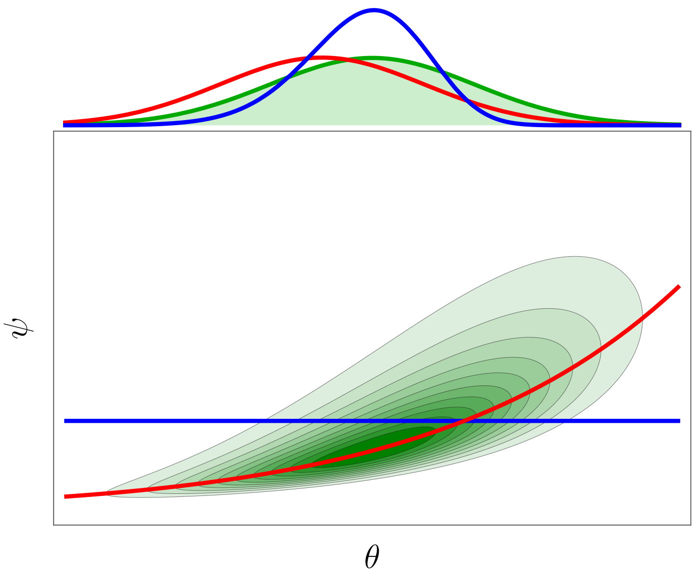

CMP++ Technical Documentation

Version 1.0

Omar Kahol

March 28, 2024

Contents

List of Code Listings

1 Introduction

This document is the technical documentation for the library CMP++[1] which can be used for the Bayesian calibration of computer models using various approches such as

1. Full Bayes.
2. Adaptive approaches (like CMP and FMP) .
3. Sequential approaches (like KOH).

The library is written entirely in C++ and currently provides no high level configuration file hence the pre-processing should be done using a C++ application. The post-processing can be performed using any library or software capable of handling csv input files. The test case in this documentation relies on python and seaborn.
The library can be compiled as a static library or dynamic library and depends on the following libraries:

1. Eigen 3.4.0[2] for linear algebra.
2. NLopt 2.7.1[3] for gradient-free and gradient-based optimization.
3. spdlog[4] for input/output.
4. Doxygen[5] to generate the documentation.

Though not strictly necessary, I also suggest the use of the following libraries:

1. cubature[6] for the computation of multidimensional integrals.
2. finte-diff[7] for finite difference approximations of gradients and hessians.
3. openmp[8] for code parallelization.

These libraries have proven themselves to be excellent tools to perform routine operations. For example, finte-diff[9] can be used to compute the finite difference approximation of the hessian matrix (used by the CMP method) in case a closed form formula is not available.

[1] `https://github.com/omarkahol/cmp.git`
[2] `https://eigen.tuxfamily.org/index.php?title=Main_Page`
[3] `https://nlopt.readthedocs.io/en/latest/`
[4] `https://github.com/gabime/spdlog`
[5] `https://www.doxygen.nl`
[6] `https://github.com/stevengj/cubature`
[7] `https://github.com/zfergus/finite-diff.git`
[8] `https://www.openmp.org`
[9] `https://github.com/zfergus/finite-diff.git`

2 Installation

This section is dedicated to the installation of the CMP++ library and its dependencies. Expert users can skip this section, install the dependencies themselves, compile CMP++ using the provided Makefile and link it to their target application either as a static or dynamic library. The following workflow is, however, strongly recommended.

First and foremost, make sure that you have a C++ compiler and cmake installed in your system. The latter can be checked by calling the command *make* anywhere in your system. You should see the following message

```
make: *** No targets specified and no makefile found.  Stop.
```

I **strongly** suggest to keep all your libraries in a well known location; I keep all the libraries that I use in HOME/opt. If you choose a different location make sure you known how to reference it.

2.1 Installing Eigen

Eigen is a header only library so compilation is not required. Simply download the library from the official website and copy it in your HOME/opt folder. To use it, you will just have to type, in your c++ application,

```
#include <Eigen/Dense>
```

2.2 Installing NLopt

NLopt is compiled and linked as a dynamic library. This means that we will compile it as .dylib (in MacOs) file and link it dynamically to our application. Make sure you are familiar with this concept before proceeding.

