameters **name** (*str*) – the project name to be deleted

get_scheduler(*name=None*, ***kwargs*)

Parameters

- **name** (*str*) – name of the scheduler class provided by the *Resource* used in this project. If None (default) the cluster/queue `default` is used that needs to be implemented for every resource
- **kwargs** (***kwargs*) – Additional arguments to initialize the cluster scheduler provided by the *Resource*

Notes

the scheduler is automatically entered/opened so the pilot jobs is submitted to the queueing system and it counts against your simulation time! If you do not want to do so directly. Create the *Scheduler* by yourself and later call `scheduler.enter(project)` to start using it. To close the scheduler call `scheduler.exit()`

Returns the scheduler object that can be used to execute tasks on that cluster/queue

Return type *Scheduler*

close()

Close the project and all related sessions and DB connections

queue(**tasks*)

Submit jobs to the worker queue

Parameters **tasks** ((list of) *Task* or *Trajectory*) – anything that can be run like a *Task* or a *Trajectory* with engine

new_trajectory(*frame*, *length*, *engine=None*, *number=1*)

Convenience function to create a new *Trajectory* object

It will use incrementing numbers to create trajectory names used in the engine executions. Use this function to always get an unused trajectory name.

Parameters

- **frame** (*File* or *Frame*) – if given a *File* it is assumed to be a `.pdb` file that contains initial coordinates. If a frame is given one assumes that this *Frame* is the initial structure / frame zero in this trajectory
- **length** (*int*) – the length of the trajectory
- **engine** (*Engine* or None) – the engine used to generate the trajectory. The engine contains all the specifics about the trajectory internal structure since it is the responsibility of the engine to really create the trajectory.
- **number** (*int*) – the number of trajectory objects to be returned. If `1` it will be a single object. Otherwise a list of *Trajectory* objects.

Returns

Return type *Trajectory* or list of *Trajectory*

on_ntraj(*numbers*)

Return a condition that is true as soon a the project has n trajectories

Parameters **numbers** (*int or iterator of int*) – either a single int or an iterator that returns several ints

Returns the single condition or a generator of conditions matching the ints in the iterator

Return type *NTrajectories* or generator of *NTrajectories*

on_nmodel(*numbers*)

Return a condition representing the reach of a certain number of models

Parameters **numbers** (*int or iterator of int*) – the number(s) of the models to be reached

Returns a (list of) *Condition*

Return type (generator of) *Condition*

find_ml_next_frame(*n_pick=10*)

Find initial frames picked by inverse equilibrium distribution

This is the simplest adaptive strategy possible. Start from the states more likely if a state has not been seen so much. Effectively stating that less knowledge of a state implies a higher likelihood to find a new state.

Parameters **n_pick** (*int*) – number of returned trajectories

Returns the list of trajectories with the selected initial points.

Return type list of *Frame*

new_ml_trajectory(*engine*, *length*, *number*)

Find trajectories that have initial points picked by inverse eq dist

Parameters

- **engine** (*Engine*) – the engine to be used
- **length** (*int*) – length of the trajectories returned
- **number** (*int*) – number of trajectories returned

Returns the list of *Trajectory* objects with initial frames chosen using *find_ml_next_frame()*

Return type list of *Trajectory*

See also:

find_ml_next_frame()

events_done()

Check if all events are done

Returns True if all events are done

Return type bool

add_event(*event*)

Attach an event to the project

These events will not be stored and only run in the current python session. These are the parts responsible to create tasks given certain conditions.

Parameters **event** (*Event* or generator) – the event to be added or a generator function that is then converted to an *ExecutionPlan*

Returns the actual event used

Return type *Event*

trigger()

Trigger a check of state changes that leads to task execution

This needs to be called regularly to advance the simulation. If not, certain checks for state change will not be called and no new tasks will be generated.

run()

Starts observing events in the project

This is still somehow experimental and will call a background thread to call *Project.trigger()* in regular intervals. Make sure to call *Project.stop()* before you quit the notebook session or exit. Otherwise there might be a job in the background left (not confirmed but possible!)

stop()

Stop observing events

wait_until(*condition*)

Block until the given condition evaluates to true

Parameters **condition** (*callable*) – function that is called in regular intervals. If it evaluates to True the function returns

class **EventTriggerTimer**(*event*, *project*)

A special thread to call the project trigger mechanism

Resources

A Resource specifies a shared filesystem with one or more clusteres attached to it. This can be your local machine or just a regular cluster or even a group of cluster that can access the same FS (like Titan, Eos and Rhea do).

Once you have chosen your place to store your results t is set for the project and can (at least should) not be altered since all file references are made to match this resource.

Let us pick a local resource on your laptop or desktop machine; no cluster / HPC involved for now.

```
from adaptivemd import LocalResource
```

We now create the Resource object

```
resource = LocalResource()
```

Since this object defines the path where all files will be placed, let's get the path to the shared folder. The one that can be accessed from all workers. On your local machine this is trivially the case.

```
resource.shared_path
```

```
'$HOME/adaptivemd/'
```

Okay, files will be placed in `$HOME/adaptivemd/`. You can change this using an option when creating the Resource

```
LocalCluster(shared_path='$HOME/my/adaptive/folder/')
```

Configuring your resource

Now you can add some additional paths, conda environment, etc, before we setup the project. This works by setting a special task `.wrapper` (see notebook 4 for more things you can do with `Task` objects.)

```
resource.wrapper
```

```
<adaptivemd.task.DummyTask at 0x110d93d50>
```

In a nutshell, this dummy task has a `.pre` and `.post` list of commands you can add any command you want to be executed before every task you run.

```
resource.wrapper.pre.append('echo "Hello World"')
```

A task can also automatically add to the `PATH` variable, set environment variables and you can add conda environments

```
resource.wrapper.add_conda_env('my_env_python_27')
```

```
resource.wrapper.add_path('/x/y/z')
```

```
resource.wrapper.environment['CONDA'] = 'True'
```

```
print resource.wrapper.description
```

```
Task: DummyTask
<pre>
export PATH=/x/y/z:$PATH
export CONDA=True
echo "Hello World"
</pre>
<main />
<post>
</post>
```

Let's reset that now and just add a little comment

```
resource = LocalResource()
resource.wrapper.pre.append('# This is part of the adaptivemd tutorial')
```

Finalize the `Resource`

Last, we save our configured `Resource` and initialize our empty prohect with it. This is done once for a project and should not be altered.

```
project.initialize(resource)
```

Classes

LocalResource([shared_path, wrapper])	Run tasks locally and store results in `$HOME/adaptivemd/`

adaptivemd.LocalResource

class `adaptivemd.`**`LocalResource`**(*shared_path=None*, *wrapper=None*)

Run tasks locally and store results in `$HOME/adaptivemd/`

`__init__`(*shared_path=None*, *wrapper=None*)

Methods

`__init__`([shared_path, wrapper])	
`args`()	Return a list of args of the *__init__* function of a class
`base`()	Return the most parent class actually derived from StorableMixin
`descendants`()	Return a list of all subclassed objects
`from_dict`(dct)	Reconstruct an object from a dictionary representation
`get_uuid`()	Create a new unique ID
`idx`(store)	Return the index which is used for the object in the given store.
`named`(name)	Attach a .name property to an object
`objects`()	Returns a dictionary of all storable objects
`to_dict`()	Convert object into a dictionary representation

Attributes

`ACTIVE_LONG`	
`CREATION_COUNT`	
`INSTANCE_UUID`	
`base_cls`	Return the base class
`base_cls_name`	Return the name of the base class
`cls`	Return the class name as a string

Files

The *`File`* object. Instead of just a string, these are used to represent files anywhere, on the cluster or your local application. There are some subclasses or *extensions* of *`File`* that have additional meta information like *`Trajectory`* or *`Frame`*. The underlying base object of a *`File`* is called a *`Location`*.

All of these objects share the `location` property. A string that represents a location for a file in general.

```
f = File('system.pdb')
```

This representation is so far useless unless we specify where this file is located. It could be on the HPC somewhere or on the local computer. To do that we use prefixes

1. `{drive}://{relative_path}` or
2. `{drive}:///{absolute_path}` (for local files)

You can use the following prefixes

- `file://` points to files on your local machine.

- `worker://` specifies files on the current working directory of the executing node. Usually these are temprary files for a single execution.
- `shared://` specifies the root shared FS directory (e.g. `NO_BACKUP/` on Allegro) Use this to import and export files that are already on the cluster.
- `staging://` a special scheduler-specific *caching* directory. Use this to relate to files that should be reused, but not stored long-time. A typical example is a PDB file. This is required by every simulation but an input file. You want to copy it once to the cluster and use it over and over.
- `sandbox://` this is a specia folder where all temporary worker directories are located. It also contains the session folders for RP.
- `project://` this folder contains all the project data for your current project and is the place where all the data should be stored for long-time storage

Later you might want to transfer a file from a project folder to the current working directory (whereever this will be) and you would specify locations in this way

```
project://models/my_model.json >> worker://input_model.json
```

We start with a first PDB file that is located on this machine at a relative path

```
pdb_file = File('file://../files/alanine/alanine.pdb')
```

`File` like any complex object in adaptivemd can have a `.name` attribute that makes them easier to find later. You can either set the `.name` property after creation, or use a little helper method `.named()` to get a one-liner. This function will set `.name` and return itself.

```
pdb_file.name = 'initial_pdb'
```

The `.load()` at the end is important. It causes the `File` object to load the content of the file and if you save the `File` object, the actual file is stored with it. This way it can simply be rewritten on the cluster or anywhere else.

```
pdb_file.load()
```

```
'alanine.pdb'
```

Now you can access the content

```
print pdb_file.get_file()[:500]
```

```
REMARK   1 CREATED WITH MDTraj 1.8.0, 2016-12-22
CRYST1   26.063   26.063   26.063  90.00  90.00  90.00 P 1           1
MODEL        0
ATOM      1  H1  ACE A   1      -1.900   1.555  26.235  1.00  0.00           H
ATOM      2  CH3 ACE A   1      -1.101   2.011  25.651  1.00  0.00           C
ATOM      3  H2  ACE A   1      -0.850   2.954  26.137  1.00  0.00           H
ATOM      4  H3  ACE A   1      -1.365   2.132  24.600  1.00  0.00           H
ATOM      5  C   ACE A   1       0.182
```

There are a few other things that you can access from a file. There is a time when it was initiated (like any storable object).

```
print 'timestamp', pdb_file.__time__
print 'uuid', hex(pdb_file.__uuid__)
```

```
timestamp 1490777436
uuid 0x5eadd73145711e7a9d3000000000042L
```

Access the drive (prefix)

```
print pdb_file.drive
```

```
file
```

Get the path on the drive (see we have converted the relative path to an absolute)

```
print '...' + pdb_file.dirname[35:]
```

```
.../adaptivemd/examples/files/alanine
```

or the basename

```
print pdb_file.basename
```

```
alanine.pdb
```

Classes

Location(location)	A representation of a path in adaptiveMD
File(location)	Represents a file object at a specific location
Trajectory(location, frame, length[, engine])	Represents a trajectory *File* on the cluster
Frame(trajectory, index)	Represents a frame of a trajectory
JSONFile(location)	A special file which as assumed JSON readable content
DataDict(data)	Delegate to the contained .data object

adaptivemd.Location

class adaptivemd.**Location**(*location*)

A representation of a path in adaptiveMD

This is an important part of adaptiveMD. It allows you to specify file paths also relative to certain special folders in adaptiveMD, like the project folder. These special paths will be interpreted by the schedulers when they actually execute tasks

Note that folder names ALWAYS end in / while filenames NEVER

You can use special prefixes

- `file://{relative}/{path}` references local files. If you want absolute paths you start with `file:///{absolute}/{path}`
- `worker://{relative_to_worker}` relative to the working directory
- `staging://` relative to staging directory
- `sandbox://` relative to the sandbox, the folder that contains worker directories
- `shared://` relative to the main shared FS folder

- `project://` relative to the specific project folder. Usually in `shared://projects/{project-name}/`

Variables **`location`** (*str*) – the full location using special prefixed

`__init__`(*location*)

Methods

`__init__`(location)	
`args`()	Return a list of args of the *__init__* function of a class
`base`()	Return the most parent class actually derived from StorableMixin
`clone`()	Make a deep copy of the objects
`descendants`()	Return a list of all subclassed objects
`from_dict`(dct)	Reconstruct an object from a dictionary representation
`get_uuid`()	Create a new unique ID
`idx`(store)	Return the index which is used for the object in the given store.
`named`(name)	Attach a .name property to an object
`objects`()	Returns a dictionary of all storable objects
`to_dict`()	Convert object into a dictionary representation

Attributes

`ACTIVE_LONG`	
`CREATION_COUNT`	
`INSTANCE_UUID`	
`allowed_drives`	
`base_cls`	Return the base class
`base_cls_name`	Return the name of the base class
`basename`	returns: the file basename
`basename_short`	returns: the basename without extension
`cls`	Return the class name as a string
`default_drive`	
`dirname`	returns: the path of the directory, like os.path.dirname
`drive`	return the prefix name
`extension`	returns: the filename extension or ‘’ of non exists
`is_folder`	returns: True if location is a folder
`is_temp`	returns: True when the location is a temporary folder that might be
`path`	returns: the complete path without prefix
`short`	returns: a shortened form of the path
`split`	returns:
`split_drive`	returns: * *str* – the drive (prefix with ://)
`url`	returns: return the full form always with a prefix
`use_absolute_local_paths`	

clone()

Make a deep copy of the objects

Returns the deep copy

Return type *Location*

is_temp

Returns True when the location is a temporary folder that might be deleted

Return type bool

short

Returns a shortened form of the path

Return type str

url

Returns return the full form always with a prefix

Return type str

basename

Returns the file basename

Return type str

is_folder

Returns True if location is a folder

Return type bool

path

Returns the complete path without prefix

Return type str

split

Returns

Return type os.path.split on the `path` without prefixes

dirname

Returns the path of the directory, like os.path.dirname

Return type str

drive

return the prefix name

Returns the prefix name like *staging*, *project*, *worker*, file‘

Return type str

extension

Returns the filename extension or ‘’ of non exists

Return type str

basename_short

Returns the basename without extension

Return type str

split_drive

Returns

- *str* – the drive (prefix with ://)
- *str* – the full path without prefix

adaptivemd.File

class adaptivemd.**File**(*location*)

Represents a file object at a specific location

File objects can but do not have to exist - you can check using the `File.created` attribute. If it is a positive number it represents the time stamp when it was created.

