The NoisyFunOpt C++ Library

Francesco Calcavecchia

January 12, 2016

NoisyFunOpt is a C++ library that contains simple tools for optimising (minimize or maximize) a noisy function, such as an integral computed with the Monte Carlo technique. It currently supports only the Conjugate Gradient algorithm, with some minor modifications made to handle the noise.

More specifically, two function values `a` and `b` with associated standard deviations `da` and `db` are considered to be:

- $\texttt{a} > \texttt{b}$ iff $\texttt{a} - 2\texttt{da} > \texttt{b} - 2\texttt{db}$;
- $\texttt{a} < \texttt{b}$ iff $\texttt{a} + 2\texttt{da} < \texttt{b} + 2\texttt{db}$;
- $\texttt{a} = \texttt{b}$ otherwise.

The code has been developed using the standard C++11.

1 Include the library

First of all one has to include the MCIntegrator library, using the instruction:

```
#include "MCIntegrator.hpp"
```

2 Integrator's declaration and inizialization

```
> MCI(const int & ndim)
> int getNDim()
```

In the following we will use the variable name `mci` for labeling a `MCIntegrator` object. An integrator can be declared as

```
MCI mci(ndim);
```

where `ndim` in an `int` that specifies the dimensionality of the integral.

One can know `ndim` at any time by invoking

```
mci.getNDim()
```

In the following we will always refer to `mci` as the default `MCI` object.

3 Settings

It is not strictly necessary to set all the following settings because there are already default values, however the user should be aware of their existence.

3.1 Integral domain

```
> void setIRange(const double * const * irange)
> double getIRange(const int &i, const int &j)
```

One can set the integral domain with the method

```
mci.setIRange(irange);
```

where `irange` is a `double **` of dimension `ndim` $\times$ 2. For example, in the one-dimensional case, to integrate between `La` and `Lb`, one has to set `irange[0][0]=La` and `irange[0][1]=Lb`. It is always assumed that the boundaries are given in increasing order, e.g. `La` $<$ `Lb`. If the integral range is not set, it is assumed by default to be $\mathbb{R}^{\texttt{ndim}}$.

Once the integral domain is set, the initial coordinates and the $\mathrm{M(RT)}^2$ step are set accordingly. Specifically, the coordinates are set to be in the middle of the integration volume, whereas the step is set to be half of the integration sides for each direction.

One can get at any moment the domain's boundary `irange[i][j]` by invoking

```
mci.getIRange(i,j)
```

3.2 Initial coordinates

```
> void setX(const double * x);
> double getX(const int &i);
```

The Markov chain is built starting from an initial point. By default this is assumed to be in the middle of the integration space, however it might be convenient to set it manually in certain specific situations. This can be done with the command

```
mci.setX(x);
```

where `x` is `double *` with size `ndim`.

One can get the value of `x` at any moment by invoking

```
mci.getX()
```

3.3 M(RT)^2 step

```
> void setMRT2Step(const double * mrt2step);
> double getMRT2Step(const int &i);
```

The initial M(RT)^2 step is set equal to 0.1 by default for every direction.

It is not essential to provide this parameter, because before proceeding with the integration, `MCIntegrator` adjust the step in order to obtain an acceptance close to the target one (by default 50%, see subsection 3.4). In any case, setting by hand a reasonable value can result in a tuning speed-up. Use

```
mci.setMRT2Step(mrt2step);
```

where `mrt2step` is a `double *` of size `ndim`.

One can get the value of `mrt2step` (most notably after a tuning process, e.g. after an integration) by invoking

```
mci.getMRT2Step(i)
```

where `i` is a `int` between 0 and `ndim` $-$ 1.

3.4 Acceptance rate

```
> void setTargetAcceptanceRate(const double * targetaccrate);
> double getTargetAcceptanceRate();
> double getAcceptanceRate();
```

In the context of the M(RT)^2 algorithm, the acceptance rate is one of the most important parameters to control. Before proceeding with the integration, `MCIntegrator` automatically adjust the M(RT)^2 step in order to obtain an acceptance rate close to the provided target one (by default 50%). A target acceptance rate of 50% provides very good performance in almost all cases. We remark that if the user does not specify a sampling function, the acceptance rate will always be 100%, independently of the M(RT)^2 step, since the M(RT)^2 algorithm will not be employed.

For setting the target acceptance rate use

```
imc.setTargetAcceptanceRate(targetaccrate)
```

where `targetaccrate` is a `double *` of dimension `1` and must point tot a number between 0 and 1. This value can be checked at any time by invoking

```
imc.getTargetAcceptanceRate()
```

After performing an integration one can check the actual acceptance rate by invoking

```
imc.getAcceptanceRate()
```

4 Observable function

```
>  void addObservable(MCIObservableFunctionInterface * obs);
>  int getNObs();
>  int getNObsDim();
>  MCIObservableFunctionInterface * getObservable(const int &i)
>  void clearObservables();
```

In order to perform an integral it is essential to specify at least an observable function. Actually, one can even add more than one and form an *observables' stack*. The integral of all of these functions will be done simultaneously. In subsection 4.1 we will discuss how it is possible to create an object that implements an observable function, scalar or multidimensional.

