Copernicus Documentation

Release 2.1

Sander Pronk, Iman Pouya, Erik Lindahl and others

May 11, 2016

Contents

Contents:

CHAPTER 1

Getting Started

This section covers the steps required to setup Copernicus. After going through the instructions below you will have a running copernicus server with a connected worker.

1.1 Prerequisites

Before we start installing there are some basic prerequisites that has to be met. Apart from these listed below the server and the worker have some additional prerequisites which we will cover in the setup sections for the server and the worker.

1.1.1 Python

Copernicus requires python 2.7 to be installed. The installation must also include openssl. On *nix systems python is usually preinstalled with openssl. running the command `python --version` will show you what version is installed.

1.1.2 Network Ports

The default network ports that Copernicus communicates via is 14807 for HTTP and 13807 for HTTPS. Please ensure that these ports are open in the network. These must be open for both inward and outward communication on any machine that is running the client, worker or server. If these ports cannot be used it is possible to specify other ports when setting up the server.

1.1.3 Git

Git is a tool to download software from a source code repository. To see if it is installed try to run the command `git` from a terminal. If an installation is needed please download it from http://git-scm.com/download

If an installation is needed please download it from http://git-scm.com/download

1.2 Downloading Copernicus

Copernicus can be downloaded from our public git repository. To download the source , use the following command `https://github.com/gromacs/copernicus.git`

Set the *CPC_HOME* environment variable to the location where copernicus is installed

- On UNIX/Linux systems this is usually done with `export CPC_HOME=path/to/cpc`
- Now add *CPC_HOME* to your *PATH* variable. On UNIX/Linux systems this is usually done with `export PATH=$PATH:$CPC_HOME`

You should preferably but this in the startup file of your shell. Otherwise you will have to do this everytime you open up a terminal.

You should now be able to run the following three commands

- `'cpcc -h'`
- `'cpc-server -h'`
- `'cpc-worker -h'`

1.3 Server Setup

1.3.1 Additional prerequisites

The server has some additional prerequisites. if you are planning to run the prepackaged workflows GROMACS must be installed and accessible for the server. The easiest way is if GROMACS is accessible from the path. For other ways to specify this please refer to the section server.

Optionally, if you will use the MSM workflow you will need to make sure that scipy, numpy and py-tables are installed. You will also need to fetch the MSMbuilder source from https://simtk.org/home/msmbuilder.

1.3.2 Installation

To install the server run the command

```
cpc-server setup <PROJECT_DIR>
```

where `PROJECT_DIR` is the directory where the server will store project data for all the projects that it will run. Make sure to specify the `PROJECT_DIR` in a writable location. Note that `PROJECT_DIR` is only specified during setup, however the creation of the directory occurs when we first create a project. During the setup, the server will ask for a password to the cpc-admin user (super user). You will also notice that a directory named .copernicus is created in your home directory. This is where Copernicus will store its settings.

To ensure that the server is properly installed, call the command

```
cpc-server start
```

The server is will now run as a background process. For the worker to be able to communicate with the server a connection bundle must be created: this is a file with a description of how to connect to a server together with a key pair. During the server setup a bundle is alredy created and put under the .copernicus folder. When we later on start a worker it will look for a connection bundle in the .copernicus folder, unless a bundle is specified

If you need to create additional bundles you can create them with the command

```
cpc-server bundle
```

Which generates the bundle, with the name client.cnx. This file can then be used on any machine to communicate with the server.

1.3.3 Optmizing the server

For larger workflows with thousands of jobs the load on the server can be rather heavy. To optimize the code execution it is more efficient to not run those parts as ordinary Python code If you have Cython installed you can run the bash script `compileLibraries.sh` that is located in the Copernicus installation folder: This will generate C code from Python files that will then be compiled to shared libraries. The other Python files will automatically use these shared libraries instead of the corresponding Python code, which improves the server efficiency. Remember that if you modify any Python file in Copernicus it is best to rerun the script to regenerate the shared libraries if any of the relevant files have changed. There is no further optimization used when generating the C code from the Python code.

1.4 Client Setup

The client is a command line tool use to send commands to the server. it can be run directly from your laptop. But before sending commands to a server it needs to know its address This is done with the‘‘add-server‘‘ command:

```
cpcc add-server my.serverhostname.com
```

```
cpcc add-server my.serverhostname.com 14807
```

1.4.1 Logging in

To start sending commands to the server you need to first login. `cpcc login cpc-admin`

Then type the password you set for the user cpc-admin during setup. After logging in you and will be able to send commands to the server.

To verify that you are logged in try the command `cpcc server-info`. This should display the server name and version.

1.5 Worker Setup

1.5.1 Prerequisites

The worker has 2 prerequisites

- A client.cnx file, If you are running the worker on a different machine than the server you probably do not have a .copernicus folder in you home directory. However you can create one manually and drop in the client.cnx file there. If you wish you can also specify the file manually as we will later below.
- GROMACS must be installed and accessible for the worker. The easiest way is if GROMACS is accessible from the path. For other ways to specify this please refer to the section *Worker*.

1.5.2 Installation

Workers do not need any specific project directory. Provided that the prerequisites are met no installation procedure is needed. To verify that the worker can connect to a server start it with

```
cpc-worker smp
```

By default the worker looks in .copernicus for the connection bundle. however you can also specify the location onf the connection bundle.

```
cpc-worker -c client.cnx smp
```

When started the worker will output its Worker Id, available executables and then start requesting work from the server. An example output is shown below

```
INFO, cpc.worker: Worker ID: 130-229-12-163-dhcp.wlan.ki.se-26108.
Available executables for platform smp:
gromacs/mdrun 4.5.3
INFO, cpc.worker: Got 0 commands.
INFO, cpc.worker: Have free resources. Waiting 30 seconds
```

you will notice the parameter `smp` in the above command. This means that we start the worker with the platform type smp. We will cover this in greater detail in the section *Platform types*.

Shut down the worker simply hit `CTRL-C`

In case you try to connect with the wrong connection bundle the following error message will be displayed.

```
ERROR: [Errno 1] _ssl.c:503:error:14090086:
SSL routines:SSL3_GET_SERVER_CERTIFICATE:certificate verify failed
```

1.6 Summary

After going throught the the installation process you should have one running server with one worker connected to it.

you can check the status of the server with `cpc-server status` which will show you that the server is up and running and has one worker connected to it.

CHAPTER 2

Copernicus overlay network

2.1 Server

Servers are what manages your copernicus project. They are responsible for generating jobs and the monitoring of these. When you work with a project you use the `cpcc` command line tool to send messages to the server. The server will process this commands and setup your project and generate jobs for it.

the command line program for the server is called `cpc-server`

2.1.1 Where to run the server

Since the server is responsible for running all of your project it is advisable to deploy it on a machine that is up and running all the time. For example running the server on your laptop would not be a good idea for many reasons:

- You move your laptop around : When moving your laptop between location your machine gets assigned different ip addresses. Workers connected to this server would not be able to communicate with the server once the address changes.
- Laptops are not on all the time: You close the lid on you laptop, it runs out of batteries ....

2.1.2 Fault tolerance of projects

The server is very fault tolerant and handles your projects with great care. It regularly saves the state of each project. In case a server would shutdown or crash due to software or hardware failure you can simply restart the server and it will recover to the previous state. Jobs that are already sent out workers are also fault tolerant. The server can handle cases where the worker goes down or if the server itself goes down. This is done by the server heartbeat monitor. Whenever a server has sent a job to a worker it expects a small heartbeat message from the worker once in a while (default is every 2 minutes however this can be configured). If the server doesn't receive a heartbeat message it will mark that job as failed and put it back on the queue. The same procedure is actually used when a server goes down. Whenever it starts again it will go through its list of jobs sent out(referred to as heartbeat items). And see which ones has gone past the heartbeat interval time. These jobs that have timed out will then be put back on the queue.

Although the server is fault tolerant make sure to backup your project data to a second disk regularly. The server will not be able to recover a project from disk failures.

2.1.3 Creating a Copernicus network with many servers

- You want to share worker resources: If your workers are not being utilized at 100% you can share your workers by connecting to other copernicus servers. Whenever your server is out of jobs it will ask its neighbouring

servers for jobs. More on this in the section worker delegation.

- You are running too many projects for one server to handle: If you have too many projects for one server to handle you can offload it by running a second server on another machine. You can then connect the second server to the first and still share resources with worker delegation.
- Your workers are running on an internal network while the server is not. In cluster environments the compute nodes can only communicate with the head nodes so you would need a server running on the head node. However as soon as you start running projects the server will consume a bit of resources on the head node which is not advisable. A better setup is to run one server on the head node and connect it to a project server outside the cluster environment. The server on the head node will only be managing connections to the workers and pass these on to the project server.