__init__(*location*)

Methods

__init__(location)	
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
clone()	create a cloned object with equal attributes
copy([target])	copy file to a target
create(scheduler)	Mark file as being existent on a specific scheduler.
descendants()	Return a list of all subclassed objects
from_dict(dct)	
get_file()	Return the file content it has been loaded
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
link([target])	link file to a target
load([scheduler])	Load a local file into memory
modified()	Mark a file as being altered and not existent anymore
move([target])	move file to a target
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
remove()	remove file
set_file(content)	Set the file content.
to_dict()	
touch()	touch file
transfer([target])	transfer file to a target

Attributes

ACTIVE_LONG
CREATION_COUNT

Continued on next page

Table 1.10 – continued from previous page

`INSTANCE_UUID`	
`allowed_drives`	
`base_cls`	Return the base class
`base_cls_name`	Return the name of the base class
`basename`	returns: the file basename
`basename_short`	returns: the basename without extension
`cls`	Return the class name as a string
`created`	
`default_drive`	
`dirname`	returns: the path of the directory, like os.path.dirname
`drive`	return the prefix name
`exists`	returns: True if the file exists, i.e. has a positive *created* timestamp
`extension`	returns: the filename extension or ‘’ of non exists
`generator`	
`has_file`	returns: True if the file content is attached.
`is_folder`	returns: True if location is a folder
`is_temp`	returns: True when the location is a temporary folder that might be
`path`	returns: the complete path without prefix
`short`	returns: a shortened form of the path
`split`	returns:
`split_drive`	returns: * *str* – the drive (prefix with ://)
`task`	
`url`	returns: return the full form always with a prefix
`use_absolute_local_paths`	

clone()

create a cloned object with equal attributes

Returns the same type as this object

Return type *Location*

create(*scheduler*)

Mark file as being existent on a specific scheduler.

This should only work for file in `staging://`, `shared://`, `sandbox://` or `file://` Files in `worker://` will potentially be deleted, others are already existing

Notes

We usually assume that objects are immutable. The way to think about creation is that a file is something like a *Promise* and it promises a certain file with a name. Once it is created it is still the same file but now it exists and can be used.

The change of location is also a re-expression of the same location so that it is reusable.

modified()

Mark a file as being altered and not existent anymore

Notes

Negative timestamps indicate the (negative) time when the object disappeared in the form described

exists

Returns True if the file exists, i.e. has a positive *created* timestamp

Return type bool

copy(*target=None*)

copy file to a target

Shortcut for `Copy(self, target)`

Parameters **target** (*Location* or str) – the target location

Returns the copy action

Return type *adaptivemd.FileTransaction*

move(*target=None*)

move file to a target

Shortcut for `Move(self, target)`

Parameters **target** (*Location* or str) – the target location

Returns the move action

Return type *adaptivemd.FileTransaction*

link(*target=None*)

link file to a target

Shortcut for `Link(self, target)`

Parameters **target** (*Location* or str) – the target location

Returns the link action

Return type *adaptivemd.FileTransaction*

transfer(*target=None*)

transfer file to a target

Shortcut for *Transfer(self, target)*

Parameters **target** (*Location* or str) – the target location

Returns the transfer action

Return type *adaptivemd.FileTransaction*

remove()

remove file

Shortcut for *Remove(self)*

Returns the remove action

Return type *adaptivemd.FileAction*

touch()

touch file

Shortcut for *Touch(self)*

Returns the touch action

Return type *adaptivemd.FileAction*

load(*scheduler=None*)
Load a local file into memory

If you later store the file its content will be stored as well

Parameters **scheduler** (*Scheduler* or None) – if specifiied the scheduler can alter the filelocation with its usual rules. Normally you should not have to use it

Returns

Return type self

get_file()
Return the file content it has been loaded

Returns the file content, if it exists None else

Return type str or None

has_file

Returns True if the file content is attached.

Return type bool

set_file(*content*)
Set the file content.

Can only be set once!

Parameters **content** (*str*) – the content of the file

adaptivemd.Trajectory

class adaptivemd.**Trajectory**(*location*, *frame*, *length*, *engine=None*)
Represents a trajectory *File* on the cluster

Variables

- **location** (str or *File*) – the *File* location
- **frame** (*Frame* or *File*) – the initial frame used for the trajectory
- **length** (*int*) – the length of the trajectory in frames
- **engine** (*Engine*) – the engine used to create the trajectory

__init__(*location*, *frame*, *length*, *engine=None*)

Methods

__init__(location, frame, length[, engine])	
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
clone()	
copy([target])	copy file to a target
create(scheduler)	Mark file as being existent on a specific scheduler.

Continued on next page

Table 1.11 – continued from previous page

descendants()	Return a list of all subclassed objects
extend(length)	Get a task to extend this trajectory if the engine is set
file(f)	Return a file location to a file inside the trajectory folder
from_dict(dct)	
get_file()	Return the file content it has been loaded
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
link([target])	link file to a target
load([scheduler])	Load a local file into memory
modified()	Mark a file as being altered and not existent anymore
move([target])	move file to a target
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
outputs(outtype)	Get a location to the file containing the output by given name
pick()	Return a random frame from all possible full frames
remove()	remove file
run()	Return a task to run this engine
set_file(content)	Set the file content.
to_dict()	
touch()	touch file
transfer([target])	transfer file to a target

Attributes

ACTIVE_LONG	
CREATION_COUNT	
INSTANCE_UUID	
allowed_drives	
base_cls	Return the base class
base_cls_name	Return the name of the base class
basename	returns: the file basename
basename_short	returns: the basename without extension
cls	Return the class name as a string
created	
default_drive	
dirname	returns: the path of the directory, like os.path.dirname
drive	return the prefix name
engine	
existing_frames	returns: a sorted list of frame indices with full coordinates that can be
exists	returns: True if the file exists, i.e. has a positive *created* timestamp
extension	returns: the filename extension or ‘’ of non exists
generator	
has_file	returns: True if the file content is attached.
is_folder	

Continued on next page

Table 1.12 – continued from previous page

is_temp	returns: True when the location is a temporary folder that might be
path	returns: the complete path without prefix
short	returns: a shortened form of the path
split	returns:
split_drive	returns: * *str* – the drive (prefix with ://)
task	
types	Return the OutputTypeDescriptions for this trajectory
url	returns: return the full form always with a prefix
use_absolute_local_paths	

pick()

Return a random frame from all possible full frames

Returns the frame you can restart from

Return type *Frame*

file(*f*)

Return a file location to a file inside the trajectory folder

Parameters **f** (str or *OutputTypeDescription*) – the filename to be appended to the trajectories directory

Returns the object containing the location

Return type *File*

run()

Return a task to run this engine

Returns the task object that can be submitted to the queue

Return type *Task*

extend(*length*)

Get a task to extend this trajectory if the engine is set

Parameters **length** (*int or list of int*) – the length to extend by as a single int or a list of ints

Returns the task object to extend the trajectory

Return type *Task*

outputs(*outtype*)

Get a location to the file containing the output by given name

Parameters **outtype** (str or *OutputTypeDescription*) – the name of the outputtype as str or the full description object

Returns a file location that points to the concrete file that contains the data for a particular output type

Return type *File*

types

Return the OutputTypeDescriptions for this trajectory

Returns **dict str** – the output description dict of the engine

Return type *OutputTypeDescription*

existing_frames

Returns a sorted list of frame indices with full coordinates that can be used for restart. relative to the engines timesteps

Return type list of int

adaptivemd.Frame

class adaptivemd.**Frame** (*trajectory*, *index*)

Represents a frame of a trajectory

Variables

- **trajectory** (*Trajectory*) – the origin trajectory
- **index** (*int*) – the frame index staring from zero

__init__ (*trajectory*, *index*)

Methods

__init__(trajectory, index)	
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
from_dict(dct)	Reconstruct an object from a dictionary representation
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
to_dict()	Convert object into a dictionary representation

Attributes

ACTIVE_LONG	
CREATION_COUNT	
INSTANCE_UUID	
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string
exists	returns: if True there is a concrete trajectory file with full
index_in_outputs	Return output type and effective frame index for this frame

index_in_outputs

Return output type and effective frame index for this frame

Returns

- *str* – the name of the output type
- *int* – the effective index within this trajectory obeying the trajectories own stride

exists

Returns if True there is a concrete trajectory file with full coordinates for this frame

Return type bool

adaptivemd.JSONFile

class adaptivemd.**JSONFile**(*location*)

A special file which as assumed JSON readable content

__init__(*location*)

Methods

__init__(location)	
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
clone()	create a cloned object with equal attributes
copy([target])	copy file to a target
create(scheduler)	Mark file as being existent on a specific scheduler.
descendants()	Return a list of all subclassed objects
from_dict(dct)	
get([scheduler])	Read data from the JSON file at the files location without storing
get_file()	
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
link([target])	link file to a target
load([scheduler])	
modified()	Mark a file as being altered and not existent anymore
move([target])	move file to a target
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
remove()	remove file
set_file(content)	Set the file content.
to_dict()	
touch()	touch file
transfer([target])	transfer file to a target

Attributes

ACTIVE_LONG
CREATION_COUNT
INSTANCE_UUID

Continued on next page

Table 1.16 – continued from previous page

allowed_drives	
base_cls	Return the base class
base_cls_name	Return the name of the base class
basename	returns: the file basename
basename_short	returns: the basename without extension
cls	Return the class name as a string
created	
data	returns: the parsed JSON content
default_drive	
dirname	returns: the path of the directory, like os.path.dirname
drive	return the prefix name
exists	
extension	returns: the filename extension or ‘’ of non exists
generator	
has_file	
is_folder	returns: True if location is a folder
is_temp	returns: True when the location is a temporary folder that might be
path	returns: the complete path without prefix
short	returns: a shortened form of the path
split	returns:
split_drive	returns: * *str* – the drive (prefix with ://)
task	
url	returns: return the full form always with a prefix
use_absolute_local_paths	

data

Returns the parsed JSON content

Return type dict

get(*scheduler=None*)

Read data from the JSON file at the files location without storing

Parameters **scheduler** (*Scheduler* or None) – if given use the prefixing from the scheduler

Returns the data in the file

Return type dict

adaptivemd.mongodb.DataDict

class adaptivemd.mongodb.**DataDict**(*data*)

Delegate to the contained .data object

__init__(*data*)

Methods

__init__(data)	

Continued on next page

Table 1.17 – continued from previous page

args()	Return a list of args of the __*init*__ function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
from_dict(dct)	Reconstruct an object from a dictionary representation
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
to_dict()	Convert object into a dictionary representation

Attributes

ACTIVE_LONG	
CREATION_COUNT	
INSTANCE_UUID	
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string

Bundles

A *Bundle* - A set-enhancement to add filtering and store handling capabilities

Bundles can be accessed like a normal set using iteration. You can add objects using `.add(item)` if the bundle is not a view

Examples

Some basic functions

```
bundle = Bundle(['10', '20', 1, 2, 3])
str_view = bundle.c(basestring)  # only how strings
print list(str_view)  # ['10', '20']
fnc_view = bundle.v(lambda x: int(x) < 3)
print list(fnc_view) # [1, 2]
```

Some *File* specific functions

```
import adaptivemd as amd
bundle = Bundle([amd.File('0.dcd'), amd.File('a.pdb')])
file_view = bundle.f('*.dcd')
print list(file_view)  # [File('0.dcd')]
```

Logic operations produce view on the resulting bundle

```
and_bundle = str_view & fnc_view
print list(and_bundle)  # []
and_bundle = str_view | fnc_view
print list(and_bundle)  # [1, 2, '10', '20']
```

A `StoredBundle` is attached to a mongodb store (a stored object list). Adding will append the object to the store if not stored yet. All iteration and views will always be kept synced with the DB store content.

```
p = amd.Project('test-project')
store = StoredBundle()  # new bundle
store.set_store(p.storage.trajectories)  # attach to DB
print list(store)  # show all trajectories
len_store = store.v(lambda x: len(x) > 10)  # all trajs with len > 10
print list(len_store)
```

Set do not have ordering so some functions do not make sense. As long as you are working with storable objects (subclassed from `adaptivemd.mongodb.StorableMixin`) you have some time-ordering (accurate to seconds)

```
print store.first  # get the earliest created object
print store.one    # get one (any) single object
print store.last   # get the last created object
```

A bundle is mostly meant to work with storable objects (but does not have to) To simplify access to certain attributes or apply function to all members you can use the `BaseBundle.all()` attribute and get a *delegator* that will apply an attribute or method to all objects

```
print len_store.all.length  # print all lengths of all objects in len_store
print store.all.path  # print all path of all trajectories
# call `.execute('shutdown') on all workers in the `.workers` bundle
print p.workers.all.execute('shutdown')
```

Classes

`Bundle`([iterable])	A container of objects
`StoredBundle`()	A stored bundle in a mongodb
`SortedBundle`(bundle, key)	Sorted view of a bundle
`ViewBundle`(bundle, view)	A view on a bundle where object are filtered by a bool function
`BaseBundle`	BaseClass for Bundle functionality a special set of storable objects
`LogicBundle`(bundle1, bundle2)	Implement simple and and or logic for bundles
`AndBundle`(bundle1, bundle2)	And logic
`OrBundle`(bundle1, bundle2)	Or logic
`BundleDelegator`(bundle)	Delegate an attribute call to all elements in a bundle
`FunctionDelegator`(bundle, item)	Delegate a function call to all elements in a bundle

adaptivemd.Bundle

class `adaptivemd.`**`Bundle`**(*iterable=None*)

A container of objects

`__init__`(*iterable=None*)

Methods

`__init__`([iterable])	
`add`(item)	Add a single item to the bundle
`c`(cls)	Return a view bundle on all entries that are instances of a class
`f`(pattern)	Return a view bundle on all entries that match a location pattern
`pick`()	Pick a random element
`sorted`(key)	Return a view bundle where all entries are sorted by a given key attribute
`update`(iterable)	Add multiple items to the bundle at once
`v`(fnc)	Return a view bundle on all entries that are filtered by a function

Attributes

`all`	Return a Delegator that will apply attribute and function call to all bundle elements
`one`	Return one element from the list

update (*iterable*)
Add multiple items to the bundle at once

Parameters **iterable** (*Iterable*) – the items to be added

add (*item*)
Add a single item to the bundle

Parameters **item** (*object*) –

adaptivemd.StoredBundle

class adaptivemd.**StoredBundle**
A stored bundle in a mongodb

This is a useful wrapper to turn a store of the MongoDB into a bundle of objects. Adding files will store new elements. The bundle is always in sync with the DB.

