pmclib Documentation

Release 1.0

K. Benabed

December 17, 2010

CONTENTS

Pmclib consists of a set of C libraries and command lines tools to perform monte-carlo explorations of likelihood functions. A few helpers libraries (error management, parameter file parsing) as well as a post-processing python tool are also provided.

Contents:

CHAPTER

ONE

HOW TO INSTALL?

1.1 Prerequisites

Mandatory prerequisite, needed to compile the pmclib library:

- The Gnu Scientific Library (version >=1.10). Only the random number generator and a few simple linear algebra routines will be used.

Optional prerequisites, needed to compile the **compute_lkl**, **vanilla_importance**, **vanilla_pmc** and **vanilla_amcmc** command-line tools:

- lua (version >=5.1.4) is used to parse the parameter files
- hdf5 (version >=1.8.4) is used as the output format for the chain and importance samples.

Optional prerequisites, needed to compile the **importance_mpi**, **pmc_mpi** and **vanilla_grid** command-line tools:

- Some MPI distribution, as long as it provides the **mpicc** compilation tool, for example, openmpi.

Optional prerequisites, needed for the python post-processing module `pmctool`:

- numpy (version>=1.3).
- matplotlib (version>=0.98).
- h5py (version>=1.3).

1.2 Compilation & install of the C libraries and tools

Pmclib uses the waf build system to compile the library and command-line tools. The python module `pmctool` is installed using the usual python tools.

Waf does both the configuration and compile/install steps.

1.2.1 Configure step

To configure, at the root of the pmclib source distribution, type the following command

```
$ ./waf configure <SOME OPTIONS>
```

Waf will check whether your system have all the needed libraries, and will decide which part of the pmclib distribution to compile. At this step, waf will also decide where to install the codes (default is /usr/local/lib). Behavior of the configure step can be modified using the following command line options:

compilation options

-m32
compile & link in 32bits

-m64
compile & link in 64bits

install options

-prefix=<PREFIX>
set install directory to PREFIX (default is /usr/local/)

-local
install in current directory

describing where external libraries are located

-gsl_islocal
gsl has been installed with installgsl

-gsl_prefix=<GSL_PREFIX>
gsl include/lib path prefix

-gsl_include=<GSL_INCLUDE>
gsl include path

-gsl_lib=<GSL_LIB>
gsl lib path

-lua_islocal
lua has been installed with installlua

-lua_prefix=<LUA_PREFIX>
lua include/lib path prefix

-lua_include=<LUA_INCLUDE>
lua include path

-lua_lib=<LUA_LIB>
lua lib path

-hdf5_islocal
hdf5 has been installed with installhdf5

-hdf5_prefix=<HDF5_PREFIX>
hdf5 include/lib path prefix

-hdf5_include=<HDF5_INCLUDE>
hdf5 include path

-hdf5_lib=<HDF5_LIB>
hdf5 lib path

1.2.2 Optionaly download and compile external libraries

During configuration, if one of the external libraries (gsl, lua and hdf5) is missing, waf will propose to dowload and install it. The libraries will be installed in the install path defined by a first run of the configuration step (see the *--local* and *--prefix* options). To do so, for each library to be installed, after a first configuration step, run:

```
$> ./waf installXXX
```

where XXX is one of gsl, lua or hdf5.

After installing one (or more) external library, you will have to reconfigure waf accordingly with the *--hdf5_islocal*, *--gsl_islocal* and/or *--lua_islocal* options:

```
$> ./waf <YOUR OPTIONS> [--hdf5_islocal] [--gsl_islocal] [--lua_islocal]
```

Waf will not propose to install MPI

1.2.3 Compiling

To compile run:

```
$> ./waf
```

1.2.4 Installing

To install run:

```
$> ./waf install
```

This command will only install the C library and tools. The includes, libraries and `exectools` will be installed in the `include`, `lib` and `bin` path relative to the install path defined during the configuration step (see the *--local* and *--prefix* options). Furthermore, a `share/pmclib/example` directory will contain an example plugin library along with some example parameter files for the pmclib tools.

1.3 Compilation & install of the python module (`pmctool`)

At the root of the pmclib distribution, do:

```
$> python setup.py install
```

This is the regular python install procedure, and follows the usual python install rules.

CHAPTER

TWO

PMCLIB DISTRIBUTION CONTENT

Assuming all the optional external libraries are present, the Pmclib distribution consists in 6 libraries allowing you to build your own adaptative mcmc, importance or pmc sampler, and 7 command-line tools that use the plugin system to perform simple mcmc or pmc runs on user defined likelihoods.

2.1 Libraries

The following three libraries are always available. The first two are support libraries, the last one is the core of the pmclib

- `errorio` contains support function for reporting error tracebacks, allong with error reporting aware memory allocation and basic io operations.
- `mvdens` contains basic operations related to mixture of multivariate gaussian and student-t distributions. It is a very low-level library and is not supposed to be called directly. It will not be descrived in the document.
- `pmc` is the core pmclib library. It contains the definition of the `distribution` structure, high-level interface for the mixture of multivariate gaussian and student-t distribution, objects defining an importance, pmc and mcmc run along with their interface.

Depending of the availlability of an MPI external library, two extra libraries are built

- `pmc_mpi` is a parallel version of the pmc library. it provides parallel importance and pmc routines.
- `gridMe` provides functions to computes the values of a `distribution` object on a multi-dimentional grid in parallel. Load-balancing on the different cpus is computed automatically.

Depending on the availability of lua one extra library, and extra functions in the other library are available

- `readConf` is a wrapper around lua to simplify reading the content of a lua parameter file.

Finally, if hdf5 is available, hdf io function are included in the `pmc` and `gridMe` libraries.

2.2 Command-line tools (see `exectools`)

The command-line tools are built only if lua and hdf5 are available. They all allow the exploration of a distribution.

- **compute_lkl** compute the value of a distribution in one or more location of the parameter space, print the results, and optionally compares it with expected results.
- **simple_grid** compute the values of a distribution on a grid in the parameter space.

- **simple_amcmc** performs a (possibly adaptive) Metropolis-Hasting Markov Chain Monte Carlo exploration of a distribution.
- **simple_importance** performs an importance sample of a target distribution, from a proposal distribution
- **simple_importance_mpi** performs the same task in parallel.
- **simple_pmc** performs an adaptive importance sample of a target distribution, from a mixture of multivariate gaussian and student-t proposal.
- **simple_pmc_mpi** performs the same task in parallel.