Worker delegation

The concept of worker delegation allows servers to share work resources between each other. Whenever servers are connected worker delegation is enabled automatically. The server that the workers are connected to will always have the first priority and if there is work in its queue it will utilize its workers. However if there is no work in the queue it will ask its connected servers for work. This is done in a proioritized order. The order of priority can be seen with the command `cpcc list-nodes` . Servers can be reprioritized with `cpcc node-pri`.

Connecting servers

To connect two copernicus servers you need to send a connection request which in turn has to be approved by the other side.

A connection request can be sent to a server using the command `cpcc connect-node HOSTNAME`

```
 >cpcc connect-server server2.mydomain.com
Connection request sent to server2.mydomain.com 14807
```

By default a request is sent using standard copernicus unsecure port 14807. If the destination server has changed the unsecure port number you will need to specify it.

```
cpcc connect-server server2.mydomain.com 15555
```

After sending a connection request you can list it with `cpcc connected-servers`

```
>cpcc connected-servers
Sent connection requests
Hostname                      Port       Server Id
server2.mydomain.com          14807      b96add9c-aff5-11e2-953a-00259018db3a
```

Received connection requests needs approval before a secure communication can be established between 2 servers. To approve a connection request you use the `cpcc trust SERVER_ID command`. Only servers that have sent connection request to your server can be trusted. To see which servers that have requested to connect you can use `cpcc connected-servers`

```
>cpcc connected-servers
Received connection requests
Hostname                        Port       Server Id
server1.mydomain.com            14807      dc75c998-acf1-11e2-bfe2-00259018db3a
```

The list above specifies that we have one incoming connection request. The text string under the column “Server Id” is what we need to specify in the `cpcc trust` command.

```
>cpcc trust dc75c998-acf1-11e2-bfe2-00259018db3a

Following nodes are now trusted:
Hostname                        Port       Server Id
server1.mydomain.com            14807      dc75c998-acf1-11e2-bfe2-00259018db3a
```

after calling `cpcc trust` your server will communicate with the requesting server and establish a secure connection. To list the connected connected nodes simply use `cpcc connected-servers`

```
>cpcc connected-servers
Connected nodes:
Priority   Hostname                        Port       Server Id
0          server1.mydomain.com            14807      dc75c998-acf1-11e2-bfe2-00259018db3a
```

If you change the hostname or the ports of one server it will upon restart communicate to its connected servers and notify them on these changes.

Connecting to a server that is behind a firewall.

If one of the servers is behind a firewall, it is not possible to send a connection request directly. The workaround for this is to first create an ssh tunnel to the server behind the firewall. The procedure will then be.

1. Create an ssh tunnel to the firewalled server

```
ssh -f server_behind_firewall -L 13808:server_behind_firewall:13807 -L 14808:server_behind_firewall:1
```

the syntax 13808:server_behind_firewall:13807 means "anything from localhost port 13808 should be sent to server_behind_firewall port 13807 The port numbers 13807 and 14807 are the standard copernicus server ports. in case you have changed these setting please make sure that those port numbers are provided in the tunnelling command.

2. Send a connection request using the tunnel port.

```
cpcc connect-server localhost 14808
```

3. Approve the connection request

```
cpcc trust SERVER_ID
```

where `SERVER_ID` is the id of the server that sent the connection request. you can look it up with the command `cpcc connected-servers`. When a connection is established you no longer need the ssh tunnel.

User management

Copernicus has support for multiple users with access roles. Regular users have either full access or no access to a project. Super users (like cpc-admin) have access to all projects and may add other users using:

```
cpcc add-user username
```

A user may grant another user access to its current project by issuing

```
cpcc grant-access username
```

2.2 Worker

Workers are responsible for most of the computational work in copernicus. They act as simple programs connecting to a server and asking for work. Workers can have very different capabilites with regard to cpu capabilites, and what

programs they can execute. When you start a worker it will establish a secure connection to your server and announce the programs and their versions it can execute. The server will then match the capabilities of the worker to the available jobs in the queue. By default a worker will try to use all available cores on a machine however this can be configured.

2.2.1 Worker and Server communication

You can connect as many workers as you want to a Server. And the only thing you need to do this is a connection bundle. Workers and Server communication is one sided. It is always initiated by the Worker and the Server is only able to send responses back.

2.2.2 Automatic partitioning

Workers always try to fully utilize the maximum number of cores available to them. Thus they able to partition themselves to run multiple jobs at once. For example if you have a worker with 24 available cores it can run one 24 core job or one 12 core job and 12 single core jobs. As long as the worker has free cores it will announce itself as available to the Server. However, workers do not monitor the overall CPU usage of the computer they are running on, but assume that the CPU is not used for other tasks.

2.2.3 Limiting the number of cores

By default a worker tries to use all of the cores available on a machine. However you can limit this with the flag `-n`

```
cpc-worker smp -n 12
```

You can also define how the partitioning of each individual job should be limited with the flag `-s`. For example to limit your worker to use only 12 cores and only 2 cores per job you can do:

```
cpc-worker smp -n 12 -s 2
```

2.2.4 Running jobs from a specific project

A worker can be dedicated to run jobs from a specific project this is done with the flag `-p`

```
cpc-worker -p my-project smp
```

2.2.5 Avoiding idle workers

If you want to avoid having idle workers you can instruct them to shutdown after an amount of idle time. This is done with the flag `-q`

```
cpc-worker -q 10 smp
```

2.2.6 Specifying work directory

When a job is running Workers store their work information in work directory. This work directory is by default created in the same location as where the worker is started. If you want to specify another work directory you can do it with the flag `-wd`

```
cpc-worker -wd my-worker-dir smp
```

2.2.7 Platform types

Workers can be started with different platform types. The standard platform type is `smp`. This one should be used to run a worker on a single node. The platform type `mpi` should be used when one has binaries using OpenMPI. Any binary that you usually start with mpirun from the command line should use this platform type.

2.2.8 Executing workers in a cluster environment

Starting workers in a cluster environment is very straightforward. You will only need to call the worker from your job submission script. You only need to start one worker for all the resources that you allocate.

This is the general structure to use for starting a copernicus worker.

```
## 1.ADD specific parameters for your queuing system ##

## 2. starting a copernicus worker ##
cpc-worker mpi -n NUMBER_OF_CORES
```

Here is a specific version for the slurm queing system

```
#!/bin/bash
#SBATCH -N 12
#SBATCH --exclusive
#SBATCH --time=05:00:00
#SBATCH --job-name=cpc

#Assuming each node has 16 cores, 12*16=192
cpc-worker mpi -n 192
```

2.2.9 Best practices when using 1000+ cores

Copernicus is a very powerful tool and can start using thousands of cores at an instant. When starting large scale resources it is advisable to gradually ramp up the resource usage and monitor the project to see if any critical errors occur in the project or your cluster environment. If everything looks fine start allocating more and more resources.

CHAPTER 3

Tutorials

Contents:

3.1 Introductory Tutorial

This tutorial will get you introduced to the basics of copernicus. After finishing this tutorial you will be able to:

- Setup a copernicus network ready to run projects and distribute tasks for you.
- Create a project workflow for molecular simulations.
- Learn how to monitor project progress, fetch results and alter input data while a project is running.

3.1.1 Installation

The section getting-started of the User guide covers the installation of Copernicus.

Tutorial files can be found here

3.1.2 Creating a workflow for molecular simulations

In projects where one runs many variations of a simulation many small things add up and the process can become tedious. Imagine having to manually copy over all files to various machines, run the simulations and download them to another machine for postprocessing. When work adds up the possibility for errors in simulations or machine errors also increases.

In this example we will build a molecular simulation workflow that lets us handle multiple simulations. By providing a couple of input files we want this workflow to generate molecular simulation jobs. Copernicus will then automatically distribute these jobs to available worker units and monitor them. In cases things jobs go wrong they will be automatically regenerated and put back on the Copernicus queue. In some cases things go wrong and requires user intervention. Copernicus will in these cases halt that job and ask for a user to interact. All of this allows the user to focus on the problem instead of having to focus on what things should run where.

Workflow components are called functions. Functions operate on inputs and generate outputs. Outputs of one function can be connected to the input of another functions. Copernicus already has many ready made functions for Gromacs. We will utilize these to create our workflow.