__init__ ()

Methods

`__init__`()	
`add`(item)	Add an element to the bundle
`c`(cls)	Return a view bundle on all entries that are instances of a class
`close`()	Close the connection to the bundle.
`consume_one`()	Picks and removes one (random) element in one step.
`f`(pattern)	Return a view bundle on all entries that match a location pattern

Continued on next page

Table 1.22 – continued from previous page

find_all_by(key, value)	Return all elements from the bundle where its key matches value
pick()	Pick a random element
set_store(store)	Set the used store
sorted(key)	Return a view bundle where all entries are sorted by a given key attribute
update(iterable)	Add multiple items to the bundle at once
v(fnc)	Return a view bundle on all entries that are filtered by a function

Attributes

all	Return a Delegator that will apply attribute and function call to all bundle elements
first	Return the entry with the earliest timestamp
last	Return the entry with the latest timestamp
one	Return one element from the list

set_store(*store*)

Set the used store

Parameters **store** (*ObjectStore*) – a mongodb store that contains the elements in the bundle

close()

Close the connection to the bundle.

A not connected bundle will have no entries and none can be added

add(*item*)

Add an element to the bundle

Parameters **item** (*object*) – the item to be added to the bundle

last

Return the entry with the latest timestamp

Returns the latest object

Return type object

first

Return the entry with the earliest timestamp

Returns the earliest object

Return type object

consume_one()

Picks and removes one (random) element in one step.

Returns The deleted object if possible otherwise None

Return type *StorableMixin* or None

find_all_by(*key*, *value*)

Return all elements from the bundle where its key matches value

Parameters

- **key** (*str*) – the attribute
- **value** (*object*) – the value to match against using ==

Returns a list of objects in the bundle that match the search

Return type list of *StorableMixin*

adaptivemd.SortedBundle

class adaptivemd.**SortedBundle**(*bundle*, *key*)

Sorted view of a bundle

__init__(*bundle*, *key*)

Methods

__init__(bundle, key)	
c(cls)	Return a view bundle on all entries that are instances of a class
f(pattern)	Return a view bundle on all entries that match a location pattern
pick()	Pick a random element
sorted(key)	Return a view bundle where all entries are sorted by a given key attribute
v(fnc)	Return a view bundle on all entries that are filtered by a function

Attributes

all	Return a Delegator that will apply attribute and function call to all bundle elements
first	object
one	Return one element from the list

first

object Return the first of the sorted elements

adaptivemd.ViewBundle

class adaptivemd.**ViewBundle**(*bundle*, *view*)

A view on a bundle where object are filtered by a bool function

__init__(*bundle*, *view*)

Methods

__init__(bundle, view)	

Continued on next page

Table 1.26 – continued from previous page

`c`(cls)	Return a view bundle on all entries that are instances of a class
`f`(pattern)	Return a view bundle on all entries that match a location pattern
`pick`()	Pick a random element
`sorted`(key)	Return a view bundle where all entries are sorted by a given key attribute
`v`(fnc)	Return a view bundle on all entries that are filtered by a function

Attributes

`all`	Return a Delegator that will apply attribute and function call to all bundle elements
`one`	Return one element from the list

adaptivemd.BaseBundle

class `adaptivemd.`**`BaseBundle`**

BaseClass for Bundle functionality a special set of storable objects

`__init__``()`

x.__init__(...) initializes x; see help(type(x)) for signature

Methods

`c`(cls)	Return a view bundle on all entries that are instances of a class
`f`(pattern)	Return a view bundle on all entries that match a location pattern
`pick`()	Pick a random element
`sorted`(key)	Return a view bundle where all entries are sorted by a given key attribute
`v`(fnc)	Return a view bundle on all entries that are filtered by a function

Attributes

`all`	Return a Delegator that will apply attribute and function call to all bundle elements
`one`	Return one element from the list

`c` (*cls*)

Return a view bundle on all entries that are instances of a class

Parameters **`cls`** (*type*) – a class to be filtered by

Returns the read-only bundle showing filtered entries

Return type *ViewBundle*

`f`(*pattern*)

Return a view bundle on all entries that match a location pattern

Works only when all objects are of type *File*

Parameters **`pattern`** (*`str`*) – a string CL pattern using wildcards to match a filename

Returns the read-only bundle showing filtered entries

Return type *ViewBundle*

`sorted`(*key*)

Return a view bundle where all entries are sorted by a given key attribute

Parameters **`key`** (*`function`*) – a function to compute the key to be sorted by

Returns the read-only bundle showing sorted entries

Return type *ViewBundle*

`v`(*fnc*)

Return a view bundle on all entries that are filtered by a function

Parameters **`fnc`** (*`function`*) – a function to be used for filtering

Returns the read-only bundle showing filtered entries

Return type *ViewBundle*

`pick`()

Pick a random element

Returns a random object if bundle is not empty

Return type object or None

`one`

Return one element from the list

Use only if you just need one and do not care which one it is

Returns one object (there is no guarantee that this will always be the same element)

Return type object

`all`

Return a Delegator that will apply attribute and function call to all bundle elements

Returns the delegator object to map to all elements in the bundle

Return type *BundleDelegator*

adaptivemd.LogicBundle

class `adaptivemd.LogicBundle`(*bundle1*, *bundle2*)

Implement simple and and or logic for bundles

`__init__`(*bundle1*, *bundle2*)

Methods

`__init__`(bundle1, bundle2)	
`c`(cls)	Return a view bundle on all entries that are instances of a class
`f`(pattern)	Return a view bundle on all entries that match a location pattern
`pick`()	Pick a random element
`sorted`(key)	Return a view bundle where all entries are sorted by a given key attribute
`v`(fnc)	Return a view bundle on all entries that are filtered by a function

Attributes

`all`	Return a Delegator that will apply attribute and function call to all bundle elements
`one`	Return one element from the list

adaptivemd.AndBundle

class `adaptivemd.`**`AndBundle`**(*bundle1*, *bundle2*)

And logic

`__init__`(*bundle1*, *bundle2*)

Methods

`__init__`(bundle1, bundle2)	
`c`(cls)	Return a view bundle on all entries that are instances of a class
`f`(pattern)	Return a view bundle on all entries that match a location pattern
`pick`()	Pick a random element
`sorted`(key)	Return a view bundle where all entries are sorted by a given key attribute
`v`(fnc)	Return a view bundle on all entries that are filtered by a function

Attributes

`all`	Return a Delegator that will apply attribute and function call to all bundle elements
`one`	Return one element from the list

adaptivemd.OrBundle

class `adaptivemd.`**`OrBundle`**(*bundle1*, *bundle2*)

Or logic

__init__(*bundle1*, *bundle2*)

Methods

`__init__`(bundle1, bundle2)	
`c`(cls)	Return a view bundle on all entries that are instances of a class
`f`(pattern)	Return a view bundle on all entries that match a location pattern
`pick`()	Pick a random element
`sorted`(key)	Return a view bundle where all entries are sorted by a given key attribute
`v`(fnc)	Return a view bundle on all entries that are filtered by a function

Attributes

`all`	Return a Delegator that will apply attribute and function call to all bundle elements
`one`	Return one element from the list

adaptivemd.BundleDelegator

class `adaptivemd.`**`BundleDelegator`**(*bundle*)

Delegate an attribute call to all elements in a bundle

__init__(*bundle*)

Methods

`__init__`(bundle)	

adaptivemd.FunctionDelegator

class `adaptivemd.`**`FunctionDelegator`**(*bundle*, *item*)

Delegate a function call to all elements in a bundle

__init__(*bundle*, *item*)

Methods

`__init__`(bundle, item)	

Actions

Actions are descriptions for executions on the HPC - basically a bash command

Classes

Action()	A bash-command-like action to be executed in a Task
FileAction(source)	An Action that involves (at least) one file called source
FileTransaction(source, target)	An action involving a source and a target file
Copy(source, target)	An action that copies a file from source to target
Move(source, target)	An action that moves a file from source to target
Link(source, target)	An action that links a source file to a target
Touch(source)	An action that creates an empty file or folder
Remove(source)	An action that removes a file
MakeDir(source)	An action that creates a folder

adaptivemd.Action

class adaptivemd.**Action**

A bash-command-like action to be executed in a Task

The main purpose is to have a worker/hpc independent description of what should happen. This objects carry all the necessary information and will be parsed into a bash script on the actual HPC / worker

__init__()

Methods

__init__()	
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
from_dict(dct)	Reconstruct an object from a dictionary representation
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
to_dict()	Convert object into a dictionary representation

Attributes

ACTIVE_LONG	
CREATION_COUNT	
INSTANCE_UUID	
base_cls	Return the base class

Continued on next page

Table 1.40 – continued from previous page

base_cls_name	Return the name of the base class
cls	Return the class name as a string

adaptivemd.FileAction

class adaptivemd.**FileAction**(*source*)

An Action that involves (at least) one file called source

Variables source (*File*) – the source file for the action

__init__(*source*)

Methods

__init__(source)	
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
from_dict(dct)	Reconstruct an object from a dictionary representation
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
to_dict()	Convert object into a dictionary representation

Attributes

ACTIVE_LONG	
CREATION_COUNT	
INSTANCE_UUID	
added	returns: the list of files added to the project by this action
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string
removed	returns: the list of files removed by this action
required	returns: the necessary list of files to be functional

required

Returns the necessary list of files to be functional

Return type list of *File*

added

Returns the list of files added to the project by this action

Return type list of *File*

removed

Returns the list of files removed by this action

Return type list of *File*

adaptivemd.FileTransaction

class adaptivemd.**FileTransaction**(*source*, *target*)

An action involving a source and a target file

Variables **target** (*File*) – the target file

Parameters

- **source** (*File*) – the source file for the action
- **target** (*File* or *Location* or str) – the target location for the action

__init__(*source*, *target*)

Parameters

- **source** (*File*) – the source file for the action
- **target** (*File* or *Location* or str) – the target location for the action

Methods

__init__(source, target)	**param source** the source file for the action
args()	Return a list of args of the __init__ function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
from_dict(dct)	Reconstruct an object from a dictionary representation
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
to_dict()	Convert object into a dictionary representation

Attributes

ACTIVE_LONG	
CREATION_COUNT	
INSTANCE_UUID	
added	
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string
removed	returns: the list of files removed by this action
required	returns: the necessary list of files to be functional

__init__(*source*, *target*)

Parameters

- **source** (*File*) – the source file for the action
- **target** (*File* or *Location* or str) – the target location for the action

adaptivemd.Copy

class adaptivemd.**Copy**(*source*, *target*)

An action that copies a file from source to target

Parameters

- **source** (*File*) – the source file for the action
- **target** (*File* or *Location* or str) – the target location for the action

__init__(*source*, *target*)

Parameters

- **source** (*File*) – the source file for the action
- **target** (*File* or *Location* or str) – the target location for the action

Methods

__init__(source, target)	**param source** the source file for the action
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
from_dict(dct)	Reconstruct an object from a dictionary representation
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
to_dict()	Convert object into a dictionary representation

Attributes

ACTIVE_LONG	
CREATION_COUNT	
INSTANCE_UUID	
added	
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string

Continued on next page

Table 1.46 – continued from previous page

`removed`	returns: the list of files removed by this action
`required`	returns: the necessary list of files to be functional

adaptivemd.Move

class `adaptivemd.`**`Move`**(*source*, *target*)

An action that moves a file from source to target

The source is removed in the process

Parameters

- **`source`** (*File*) – the source file for the action
- **`target`** (*File* or *Location* or str) – the target location for the action

`__init__`(*source*, *target*)

Parameters

- **`source`** (*File*) – the source file for the action
- **`target`** (*File* or *Location* or str) – the target location for the action

Methods

`__init__`(source, target)	**param source** the source file for the action
`args`()	Return a list of args of the *__init__* function of a class
`base`()	Return the most parent class actually derived from StorableMixin
`descendants`()	Return a list of all subclassed objects
`from_dict`(dct)	Reconstruct an object from a dictionary representation
`get_uuid`()	Create a new unique ID
`idx`(store)	Return the index which is used for the object in the given store.
`named`(name)	Attach a .name property to an object
`objects`()	Returns a dictionary of all storable objects
`to_dict`()	Convert object into a dictionary representation

Attributes

`ACTIVE_LONG`	
`CREATION_COUNT`	
`INSTANCE_UUID`	
`added`	
`base_cls`	Return the base class
`base_cls_name`	Return the name of the base class
`cls`	Return the class name as a string
`removed`	
`required`	returns: the necessary list of files to be functional

adaptivemd.Link

class `adaptivemd.`**`Link`**(*source*, *target*)

An action that links a source file to a target

Parameters

- **`source`** (*File*) – the source file for the action
- **`target`** (*File* or *Location* or str) – the target location for the action

`__init__`(*source*, *target*)

Parameters

- **`source`** (*File*) – the source file for the action
- **`target`** (*File* or *Location* or str) – the target location for the action

Methods

`__init__`(source, target)	**param source** the source file for the action
`args`()	Return a list of args of the *`__init__`* function of a class
`base`()	Return the most parent class actually derived from StorableMixin
`descendants`()	Return a list of all subclassed objects
`from_dict`(dct)	Reconstruct an object from a dictionary representation
`get_uuid`()	Create a new unique ID
`idx`(store)	Return the index which is used for the object in the given store.
`named`(name)	Attach a .name property to an object
`objects`()	Returns a dictionary of all storable objects
`to_dict`()	Convert object into a dictionary representation

Attributes

`ACTIVE_LONG`	
`CREATION_COUNT`	
`INSTANCE_UUID`	
`added`	
`base_cls`	Return the base class
`base_cls_name`	Return the name of the base class
`cls`	Return the class name as a string
`removed`	returns: the list of files removed by this action
`required`	returns: the necessary list of files to be functional

adaptivemd.Touch

class `adaptivemd.`**`Touch`**(*source*)

An action that creates an empty file or folder

__init__(*source*)

Methods

__init__(source)	
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
from_dict(dct)	Reconstruct an object from a dictionary representation
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
to_dict()	Convert object into a dictionary representation

Attributes

ACTIVE_LONG	
CREATION_COUNT	
INSTANCE_UUID	
added	returns: the list of files added to the project by this action
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string
removed	returns: the list of files removed by this action
required	returns: the necessary list of files to be functional

adaptivemd.Remove

class adaptivemd.**Remove**(*source*)

An action that removes a file

__init__(*source*)

Methods

__init__(source)	
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
from_dict(dct)	Reconstruct an object from a dictionary representation
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.