To install NLopt in HOME/opt copy the downloaded file in that folder and make sure that the name is something like nlopt-2.7.1 (otherwise simply rename the folder), access it via the terminal and type the following commands.

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=$HOME/opt/nlopt-2.7.1 ..
make
make install
```

This will compile the library into HOME/opt/nlopt-2.7.1 but you can change that by substituting HOME/opt in line 3 with whatever folder you desire. If the installation was successful, you should see that, in the nlopt folder you have an include folder which contains some **header files** and a lib folder which should contain (in MacOs) a **libnlopt.dylib** file. These two folders and files are the only one required so the rest can be safely removed.

Now that we have compiled the library another operation must be performed. It is indeed necessary to append the lib folder path to the system path so that the executable knows where to look for the library during runtime. In MacOs, it is sufficient to add the following line to the .zshrc file in the home folder and restart the terminal

```
export DYLD_LIBRARY_PATH=${DYLD_LIBRARY_PATH}:$HOME/opt/nlopt-2.7.1/lib
```

If this is not properly done, your code will compile and link properly but you will see an *unresolved external symbol* error message at runtime.

2.3 Installing spdlog

Just like Eigen, spdlog can be used as header only library so you just need to copy the downloaded folder into your opt folder and make sure you have the include folder aswell.

2.4 Installing CMP++

In this subsection, I will show how to compile CMP++ as a static library. Currently, i have not added the possibility to compile it as a dynamic library but, if needed, you can add this option in the make file. To do so, you need to copy the downloaded folder into your opt folder and modify the makefile. The makefile instructs your system on how to compile CMP++

```
# define the Cpp compiler to use
CXX = g++

# define any compile-time flags
CXXFLAGS        := -std=c++17 -Wl,-ld_classic -O3 -g

# define external includes
EIGEN    = $(HOME)/opt/eigen-3.4.0/
NLOPTINC = $(HOME)/opt/nlopt-2.7.1/include
SPDLOGINC   = $(HOME)/opt/spdlog/include
SELF = ./include

# Extrenals include files and folders
INCLUDES = -I$(EIGEN) -I$(NLOPTINC) -I$(SPDLOGINC) -I$(SELF)
```

The first variable is the c++ compiler chosen, you can modify it to use your preferred one. The CXXFLAGS variable defines the compile-time flags to be passed to the compiler and the variables EIGEN, NLOPTINC, SPDLOGINC and SELF contain the path to the include folder of the dependencies. If you have installed them in a different folder, you have to modify these variables accordingly.

You should then type the command make in the terminal to proceed.

If the compilation is successful you should now see a file called **libcmp.a** in the lib folder.

This step should also generate the Doxygen documentation for the code in a folder called docs.

2.5 Linking CMP++

I will now walk you through the suggested use of the cmp++ library. To do so, I will use the test_cmp case study included in the cmp library folder which will be thoroughly discussed in the following sections. In that folder will see the example of a workflow for the calibration of a simple computer model which can be adapted to numerous cases. In the folder you should have a main.cpp file which contains the actual code for the calibration and a Makefile which tells your system how to compile and link the code to existing libraries. I will now focus on the latter and discuss the contents of the main.cpp file in a subsequent section.

```
# define the Cpp compiler to use
CXX = g++-13

# define any compile-time flags
CXXFLAGS        := -std=c++17 -Wl,-ld_classic -O3 -g

# define external includes
EIGEN    = $(HOME)/opt/eigen-3.4.0/
NLOPTINC = $(HOME)/opt/nlopt-2.7.1/include
SPDLOG   = $(HOME)/opt/spdlog/include
CMPINC = $(HOME)/opt/CMP++/include

# define external libs
NLOPTLIB = $(HOME)/opt/nlopt-2.7.1/lib
CMPLIB = $(HOME)/opt/CMP++/lib

#include specific paths
INCLUDES = -I$(EIGEN) -I$(NLOPTINC) -I$(SPDLOG) -I$(CMPINC)

#include specific libraries
LIBS = -L$(NLOPTLIB) -L$(CMPLIB)

