The VMC++ Library

Francesco Calcavecchia

February 1, 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"
class WaveFunction: public MCISamplingFunctionInterface
{
   public:
      // Constructor and destructor
      WaveFunction(const int &nspacedim, const int &npart,
            const int &ncomp, const int &nvp);
      virtual ~WaveFunction();

      // Getters
      int getNSpaceDim();
      int getNPart();
      int getNVP();

      // Manage the variational parameters
      virtual void setVP(const double *vp) = 0;  // TO DO
      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,
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 $\frac{\partial^2 \psi(x)}{\partial x_i^2}$. 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:

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

where a and b are the two variational parameters.

```
class QuadrExponential1D1POrbital: public WaveFunction
{
   protected:
      double _a, _b;

   public:
      QuadrExponential1D1POrbital(const double a, const double b
          ): WaveFunction(1,1,1,2) {_a=a; _b=b;}

      void setVP(const double *in)
      {
         _a=in[0];
         //if (_a<0.01) _a=0.01;
         _b=in[1];
         //if (_b<0.01) _b=0.01;
         using namespace std;
         //cout << "change a and b! " << _a << "    " << _b <<
             endl;
      }
      void getVP(double *out)
      {
         out[0]=_a; out[1]=_b;
      }

      void samplingFunction(const double *in, double *out)
      {
         *out = -2.*(_b*(in[0]-_a)*(in[0]-_a));
      }

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

      double d1(const int &i, const double *in)
      {
         return (-2.*_b*(in[0]-_a) );
      }

      double d2(const int &i, const double *in)
```

```
39          {
40              return ( -2.*_b + (-2.*_b*(in[0]-_a))*(-2.*_b*(in[0]-_a
                    )) ) ;
41          }
42
43          double vd1(const int &i, const double *in)
44          {
45              if (i==0)
46              {
47                  return (2.*_b*(in[0]-_a));
48              } else if (i==1)
49              {
50                  return (-(in[0]-_a)*(in[0]-_a));
51              } else
52              {
53                  using namespace std;
54                  cout << "ERROR vd1 QuadrExponential! " << endl;
55                  return 0.;
56              }
57          }
58  };
```

1.1 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 input. The FFNN must support first, second, and variational derivatives.

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

```
1  class FFNNWaveFunction: public WaveFunction{
2
3  private:
4      FeedForwardNeuralNetwork * _ffnn;
5
6  public:
7      // --- Constructor
8      // IMPORTANT: The provided ffnn should be ready to use (
           connected) and have the first, second and variational
           derivatives substrates
9      FFNNWaveFunction(const int &nspacedim, const int &npart,
           FeedForwardNeuralNetwork * ffnn);
10
11
12     // --- Getters
13     FeedForwardNeuralNetwork * getFFNN();
14
15 };
```

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(){}

      // Getters
      int getNSpaceDim();
      int getNPart();

      // Kinetic energy
      double localPBKineticEnergy(const double *in);
      double localJFKineticEnergy(const double *in);

      // Potential energy
      virtual double localPotentialEnergy(const double *in) = 0;
            // TO DO

      // Heritage from MCIObservableFunctionInterface
      // already implemented
      void observableFunction(const double * in, double *out)
};
```

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:

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

```
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);

```

```
16
17      // Wave Function Optimization Methods
18      void conjugateGradientOptimization(const long &E_Nmc, const
            long &grad_E_Nmc);
19
20      void stochasticReconfigurationOptimization(const long &Nmc);
21
22      void simulatedAnnealingOptimization(const long &Nmc, const
            double &iota, const double &kappa, const double &lambda,
            gsl_siman_params_t &params);
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

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

is minimised. In Eq. (3) 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`.