Continued on next page

Table 1.53 – continued from previous page

named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
to_dict()	Convert object into a dictionary representation

Attributes

ACTIVE_LONG	
CREATION_COUNT	
INSTANCE_UUID	
added	
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string
removed	
required	returns: the necessary list of files to be functional

adaptivemd.MakeDir

class adaptivemd.**MakeDir**(*source*)

An action that creates a folder

__init__(*source*)

Methods

__init__(source)	
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
from_dict(dct)	Reconstruct an object from a dictionary representation
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
to_dict()	Convert object into a dictionary representation

Attributes

ACTIVE_LONG	
CREATION_COUNT	
INSTANCE_UUID	
added	returns: the list of files added to the project by this action
base_cls	Return the base class
base_cls_name	Return the name of the base class

Continued on next page

Table 1.56 – continued from previous page

cls	Return the class name as a string
removed	returns: the list of files removed by this action
required	returns: the necessary list of files to be functional

Task

A *Task* is in essence a bash script-like description of what should be executed by the worker. It has details about files to be linked to the working directory, bash commands to be executed and some meta information about what should happen in case we succeed or fail.

The execution structure

Let's first explain briefly how a task is executed and what its components are. This was originally build so that it is compatible with radical.pilot and still is. So, if you are familiar with it, all of the following information should sould very familiar.

A task is executed from within a unique directory that only exists for this particular task. These are located in `adaptivemd/workers/` and look like

```
worker.0x5dcccd05097611e7829b000000000072L/
```

the long number is a hex representation of the UUID of the task. Just if you are curious type

```
print hex(my_task.__uuid__)
```

Then we change directory to this folder write a `running.sh` bash script and execute it. This script is created from the task definition and also depends on your resource setting (which basically only contain the path to the workers directory, etc)

The script is divided into 1 or 3 parts depending on which `Task` class you use. The main `Task` uses a single list of commands, while `PrePostTask` has the following structure

1. **Pre-Exec**: Things to happen before the main command (optional)
2. **Main**: the main commands are executed
3. **Post-Exec**: Things to happen after the main command (optional)

Okay, lots of theory, now some real code for running a task that generated a trajectory

```
task = engine.task_run_trajectory(project.new_trajectory(pdb_file, 100))
```

```
task.script
```

```
[Link('staging:///alanine.pdb' > 'worker://initial.pdb),
 Link('staging:///system.xml' > 'worker://system.xml),
 Link('staging:///integrator.xml' > 'worker://integrator.xml),
 Link('staging:///openmmrun.py' > 'worker://openmmrun.py),
 Touch('worker://traj/'),
 'python openmmrun.py -r --report-interval 1 -p CPU --store-interval 1 -t worker://
↪initial.pdb --length 100 worker://traj/',
 Move('worker://traj/' > 'sandbox:///{}/00000076/)]
```

We are linking a lot of files to the worker directory and change the name for the .pdb in the process. Then call the actual `python` script that runs openmm. And finally move the `output.dcd` and the restart file back tp the trajectory folder.

There is a way to list lot's of things about tasks and we will use it a lot to see our modifications.

```
print task.description
```

```
Task: TrajectoryGenerationTask(OpenMMEngine) [created]

Sources
- staging:///integrator.xml
- staging:///alanine.pdb
- staging:///openmmrun.py
- staging:///system.xml
Targets
- sandbox:///{}/00000076/
Modified

<pretask>
Link('staging:///alanine.pdb' > 'worker://initial.pdb)
Link('staging:///system.xml' > 'worker://system.xml)
Link('staging:///integrator.xml' > 'worker://integrator.xml)
Link('staging:///openmmrun.py' > 'worker://openmmrun.py)
Touch('worker://traj/')
python openmmrun.py -r --report-interval 1 -p CPU --store-interval 1 -t worker://
↪initial.pdb --length 100 worker://traj/
Move('worker://traj/' > 'sandbox:///{}/00000076/)
<posttask>
```

Modify a task

As long as a task is not saved and hence placed in the queue, it can be altered in any way. All of the 3 / 5 phases can be changed separately. You can add things to the staging phases or bash phases or change the command. So, let's do that now

Add a bash line

First, a `Task` is very similar to a list of bash commands and you can simply append (or prepend) a command. A text line will be interpreted as a bash command.

```
task.append('echo "This new line is pointless"')
```

```
print task.description
```

```
Task: TrajectoryGenerationTask(OpenMMEngine) [created]

Sources
- staging:///integrator.xml
- staging:///alanine.pdb
- staging:///openmmrun.py
- staging:///system.xml
Targets
- sandbox:///{}/00000076/
```

```
Modified

<pretask>
Link('staging:///alanine.pdb' > 'worker://initial.pdb)
Link('staging:///system.xml' > 'worker://system.xml)
Link('staging:///integrator.xml' > 'worker://integrator.xml)
Link('staging:///openmmrun.py' > 'worker://openmmrun.py)
Touch('worker://traj/')
python openmmrun.py -r --report-interval 1 -p CPU --store-interval 1 -t worker://
↪initial.pdb --length 100 worker://traj/
Move('worker://traj/' > 'sandbox:///{}/00000076/)
echo "This new line is pointless"
<posttask>
```

As expected this line was added to the end of the script.

Add staging actions

To set staging is more difficult. The reason is, that you normally have no idea where files are located and hence writing a copy or move is impossible. This is why the staging commands are not bash lines but objects that hold information about the actual file transaction to be done. There are some task methods that help you move files but also files itself can generate this commands for you.

Let's move one trajectory (directory) around a little more as an example

```
traj = project.trajectories.one
```

```
transaction = traj.copy()
print transaction
```

```
Copy('sandbox:///{}/00000010/' > 'worker://)
```

This looks like in the script. The default for a copy is to move a file or folder to the worker directory under the same name, but you can give it another name/location if you use that as an argument. Note that since trajectories are a directory you need to give a directory name (which end in a /)

```
transaction = traj.copy('new_traj/')
print transaction
```

```
Copy('sandbox:///{}/00000010/' > 'worker://new_traj/)
```

If you want to move it not to the worker directory you have to specify the location and you can do so with the prefixes (`shared://`, `sandbox://`, `staging://` as explained in the previous examples)

```
transaction = traj.copy('staging:///cached_trajs/')
print transaction
```

```
Copy('sandbox:///{}/00000010/' > 'staging:///cached_trajs/)
```

Besides `.copy` you can also `.move` or `.link` files.

```
transaction = pdb_file.copy('staging:///delete.pdb')
print transaction
transaction = pdb_file.move('staging:///delete.pdb')
print transaction
```

```
transaction = pdb_file.link('staging:///delete.pdb')
print transaction
```

```
Copy('file://{}/alanine.pdb' > 'staging:///delete.pdb)
Move('file://{}/alanine.pdb' > 'staging:///delete.pdb)
Link('file://{}/alanine.pdb' > 'staging:///delete.pdb)
```

Local files

Let's mention these because they require special treatment. We cannot (like RP can) copy files to the HPC, we need to store them in the DB first.

```
new_pdb = File('file://../files/ntl9/ntl9.pdb').load()
```

Make sure you use `file://` to indicate that you are using a local file. The above example uses a relative path which will be replaced by an absolute one, otherwise we ran into trouble once we open the project at a different directory.

```
print new_pdb.location
```

```
file:///Users/jan-hendrikprinz/Studium/git/adaptivemd/examples/files/ntl9/ntl9.pdb
```

Note that now there are 3 `/` in the filename, two from the `://` and one from the root directory of your machine

The `load()` at the end really loads the file and when you save this `File` now it will contain the content of the file. You can access this content as seen in the previous example.

```
print new_pdb.get_file()[:300]
```

```
CRYST1   50.000   50.000   50.000  90.00  90.00  90.00 P 1
ATOM      1  N   MET     1      33.720  28.790  34.120  0.00  0.00           N
ATOM      2  H1  MET     1      33.620  29.790  33.900  0.00  0.00           H
ATOM      3  H2  MET     1      33.770  28.750  35.120  0.00  0.00
```

For local files you normally use `.transfer`, but `copy`, `move` or `link` work as well. Still, there is no difference since the file only exists in the DB now and copying from the DB to a place on the HPC results in a simple file creation.

Now, we want to add a command to the staging and see what happens.

```
transaction = new_pdb.transfer()
print transaction
```

```
Transfer('file://{}/ntl9.pdb' > 'worker://ntl9.pdb)
```

```
task.append(transaction)
```

```
print task.description
```

```
Task: TrajectoryGenerationTask(OpenMMEngine) [created]

Sources
- staging:///integrator.xml
- staging:///alanine.pdb
- staging:///openmmrun.py
- file://{}/ntl9.pdb [exists]
- staging:///system.xml
Targets
- sandbox:///{}/00000076/
Modified

<pretask>
Link('staging:///alanine.pdb' > 'worker://initial.pdb)
Link('staging:///system.xml' > 'worker://system.xml)
Link('staging:///integrator.xml' > 'worker://integrator.xml)
Link('staging:///openmmrun.py' > 'worker://openmmrun.py)
Touch('worker://traj/')
python openmmrun.py -r --report-interval 1 -p CPU --store-interval 1 -t worker://
↪initial.pdb --length 100 worker://traj/
Move('worker://traj/' > 'sandbox:///{}/00000076/)
echo "This new line is pointless"
Transfer('file://{}/ntl9.pdb' > 'worker://ntl9.pdb)
<posttask>
```

We now have one more transfer command. But something else has changed. There is one more files listed as required. So, the task can only run, if that file exists, but since we loaded it into the DB, it exists (for us). For example the newly created trajectory `25.dcd` does not exist yet. Would that be a requirement the task would fail. But let's check that it exists.

```
new_pdb.exists
```

```
True
```

Okay, we have now the PDB file staged and so any real bash commands could work with a file `ntl9.pdb`. Alright, so let's output its stats.

```
task.append('stat ntl9.pdb')
```

Note that usually you place these stage commands at the top or your script.

Now we could run this task, as before and see, if it works. (Make sure you still have a worker running)

```
project.queue(task)
```

And check, that the task is running

```
task.state
```

```
u'success'
```

If we did not screw up the task, it should have succeeded and we can look at the STDOUT.

```
print task.stdout
```

```
13:11:19 [worker:3] stdout from running task
GO...
Reading PDB
Done
Initialize Simulation
Done.
('# platform used:', 'CPU')
('# temperature:', Quantity(value=300.0, unit=kelvin))
```

```
START SIMULATION
DONE
Written to directory traj/
This new line is pointless
16777220 97338745 -rw-r--r-- 1 jan-hendrikprinz staff 0 1142279 "Mar 21␣
↪13:11:18 2017" "Mar 21 13:11:15 2017" "Mar 21 13:11:15 2017" "Mar 21␣
↪13:11:15 2017" 4096 2232 0 ntl9.pdb
```

Well, great, we have the pointless output and the stats of the newly staged file `ntl9.pdb`

How does a real script look like

Just for fun let's create the same scheduler that the `adaptivemdworker` uses, but from inside this notebook.

```python
from adaptivemd import WorkerScheduler
```

```python
sc = WorkerScheduler(project.resource)
```

If you really wanted to use the worker you need to initialize it and it will create directories and stage files for the generators, etc. For that you need to call `sc.enter(project)`, but since we only want it to parse our tasks, we only set the project without invoking initialization. You should normally not do that.

```python
sc.project = project
```

Now we can use a function `.task_to_script` that will parse a task into a bash script. So this is really what would be run on your machine now.

```python
print '\n'.join(sc.task_to_script(task))
```

```
set -e
# This is part of the adaptivemd tutorial
ln -s ../staging_area/alanine.pdb initial.pdb
ln -s ../staging_area/system.xml system.xml
ln -s ../staging_area/integrator.xml integrator.xml
ln -s ../staging_area/openmmrun.py openmmrun.py
mkdir -p traj/
python openmmrun.py -r --report-interval 1 -p CPU --store-interval 1 -t␣
↪initial.pdb --length 100 traj/
mkdir -p ../../projects/tutorial/trajs/00000076/
mv traj/* ../../projects/tutorial/trajs/00000076/
rm -r traj/
echo "This new line is pointless"
# write file ntl9.pdb from DB
stat ntl9.pdb
```

Now you see that all file paths have been properly interpreted to work. See that there is a comment about a temporary file from the DB that is then renamed. This is a little trick to be compatible with RPs way of handling files. (TODO: We might change this to just write to the target file. Need to check if that is still consistent)

A note on file locations

One problem with bash scripts is that when you create the tasks you have no concept on where the files actually are located. To get around this the created bash script will be scanned for paths, that contain prefixed like we are used to and are interpreted in the context of the worker / scheduler. The worker is the only instance to know all that is necessary so this is the place to fix that problem.

Let's see that in a little example, where we create an empty file in the staging area.

```
task = Task()
task.append('touch staging:///my_file.txt')
```

```
print '\n'.join(sc.task_to_script(task))
```

```
set -e
# This is part of the adaptivemd tutorial
touch ../staging_area/my_file.txt
```

And voila, the path has changed to a relative path from the working directory of the worker. Note that you see here the line we added in the very beginning of example 1 to our resource!

A Task from scratch

If you want to start a new task you can begin with

```
task = Task()
```

as we did before.

Just start adding staging and bash commands and you are done. When you create a task you can assign it a generator, then the system will assume that this task was generated by that generator, so don't do it for you custom tasks, unless you generated them in a generator. Setting this allows you to tell a worker only to run tasks of certain types.

The Python RPC Task

The tasks so far a very powerful, but they lack the possibility to call a python function. Since we are using python here, it would be great to really pretend to call a python function from here and not taking the detour of writing a python bash executable with arguments, etc... An example for this is the PyEmma generator which uses this capability.