# define library flags
LFLAGS = -lcmp -lnlopt

# define target file
TARGET = main.cpp

# define executable
OUTPUT = out
```

This is the makefile you should modify. Apart from the c++ executable and flags you can see the *define external includes section* and the *define external libs* one. In the first one we will put the path of the header files of our 4 libraries (eigen, spdlog, nlopt, cmp++) and then add it to the **INCLUDE** variable with an -I in front. The second one contains the path of the lib forlder of our libraries (note that spdlog and eigen are header only so they do not have such folder) and should be added into the LIBS variables with an -L in front. The reader is reminded that cmp++ is compile and linked as a static library while nlopt as a dynamic library. At this point we add the library name that we want to use to the LFLAGS variable. If the name of the library is lib**mylib**.dylib the convention is to discard the prefix lib and extension .dylib and append an -l at the beginning; so it becomes -lcmp.

The last line contain the name of the target to be compiled, in this case main.cpp, and the name of the executable to be generated, in this case out.

3 Test-cases

3.1 Test gp

This test case showcases the Gaussian Processes (GPs) library included in CMP++. The library has the capability to train GPs and use them for prediction. We assume to have observations, in the range $[0, 1]$, from the function

$$f(x) = \sin x + 1 \tag{1}$$

And we choose to train a GP with the following mean and covariance function

$$\begin{aligned} \mu_\psi(x) &= \psi_0 \\ c_\psi(x,y) &= \eta^2 \delta_{x-y} + \psi_1^2 \exp -\frac{1}{2}\left(\frac{x-y}{\psi_2}\right)^2 . \end{aligned} \tag{2}$$

Where η, termed nugget, is a small constant to ensure positivity of the resulting covariance matrix and $\boldsymbol{\psi} = (\psi_0, \psi_1, \psi_2)$ are the hyperparameters of the GP.

To proceed we first define the observations which are n points sampled uniformly in the interval [0,1] and corresponding to the function $sin(x) + 1$.

After having generated the observations and defined a few quantities we proceed by defining the most important functions for the Gaussian Process which are the mean, kernel and the prior on the hyper-parameters

```cpp
// Define GP-kernel
auto kernel = [](vector_t x, vector_t y, vector_t hpar){
    return ....
};

// Define GP-mean
auto mean = [](const vector_t &x, const vector_t &hpar){
    return ...;
};

// Define log-prior
auto logprior = [](const vector_t &par){
    return ...;
};
```

We make use of lamnda functions to simplify the code. After these quantities are created, we can proceed by defining the GP

```cpp
gp my_gp;
my_gp.set_mean(mean);
my_gp.set_kernel(kernel);
my_gp.set_par_bounds(-2*vector_t::Ones(3),2*vector_t::Ones(3));
my_gp.set_logprior(logprior);
my_gp.set_obs(v_to_vvxd(x_obs),y_obs);
```

Here we have initialized the gp object and used the set method to define the mean, the kernel, the log-prior and the bounds to be used during the optimization.

Now the optimal hyper-parameters can be found by using the built in optimization routine which implements the MAP optimization

```cpp
vector_t par_opt = my_gp.par_opt(par_guess,1E-5);

// Compute reisiduals, and the LDLT decomp.
auto cov = my_gp.covariance(par_opt);
auto res = my_gp.residual(par_opt);
auto ldlt = Eigen::LDLT<matrix_t>(cov);
```

If the required quantities quantities have not been defined the routine will not work at runtime. Here, as you can see, it is only necessary to pass a suitable guess and the tolerance.

We immediately compute the residuals and the LDLT decomposition of the covariance matrix which can be saved and re-used for many computations. An example, is their use in the predictive equations to compute the mean and the variance at a new point $\mathbf{x}^*$. This is done by the function *predict* which return these quantities in a matrix. The data is then saved in a .csv file and read by a python script.

3.2 test cmp

This test-case is designed in order to test the cmp method on the calibration of a simple model. The model was inspired by my master thesis[10] and the computer model should be an inadequate representation of the functional relationship between the thrust coefficient, C_T and the dimensionless voltage, $\hat{V}$, of an ionic thruster. The model, depends on two parameters, θ_1 and θ_2 and reads:

$$C_T = \theta_1 \tanh \frac{\theta_2}{\theta_1} \left(\hat{V} - 1 \right) . \tag{3}$$

To calibrate it, we use 8 experimental points contained in the file data.csv. The error term is assumed to be made up of two components. This first one is the model error term which is a Gaussian process with zero mean and squared exponential kernel as covariance. The second one is the experimental error and we choose a white noise process. The error covariance function reads

$$c_\psi(x, y) = \sigma_e^2 \delta_{x-y} + \sigma^2 \exp -\frac{1}{2} \left(\frac{x-y}{l} \right)^2 . \tag{4}$$

So we have 2 parameters and 3 hyperparameters. To avoid constraints, we use the log of the hyperparameters. The prior is left to be uniform for the model parameters and a combination of inverse gamma functions for the hyperparamters.

[10]`https://omarkahol.github.io/projects/1_project/`

