Available online at www.sciencedirect.com

ScienceDirect

Computer Physics Communications 177 (2007) 799–814

Computer Physics Communications

www.elsevier.com/locate/cpc

MontePython: Implementing Quantum Monte Carlo using Python ☆

Jon Kristian Nilsen

[a] *USIT, Postboks 1059 Blindern, N-0316 Oslo, Norway*
[b] *Fysisk institutt, Postboks 1048 Blindern, N-0316 Oslo, Norway*

Received 30 August 2006; received in revised form 24 April 2007; accepted 16 June 2007

Available online 29 June 2007

Abstract

We present a cross-language C++/Python program for simulations of quantum mechanical systems with the use of Quantum Monte Carlo (QMC) methods. We describe a system for which to apply QMC, the algorithms of variational Monte Carlo and diffusion Monte Carlo and we describe how to implement theses methods in pure C++ and C++/Python. Furthermore we check the efficiency of the implementations in serial and parallel cases to show that the overhead using Python can be negligible.

Program summary

Program title: MontePython
Catalogue identifier: ADZP_v1_0
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/ADZP_v1_0.html
Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland
Licensing provisions: Standard CPC licence, http://cpc.cs.qub.ac.uk/licence/licence.html
No. of lines in distributed program, including test data, etc.: 49 519
No. of bytes in distributed program, including test data, etc.: 114 484
Distribution format: tar.gz
Programming language: C++, Python
Computer: PC, IBM RS6000/320, HP, ALPHA
Operating system: LINUX
Has the code been vectorised or parallelized?: Yes, parallelized with MPI
Number of processors used: 1–96
RAM: Depends on physical system to be simulated
Classification: 7.6; 16.1
Nature of problem: Investigating ab initio quantum mechanical systems, specifically Bose–Einstein condensation in dilute gases of ^{87}Rb
Solution method: Quantum Monte Carlo
Running time: 225 min with 20 particles (with 4800 walkers moved in 1750 time steps) on 1 AMD Opteron$^{\text{TM}}$ Processor 2218 processor; Production run for, e.g., 200 particles takes around 24 hours on 32 such processors.

PACS: 03.75.Hh; 03.75.Lm

Keywords: Python; C++; Quantum Monte Carlo; Bose–Einstein condensation; MPI

☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
E-mail address: j.k.nilsen@usit.uio.no.

0010-4655/$ – see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2007.06.013

1. Introduction

In scientific programming there has always been a struggle between computational efficiency and programming efficiency. On one hand, we want a program to go as fast as possible, resorting to low-level programming languages like FORTRAN77 and C which can be difficult to read and even harder to debug. On the other hand, we want the programming process to be as efficient as possible, turning to high-level software like Matlab, Octave, Maple, R and S+. Here features like clean syntax, interactive command execution, integrated simulation and visualization and rich documentation make us feel more productive. However, if we have some well tested and fast routines written in low-level language, interfacing these routines with, e.g., Matlab is rather cumbersome. Most often, we will end up using similar Matlab routines, which are often written as generally as possible at the cost of computing efficiency.

Recently, the programming language Python [1] has emerged as a potential competitor to Matlab. Python is a very powerful programming language which, when extended with numerical and visual modules like SciPy [2], shares many of the features of Matlab. In addition, Python was designed to be extendible with compiled code for efficiency and several tools are available for doing so.

In this paper we will demonstrate how Python can be extended with compiled code to yield an efficient scientific program. Specifically, we will start with a Monte Carlo simulator written in C++ and, with the help of SWIG [3], reuse the C++ code in a Python Monte Carlo simulator. We will show that this porting from low-level to high-level code can be achieved without significant loss of efficiency.

The remainder of this paper is organized as follows. In Section 2 we define the system we apply the Monte Carlo simulator to. Section 3 discuss briefly the diffusion Monte Carlo algorithm. Next, we go through the implementations of diffusion Monte Carlo, both in C++ and Python, in Section 4. Furthermore, Sections 5 and 6 compare the efficiency of C++ and Python for varying numbers of CPUs and Section 7 visualize the output from diffusion Monte Carlo with the use of Python. Finally, we round off with some remarks in Section 8.

2. The system

Quantum Monte Carlo (QMC) has a wide range of applications, for example studies of Bose–Einstein condensates of dilute atomic gases (bosonic systems) [4] and studies of so-called quantum dots (fermionic systems) [5], electrons confined between layers in semi-conductors. In this paper we will focus on a model which is meant to reproduce the results from an experiment by Anderson et al. [6]. Anderson et al. cooled down 4×10^6 ^{87}Rb to temperatures in the order of 100 nK to observe Bose–Einstein condensation in the dilute gas. Our physical motivation in this paper is to model numerically this fascinating experiment. This should be done in an as general as possible way, so that we can expand our computations to systems not yet explored in experiments. We will in this section go through the steps needed to put the experiment into the framework of QMC.

In QMC the goal is to solve the Schrödinger equation

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{R},t) = H\Psi(\mathbf{R},t), \tag{1}$$

or rather the time independent version

$$H\Psi(\mathbf{R}) = E\Psi(\mathbf{R}). \tag{2}$$

Thus, to model the experiment above using Quantum Monte Carlo methods, all we need is a Hamiltonian and a trial wave function. The Hamiltonian for N trapped interacting atoms is given by

$$H = -\frac{\hbar^2}{2m}\sum_{i=1}^{N}\nabla_i^2 + \sum_{i=1}^{N} V_{\text{ext}}(\mathbf{r}_i) + \sum_{i<j}^{N} V_{\text{int}}\big(|\mathbf{r}_i - \mathbf{r}_j|\big). \tag{3}$$

Taking advantage of the fact that the gas is dilute, we can describe the two-body interaction $V_{\text{int}}(|\mathbf{r}_i - \mathbf{r}_j|)$ by a hard-core potential of radius a, where a is the scattering length, thus treating the atoms as hard spheres [7].

We define the trial wave function by

$$\Psi_T = \prod_i g(\mathbf{r}_i)\prod_{i<j} f(r_{ij}), \tag{4}$$

where $g(\mathbf{r}_i)$ describes the interaction between one particle and the external potential, V_{ext}, while the two-body correlation function $f(r_{ij})$ describes the interaction between two particles. The function $f(r_{ij})$ is the solution of the Schrödinger equation for a pair of atoms at very low energy interacting via a hard-core potential of radius a. The ansatz for $f(r)$ reads

$$f(r) = \begin{cases} (1 - a/r) & r > a, \\ 0 & r \leqslant a. \end{cases} \tag{5}$$

Besides being physically motivated, this type of correlation has been successfully used in Refs. [8] and [4] to study both spherically symmetric and deformed traps. In the experiment of Anderson et al., the particles were trapped in a disk-shaped harmonic oscillator potential. This corresponds to using an external potential

$$V_{\text{ext}} = \frac{m}{2}\big(\omega_\perp x^2 + \omega_\perp y^2 + \omega_z z^2\big), \tag{6}$$

If we neglect the particle–particle interaction and insert the potential of Eq. (6) into Eq. (2) we obtain

$$g(\mathbf{r}) = A(\alpha)\lambda^{1/4}\exp\big(-\alpha\big(x^2 + y^2 + \lambda z^2\big)\big), \tag{7}$$

where α is taken as the variational parameter of the calculation and $A(\alpha) = (2\alpha/\pi)^{3/4}$ is a normalization constant. The parameter $\lambda = \omega_z/\omega_\perp$ is kept constant and set equal to the asymmetry of the trap. Still following Anderson et al. we let $\lambda = \sqrt{8}$ throughout this paper.