2.3 Python module

The python module `pmctool` provides tools to read results from the pmc, importance, mcmc and grid codes saved in the hdf5 format. It also provides a few utility to compute means, variance, covariance, nd-histograms... from the pmc, importance and mcmc results. Finally it also contains a routine to perform *triangle plots* that stack 1d and 2d histograms describing the sample.

2.4 Example parameter files and codes

A sample plugin `distribution` is provided in the `sampleTarget` library. This distribution is the simple bent gaussian distrubtion used as a toy model in Wraith et al. and Kilbinger et al.. Example parameter files for the different command-line tools, allowing to explore the bent gaussian distribution defined in `sampleTarget` are also provided. Finally a sample code demonstrating some of the functionnalities of the `readConf` library along with a test parameter file is also available. All of this files are installed in `PREFIX/share/pmclib/example`.

CHAPTER

THREE

HANDLING ERRORS WITH THE ERRORIO LIBRARY

3.1 An overview

Pmclib uses a simple error reporting system which allows for the reporting of the exact location of where an error occured in the code. Almost all functions in pmclib have an extra last argument which contains a double pointer on an `error` type. When the function returns, a macro can be used to test whether the pointer contains an actual error. Helpers macro are also available to ignore, or propagate the error to the calling code.

This is best demonstrated with and example. Assume we have a function `fA` which will compute some result using from its input parameters, using results from a function `fB`. Both function can fail, and report errors. Then the function `fB` and `fA`, along with a `main` that call them will look like this

```
#include <stdio.h>
#include <math.h>
#include "pmclib/tools/errorio.h"

double fB(double someArg, error **err) {

    // assume that we cannot compute when someArg<0

    testErrorRetVA(someArg<0,                // some test
                   -1,                       // an error code
                   "Cannot perfom computation when argument is negative, got %g, I would rather have
                   *err,                     // the pointer one the error object to be updated
                   __LINE__,                 // the line at which the error occured
                   -1,                       // the return value
                   someArg, fabs(someArg)); // the variables to include in the string

    // if we are here, it means that no error was reported
    return sqrt(someArg);
}

double fA(double someArg, error **err) {
    double intermediateRes;

    intermediateRes = fB(someArg,err); //note that we are passing the pointer on the pointer on the e
    forwardError(*err,      // test if *err points on an error. If it is the case, propagate it
                 __LINE__,  // note that propagation occured in the current function at this line
                 -1);       // and return -1

    // if we are here, no error was reported by fB
```

```
30
31      return intermediateRes * someArg;
32  }
33
34  int main(int argc,char** argv) {
35      error *_err;
36      error **err;
37      double res;
38      double val;
39
40      // init error to NULL
41      _err = initError()
42      err = &_err;
43
44      // check that there is exactly one argument on the command line
45      testErrorExitVA(argc!=2,-1,"Expected one argument on the command line, got %d",
46                      *err,__LINE__,argc);
47
48      val = atof(argv[1]);
49      res = fA(val,err);
50      quitOnError(*err,__LINE__,stderr);
51
52      printf("result is %g\n",res);
53
54      endError(err);
55
56      return 0;
57  }
```

And this piece of code will produce the following results:

```
$> a.out
main(testerror.c:46)::Error -1 (Expected one argument on the command line, got 1)

$>a.out -10
main(testerror.c:50)::ForwardError
  fA(testerror.c:27)::ForwardError
    fB(testerror.c:15)::Error -1 (Cannot perfom computation when argument is negative, got -10, I wou

$>a.out 10
result is 31.6228
```

In the first case, an error is reported at line 46 in the `main` function. in the second case, an error occurs in a call to `fB` caused by a call to `fA` which was called by the `main`. We have the full traceback, and can go back to the origin of the error. This is very usefull when, for instance, a util function is called by many different function. Knowing only that this function has failed is not enough, one must know why it was called and by which other function.

3.2 Using `errorio`

3.2.1 Compilation

Code using `errorio`, must include `pmctools/errorio.h` and link with `-lerrorio`.

3.2.2 Initialization and termination of the errorio session

The main program must initialize and terminate the errorio session. This is done by initializing an empty error variable using initError(). A pointer on the initError() return value will then be passed all functions that can report errors. At the end of the main program, the errorio session must be closed using endError().

error
A container for errors

error * **initError** (void)
Initialize the error container. Returns a pointer on an initialized error. Should be called just once by the main program.

void **initError** (error **err)
Terminates the session. On return, *err = NULL

3.2.3 Reporting an error

Any function can report an error using the TestErrorRet() or TestError() macro.

testError (test, error_type, message, err, line)
if test is true, create an error with value error_type and message message, store it in *err, note that the line at which the error occured is line. err must be of type error* This is typically used this way

```
testError(1==1,-1,"wow 1==1 !!!", *err,__LINE__);
```

testErrorVA (test, error_type, message, err, line, ...)
same as testError but the message can contain printf format command that will be filled using the arguments after line

testErrorRet (test, error_type, message, err, line, returnvalue)
same as testError but returns with value returnvalue if test is true. returnvalue can be left empty to return with no value.

testErrorRetVA (test, error_type, message, err, line, returnvalue, ...)
same as testErrorVA but returns with value returnvalue if test is true. returnvalue can be left empty to return with no value.

testErrorExit (test, error_type, message, err, line)
same as testError but exits value errortype if test is true.

testErrorExitVA (test, error_type, message, err, line, ...)
same as testErrorVA but exits value errortype if test is true.

3.2.4 Handling and discarding an error

The code of any error can be obtained using the function getErrorValue(). Its message can be retrieved using stringError(). Finally, an error can be discarded using purgeError().

int **getErrorValue** (error* *err*)
returns the error code of an err. Returns NoErr if err is not an error.

void **purgeError** (error **err)
purge an error.

void **stringError** (char* *str*, error *err)
on return, str contains the message of the error err. str must be big enough to contain the text.

3.2.5 Testing for error and forwarding errors

The macro `isError` allow for testing whether an error was reported. A function can forward an error that occured after a call to another function using the `forwardError` macro. Finally, the main code can report an error and return using `quitOnError`.

isError (err)
: returns 1 if `err` contains an error. `err` must be of type `error*`.

forwardErrorNoReturn (err, line)
: if `err` contains an error, forward it noting that the forward happened at line `line`. `err` must be of type `error*`.

forwardError (err, line, retvalue)
: same as `forwardErrorNoReturn` but if `err` contains an error, also return with return value `retvalue`. `retvalue` can be left empty to return with no value.

quitOnError (err, line, F)
: if `err` contains an error, exit with value the error code, print the message on opened file `F`, note that the exit occured at line `line`. `F` is usually `stderr`.