Before we create a workflow we need to start a new project.

```
> cpcc start md_simulations
Project md_simulations created
```

You can now see the newly created project in the projects list.

```
> cpcc projects
Projects:
     md_simulations
```

The gromacs functions are available via the gromacs module We import it with

```
> cpcc import gromacs

> cpcc info gromacs

\.\.\.\.\.\.

\.\.\.\.\.\.

grompps
     Prepares multiple simulations; can handle multiple input
     files.Each input is an array that can have N or 1 elements,
     where N is the number of desired outputs. If the number of
     inputs is Neach input will go to its own instance of grompp.
     If the number of inputs in an array is 1, this will be
     distributed along all the grompp instances. The output is an
     array of .tprs.
mdruns
     Runs a set of MD simulations. Inputs are spread in the same
     way as grompps

\.\.\.\.\.\.
```

Among the functions you will see two called grompps and mdruns. These two functions can handle multiple simulations. lets start by creating one instance of grompps.

```
> cpcc instance gromacs::grompps grompp
```

This roughly translates to "create a workflow block that will use the grompps function from gromacs and lets call it grompp". Now we'll create an instance of the mdruns function:

```
> cpcc instance gromacs::mdruns mdrun
```

If we list the currently instantiated workflow blocks:

```
> cpcc ls
Network '':
Network function instances:
    grompp (held)
    mdrun (held)
```

we see that the our grompp and mdrun blocks are there an there status is "held", This means that they are currently inactivated and not running anything. When we have finished building the workflow we will activate them. We have now created an instance to each function, however they are not connected yet.

We will now connect the two block so it looks like this.

The output of grompp is tpr files, and we'd like grompp to pass these on to mdrun once it's produced.

First we call the command `cpcc transact`. This way we can send many commands at once and treat them as if they are a single update. When we are finished sending commands, we'll finish the transaction block with the command `cpcc commit`. Once the transaction has started, we connect the blocks. We do this by connecting the grompp tpr output to the mdrun tpr input.

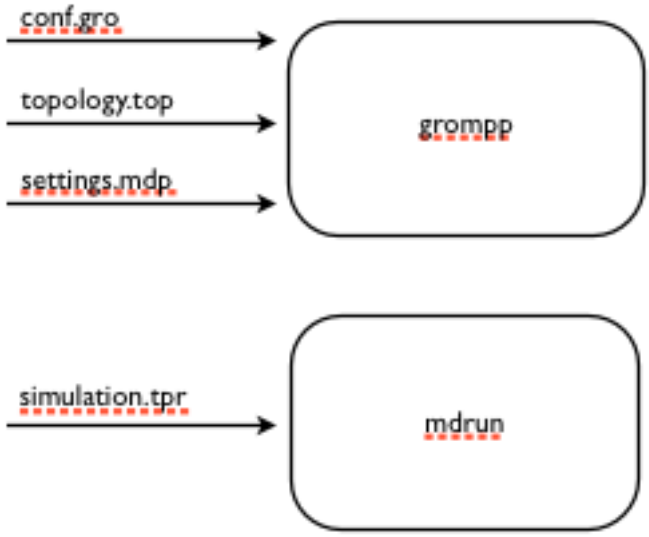

Fig. 3.1: The current progress. We have now defined our first blocks but we still have not connected them

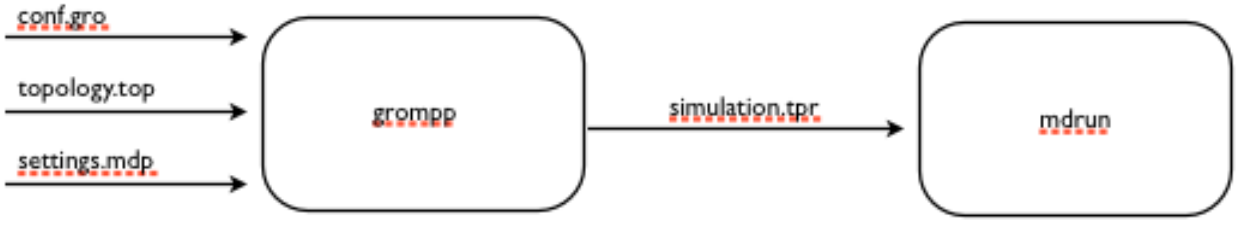

Fig. 3.2: The look of the workflow after we have connected our blocks.

```
> cpcc transact
> cpcc connect grompp:out.tpr mdrun:in.tpr
Scheduled to connect grompp.out.tpr to mdrun.in.tpr at commit
```

here you will notice syntax like grompp.out.tpr, These are called workflow paths. The next section we'll cover them in more detail. We can now activate all the workflow blocks. This means that they should start listening to incoming input.

```
> cpcc activate
```

We finish the transaction by calling

```
> cpcc commit
```

This tells Copernicus that it should start processing all the commands that we have called during this transaction.

At this stage we have finished building our workflow, and it's ready to take input.

We will now use our created workflow to start some simulations

3.1.3 Listing workflow functions

```
> cpcc ls
Network '':
Network function instances:
    grompp (active)
    mdrun(active)
```

Earlier when we used this command the functions where in a held state. This time we see that they are active meaning that they will react if we provide input. We can also take a deeper look at an individual function to see what inputs and outputs it handles. This is done by calling `cpcc ls` followed by the name of the function.

```
 > cpcc ls grompp
Instance 'grompp':
Instance of: gromacs::grompps
State: active
Inputs:
    conf
    mdp
    top
    ndx
    settings
    include
Outputs:
    tpr
Subnet function instances:
```

Function inputs types can be inspected this way

```
> cpcc ls grompp.in
Input/Output Value 'grompp:in':
Type: grompp:in
Sub-items:
    conf: conf_array
    mdp: mdp_file_array
    top: top_array
    ndx: ndx_array, optional
    settings: mdp_array_array, optional
    include: grompp_include_array_array, optional
```

This shows us that the conf, mdp and top inputs are array types, which means that we can provide multiple conf files, leading to multiple simulations in this case. Later in the project we will use this to actually start multiple simulations. We also see that the inputs ndx and settings have the description "optional", which means that the function can run without having set those inputs. We can also look at the output types in a similar way:

```
> cpcc ls grompp.out
```

To inspect or set the actual values, we can use the `cpcc set` and `cpcc get` commands. We will use these soon to provide input files to our projects and later take a look at the results.

3.1.4 Providing input files to our project.

To get things to run, we need to provide 3 input files: a configuration file, a topology file which describes the system that we want to simulate, and an mdp file which contain the simulation settings . These files are standard Gromacs file types, and for this tutorial you can find them ready made in the tutorials directory.

we will use the `cpcc setf` command to provide the input files for our grompp function.

```
> cpcc setf grompp:in.conf[+] conf.gro
Committing scheduled changes:
- Set grompp:grompp:in.conf[0] to _inputs/0000/conf.gro
```

The section `grompp:in.conf[+]` specifies where a file should be sent to. in this case we want to send it to the conf input. the last section `[+]` means "add this file". Remember when we did `cpcc ls grompp:in` and we saw that the conf input was an array? that is why we can add files by using the plus. lets add the topology file and the mdp file.

```
> cpcc setf grompp:in.top[+] topol.top
Committing scheduled changes:
- Set grompp:grompp:in.top[0] to _inputs/0001/topol.top

> cpcc setf grompp:in.mdp[+] grompp.mdp
Committing scheduled changes:
- Set grompp:grompp:in.mdp[0] to _inputs/0002/grompp.mdp
```

Our grompp block has now gotten enough information to generate an output, and send it to the mdrun block, The mdrun block will then send a simulation job to the work queue. By now the workflow should have gotten the input it needs to prepare a simulation. If we take a look at the queue we should see that an item should have appeared.

```
> cpcc q
Queue:
  0 mdrun:mdrun_0.1: gromacs/mdrun
```

This means server has generated a job and waiting for a worker to send it to. If the worker is still running it should receive this job within maximum 30 seconds. For the sake of this tutorial the simulation is very short and should be finished within a minute. In reality a simulation could take days. The worker would in these cases send back intermediate results to the server in one hour intervals.

3.1.5 Looking at the results

After finishing the last job, we should have some results to look at. The outputs can be found in the outputs of our mdrun block. We can simply download them to our computer by running a workflow query and direct its output to a file. for example downloading the trajectory file

```
> cpcc getf mdrun.out.xtc[0] > ~/trajectory.xtc
```

which would download the trajectory to our home directory. Again we see the square bracket syntax, this time with a digit instead of a plus sign. As we noted earlier some inputs and outputs are array types. In the case of the output this means that we can have multiple outputs. We specify which output we want to look at by specifying an index number.