3. The Diffusion Monte Carlo algorithm

In this section we will go through the Diffusion Monte Carlo algorithm used in this paper. Diffusion Monte Carlo is built on Monte Carlo integration and the Metropolis algorithm [9] and usually needs input from a Variational Monte Carlo algorithm. The interested reader may find more information on these algorithms in, e.g., [10].

In Diffusion Monte Carlo (DMC) we seek to solve the Schrödinger equation in imaginary time. This involves Monte Carlo integration of a Green's function. As the Green's 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.

3.1. Basic ideas of DMC

The basic ingredients of DMC are [11]:

(1) It considers the Schrödinger equation in imaginary time,

$$-\frac{\partial\psi(\mathbf{R},t)}{\partial t} = [H - E]\psi(\mathbf{R},t), \tag{8}$$

where $\mathbf{R}$ represents the set of all coordinates. The formal solution of (8) is

$$\psi(\mathbf{R},t) = e^{-[H-E]t}\psi(\mathbf{R},0), \tag{9}$$

where $\exp[-(H-E)t]$ is called the *Green's function*, and E is a convenient 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. (This assumption leads to difficulties when we consider fermionic systems, where the wave functions are anti-symmetric and special care needs to be made.)

(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's function is approximated accordingly,

$$e^{-[H-E]t} = \prod_{i=1}^{n} e^{-[H-E]\tau}, \tag{10}$$

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

$$\psi(\mathbf{R},0) = \sum_\nu C_\nu \phi_n u(\mathbf{R}) \tag{11}$$

is given by

$$\psi(\mathbf{R},t) = \sum_\nu e^{-[E_\nu - E]t} C_\nu \phi_\nu(\mathbf{R}), \tag{12}$$

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]

[1] This can easily be seen by replacing E with the ground state energy E_0 in Eq. (12). As E_0 is the lowest energy, we will get $\lim_{t\to\infty}\sum_\nu \exp[-(E_\nu - E_0)t]\phi_\nu = C_0\phi_0$.

(6) An 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's 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.

3.2. Importance sampling

An important improvement to the DMC scheme above is, as mentioned above, the use of *importance sampling*. In problems with singularities in the potential (e.g., the Coulomb potential) the Green's function $\exp[-(H-E)t]$ will reach unbounded values, leading to an unstable algorithm. Even without singularities the scheme above is inefficient. This is due to the fact that we have imposed no restrictions as to where the walkers will walk.

To impose such a restriction, we substitute our wave function $\psi(\mathbf{R},t)$ with a new quantity $f(\mathbf{R},t)=\psi_T(\mathbf{R})\psi(\mathbf{R},t)$ where $\psi_T(\mathbf{R})$ is a time-independent trial wave function, which should be as close as possible to the true ground state. This substitution can be shown [10, p. 92] to lead to a transformed Hamilton operator which may be written as a sum of three terms $H=K+F+L$, where

$$K=-D\nabla^2, \qquad F=-D\big(\nabla\cdot\mathbf{F}(\mathbf{R})\big)+\mathbf{F}(\mathbf{R})\cdot\nabla, \qquad L=E_L(\mathbf{R}), \tag{13}$$

corresponding, respectively, to the kinetic part, the drift part and the local energy part.

An $\mathcal{O}(\tau^2)$ approximation of the Green's function is given by [12]:

$$\langle\mathbf{R}'|G|\mathbf{R}\rangle=\frac{1}{(4\pi D\tau)^{3N/2}}e^{-[\mathbf{R}'-\mathbf{R}-D\tau\mathbf{F}(\mathbf{R})]^2/4D\tau}e^{E\tau-[E_L(\mathbf{R}')+E_L(\mathbf{R})]\tau/2}+\mathcal{O}(\tau^2), \tag{14}$$

while an $\mathcal{O}(\tau^3)$ approximation the Green's function is obtained from [13]

$$G=e^{E\tau}e^{-L/2\tau}e^{-F/2\tau}e^{-K\tau}e^{-F/2\tau}e^{-L/2\tau}+\mathcal{O}(\tau^3). \tag{15}$$

3.3. DMC algorithm

In Algorithm 1 we state the DMC algorithm corresponding to Eq. (14). The algorithm corresponding to Eq. (15) is similar except that the move is split into four parts due to the splitting of the drift operator. ξ in the move part of Algorithm 1 is drawn from the multivariate Gaussian distribution with null mean and $\sigma=\sqrt{2D\tau}$, the solution of the kinetic Green's function.

4. The implementations

In the previous sections we have identified a physical system to simulate and found algorithms to use in the simulations. One important question that remains is how we implement the system and the algorithms. In this section we will propose three different approaches. They all use the same algorithms, they solve the same systems, with identical results, and they are all written in an object oriented way. In fact, most of the code is the same for all three approaches. The only difference in the implementations is the amount of time spent in low level, compiled language (represented by C++) versus time spent in high level, interpreted language

```
Generate an initial set of random walkers with the Metropolis algorithm
for 0 to time
  for 0 to $N_{\text{walkers}}$
    Diffusion:
      for 0 to particles
         propose move $\mathbf{r}'=\mathbf{r}+D\tau\mathbf{F}(\mathbf{r})+\xi$
      Branching; calculate replication factor:
      $n=\text{int}(\exp\{\tau(E_L(\mathbf{R})/2+E_L(\mathbf{R}')/2-E)\})$
      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
```

