Implementing Quantum Monte Carlo using Python

Jon Kristian Nilsen

March 28, 2006

1 Introduction

Motivation: We have a serial code written in C++. The code is object oriented and uses *diffusion Monte Carlo* (DMC) to find the ground state energy (and some excited states) of a system given the wave function of the system, i.e. in an ab-initio way. The goal is then to port parts of the code to Python to take advantage of the flexibility and simplicity of Python and ease the process of parallellizing of the code.
We first present the basics and the algorithm of Diffusion Monte Carlo (DMC) in section 2. Then we show how to implement DMC in Python in section 3.

2 DMC

In DMC we seek to solve the Schrödinger equation in imaginary time. This involves Monte Carlo integration of a Green function where the walkers are guided by the Metropolis algorithm. As the Green function is approximated by splitting it up in a diffusional part (which has the form of a Gaussian) and a branching part we also need a Gaussian random generator and a way to create and destroy walkers.

2.1 Basic Ideas of DMC

The basic ingredients of DMC are (Guardiola 1997):

1. It considers the Schrödinger equation in imaginary time,

   $$-\frac{\partial d\psi(\mathbf{R},t)}{\partial t} = [H-E]\psi(\mathbf{R},t), \tag{1}$$

   where $\mathbf{R}$ represents the set of all coordinates. The formal solution of (1) is

   $$\psi(\mathbf{R},t) = e^{-[H-E]t}\psi(\mathbf{R},0), \tag{2}$$

   where $\exp[-(H-E)t]$ is called the *Green function*, and E is a convinient energy shift.

2. The wave function is positive definite everywhere, as it happens with the ground state of a bosonic system, so it may be considered as a probability distribution function.

3. The wave function is represented by a set of random vectors $\{R_1, R_2, \ldots, R_M\}$, in such a form that the time evolution of the wave function is actually represented by the evolution of the set of walkers.

4. The actual computation of the time evolution is done in small time steps τ, and the Green function is accordingly approximated,

$$e^{-[H-E]t} = \prod_{i=1}^{n} e^{-[H-E]\tau}, \tag{3}$$

   where $\tau = t/n$.

5. The imaginary time evolution of an arbitrary starting state $\psi(\mathbf{R}, 0)$, once expanded in the basis of stationary states of the Hamilton operator

$$\psi(\mathbf{R}, 0) = \sum_{\nu} C_{\nu}\phi_{n}u(\mathbf{R}) \tag{4}$$

   is given by

$$\psi(\mathbf{R}, t) = \sum_{\nu} e^{-[E_{\nu}-E]t} C_{\nu}\phi_{\nu}(\mathbf{R}), \tag{5}$$

   in such a way that the lowest energy components will have the largest amplitudes after a long elapsed time, and in the $t \to \infty$ limit the most important amplitude will correspond to the ground state (if $C_0 \neq 0$)[1].

6. An important improvement of this scheme is the introduction of *importance sampling*.

The scheme is quite simple; once we have found an appropriate approximation for the short-time Green function and determined a starting state, the job consists in representing the starting state by a collection of walkers and letting them evolve in time, i.e., obtaining a collection of walkers from the old collection of walkers, up to a time large enough so that all other states than the ground state are negligible.

[1]This can easily be seen by replacing E with the ground state energy E_0 in eq. (5). As E_0 is the lowest energy, we will get $C_0\phi_0 + \sum_{\nu} \exp[-(E_{\nu} - E_0)t]\phi_{\nu} \underset{t\to\infty}{=} C_0\phi_0$.

2.2 DMC Algorithm

Algorithm 1: Algorithm

```
Generate an initial set of random walkers
for 0 to time
    for 0 to $N_{walkers}$
      Diffusion:
      for 0 to particles
        propose move $\mathbf{r}' = \mathbf{r} + D\tau\mathbf{F}(\mathbf{r}) + \xi$
        Accept or reject from $min(1, \left|\frac{\psi_T(\mathbf{R}')}{\psi_T(\mathbf{R})}\right|^2 \frac{G(\mathbf{R},\mathbf{R}';\tau)}{G(\mathbf{R}',\mathbf{R};\tau)})$
      Branching: calculate replication factor $n = int(\exp\{\tau(E_L(\mathbf{R})/2 + E_L(\mathbf{R}')/2 - E)\})$
      if $n = 0$
        Kill the walker
      if $n > 0$
        Allow the walker to make $n - 1$ clones
      Remove dead walkers, and make new clones
      Check walker population and adjust trial energy
      sample contributions to observable
```