Let's do an example of this as well. Assume we have a python function in a file (you need to have your code in a file so far so that we can copy the file to the HPC if necessary). Let's create the `.py` file now.

```
%%file my_rpc_function.py

def my_func(f):
    import os
    print f
    return os.path.getsize(f)
```

```
Overwriting my_rpc_function.py
```

Now create a PythonTask instead

```
task = PythonTask()
```

and the call function has changed. Note that also now you can still add all the bash and stage commands as before. A PythonTask is also a subclass of `PrePostTask` so we have a `.pre` and `.post` phase available.

```
from my_rpc_function import my_func
```

We call the function `my_func` with one argument

```
task.call(my_func, f=project.trajectories.one)
```

```
print task.description
```

```
Task: PythonTask(NoneType) [created]

Sources
- staging:///_run_.py
- file://{}/_rpc_input_0x71bdd2d10e2f11e7a0f00000000002eaL.json
- file://{}/my_rpc_function.py [exists]
Targets
- file://{}/_rpc_output_0x71bdd2d10e2f11e7a0f00000000002eaL.json
Modified

<pretask>
Transfer('file://{}/_rpc_input_0x71bdd2d10e2f11e7a0f00000000002eaL.json' > 'worker://
↪input.json)
Link('staging:///_run_.py' > 'worker://_run_.py)
Transfer('file://{}/my_rpc_function.py' > 'worker://my_rpc_function.py)
python _run_.py
Transfer('worker://output.json' > 'file://{}/_rpc_output_
↪0x71bdd2d10e2f11e7a0f00000000002eaL.json)
<posttask>
```

Well, interesting. What this actually does is to write the input arguments to the function into a temporary `.json` file on the worker, (in RP on the local machine and then transfers it to remote), rename it to `input.json` and read it in the `_run_.py`. This is still a little clumsy, but needs to be this way to be RP compatible which only works with files! Look at the actual script.

You see, that we really copy the `.py` file that contains the source code to the worker directory. All that is done automatically. A little caution on this. You can either write a function in a single file or use any installed package, but in this case the same package needs to be installed on the remote machine as well!

Let's run it and see what happens.

```
project.queue(task)
```

And wait until the task is done

```
project.wait_until(task.is_done)
```

The default settings will automatically save the content from the resulting output.json in the DB an you can access the data that was returned from the task at `.output`. In our example the result was just the size of a the file in bytes

```
task.output
```

```
136
```

And you can use this information in an adaptive script to make decisions.

success callback

The last thing we did not talk about is the possibility to also call a function with the returned data automatically on successful execution. Since this function is executed on the worker we (so far) only support function calls with the following restrictions.

1. you can call a function of the related generator class. for this you need to create the task using `PythonTask(generator)`
2. the function name you want to call is stored in `task.then_func_name`. So you can write a generator class with several possible outcomes and chose the function for each task.
3. The `Generator` needs to be part of `adaptivemd`

So in the case of `modeller.execute` we create a `PythonTask` that references the following functions

```
task = modeller.execute(project.trajectories)
```

```
task.then_func_name
```

```
'then_func'
```

So we will call the default `then_func` of modeller or the class modeller is of.

```
help(modeller.then_func)
```

```
Help on function then_func in module adaptivemd.analysis.pyemma.emma:

then_func(project, task, model, inputs)
```

These callbacks are called with the current project, the resulting data (which is in the modeller case a `Model` object) and array of initial inputs.

This is the actual code of the callback

```
@staticmethod
def then_func(project, task, model, inputs):
    # add the input arguments for later reference
    model.data['input']['trajectories'] = inputs['kwargs']['files']
    model.data['input']['pdb'] = inputs['kwargs']['topfile']
    project.models.add(model)
```

All it does is to add some of the input parameters to the model for later reference and then store the model in the project. You are free to define all sorts of actions here, even queue new tasks.

Task([generator])	A description for a task running on an HPC
PythonTask([generator])	A special task that does a RPC python calls

adaptivemd.Task

class `adaptivemd.`**`Task`**(*generator=None*)

A description for a task running on an HPC

Variables

- **`worker`** (`WorkingScheduler`) – the currently assigned Worker instance (not the scheduler!)

- **generator** (TaskGenerator) – if given the TaskGenerator that was used to create this task
- **state** (*str*) – a string representing the current state of the execution. One of - 'create' : task has been created and is available for execution - 'running': task is currently executed by a scheduler - 'queued' : task has been captured by a worker for execution - 'fail' : task has completed but failed. You can restart it - 'succedd' : task has completed and succeeded. - 'halt' : task has been halted by user. You can restart it - 'cancelled' : task has been cancelled by user. You CANNOT restart it
- **stdout** (LogEntry) – After completion you can access the stdout of the task here
- **stderr** (LogEntry) – After completion you can access the stderr of the task here

__init__(*generator=None*)

Methods

__init__([generator])	
add_cb(event, cb)	Add a custom callback
add_conda_env(name)	Add loading a conda env to all tasks of this resource
add_files(files)	Add additional files to the task execution
add_path(path)	**param path** a (list of) path(s) to be added to the $PATH variable before task execution
append(cmd)	Append a command to this task
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
cancel()	Mark a task as cancelled if it it not running or has been halted
descendants()	Return a list of all subclassed objects
fire(event, scheduler)	Fire an event like success or failed.
from_dict(dct)	
get(f[, name])	Get a file and make it available to the task in the main directory
get_uuid()	Create a new unique ID
has_failed()	Check if the task is done executing and has failed
idx(store)	Return the index which is used for the object in the given store.
is_done()	Check if the task is done executing.
link(f[, name])	Add an action to create a link to a file (under a new name)
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
prepend(cmd)	Append a command to this task
put(f, target)	Put a file back and make it persistent
remove(f)	Add an action to remove a file or folder
restart()	Mark a task as being runnable if it was stopped or failed before

Continued on next page

Table 1.58 – continued from previous page

setenv(key, value)	Set an environment variable for the task
to_dict()	
touch(f)	Add an action to create an empty file or folder at a given location
was_successful()	Check if the task is done executing and was successful

Attributes

ACTIVE_LONG	
CREATION_COUNT	
FINAL_STATES	
INSTANCE_UUID	
RESTARTABLE_STATES	
RUNNABLE_STATES	
additional_files	list of *Location*
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string
dependency_okay	Check if all dependency tasks are successful
description	Return a lengthy description of the task for debugging and information
environment	*dict str* – str
main	list of str or *Action*
modified_files	A set of all input files whose names match output names and hence will be overwritten
new_files	Return a set of all files the will be newly created by this task
pre_add_paths	list of str
pre_exec	list of str or *Action*
ready	Check if this task is ready to be executed
script	list of str or *Action*
source_locations	Return a set of all required file urls
sources	Return a set of all required input files
staged_files	Set of all staged files by the tasks generator
state	
stderr	
stdout	
target_locations	Return a set of all new and overwritten file urls
targets	Return a set of all new and overwritten files
unstaged_input_files	Return a set of *File* objects that are used but are not part of the generator stage
worker	

restart()
Mark a task as being runnable if it was stopped or failed before

cancel()
Mark a task as cancelled if it it not running or has been halted

dependency_okay
Check if all dependency tasks are successful

Returns True if all dependencies are fulfilled

Return type bool

ready

Check if this task is ready to be executed

Usually this only checks dependencies but might involve more elaborate checks for specific Task classes

Returns if True the task can now be executed

Return type bool

description

Return a lengthy description of the task for debugging and information

Returns the information text

Return type str

fire(*event*, *scheduler*)

Fire an event like success or failed.

Notes

You should never have to call this yourself. The scheduler does that.

Parameters

- **event** (*str*) – the events name like *fail*, *success*, *submit*
- **scheduler** (*Scheduler*) – the scheduler that issued the events to be fired

is_done()

Check if the task is done executing. Can be failed, successful or cancelled

Returns True if the task has finished its execution

Return type bool

was_successful()

Check if the task is done executing and was successful

Returns True if the task has finished successfully

Return type bool

has_failed()

Check if the task is done executing and has failed

Returns True if the task has finished but failed

Return type bool

add_cb(*event*, *cb*)

Add a custom callback

Parameters

- **event** (*str*) – name of the event to be called upon firing
- **cb** (*function*) – the function to be called. It must be a function that takes a task and a scheduler

`additional_files`

list of *Location* return the list of files created other than taken care of by actions. Should usually not be necessary. If you do some bad hacks with the bash you can add files that you transferred yourself to the project folders.

`add_files` (*files*)

Add additional files to the task execution

Should usually not be necessary. If you do some bad hacks with the bash you can add files that you transferred yourself to the project folders.

Parameters **`files`** (list of *File*) – the list of files to be added to the task

`targets`

Return a set of all new and overwritten files

Returns the list of files that are created or overwritten by this task

Return type set of *File*

`target_locations`

Return a set of all new and overwritten file urls

Returns the list of file urls that are created or overwritten by this task

Return type set of str

`sources`

Return a set of all required input files

Returns the list of files that are required by this task

Return type set of *File*

`source_locations`

Return a set of all required file urls

Returns the list of file urls that are required by this task

Return type set of str

`new_files`

Return a set of all files the will be newly created by this task

Returns the set of files that are created by this task

Return type set of *File*

`modified_files`

A set of all input files whose names match output names and hence will be overwritten

Returns the list of potentially overwritten input files

Return type list of *File*

`staged_files`

Set of all staged files by the tasks generator

Returns files that are staged by the tasks generator

Return type set of *File*

Notes

There might be more files stages by other generators

unstaged_input_files

Return a set of *File* objects that are used but are not part of the generator stage

Usually a task requires some reused files from staging and specific others. This function lists all the files that this task will stage to its working directory but will not be available from the set of staged files of the tasks generator

Returns the set of *File* objects that are needed and not staged

Return type set of *File*

setenv(*key*, *value*)

Set an environment variable for the task

Parameters

- **key** (*str*) –
- **value** (*str*) –

append(*cmd*)

Append a command to this task

prepend(*cmd*)

Append a command to this task

get(*f*, *name=None*)

Get a file and make it available to the task in the main directory

Parameters

- **f** (*File*) –
- **name** (*Location* or str) –

Returns the file instance of the file to be created in the unit

Return type *File*

touch(*f*)

Add an action to create an empty file or folder at a given location

Parameters **f** (*Location*) – the location (file or folder) to be used

link(*f*, *name=None*)

Add an action to create a link to a file (under a new name)

Parameters

- **f** (*Location*) – the source location (file or folder) to be used
- **name** (*Location* or str) – the target location to be used. For source files and target folders the basename is copied

Returns the actual target location

Return type *Location*

put(*f*, *target*)

Put a file back and make it persistent

Corresponds to output_staging

Parameters

- **f** (*File*) – the file to be used

- **target** (str or *File*) – the target location. Need to contain a URL like *staging://* or *file://* for application side files

Returns the actual target location

Return type *Location*

remove (*f*)

Add an action to remove a file or folder

Parameters **f** (*File*) – the location to be removed

Returns the actual location

Return type *Location*

add_conda_env (*name*)

Add loading a conda env to all tasks of this resource

This calls *resource.wrapper.append('source activate {name}')* :param name: name of the conda environment :type name: str

adaptivemd.PythonTask

class adaptivemd.**PythonTask** (*generator=None*)

A special task that does a RPC python calls

Variables

- **then_func_name** (*str or None*) – the name of the function of the *TaskGenerator* to be called with the resulting output
- **store_output** (*bool*) – if True then the result from the RPC called function will also be stored in the database. It can later be retrieved using the *.output* attribute on the task completed successfully

__init__ (*generator=None*)

Methods

__init__([generator])	
add_cb(event, cb)	Add a custom callback
add_conda_env(name)	Add loading a conda env to all tasks of this resource
add_files(files)	Add additional files to the task execution
add_path(path)	**param path** a (list of) path(s) to be added to the $PATH variable before task execution
append(cmd)	Append a command to this task
args()	Return a list of args of the *__init__* function of a class
backup_output_json(target)	Add an action that will copy the resulting JSON file to the given path
base()	Return the most parent class actually derived from StorableMixin
call(command, **kwargs)	Set the python function to be called with its arguments

Continued on next page

Table 1.60 – continued from previous page

cancel()	Mark a task as cancelled if it it not running or has been halted
descendants()	Return a list of all subclassed objects
fire(event, scheduler)	Fire an event like success or failed.
from_dict(dct)	
get(f[, name])	Get a file and make it available to the task in the main directory
get_uuid()	Create a new unique ID
has_failed()	Check if the task is done executing and has failed
idx(store)	Return the index which is used for the object in the given store.
is_done()	Check if the task is done executing.
link(f[, name])	Add an action to create a link to a file (under a new name)
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
prepend(cmd)	Append a command to this task
put(f, target)	Put a file back and make it persistent
remove(f)	Add an action to remove a file or folder
restart()	Mark a task as being runnable if it was stopped or failed before
setenv(key, value)	Set an environment variable for the task
then(func_name)	Set the name of the function to be called from the generator after success
to_dict()	
touch(f)	Add an action to create an empty file or folder at a given location
was_successful()	Check if the task is done executing and was successful

Attributes

ACTIVE_LONG	
CREATION_COUNT	
FINAL_STATES	
INSTANCE_UUID	
RESTARTABLE_STATES	
RUNNABLE_STATES	
additional_files	list of *Location*
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string
dependency_okay	Check if all dependency tasks are successful
description	Return a lengthy description of the task for debugging and information
environment	*dict str* – str
main	
modified_files	A set of all input files whose names match output names and hence will be overwritten

Continued on next page

Table 1.61 – continued from previous page

new_files	Return a set of all files the will be newly created by this task
output	Return the data contained in the output file
pre_add_paths	list of str
pre_exec	
ready	Check if this task is ready to be executed
script	list of str or *Action*
source_locations	Return a set of all required file urls
sources	Return a set of all required input files
staged_files	Set of all staged files by the tasks generator
state	
stderr	
stdout	
target_locations	Return a set of all new and overwritten file urls
targets	Return a set of all new and overwritten files
then_func	
unstaged_input_files	Return a set of *File* objects that are used but are not part of the generator stage
worker	

backup_output_json (*target*)

Add an action that will copy the resulting JSON file to the given path

Parameters **target** (*Location*) – the place to copy the resulting *output.json* file to

output

Return the data contained in the output file

Returns

Return type object

then (*func_name*)

Set the name of the function to be called from the generator after success

Parameters **func_name** (*str*) – the function name to be called after success

call (*command*, ***kwargs*)

Set the python function to be called with its arguments

Parameters

- **command** (*function*) – a python function defined inside a package or a function. If in a package then the package needs to be installed on the cluster to be called. A function defined in a local file can be called as long as dependencies are installed.
- **kwargs** (***kwargs*) – named arguments to the function

Engines

The `Trajectory` object

Before we talk about adaptivity, let's have a look at possibilities to generate trajectories.

We assume that you successfully ran a first trajectory using a worker. Next, we talk about lot's of ways to generate new trajectories.