Algorithm 1. DMC algorithm

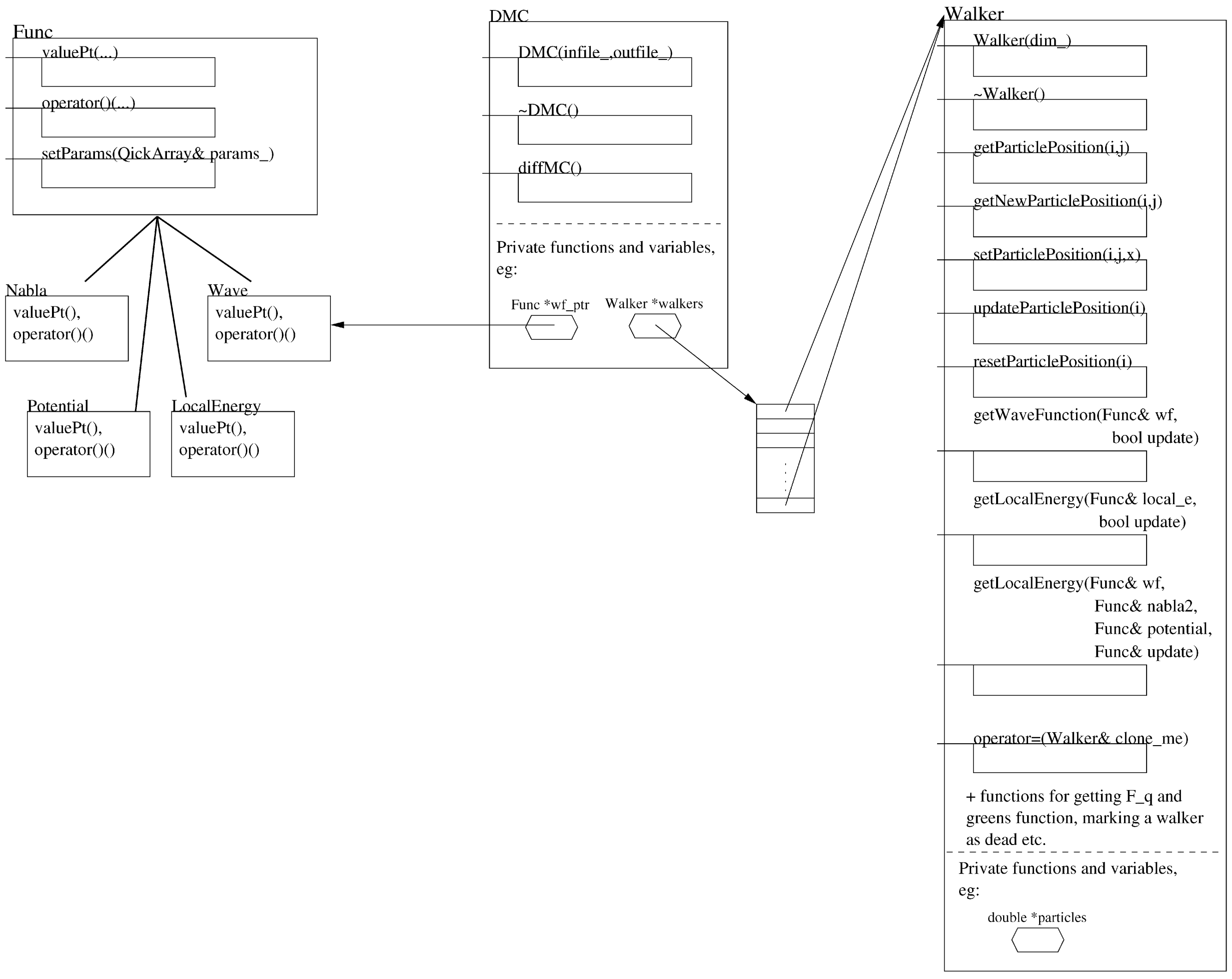

Func
valuePt(...)
operator()(...)
setParams(QickArray& params_)
Nabla
valuePt(),
operator()()
Wave
valuePt(),
operator()()
Potential
valuePt(),
operator()()
LocalEnergy
valuePt(),
operator()()
DMC
DMC(infile_,outfile_)
~DMC()
diffMC()
Private functions and variables, eg:
Func *wf_ptr
Walker *walkers
Walker
Walker(dim_)
~Walker()
getParticlePosition(i,j)
getNewParticlePosition(i,j)
setParticlePosition(i,j,x)
updateParticlePosition(i)
resetParticlePosition(i)
getWaveFunction(Func& wf, bool update)
getLocalEnergy(Func& local_e, bool update)
getLocalEnergy(Func& wf, Func& nabla2, Func& potential, Func& update)
operator=(Walker& clone_me)
+ functions for getting F_q and greens function, marking a walker as dead etc.
Private functions and variables, eg:
double *particles

Fig. 1. Class diagram of DMC.

(represented by Python). The assumption is that compiled code is faster while interpreted code is clearer, easier to debug and easier to expand. We will in this section go through a pure C++ implementation, a straight forward Python approach and a more involved Python approach.

4.1. C++ implementation

The base of our implementations is a serial diffusion Monte Carlo (DMC) solver written in C++. The Python solvers are both heavily based on this code. We will therefore first go through the C++ implementation of DMC. In Figs. 1 and 2 we present the class diagram and float diagram of the C++ implementation.

In Fig. 1 we show three classes, class DMC, class Func and class Walker.

- 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, i.e. classes whose only purpose is to receive a set of numerical values and transform these to numerical output (not unlike mathematical functions). Specifically, Func contains different wave functions (with corresponding analytic local energies and quantum forces if implemented) along with generic functions for the gradient and Laplace operator. The different functions of the systems are subclasses derived from general functions to ensure that the functions of all the systems have the same input and output.
- 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.

The advantage of this division of the program is quite clear. The class DMC contains the DMC algorithm and may easily be replaced with other Monte Carlo algorithms, like the already mentioned variational Monte Carlo, Green's Function Monte Carlo and

