The VMC++ Library

Francesco Calcavecchia

March 9, 2018

VMC++ (Variational Monte Carlo Plus Plus) is a C++ library which allows with few simple commands for the implementation of a Variational Monte Carlo calculation. It provides the tools for describing a trial wave function and an Hamiltonian, and it uses this information to find the optimal values of the variational parameters embedded in the trial wave function, and therefore find the best approximation of the ground state.

The code has been developed using the standard C++11, and requires the MCI++ (https://github.com/francesco086/MCIntegratorPlusPlus) and NoisyFunMin (https://github.com/francesco086/NoisyFunMin) libraries.

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).

First, we will present the virtual classes that should be instantiated for describing the trial wave function and the Hamiltonian. Then, we will introduce the class that performs the whole Variational Monte Carlo calculation.

1 WaveFunction

```
// #import "WaveFunction.hpp"

/*
IMPLEMENTATIONS OF THIS INTERFACE MUST INCLUDE:

    - void setVP(const double *vp)
            set the variational parameters

    - void getVP(double *vp)
            get the variational parameters

    - void samplingFunction(const double * in, double * out)
            heritage from MCISamplingFunctionInterface, uses Psi
                ^2

    - double getAcceptance()
            heritage from MCISamplingFunctionInterface
```

```
17
18      - void computeAllDerivatives(const double *x)
19              use the setters for derivatives values (setD1DivByWF
                    , setD2DivByWF, etc.)
20
21  */
22
23  class WaveFunction: public MCISamplingFunctionInterface, public
        MCICallBackOnAcceptanceInterface{
24  public:
25      WaveFunction(const int &nspacedim, const int &npart, const
            int &ncomp, const int &nvp,
26          bool flag_vd1=true, bool flag_d1vd1=true, bool
                flag_d2vd1=true);
27      virtual ~WaveFunction();
28
29      int getNSpaceDim();
30      int getTotalNDim();
31      int getNPart();
32      int getNVP();
33
34
35      // --- interface for manipulating the variational parameters
36      virtual void setVP(const double *vp) = 0;     // --- MUST BE
            IMPLEMENTED
37      virtual void getVP(double *vp) = 0;     // --- MUST BE
            IMPLEMENTED
38
39
40      // --- computation of the derivatives
41      // When called, this method computes all the internal values
            , such as the derivatives,
42      // and stored them internally, ready to be accessed with the
             getters methods.
43      // It requires the positions as input
44      virtual void computeAllDerivatives(const double *x) = 0;
            // --- MUST BE IMPLEMENTED
45
46
47      // --- getters and setters for the derivatives
48      // first derivative divided by the wf
49      void setD1DivByWF(const int &id1, const double &d1_logwf);
50      double getD1DivByWF(const int &id1);
51
52      // second derivative divided by the wf
53      void setD2DivByWF(const int &id2, const double &d2_logwf);
54      double getD2DivByWF(const int &id2);
55
56      // variational derivative divided by the wf
57      bool hasVD1();
58      void setVD1DivByWF(const int &ivd1, const double &vd1_logwf)
            ;
59      double getVD1DivByWF(const int &ivd1);
60
```

```
61      // cross derivative: first derivative and first variational
            derivative divided by the wf
62      bool hasD1VD1();
63      void setD1VD1DivByWF(const int &id1, const int &ivd1, const
            double &d1vd1_logwf);
64      double getD1VD1DivByWF(const int &id1, const int &ivd1);
65
66      // cross derivative: second derivative and first variational
            derivative divided by the wf
67      bool hasD2VD1();
68      void setD2VD1DivByWF(const int &id2, const int &ivd1, const
            double &d2vd1_logwf);
69      double getD2VD1DivByWF(const int &id2, const int &ivd1);
```

The class `WaveFunction` is a pure virtual class introduced for representing the trial wave function. The first thing that should be noticed, is that it is a child class of the MCI++ classes `MCISamplingFunctionInterface` and `MCICallBackOnAcceptanceInterface`. Please refer to the user manual of MCI++ to see the details of this class. The user must be able to correctly implement the `MCISamplingFunctionInterface`'s methods `samplingFunction` and `getAcceptance`.