3.1.6 Running more simulations.

Usually, we want to run more than one simulation to obtain more samples and trajectories – with each simulation having very similar settings. To run more simulations in our case, we can use some of the inputs we have already provided. We will simply provide a few new configurations. We can cheat, and provide the simulation the same configuration as before:

```
> cpcc setf grompp.in.conf[+] conf.gro
```

You can check the work queue to see the progress of this simulation. When it's done, try to call the command

```
> cpcc get mdrun:out.xtc
mdrun:out.xtc: [
  mdrun/mdrun_0/_run_0000/traj.xtc,
  mdrun/mdrun_1/_run_0000/traj.xtc
]
```

which now gives two trajectory files. We can fetch the latest simulation trajectory with

```
> cpcc getf mdrun:out.xtc[1] > ~/trajectory_1.xtc
```

3.1.7 Pausing a project

Projects usually run until you decide its finished you can always pause them temporarily by calling.

```
> cpcc deactivate
```

To start the project again you call

```
> cpcc activate
```

3.1.8 Finishing a project

At one time you might want to finish a project and move it away from the server. The command cpcc save will save your project and backup everything in a compressed file.

```
> cpcc save md_simulations
Saved project to md_simulations.tar.gz
```

`cpcc save` will also deactivate your project however it will not delete the project from the server. You will need to do that yourself.

```
> cpcc rm md_simulations
```

To load a saved projects you call

```
> cpcc load md_simulations.tar.gz md_simulations
Project restored as md_simulations
```

The project is restored but held in a deactivated state. you can start it with `cpcc activate`.

3.2 MSM Tutorial

In this tutorial we will try out the MSM workflow of Copernicus. Markov State models (MSM). This tutorial includes two examples where we fold proteins. The first one is Alanine dipeptide which is a very small system that can be run on pretty much an modern machine within a reasonable time. The second is fs-peptide which might take about 24 hours depending on the computing resources used.

3.2.1 Prerequisites

Make sure you have Gromacs and MSMBuilder installed. MSMBuilder needs to be installed on the same system that the server is running on. For unix machines there is an already prepared instance under examples/msm/msmbuilder-known-good.tar.gz

Gromacs needs to be installed on the system that the server is running on and all machines where you will run workers.

3.2.2 Learning the method

Although Copernicus automates MSM:s it will not automatically find the optimal parameters for you That is where you the user need to have a solid understanding of the method.

3.2.3 Overview of the workflow

The MSM workflow starts by spawning simulations from a set of provided starting conformations. Simulations continue until enough data is gatehered to start building MSM:s. The MSM building starts by first clustering all conformations using a hybrid K-means, K-medoids algorithm[REF]. The clustering metric used Is RMSD. After this a microstate MSM is built and then a macrostate MSM. At each MSM generation it is possible to view at the current macrostates. This process continues as many times as specified by the user. Usually it should be run until the MSM converges toward a state.

The MSM workflow is built using gromacs modules and MSMBuilder. At the moment it supports gromacs 4.6,gromacs 5.0 and MSMBuilder 2.0. To start the workflow the following input is neecessary.

3.2.4 Settings

num_microstates (int) The number of microstates.

num_macrostates (int) The number of macrostates.

lag_time (int) The lag time of the Markov state model

grpname (string) Defines what part of the system to use during clustering. Gromacs selection syntax is used here.

recluster (int) The reclustering frequency. Determines how often clustering should be done. Defined in nanoseconds. As simulation time has summed up to this amount a new generation of MSM building will take place.

num_generations (int) Number of MSM generations. After the defined amount of generations the workflow will finish.

num_sim (int) Number of parallel that will be run at each generation

confs (gro files) A set of starting conformations to use as initial seeds. If the number of simulations are higher than the starting conformations the workflow will simply choose starting conformation in a cyclic fashion.

grompp A collection of settings needed for grompp. The required parameters are the following:

- **mdp** A gromacs mdp file
- **top** A gromacs topology file

The optional parameters are the following

- **include** array of additional files. For example if the top file includes other files these have to be included here.

3.2.5 How to determine what parameters to set?

Number of microstates, macrostates and Lag time will be very specific depending on the system that you are analyzing. To determine what might work for your system you need to have a good understanding of the method. Other things that might affect the quality if the MSM is the simulation length and output frequency.

Example 1: Alanine dipeptide

Alanine dipeptide is a very small system that is possible to run on pretty much any computer. In this example we will build an MSM that will find the folded state of this peptide

For this tutorial we need some example files that are located under examples/msm-test in the Copernicus source. A set of files are prepared in this bundle. It includes the following.

1. 4 starting conformations equil0.gro,equil1.gro,equil2.gro,equil3.gro
2. Simulation settings in the file grompp.mdp.
3. A topology file.
4. The script runtest.sh. It includes a short script with all the commands necessary to get the project up and running. Each of the commands will be explained below.

We will first start a project

```
cpcc start alanine-dipeptide-msm
```

We then need to import the msm module, create a new instance of it and activate it.

```
cpcc import msm
cpcc instance msm:msm_gmx_adaptive msm
cpcc activate
```

Now we are going to provide all the necessary input. The workflow will start running as soon as all necessary input is provided. Since there is no lower limit on the number of starting conformations, it will start running as soon as it has at least one. In this case we do not want this. We want to provide 4 starting conformations thus we need to tell the workflow to wait until told to start. This is done with the transact command.

```
cpcc transact
```

Now we provide all the necessary input

```
cpcc setf msm:in.grompp.top examples/msm/alanine-dipeptide-msm/topol.top
cpcc setf msm:in.grompp.mdp examples/msm/alanine-dipeptide-msm/grompp.mdp

cpcc setf msm:in.confs[+] examples/msm/alanine-dipeptide-msm/equil0.gro
cpcc setf msm:in.confs[+] examples/msm/alanine-dipeptide-msm/equil1.gro
cpcc setf msm:in.confs[+] examples/msm/alanine-dipeptide-msm/equil2.gro
cpcc setf msm:in.confs[+] examples/msm/alanine-dipeptide-msm/equil3.gro

cpcc set msm:in.recluster 1.0
cpcc set msm:in.num_sim 20
```

```
cpcc set msm:in.num_microstates 100
cpcc set msm:in.num_macrostates 10
cpcc set msm:in.lag_time 2

cpcc set msm:in.grpname  Protein
cpcc set msm:in.num_generations 6
```

We finally commit the transact block.

```
cpcc commit
```

Copernicus will now start spawning simulations and put them on the queue.

Check the status of the project with `cpcc status`. This will inform you on the state of the project and how many jobs are in the queue and how many are currently running. If you want to see in detail what jobs are in the queue use the command `cpcc queue`.

After a short while (specify time) enough simulation data has been collected and MSM building will start

If you run the command `cpcc ls msm` you will see a new function instance `build_msm_0`. Traverse it and you will be able to fetch the macrostates and also a current max state which is the most likely end state for the system at the moment.

After a little longer you will notice that more build_msm instances will show up. Looking at the max states of these you will notice that after about 4 iterations we end up in the same state, meaning a folded state of the peptide has been found.

Example 2: fs-peptide

Fs-peptide is a little larger bit larger system that can be folded. However it might take a couple of days depending on the computing resources you have. For example on 4 32 core machines with 2 GPUS each it took about 24 hours to reach a folded state.

All the necessary files for fs-peptide can be found under examples/msm/msm-fs-peptide

3.3 Free Energy Tutorial

In this tutorial we will go through the Free Energy workflow of Copernicus. The tutorial includes two separate parts, demonstrating solvation free energy and binding free energy calculations, respectively.

3.3.1 Prerequisites

Make sure you have GROMACS installed before starting.

GROMACS needs to be installed on the system that the server is running on and all machines where you will run workers.

3.3.2 Learning the method

Although Copernicus automates large parts of the free energy calculation workflow, including lambda point placement, it will not automatically set all parameters for you. The user needs to have a solid understanding of the method, both for setting up the jobs and for critically assessing the results afterwards.

3.3.3 Overview of the workflow

The free energy workflow for solvation free energy calculations consists of a stepwise decoupling of the studied molecule in a solvent.

The binding free energy workflow consists of two parts. The first part is the decoupling of the molecule in a pure solvent (just as in the solvation free energy workflow) and the second part is the decoupling of the molecule bound to, e.g., a protein.

The calculations are made up of a number of iterations. The length of each iteration is based on the relaxation time input (20 * relaxation time) to ensure proper sampling. After each iteration the current precision (estimated error of the sampling) is compared to the requested precision. If the requested precision is not yet reached another iteration is added.