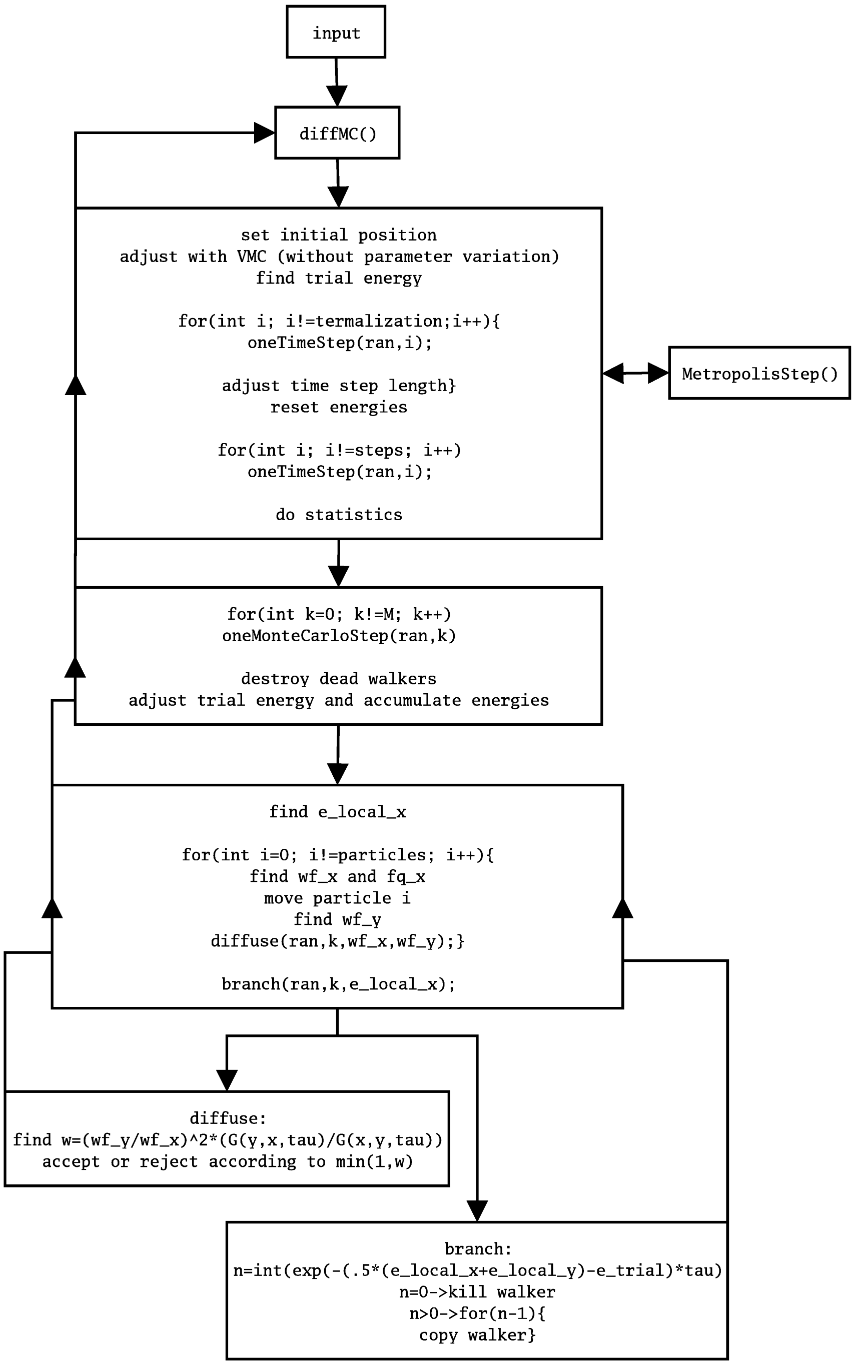


Fig. 2. Float diagram of DMC.

so on. These replacements will neither affect the systems implemented in class Func nor the physical information of the walkers. Likewise, new systems may be implemented without changing the code of the algorithm.[2] The wave function and the potential (or optionally an analytic expression of the local energy and the quantum force) are sent to the walkers as pointers to Func objects and are as such not known to the walkers at compile time.

Fig. 2 shows the float diagram 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.

[2] Except, of course, that the DMC class has to know that the new system exists.

Looking at the float diagram, Fig. 2, it is easy to realize that most of the 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.

4.1.1. Checkpointing

One aspect which is frequently forgotten when writing a scientific program is that of checkpointing. A Monte Carlo simulation may easily take several days, or even weeks and months. This would be infeasible without some way to stop and start the simulation in case of computer crashes, power losses or overeager computer managers. In checkpointing we store all information needed to resume the computation at given steps of the simulation. The challenge is to identify the steps at which the checkpointing should be made. The checkpoints should be made frequently enough to save time compared to starting all over, but not so frequently that the simulation is significantly slowed down.

In variational Monte Carlo it suffices to write a new initialization file where the number of steps is reduced to what is remaining of the original number of steps and a random seed so that we continue the random stream we have started. The latter is important if we want to get reproducible results. As the amount of data stored in a checkpoint is so small, we can do it after every step without reducing speed. However, making a checkpoint during the movement of the particles would be quite cumbersome and the amount of data needed to store the checkpoint would increase dramatically.

Again, diffusion Monte Carlo is more challenging. As generating a starting state in effect takes a variational Monte Carlo run, we have to store all the walkers at every checkpoint. This involves storing all particle positions, the last calculated local energy and quantum force (which is a vector) and so on, for every walker. In the C++ implementation this is realized by the functions getBuffer and setBuffer in the Walker objects. In a call to getBuffer the walker puts all it's information into a character array. In a checkpoint these arrays are concatenated and dumped to file. When restarting the program, the arrays are read from the file and sent to the walkers through setBuffer. The checkpoints are, as for variational Monte Carlo, made after every time step.

4.1.2. Generating random numbers

Central to a Monte Carlo method is the random number generator. The Monte Carlo integration depends on a walker's ability to reach all points in phase space from its starting point. If the random numbers determining the movement of the walker are in some way correlated, the walker will lose this ability. A good random number generator is therefore of great importance. Consider the simple one-dimensional definite integral

$$F = \int_0^1 f(x)\,\mathrm{d}x. \tag{16}$$

To solve this equation numerically, we approximate F in terms of F_N:

$$F = \lim_{N\to\infty} F_N, \tag{17}$$

where

$$F_N = \frac{1}{N}\sum_{i=1}^{N} f(x_i). \tag{18}$$

When we solve Eq. (16) using Monte Carlo integration, we draw the sample points $\{x_i\}$ randomly from a given probability density function. However, as a computer only has a finite sized set of numbers available, we have to use random numbers generated from a pseudo-random number generator (PRNG). For every PRNG there is a finite number of pseudo-random numbers, known as the cycle length of the PRNG. When this cycle length is reached Eq. (17) will cease to converge. This may not seem like a serious problem as the cycle length can be made quite large by using better PRNGs. However, we have to take care to choose a good PRNG. To take an example, the PRNG `ran0` (see [14]) has a cycle length of about 2.1×10^9. On a 3.40 GHz Intel(R) Xeon(TM) CPU `ran0` takes 40 seconds to run through one cycle. It is obvious that using this (widely used) PRNG will lead to problems when a Diffusion Monte Carlo simulation takes several days of CPU time.

The generation of random numbers is a science in itself and, though of great importance to Monte Carlo methods, we will not go through this aspect in detail. We can advise the interested reader to read the introduction of [15]. In the simulations in this paper we have used a 64-bit linear congruential generator with prime addend [16,17] which has a period of 2^{64}. Linear congruential generators may have correlations between numbers that are separated by a power of 2. We should therefore take care to avoid using this generator in batches of powers of 2.

4.2. Parallelizing the C++ implementation

To parallelize Variational Monte Carlo (VMC) is embarrassingly easy. As long as you ensure that all the calculations use different sets of random numbers (and thereby ensuring that the calculations are uncorrelated) the algorithm is parallelized by running an independent calculation on each node. The communication between the nodes is restricted to spreading the input parameters before the calculations and collecting the output after calculation. The parallel efficiency is essentially 100%, and the calculation can theoretically use any number of nodes without efficiency loss.

The parallelization of Diffusion Monte Carlo (DMC) is more cumbersome. This is due to the branching part in Algorithm 1 where walkers are killed or reproduced. If we had parallelized DMC in a straight forward way, i.e. by starting one DMC run per node with different sets of random numbers and collected the results at the end, the walkers would be unevenly distributed among the processes, leading to an inefficient DMC code. For a DMC code to function properly it needs an as large as possible number of walkers to get a good representation of the wave function. A lot of unconnected DMC simulations will basically yield a set of not-so-good wave functions. We therefore have to collect all the walkers, remove dead walkers and make copies of the more virile walkers according to the branching process and then redistribute the walkers at every time step.