Let us now see in details the methods of the class `WaveFunction`:

- `WaveFunction`: The constructor. It is necessary to specify the number of dimensions of the space in which it is used (`nspacedim`, typically 3 dimensions), the number of particles that the wave function describes (`npart`), the number of components of the wave function (`ncomp`, which corresponds to `nproto` of the class `MCISamplingFunctionInterface`), and the number of variational parameters used (`nvp`). Moreover, with the optional parameters `flag_vd1`, `flag_d1vd1`, and `flag_d2vd1`, one can specify if the variational derivatives $\frac{1}{\psi(x)}\frac{\partial\psi(x)}{\partial\alpha_i}$, $\frac{1}{\psi(x)}\frac{\partial^2\psi(x)}{\partial x_i\partial\alpha_i}$, and $\frac{1}{\psi(x)}\frac{\partial^3\psi(x)}{\partial x_i^2\partial\alpha_i}$ are computed or not;
- `getNSpaceDim`: Returns `nspacedim`;
- `getNPart`: Returns `npart`;
- `getNVP`: Returns `nvp`
- `setVP`: Set the variational parameters according to the provided `nvp`-dimensional array `vp`. Must be implemented by the user;
- `getVP`: Store in the `nvp`-dimensional array `vp` the values of the variational parameters. Must be implemented by the user;
- `computeAllDerivatives`: This method must be implemented to compute all the derivatives that the wave function should implement. Once computed, all the derivatives, divided by the value of the wave function

itself, should be stored internally using the methods `setD1DivByWF`, `setD2DivByWF`, `setVD1DivByWF`, `setD1VD1DivByWF`, `setD2VD1DivByWF`;

- `hasVD1`, `hasD1VD1`, `hasD2VD1`: Tells whether the wave function implements the variational derivatives or not;
- `getD1DivByWF`, `getD2DivByWF`, `getVD1DivByWF`, `getD1VD1DivByWF`, `getD2VD1DivByWF`: In case the values were computed, returns the derivatives divided by the wave function value.

1.1 Two-Body Jastrow

There is a set of classes already prepared if you want to define a Two-Body Jastrow.

First of all you need to define the metric of your system, by implementing a child class of `Metric`:

```
class Metric{
public:
    Metric(const int &nspacedim);

    // --- Methods that must be implemented
    virtual double dist(const double * r1, const double * r2) =
        0;
    virtual void distD1(const double * r1, const double * r2,
        double * out) = 0;
    virtual void distD2(const double * r1, const double * r2,
        double * out) = 0;
};
```

The method `dist` returns the distance between two particles. The method `distD1` computes the first derivative of the distance function in respect to the coordinates of `r1` and `r2`. Therefore `out` must be `2*_nspacedim`-dimensional. The method `distD2` computes the second derivative. In case you are interested in the Euclidean metric, you can use the `EuclideanMetric` class, which can be instanciated simply specifying the number of space dimensions. For example, for a 3-dimensional space:

```
EuclideanMetric * em = new EuclideanMetric(3);
```

Then you need to specify the Two-Body Pseudopotential by implementing a child class of `TwoBodyPseudoPotential`:

```
class TwoBodyPseudoPotential{
public:
    TwoBodyPseudoPotential(Metric * metric, const int &nvp, bool
         flag_vd1=true, bool flag_d1vd1=true, bool flag_d2vd1=
        true);
    virtual ~TwoBodyPseudoPotential();

    // --- Methods that must be implemented
```

```
    // manage variational parameters
    virtual void setVP(const double *vp) = 0;
    virtual void getVP(double *vp) = 0;
    // functions of the distance that define the pseudopotential
    virtual double ur(const double &r) = 0;
    virtual double urD1(const double &r) = 0;
    virtual double urD2(const double &r) = 0;
    virtual void urVD1(const double &r, double * vd1) = 0;
    virtual void urD1VD1(const double &r, double * d1vd1) = 0;
    virtual void urD2VD1(const double &r, double * d1vd1) = 0;
};
```