The distribution of lambda points can be automatically determined. The GROMACS tool g_bar cannot use input with different lambda point distributions for one calculation. If the distribution does not change, the results from multiple simulations will be used to calculate the estimated delta G (or delta F). Otherwise there will be separate estimations from different iterations and they will all be used to generate an average delta G, weighted by the estimated errors of the contributing values and the error is propagated from the terms.

The free energy workflow supports GROMACS 4.6 and GROMACS 5.0. However, the restraints arguments in the binding free energy workflow is only compatible with GROMACS 5.0. If using GROMACS 4.6 restraints must be specified in the mdp file instead and taken account for afterwards.

To start the workflow the following input is neecessary.

3.3.4 Settings

Solvation Free Energy

- **grompp** A collection of settings needed for grompp. The required parameters are the following:
  - **mdp** A GROMACS mdp file
  - **top** A GROMACS topology file
- **conf (gro file)** A coordinate file of the solvated molecule.
- **molecule_name** The name or group number of the molecule that is studied.
- **solvation_relaxation_time (int)** The estimated relaxation time for solvation in simulation time steps. A value of 0.1 ns is a good guide line if the solvent is water. If the simulation time step is 2 fs, that means that this value should be 50000.

The optional parameters are the following:

- **precision (float)** The desired precision in kJ/mol. The simulations stop when this precision is reached. The default value is 1 kJ/mol.
- **min_iterations (int)** A minimum number of iterations to perform. Even if the desired precision is reached there will at least be this many iterations performed.
- **optimize_lambdas (bool)** After each iteration (and the first setup stages) a lambda distribution is calculated. If this optimize_lambdas is False the lambda point distribution will still be calculated, but not used. If this is True the calculated lambda distribution will be used for the next iteration, if the number of lambda points changed or if the spacing between at least two lambda points differ more than the optimization_tolerance. The default is True.
- **lambdas_all_to_all (bool)** If this is set to True the delta H from each lambda state to all other lambda states will be calculated, instead of just to its neighbors. The default is False.

- **optimization_tolerance (float)** The tolerance (percent) for deciding when the difference between lambda spacing in two subsequent runs is so large that the new lambda values are kept, i.e., the lambda point distribution is optimized. If set to 0 lambda values will be optimized every iteration. The default value is 20.
- **stddev_spacing (float)** The target standard deviation spacing of lambda points in kT for lambda point spacing optimization. A higher value will give fewer lambda points, but might require more iterations than a lower value. This value is not used if optimize_lambdas is False. Default value is 1 kT.
- **n_lambdas_init (int)** The number of lamba points from which to start lambda optimizations. If optimizations are disabled this number of lambda points will be used for all iterations (unless lambdas _q, lambdas_lj or lambdas_ljq are used to specify a specific lambda point distribution). The lambda points will be evenly distributed between lambda 0 and 1. By default 16 lambda points are used to start with, but if the lambda point distribution of the system is very irregular more lambda points might be needed from the start to make a good optimization.
- **simultaneous_decoupling (bool)** If this is True Coulomb and Lennard-Jones are decoupled at the same time. Default is False.
- **lambdas_q (list of floats)** A list of lambda values for electrostatics decoupling. If this is set there will be no lambda point optimization for electrostatics.
- **lambdas_lj (list of floats)** A list of lambda values for Lennard-Jones decoupling. If this is set there will be no lambda point optimization for Lennard-Jones.
- **lambdas_ljq (list of floats)** A list of lambda values for simultaneous electrostatics and Lennard-Jones decoupling. If this is set there will be no lambda point optimization.

Binding Free Energy

- **ligand_name** The name of the ligand that will be decoupled while binding as well as free in solution.
- **receptor_name** The name of the receptor.
- **grompp_bound** Input values of the bound state for grompp, see *Solvation Free Energy*.
- **grompp_solv** Input values of the state free in solution for grompp, see *Solvation Free Energy*.
- **conf_bound (gro file)** A coordinate file of the bound state.
- **conf_solv (gro file)** A coordinate file of the state in solution.
- **binding_relaxation_time (int)** The estimated relaxation time for the bound configuration in simulation time steps. A value of 10 ns is a good guide line. If the simulation time step is 2 fs, that means that this value should be 50000.
- **solvation_relaxation_time (int)** See *Solvation Free Energy*.

Most of the optional parameters are described in *Solvation Free Energy*, but some of them are duplicated with different names for the bound and solvated states:

- **restraints_bound** An array of restraints on a ligand. Each element in the array can contain:
  - **resname** The name of the residue to restrain to.
  - **pos** Relative location to restrain to.
  - **strength** Coupling strength (in kJ/mol/nm).
- **precision**
- **min_iterations**
- **optimize_lambdas**

- **lambdas_all_to_all**
- **optimization_tolerance**
- **stddev_spacing**
- **binding_n_lambdas_init**
- **solvation_n_lambdas_init**
- **simultaneous_decoupling**
- **solvation_lambdas_q**
- **solvation_lambdas_lj**
- **solvation_lambdas_ljq**
- **binding_lambdas_q**
- **binding_lambdas_lj**
- **binding_lambdas_ljq**

3.3.5 How to determine what parameters to set?

Apart from the required input, in most cases only n_lambdas_init depends on the system, but 16 or 21 initial lambda points are enough in most cases.

3.3.6 Example 1: Hydration Free Energy of Ethanol

The calculation of free energy of solvation of ethanol in water is used to demonstrate how to use the module.

For this tutorial we need some example files that are located under test/lib/fe/ in the Copernicus source. A set of files are prepared in this bundle. It includes the following.

1. A starting conformation, conf.gro
2. Simulation settings in the file grompp.mdp.
3. A topology file, topol.top.
4. The script runtest.sh. It includes a short script with all the commands necessary to get the project up and running. Each of the commands will be explained below.

If running the script itself it must be executed from the copernicus root directory, e.g.

```
test/lib/fe/runtest.sh <projectname>
```

We first start a project with the name specified by the input argument to the script.

```
cpcc start $projectname
```

We then need to import the fe module, create a new instance of it and activate it.

```
cpcc import fe
cpcc instance fe::solvation fe
cpcc activate
```

Now we are going to provide all the necessary input. The workflow will start running as soon as all necessary input is provided. In order to be able to specify the input in any order we tell the workflow to wait until told to start, otherwise it would start when all required input is given, i.e. before fe:in.precision is set in the example below. This is done with the transact command.

```
cpcc transact
```

Now we provide all the necessary input, observe that solvation_relaxation_time is lower than recommended to enable short iterations in order to make it possible to view the output sooner than what would otherwise be possible.

```
cpcc set-file fe:in.grompp.top examples/fe/topol.top
cpcc set-file fe:in.grompp.include[0]  examples/fe/ana.itp
cpcc set-file fe:in.grompp.mdp examples/fe/grompp.mdp

cpcc set-file fe:in.conf examples/fe/conf.gro

cpcc set fe:in.molecule_name  ethanol
cpcc set fe:in.solvation_relaxation_time 500
cpcc set fe:in.precision 0.50
```

We finally commit the transact block.

```
cpcc commit
```

Copernicus will now start spawning simulations and put them on the queue.

Check the status of the project with `cpcc status`. This will inform you on the state of the project and how many jobs are in the queue and how many are currently running. If you want to see in detail what jobs are in the queue use the command `cpcc queue`. If no Copernicus worker is active no simulations will start.

The simulations will continue until the estimated error of the calculations is equal to or lower than the specified precision. If running only one worker this can take quite a while.

It is possible to check the output before the instance is finished. The following command will show you the current output:

```
cpcc get fe.out
```

To see just the currently estimated free energy of solvation run:

```
cpcc get fe.out.delta_f
```

When finished the delta_f should be approximately -19 kJ/mol, which is not too far from the experimental value of -20.93 $\pm$ 0.8 kJ/mol.

3.3.7 Example 2: Binding Free Energy of Ethanol to Ethanol

Binding free energies are most often calculated for a small molecule to a larger receptor. In this example we calculate the binding free energy of one molecule of ethanol to another molecule of ethanol.

For this tutorial we need some example files that are located under test/lib/fe/binding in the Copernicus source. A set of files are prepared in this bundle. It includes the following.

1. Starting conformations, solv/conf.gro and bound/conf.gro
2. Simulation settings in the files solv/grompp.mdp and bound/grompp.mdp.
3. Topology files, solv/topol.top and bound/topol.top.
4. An index file for the bound state, bound/index.ndx
5. The script runtest.sh. It includes a short script with all the commands necessary to get the project up and running. Each of the commands will be explained below.