The parallelization is realized by a division of the walker array. A master node stores an array of the full number of walkers and distribute these walkers evenly between the slave nodes where the walkers are stored in smaller walker blocks. The preparation to sending and receiving the walker blocks is identical to the checkpoint procedure mentioned above, apart from the file writing and reading. In fact we use the MPI_Pack procedure to pack the walkers for checkpointing, even in the serial program. The only difference is that we send and receive the walkers instead of writing to and reading from file.

The main problem left is then to ensure that the sets of random numbers in fact are independent.

4.2.1. *Generating random numbers in parallel*

Generating random numbers in parallel is not as straightforward as one may think. A common first approach is to start the same random generator on every node, varying the seed with the rank of the node as a factor to get independent streams and hoping that these streams are uncorrelated. The main problem with this approach is that random generators often have long-term correlations which is of little importance in the serial case, but may appear as short-term correlations in a parallel case [16,17]. In the extreme case, we may chose seeds yielding random numbers separated with exactly one cycle. In this case we will end up with N_{CPU} identical streams, yielding N_{CPU} identical simulations and extremely good (but wrong) statistics in the results. Several approaches to get safe streams in parallel are suggested in [16,17] and implemented in the SPRNG library which we use in our simulations.

4.3. *Python implementation I*

The C++ implementation uses about 90% of the time 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 are therefore linked into a shared library readable from Python, through a thin wrapper module, together with the functions from the DMC class below the function 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. Another approach is to make a walker array class in C++. This way we can avoid explicit looping in the Python code, but we are again left to take care of varying array sizes in C++.

To understand the first approach, we must have a look at how to put the walkers into a native array. To get a C++ class visible from Python it has to be compiled and linked into a shared library. This step is taken care of by the use of SWIG [3]. The walker array is then realized by the function warray:

```
def warray(self,size,particles,dim):
    w = []
    for i in range(size):
        w += [Walker()]
        w[i].pyInitialize(particles,dim)
    return w
```

A great advantage with Python is that you can expand a class in run-time (or in fact build an entire class in run-time). Utilizing this advantage, we have inserted the function warray into the class Walker where it naturally belongs, as can be seen in the function funcToMethod:

```
def funcToMethod(func, clas, method_name=None):
    setattr(clas, method_name or func.__name__, func)
funcToMethod(warray,Walker) # insert function warray in class Walker
```

[3] Which is not a straight forward problem with C++ arrays.

This approach is particularly handy if we want to expand the Func class with new physical systems, enabling us to write the new functors in pure Python code. However, as the functors are where most of the computation time takes place, this approach will severely hinder the effectiveness of the simulation.

In the native array approach most of the parallelization is realized with the functions spread_walkers and gather_walkers. spread_walkers is as follows:

```
def spread_walkers(self):
   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)
```

If the master node calls spread_walkers, a walker buffer is made for each slave node and then sent. If the caller is a slave, it receives the relevant buffer and add it to the local walker block. The function gather_walkers is very similar, except that the slaves sends the buffers and the master receives and concatenates the buffers to the global walker array. The functions walkers2py and py2walkers are functions for converting a walker to a NumPy array and back again, taking advantage of the functions getBuffer and setBuffer in the Walker class.

A simple Python script for parallel DMC computations is as follows. We start with defining a function for one time step, or iteration:

```
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:
     # kill and replicate walkers
     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:
      # find energy for this state (details skipped)
      # 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)
```

This function may be divided into three steps. We start with running through all the walkers, doing a regular Monte Carlo step. Next we kill strayed walkers and replicate good walkers defined by the replication factor in Algorithm 1. Last, we calibrate the trial energy for the next time step. Given the `timestep` function the DMC script is as follows:

```
import pypar,math
from DMC import DMC

d = DMC(pypar)

# set initial walker positions:
d.uni_dist()

# do initial termalization using metropolis:
for i in range(d.metropolis_termalization):
    for walker in d.w_block:
        d.metropolis_step(walker)

# find initial energy (details skipped)

# do Monte Carlo iterations:
for i in range(d.steps):
    timestep(i)

if d.master:
    # calculate final energy and store to file (details skipped)
pypar.Finalize()
```

Here we generate a DMC object, setup an initial uniform walker distribution, do a termalization to distribute the walkers along the ground state of the system, make an estimation for the trial energy, and then start the production by iterating through the time steps. This implementation is capable of doing DMC computations, albeit not very efficiently, and, as you will see, there is room for improvement.

4.4. Python implementation II

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 parallelization) supports sending NumPy arrays directly, is of course helping in that choice. 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 in NumPy arrays, but this will mainly make NumPy array comparable to native arrays.

The approach with native Python arrays is quite straightforward and easy to implement. It is, however, quite inefficient as well. There are two main reasons for this. First, looping is known to be an inefficient construct in Python. With a native Python array, the loops over walkers have to be done in Python. Second, native Python arrays are slower than, e.g., NumPy arrays, because native arrays are written for a much more general use than just numerics. The question is how we can use the most of the C++ walker code as-is while avoiding explicit for-loops in Python.

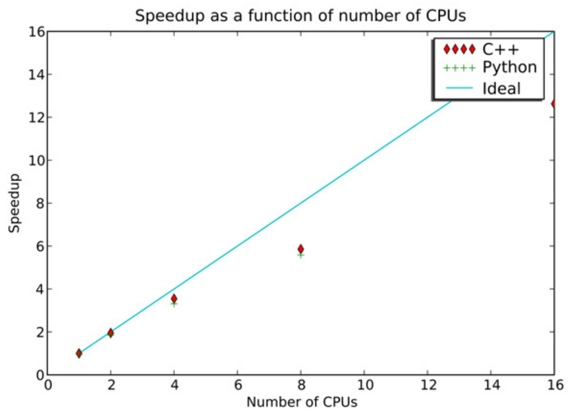
One solution is to implement an array class (lets call it WalkerArray) in Python which is wrapper class to a C++ class containing a C++ array of walkers and functionality to create and kill walkers. Even though this is a good approach in a serial implementation, we still have to convert these arrays to NumPy arrays to be able to send the walkers in MPI.

In our Python implementation, we keep the WalkerArray, but store all the walker data in a NumPy array. To do this we have modified the C++ Walker class so that it only uses pointers to an array for all arrays and variables that should be stored. This array is then provided by NumPy. Even though this approach taints the C++ implementation of the Walker class, we get the advantage that the Python DMC class only has to care about NumPy arrays, providing us with powerful tools for vectorizing the Python code.

4.4.1. WalkerArray

To implement the class WalkerArray we need to have some knowledge of the C++ Walker class. The Walker class contains some arrays storing all particle positions and all previous particle positions and variables to know if the walker should be removed or duplicated. In addition it stores the last computed local energy, quantum force and wave function to minimize the number of times we compute these quantities. In the C++ code this information is allocated and stored in each walker, making the creation of a walker rather costly. When we send walkers in MPI the information is collected from each walker and concatenated into one array before communication and inserted into the walkers after communication. Now, we want turn it the other way, i.e. we want to allocate and store all information from all walkers in one array (preferably a NumPy array) and let the walkers operate on pointers to this array. This way the time to initialize a walker is reduced dramatically and all information on walkers are readily available from Python. This change of view for the Walker class is realized by changing all variables that define the walker to references to the corresponding pointers to the NumPy array.

