The NoisyFunOpt C++ Library

Francesco Calcavecchia

October 8, 2018

NoisyFunOpt is a C++ library that contains simple tools to optimise (minimize) a noisy function, e.g. an integral computed with the Monte Carlo technique. More specifically, two function values `a` and `b` with associated standard deviations `da` and `db` are considered to be:

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

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

In the following we will present the classes made available by the library. At the beginning we will report the necessary `#include` call and the prototype of the class. The comment `TO DO` indicates that the method needs to be implemented (as in the case of a pure virtual class).

1 NoisyFunction

```
\\ #include "NoisyFunction.hpp"
class NoisyFunction
{
   NoisyFunction(int);
   virtual ~NoisyFunction();

   int getNDim();

   virtual void f(const double *, double &, double &) = 0;   \\
      TO DO
};
```

A `NoisyFunction` implements a function $f : \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}$ which takes a multidimensional vector as input, and returns a scalar value with an associated error bar.

- `NoisyFunction(int ndim)`: The constructor. `ndim` must contains the value of n, i.e. the number of dimensions of the input;

- `~NoisyFunction()`: Virtual destructor. It does nothing;
- `getNDim()`: It returns the `ndim` provided with the constructor;
- `f(const double *x, double &val, double &dval)`: The implementation of f. It takes an array `x` as input and returns in `val` the value of the function and in `dval` the error bar associated with it.

2 NoisyFunctionWithGradient

```
\\ #include "NoisyFunction.hpp"
class NoisyFunctionWithGradient: public NoisyFunction
{
   NoisyFunctionWithGradient(int);
   virtual ~NoisyFunctionWithGradient();

   virtual void grad(const double *, double *, double *) = 0; //
       TO DO
};
```

`NoisyFunctionWithGradient` is a child class of `NoisyFunction` (therefore the implementation of the method `f` is required), and it aims to cover the cases in which the gradient of f is known.

- `NoisyFunctionWithGradient(int ndim)`: The constructor. As in `NoisyFunction`;
- `~NoisyFunctionWithGradient()`: The destructor. It does nothing;
- `grad(const double *x, double *grad, double *dgrad)`: The gradient of f. It takes `x` as input, and returns the gradient in `grad` and the corresponding errors in `dgrad`.

3 NFM

```
// #include "NoisyFunMin.hpp"
class NFM{
   NFM(NoisyFunction *);
   virtual ~NFM();

   void setX(const double *);
   void setX(const int &, const double &);
   void setGradientTargetFun(NoisyFunctionWithGradient *);
   void setEpsTargetFun(double &);
   void setEpsX(double &);

   int getNDim();
   double getX(const int &);
   void getX(double * x);
```

```
    double getF();
    double getDf();
    NoisyFunctionWithGradient* getGradientTargetFun();
    double getEpsTargetFun();
    double getEpsX();

    virtual void findMin() = 0; // TO DO
};
```

NFM is an interface for a generic minimisation method. Any actual implementation of this interface will have to specify the findMin method.

The minimisation process will start from a point x, set with the method setX, and wil continue until the minimum has been found. At the end of this process, x will have to be in this minimum and will be accessible with the method getX.

- NFM(NoisyFunction *f): The constructor. It takes as input a target function f;
- ~NFM(): The destructor. It does nothing;
- setX(const double *x): Set the starting point of the minimisator. By default it is set equal to 0;
- setX(const int &, const double &): Like above, but set only one x according to index.
- setGradientTargetFun(NoisyFunctionWithGradient * grad): Set a target function with gradient, in case it is available and/or the actual implementation of NFM requires it;
- setEpsTargetFun(double &eps): Set the value of the minimum change in the target function value that will make continue the minimisation process. This parameter might be irrelevant, depending on the actual implementation of the minimisation algorithm;
- setEpsX(double &eps): Set the value of the minimum change in the minimum point that will make continue the minimisation process. This parameter might be irrelevant, depending on the actual implementation of the minimisation algorithm;
- getNDim(): It returns the number of dimensions of the space in which the function is minimised;
- getX(const int &i): It returns the i-th coordinate of the actual position of x, which will coincide with the minimum after the minimisation process;
- getF(): It returns the value of f in x;

- `getDf()`: It returns the error bar associated with the value of $f(\mathtt{x})$;
- `getGradientTargetFun()`: It returns the pointer to the target function with gradient, if there is one. Otherwise returns `0`;
- `getEpsTargetFun()`: It returns the value set with `setEpsTargetFun`, which is 0 by default;
- `getEpsX()`: It returns the value set with `setEpsX`, which is 0 by default;
- `findMin()`: The minimisation subroutine. At the end of this process, `x` is supposed to be in the minimum of the target function. Therefore, after calling this method, it will be possible to know where the minimum is by using the method `getX`, while its corresponding value and error bar will be directly accessible with `getF` and `getDf`.