```
// Model to calibrate
double model(const vector_t &x, const vector_t &par) {
    double V_hat = x(0)-1;
    return par(0)*tanh(par(1)*V_hat/par(0));
}

// Kernel
double err_kernel(const vector_t & x, const vector_t &y, const
 ↪  vector_t & hpar) {
    double sigma_e = exp(hpar(0));
    double sigma_k = exp(hpar(1));
    double l = exp(hpar(2));

    return white_noise_kernel(x,y,sigma_e) +
     ↪  squared_kernel(x,y,sigma_k,l);
}

// kernel gradient, i is the component required
double err_kernel_gradient(const vector_t & x, const vector_t
 ↪  &y, const vector_t & hpar, const int &i) {
    return ...;
}

// kernel hessian, i and j are the row and colum respectively
double err_kernel_hessian(const vector_t & x, const vector_t
 ↪  &y, const vector_t & hpar, const int &i, const int &j) {
    return ...;
}

// hyperparameter prior
double logprior_hpar(const vector_t & hpar) {
    double sigma_e = exp(hpar(0));
    double sigma_k = exp(hpar(1));
    double l = exp(hpar(2));
    return log_inv_gamma_pdf(exp(hpar(0)),3,0.04)+...;

// prior hessian
double logprior_hpar_hessian(const vector_t &hpar, const int
 ↪  &i, const int &j) {
        return ...;
}
```

Before the main function, we need to define a few important quantities. In particular we should define the model function which should evaluate the computer model on a point. Note that the dimension can be larger than 1 (for example **x** = x,y,z,t) so we use a Eigen::VectorXd type. We should also define the error kernel which depends on two points x and y and the vector of the hyperparameters and the log-prior. A handful of covariance function, their gradients, pdfs and their gradients is already implemented in pdf.h and kernel.h.

The first few lines of the main function are dedicated to setup operations like reading an input file containing the data and initializing the simulation. note that we should define

1. An initial proposal value for the parameters (to be used as starting point for the MCMC)
2. An initial proposal value for the hyperparameters and their bounds (all in log) to be used as starting point for the MCMC and during the optimization.
3. A proposal covariance function for the simulation.

When this is done, we have to define the mean function of the error model and the log prior of the parameters. Because we choose uniform prior and zero mean we can implement these function using lambdas.

```

//Create the model error
gp model_error;
model_error.set_mean(err_mean);
model_error.set_kernel(err_kernel);
model_error.set_kernel_gradient(err_kernel_gradient);
model_error.set_kernel_hessian(err_kernel_hessian);

model_error.set_logprior(logprior_hpar);
model_error.set_logprior_hessian(logprior_hpar_hessian);
model_error.set_par_bounds(lb_hpar,ub_hpar);
model_error.set_obs(v_to_vvxd(x_obs),y_obs);


//create density
density main_density;

// set up the main properties
main_density.set_model(model);
main_density.set_model_error(&model_error);
main_density.set_log_prior_par(logprior_par);
main_density.set_obs(v_to_vvxd(x_obs),y_obs);

```

Now we initialize the two main objects for the simulation which are the model error gp class and the main density class. Make sure to initialize every quantity properly or the simulation will not work.