2.3 C++ implementation

Here we present the C++ implementation of DMC. Presented in figs. 1 and 2 are, in order, the class diagram and float diagram of the implementation.
In fig. 1 we see that we have three classes[2].
The class DMC contains the DMC algorithm, implemented in the function diffMC() (and helper functions to clean up the code). It also contains a pointer to the class Func and an array of walker objects (or just walkers).
The class Func contains functors for different wave functions (with corresponding analytic local energies and quantum forces if implemented) along with generic functions for the gradient and Laplace operator.
The class walker contains all the physical information of a walker, that is, its position in phase space (and function for setting and getting the position) and functions for getting physical values like the energy of the walker and the wave function of the walker.
Fig. 2 shows the float of the DMC program. The algorithm is divided into functions so that, e.g., the function diffMC() contains a loop calling the function oneTimeStep(), which in turn loops over oneMonteCarloStep() and so on. Each such function is represented by a box in the float diagrams.
Looking at the float diagram, fig. 2, it is easy to realize that most of time of computation is spent in the bottom boxes of the diagram. When implementing the DMC in Python the bottom boxes should be kept in C++ while only diffMC() (which is in broad lines the hole DMC algorithm) will be in Python code.

[2]at least when I get time to update the figure to the current implementation where the class Particle is removed. Just hold your hand over the Particle-box to get the current class diagram.