3.2.6 IO and memory allocation facilities

The library contains a few helper error reporting functions for IO and memory management.

void* **malloc_err** (size_t *sz*, error ***err*)
: Allocate sz bytes. Report error on failure.

void* **calloc_err** (size_t *nl*, size_t *sz1*, error ***err*)
: Allocate nl objects of size sz1. Set the buffer to 0, report error on failure.

void* **resize_err** (void* *orig*, size_t *sz_orig*, size_t *sz_new*, int *free_orig*, error ***err*)
: Return a newly allocated buffer of size sz_new. Copy sz bytes of buffer at address orig to the new buffer. If free_orig==1, memory at orig is freed after copy. If sz>new_sz, only new_sz bytes will be copied.

FILE* **fopen_err** (const char* *fname*, const char **mode*, error ***err*)
: Open file fname with mode mode. Report error on failure

double* **read_double_vector** (const char* *fnm*, size_t *n*, error ***err*)
: Read exactly n double floating point numbers from an ascii file fnm. The file can contain comments prepended by #, ; od !. Returns a newly allocated buffer on the vector list. Report error on failure, and if less than n number can be read.

float* **read_float_vector** (const char* *fnm*, size_t *n*, error ***err*)
: Same as `read_double_vector()` for floats.

int* **read_int_vector** (const char* *fnm*, size_t *n*, error ***err*)
: Same as `read_double_vector()` for ints.

long* **read_long_vector** (const char* *fnm*, size_t *n*, error ***err*)
: Same as `read_double_vector()` for longs.

double* **read_double_list** (const char* *fnm*, size_t **n*, error ***err*)
: Same as `read_double_vector()` but for files whose size is not known in advance. The number of read number is returned in *n.

float* **read_float_list** (const char* *fnm*, size_t **n*, error ***err*)
: Same as `read_double_list()` for floats.

int* **read_int_list** (const char* *fnm*, size_t **n*, error ***err*)
: Same as `read_double_list()` for ints.

long* **read_long_list** (const char* *fnm*, size_t **n*, error ***err*)
Same as `read_double_list()` for longs.

CHAPTER

FOUR

READING LUA PARAMETER FILES WITH READCONF LIBRARY

Pmclib uses lua as its language for writting parameter files. This allows for writting and parsing complex parameter files, including loops or simple computations. To simplify the parsing of the lua file, the readConf library provide a simpler api than the lua api, and is used in pmclib.

4.1 Syntax of the parameter files

The reader is refered to the LUA book to learn about lua language specifics. We just recall here a very few specificities (pitfalls) of the language.

Comments are prepended by `--`. String concatenantion operator is `...` Arrays are indexed using square brakets `[]` and their first element is numbered 1. Array can mix whatever types. Hash tables and structures (in c parliance) are the same type; i.e. to access the element `x` of a structure or hash-table `tbl` one can either use:

```
tbl.x
```

or:

```
tbl["x"]
```

Moreover, an array is in fact a special kind of hash-table whose members are unammed, and elements from a hash-table can also be accessed through numbering (in the order in which they have been initialized). All those types are just one sype in lua and is called table. There is no unmutable table, which means that table can grow. Nested tables are of course possible in lua and are accessible using readConf. Finally, the empty table is `{}`.

4.2 Using `readConf`

4.2.1 Compilation

Code using `readConf`, must include `pmctools/readConf.h` and link with `-lreadConf -llua`.

4.2.2 Initialization and termination of the `readConf` session

The program can read as many parameter file as necessary. Each parameter file is handled thru a `confFile` variable. There are two ways of opening a new parameter file, either using `rc_open()` which simply open a file, or with

`rc_init_from_args()` which will use the command line to determine the name of the parameter file and possibly modify some of the parameters. At the end of the session, the parameter file has to be closed using `rc_close()`.

Since lua can have nested table, a parameter file can be seen as a tree of parameters. To more easily play with this tree, a `confFile` can either refer to a parameter file, or to a section of this tree of parameter, rebased at any point in the tree structure. This is done using `rc_alias()`. Such a virtual parameter file also has to be closed with `rc_close()`.

confFile
A container for a parameter file

confFile* **rc_open** (char* *fnm*, error ****err*)
Open a parameter file from file `fnm`.

confFile* **rc_init_from_args** (int *argc*, char** *argv*, error ****err*)
Look for the parameter file name as the last argument in the command line. Each time `-e SOMELUACOMMAND` or `--extra SOMELUACOMMAND` is found in the command line, the string SOMELUACOMMAND will be executed by lua AFTER reading of the parameter file. This allow for command line modification of the parameters. Finally if the parameter `rc.verbose` or `verbose` is found in the parameter file, its value will set the verbosity level of `readConf`.

void **rc_close** (confFile ****self*)
Close the parameter file. `*self` is set to NULL on return.

confFile* **rc_alias** (confFile **rc*, char* *prefix*, error ****err*)
Open a virtual parameter file from the opened parameter file `rc` whose root level is at `prefix`. After usage the virtual parameter file must be closed using `rc_close()`. The following code

```
rca = rc_alias(rc,"object1.object2",err);
quitOnError(*err,__LINE__,stderr);

result = rc_get_real(rca,"value",err);
quitOnError(*err,__LINE__,stderr);
```

and the following one

```
result = rc_get_real(rc,"object1.object2.value",err);
quitOnError(*err,__LINE__,stderr);
```

gives the same result.

4.2.3 Reading parameters

int **rc_has_key** (confFile* *rc*, char* *key*, error ****err*)
rturns `1` if the parameter file contains the path `key`.

double **rc_get_real** (confFile **rc*, char **key*, error ****err*)
reads from parameter file `rc` the parameter at path `key`. Try to convert it as a `double`.

long **rc_get_integer** (confFile **rc*, char **key*, error ****err*)
reads from parameter file `rc` the parameter at path `key`. Try to convert it as an `long`.

char* **rc_get_string** (confFile **rc*, char **key*, error ****err*)
reads from parameter file `rc` the parameter at path `key`. Try to convert it as an array of `char`. The array is automatically allocated to the correct size. It will be deallocated when the parameter file will be closed with `rc_close()`. This means that after this call, the return pointer from the function will point on a meaningless memory zone.

double **rc_safeget_real** (confFile **rc*, char **key*, double *safeguard*, error ****err*)
reads from parameter file `rc` the parameter at path `key`. Try to convert it as a `double`. If the parameter file does not contain a variable at path `key` or if it cannot be converted in `double`, return `safegard`.