If running the script itself it must be executed from the copernicus root directory, e.g.

    test/lib/fe/binding/runtest.sh <projectname>

We first start a project with the name specified by the input argument to the script.

```
cpcc start $projectname
```

We then need to import the fe module, create a new instance of it and activate it.

```
cpcc import fe
cpcc instance fe::binding fe
cpcc activate
```

Now we are going to provide all the necessary input. The workflow will start running as soon as all necessary input is provided. In order to be able to specify the input in any order we tell the workflow to wait until told to start, otherwise it would start when all required input is given. This is done with the transact command.

```
cpcc transact
```

Now we provide all the necessary input, observe that solvation_relaxation_time is lower than recommended to enable short iterations in order to make it possible to view the output sooner than what would otherwise be possible.

```
cpcc set fe:in.ligand_name  ethanol
cpcc set fe:in.receptor_name  ethanol2

# bound state
cpcc set-file fe:in.grompp_bound.top test/lib/fe/binding/bound/topol.top
cpcc set-file fe:in.grompp_bound.include[0]  test/lib/fe/binding/bound/ana.itp
cpcc set-file fe:in.grompp_bound.include[1]  test/lib/fe/binding/bound/ana2.itp
cpcc set-file fe:in.grompp_bound.mdp test/lib/fe/binding/bound/grompp.mdp
cpcc set-file fe:in.grompp_bound.ndx  test/lib/fe/binding/bound/index.ndx

cpcc set-file fe:in.conf_bound test/lib/fe/binding/bound/conf.gro

cpcc set fe:in.restraints_bound[0].resname ethanol2
cpcc set fe:in.restraints_bound[0].pos.x 0
cpcc set fe:in.restraints_bound[0].pos.y 0
cpcc set fe:in.restraints_bound[0].pos.z 0
cpcc set fe:in.restraints_bound[0].strength 1000

# solvated state
cpcc set-file fe:in.grompp_solv.top test/lib/fe/binding/solv/topol.top
cpcc set-file fe:in.grompp_solv.include[0]  test/lib/fe/binding/solv/ana.itp
cpcc set-file fe:in.grompp_solv.mdp test/lib/fe/binding/solv/grompp.mdp

cpcc set-file fe:in.conf_solv test/lib/fe/binding/solv/conf.gro


cpcc set fe:in.solvation_relaxation_time 1000
cpcc set fe:in.binding_relaxation_time 2000
cpcc set fe:in.precision 2
```

We finally commit the transact block.

```
cpcc commit
```

Copernicus will now start spawning simulations and put them on the queue.

Check the status of the project with `cpcc status`. This will inform you on the state of the project and how many jobs are in the queue and how many are currently running. If you want to see in detail what jobs are in the queue use the command `cpcc queue`. If no Copernicus worker is active no simulations will start.

The simulations will continue until the estimated error of the calculations is equal to or lower than the specified precision. If running only one worker this can take quite a while.

It is possible to check the output before the instance is finished. The following command will show you the current output:

```
cpcc get fe.out
```

To see just the currently estimated free energy of binding run:

```
cpcc get fe.out.delta_f
```

3.4 String-of-swarms method

3.4.1 Introduction

The string-of-swarms method is a method to iteratively refine an initial guess of a molecular system's transition between two or more known configurations, in order to find the transition that minimizes the reversible work (i.e. free energy) done by the system during the transition.

This and related methods are useful especially when the timescale on which a transition takes place is much longer than the timescale you can typically simulate in a molecular dynamics simulation, thus making it improbable to directly observe a sought-for transition spontaneously.

The method as implemented in Copernicus was originally described by Pan and Roux 2009 [add citation] building on the work of Maragliano2006 [add citation] and they demonstrated it on the di-alanine peptide isomerization transition, which has been well studied and due to its low complexity but high energy barrier is a good example for illustrating the string methods.

The method describes the transition using the evolution of collective variables (CVs), which form a reduced dimensionality description of the configuration, along a one-dimensional string from a starting point to an ending point.

The CVs can theoretically consist of any mapping from the full system dimensionality, but one common choice and the one which is implemented in the Copernicus module is the peptide bond dihedral angles phi and psi for a user-selected set of protein residues.

The method works by linearily interpolating the CVs between a given starting configuration and ending configuration guess for a selectable number of intermediate configurations, and then iteratively refining these configurations by starting a lot of unrestrained very short molecular dynamics simulations from the randomly perturbed intermediates and averaging their drift. Each iteration, the intermediates (and starting/ending points, if so desired) will move a bit down the free-energy gradient of the system's energy landscape.

This would quickly result in all intermediates falling down onto each other into the nearest local minima, and to prevent this, each iteration ends with a so-called reparametrization step, where the CV euclidean distances between the configurations along the string are equalized. In essence, the intermediate configurations are adjusted to lie equidistant from each other along the string, thus forcing an even coverage along the transition even if the string in the initial guess crosses one or more high energy barriers.

When the intermediates stop moving, the string has converged to a lowest possible energy state. A caveat is that this might or might not represent the transition of lowest energy globally speaking, since there could be many. Depending on the complexity of the system, it might be necessary to try different initial guesses of string configurations therefore and see if they converge to the same transition or not.

3.4.2 Prerequisites

Make sure you have GROMACS installed before starting.

GROMACS needs to be installed on the system that the server is running on and all machines where you will run workers.

3.4.3 Method protocol

The string module starts up by running the swarm Python script, which reads the initial string guess as input and other parameters, extracts the initial CVs and creates Copernicus functions for all iterations to be done.

Each iteration consists of the following interconnected Copernicus pipeline steps. The corresponding module Python script's name is given in parantheses.

Restrained minimization (run_minimization)

The restraints are from the previous iterations' updated string configuration (reparametrization step's output) and by minimizing using them, the system will now hopefully be forced into the new state of CVs.

Restrained thermalization (thermalization)

After minimization the system will be at 0 temperature. This step brings up the system in temperature again using the selected thermostat.

Restrained equilibration (run_restrained)

Runs the system for a while using the same parameters as the swarm step that follows, but for longer and with restraints enabled.

The output trajectory from the equilibration is written sparsely, with an interval selected so the written number of configurations equal the number of swarms to issue per string configuration below. We use the system's natural evolution during the restrained step here to seed the swarms.

Swarm preparation (prep_swarms)

The trajectories output in the previous step are converted to system configurations again, and sent to the next step.

Swarm (swarms)

Each configuration written for each equilibration trajectory above is used as starting point for a very short, unrestrained run. The runs belonging to the same string configuration are called a swarm, and the final configurations in each run in a swarm is sent as output to the next step.

CV extraction (get_cvs)

This step extracts the CVs from each swarm run configuration output. In the dihedral angles case for example, the peptide bond angles phi and psi are calculated for each protein residue selected to participate as CV.

Reparametrization (reparametrize)

The CVs that resulted from each swarm run are read and the swarms averaged to get a proposed drift for each string configuration.

An iterative algorithm is then executed that adjusts the updated configurations so they keep their distances along the string, to avoid them all falling down into the minimas in the start and end of the string (or in between).

The result is that each configuration along the string can move orthogonally to the string but stay fixed longitudinally. The endpoints (if not fixed) can move in any direction though.

3.4.4 Demo running

After starting a Copernicus server and making sure you can log into it using cpcc, in the copernicus folder run:

$ test/lib/swarms/swarm/runtest.sh

and it will setup a project called "test_singlechain" where the alanine dipeptide is setup to run to optimize its isomerization transition (described in the introduction above) for 80 iterations. If you just want to try if something works, edit runtest.sh first and in the Niterations setting near the bottom, change 80 to 5 or something.

To monitor the status of the run, you can do

```
$ cpcc status
```

to see if there are commands queued. If there are no more commands queued, you can proceed to extract the result by using these scripts, from within the test/lib/swarms/swarm folder:

```
$ mkdir res
$ cd res
$ ../get_result.sh
< this will extract the resulting string configuration from the cpc-server through cpcc >
$ cd ..
$ ./vis_string.sh
< this will make an .xtc from the final string transition and extract a table of the dihedrals along
```

3.4.5 Parameter reference

General settings

- run:in.fixed_endpoints (integer 0 or 1)

Option that controls whether the starting and ending point in the string should be fixed or also updated just like all the other points. If you are not completely sure that your initial starting or ending point are actually the true local minimas for the force-field used, it is recommended to try running with the endpoints not fixed.

- run:in.Ninterpolants (integer)

The number of points in the string, including the starting and ending point.

Usually 10-50 points are enough, but depends on the system and free-energy landscape. The total computational time, storage space and network bandwidth increases linearly with the number of points, and it might be good to start with a smaller number of points to evaluate the method for the system and then increase.