[4] To the best of my knowledge.



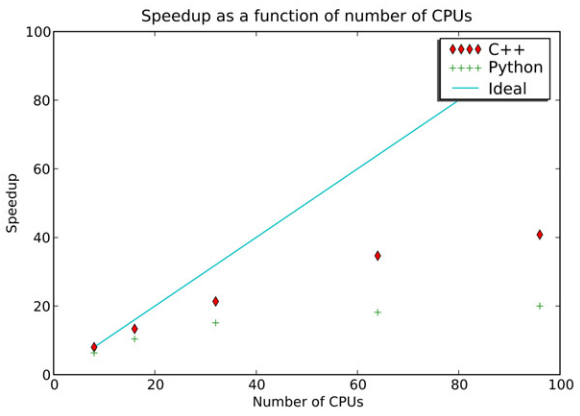


Fig. 3. Speedup of a simulation as a function of the number of CPUs used. In the left figure the serial run took about 30 minutes and was run with an initial 480 walkers moved in 3500 time steps. In the right figure the serial run took about 4 hours 30 minutes and was run with an initial 4800 walkers moved in 1750 time steps.

5. Python vs. C++

Now we know how to implement a Python version of a Monte Carlo solver. We then need to know if MontePython is efficient enough. We can assume that the Python implementation will never be faster that the corresponding C++ code, as Python will always have some degree of overhead just to access the C++ code. The question is how big this overhead may be. In Fig. 3 we have plotted the speedup of a Monte Carlo simulation as a function of the number of CPUs.[5] To the left we have done a relatively light simulation with 20 particles in 3 dimensions using 480 walkers. The walkers are spread out evenly and communicated from the master node to the slave nodes and back again 3500 times so that all walkers are sent 7000 times. The size of one walker is in this case 1.2 kB which means that for, e.g., 4 CPUs the size of each message is about 144 kB. We can see that for this message size the overhead of using Python is almost none.

To the right in Fig. 3 we have increased the number of walkers to 4800. However, we are also using a much higher number of CPUs. The message size for, e.g., 64 CPUs is only 9 kB. As we explicitly loop over the number of CPUs when we send walkers to and from the master, the overhead increases dramatically when compared to the time to send one message. The send and receive methods should therefore be vectorized with scatter and gather routines. Unfortunately we do not have uniform message sizes, making generic scatter and gather routines unusable. We will therefore have to write these routines ourselves.

Python has similar speed to C++, but the curve flattens out much faster as we increase the number of CPUs. As Monte Carlo simulations are known to have perfect speedup, we cannot be satisfied with the parallel algorithms of either the C++ version or the Python version.

6. Optimizing MontePython

One of the points in using Python in scientific programming is that you can implement new and improved algorithms efficiently. We have seen that distribution of the random walkers over the compute nodes leads to a bottleneck due to communication when the number of CPUs grows large. This bottleneck is evident both for the C++ implementation and the Python implementation. In this section we will improve the algorithm for load balancing of the walker in two ways. First, we will improve on the way the walkers are killed and reproduced. Second, we will improve on the load balancing itself by optimizing for heterogeneous clusters of CPUs.

6.1. *Distributing walkers in parallel*

The C++ Diffusion Monte Carlo application was originally written in serial and then ported to parallel using MPI. In the serial version we used an algorithm where, in order to kill a walker, we moved the last walker in the sequence onto the walker that was to be killed and decreased the number of walkers with one. To reproduce a walker, we copied the walker to the end of the sequence. This algorithm is optimal and widely used in serial Diffusion Monte Carlo. When we parallelized the code, we kept this serial algorithm by gathering all the walkers to the master node, let the master node do the killing and reproducing in serial, and then spread the walkers evenly among the slave nodes again. This way the load was always balanced, and the master had full control of the walkers at all times. The main problem is that, apart from memory issues as the master needs to store a lot of walkers, the

[5] Speedup is the time of a serial simulation divided by the walltime of the simulation.

serial work load for the master increases fast when we increase the number of CPUs in the calculation. This problem is very clear in Fig. 3, where the speedup is quite poor already for 32 CPUs, both for the C++ version and the Python version.

Again, the algorithm is best explained through the source code. First we let the slave nodes individually move their walkers and kill and reproduce their local walkers, then the function `DMC.load_balancing()` balances the load:

```
1    def load_balancing(self):
         self.t1 = time.time()
3        w_numbers = self.pypar.gather(Numeric.array([len(self.w_block)]),
                                       self.master_rank)
5        tmp_w_numbers = copy.deepcopy(w_numbers)
         w_numbers = self.pypar.broadcast(tmp_w_numbers,
7                                           self.master_rank)
9        self.no_of_walkers = Numeric.sum(w_numbers)

11       self.__find_opt_w_p_node()

13       self.first_balance = False
         balanced = Numeric.array(self.loc_walkers)
15
         difference = w_numbers-balanced
17
         diff_sort = Numeric.argsort(difference)
19       prev_i_min = diff_sort[0]

21       while sum(abs(difference))!=0:
             diff_sort = Numeric.argsort(difference)
23           i_max = diff_sort[-1]
             i_min = diff_sort[0]
25
             if i_min == prev_i_min:
27               i_min = diff_sort[1]

29           if self.myrank==i_max:
                 self.pypar.send(self.w_block[balanced[i_max]:],i_min)
31               args = [balanced[i_max],
                         self.particles,
33                       self.dimensions,
                         self.w_block[0:balanced[i_max]]]
35               self.w_block = WalkerArray.WalkerArray(*args)
             elif self.myrank==i_min:
37               recv_buff = self.pypar.receive(i_max)
                 args = [len(self.w_block)+difference[i_max],
39                       self.particles,
                         self.dimensions,
41                       Numeric.concatenate((self.w_block[:],recv_buff))]
                 self.w_block = WalkerArray.WalkerArray(*args)
43           difference[i_min]+=difference[i_max]
             difference[i_max]=0
45           prev_i_min = i_min
```

This function deserves some explanation. From line 4 to 9 we update the current walker distribution and total number of walkers. In line 11 we determine the optimal distribution of walkers. to be explained in Section 6.2. At this point we know the actual distribution of walkers and the optimal distribution of walkers. The idea is then to find the length of the difference between the optimal and actual distribution and move walkers among nodes until the length, or the sum of the absolute value of the differences is zero, see line 21. This is realized in lines 29–44 by moving the excess walkers from the node with maximum difference to the walker with minimum difference recursively. A problem with this procedure is that the same node can have a minimum difference in subsequent cycles of the while-loop.[6] This leads to unnecessary waiting in the program. The remedy is seen in line 26 where we

[6] E.g., if a node with minimum difference needs 4 walkers and the second minimum is 1, while the maximum difference is 2, the minimum node is still the same after the first cycle. Then the message of the next cycle will have to wait till the first message is sent.

take the second minimum node if the minimum node is the same node as in the previous cycle. Of course this problem may just be transferred to the second minimum node, but this is much less likely to happen.