4 Log report

It is possible to activate the log report for following the algorithm internal dynamic. To do so, simply use the following commands:

```
\\ #include "LogNFM.hpp"

NFMLogManager log;
log.setLoggingOn();
```

This will automatically output the log messages on the stdout (i.e. print on screen). It is possible to make it write it on a file instead. To do so:

```
log.setLoggingPathFile("NFM.log");
```

Finally, it is possible to interrupt the log output in any moment using the command

```
log.setLoggingOff();
```

and to explicitly write a message in the log report

```
log.writeOnLog("message to write on the log");
```

5 ConjGrad

```
\\ #include "ConjGrad.hpp"
class ConjGrad: public NFM
{
   ConjGrad(NoisyFunctionWithGradient *);
```

```
    ~ConjGrad();

    void configureToFollowSimpleGradient()
};
```

Implementation of the Conjugate Gradient algorithm, using the *Para-Gold Search* method for the one-dimensional minimisation (see https://publications.ub.uni-mainz.de/theses/volltexte/2014/3805/pdf/3805.pdf, page 61).

It requires a `NoisyFunctionWithGradient` for running.

- `ConjGrad(NoisyFunctionWithGradient *f)`: The constructor. It requires an input target function with gradient `f`;
- `~ConjGrad()`: The destructor. It does nothing;
- `configureToFollowSimpleGradient()`: It asks the algorithm to ignore computing the conjugate gradient, but rather use the simple gradient. Needless to say, this method should be called before the subroutine `findMin`;

6 DynamicDescent

```
\\ #include "DynamicDescent.hpp"
class DynamicDescent: public NFM
{
    DynamicDescent(NoisyFunctionWithGradient * targetfun, bool
        useGradientError = false, size_t &max_n_const_values = 20)
        ;
    ~DynamicDescent();
};
```

This implementation is extremely simple, however it can be useful in certain circumstances. To minimise the function it uses the gradient (actually $-\nabla$) of the function as direction for moving. For deciding "how much" to move in that direction, it multiplies the gradient with a scalar value λ (named `_inertia` in the code) set in the following way:

1. $\lambda_0 \equiv \frac{Ndim}{|\nabla_0|}$ at the first step;
2. $\lambda_{i+1} = \lambda_i + \frac{1}{2}\lambda_i < \nabla_{i+1}, \nabla_i >$ ($< \cdots >$ symbolises a scalar product) at any other step;

This iterative procedure should ensure that when we are far from the minimum and we keep moving in the same direction, *lambda* grows, whereas when we are close to the minimum, and the direction keeps changing, *lambda* is quickly reduced to almost zero.

The aforementioned iterative procedure stops when the last `max_n_const_values` (default to 20) points found are equal, meaning that the procedure has stabilised around a minimum. If `useGradientError` is `true`, the optimization will also terminate when all gradient elements are smaller than their respective errors.

- `DynamicDescent(NoisyFunctionWithGradient *f)`: The constructor. It requires an input target function with gradient `f`;
- `~DynamicDescent()`: The destructor. It does nothing;

7 Adam

```
\\ #include "Adam.hpp"
class Adam: public NFM
{
   Adam(NoisyFunctionWithGradient * targetfun, bool
       useGradientError = false, size_t &max_n_const_values = 20,
       double &alpha = 0.001, double &beta1 = 0.9, double &beta2
       = 0.999, double &epsilon = 10e-8);
};
```

This is an implementation of the popular Adam minimization algorithm (https://arxiv.org/pdf/1412.6980.pdf). The article provides reference for algorithm and parameters above, which have the same names as in the paper.

The optimizer requires a gradient, but no error terms on that. The procedure stops on convergence, handled in the same way as in DynamicDescent.