You will do this in the beginning. Remember we already have a PDB stored from setting up the engine. if you want to start from this configuration do as before

1. create the `Trajectory` object you want
2. make a task
3. submit the task to craft the object into existance on the HPC

A trajectory contains all necessary information to make itself. It has

1. a (hopefully unique) location: This will we the folder where all the files that belong to the trajectory go.
2. an initial frame: the initial configuration to be used to tell the MD simulation package where to start
3. a length in frames to run
4. the `Engine`: the actual engine I want to use to create the trajectory.

Note, the `Engine` is technically not required unless you want to use `.run()` but it makes sense, because the engine contains information about the topology and, more importantly information about which output files are generated. This is the essential information you will need for analysis, e.g. what is the filename of the trajectory file that contains the protein structure and what is its stride?

Let's first build a `Trajectory` from scratch

```
file_name = next(project.traj_name)              # get a unique new filename

trajectory = Trajectory(
    location=file_name,                          # this creates a new filename
    frame=pdb_file,                              # initial frame is the PDB
    length=100,                                  # length is 100 frames
    engine=engine                                # the engine to be used
)
```

Since this is tedious to write there is a shortcut

```
trajectory = project.new_trajectory(
    frame=pdb_file,
    length=100,
    engine=engine,
    number=1          # if more then one you get a list of trajectories
)
```

Like in the first example, now that we have the parameters of the `Trajectory` we can create the task to do that.

OpenMMEngine

Let's do an example for an OpenMM engine. This is simply a small python script that makes OpenMM look like a executable. It run a simulation by providing an initial frame, OpenMM specific system.xml and integrator.xml files and some additional parameters like the platform name, how often to store simulation frames, etc.

```
engine = OpenMMEngine(
    pdb_file=pdb_file,
    system_file=File('file://../files/alanine/system.xml').load(),
    integrator_file=File('file://../files/alanine/integrator.xml').load(),
    args='-r --report-interval 1 -p CPU'
).named('openmm')
```

We have now an OpenMMEngine which uses the previously made pdb `File` object and uses the location defined in there. The same for the OpenMM XML files and some args to run using the `CPU` kernel, etc.

Last we name the engine `openmm` to find it later.

```
engine.name
```

Next, we need to set the output types we want the engine to generate. We chose a stride of 10 for the `master` trajectory without selection and a second trajectory with only protein atoms and native stride.

Note that the stride and all frame number ALWAYS refer to the native steps used in the engine. In out example the engine uses `2fs` time steps. So master stores every `20fs` and protein every `2fs`

```
engine.add_output_type('master', 'master.dcd', stride=10)
engine.add_output_type('protein', 'protein.dcd', stride=1, selection='protein')
```

Classes

Engine()	An generator for trajectory simulation tasks
Trajectory(location, frame, length[, engine])	Represents a trajectory *File* on the cluster
OpenMMEngine(system_file, integrator_file, ...)	OpenMM Engine to be used by Adaptive MD

adaptivemd.Engine

class adaptivemd.**Engine**

An generator for trajectory simulation tasks

__init__()

Methods

__init__()	
add_output_type(name[, filename, stride, ...])	Add an output type for a trajectory kind to be generated by this engine
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
extend(target, length)	Create a task that extends a trajectory given in the input
from_dict(dct)	
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
items()	
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
run(target)	Create a task that returns a trajectory given in the input
stage(obj[, target])	Short cut to add a file to be staged
to_dict()	

Attributes

`ACTIVE_LONG`	
`CREATION_COUNT`	
`INSTANCE_UUID`	
`base_cls`	Return the base class
`base_cls_name`	Return the name of the base class
`cls`	Return the class name as a string
`files`	
`full_strides`	list of strides for trajectories that have full coordinates
`native_stride`	The least common multiple stride of all generated trajectories.
`stage_in`	Return a list of actions needed before tasks can be generated

`run`(*target*)

Create a task that returns a trajectory given in the input

Parameters **`target`** (*Trajectory*) – location of the created target trajectory

Returns the task object containing the job description

Return type *Task*

`extend`(*target*, *length*)

Create a task that extends a trajectory given in the input

Parameters

- **`target`** (*Trajectory*) – location of the target trajectory to be extended
- **`length`** (`int`) – number of additional frames to be computed

Returns the task object containing the job description

Return type *Task*

`add_output_type`(*name*, *filename=None*, *stride=1*, *selection=None*)

Add an output type for a trajectory kind to be generated by this engine

Parameters

- **`name`** (`str`) – the name to call the output type by
- **`filename`** (`str`) – a filename to be used for this output type
- **`stride`** (`int`) – the stride used by this particular trajectory relative to the native steps of the engine.
- **`selection`** (`str`) – an mdtraj.Topology.select type filter string to store only a subset of atoms

`native_stride`

The least common multiple stride of all generated trajectories.

If you want consistent trajectory length your simulation length need to be multiples of this number. The number is relative to the native time steps

Returns the lcm stride relative to the engines timesteps

Return type int

full_strides
list of strides for trajectories that have full coordinates

this is useful to figure out from which frames you can restart a new trajectory. Usually you only have a single one with full frames.

Returns the list of strides for full trajectories

Return type list of int

adaptivemd.OpenMMEngine

class adaptivemd.**OpenMMEngine**(*system_file*, *integrator_file*, *pdb_file*, *args=None*)
OpenMM Engine to be used by Adaptive MD

Variables

- **system_file** (*File*) – the system.xml file for OpenMM
- **integrator_file** (*File*) – the integrator.xml file for OpenMM
- **pdb_file** (*File*) – the .pdb file for the topology
- **args** (*str*) – a list of arguments passed to the *openmmrun.py* script

__init__(*system_file*, *integrator_file*, *pdb_file*, *args=None*)

Methods

__init__(system_file, integrator_file, pdb_file)	
add_output_type(name[, filename, stride, ...])	Add an output type for a trajectory kind to be generated by this engine
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
extend(source, length)	
from_dict(dct)	
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
items()	
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
run(target)	
stage(obj[, target])	Short cut to add a file to be staged
then_func_import(project, task, data, inputs)	
to_dict()	

Attributes

ACTIVE_LONG
CREATION_COUNT

Continued on next page

Table 1.66 – continued from previous page

INSTANCE_UUID	
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string
files	
full_strides	list of strides for trajectories that have full coordinates
native_stride	The least common multiple stride of all generated trajectories.
stage_in	Return a list of actions needed before tasks can be generated

Generators

TaskGenerators are instances whose purpose is to create tasks to be executed. This is similar to the way Kernels work. A TaskGenerator will generate *Task* objects for you which will be translated into a radical.pilot.ComputeUnitDescription and executed. In simple terms:

The task generator creates the bash scripts for you that run a task.

A task generator will be initialized with all parameters needed to make it work and it will now what needs to be staged to be used.

Add generators to project

To add a generator to the project for later usage. You pick the Project.generators() store and just *Bundle.add()* it.

Consider a store to work like a set() in python. It contains objects only once and is not ordered. Therefore we need a name to find the objects later. Of course you can always iterate over all objects, but the order is not given.

To be precise there is an order in the time of creation of the object, but it is only accurate to seconds and it really is the time it was created and not stored.

```
project.generators.add(engine)
project.generators.add(modeller)
```

Note, that you cannot add the same engine twice. But if you create a new engine it will be considered different and hence you can store it again.

Classes

TaskGenerator()	A generator helper for *Task* object creation
Engine()	An generator for trajectory simulation tasks
Analysis()	A generator for tasks that represent analysis of trajectories

adaptivemd.TaskGenerator

class adaptivemd.**TaskGenerator**

A generator helper for *Task* object creation

This is an important group and is supposed to make it easy for you to create *Task* object. In a real situation a user should not be faced with the *Task* details, or at least the programming of a generator is a separate problem. Once you have the generators use them in your adaptive scripts.

Examples

Variables

- **`initial_staging`** (list of dict or str or *Action*) – a list of actions to be run once before this generator can be used
- **`items`** (dict of *File*) – a dictionary of *File* by name to simplify access to certain files

`__init__``()`

Methods

`__init__`()	
`args`()	Return a list of args of the *__init__* function of a class
`base`()	Return the most parent class actually derived from StorableMixin
`descendants`()	Return a list of all subclassed objects
`from_dict`(dct)	
`get_uuid`()	Create a new unique ID
`idx`(store)	Return the index which is used for the object in the given store.
`items`()	
`named`(name)	Attach a .name property to an object
`objects`()	Returns a dictionary of all storable objects
`stage`(obj[, target])	Short cut to add a file to be staged
`to_dict`()	

Attributes

`ACTIVE_LONG`	
`CREATION_COUNT`	
`INSTANCE_UUID`	
`base_cls`	Return the base class
`base_cls_name`	Return the name of the base class
`cls`	Return the class name as a string
`files`	
`stage_in`	Return a list of actions needed before tasks can be generated

`stage_in`

Return a list of actions needed before tasks can be generated

Returns the list of Actions to be parsed into stage in steps

Return type list of *Action*

`stage``(`*obj*, *target=None*`)`

Short cut to add a file to be staged

Parameters

- **obj** (*File*) – the file to be staged in the initial staging phase
- **target** (*Location* or str) – the (different) target name to be used

adaptivemd.Analysis

class adaptivemd.**Analysis**

A generator for tasks that represent analysis of trajectories

__init__()

Methods

__init__()	
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
from_dict(dct)	
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
items()	
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
stage(obj[, target])	Short cut to add a file to be staged
to_dict()	

Attributes

ACTIVE_LONG	
CREATION_COUNT	
INSTANCE_UUID	
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string
files	
stage_in	Return a list of actions needed before tasks can be generated

Scheduler Functions

WorkerScheduler(resource[, verbose])	A single instance worker scheduler to interprete *Task* objects

Continued on next page

Table 1.72 – continued from previous page

Scheduler(resource[, queue, runtime, cores])	Class to handle task execution on a resource

adaptivemd.WorkerScheduler

class adaptivemd.**WorkerScheduler**(*resource*, *verbose=False*)

A single instance worker scheduler to interprete *Task* objects

Parameters

- **resource** (*Resource*) – the resourse this scheduler should use.
- **verbose** (*bool*) – if True the worker will report lots of stuff

__init__(*resource*, *verbose=False*)

A single instance worker scheduler to interprete *Task* objects

Parameters

- **resource** (*Resource*) – the resourse this scheduler should use.
- **verbose** (*bool*) – if True the worker will report lots of stuff

Methods

__init__(resource[, verbose])	A single instance worker scheduler to interprete *Task* objects
add_event(event)	
advance()	Advance checking if tasks are completed or failed
cancel_events()	Remove all pending events and stop them from further task execution
change_state(new_state)	
enter([project])	
exit()	Shut down the scheduler
flatten_location(obj)	
get_path(f)	Get the schedulers representation of the path in *Location* object
on(condition)	Shortcut for creation and appending of a new Event
release_queued_tasks()	Release captured tasks scheduled for execution (if not started yet)
remove_task(task)	
replace_prefix(path)	
shut_down([wait_to_finish])	
stage_generators()	
stage_in(staging)	
stage_project()	Create paths necessary for the current project
stop_current()	Stop execution of the current task immediately
submit(submission)	Submit a *Task* or a *Trajectory*
task_to_script(task)	Convert a task to an executable bash script
trigger()	Trigger a check of state changes that leads to task execution
unroll_staging_path(location)	Convert a staging location into an adaptiveMD location
wait()	Wait until no more units are running and hence no more state changes

Attributes

current_task_dir	Return the current path to the worker directory
folder_name	
generators	Return the generators of the attached project
is_idle	
path	
staging_area_location	

__init__(*resource*, *verbose=False*)

A single instance worker scheduler to interprete *Task* objects

Parameters

- **resource** (*Resource*) – the resourse this scheduler should use.
- **verbose** (*bool*) – if True the worker will report lots of stuff

task_to_script(*task*)

Convert a task to an executable bash script

Parameters **task** (*Task*) – the *Task* instance to be converted

Returns a list of bash commands

Return type list of str

submit(*submission*)

Submit a *Task* or a *Trajectory*

Parameters **submission** ((list of) *Task* or *Trajectory*) –

Returns the list of tasks actually executed after looking at all objects

Return type list of *Task*

current_task_dir

Return the current path to the worker directory :returns: the path or None if no task is executed at the time :rtype: str or None

stop_current()

Stop execution of the current task immediately

Returns if True the current task was cancelled, False if there was no task running

Return type bool

advance()

Advance checking if tasks are completed or failed

Needs to be called in regular intervals. Usually by the main worker instance

release_queued_tasks()

Release captured tasks scheduled for execution (if not started yet)

You can prefetch tasks (although not recommended for single workers) and this releases not started jobs back to the queue

stage_project()

Create paths necessary for the current project

adaptivemd.Scheduler

class adaptivemd.**Scheduler**(*resource*, *queue=None*, *runtime=240*, *cores=1*)

Class to handle task execution on a resource

Notes

In RP this would correspond to a Pilot with a UnitManager

Variables

- **project** (*Project*) – a back reference to the project that uses this scheduler
- **tasks** (dict uid : *Task*) – dict that references all running task by the associated CU.uid
- **wrapper** (*Task*) – a wrapping task that contains additional commands to be executed around each task running on that scheduler. It usually contains adding certain paths, etc.