```


// create the getter function for the hyperparameters. It is an
↪  optimization
get_hpar_t get_hpar = [&main_density](const vector_t &par,
↪  const vector_t &hpar) {
    return main_density.hpar_opt(par,hpar,1e-4);
};

// create the score function
score_t score = [&main_density,&model_error,&x_obs](const
↪  vector_t & par, const vector_t & hpar) {

    auto res = main_density.residuals(par);
    auto cov = model_error.covariance(hpar);
    Eigen::LDLT<matrix_t> cov_inv(cov);

    return main_density.loglikelihood(res,cov_inv) +
     ↪  main_density.logprior_par(par) +
     ↪  main_density.model_error()->logprior(hpar);
};

in_bounds_t in_bounds = [&main_density](const vector_t & par) {
    return true;
};

```

Using lambda functions we initialize two other important functions. The first one is the getter function for the hyper-parameters which given the current value of the model parameters and the previous value of the hyper-parameters should return the current value of the hyperparameters. Depending on the value returned we can implement different methods

1. We can return some elements of the par vector if the hyperparameters are sampled together with the model parameters (for the full bayesian method)
2. We can return the maximizer of the posterior (like the CMP and FMP methods)
3. We can return a fixed value (like the KOH method)

In our case we choose to maximize the posterior so we call the *hpar_opt* function which performs the optimization using the NLopt library.

the other function required is the one to compute the MCMC score. For the CMP method this is equivalent to the CMP approximation of the un-normalized posterior distribution which is made up by three contributions

1. log-likelihood
2. log-prior
3. log-correction factor

To compute them we first evaluate the covariance matrix and the residuals and then proceed by finding the Cholesky decomposition of the covariance matrix and then call the respective functions.

When this is done we can just proceed by performing the MCMC simulation. *Tutorial on MCMC coming soon*.

3.2.1 results

Once saved, the samples and the predictions can be used together with the python scripts to generate visualizations.

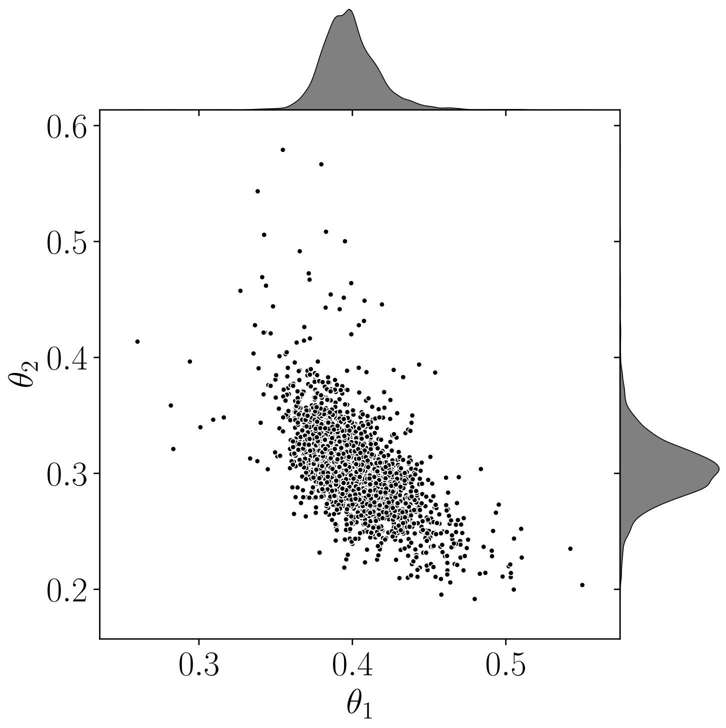

Figure 1: Posterior distribution

Fig. 1 shows the resulting posterior distribution along with the KDE of the marginal posteriors.

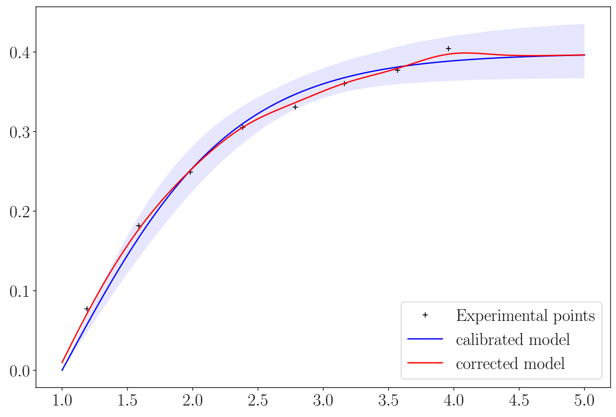

Figure 2: Predictive distribution

Fig. 2 shows the resulting predictive posterior distribution of the corrected and calibrated model.