long **rc_safeget_integer** (confFile *rc, char *key, long *safeguard*, error **err)
reads from parameter file `rc` the parameter at path `key`. Try to convert it as an `long`. If the parameter file does not contain a variable at path `key` or if it cannot be converted in `int`, return `safegard`.

char* **rc_safeget_string** (confFile *rc, char *key, char* *safeguard*, error **err)
reads from parameter file `rc` the parameter at path `key`. Try to convert it as an array of `char`. The array is automatically allocated to the correct size. It will be deallocated when the parameter file will be closed with `rc_close()`. This means that after this call, the return pointer from the function will point on a meaningless memory zone. If the parameter file does not contain a variable at path `key` or if it cannot be converted in an array of `char`, return `safegard`. In this case, no allocation will be performed, the return value will simply be the same pointer as the one contained by `safegard`.

int **rc_is_array** (confFile *rc, char* *key*, error **err)
returns `1` if the parameter file contains a table at path `key`. If the parameter file contains no varaible at this path, reports an error.

size_t **rc_array_size** (confFile *rc, char* *key*, error **err)
returns the size of the array at path `key`.

size_t **rc_get_real_array** (confFile *rc, char* *key*, double** *parray*, error **err)
returns the size of the array at path `key`. The content of the array will be copied in a memory zone whose pointer will be passed in `*parray`. Each element of the array will be converted in `double`. The array is automatically allocated to the correct size. It will be deallocated when the parameter file will be closed with `rc_close()`. This means that after this call, the return pointer from the function will point on a meaningless memory zone.

size_t **rc_get_integer_array** (confFile *rc, char* *key*, long** *parray*, error **err)
returns the size of the array at path `key`. The content of the array will be copied in a memory zone whose pointer will be passed in `*parray`. Each element of the array will be converted in `long`. The array is automatically allocated to the correct size. It will be deallocated when the parameter file will be closed with `rc_close()`. This means that after this call, the return pointer from the function will point on a meaningless memory zone.

size_t **rc_get_string_array** (confFile *rc, char* *key*, char*** *parray*, error **err)
returns the size of the array at path `key`. The content of the array will be copied in a memory zone whose pointer will be passed in `*parray`. Each element of the array will point on an array of `char`. The array is automatically allocated to the correct size. It will be deallocated when the parameter file will be closed with `rc_close()`. This means that after this call, the return pointer from the function will point on a meaningless memory zone.

size_t **rc_safeget_real_array** (confFile *rc, char* *key*, double** *parray*, error **err)
same as `rc_get_real_array()` but if the array at `key` is absent, return 0 and does not modify `*parray`.

size_t **rc_safeget_integer_array** (confFile *rc, char* *key*, long** *parray*, error **err)
same as `rc_get_real_array()` but if the array at `key` is absent, return 0 and does not modify `*parray`.

size_t **rc_safeget_string_array** (confFile *rc, char* *key*, char*** *parray*, error **err)
same as `rc_get_real_array()` but if the array at `key` is absent, return 0 and does not modify `*parray`.

CHAPTER

FIVE

THE PMCLIB LIBRARY

5.1 An overview

The pmclib library contains all the facilities needed to perform mcmc, importance sampling and adaptive importance sampling exploration of probabilities. It defines a few C structures that represent both the distributions and the different MC runs. The basic one is distribution which describe a probability distribution in an n-dimensional space. Most of this documentation will describe how to build fill such a structure with both data, and user defined functions, and how to manipulate it to build products of probability distributions. In this version, the library for building once own MC codes will not be documented, which means that both *lua* <*http://www.lua.org/*> and *hdf5* <*http://www.hdfgroup.org/*> will be mandatory. *CosmoPMC* <*cosmopmc.info/*> is an example of codes which use the undocumented low level functionnality of the library to build specialized MC codes.

5.2 The distribution structure

This structure contains all the information needed to define a probability distribution and links to functions allowing to compute log probability of any point of the space. Optionally, it can define names for the direction of the n-dimensional space and can provide functions for the estimation of first and second derivatives of the distribution. Please note that the log probability estimates can be offset by an arbitrary constant; i.e. the distribution is not necessary normalized. Indeed, performing MC exploration is often the only way to normalize a distribution!

5.2.1 Initialization

distribution
: defines a distribution. The content of the structure will not be described and can change in the future.

distribution* **init_simple_distribution**(int *ndim*, void* *privatedata*, log_pdf_func* *log_pdf*, free_func* *freef*, error **err*)
: This is the simplest way to initialize a distribution function. It provides the minimum amount of information, which is: ndim, the number of dimension of the space, privatedata a pointer on a chunck of memory that can possibly contains the data needed to perform log probability estimates, log_pdf a pointer on a function of type log_pdf_func (this is the function that will actually computes the log probability, using the data pointed by privatedata) and finally, freef a pointer on a c:type:*free_func* fuction that can release the memory pointed by privatedata and that will be called at the deallocation of the distribution structure, i.e. when calling free_distribution(). freef can be set to NULL if no such function is needed.

double **log_pdf_func**(void * *privatedata*, const double **arg*, error **err*)
: a function implementing this type can be used to compute the log probability of a point in the n-dimensional space. A function of this type must be passed at the initialization of a distribution structure to be called when computing log probability estimate. This function accept 3 arguments, a pointer to a chunck of memory

that can contain data needed by the function to perform its calculation, a pointer on an array of `double` values that represent the point at which the probability have to be estimated, and an error. The function must return a `double` representing an estimate of the log probability, or return an error in case of problem. he estimate can be off by a constant amount : the distribution does not have to be normalized. Here is a simple example for a 1D gaussian distribution, we assume that an array of two `double` values have been initialized in memory, containing the mean and standard devaition of the model.

```
double gauss_1d_logpdf(void *privatedata, const double *arg, error **err) {
    double res;
    double *ms, mean, std, x;

    // retrieve info from the chunck of memory pointed by privatedata
    ms = (double*) privatedata;
    mean = ms[0];
    std = ms[1];

    // get the location at which to estimate the log proba
    x = arg[0];
    x -= mean;
    x/= std;

    // return un-normallized log probability
    res = - 0.5 * x * x;
    return res;
}
```

The function is allowed to modify the memory pointed at `privatedata`

void **`free_func`**(void ** *pdata*)
a function implementing this type must free the memory at `*pdata`, along with all the memory it possibly refers to. On return `*pdata` must be set to `NULL`.