Parameters

- **resource** (*Resource*) – a *Resource* where this scheduler works on
- **queue** (*str*) – the name of the queue to be used for pilot creation
- **runtime** (*int*) – max runtime in minutes for the created pilot
- **cores** – number of used cores to be used in the created pilot

__init__(*resource*, *queue=None*, *runtime=240*, *cores=1*)

Parameters

- **resource** (*Resource*) – a *Resource* where this scheduler works on
- **queue** (*str*) – the name of the queue to be used for pilot creation
- **runtime** (*int*) – max runtime in minutes for the created pilot
- **cores** – number of used cores to be used in the created pilot

Methods

__init__(resource[, queue, runtime, cores])	**param resource** a *Resource* where this scheduler works on
add_event(event)	
cancel_events()	Remove all pending events and stop them from further task execution
change_state(new_state)	
enter([project])	Call a preparations to use a scheduler
exit()	Shut down the scheduler
flatten_location(obj)	
get_path(f)	Get the schedulers representation of the path in *Location* object
on(condition)	Shortcut for creation and appending of a new Event

Continued on next page

Table 1.75 – continued from previous page

remove_task(task)	
replace_prefix(path)	Interprete adaptive paths and replace prefixes with real os paths
shut_down([wait_to_finish])	Do a controlled shutdown.
stage_generators()	Prepare files and folder for all generators
stage_in(staging)	
submit(submission)	Submit a task in form of an event, a task or an task-like object
trigger()	Trigger a check of state changes that leads to task execution
unroll_staging_path(location)	Convert a staging location into an adaptiveMD location
wait()	Wait until no more units are running and hence no more state changes

Attributes

folder_name	
generators	Return the generators of the attached project
is_idle	
staging_area_location	Return the path to the staging area used by this scheduler

__init__(*resource*, *queue=None*, *runtime=240*, *cores=1*)

Parameters

- **resource** (*Resource*) – a *Resource* where this scheduler works on
- **queue** (*str*) – the name of the queue to be used for pilot creation
- **runtime** (*int*) – max runtime in minutes for the created pilot
- **cores** – number of used cores to be used in the created pilot

staging_area_location

Return the path to the staging area used by this scheduler

generators

Return the generators of the attached project

Returns

Return type list of *TaskGenerator*

get_path(*f*)

Get the schedulers representation of the path in *Location* object

Parameters **f** (*Location*) – the location object

Returns a real file path

Return type str

unroll_staging_path(*location*)

Convert a staging location into an adaptiveMD location

Parameters **location** (*Location*) – the location to the changed

enter(*project=None*)
Call a preparations to use a scheduler

Parameters **project** (*Project*) – the project the worker should execute for

exit()
Shut down the scheduler

stage_generators()
Prepare files and folder for all generators

submit(*submission*)
Submit a task in form of an event, a task or an task-like object

Parameters **submission** ((list of) [*Task* or object or *Event*]) –

Returns the list of tasks actually executed after looking at all objects

Return type list of *Task*

trigger()
Trigger a check of state changes that leads to task execution

shut_down(*wait_to_finish=True*)
Do a controlled shutdown. Cancel all units and wait until they finish.

Parameters **wait_to_finish** (`bool`) – if True default the function will block until all tasks report finish

on(*condition*)
Shortcut for creation and appending of a new Event

Parameters **condition** (*Condition*) –

Returns

Return type *Event*

wait()
Wait until no more units are running and hence no more state changes

cancel_events()
Remove all pending events and stop them from further task execution

replace_prefix(*path*)
Interprete adaptive paths and replace prefixes with real os paths

Parameters **path** (`str`) – the path with an adaptiveMD prefix

Returns the path without any adaptiveMD prefixes

Return type str

Workers

`adaptive.Worker`s are the main execution units of your :class:`adaptive.Task` instances. While the `adaptive.Task` object contains specifics about what you want to happen, like create a trajectory with this length, it does not know anything about where to run it and how to achieve the goal there. The `adaptive.Task` definition is concrete but it misses knowlegde that only the actual `adaptive.Worker` that executes it has. Things like the actual working directory, (you do not want to interfere with other workers), how to copy a file from A to B, etc...

There are two ways to use a `adaptive.Worker`,

1. a manual way in a script, or
2. through a stand-alone bash command. That will run a python script which creates a Worker with some options and just runs it until it is shut down.

You will be mostly using the 2. way since it is much simpler and you will typically submit it to the queue and then it will listen in the DB for task to be run in regular intervals.

How does it work

Technically a worker gets a task to execute (the task of picking a task from the DB is not solved by the worker!). Then

1. A new worker directory is created named according to the task
2. It will convert the given task into a bash script (this might involve already copying files from the DB to some folders since this is something that is not handled in a bash script)
3. The bash script is executed within the current working directory
4. Once it is finished and succeeded the outputs are stored and created files are registered as being existent now.
5. A Callback is run, if the task had one

Communication

The actual worker will run somewhere on the HPC or as a separate process on your local machine. In both cases the Worker instance will not be present in your execution script or notebook. Hence changes or function you call in your notebook will have no effect to the worker running somewhere else.

Still, any worker that you create through the `adaptivemdworker` script will be stored in the project, so its settings are visible to anyone with access you your project DB.

Using the BD, you have a way to connect to the worker. You can set a specaicl property which is checked by the running worker in regular intervals and if it takes special values the Worker will act. You could try

The other typical thing that is of interest is the status of the worker

Dead workers

This is bad and should not happen, but it can. When a worker dies it does not mean that its execution thread died. The bash script will be run in another thread that is monitored (and should also die if the worker is killed).

Now the worker stalls and stops accepting tasks, etc. What happens?

The worker will continuously send a heartbeat to the DB, which is just a current timestamp. It does this every 10 seconds. You can simply check this by

with the `.seen` property.

If it is supposed to write it every 10 seconds and it does not do that for a minute we get suspicious. When calling `project.trigger()` which will also look for open events to be run, the project also checks, if all workers are still alive – where alive means that there last alive time is > 60s.

So, if a worker is considered dead, it is sends the `kill` command just to make sure that it will be dead when we will consider it being so and not secretly keep on working. There would be no problem, if it would sill run correctly but if it really had failed we want to retry the failed job.

Next, the current task is considered failed and will be restarted. This means just to set the `task.state` to `created`. And another worker that is responding can pick it up. This task will overwrite all files that the failed task would have generated and so we keep consistent in the database.

RUN `adaptivemdworker`

the tool `adaptivemdworker` takes some options

usage: adaptivemdworker [-h] [-t [WALLTIME]] [-d [MONGO_DB_PATH]] [-g [GENERATORS]] [-w [WRAPPERS]] [-l] [-v] [-a] [–sheep] [-s [SLEEP]] [–heartbeat [HEARTBEAT]] project_name

Run an AdaptiveMD worker

positional arguments: project_name project name the worker should attach to

optional arguments:

-h, --help show this help message and exit

-t [WALLTIME], –walltime [WALLTIME] minutes until the worker shuts down. If 0 (default) it will run indefinitely

-d [MONGO_DB_PATH], –mongodb [MONGO_DB_PATH] the mongodb url to the db server

-g [GENERATORS], –generators [GENERATORS] a comma separated list of generator names used to dispatch the tasks. the worker will only respond to tasks from generators whose names match one of the names in the given list. Example: –generators=openmm will only run scripts from generators named *openmm*

-w [WRAPPERS], –wrappers [WRAPPERS] a comma separated list of simple function call to the resource. This can be used to add e.g. CUDA support for specific workers. Example: –wrappers=add_path("something"),add_cuda_module()

-l, --local if true then the DB is set to the default local port

-v, --verbose if true then stdout and stderr of subprocesses will be rerouted. Use for debugging.

-a, --allegro if true then the DB is set to the default allegro setting

--sheep if true then the DB is set to the default sheep setting

-s [SLEEP], –sleep [SLEEP] polling interval for new jobs in seconds. Default is 2 seconds. Increase to get less traffic on the DB

–heartbeat [HEARTBEAT] heartbeat interval in seconds. Default is 10 seconds.

Examples

Run using the local DB setting `mongodb://localhost:27019` for `my_project`

adaptivemdworker -l my_project

Classes

Worker([walltime, generators, sleep, ...])	A Worker instance the will submit tasks from the DB to a scheduler

adaptivemd.Worker

class adaptivemd.**Worker**(*walltime=None*, *generators=None*, *sleep=None*, *heartbeat=None*, *prefetch=1*, *verbose=False*)

A Worker instance the will submit tasks from the DB to a scheduler

__init__(*walltime=None*, *generators=None*, *sleep=None*, *heartbeat=None*, *prefetch=1*, *verbose=False*)

Methods

__init__([walltime, generators, sleep, ...])	
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
create(project)	
descendants()	Return a list of all subclassed objects
execute(command)	Send and execute a single command to the worker
from_dict(dct)	
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
run()	Start the worker to execute tasks until it is shut down
shutdown([gracefully])	Shut down the worker
to_dict()	

Attributes

ACTIVE_LONG	
CREATION_COUNT	
INSTANCE_UUID	
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string
command	
current	
n_tasks	
prefetch	
project	returns: the currently used project
scheduler	returns: the currently used scheduler to execute tasks
seen	
state	
verbose	

scheduler

Returns the currently used scheduler to execute tasks

Return type *WorkerScheduler*

project

Returns the currently used project

Return type *Project*

execute (*command*)

Send and execute a single command to the worker

Note that the worker is registered on the DB but running on your HPC. Just loading it does not allow you to call functions like *.shutdown*. These would only be called on your local instance. All you can do is use *execute* which will store a command in the DB and once the real running worker executed it. The command will be cleared from the DB.

Parameters **command** (*str*) – the command to be executed

run ()

Start the worker to execute tasks until it is shut down

shutdown (*gracefully=True*)

Shut down the worker

Parameters **gracefully** (*bool*) – if True the worker is allowed some time to finish running tasks

The folder structure

For reference, this is the file structure of *adaptiveMD*.

```
{shared_folder}/             # referenced by `shared://` and set in the `Resource`
  adaptivemd/                #                               set in the `Resource`
    projects/
      {project-name-1}/      # referenced by `project://`
        trajs/
          00000000/
          00000001/
          ...
        models/
    workers/                 # referenced by `sandbox://`
      staging_area/          # referenced by `staging://`
      worker.{task_UUID}/    # referenced by `worker://` (only the current one)
      ...
```

1. `{shared_folder}`: is specific to your HPC or locally is usually chosen to be `$HOME`. The 2. `adaptivemd`: is the main folder where we will place all files. You can access the shared folder, there are no restrictions, but this should be restricted to loading input files like previous existing projects, etc. A stored files are place within this directory.

2. `projects`: will contain a single folder per `Project`, make sure that your project names are short but descriptive to later find files. All files you want to keep for later should be placed here.

3. `workers`: this folder is specific to the worker scheduler (there is also the possibility to use *radical.pilot* which uses `radical.pilot.sandbox`). It contains all temporary folders used by the workers to execute your tasks. Each task get a unique folder that also contains the UUID of the task to be handle. It is set up with all files and then in it your task is executed.

4. `staging_area`: This is also a temporary folder that contains files that are used by the workers for multiple tasks. Normally a task generating factory knows which files it will need multiple times

5. `trajs`: is a folder used by engines to place trajectories in.

Why do we need a trajectory object?

You might wonder why a `Trajectory` object is necessary. You could just build a function that will take these parameters and run a simulation. At the end it will return the trajectory object. The same object we created just now.

The main reason is to familiarize you with the general concept of asyncronous execution and so-called *Promises*. The trajectory object we built is similar to a *Promise* so what is that exactly?

A *Promise* is a value (or an object) that represents the result of a function at some point in the future. In our case it represents a trajectory at some point in the future. Normal promises have specific functions do deal with the unknown result, for us this is a little different but the general concept stands. We create an object that represents the specifications of a `Trajectory` and so, regardless of the existence, we can use the trajectory as if it would exists:

Get the length

```
print trajectory.length
```

```
100
```

and since the length is fixed, we know how many frames there are and can access them

```
print trajectory[20]
```

```
Frame(sandbox:///{}/00000001/[20])
```

ask for a way to extend the trajectory

```
print trajectory.extend(100)
```

```
<adaptivemd.engine.engine.TrajectoryExtensionTask object at 0x110e6e210>
```

ask for a way to run the trajectory

```
print trajectory.run()
```

```
<adaptivemd.engine.engine.TrajectoryGenerationTask object at 0x110dd46d0>
```

We can ask to extend it, we can save it. We can reference specific frames in it before running a simulation. You could even build a whole set of related simulations this way without running a single frame. You might understand that this is pretty powerful especially in the context of running asynchronous simulations.

Last, we did not answer why we have two separate steps: Create the trajectory first and then a task from it. The main reason is educational: **> It needs to be clear that a ‘‘Trajectory‘‘ *can exist* before running some engine or creating a task for it. The ‘‘Trajectory‘‘ *is not* a result of a simulation action.**

Execution Plans

You are free to conduct your simulations from a notebook but normally you will use a script. The main point about adaptivity is to make decision about tasks along the way.

We want to first look into a way to run python code asynchroneously in the project. For this, we write a function that should be executed. Inside you will create tasks and submit them.

If the function should pause, use `yield` as if you would `return` and exit the function. `Yield` will allow you to continue at this

```
yield {condition_to_continue}
```

This will interrupt your script until the function you return will return `True` when called. An example

```
def strategy(loops=10, trajs_per_loop=4, length=100):
    for loop in range(loops):
        # submit some trajectory tasks
        trajectories = project.new_ml_trajectory(length, trajs_per_loop)
        tasks = map(engine.task_run_trajectory, trajectories)
        project.queue(tasks)

        # continue if ALL of the tasks are done (can be failed)
        yield [task.is_done for task in tasks]

        # submit a model job
        task = modeller.execute(list(project.trajectories))
        project.queue(task)

        # when it is done do next loop
        yield task.is_done
```

and add the event to the project (these cannot be stored yet!)

```
project.add_event(strategy(loops=2))
```

```
<adaptivemd.event.FunctionalEvent at 0x10d615050>
```

What is missing now? The adding of the event triggered the first part of the code. But to recheck if we should continue needs to be done manually.

> RP has threads in the background and these can call the trigger whenever something changed or finished.

