

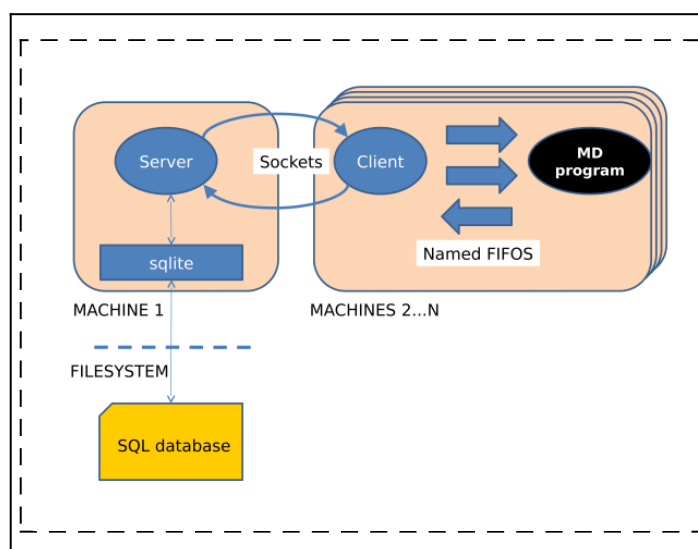
# Documentation

## Introduction

FRESHS is intended to bring together the many rare event sampling methods which are appearing in the literature, making them easy to test against each other or to use for practical purposes together with efficient simulation codes. FRESHS DOES NOT REQUIRE RECOMPILATION OF YOUR SIMULATION PROGRAM.

FRESHS is organised to run in parallel, with small trajectory fragments being farmed out to multiple instances of your chosen simulation program. Communications take place using sockets (between the FRESHS server and clients) and using named pipes (between the FRESHS clients and simulation programs) (Figure 1). If your OS or filesystem don't support pipes it is possible to communicate with the simulation programs directly through writing and reading to files instead. This, however, imposes a performance penalty. It is documented that NFS filesystems may have problems with named pipes.

FRESHS is designed as a harness system, so it should be easy to wrap around any given simulation program, without changing the FRESHS code itself. Application-specific stuff is contained in the "harness" directory. Please refer to [harness scripts](#) for examples.



## Server description

Normally you will run one server (the program that actually knows the sampling algorithm) and many clients (the programs which accept simulation jobs and run them, then report back to the server).

Always start the server first (see also the [quickstart](#) page):

```
cd server
```

```
python main_server.py -c server-sample.conf
```

there are a few options for the server, the most important argument is the configuration file.

The configuration file is broken up into sections; it has the structure:

```
[General]
an_input_variable = 1
another_variable = 2

[Another_section]
another_variable = False
```

Depending on the algorithm in use, different sections will be read or ignored.

## Configuration file options: general server options

```
[general]
# the port to listen on
listenport = 10000
# algorithm name, e.g. ffs, spres
algo_name = ffs
# warn or disconnect, if client was not seen since this amount of seconds
check_alive = 3600
# if this is set to 1, clients will be kicked if they do not report
# something during the check_alive interval
kick_absent_clients = 0
# interval for periodic info and check
# (clients are checked for timeout in this interval)
t_infocheck = 15
# use ghost runs for precalculation
use_ghosts = 0
# Directory structure
folder_out  = OUTPUT
folder_conf = USERCONF
folder_db   = DB
folder_log  = LOG
# User-defined message string for the clients in the form of a python dict
# Do not use curly brackets. Use double quotes.
# example: "servername": "default-server", "pressure": 50
user_msg = "servername": "default-server"
```

## Configuration file options: “spres\_control”

This heading groups together options specific to the SPRES algorithm. This section will not be read unless the option “algo\_name” in the section “General” is set to “spres”.

### [spres\_control]

*# The SPRES algorithm in its 'vanilla' flavour does not monitor reaction fluxes. If a flux across the hyperplane defined by the reaction coordinate value  $B$  is desired, then the value 'test\_absorb\_at\_B\_every' must be set non-zero, typically 1. This is because reaction flux is defined as the rate of first crossings of  $B$ ; so trajectories must be interrupted and checked for crossing after every timestep if a reaction flux is desired. See also the entry 'test\_absorb\_at\_B\_after\_bin' in connection with this setting*

`test_absorb_at_B_every = 0`

*# If the variable 'test\_absorb\_at\_B\_every' is set non-zero then this can lead to expensive, repeated, evaluations of the reaction coordinate at very short intervals, even far from the hyperplane  $B$ . Setting 'test\_absorb\_at\_B\_after\_bin' as greater than zero will turn off this repeated evaluation for bins below the index specified. You should have some understanding of the timescales of your system before setting this variable*