In order to add an observable object to the stack, just use

```
mci.addObservable(obs);
```

where `obs` is a `MCIObservableFunctionInterface *`. Subsection 4.1 provides more detail about this virtual class and how an actual observable class can be implemented.

The number of observables in the stack can be obtained with

```
mci.getNObs()
```

The total number of observable functions (i.e. the sum of the `nobs` values of all observable objects added in the stack - see subsection 4.1) in the stack can be obtained by invoking

```
mci.getNObsDim()
```

One can obtain the pointer to an observable object by means of

```
mci.getObservable(i)
```

where `i` is an `int` between 0 and `mci.clearObservables()` $-$ 1 and refers to its position in the observables' stack.

Finally, one can clean the stack of observable objects by using

```
mci.clearObservables();
```

4.1 MCI observable function interface

```
<  MCIObservableFunctionInterface(const int &ndim, const int &nobs);
<  virtual void observableFunction(const double * in, double *out) = 0;
>  int getNDim();
>  int getNObs();
>  void computeObservables(const double *in);
>  double getObservable(const int &i);
```

An observable class must be declared as a child class of `MCIObservableFunctionInterface`, and must implement its *virtual protected method* `observableFunction`. This function will take an input `in`, an array of size `ndim`, and has to set the values to an output array `out`, of size `nobs`.

As an example, here is the code for having an observable for a quadratic function x^2:

```
class QuadraticObservable: public MCISamplingFunctionInterface
{
   public:
      Polynom(const int &ndim): MCIObservableFunctionInterface(ndim, 1) {}
      // nobs will always be equal to 1,
      // since quadraticObservable is a scalar function

   protected:
      void observableFunction(const double * in, double * out)
      {
         out[0]=0.;
         for (int i=0; i<this->getNDim(); ++i)
         {
            out[0] += in[i]*in[i];
         }

      }
};
```

Once the class has been defined, one can allocate an observable object (here we use the example of the quadratic observable)

```
MCIObservableFunctionInterface * obs = new QuadraticObservable(ndim);
```

Notice that we are here making use of *polymorphism*.

Then one can access the values of `ndim` and `nobs` by invoking

```
obs->getNDim()
```

and

```
obs->getNObs()
```

For debugging purposes, one might want to compute some values with the observable function. This can be achieved by using

```
obs->computeObservables(in);
```

where `in` will be the input array of size `obs->getNDim()`, followed by the getter

```
obs->getObservable(i)
```

where `i` is an `int` between 0 and `obs->getNObs()` -1 and refers to the index of `out` that on is interested into.

5 Sampling function

```
>  void addSamplingFunction(MCISamplingFunctionInterface * sf);
>  int getNSampF()
>  MCISamplingFunctionInterface * getSamplingFunction(const int &i)
>  void clearSamplingFunctions();
```

Whenever it is possible, it is convenient to incorporate as much as possible of the integrand inside the sampling function, following the *importance sampling* principle. the user is responsible for providing a correct sampling function, i.e. positive definite and normalised to 1. In subsection 5.1 we will describe how is it possible to create a class that implements a sampling function.

One can add a sampling function to the Monte Carlo integration object with the command

```
mci.addSamplingFunction(sf);
```

where `sf` is a `MCISamplingFunctionInterface *`, a virtual class whose usage will be clarified in subsection 5.1. It is possible to add multiple sampling functions, the final sampling function will be the product of all of them.

The number of sampling functions in the stack can be obtained with

```
mci.getNSampF()
```

It is possible to get the pointer to a provided sampling function at any moment by invoking

```
mci.getSamplingFunction(i)
```

where `i` is an `int` between 0 and

Finally, It is possible to clean the sampling functions' stack by means of

```
mci.clearSamplingFunctions();
```

5.1 MCI sampling function interface

```
<  MCISamplingFunctionInterface(const int &ndim, const int &nproto);
<  virtual void samplingFunction(const double *in, double * protovalue) = 0;
<  virtual double getAcceptance() = 0;
>  int getNDim();
>  int getNProto();
>  double getProtoNew(const int &i);
>  double getProtoOld(const int &i);
>  void computeNewSamplingFunction(const double * in);
>  void newToOld();
```

A sampling function class must be declared as a child class of `MCISamplingFunctionInterface`, and must implement its *virtual public methods* `samplingFunction` and `getAcceptance`.