void **`distribution_set_names`**(distribution **dist*, char** *name*, error ****err*)
allows to name the directions of the n-dimensional space where the distribution operates. `dist` is an already initialized distribution, `names` an array of pointers on arrays of `\0` terminated `char`. There must be at least `ndim` element in this array, where `ndim` is the number of dimension given when initializing the distribution with `init_simple_distribution()`. This is usefull when creating `distribution` that are product of distributions operating on different dimensions.

void **`distribution_set_derivative_func`**(distribution **dist*, first_derivative_func* *f_der*, second_derivative_func* *d_der*, error ****err*)
allows to set the functions for computing the first and second derivatives of a `distribution` `dist`. `f_der` and `d_der` can be set to `NULL` if they do not exists. Those function will be used by `distribution_first_derivative()`, `distribution_second_derivative()`, `distribution_second_derivative_matrix()` and `distribution_deriv_along()`.

double **`first_derivative_func`**(void* *privatedata*, int *idir*, const double* *arg*, error ****err*)
a function that implement this type will compute the derivative of a log probability at location `arg` in the `idir` direction using the data contained at `privatedata`.

double **`second_derivative_func`**(void* *privatedata*, int *idir*, int *jdir*, const double* *arg*, error ****err*)
a function that implement this type will compute the derivative of a log probability at location `arg` in the `idir` and `jdir` directions using the data contained at `privatedata`.

5.2.2 Using `distribution`

double **`distribution_log_pdf`**(distribution* *dist*, const double* *arg*, error ****err*)
computes the log probability for the c:type:*distribution* `dist` at the location of the space given by the array `arg`.

Internally, `arg will be passed, to the function of type :c:type:`log_pdf_func` along the the pointer ``privatedata` defined when initializing `dist`.

void **free_distribution** (distribution **pdist*)
free `*pdist`, also free the memory pointed at `privatedata` provided during the initialization, using the function of type `free_func` provided during initialization.

double **distribution_first_derivative** (distribution *dist, int idir, double* arg, error **err)
compute the first derivative of the log probability for the `distribution dist` at the location `arg`, along the ‘‘idir’’th direction. If a function to compute the derivative have been provided using `distribution_set_derivative_func()`, it will be used, otherwise, an iterative numerical method will be used.

double **distribution_second_derivative** (distribution *dist, int idir, int jdir, double* arg, error **err)
compute the second derivative of the log probability for the `distribution dist` at the location `arg`, along the ‘‘idir’’th and ‘‘jdir’’th directions. If a function to compute the derivative have been provided using `distribution_set_derivative_func()`, it will be used, otherwise, an iterative numerical method will be used.

double* **distribution_second_derivative_matrix** (distribution *dist, double *par, error **err)
use `distribution_second_derivative()` to compute all the second derivatives of the log probability for `distribution`. Returns a newly allocated array that will have to be deallocated by the caller.

double **distribution_deriv_along** (distribution *dist, double *arg, double *along, error **err)
comute the derivative of the log probability for the `distribution dist` at location `arg` and along the vector `along`. If a function to compute the derivative have been provided using `distribution_set_derivative_func()`, it will be used, otherwise, an iterative numerical method will be used. In the first case, this function will be called once for each direction `i` for which `along[i]` is not `0` and the scalar product will be computed. In the second case, only one numerical estimate will be produced for the reparametrized log probability.

5.2.3 Combining `distribution`

One can multiply different `distribution` to produce a new one which will be their product. Each distribution does not have to operate on the same space. It is easier to combine distribution when the dimension of their space have been named.

distribution * **combine_distribution_simple_init** (int *ndim*, error **err)
create a new distribution which will be the product of different distributions. The dimension of the space on which the product will operate is `ndim`.

void **add_to_combine_distribution** (distribution *prod, distribution *addon, int *dim_idx, error **err)
add the `distribution addon` to the product of `distribution prod`. `dim_idx` is an array of integer that have the same number of dimension than the space on which `addon` is operating. It gives the mapping between the space for `addon` and the one of `prod`, i.e. `dim_idx[3]` is the direction in the `prod` space which correspond to the fourth (remember we are in C !) one in the `addon` space.

add_to_combine_distribution_name (distribution *prod, distribution *addon, error **err)
same as `add_to_product()` but the `dim_idx` array is automatically computed based on the name of the direction of the different spaces. The name of the directions for `prod` and `addon` must have been set before, using calls to `distribution_set_name()`

distribution ***add_gaussian_prior** (distribution *orig, int ndim, int* dim_idx, double* loc, double *var, error **err)
returns a new product of `distribution` between a `orig` and a gaussian distribution of `ndim` dimensions,

operating on the directions of orig given by the array `ndim_idx`, whose means are given by the array `loc` and diagonal of the covariance matrix `var`.

distribution ***add_gaussian_prior_name_2** (distribution **orig*, int *ndim*, int* *dim_idx*, double* *loc*, double **var*, error ****err*)
returns a new product of `distribution` between a `orig` and a gaussian distribution of `ndim` dimensions, operating on the directions of orig given by the array `dim_idx`, whose means are given by the array `loc` and covariance matrix `var`.

distribution ***add_gaussian_prior_name** (distribution **orig*, int *ndim*, char** *names*, double* *loc*, double **var*, error ****err*)
returns a new product of `distribution` between a `orig` and a gaussian distribution of `ndim` dimensions, operating on the directions of orig given by the array `name`, whose means are given by the array `loc` and diagonal of the covariance matrix `var`.

distribution ***add_gaussian_prior_name_2** (distribution **orig*, int *ndim*, char** *names*, double* *loc*, double **var*, error ****err*)
returns a new product of `distribution` between a `orig` and a gaussian distribution of `ndim` dimensions, operating on the directions of orig given by the array `name`, whose means are given by the array `loc` and covariance matrix `var`.

5.3 Compilation and usage of a library describing a `distribution` for use with the exec tools

5.3.1 Compilation

Code using `pmclib`, must include `pmclib/distribution.h` and `pmclib/pmc_rc.h`, and link with `-lerrorio -lreadConf -lpmc -lgsl -lgsl -lgslcblas -llua -lhdf5`.

5.3.2 Usage

To be usable with the different executable tools provided, the code describing one or more user defined distribution must be compiled as a dynamic library that will be loaded at run time. The library must contain a function of type `rcinit_smthng`, and named `rcinit_NAMEOFTHEDISTRIBUTION`.

void* **`rcinit_smthng`** (confFile* *rc*, char* *root*, error ****err*)
a function implementing this type must use the opened parameter file `rc` and retrieve parameters at path `root` to initialize a `distribution` that will be returned as a `void` pointer.