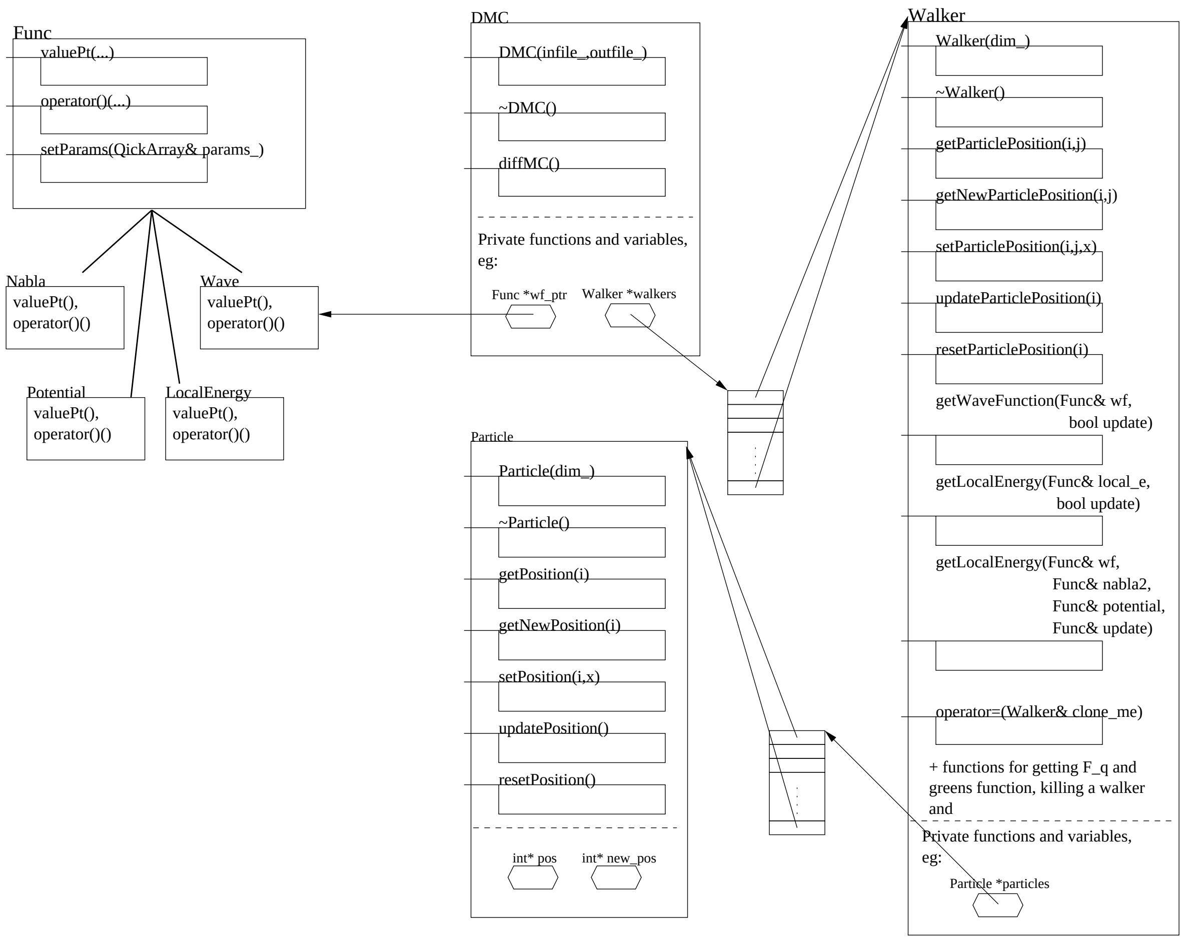

Figure 1: Class diagram of DMC

3 pyDMC

3.1 Python implementation

The C++ implementation uses about 90% in the walker objects and most of this time in computing local energies (N^3 operations where N is the number of particles) in functions located in Func. In the python implementation of diffusion Monte Carlo (pyDMC) the classes Walker and Func is therefor swiged and put into a module DMC_cxx, together with the functions from the DMC class below oneTimeStep() in fig. 2.
The main obstacle in implementing pyDMC is the handling of the walkers. In a straight forward approach we put the walker objects in a native Python array. This is a very tempting approach; we can leave the entire problem of creating and killing walkers[3] to Python. However, Python arrays have a serious efficiency problem, specially when we move to the parallellization of the program.
When thinking performance of arrays in Python the add-on package *Numerical Python* (NumPy) springs to mind as an obvious choice. The fact that the module pypar (which we are going to use in the parallellization) suports sending NumPy arrays directly, is of course helping in that choice.