It should be noted that this optimization does not preserve the result from the non-optimized code, in the sense that we will not get an identical output in the end. This is due to the fact that the random sequences are distributed per node and not per walker, meaning that each walker will get a different series of random numbers depending on which node it is sent to. The output will, however, be within the error range of the non-optimized code.

6.2. Heterogeneous clusters

Most new high performance clusters are more or less homogeneous, in the sense that the computation nodes have identical specifications with respect to CPU, RAM, network and storage. However, as a cluster usually expands in time, due to more funds and need for more resources, it is very likely that it will become a heterogeneous cluster. Also, with the trend of multiple cores and CPUs per computation nodes, combined with the fact that there is more than one user per cluster, different nodes will have different (and possibly too high) load and therefore different computational speed. This means that even if a cluster is homogeneous on paper, it will act like a heterogeneous cluster in practice. If we want to gain the optimal performance from a cluster, we need to take into account this heterogeneity.

In the function `__find_opt_w_p_node()` we use the time from the DMC class is initialized to the function is called to determine how the optimal distribution of walkers at every time step. The function itself goes as follows:

```
1  def __find_opt_w_p_node(self):
       self.t1 = time.time()
3      timings = self.pypar.gather(Numeric.array([abs(self.t1-self.t0)]),
                                   self.master_rank)
5      tmp_timings = copy.deepcopy(timings)
       timings = self.pypar.broadcast(tmp_timings,
7                                         self.master_rank)

9      C = self.no_of_walkers/sum(1./timings)

11     tmp_loc_walkers = C/timings

13     self.loc_walkers = self.NumericFloat2IntList(tmp_loc_walkers)
       remainders = tmp_loc_walkers-self.loc_walkers
15
       while sum(self.loc_walkers) < self.no_of_walkers:
17         maxarg = Numeric.argmax(remainders)
           self.loc_walkers[maxarg] += 1
19         remainders[maxarg] = 0

21     if self.master and self.first_balance:
           print timings
23         print self.loc_walkers

25     return self.loc_walkers
```

To understand this function we just need some simple linear algebra. Say that we have set of walkers $[x_1, x_2, \dots x_N]$ spread in an optimal way over N nodes. On node i the time to move one walker is given by a_i yielding a set $[a_1, a_2, \dots a_N]$. By *optimal distribution* we mean a distribution of walkers were each node finishes the work assigned to it between synchronizations at the same time C, i.e. $a_i x_i = C$. In addition we know the total number of walkers, T. We know that

$$\sum x_i = \sum \frac{C}{a_i}, \tag{19}$$

so that the problem reduces to finding C. However, as $T = \sum x_i$, we have

$$C = \frac{T}{\sum 1/a_i}. \tag{20}$$

Here Eq. (20) corresponds to line 9 of `__find_opt_w_p_node()`. The rest of the function is merely taking care of the fact that x_i are integers while a_i and C are real numbers.

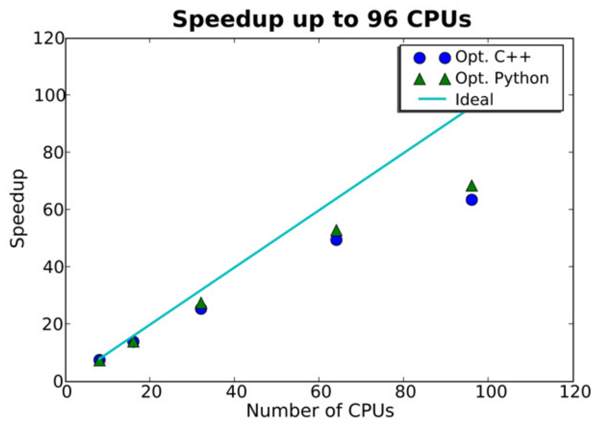


Fig. 4. The figure shows the speedup for the simulation as a function of the number of CPUs used. The serial run took about 209 minutes for C++, 225 minutes for Python and was run with an initial 4800 walkers moved in 1750 time steps.

6.3. *Optimized results*

Fig. 4 shows the speedup of the improved walker distribution compared to the same improvement in C++. We see that the speedup is actually better for Python for higher numbers of CPUs. C++ version. The main reason for this is that the serial version of C++ is faster than that of Python, with 209 minutes compared to 225 minutes. When increasing the number of CPUs the Python version gets closer to the C++ version in walltime, and already for 32 CPUs the walltimes are inseparable, with differences of less than 1%. Due to storing the data in a NumPy array, Python does not need to allocate memory for every moved walker separately every time a block of walkers is moved. It should be noted that we again see a slope in the speedup curve when going to large numbers of CPUs. This is again due to the small number of walkers on each node when having a constant global number of walkers. This effect is now independent on whether we use Python or C++.

The simple syntax in Python and the use of NumPy arrays to store walkers also allow us to concentrate our effort directly on the optimization of the algorithm instead of dealing with, e.g., how to send, receive and concatenate a slice of walkers. As a simple comparison, the author used approximately two working days to implement the optimization in Python. Due to some problems with segmentation faults in the timer function, it took the author about seven working days to implement the C++ version. This could be just as much due to lack of programming skills from the author as problems with C++, but it is quite clear that debugging is more efficient in Python than C++.

7. Visualizing with Python

We mentioned in the introduction that integration of simulation and visualization is an important feature of Matlab, Maple and others. This feature is maybe even more powerful in Python. In Figs. 5 and 6 we have used pyVTK and Mayavi [18] to plot the particle density, which is an output of diffusion Monte Carlo. pyVTK [19] is a Python interface to the Visualization ToolKit (VTK), while Mayavi, which is built on pyVTK, is a scriptable graphic interface for 3D visualization.

A signature of a Bose–Einstein condensate is that it is irrotational. If we try to rotate the condensate, it will compensate by setting up quantum vortices along the rotational axis. Vortices is therefore crucial to the study of Bose–Einstein condensates. We need only small modifications to the Hamiltonian (Eq. (3)) and trial wave function (Eq. (4)) to consider a single vortex along the z-axis in our system [7].

Figs. 5 and 6 shows the change in the ground state when inserting the vortex. The repulsive nature of the vortex pushes the particles away from the z-axis, decreasing the maximum density when compared to the ground state.

8. Conclusion

We have implemented a Monte Carlo solver using three different approaches; pure C++, a Python implementation, and an efficient, vectorized Python implementation. Furthermore we have compared the vectorized Python implementation with a corresponding C++ implementation and shown that the overhead of using Python is small for sufficiently large problems. In fact, with only two rather simple functions we were able to introduce an improved parallel algorithm for walker distribution, making the difference between Python and C++ close to negligible. In addition, we have shown that Python can be used directly as a visualization tool for rendering three dimensionally scientific visualizations. We conclude that Python can serve as a powerful tool in scientific programming.