`test_absorb_at_B_after_bin = 0`

*# sets the length of trajectory fragments in the SPRES algorithm*

`tau = 100`

*# sets the maximum time covered during an SPRES run*

`max_epoch = 1000`

*# maximum number of shots made from a given bin at a given timestep in a SPRES calculation. A high value of this gives maximum flexibility to the importance sampling component of the algorithm to achieve an even number of transitions over phase-space/time; however in practice it is sometimes better to have a bound on the cost of your computation. This is achieved by setting this value relatively low. If your system is not achieving good coverage of transitions between bins, often it is better to decrease the bin size (see the section on Hypersurfaces) rather than increasing the ceiling of shots per bin*

`max_shots_per_bin = 10`

*# sets the typical sampling density of interface crossings per timestep 'tau'. Logically, this cannot be greater than the 'max\_shots\_per\_bin'*

`target_forward = 2`

*# use several databases to store the information*

`use_multDB = 0`

## Configuration file options: "ffs\_control"

### Basic options

#### [ffs\_control]

*# require number of points on interface.*

*# If 0, only the number of runs which reached the interface is used*

`require_runs = 1`

*# minimum points per interface to proceed*

`min_success = 2`

*# If this option is enabled, the clients must support max\_steps*

```
parallel_escape = 1
# if parallel_escape is 1, the following number of simulation steps
# are used for the escape run
escape_steps = 10000
```

## Advanced options

```
# configpoints must have at least this number of different origin
# points when tracing back to first interface
min_origin = 5
# fraction of successful different traces from interface to interface
min_origin_decay = 0.3
# try to increase the number of points this many times
min_origin_increase_count = 2
# maximum ghost point transfers in a row between real runs
# (decrease this number, if you use ghosts and the server slows down on
# an interface change)
max_ghosts_between = 3
```

## Configuration file options: section “Hypersurfaces”

This heading selects values of the reaction coordinate which define hypersurfaces in the phase space of the system. The reaction coordinate is calculated from the configuration (potentially including velocities) in way which is inherently system-dependent; example means of doing this are found in the various ‘harness’ scripts.

Option	Default	Description
lambdacount	2	This variable sets the number of non-boundary interfaces. The total number of interfaces will be equal to this value +2.
borderA	0.0	This variable sets the reaction coordinate value which defines the lowest boundary interface (be it open or absorbing) of the system. RC values $\lambda \leq A$ are considered to be in ‘region A’ of the phase space.
borderB	10.0	This variable sets the reaction coordinate value which defines the highest boundary interface (be it open or absorbing) of the system. RC values $\lambda > B$ are considered to be in ‘region B’ of the phase space.

## lambdas

```
lambda1 = 1.0
lambda2 = 2.0
```

The variables ‘lambda1’, ‘lambda2’ . . . etc set the positions of the non-boundary interfaces. RC values  $\lambda \leq \text{lambda2}$ ,  $\lambda > \text{lambda1}$  are considered to be in the ‘ $\lambda_1 - \lambda_2$ ’ region.

## Configuration file options: section “Runs\_per\_interface”

```
borderA = 10
borderB = 15
lambda1 = 20
lambda2 = 20
.
.
```

This section is read for both SPRES and FFS, but has different meanings. Each entry sets the initial number of runs per interface - the number of shots made from within the region  $\lambda N - \lambda N + 1$  the first time that this region is found to contain a configuration. After the first shots from the region, the importance sampling component of the SPRES algorithm will then adjust this number up or down. In FFS, each entry sets the total number of shots which are required to reach the interface  $\lambda N$  before beginning to make shots which themselves begin from this interface.

## Configuration file options: Automatic, optimized interface placement options

```
[auto_interfaces]
# use automatic interface placement
auto_interfaces = 0
# minimal distance between interfaces
auto_mindist = 0.001
# order parameter is integer
auto_lambda_is_int = 0
# maximum number of calculation steps for exploring runs
auto_max_steps = 10000
# number of trials
auto_trials = 20
# number of runs per newly placed interface (M_0-counter)
auto_runs = 30
# minimum fraction of desired points on last interface before starting
explorer runs
auto_min_points = 0.95
# minimum acceptable estimated flux
auto_flux_min = 0.3
# maximum acceptable estimated flux
auto_flux_max = 0.6
# moveunit for the trial interface method
auto_moveunit = 0.25
# use exploring scouts method. Clients must support this by returning
max(reaction_coordinate).
auto_histo = 0
```

## Client description

You can start as many clients as you want by repeating the following command:

```
cd client
python main_client.py -c client-sample.conf
```

The executable and the harness script for the simulation are thereby specified in the configuration file:

### Configuration file options: general client options

```
[general]
# the host to connect to
host = localhost
# the port to listen on
port = 10000
# ssh tunnel
ssh_tunnel = 0
# ssh tunnelcommand
ssh_tunnelcommand = ssh -N -L 10000:localhost:10000 tunneluser@tunnelhost
# refuse to accept new jobs after timeout(hours)
timeout = 0.0
# the executable which should be called (no quotes!)
executable = /usr/bin/espresso
# the harness to use,
# location of the harness dir where the 'job_script' is located (no quotes!)
harness = ../harnesses/espresso
# give number of lines to expect in first line of input fifo to simulation
nlines_in = 0
# set niceness of executable, 0 = disable
nice_job = 0
```

The client program has been kept as simple and lightweight as possible. Its function is to start a simulation program, relay initial coordinates from the server program to the simulation program, then collect outputs and return them to the server.