For example, the implementation of the Pseudopotential typically used for simulating He atoms:

```
class He3u2: public TwoBodyPseudoPotential{
private:
    double _b;
public:
    He3u2(EuclideanMetric * em):
    TwoBodyPseudoPotential(em, 1, true, true, true){
        _b = -1.;
    }

    void setVP(const double *vp){_b=vp[0];}
    void getVP(double *vp){vp[0]=_b;}

    double ur(const double &dist){
        return _b/pow(dist, 5);
    }
    double urD1(const double &dist){
        return -5.*_b/pow(dist, 6);
    }
    double urD2(const double &dist){
        return 30.*_b/pow(dist, 7);
    }
    void urVD1(const double &dist, double * vd1){
        vd1[0] = 1./pow(dist, 5);
    }
    void urD1VD1(const double &dist, double * d1vd1){
        d1vd1[0] = -5./pow(dist, 6);
    }
    void urD2VD1(const double &dist, double * d1vd1){
        d1vd1[0] = 30./pow(dist, 7);
    }
};
```

Finally, we are ready to declare a Two-Body Jastrow, simply by making use of the ready-to-use class `TwoBodyJastrow`:

```
EuclideanMetric * em = new EuclideanMetric(NSPACEDIM);
He3u2 * u2 = new He3u2(em);
TwoBodyJastrow * J = new TwoBodyJastrow(NPART, u2);
```

1.2 FFNNWaveFunction

In case you want to use a Neural Network as Wave Function, there is an already prepared WaveFunction. One needs to provide only the number of spacial dimensions (`nspacedim`) and number of particles (`npart`), and a FeedFowardNeuralNetwork (see the library) that has `nspacedim` $\times$ `npart` inputs and only one output. This FFNN should be already connected, but should not have any derivative substrate. As for the `WaveFunction`, some flags can be specified in the constructor to tell whether the variational derivatives are computed or not. Internally, this class create two separated instances of this FFNN, a bare one, which does not have any substrate, and is used only for sampling, and a more complex one, that is used for computing all the derivatives.

Of course, it inherits all the methods from the WaveFunction class, and we do not report it in the following.

```
class FFNNWaveFunction: public WaveFunction{

private:
    FeedForwardNeuralNetwork * _ffnn;

public:
    // --- Constructor
    // IMPORTANT: The provided ffnn should be ready to use (
        connected) and have the first, second and variational
        derivatives substrates
    FFNNWaveFunction(const int &nspacedim, const int &npart,
        FeedForwardNeuralNetwork * ffnn, bool flag_vd1=true, bool
         flag_d1vd1=true, bool flag_d2vd1=true);


    // --- Getters
    FeedForwardNeuralNetwork * getBareFFNN(){return _bare_ffnn;}
    FeedForwardNeuralNetwork * getDerivFFNN(){return _deriv_ffnn
        ;}

};
```

2 Hamiltonian

```
// import "Hamiltonian.hpp"
class Hamiltonian: public MCIObservableFunctionInterface
{
   public:
      // Constructor and destructor
      Hamiltonian(const int &nspacedim, const int &npart,
          WaveFunction * wf):
         MCIObservableFunctionInterface(nspacedim*npart,4)
      virtual ~Hamiltonian(){}
```

```
9
10      // Getters
11      int getNSpaceDim();
12      int getNPart();
13
14      // Kinetic energy
15      double localPBKineticEnergy(const double *in);
16      double localJFKineticEnergy(const double *in);
17
18      // Potential energy
19      virtual double localPotentialEnergy(const double *in) = 0;
            // TO DO
20
21      // Heritage from MCIObservableFunctionInterface
22      // already implemented
23      void observableFunction(const double * in, double *out)
24  };
```

This pure virtual class is used to define the Hamiltonian of the system. The user must take care of implementing only the `localPotentialEnergy` method, while all the rest is already provided.

- `Hamiltonian`: The constructor. It requires the spacial number of dimensions (`nspacedim`), the number of particles (`npart`), and a pointer to an implementation of a `WaveFunction`. Notice that the constructor inform the constructor of `MCIObservableFunctionInterface` that the Monte Carlo integral will be performed in a `nspacedime` $\times$ `npart` space, and that there will be 4 observables, corresponding to the total, potential, kinetic, and Jackson-Feenberg kinetic energies, respectively. All the energies are computed as *energy unit per particle*;
- `getNSpaceDim`: Returns `nspacedim`;
- `getNPart`: Returns `getNPart`;
- `localPBKineticEnergy`: Returns the standard (Pandharipande-Bethe) local kinetic energy, in units of energy per particle. It requires the particle positions, provided through the array `in`;
- `localJFKineticEnergy`: Returns the Jackson-Feenberg local kinetic energy;
- `localPotentialEnergy`: Returns the local potential energy. Must be implemented by the user;
- `observableFunction`: Returns the 4 observables for the particle coordinates contained in the array `in`. The observables will be contained in the array `out`, and are the local total, potential, kinetic, and Jackson-Feenberg kinetic energies.