Since every string point that can move involves minimizing and running a system simulation, the number of moving string points should usually be correlated with the number of workers and CPU cores attached to the Copernicus server so an even fraction of the total workload can be run in parallel at every time.

For example, if you have 16 worker nodes attached it makes sense to use 16+2 (or 32+2) stringpoints in total if the endpoints don't move, and 16 stringpoints in total if the endpoints move.

- run:in.Niterations (integer)

The number of string iterations to run. Common numbers are from 10 to hundreds, this depends a lot on the system size and energy landscape. Note that this is correlated with the swarm simulation step count described further below - doing more steps each swarm iteration results in less iterations needed to evolve the string the same distance, but it will be less accurate and might result in the string not converging as it might "step over" the true minima.

Each iteration requires a certain amount of memory and disk resources at the Copernicus server, and for large systems, it makes sense to restrict the number of iterations scheduled in each invocation for many reasons, including giving the ability to monitor the convergence better and see if bad things happen to the evolving string point systems.

- run:in.top (.top)

The Gromacs topology file to use for simulating the string point systems.

- run:in.tpr (.tpr)

A Gromacs run-file corresponding to the topology above, which is needed by various Gromacs helper tools invoked by the module, like g_rama. It is never run, and it doesn't matter if it corresponds to a minimization or run simulation, as long as it comes from the same topology and base system.

- run:in.Nchains (integer)

Normally 1. For polymers, this should correspond to the number of separate peptide chains in the system topology. If you use polymers, and have separate .itp files for each subunit, you need to provide them in the in.include[] array.

- run:in.include (array of .itp, optional)

See above regarding polymers

- run:in.cv_index (.ndx)

The specification of the collective variables (CVs) that monitors and controls the string evolution. For the dihedral string case, this is the set of atoms with at least one atom listed per residue whose dihedral phi/psi angles should be used. It does not matter if the index contains one atom per residue or all atoms in the residues, and it accepts the output format used by the make_ndx Gromacs helper which might be useful for generating an index for large proteins.

There should be one single index group in the file only but the name of the group is irrelevant.

An example would be:

[CA_&_Protein] 7 27 46 60 74 88 94 117 129

which is generated by make_ndx, selecting the Ca atoms from each residue in the Protein, or the following index file used by the alanine dipeptide demo:

[r_2] 7 8 9 10 11 12 13 14 15 16

String specification

- run:in.start_conf (.gro)

The system configuration corresponding to the starting point of the string. This is also used by the dihedral CV mode as base configuration for all other string points.

- run:in.end_conf (.gro)

The system configuration corresponding to the ending point of the string. In the dihedral CV mode, this is unused (see start_conf above, which is used for all stringpoints).

- run:in.start_xvg (.xvg)

The dihedral CV values corresponding to the starting configuration of the string. Together with end_xvg, these are used by the module to linearily interpolate an initial estimate of all intermediary stringpoints. The starting point will stay locked to the start_xvg CV values during all iterations.

This file can either be created manually using known desired values for the CVs, or if a starting system configuration is available, g_rama can be invoked to generate an .xvg containing all dihedral phi/psi values for all peptide bonds in the system.

- run:in.end_xvg (.xvg)

The dihedral CV values corresponding to the ending configuration of the string. See start_xvg. The ending point will stay locked to the end_xvg CV values during all iterations.

Minimization stage settings

The minimization stage is run using the specification files below, with restraints on all CVs for the values generated by the interpolation (if this is the first iteration) or the values generated by the previous iteration's output. In both cases, the system will be forced a the new state by the restraints as opposed to simply staying locked still as is usually done in the minimization step in a simulation. This can pose a problem if the structure is complicated and the new CV values try to change it a lot. If there are issues, tweaks to the minimization grompp input (described below) are needed to improve minimization performance, as well as possibly tweaks to the following stages as well.

- run:in.minim_grompp.mdp (.mdp)

The Gromacs configuration file to use for minimizing your system. Use settings that you know are capable of producing a good energy minimization for the system in general.

- run:in.minim_grompp.top (.top)

The Gromacs topology to use for minimization (usually the same as the other steps' topologies)

- run:in.minim_grompp.ndx (.ndx, optional)

A Gromacs atom index file, if needed for simulations (for example if you specify atom groups in the .mdp which are specified in the .ndx file)

- run:in.em_tolerance (float)

Minimization tolerance to set during the minimization stage.

- run:in.minim_restrforce (float, optional)

The restraint k-value to use during the minimization stage for the CVs. It should be fairly large (500.0-4000.0 kJ/mol/rad^2 for the dihedral case for example) but ultimately depends on the protein and the amount of distortions of the starting string compare with the start_conf, or the amount of string point evolution per iteration.

If no value is given, 500.0 is used.

Thermalization stage settings

- run:in.therm_grompp.mdp (.mdp)

The Gromacs configuration file to use for thermalizing your system after minimization is done (system will start at 0 K). Use normal settings that work for running your system in general, including a proper thermostat. It is not recommended to use pressure coupling here as the temperature is not stabilized yet.

The step size should be set very conservatively at 0.5 fs for example, to help with structures that are difficult to minimize into their new configurations.

The number of steps required to thermalize depends on the step size and the thermostat time coefficient, but in general 1000-2000 steps or so is enough for a step size of 0.5 fs and a tau of 1.0 ps.

- run:in.therm_grompp.top (.top)

The Gromacs topology to use for thermalization (usually the same as the other steps' topologies)

- run:in.therm_grompp.ndx (.ndx, optional)

A Gromacs atom index file, if needed for simulations (for example if you specify atom groups in the .mdp which are specified in the .ndx file)

- run:in.therm_restrforce (float, optional)

The restraint k-value to use during the thermalization stage for the CVs. See the discussion for minim_restrforce above.

If no value is given, 750.0 is used.

Equilibration and Swarm stage settings

- run:in.equil_grompp.mdp (.mdp)

The Gromacs configuration file to use for running your system normally. Use settings that work for running your system in general, including a proper thermostat. Pressure coupling may be used for solvated systems.

This will be used for the equilibration stage together with restraints on all CVs, as well as on the swarm stage without restraints.

Note:The number of simulation steps will be set automatically by the module to use this configuration file for both equilibration and swarms.

- run:in.equil_grompp.top (.top)

The Gromacs topology to use for equilibration (usually the same as the other steps' topologies)

- run:in.equil_grompp.ndx (.ndx, optional)

A Gromacs atom index file, if needed for simulations (for example if you specify atom groups in the .mdp which are specified in the .ndx file)

- run:in.equil_restrforce (float, optional)

The restraint k-value to use during the equilibration stage for the CVs. See the discussion for minim_restrforce above.

If no value is given, 750.0 is used.

- run:in.restrained_steps (integer)

The number of simulation steps to equilibrate for using the equil_grompp.mdp settings.

- run:in.swarm_steps (integer)

The number of simulation steps used for the swarm run. This should be very short, often between 15-300 steps depending on the system size.

- run:in.Nswarms (integer)

The number of swarm simulations to issue for each string point during the swarm stage. They will be started from a selection of configurations extracted from the equilibration stage and their ending coordinates will be averaged to get the average drift of the string point. The averaging is needed to counteract the randomness induced by simulating a system at a non-zero temperature.

More swarm simulations per point is always better to average the thermal fluctuations, but require more simulation time, storage space and network bandwidth, which might be a concern if the system is very big.

In general, start with small values (10-20) to verify the method and string convergence, and if the stringpoints evolve erratically or with too much noise, increase the number of swarms.

3.5 Module Tutorial

This tutorial will walk through the steps required to create a simple copernicus module

Tutorial files can be found here

3.5.1 The components of a module

The components you need to create a copernicus module is

- An xml definition describing the inputs and outputs of the module.
- A run method that contains the logic of the module.
- A plugin for the executable that runs on the worker.

3.5.2 Defining the xml and the run method

We will create a module that takes integer and doubles their values

We first create a directory under cpc/lib that will hold these two files.