The directory `example/target` contains a full example of the definition of a `distribution` along with its `rcinit_smthng` function. Here is an example of a `rcinit_smthng` function for the simple example shown above.

```
1  distribution* rcinit_gauss_1d(confFile *rc, char* root, error **err) {
2      confFile *rca;
3      double *leak;
4      distribution *dist;
5
6      rca = rc_alias(rc,root,err);
7      forwardError(*err,__LINE__,NULL);
8
9      leak = malloc(sizeof(double)*2);
10
11     leak[0] = rc_get_real(rca,"mean",err);
12     forwardError(*err,__LINE__,NULL);
```

```
13
14      leak[1] = rc_get_real(rca,"std",err);
15      forwardError(*err,__LINE__,NULL);
16
17      // I am leaking ! I do not provide a free function !
18      dist = init_simple_distribution(1,leak,gauss_1d_logpdf,NULL,err);
19      forwardError(*err,__LINE__,NULL);
20
21      return dist;
22  }
```

Once this is done, a the parameter of a distribution can be given in the parameter files of the different `exectools`. For each of this tools, the `target` parameter must contain a list of the parameter for the `distribution` to be used, along with the path (parameter `libpath`) at which the shared library is installed, and the name of the `distribution`. The `name` parameter will be used to form the name of the `rcinit_smthng` function to be called in the shared library to initialize the `distribution`. The path in the parameter file to the `target` parameter will be passed as the `root` argument of the `rcinit_smthng` function.

Here is an example extract of a parameter file describing a particular instance of ou gauss_1d example (in this case a standard normal).

```
exampletarget = {}
exampletarget.name = "gauss_1d"
exampletarget.mean = 0
exampletarget.std = 1
```

The parameter file can also contain some special extra parameters to specialize the `distribution()`. One can add a parameter `parameter_name` which will contain a list of string that will be used as names for the direction of the space on which the `distribution` operates. This will superseed any such names given in the c functions that initialize the `distribution`. One can also add a `conditional` parameter that can contains two lists (`conditional.id` and `conditional.value`). This allows to transform an n-D probability into an (n-m)-D probability conditioned on the m directions whose index are given by `conditional.id` and at location given by `conditional.value`. Note that `conditional.id` can either contain strings corresponding to the name of the direction which have to be set, or integer giving their rank (starting at 0), not a mix of the two.

Please have a look at `$INSTALLPREFIX/share/pmclib/examples/test_sampleTarget.lua` for an example of how to parametrize the example `distribution`.

5.3.3 Building product of `distribution` from the parameter file

The `combine distribution` that can be initialized programmatically using `combine_distribution_simple_init()` can also be defined completely from the parameter file. The `name` parameter has to be set to *combine_distributions*, the `libpath` is not needed. The only mandatory parameters are `ndim` the number of dimension of the space, and `distributions` a list containing in each elements two parameters `distributions[i].distribution`, the parameters needed to describe each distribution (as explained above) and optionnaly `dim_idx` a list that will contain the equivalent of the argument of the same name in the function `add_to_combine_distribution()`. The `dim_idx` parameter can be omitted if a `parameter_name` parameter list is present at the same level as the `ndim` argument and if the distribution to multiply all have named direction (either programmatically or using `parameter_name` parameters in each dsitribution parameter list).

Please have a look at `$INSTALLPREFIX/share/pmclib/examples/test_sampleTarget.lua` for an example of how to combine two example `distribution`.

5.3.4 Adding gaussian priors to directions of the space of a `distribution` from the parameter file

To do so, one has to set the `name` parameter to *add_gaussian_prior*. A parameter `distribution` nust contain the parameter list for the distribution which will be multiply by the gaussian distribution. Two parameters `mean` and `var` provide the arrays of double for the mean and variance of the gauss function. If `var` is a list of numbers whose size is the same than `mean` it is understood as the diagonal of the matrix, otherwise, the full matrix have to be provided. The direction on which the gaussian prior operates are either given using the parameter `prior_idx` (same as `dim_idx` above and same as argument `idim` of function c:func:*add_gaussian_prior*) or using the `parameter_name` parameter and assuming that the distribution have named direction (programmatically, or using `parameter_name`).

Please have a look at `$INSTALLPREFIX/share/pmclib/examples/test_sampleTarget.lua` for an example of how to add a gaussian priorto an example `distribution`.

CHAPTER

SIX

THE EXECTOOLS

6.1 compute_lkl

`compute_lkl someparfile.lua`

Computes the log probability value of a distribution in one location of the parameter space, print the results, and optionally compares it with expected results.

6.1.1 Parameter file

The parameters for this tool mst be grouped in `testbed`. The distribution whose value value at some point must be computed is defined in the `testbed.target` parameter. The location at which the probability will be evaluated is given by `testbed.pars`, a list of numbers. Optionally, `testbed.result` can contain a number that will be compared to the result of the computation. Here is an example parameter file. Another is available at `$INSTALLPREFIX/share/pmclib/examples/test_sampleTarget.lua`

```
sometarget = {}
sometarget.ndim = 2
sometarget.bent = .001
sometarget.sig1_square = 100
sometarget.name = "bentGauss"
sometarget.libpath = "$INSTALLPREFIX/share/pmclib/examples/libsampleTarget.dylib"

testbed = {}
testbed.pars = {4,1}
testbed.target = sometarget
```

6.2 simple_grid

`mpirun -np XX simple_grid someparfile.lua`

Compute the log probability values of a distribution on a grid in the parameter space. Save the result as an hdf5 file that can be read in python using `grid`. The computation of the values of the probability is scattered on different computers using mpi.

6.2.1 Parameter file

The parameters for this tool mst be grouped in `grid`. The distribution whose value value at some point must be computed is defined in the `grid.target` parameter. A parameterbox must be provided in `grid.pb` and the output

file path in `grid.file`.

There are numerous ways to provide the location at which the probability must be computed.

- The min and max in each direction can be retrieved from the parameterbox by setting `grid.usepb = 1`. The number of bins is given by the parameter `grid.nbin` which can be either a single number (square grid) or a list of number giving the number of bins in each direction. The coordinates in each direction will be located at the middle of the nbin bins between min and max. The repartition and the size of the bins are linear by default but can be made logarithmic is `grid.islog` is set to 1.
- Alternatively, the min and max can be given by setting the `grid.min` and `grid.max` parameters.
- The coordinates of the grid can also be given by filling `grid.pos` with an list of array of number, one for each dimension.
- The coordinates of the grid can also be retrieved from an hdf5 file produced with a previous run of `simple_grid` setting the path to the file in `grid.pos`.

