

The VMC++ Library

Francesco Calcavecchia

March 16, 2016

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

```
1 // #import "WaveFunction.hpp"
2 class WaveFunction: public MCISamplingFunctionInterface
3 {
4     public:
5         // Constructor and destructor
6         WaveFunction(const int &nspacedim, const int &npart,
7                     const int &ncomp, const int &nvp);
8         virtual ~WaveFunction();
9
10        // Getters
11        int getNSpaceDim();
12        int getNPart();
13        int getNVP();
14
15        // Manage the variational parameters
16        virtual void setVP(const double *vp) = 0; // TO DO
17        virtual void getVP(double *vp) = 0; // TO DO
```

```

18
19 // // Heritage from MCISamplingFunctionInterface,
20 // // which must be implemented
21 // void samplingFunction(const double *in,
22 //     double *out) = 0; // TO DO
23 // double getAcceptance() = 0; // TO DO
24
25 // Derivatives of the trial wave function
26 virtual double d1(const int &i,
27     const double *in) = 0; // TO DO
28 virtual double d2(const int &i, const int &j,
29     const double *in) = 0; // TO DO
30 virtual double vd1(const int &i,
31     const double *in) = 0; // TO DO
32
33 }

```

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++ class **MCISamplingFunctionInterface**. Please refer to the user manual of MCI++ to see the details of this class. The user must be able to correctly implement its 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**);
- **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;
- **d1**: Returns the value of $\frac{\partial \psi(x)}{\partial x_i}$, computed in x equal to the array given in **in**. Here ψ is the wave function, and **i** the coordinate's index. For example, in a system with **npart=2** and **nspacedim=3**, x_0, x_1, x_2 are the x, y, z coordinates of the first particle, and x_3, x_4, x_5 the coordinates of the second particle. Must be implemented by the user;

- d2: Returns the value of $\frac{\partial^2 \psi(x)}{\partial x_i \partial x_j}$. Must be implemented by the user;
- vd1: Returns the value of $\frac{\partial \psi(x)}{\partial \alpha_i}$, where α_i is the i -th variational parameter of ψ . Must be implemented by the user.

As example, we report the implementation of a trial wave function for 1 particle in a one-dimensional space, that uses 2 variational parameters, and has only one component. The functional form is:

$$\psi(x) = e^{-b(x-a)^2} \quad (1)$$

where a and b are the two variational parameters.

```

1 class QuadrExponential1D1POrbital: public WaveFunction
2 {
3     protected:
4         double _a, _b;
5
6     public:
7         QuadrExponential1D1POrbital(const double a, const double b
8             ): WaveFunction(1,1,1,2) {_a=a; _b=b;}
9
10        void setVP(const double *in)
11        {
12            _a=in[0];
13            // if (_a<0.01) _a=0.01;
14            _b=in[1];
15            // if (_b<0.01) _b=0.01;
16            using namespace std;
17            //cout << "change a and b! " << _a << " " << _b <<
18                endl;
19        }
20        void getVP(double *out)
21        {
22            out[0]=_a; out[1]=_b;
23        }
24
25        void samplingFunction(const double *in, double *out)
26        {
27            *out = -2.*(_b*(in[0]-_a)*(in[0]-_a));
28        }
29
30        double getAcceptance()
31        {
32            return exp(getProtoNew(0)-getProtoOld(0));
33        }
34
35        double d1(const int &i, const double *in)
36        {
37            return (-2.*_b*(in[0]-_a));
38        }
39
40        double d2(const int &i, const int &j, const double *in)

```

```

39     {
40         return ( -2.*_b + (-2.*_b*(in[0]-_a))*(-2.*_b*(in[0]-_a
41             )) ) ;
42     }
43
44     double vdl(const int &i, const double *in)
45     {
46         if (i==0)
47         {
48             return (2.*_b*(in[0]-_a));
49         } else if (i==1)
50         {
51             return (- (in[0]-_a)*(in[0]-_a));
52         } else
53         {
54             using namespace std;
55             cout << "ERROR vdl QuadrExponential! " << endl;
56             return 0.;
57         }
58     };

```

2 Hamiltonian

```

1 // import "Hamiltonian.hpp"
2 class Hamiltonian: public MCIObservableFunctionInterface
3 {
4     public:
5         // Constructor and destructor
6         Hamiltonian(const int &nspacedim, const int &npart,
7             WaveFunction * wf):
8             MCIObservableFunctionInterface(nspacedim*npart,4)
9             virtual ~Hamiltonian() {}
10
11         // Getters
12         int getNSpaceDim();
13         int getNPart();
14
15         // Kinetic energy
16         double localPBKineticEnergy(const double *in);
17         double localJFKineticEnergy(const double *in);
18
19         // Potential energy
20         virtual double localPotentialEnergy(const double *in) = 0;
21         // TO DO
22
23         // Heritage from MCIObservableFunctionInterface
24         // already implemented
25         void observableFunction(const double * in, double *out)
26 };