[3]which is not a straight forward problem with static arrays in C++

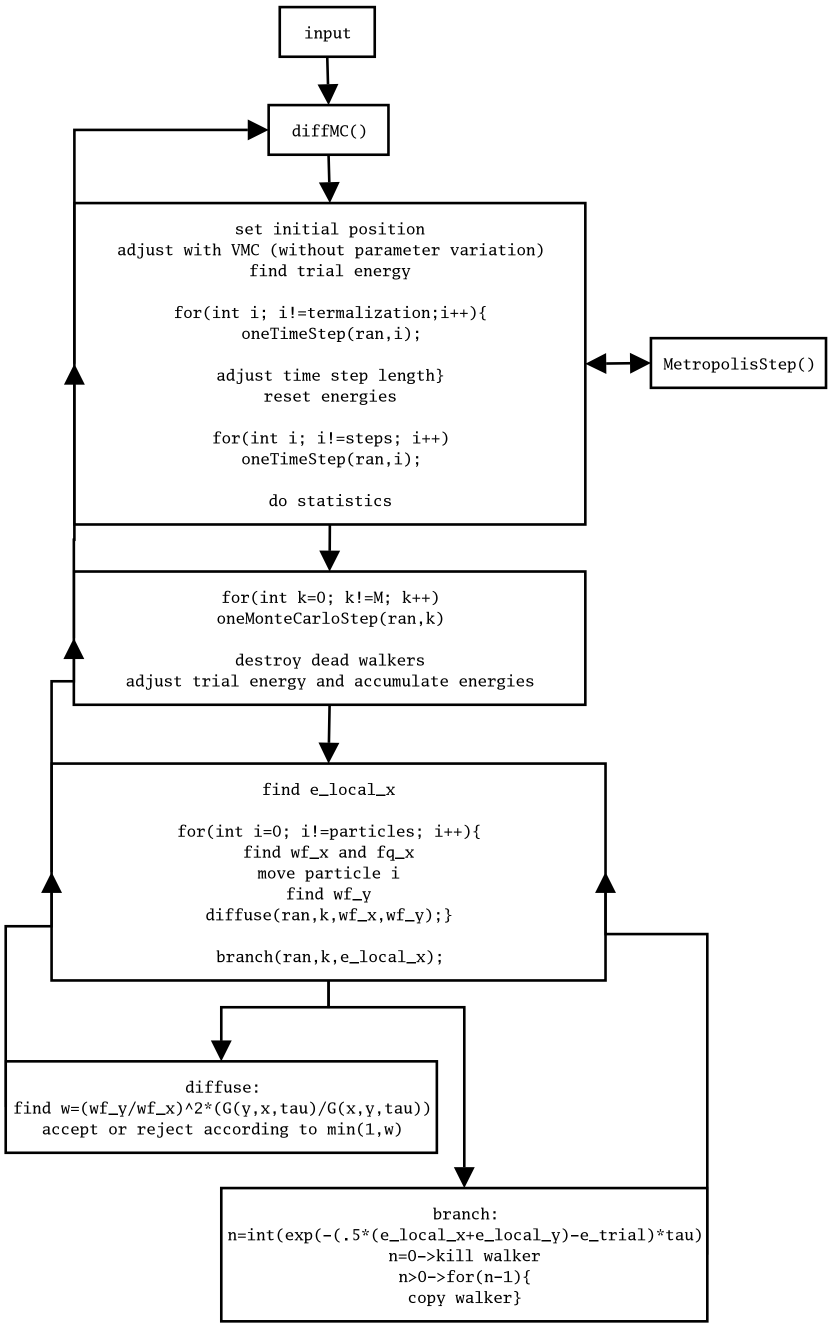

Figure 2: Float diagram of DMC

However, even though NumPy supports a lot of types, (such as integers, floats, chars etc.) there is no support for walkers as a type[4]. It is possible to use generic python objects, but this will mainly make NumPy array comparable to native arrays.
Another solution is to implement an array class (lets call it WalkerArray) in Python which contains a wraps around a C++ class containing a C++ array of walkers and functionality to create and kill walkers. Even though this is a good approach in a serial implementation, we still have to convert these arrays to NumPy arrays to be able to send the walkers in MPI.
A third (and hopefully last) approach is to keep WalkerArray, but store all the walker data in a NumPy array. The C++ Walker class then has to be edited so that it only uses pointers to an array for all arrays and variables that should be stored. This array may then be provided by NumPy. Even though this approach taints the C++ implementation, we get the advantage that the Python DMC class only has to care about NumPy arrays.

3.2 Parallellizing pyDMC

In the native array approach most of the parallellization is realized with the functions spread_walkers() and gather_walkers():

```
def spread_walkers(self):
    """Function converting walkers to numpy arrays, sending,
    recieving and converting back"""
    if self.master:
        displace = self.loc_walkers[self.master_rank]
        for i in range(1,self.numproc):
            send_w = self.w[displace:displace+self.loc_walkers[i]]
            send_buff = walkers2py(send_w)
            self.pypar.send(send_buff,i)
            displace += self.loc_walkers[i]
    else:
        recv_buff = self.pypar.receive(self.master_rank)
        w_args = [self.loc_walkers[self.myrank], self.particles,\
                  self.dimensions]
        self.w_block = py2walkers(recv_buff, *w_args)


def gather_walkers(self):
    """Look at spread_walkers in a mirror"""
    if self.master:
        displace = self.loc_walkers[self.master_rank]
        for i in range(1,self.numproc):
            recv_buff = self.pypar.receive(i)
            w_args = [self.loc_walkers[i], self.particles,\
                      self.dimensions]
            self.w[displace:displace+self.loc_walkers[i]] = py2walkers(\
                recv_buff, *w_args)
            displace += self.loc_walkers[i]
    else:
        send_buff = walkers2py(self.w_block)
        self.pypar.send(send_buff,self.master_rank)
```

The functions walkers2py() and py2walkers() are functions for converting a walker to a NumPy array and back again.
The walker array is realized by the function warray():

```
def warray(self,size,particles,dim):
    """function returning an array of (initialized) walkers"""
    w = []
    for i in range(size):
        w += [Walker()]
        w[i].pyInitialize(particles,dim)
    return w
```