Still that is no problem, we can do that easily and watch what is happening

Let's see how our project is growing. TODO: Add threading.Timer to auto trigger.

```
import time
from IPython.display import clear_output
```

```
try:
    while project._events:
        clear_output(wait=True)
        print '# of files  %8d : %s' % (len(project.trajectories), '#' * len(project.
↪trajectories))
        print '# of models %8d : %s' % (len(project.models), '#' * len(project.
↪models))
        sys.stdout.flush()
        time.sleep(2)
        project.trigger()

except KeyboardInterrupt:
    pass
```

```
# of files        74 : ###############################################################
↪###########
# of models       33 : #################################
```

Let's do another round with more loops

```
project.add_event(strategy(loops=2))
```

```
<adaptivemd.event.FunctionalEvent at 0x10d633850>
```

And some analysis (might have better functions for that)

```
# find, which frames from which trajectories have been chosen
trajs = project.trajectories
q = {}
ins = {}
for f in trajs:
    source = f.frame if isinstance(f.frame, File) else f.frame.trajectory
    ind = 0 if isinstance(f.frame, File) else f.frame.index
    ins[source] = ins.get(source, []) + [ind]

for a,b in ins.iteritems():
    print a.short, ':', b
```

```
file://{}/alanine.pdb : [0, 0, 0]
sandbox:///{}/00000005/ : [95, 92, 67, 92]
sandbox:///{}/00000007/ : [11]
sandbox:///{}/00000011/ : [55]
sandbox:///{}/00000000/ : [28, 89, 72]
sandbox:///{}/00000002/ : [106]
sandbox:///{}/00000004/ : [31, 25, 60]
```

Event

And do this with multiple events in parallel.

```
def strategy2():
    for loop in range(10):
        num = len(project.trajectories)
        task = modeller.execute(list(project.trajectories))
        project.queue(task)
        yield task.is_done
        # continue only when there are at least 2 more trajectories
        yield project.on_ntraj(num + 2)
```

```
project.add_event(strategy(loops=10, trajs_per_loop=2))
project.add_event(strategy2())
```

```
<adaptivemd.event.FunctionalEvent at 0x107744c90>
```

And now wait until all events are finished.

```
project.wait_until(project.events_done)
```

Classes

ExecutionPlan(generator)	An wrap to turn python function into asynchronous execution

adaptivemd.ExecutionPlan

class adaptivemd.**ExecutionPlan**(*generator*)

An wrap to turn python function into asynchronous execution

The function is executed on start and interrupted if you use `yield {(list of )condition to continue}`

To make writing of asynchronous code easy you can use this wrapper class. Usually you start by opening a scheduler that you submit tasks to. Then submit a first task or yield a condition to wait for. Once this is met the code will continue to execute and you can submit more tasks until finally you will close the scheduler

Parameters **generator** (*function*) – the function (generator) to be used

__init__(*generator*)

Parameters **generator** (*function*) – the function (generator) to be used

Methods

__init__(generator)	**param generator** the function (generator) to be used
trigger(scheduler)	

Attributes

on_done	Return a *Condition* that is True once the event is finished

__init__(*generator*)

Parameters **generator** (*function*) – the function (generator) to be used

on_done

Return a *Condition* that is True once the event is finished

LogEntry Functions

LogEntry(logger, title, message[, level, objs])	A storable representation of a log entry

adaptivemd.LogEntry

class adaptivemd.**LogEntry**(*logger*, *title*, *message*, *level=3*, *objs=None*)

A storable representation of a log entry

Examples

```
>>> p = Project('tutorial-project')
>>> l = LogEntry('worker', 'failed execution', 'simsalabim, didnt work')
>>> print l
>>> p.logs.add(l)
```

Variables

- **logger** (*str*) – the name of the logger for classification
- **title** (*str*) – a short title for the log entry
- **message** (*str*) – the long and detailed message
- **level** (*int*) – pick *LogEntry.SEVERE*, *LogEntry.ERROR* or *LogEntry.INFO* (default)
- **objs** (*dict of storable objects*) – you can attach objects that can help with specifying the error message

__init__(*logger*, *title*, *message*, *level=3*, *objs=None*)

Methods

__init__(logger, title, message[, level, objs])	
args()	Return a list of args of the *__init__* function of a class
base()	Return the most parent class actually derived from StorableMixin
descendants()	Return a list of all subclassed objects
from_dict(dct)	Reconstruct an object from a dictionary representation
get_uuid()	Create a new unique ID
idx(store)	Return the index which is used for the object in the given store.
named(name)	Attach a .name property to an object
objects()	Returns a dictionary of all storable objects
to_dict()	Convert object into a dictionary representation

Attributes

ACTIVE_LONG	
CREATION_COUNT	
ERROR	
INFO	
INSTANCE_UUID	
SEVERE	
base_cls	Return the base class
base_cls_name	Return the name of the base class
cls	Return the class name as a string

AdaptiveMD (adaptivemd)

Hello

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

Symbols

__init__() (adaptivemd.Action method), 70
__init__() (adaptivemd.Analysis method), 101
__init__() (adaptivemd.AndBundle method), 68
__init__() (adaptivemd.BaseBundle method), 66
__init__() (adaptivemd.Bundle method), 62
__init__() (adaptivemd.BundleDelegator method), 69
__init__() (adaptivemd.Copy method), 73
__init__() (adaptivemd.Engine method), 96
__init__() (adaptivemd.ExecutionPlan method), 114
__init__() (adaptivemd.File method), 52
__init__() (adaptivemd.FileAction method), 71
__init__() (adaptivemd.FileTransaction method), 72, 73
__init__() (adaptivemd.Frame method), 58
__init__() (adaptivemd.FunctionDelegator method), 69
__init__() (adaptivemd.JSONFile method), 59
__init__() (adaptivemd.Link method), 75
__init__() (adaptivemd.LocalResource method), 47
__init__() (adaptivemd.Location method), 50
__init__() (adaptivemd.LogEntry method), 115
__init__() (adaptivemd.LogicBundle method), 67
__init__() (adaptivemd.MakeDir method), 77
__init__() (adaptivemd.Move method), 74
__init__() (adaptivemd.OpenMMEngine method), 98
__init__() (adaptivemd.OrBundle method), 68
__init__() (adaptivemd.Project method), 41
__init__() (adaptivemd.PythonTask method), 92
__init__() (adaptivemd.Remove method), 76
__init__() (adaptivemd.Scheduler method), 104, 105
__init__() (adaptivemd.SortedBundle method), 65
__init__() (adaptivemd.StoredBundle method), 63
__init__() (adaptivemd.Task method), 87
__init__() (adaptivemd.TaskGenerator method), 100
__init__() (adaptivemd.Touch method), 75
__init__() (adaptivemd.Trajectory method), 55
__init__() (adaptivemd.ViewBundle method), 65
__init__() (adaptivemd.Worker method), 109
__init__() (adaptivemd.WorkerScheduler method), 102, 103
__init__() (adaptivemd.mongodb.DataDict method), 60

A

Action (class in adaptivemd), 70
add() (adaptivemd.Bundle method), 63
add() (adaptivemd.StoredBundle method), 64
add_cb() (adaptivemd.Task method), 89
add_conda_env() (adaptivemd.Task method), 92
add_event() (adaptivemd.Project method), 44
add_files() (adaptivemd.Task method), 90
add_output_type() (adaptivemd.Engine method), 97
added (adaptivemd.FileAction attribute), 71
additional_files (adaptivemd.Task attribute), 89
advance() (adaptivemd.WorkerScheduler method), 103
all (adaptivemd.BaseBundle attribute), 67
Analysis (class in adaptivemd), 101
AndBundle (class in adaptivemd), 68
append() (adaptivemd.Task method), 91

B

backup_output_json() (adaptivemd.PythonTask method), 94
BaseBundle (class in adaptivemd), 66
basename (adaptivemd.Location attribute), 51
basename_short (adaptivemd.Location attribute), 51
Bundle (class in adaptivemd), 62
BundleDelegator (class in adaptivemd), 69

C

c() (adaptivemd.BaseBundle method), 66
call() (adaptivemd.PythonTask method), 94
cancel() (adaptivemd.Task method), 88
cancel_events() (adaptivemd.Scheduler method), 106
clone() (adaptivemd.File method), 53
clone() (adaptivemd.Location method), 51
close() (adaptivemd.Project method), 43
close() (adaptivemd.StoredBundle method), 64
close_rp() (adaptivemd.Project method), 42
consume_one() (adaptivemd.StoredBundle method), 64

Copy (class in adaptivemd), 73
copy() (adaptivemd.File method), 54
create() (adaptivemd.File method), 53
current_task_dir (adaptivemd.WorkerScheduler attribute), 103

D

data (adaptivemd.JSONFile attribute), 60
DataDict (class in adaptivemd.mongodb), 60
delete() (adaptivemd.Project class method), 42
dependency_okay (adaptivemd.Task attribute), 88
description (adaptivemd.Task attribute), 89
dirname (adaptivemd.Location attribute), 51
drive (adaptivemd.Location attribute), 51

E

Engine (class in adaptivemd), 96
enter() (adaptivemd.Scheduler method), 105
events_done() (adaptivemd.Project method), 44
execute() (adaptivemd.Worker method), 110
ExecutionPlan (class in adaptivemd), 114
existing_frames (adaptivemd.Trajectory attribute), 57
exists (adaptivemd.File attribute), 54
exists (adaptivemd.Frame attribute), 59
exit() (adaptivemd.Scheduler method), 106
extend() (adaptivemd.Engine method), 97
extend() (adaptivemd.Trajectory method), 57
extension (adaptivemd.Location attribute), 51

F

f() (adaptivemd.BaseBundle method), 67
File (class in adaptivemd), 52
file() (adaptivemd.Trajectory method), 57
FileAction (class in adaptivemd), 71
FileTransaction (class in adaptivemd), 72
find_all_by() (adaptivemd.StoredBundle method), 64
find_ml_next_frame() (adaptivemd.Project method), 44
fire() (adaptivemd.Task method), 89
first (adaptivemd.SortedBundle attribute), 65
first (adaptivemd.StoredBundle attribute), 64
Frame (class in adaptivemd), 58
full_strides (adaptivemd.Engine attribute), 97
FunctionDelegator (class in adaptivemd), 69

G

generators (adaptivemd.Scheduler attribute), 105
get() (adaptivemd.JSONFile method), 60
get() (adaptivemd.Task method), 91
get_file() (adaptivemd.File method), 55
get_path() (adaptivemd.Scheduler method), 105
get_scheduler() (adaptivemd.Project method), 43

H

has_failed() (adaptivemd.Task method), 89
has_file (adaptivemd.File attribute), 55

I

index_in_outputs (adaptivemd.Frame attribute), 58
initialize() (adaptivemd.Project method), 42
is_done() (adaptivemd.Task method), 89
is_folder (adaptivemd.Location attribute), 51
is_temp (adaptivemd.Location attribute), 51

J

JSONFile (class in adaptivemd), 59

L

last (adaptivemd.StoredBundle attribute), 64
Link (class in adaptivemd), 75
link() (adaptivemd.File method), 54
link() (adaptivemd.Task method), 91
list() (adaptivemd.Project class method), 42
load() (adaptivemd.File method), 55
LocalResource (class in adaptivemd), 47
Location (class in adaptivemd), 49
LogEntry (class in adaptivemd), 114
LogicBundle (class in adaptivemd), 67

M

MakeDir (class in adaptivemd), 77
modified() (adaptivemd.File method), 53
modified_files (adaptivemd.Task attribute), 90
Move (class in adaptivemd), 74
move() (adaptivemd.File method), 54

N

native_stride (adaptivemd.Engine attribute), 97
new_files (adaptivemd.Task attribute), 90
new_ml_trajectory() (adaptivemd.Project method), 44
new_trajectory() (adaptivemd.Project method), 43

O

on() (adaptivemd.Scheduler method), 106
on_done (adaptivemd.ExecutionPlan attribute), 114
on_nmodel() (adaptivemd.Project method), 44
on_ntraj() (adaptivemd.Project method), 43
one (adaptivemd.BaseBundle attribute), 67
OpenMMEngine (class in adaptivemd), 98
OrBundle (class in adaptivemd), 68
output (adaptivemd.PythonTask attribute), 94
outputs() (adaptivemd.Trajectory method), 57

P

path (adaptivemd.Location attribute), 51
pick() (adaptivemd.BaseBundle method), 67
pick() (adaptivemd.Trajectory method), 57
prepend() (adaptivemd.Task method), 91

project (adaptivemd.Worker attribute), 109
Project (class in adaptivemd), 40
Project.EventTriggerTimer (class in adaptivemd), 45
put() (adaptivemd.Task method), 91
PythonTask (class in adaptivemd), 92

Q

queue() (adaptivemd.Project method), 43

R

ready (adaptivemd.Task attribute), 89
reconnect() (adaptivemd.Project method), 42
release_queued_tasks() (adaptivemd.WorkerScheduler method), 103
Remove (class in adaptivemd), 76
remove() (adaptivemd.File method), 54
remove() (adaptivemd.Task method), 92
removed (adaptivemd.FileAction attribute), 71
replace_prefix() (adaptivemd.Scheduler method), 106
required (adaptivemd.FileAction attribute), 71
restart() (adaptivemd.Task method), 88
run() (adaptivemd.Engine method), 97
run() (adaptivemd.Project method), 45
run() (adaptivemd.Trajectory method), 57
run() (adaptivemd.Worker method), 110

S

scheduler (adaptivemd.Worker attribute), 109
Scheduler (class in adaptivemd), 104
set_file() (adaptivemd.File method), 55
set_store() (adaptivemd.StoredBundle method), 64
setenv() (adaptivemd.Task method), 91
short (adaptivemd.Location attribute), 51
shut_down() (adaptivemd.Scheduler method), 106
shutdown() (adaptivemd.Worker method), 110
sorted() (adaptivemd.BaseBundle method), 67
SortedBundle (class in adaptivemd), 65
source_locations (adaptivemd.Task attribute), 90
sources (adaptivemd.Task attribute), 90
split (adaptivemd.Location attribute), 51
split_drive (adaptivemd.Location attribute), 52
stage() (adaptivemd.TaskGenerator method), 100
stage_generators() (adaptivemd.Scheduler method), 106
stage_in (adaptivemd.TaskGenerator attribute), 100
stage_project() (adaptivemd.WorkerScheduler method), 103
staged_files (adaptivemd.Task attribute), 90
staging_area_location (adaptivemd.Scheduler attribute), 105
stop() (adaptivemd.Project method), 45
stop_current() (adaptivemd.WorkerScheduler method), 103
StoredBundle (class in adaptivemd), 63
submit() (adaptivemd.Scheduler method), 106
submit() (adaptivemd.WorkerScheduler method), 103

T

target_locations (adaptivemd.Task attribute), 90
targets (adaptivemd.Task attribute), 90
Task (class in adaptivemd), 86
task_to_script() (adaptivemd.WorkerScheduler method), 103
TaskGenerator (class in adaptivemd), 99
then() (adaptivemd.PythonTask method), 94
Touch (class in adaptivemd), 75
touch() (adaptivemd.File method), 54
touch() (adaptivemd.Task method), 91
Trajectory (class in adaptivemd), 55
transfer() (adaptivemd.File method), 54
trigger() (adaptivemd.Project method), 45
trigger() (adaptivemd.Scheduler method), 106
types (adaptivemd.Trajectory attribute), 57

U

unroll_staging_path() (adaptivemd.Scheduler method), 105
unstaged_input_files (adaptivemd.Task attribute), 90
update() (adaptivemd.Bundle method), 63
url (adaptivemd.Location attribute), 51

V

v() (adaptivemd.BaseBundle method), 67
ViewBundle (class in adaptivemd), 65

W

wait() (adaptivemd.Scheduler method), 106
wait_until() (adaptivemd.Project method), 45
was_successful() (adaptivemd.Task method), 89
Worker (class in adaptivemd), 109
WorkerScheduler (class in adaptivemd), 102