The sampling function class is supposed to get an input `in`, of size `ndim`, and compute `nproto` temporary values, called *proto-values*, that will be used to compute the final acceptance function. These proto-values can be accessed by means of the public methods `getProtoOld` and `getProtoNew`

As an example, here is the code for having a gaussian sampling function e^{-x^2}:

```
#include <math.h>

class GaussianSF: public MCISamplingFunctionInterface
{
   public:
      Gauss(const int &ndim): MCISamplingFunctionInterface(ndim,1) {}
      // For a gaussian sampling function, one proto-value is sufficient
      // therefore we set it equal to 1 by default

      void samplingFunction(const double * in, double * protovalue)
      {
         protovalue[0]=0.;
         for (int i=0; i<this->getNDim(); ++i)
         {
            protovalue[0] += (in[i])*(in[i]);
         }
      }

      double getAcceptance()
      {
         return exp(-( this->getProtoNew(0) - this->getProtoOld(0) ));
      }
};
```

Once the class has been defined, one can allocate an observable object (here we use the example of the gaussian sampling function)

```
MCISamplingFunctionInterface * sf = new GaussianSF(ndim);
```

Notice that we are here making use of *polymorphism*.

Then one can access the values of `ndim` and `nproto` by invoking

```
sf->getNDim()
```

and

```
sf->getNProto()
```

For debugging purposes, one might want to compute some values with the sampling function. This can be achieved by using

```
sf->computeNewSamplingFunction(in);
```

where `in` is a `double *` of size `sf->getNDim()`. As a result of this command, the proto-values *new* will have an updated value corresponding to `in`. Then, one might invoke

```
sf->newToOld()
```

so that the proto-values *old* will have the value of the new ones. Then one might call again

```
sf->computeNewSamplingFunction(in2);
```

where `in2` is a `double *` of size `sf->getNDim()`, say, different from `in`. At this point the old and new proto-values will be different from each other, as might be checked by using

```
sf->getProtoNew(i);
```

and

```
sf->getProtoOld(i);
```

where `i` is an `int` between 0 and `sf->getNProto()` -1. Finally, the acceptance rate from `in` and `in2` might be evaluated by means of

```
sf->getAcceptance()
```

6 Integration

```
> void integrate(const long &Nmc, double * average, double * error);
```

Once that all the settings are done, one can obtain the result of the integral by invoking

```
mci.integrate(Nmc, average, error)
```

where `Nmc` is a `long int` and must be provided as input, whereas `average` and `error` are of type `double *` with size `mci.getNObsDim()` (reference to this last method in 4). `Nmc` is the number of sampled points: The larger this value, the more accurate will be the result, according to the well known $1/\sqrt{\texttt{NMC}}$ rule. `average` will contain the resulting numeric estimation of the integral with an estimated standard deviation stored in `error`.

The Monte Carlo integral is composed by several substep, completely hidden to the user. We report them for general knowledge.

1. if a sampling function is provided, determine a $M(RT)^2$ step corresponding (approximately) to the target acceptance rate (by default 0.5);

2. if a sampling function is provided, perform some warming up steps. This is done with sequential blocks of 100 steps, computing the observables each time. When the new computed averages and error bar are equal to the old one (within the error bars), the warming up procedure is stopped;

3. sample `Nmc` steps and internally store the corresponding observables. This is done by means of the $M(RT)^2$ algorithm if a sampling function has been provided, and by sampling random numbers otherwise. In the first case there will be autocorrelations between successive values of the observables, in the latter case data will be completely uncorrelated;

4. compute the final averages and standard deviations of the integral. If a sampling function was provided, this is done using the blocking techniques to account for autocorrelations.

7 Estimation of the average and its standard deviation

```
>  void UncorrelatedEstimator(const long &n, const double * x,
                              double * average, double * error);
>  void BlockEstimator(const long &n, const double * x,
                       const int &nblocks,
                       double * average, double * error);
>  void CorrelatedEstimator(const long &n, const double * x,
                            double * average, double * error);
```

Given an array `double *`, one can use this library to extract its average value and standard deviation. To do so, one has to first declare the module

```
#include "Estimators.hpp"
```

and then use one of the three available functions (notice that the *namespace* `mci` is required):

- `mci::UncorrelatedEstimator(...)`
  where `n` is an `int`, and `x` a `double *` of size `n`. `average` will point to the average value, and `error` to the standard deviation. This function should be used only with uncorrelated data;

- `mci::BlockEstimator(...)`
  where `nblocks` is a `int` that refers to the number of blocks. This

function computes average and standard deviation dividing the data into `nblocks` blocks, computing their averages, and then use this new set of data to compute their average and standard deviation as if they were uncorrelated.

- `mci::CorrelatedEstimator(...)`
  here data are considered to be correlated, and the blocking technique is used to correctly estimate the standard deviation;

7.1 Multidimensional estimations

```
> void MultiDimUncorrelatedEstimator(
                    const long &n, const int &ndim,
                    const double * const * x,
                    double * average, double *error);
> void MultiDimBlockEstimator(
                    const long &n, const int &ndim,
                    const double * const * x,
                    const int &nblocks,
                    double * average, double * error);
> void MultiDimCorrelatedEstimator(
                    const long &n, const int &ndim,
                    const double * const * x,
                    double * average, double * error);
```

There is a version of the functions discussed in section 7 specifically done for multidimensional data. In this case the new parameter `ndim` enters into play (it is an `int`). Data (`x`) are supposed to be organised as a `n` $\times$ `ndim` matrix.