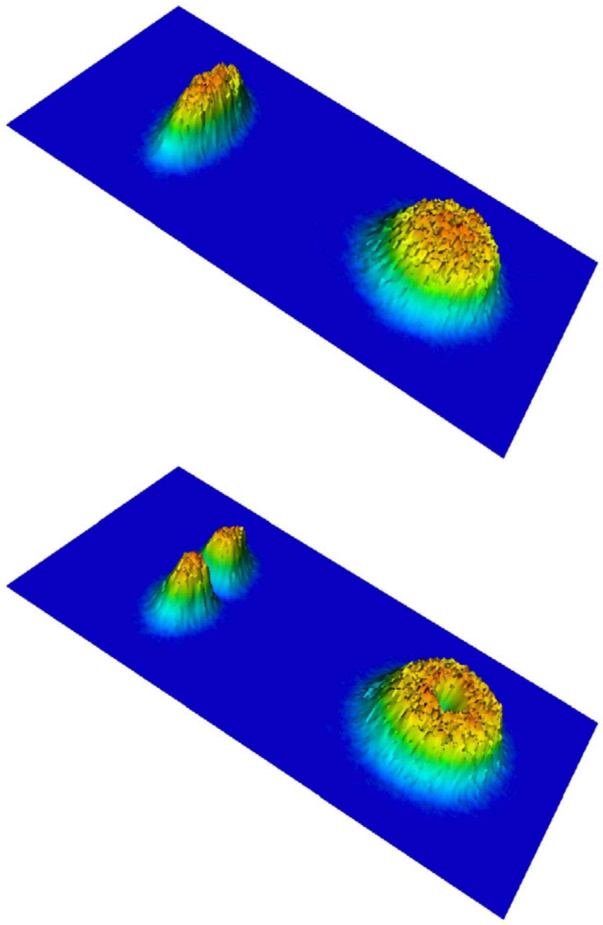


Fig. 5. The figures show where particles are detected. Plotted are the expectation values in two spatial dimensions, the yz-plane to the left and the xy-plane to the right. The topmost figure corresponds to the ground state, while the bottom figure corresponds to a state with one vortex in the center of the trap.

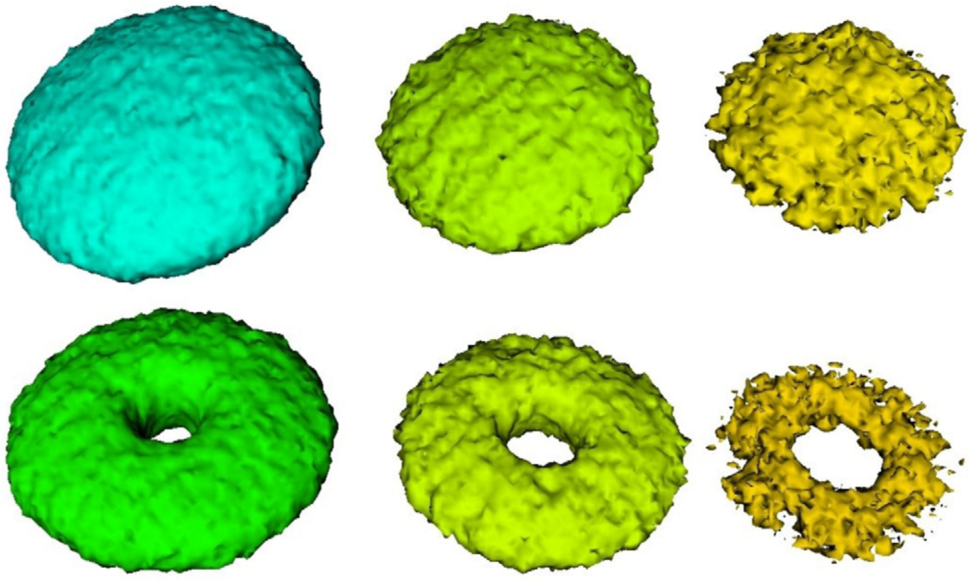


Fig. 6. The figures show where particles are detected. Plotted are the expectation values for finding, from left to right, 1, 2, 3 and 4 particles. The topmost figure corresponds to the ground state, while the bottom figure corresponds to a state with one vortex in the center of the trap.

References

[1] G. van Rossum, et al., The Python Programming Language, 1991.

[2] P. Peterson, E. Jones, T. Oliphant, et al., SciPy: Open Source Scientific Tools for Python, 2001.
[3] D. Beazley, et al., SWIG: Simplified Wrapper and Interface Generator, 1995.
[4] J.L. DuBois, H.R. Glyde, Natural orbitals and bec in traps, a diffusion Monte Carlo analysis, Phys. Rev. A 68 (2003).
[5] A. Harju, Variational Monte Carlo for interacting electrons in quantum dots, J. Low Temperature Phys. 140 (2005) 181–210.
[6] J.R. Anderson, M.R. Ensher, M.R. Matthews, C.E. Wieman, E.A. Cornel, Observation of Bose–Einstein condensation in a dilute atomic vapor, Science 269 (1995) 198.
[7] J.K. Nilsen, J. Mur-Petit, M. Guilleumas, M. Hjorth-Jensen, A. Polls, Vortices in atomic Bose–Einstein condensates in the large gas parameter region, Phys. Rev. A 71 (2005).
[8] J.L. DuBois, H.R. Glyde, Bose–Einstein condensation in trapped bosons: A variational Monte Carlo analysis, Phys. Rev. A 63 (2001).
[9] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.M. Teller, E. Teller, Equations of state calculations by fast computing machines, J. Chem. Phys. 21 (1953) 1087.
[10] B.L. Hammond, W.A. Lester Jr., P.J. Reynolds, Monte Carlo Methods in Ab Initio Quantum Chemistry, World Scientific, 1994.
[11] R. Guardiola, Monte Carlo methods in quantum many-body theories, in: J. Navarro, A. Polls (Eds.), Microscopic Quantum Many-Body Theories and their Applications, in: Lecture Notes in Physics, Springer, Verlag, 1997, pp. 269–336.
[12] P.J. Reynolds, D.M. Ceperley, B.J. Alder, W.A. Lester Jr, J. Chem. Phys. 77 (1982) 5593.
[13] A. Sarsa, J. Boronat, J. Casulleras, Quadratic diffusion Monte Carlo and pure estimators for atoms, Phys. Rev. A (2001).
[14] W.H. Press, S.A. Teukolsky, W.T. Vetterlin, B.P. Flannery, Numerical Recipes in C, second ed., Cambridge University Press, 2002.
[15] D.E. Knuth, The Art of Computer Programming, second ed., Addison-Wesley Publishing Company, 1981.
[16] M. Mascagni, A. Srinivasan, Sprng: A scalable library for pseudorandom number generation, ACM Trans. Math. Software 26 (2000) 436–461.
[17] A. Srinivasan, M. Mascagni, D. Ceperley, Testing parallel random number generators, Parallel Comput. 29 (2003) 69–94.
[18] P. Ramachandran, Mayavi: A free tool for CFD data visualization, in: 4th Annual CFD Symposium, Aeronautical Society of India, 2001.
[19] P. Peterson, PyVTK: Tools for Mmanipulating VTK Files in Python, 2001.