Create the directory cpc/lib/double

Create a file named _import.xml in this directory. The import xml is a shell that describes the input and output values and their types.

```
<?xml version="1.0"?>
<cpc>
    <!--
     A simple copernicus function that takes in an array of integers and doubles
     their values.
    -->

    <!--Inputs and outputs of our function will be Integer arrays
    Here we define the a type called int_array, and we specify that the
    member-type(contents) of the array should be ints
    -->
    <type id="int_array" base="array" member-type="int"/>

    <function id="double_value" type="python-extended">
        <desc></desc>
        <inputs>
            <!--each field has a unique id and a type-->
            <field type="int_array" id="integer_inputs">
                <desc>Integer inputs</desc>
            </field>
        </inputs>
        <outputs>
            <field type="int_array" id="integer_outputs">
                <desc>Integer outpus</desc>
            </field>
        </outputs>
        <!-- when this function is called it will call the python function
             defined below. The path is the same as when one imports a module in python
             we also specify that we want to create a persistent directory for this module
             we will use this to keep track of states.
            -->
```

```
        <controller
                function="cpc.lib.math.double_script.run"
                import="cpc.lib.math.double_script"
                persistent_dir="true"/>
    </function>
</cpc>
```

The xml contains comments that describes the different tags; type and function. There are 5 basic types; int, float, file, array, string. For our module we only want to have arrays of integers as inputs and outputs so we have created a custom type called int_array. You see that it's base type is defined as array and it's member-type is defined as int.

The function tag describes the input and output values of the module as well as a controller which links it to the actual method that will be executed. The controller contains two attributes

1. function, which points to the method that will be executed
2. import, which is the python file that contains the method

The function attribute is set to cpc.lib.double.run. notice the the method in run.py is also called run the import attribute is set to cpc.lib.double as it the python package holding the method. functions and imports are defined the same way as you import packages and files in python.

Now create a file name run.py under the same directory. It will contain the logic that will be executed once input values are added or updated. In this method we can:

- Create commands to send to workers
- Define logic for what happens when inputs are set or updated
- Create new module instances

```
import cpc
from cpc.dataflow import IntValue, Resources

__author__ = 'iman'

import logging


log=logging.getLogger(__name__)

#our run functions
#the incoming value is of type cpc.dataflow.run.FunctionRunInput
def run(inp):
    if inp.testing():
        ''' When an instance of a function is first created a test call is performed
        here you can test to see if certain prerequisites are met.
         for example if this function is ran on the server only it might need to access some binaries
        return

    fo = inp.getFunctionOutput()
    #get hold of the inputs
    # the name that is provided must match the id of the input in the xml file
    #find what changed or was added in the array
    val = inp.getInputValue('integer_inputs')
    updatedIndices = [ i for i,val in enumerate(val.value) if val.isUpdated()]

    log.debug(updatedIndices)

    # only one command is finished per call
    if inp.cmd:
```

```
        runResultLogic(inp,updatedIndices[0])
        return

    #run some login on the changes

    for i in updatedIndices:
        #THIS IS WHERE WE SHOULD PUT OUR LOGIC
        runLogic(inp,i)

    return fo


def runLogic(inp,i):

    #Sending a job for a worker to compute
    val = inp.getInputValue('integer_inputs')
    arr = inp.getInput('integer_inputs')
    #1 create command
    storageDir = "%s/%s"%(inp.getPersistentDir(),i)

    #the command name should match the executable name of the plugin
    commandName = "demo/double"

    args = [arr[i].get()]

    cmd =cpc.command.Command(storageDir
                             ,commandName
                             ,args)


    #2 define how many cores we want for this job
    resources = Resources()
    resources.min.set('cores',1)
    resources.max.set('cores',1)
    resources.updateCmd(cmd)


    #2 add the command to the function output --> will be added to the queue
    fo = inp.getFunctionOutput()
    fo.addCommand(cmd)



def runResultLogic(inp,index):

    #in this case we are getting the result directly from stdout
    # stdout = "%s/%s/stdout"%(inp.getBaseDir(),inp.cmd.dir)
    stdout = "%s/stdout"%(inp.cmd.getDir())
    with open(stdout,"r") as f:
        result = int(f.readline().strip())
        log.debug("result is %s"%result)
        fo = inp.getFunctionOutput()
        fo.setOut("integer_outputs[%s]"%index,IntValue(result))

    return fo
```

the run method does three things. First it grabs the input value, it uses this input value to create a command that will be put on the copernicus queue and sent to a worker. Lastly it handles the returned data from the worker and sets it to

the output.

3.5.3 Defining the worker plugin

We will also need to create a plugin which is what the worker runs The plugin is just an xml which defines a list of executables that this plugin can run. An executable can be anything that can be run on the command line.

There are two ways two create the xml. You can either define a static one named executable.xml or a dynamic using a python script both have the same output and should be located in cpc/plugins/executables.

create a folder called double under cpc/plugins/executables. And add the executables.xml file under it.

```
<?xml version="1.0"?>
<executable-list>

    <!--the executable has a name which it matches to the command name on the queue, an executable m
    <!--different types of platforms, for example smp or mpi. A version of the executable can be defi
    <!--when creating a command to specify the minimum version required-->
    <executable name="math/double" platform="smp" arch="" version="1.0">
        <!--the command that the executable will call is defined here. you can define a command, scri
        <!--or a program in the same way that you call it on the command line-->
        <run in_path="yes" cmdline="double.py"/>
    </executable>
</executable-list>
```

this xml calls the command double which is a python script that we have created. to make this runnable by a worker, either change the cmdline attribute to specifiy the absolute path to where the script is located, or add the path to your PATH environment variable.

We know have everything ready for our first module! Start up the copernicus server. To see if the module has loaded properly call cpcc list-modules. You should see a module name double.

Now you can create a project and start using the module. this premade script will create an instance of the module and start adding values.

CHAPTER 4

Developer Documentation

4.1 Submitting Pull requetsts

4.2 Writing Documentation

The documentation is located in the docs folder.

The format is restructuredText and the documentation generator is Sphinx

Some useful material to get started

- Restructured Text (reST) and Sphinx CheatSheet
- reStructuredText Primer

insatll the rtd theme pip install sphinx_rtd_theme

The documentation can be generated by going to the doc folder and run `make html`. This will generate html docs with the same theme as readthedocs.org. To open it up locally simple open up the file `doc/_build/html/index.html`

Documentation that has been commited will show up in readthedocs.org.

CHAPTER 5

Introduction

Copernicus is a peer to peer distributed computing platform designed for high level parallelization of statistical problems.

Many computational problems are growing bigger and require more compute resources. Bigger problems and more resources put other requirements such as

- effective distribution of computational work
- fault tolerance of computation
- reproducability and traceability of results
- Automatic postprocessing of data
- Automatic result consolidation

Copernicus is a platform aimed at making distributed computing easy to use. Out of the box it provides.

- Easy and effective consolidation of heterogeneous compute resources
- Automatic resource matching of jobs against compute resources
- Automatic fault tolerance of distributed work
- A workflow execution engine to easily define a problem and trace its results live
- Flexible plugin facilites allowing programs to be integrated to the workflow execution engine

This section will cover how copernicus works and what its possibilites are. The subsequent sections will in detail cover how to use the platform.

5.1 The architecture of Copernicus

Copernicus consists of four components; the Server, the Worker,the Client and the Workflow execution engine. The server is the backbone of the platform and manages projects, generates jobs (computational work units) and matches these to the best computational resource. Workers are programs residing on your computational resources. They are responsible for executing jobs and returning the results back to the server. Workers can reside on any type of machine, desktops, laptops, cloud instances or a cluster environment. The client is the tool where you setup your project and monitor it. Actually, nothing is running on the client ever. It only sends commands to the server. This way you can run the client on your laptop, startup a project, close your laptop, open it up some time later and see that your project has progressed. All communication between these three components is encrypted. And as you will see later all communication has to be authorized.

Copernicus is designed in a way so that any individual can set it up, consolidate any type of resource available and put them to use. There is no central server that you will communicate to. You have full control of everything.

The workflow execution engine is what allows you to define your problem in a very easy way. Think of it as a flowchart where you define what you want to do and connect different important blocks. The workflow resides in the server and is a key component in every copernicus project. By just providing a workflow with input the server will automatically generate jobs and handle the execution of those. This way you will never have to focus on what to run where. Instead you can just focus on defining your problem.

The workflow gives also gives you the possibility to trace your work progress and look at intermediate results. You will also be a able to alter inputs in the middle of the run of a project in case things have gone wrong or if you want to test another approach.

Workflow components can actually be any type program. And with the plugin utilites in copernicus you can define these programs as workflow items, also known as functions.

5.2 Authors

5.2.1 Head authors & project leaders

Sander Pronk Iman Pouya (Royal Institute of Technology, Sweden) Peter Kasson (University of Virginia, USA) Erik Lindahl (Royal Institute of Technology, Sweden)

5.2.2 Other current developers

Magnus Lundborg (Royal Institute of Technology, Sweden) Björn Wesen (Royal Institute of Technology,Sweden)

5.2.3 Previous developers and contributors

Patrik Falkman Grant Rotskoff (University of California, Berkley, US)