An example of a parallellized diffusion monte carlo program is then realized in less than 100 lines:

```
>>> import pypar,math
Pypar (version 1.9.1) initialised MPI OK with 1 processors
>>> from DMC import DMC
>>>
>>> d = DMC(pypar)
```

[4]To the best of my knowledge

```
>>> # 1 particle and alpha=0.5 yields a harmonic oscillator with
>>> # energy == 3/2:
>>> d.params[0] = 0.5
>>> d.reset_params()
>>> d.silent = True # to avoid too much noise
>>>
>>> def timestep(i_step):
...     M = d.no_of_walkers
...     d.spread_walkers()
...     for walker in d.w_block:
...         d.monte_carlo_step(walker)
...     d.gather_walkers()
...     d.update = False
...     if d.master:
...         #bring out your dead
...         for i in range(M-1,-1,-1):
...             if d.w[i].isDead():
...                 d.w[i:i+1] = [] #removing walker
...             else:
...                 while d.w[i].tooAlive():
...                     baby_walker = d.copy_walker(d.w[i])
...                     baby_walker.calmWalker()
...                     d.w += [baby_walker]
...                     d.w[i].madeWalker()
...         d.no_of_walkers = len(d.w)
...     d.no_of_walkers = d.pypar.broadcast(d.no_of_walkers,
...                                         d.master_rank)
...     d.refresh_w_blocks()
...     d.spread_walkers()
...     d.num_args[-1] = d.update
...     if d.master:
...         nrg = 0.; pot_nrg = 0.; vort_nrg = 0
...         d.time_step_counter += 1
...         for walker in d.w:
...             d.num_args[-1] = d.update
...             nrg       += walker.getLocalEnergy(*d.num_args)
...             pot_nrg   += walker.getLocalEnergy(d.pot_e)
...             if hasattr(d,'vort_e'):
...                 vort_nrg += walker.getLocalEnergy(vort_e)
...         nrg      /= float(d.no_of_walkers*d.particles*d.scale)
...         pot_nrg  /= float(d.no_of_walkers*d.particles*d.scale)
...         vort_nrg /= float(d.no_of_walkers*d.particles*d.scale)
...         d.energy  += nrg
...         d.energy2 += nrg*nrg
...         if d.time_step_counter >= d.termalization:
...             #implement pos_hist to be called here
...             d.observables[i_step,0] = nrg
...             d.observables[i_step,1] = pot_nrg
...             d.observables[i_step,2] = vort_nrg
...         # adjust trial energy (and no. of walkers)
...         nrg = -.5*math.log(float(d.no_of_walkers)/float(M))/d.tau
...         d.e_trial += nrg
...     d.e_trial = d.pypar.broadcast(d.e_trial,d.master_rank)
...     d.no_of_walkers = d.pypar.broadcast(d.no_of_walkers,
...                                         d.master_rank)
...
>>> #set initial walker positions:
>>> if d.metropolis_termalization: d.uni_dist()
...
>>> for i in range(d.metropolis_termalization):
...     for walker in d.w_block:
...         d.metropolis_step(walker)
...
>>>
>>> nrg = 0.
>>>
>>> for walker in d.w_block:
...     nrg += walker.getLocalEnergy(*d.num_args)
...
>>> d.gather_walkers()
>>> d.e_trial = d.all_reduce(nrg)
>>> d.time_step_counter = 0
>>> for i in range(d.termalization):
...     timestep(i)
...
>>> d.energy = 0; d.energy2 = 0
>>> for i in range(d.steps):
...     timestep(i)
...
>>> if d.master:
...     d.energy  /= float(d.steps)
...     d.energy2 /= float(d.steps)
...     d.energy2 -= d.energy*d.energy
...     print "energy = %g +/- %g"%(d.energy,
...                           math.sqrt(d.energy2/float(d.steps)))
...     print "sigma = %g"%d.energy2
...
energy = 1.5 +/- 0
sigma = 0
>>> pypar.Finalize()
```

4 Results

Next I should put in some results...

References

Guardiola, R. (1997), Monte carlo methods in quantum many-body theories, *in* J. Navarro & A. Polls, eds, ‘Microscopic Quantum Many-Body Theories and Their Applications’, Lecture Notes in Physics, Springer Verlag, pp. 269–336.