Here is an example parameter file. Another is available at `$INSTALLPREFIX/share/pmclib/examples/test_grid.lua`

```
sometarget = {}
sometarget.ndim = 2
sometarget.bent = .001
sometarget.sig1_square = 100
sometarget.name = "bentGauss"
sometarget.libpath = "$INSTALLPREFIX/share/pmclib/examples/libsampleTarget.dylib"

grid.file = "exploration.grid"

-- init grid
grid = {}

grid.target = bentGauss
grid.islog = 0

grid.nbin = 200 --same number of bins in each direction

grid.usepb = 1 -- use min/max from pb

-- init pb
pb = {}
pb.min = {-10,-10}
pb.max = {10,10}

grid.pb = pb
```

6.3 simple_amcmc

`simple_amcmc someparfile.lua`

Performs a mcmc exploration of a given distribution. Save the result as an hdf5 file that can be read in python using `chain`. The kernel of the mcmc algorithm is parametrizable and can be optionally adapted during the run. For now, only Metropolis Hastings kernel are available.

6.3.1 Parameter file

The parameters for this tool must be grouped in `mcmc`. The distribution whose value value at some point must be computed is defined in the `mcmc.target` parameter. A parameterbox must be provided in `mcmc.pb`. The length of the chain is given by the parameter `mcmc.nsample`. The output file path in `mcmc.file`. The file is updated each time `mcmc.nbatch` points have been simulated. The initial location of the mcmc is given by the parameter `mcmc.pars0` which is a list of number of the same length than the number of dimensions of the distribution.

The kernel is given in by the `mcmc.kernel` parameter. The way to parametrize the kernel is very similar to the parametrization of a distribution. For now, only one kernel is available, a adaptive Metropolis-Hasting one. Its parameters are `ndim` the number of dimensions, `sig0` the covariance matrix of the kernel before adaptation. It can either be a list of `ndim` numbers, which is understood as the diagonal of the matrix, or the full matrix. The `name` parameter is mandatory and must be *adaptMH*. With only this parameters, the run will consist of a regular MH-MCMC. Adaptation is turned on if the parameter `nadapt` which determine the number of simulated points between each adaptation is different from 0. Note that is `adaptifaccepted` is 1, the count will be on accepted points (i.e. effective displacement), otherwise, is will just be on the number of points. At each adaptation, the variance is recomputed from all the previous simulations. A damping of the past simulation can be added to reduce long scale correlations. The parameter `k_damp` allows to turn this damping, at 0 (default value) no damping is performed, at 1 only the last `nadapt` points will contribute. Finally, the estimated variance is enlarged by the factor `c_enlarge` square rescaled by the number of dimensions. If not set in the parameter file, the parameter is set to `2.38`.

Here is an example parameter file. Another is available at `$INSTALLPREFIX/share/pmclib/examples/test_grid.lua`

```
sometarget = {}
sometarget.ndim = 2
sometarget.bent = .001
sometarget.sig1_square = 100
sometarget.name = "bentGauss"
sometarget.libpath = "$INSTALLPREFIX/share/pmclib/examples/libsampleTarget.dylib"

-- parabox
pb = {}
pb.min = {-20,-20}
pb.max = {20,20}

-- adaptMH proposal
mykernel = {}
mykernel.ndim = 2
mykernel.sig0 = {10,100}
mykernel.nadapt = 1000
mykernel.adaptifaccepted = 1
mykernel.name = "adaptMH"

-- mcmc run
mcmc = {}
mcmc.nsample = 30000*4
mcmc.target = sometarget
mcmc.kernel = mykernel
mcmc.pb = pb
mcmc.nbatch = 1000
mcmc.file = "exploration.mcmc"
mcmc.pars0 = {4,-1}
```

6.4 simple_importance & simple_importance_mpi

```
simple_importance someparfile.lua
```

```
mpirun -np XX simple_importance_mpi someparfile.lua
```

Performs an importance sampling run on a given distribution. Save the result as an hdf5 file that can be read in python using `chain`. The proposal distribution is parametrizable. For now, only mixture of gaussian distribution are available.

6.4.1 Parameter file - single processor version

The parameters for this tool must be grouped in `importance`. The distribution whose value value at some point must be computed is defined in the `importance.target` parameter. A parameterbox must be provided in `importance.pb`. The length of the chain is given by the parameter `importance.nsample`. The output file path in `importance.file`. A message giving the completion of the importance run is print on the screen each `importance.print_pc` percent of the `importance.nsample` simulations.

The proposal density is given in the `importance.proposal` parameter. The way to parametrize the proposal is very similar to the parametrization of a distribution. For now, only one proposal distribution is available, a mixture of gaussian densities. Its only mandatory parameter is `name` which must be set to *mix_mvdens*. There are many different ways to provide the information about the mean and variance of each component of the proposal density. One can either:

- read it from an hdf5 file using the `fromfile` parameter. Optionally the *extract_component* parameter can contain a list of numbers giving the indices of the components that will be kept to build the proposal. The weights are rescaled so that they sum to one.
- set the number of dimension with `ndim` the number of components with `ncomp` and optionally the weights by setting them as a list to the `weight` parameter. If `weight` is absent, all component will have the same contribution to the mixture. The weights are of course rescaled so that they sum to 1. Finally the parameters of the components can be set by filling the parameter `component` with a list of `ncomp` lua objects that have a `mean` and a `var` parameters, the former being a list of `ndim` numbers and the former being either a list of `ndim` elements which will be understood as the diagonal of the matrix, or a full matrix.
- set the number of dimension, components and the weights in the same way as above, but fill the `draw_from` parameter to a lua object that have a `mean` and a `var` parameters (defined as above). In this case, the proposal will be built from this first component and `ncomp`-1 other whose variance is identical to the first one, but whose location are random numbers sampled from the first component.

In any case, it is good practice to fill the `save` parameter to a path where the proposal will be saved as an hdf5 file.