As example, we report the Hamiltonian for a one-dimensional, one-particle, harmonic oscillator:

$$H = -\frac{\partial^2 \psi}{\partial x^2} + \frac{1}{2}\omega^2 x^2 \tag{1}$$

```
class HarmonicOscialltor1D1P: public Hamiltonian
{
   protected:
      double _w;

   public:
      HarmonicOscialltor1D1P(const double w, WaveFunction * wf):
            Hamiltonian(1, 1, wf) {_w=w;}

      double localPotentialEnergy(const double *r)
      {
         return (0.5*_w*_w*(*r)*(*r));
      }
};
```

3 VMC

```
// #include "VMC.hpp"
class VMC{

public:
   VMC(WaveFunction * wf, Hamiltonian * H);
   ~VMC();


   // Monte Carlo Integral within VMC should be performed using
       the MCI object provided by VMC
   MCI * getMCI(){return _mci;}


   // Computation of the variational energy
   void computeVariationalEnergy(const long & Nmc, double * E,
       double * dE);


   // Wave Function Optimization Methods
   void conjugateGradientOptimization(const long &E_Nmc, const
       long &grad_E_Nmc);

   void stochasticReconfigurationOptimization(const long &Nmc);

   void simulatedAnnealingOptimization(const long &Nmc, const
       double &iota, const double &kappa, const double &lambda,
       gsl_siman_params_t &params);

};
```

The Variational Monte Carlo (VMC) method is a method that uses a minimisation method in order to find the variational parameters α of a trial wave function ψ_α such that the variational energy

$$E(\psi_\alpha) = \frac{\int dR\, \psi_\alpha(R) H \psi_\alpha(R)}{\int dR\, \psi_\alpha(R)\psi_\alpha(R)} \tag{2}$$

is minimised. In Eq. (2) H is the Hamiltonian, and we have assumed that both the Hamiltonian and the wave functions are real, as it is done by this library. The minimisation can be achieved by using several optimisation algorithms. At the end of the optimization process, the wave function will be set to have the optimal variational parameters

- `VMC`: The constructor. It requires a wave function, a Hamiltonian, and the number of sampling points to use for computing the energy (`ENmc`) and the energy gradient (`GNmc`) during the optimisation process. Notice that the constructor inform the constructor of `NoisyFunctionWithGradient` about the number of the number of variational parameters used by the wave function;

- `getMCI`: Returns a pointer to the `MCI` object used to compute the variational energy and the wave function derivatives;

- `computeVariationalEnergy`: Compute the total, kinetic, and potential energies for the given wave function. In fact `E` is expected to be an array of size 4, where `E[0]` is the total energy, `E[1]` is is the potential energy, `E[2]` is the Pandharipande-Bethe kinetic energy (i.e. the "real" kinetic energy), and `E[3]` is the Jackson-Feenberg kinetic energy;

- `conjugateGradientOptimization`: Optimise the trial wave function using the conjugate gradient method of the NoisyFunMin library. It requires the number of sampling points for computing the energy during the linear search, and the number of samplings to use for computing the energy gradient;

- `stochasticReconfigurationOptimization`: Optimise the trial wave function using the Dynamic Descent method of the NoisyFunMin library using as direction the one obtained using the Stochastic Reconfiguration (SR) approach. It requires the number of sampling points for computing the SR direction;

- `simulatedAnnealingOptimization`: Optimise the trial wave function using the Simulated Annealing algorithm contained in the GSL library (read here). The simulated annealing will try to minimise the target function

$$\iota E + \kappa \sigma_E + \lambda \frac{\sqrt{\sum_{i=1}^{N_\alpha} \alpha_i^2}}{N_\alpha}$$

where E is the energy, σ_E is the energy's standard deviation, and the last term represents a normalization factor, often used in Machine Learning. The weight of each term is controlled by the three parameters ι (`iota`), κ (`kappa`), and λ (`lambda`), which must be provided to the method. Furthermore the method requires `Nmc`, the number of sampling points for computing the target function integrals (E and σ_E), and the GSL siman library parameters.

Examples can be found in the folder `examples`.