When you are running clients over the network for the first time, it is easy to check that you have the correct address and port for the server, first start the server, then go to the client machine and type:

```
echo -n "hello" | netcat mycomputer.uni.edu 10000
```

where "mycomputer.uni.edu" is the address of the machine running the server program and "10000" is the port. If you have the right address, then the server should interpret the "hello" as a client trying to connect, and print something to screen. If nothing happens, check the address of the server machine using ifconfig.

If you are sure that the address is right, try running it on a few different ports. If you still can't get through then you have some kind of firewall problem and should speak to your network administrator.

## Simulation Programs

FRESHS is intended to be easy to use with any simulation program. The interfaces between FRESHS and each simulation program are contained in the directory **harnesses**. Each subdirectory (e.g. **harnesses/espresso**, **harnesses/lammps**) contains a script **job\_script** plus optional data files it needs.

The script is called by the client program based on the information passed to it from the server. Input and output filenames passed to the scripts are actually **FIFOs** (also called 'named pipes'). These can be treated just like files, with the exception that they do not support seeking or rewinding - data must be read or written to them strictly sequentially, just like rolling tennis balls down a drainpipe.

### job\_script arguments

Here is a detailed specification of arguments that the job\_script should process (see also [example harness scripts](#)):

option	value description
-tmpdir	temporary folder for simulation run
-initial_config	initial configuration file
-in_fifoname	input fifoname
-back_fifoname	output fifoname
-metadata_fifoname	metadata fifoname
-halt_steps	use halt steps
-check_rc_every	check reaction coordinate after a number of simulation steps
-A	the border of the initial state A in terms of the reaction coordinate
-B	the border of the final state B in terms of the reaction coordinate
-random_points	configuration point to start the simulation from
-seed	a random seed, which is also written back to the database for reproducing simulation runs
-next_interface	next interface location in terms of the reaction coordinate
-act_lambda	actual lambda value
-jobtype	jobtype of the simulation, e.g. FFS escape flux or SPRES run
-rp_id	id of the runpoint
-max_steps	maximum number of steps to calculate. If this number is set, the calculation should be interrupted after this number of steps
-clientname	name of the client
-timestamp	timestamp of the simulation
-uuid	uuid of the simulation run

### job\_script argument parsing

## BASH

This method loops over the arguments and sets the variables directly using eval (use with caution). The IFS must be set if values containing spaces are expected.

```
#!/bin/bash
OLDIFS=$IFS
IFS=$'\n'

varmod=0
for el in $*;do
    if [ $varmod == 0 ];then
        varset=`echo $el | sed s/^-//g`
        varmod=1
    else
        eval "$varset=\"\$el\""
        varmod=0
    fi
done

IFS=$OLDIFS
```

## TCL

This script loops over all (argument,value) pairs and sets the variables accordingly.

```
foreach {option value} $argv {
    switch -glob -- $option {
        -argument1      {set argument1 $value }
        -argument2      {set argument2 $value }
        default          {puts "Additional not-used parameter $option"}
    }
}
```

## Python

This code builds a dictionary called **optval**, which contains all arguments index by their keys. Alternatively, an option parser like argparse could be used.

```
import sys
import re

optval = {}
for eli in range(len(sys.argv)):
    if (eli+1) % 2 == 0:
```



```
optval[re.sub('-', '', sys.argv[eli], 1)] = sys.argv[eli+1]
```

## Using the SQLITE Database of Trajectory Fragments

A database is the standard tool used in business applications for managing large quantities of data. It has some advantages over storing data directly onto the filesystem, as is more commonly done for simulation applications, in that it is platform-independent and in that it provides various convenient tools for searching through the databases, and for using memory and disk in a smooth “just works” sort of way.

The final result of a FRESHS calculation will be a lot of log files, some algorithm-dependent data about reaction rates and statistical weights, and a large sqlite database. To look at the database directly, the free tool “sqlitebrowser” is recommended. A firefox plugin also exists for this purpose, the choice of interface is up to the user. To process the database in an organized way, a couple of python scripts are provided with FRESHS (see directory scripts). Looking at these scripts should make it possible for you to write your own. You should find SQL to be a very relaxing and transparent idiom to work in, either natively or through the python bindings as is done in the example scripts.

From:

<http://www.freshs.org/dw/> - **The Flexible Rare Event Sampling Harness System**

Permanent link:

<http://www.freshs.org/dw/doku.php?id=freshs:documentation>

Last update: **2013/05/06 14:18**