Here is an example parameter file. Another is available at `$INSTALLPREFIX/share/pmclib/examples/test_importance.`

```
sometarget = {}
sometarget.ndim = 2
sometarget.bent = .001
sometarget.sig1_square = 100
sometarget.name = "bentGauss"
sometarget.libpath = "$INSTALLPREFIX/share/pmclib/examples/libsampleTarget.dylib"

pb = {}
pb.min = {-10,-10}
pb.max = {10,10}

-- mix_mvdens proposal
myproposal = {}
myproposal.ndim = 2
myproposal.ncomp = 2
myproposal.component = {
    {mean={0,0}, var={10,10}},
    {mean={2,2}, var={5,5}},
```

```
}
myproposal.name = "mix_mvdens"
myproposal.save = "proposal.mix_mvdens"

-- importance run
importance = {}
importance.nsample = 30000
importance.target = sometarget
importance.proposal = myproposal
importance.pb = pb
importance.print_pc = 20

pmcFile = "exploration.pmc"
```

6.4.2 Parameter file - mpi version

The parameter file is identical but for one difference: the proposal name must be *mix_mvdens_mpi*.

```
simple_pmc someparfile.lua
```

```
mpirun -np XX simple_pmc_mpi someparfile.lua
```

Performs an adaptive importance sampling run using the pmc algorithm on a given distribution. Save the result as an hdf5 file that can be read in python using `chain`. The proposal distribution is parametrizable. For now, only mixture of gaussian distribution are available.

> The parameter file is identical to the one of resp. simple_importance and simple_importance_mpi, except that the base parameter is `pmc` instead of `importance`, and with the addition ofa few extra parameters. The result of each iteration will be saved using the path at ``pmc.resfile` and the adapted proposal after the iteration at `pmcfile.propfile`. In this two parameters, the string `%d` it will be replaced by the iteration count. The number of iteration is given by the parameter `niter`. Finally, the perplexity at each iteration will be saved in the hdf5 file `pmc.perpfile`. There is no `pmc.file` parameter.
>
> As for simple_importance and simple_importance_mpi, the name of the proposal must be, resp., *mix_mvdens* and *mix_mvdens_mpi*.

An example parameter file is available at `$INSTALLPREFIX/share/pmclib/examples/test_pmc.lua`.

6.5 parameter boxes

In most of the above parameter file, flat priors must have to be given to the exploration tools as parameterbox. Those are all parametrized in the same way. Only two parameters are needed : `min` and `max` which are the list of the minimum and maximum values in each direction. Here is an example:

```
pb = {}
pb.min = {-10,-10,40,-2}
pb.max = {10,10,500,0}
```

CHAPTER

SEVEN

THE PYTHON MODULE `pmctool`

This module contains two class that allow to read the results of the `exectools` in python and to manipulate and plot them. It contains two class `grid` (to read the results of simple_grid) and `chain` (for simple_amcmc, simple_importance and simple_pmc and their mpi versions).

class `pmctool.`**`grid`**(*fromfile*)

Read the result of a simple_grid run from an the hdf5 file *fromfile*. Set `pos`, `log_lkl`, `lkl`.

`pos`

The corrdinates of the grid. Stored as a list of *n* numpy arrays.

`log_lkl`

The log probability. Stored as an ‘n‘D numpy array.

`lkl`

The unnormalized probability. The maximum value of the probability on the grid is rescaled to 1. Stored as an ‘n‘D numpy array.

`pretty_contour`(*select=(0, 1)*, *loc=[ ]*, *levels=(68.26, 95.4, 99.7)*, *cmap = plt.cm.Reds*, *linecolor = "k"*, *alpha=1*)

Plot a 2D contour of a slice of the grid. It obviously only works for dim>2 grids. The *select[0]* versus *select[1]* dimensions. The other coordinates of the grid are set to *loc* which is a list of n-d elements. The countours will show the *levels* percent of the mass in the grid. the colors of the contours are given by *linecolor*. If *cmap* is set to a matplotlib colormap, also plots the grid values with this colormap and alpha transparency *alpha*. If *cmap* is None, do not plot the values.

class `pmctool.`**`chain`**(*file=None*, *pars=None*, *w=None*, *log_lkl=None*)

Read the result of a simple_amcmc, simple_importance, simple_pmc run from an the hdf5 file *file*. Or, if file is *None*, create a chain with parameters *pars*, weights *w* and log probability *log_lkl*. *w* and *log_lkl* can be set to *None*.

`pars`

The locations in space sampled by the mcmc or the importance run. Stored as a nsample x ndim numpy array.

`log_lkl`

The values of the log probability at those points. Can be left empty

`w`

The values of the normalized weights of an importance run. array of 1/nsample for a mcmc chain.

`shape`

returns (nsample,ndim)

`repars`(*lc*)

if *lc* is a function. Returns lc(`pars`)

integrate(*func*, *lc=None)*)
returns sum *func‘(‘lc‘(:py:attr:‘pars*))*:py:attr:*w* which is a montecarlo estimate of the integral of *func(lc(X))* under the distribution.

mean(*lc=None*)
return the mean. If *lc* is not None, first reparametrize using lc.

moment(*order*)
return the montecarlo estimate of the integral of (X-mean)^order under the distribution

correlation(*lc=None*)
return the correlation matrix, Reparametrize with *lc*

resample(*n*)
return resample n points from the chain. Take the weights into account ! Return the result as a chain.

K(prop=None,psi=None):
return the Kullback divergence between the sample and the proposal. This does not make sense for a mcmc chain, obviously ! If *psi* is a function that returns the log probability of a distribution, it will be used to compute the kullbakc diverence between this distribution and the one of the chain. if *prop* is not None, it will be used to recompute the probability of each sample according to the proposal density.

per(*prop=None*, *psi=None*)
same as above, but with the perplexity

ESS()
returns the effective sample size. Only have sense for importance runs.

plot_triangle(*select=None*, *scales=()*, *legend=()*, *levels=(68.26*, *95.4*, *99.7)*, *show_prop=True*, *fill=68.26*, *show_mean=True*, *show_peak=True*, *show_extra=None*, *add_legend=r”$=%(peak).3g^{+%(up).3g}_{-%(down).3g}$”*, *aspect=(8*, *8)*, *fig=None*, *tick_at_peak=False*, *show_peak_2d=True*)

plot a triangle containing on the diagonal the histograms of the 1-D marginals, and in the low part, the 2-D marginals. If *select* is not None, only plots the directions given by the list in *select*. *scales* can contain a list of scales to be applied on the parameters, *legend* a list of legends for each colum on the table; *add_legend* will be added to this legend text with the *peak up* and *down* keywords replaced by the resp. peak, -delta and +delta of the 1-D marginals. Those are computed as the highest bin of the 1-D histogram for each direction, and the location such the numerical integral of the histogram between *peak - down* and *peak + up* contains *fill* percent of the mass of the 1-d marginal. The 2-D contour plots for the marginal will have their levels at *levels* percent of the 2-D histograms. If *show_prop* is True, for importance run results, the proposal will also be shown as a lighter green lines on the 1 and 2-D plots.

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX

Symbols

C

D

E

F

G

I