```

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 `nspacedim` \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:

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

```

1 class HarmonicOscialltor1D1P: public Hamiltonian
2 {
3     protected:
4         double _w;
5
6     public:
7         HarmonicOscialltor1D1P(const double w, WaveFunction * wf):
8             Hamiltonian(1, 1, wf) {_w=w;}
9
10        double localPotentialEnergy(const double *r)
11        {
12            return (0.5*_w*_w*(*r)*(*r));
13        }
14 };

```

3 VMC

```

1 // #include "VMC.hpp"
2 class VMC: public NoisyFunctionWithGradient
3 {
4     public:
5         VMC(WaveFunction * wf, Hamiltonian * H,
6             const int &ENmc, const int &GNmc):
7             NoisyFunctionWithGradient(wf->getNVP());
8         ~VMC();
9
10        // Getters
11        MCI * getEnergyMCI();
12
13        // Conjugate Gradient minimization
14        void conjugateGradientOptimization();
15
16        // Computation of the energy
17        void computeEnergy(const long & Nmc,
18                           double * E, double * dE);
19
20        // Computation of the energy gradient
21        void computeEnergyGradient(const long & Nmc,
22                                   double * gradE, double * dgradE);
23
24 };

```

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

$$E(\psi_\alpha) = \frac{\int dR \psi_\alpha(R) H \psi_\alpha(R)}{\int dR \psi_\alpha(R) \psi_\alpha(R)} \quad (3)$$

is minimised. In Eq. (??) 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 is done by using a conjugate gradient algorithm.

- **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;
- **getEnergyMCI**: Returns a pointer to the **MCIntegrator** used to compute the variational energy;
- **conjugateGradientOptimization**: Perform the optimisation of the wave function. At the end of the process, the wave function will be set to have the optimal variational parameters;

- **computeEnergy**: Compute the total, kinetic, and potential energies for the given wave function;
- **computeEnergyGradient**: Compute the gradient of the energy (in respect to the variational parameters).

As example, we will demonstrate how to use the wave function and Hamiltonian of the examples in the previous sections for performing a VMC simulation, with some prints to demonstrate its effectiveness:

```

1 int main() {
2     const long NMC = 100001;
3
4     QuadrExponential1D1POrbital * qexp = new
        QuadrExponential1D1POrbital(1.0, 1.1);
5     HarmonicOscillator1D1P * harm_osc = new
        HarmonicOscillator1D1P(1., qexp);
6
7     double * a = new double[2];
8     qexp->getVP(a);
9     cout << "Wave Function a      = " << a[0] << endl;
10    cout << "Wave Function b      = " << a[1] << endl;
11
12    double * energy = new double[4];
13    double * d_energy = new double[4];
14    vmc2->computeEnergy(NMC, energy, d_energy);
15    cout << "Total Energy          = " << energy[0] << " +- " <<
        d_energy[0] << endl;
16    cout << "Potential Energy       = " << energy[1] << " +- " <<
        d_energy[1] << endl;
17    cout << "Kinetic (PB) Energy = " << energy[2] << " +- " <<
        d_energy[2] << endl;
18    cout << "Kinetic (JF) Energy = " << energy[3] << " +- " <<
        d_energy[3] << endl << endl;
19
20    VMC * vmc = new VMC(qexp, harm_osc, NMC, 101*NMC);
21    vmc->conjugateGradientOptimization();
22
23    qexp->getVP(a);
24    cout << "Wave Function a      = " << a[0] << endl;
25    cout << "Wave Function b      = " << a[1] << endl;
26
27    vmc2->computeEnergy(NMC, energy, d_energy);
28    cout << "Total Energy          = " << energy[0] << " +- " <<
        d_energy[0] << endl;
29    cout << "Potential Energy       = " << energy[1] << " +- " <<
        d_energy[1] << endl;
30    cout << "Kinetic (PB) Energy = " << energy[2] << " +- " <<
        d_energy[2] << endl;
31    cout << "Kinetic (JF) Energy = " << energy[3] << " +- " <<
        d_energy[3] << endl << endl;
32
33    delete [] a;

```

```
34     delete [] energy;
35     delete [] d_energy;
36     delete vmc;
37     delete harm_osc;
38     delete qexp;
39 }
```