Diffusion Monte Carlo with Guiding Function

Jon Nilsen

j.k.nilsen@fys.uio.no

Department of Physics

University of Oslo, Norway

Basic DMC

- Considers SE in imaginary time

$$\frac{\partial \psi(\mathbf{R},t)}{\partial t} = (\widehat{H}(\mathbf{R}) - E)\psi(\mathbf{R},t)$$

- The formal solution is

$$\psi(\mathbf{R},t) = e^{-(\widehat{H}-E)t}\psi(\mathbf{R},0)$$

- $\exp(-(\widehat{H}-E)t)$ is called the *Green function* and E is a convinient energy shift.

Basic DMC cont'd

- The wave function is represented by a set of *random vectors* or *walkers* $\{\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_N\}$ in such a form that the time evolution of the wave function is actually represented by the evolution of the set of walkers.
- Computation is done in small time steps τ and Green function is approximated

$$e^{-(\widehat{H}-E)t} = \prod_{i=1}^{n} e^{-(\widehat{H}-E)\tau}$$

with $\tau = t/n$.

Basic DMC cont'd

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

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

is given by

$$\psi(\mathbf{R},t)=\sum_{\nu}e^{-(\widehat{H}-E)t}C_{\nu}\phi_{\nu}(\mathbf{R})$$

so that in the $t\rightarrow\infty$ limit the most important energy amplitude will correspond to the ground state (if $C_0\neq 0$).

Basic DMC cont'd

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

Importance Sampling DMC

- The DMC method can be speeded up with a suitable trial wave function which is called a *guide function*.
- In problems where the potential is singular the Green function can lead to large fluctuations in the number of walkers. This may be avoided by clever choices of guide functions.
- We can define a function

$$\rho(\mathbf{R},t) = \Psi_T(\mathbf{R})\psi(\mathbf{R},t)$$

Importance Sampling DMC cont'd

- The (imaginary) time evolution of $\rho(\mathbf{R},t)$ is given by the differential equation (with $\hbar = m = 1$)

$$\frac{\partial\rho(\mathbf{R},t)}{\partial t}\frac{1}{2}\nabla_R[\nabla_R - \mathbf{F}(\mathbf{R})]\rho(\mathbf{R},t) - [E_L(\mathbf{R}) - E_T]\rho(\mathbf{R},t)$$

- Here the quantum force is given by

$$\mathbf{F}(\mathbf{R}) = \frac{2}{\Psi_T(\mathbf{R})}\nabla_R\Psi_T(\mathbf{R})$$

and the local energy

$$E_L(\mathbf{R}) = \frac{\widehat{H}\Psi_T}{\Psi_T}$$

The Green function

- For short time steps the Green function can be approximated

$$G \approx G_{diff} G_B$$

- $$G_{diff}(\mathbf{y}, \mathbf{x}; \Delta\tau) = \frac{1}{\sqrt{2\pi\Delta\tau}} \exp\left[-\frac{(\mathbf{y} - \mathbf{x} - \frac{1}{2}\mathbf{F}(\mathbf{x})\Delta\tau)^2}{2\Delta\tau}\right]$$

- $$G_B(\mathbf{y}, \mathbf{x}; \Delta\tau) = \exp -(\frac{1}{2}(E_L(\mathbf{x}) + E_L(\mathbf{y})) - E_T)\tau$$

Estimating the ground state energy

- It can be shown that the population of walkers changes as

$$N(\tau + \delta\tau) = e^{-(E_0 - E_T)\delta\tau} N(\tau)$$

- This can (for a sufficiently large sample of MC points) be used to estimate a *growth* energy

$$E_g = E_T + \frac{1}{\tau_2 - \tau_1} \ln \frac{N(\tau_1)}{N(\tau_2)}$$

- We can then estimate $E_0 = \langle E_g \rangle$

Estimating the ground state energy cont'd

- Because the population varies largely during a simulation, $\langle E_g \rangle$ has a large standard deviation.
- Can instead use that

$$E_0 = \langle E_L \rangle_\rho = \lim_{M \to \infty} \frac{1}{M} \sum_{k=1}^{M} E_L(\mathbf{R}_k)$$

- Since the approximated Green function is only accurate to $O(\delta\tau^2)$, this approximation of the energy will have a time bias.

Algorithm for guided DMC

- **Initialization:** Choose target number of walkers and step size $\Delta\tau$. Walkers are placed at random in the dim x n-dimensional space of the particles.
- **Monte Carlo Steps:** After some thermalization steps, average energy is measured over time. For each time step:
  - **Iteration:** For each of the N walkers do:

Algorithm for guided DMC

- **Diffusion step:** A trial move to a new position is generated for the walker

$$\mathbf{y} = \mathbf{x} + \frac{1}{2}\Delta\tau\mathbf{F}(\mathbf{x}) + \eta\sqrt{\Delta\tau}$$

  where η is chosen randomly from a Gaussian with unit variance.
- **Metropolis test:** The trial move is accepted if the test ratio w exceeds a uniform random number between 0 and 1. If the Metropolis test succeed, then

Algorithm for guided DMC

- **Branching process:** Evaluate the branching factor

$$G_B(\mathbf{y},\mathbf{x};\Delta\tau) = \exp -(\frac{1}{2}(E_L(\mathbf{x}) + E_L(\mathbf{y})) - E_T)\tau$$

  The walker is killed if $G_B = 0$, survives if $G_B = 1$ and is cloned if $G_B > 1$.

- **Adjust N :** The number of walkers is stabilized at the targe value by adjusting

$$E_T \rightarrow E_T + \alpha \ln\left(\frac{N_T}{N}\right)$$

α a small, positive parameter.

Algorithm for guided DMC

- **Clean up:** The killed walkers are removed from the ensemble.

- **Compute averages:** Find the energy.

oneMonteCarloStep()

```
void DMC::oneMonteCarloStep(Random &ran, int k){

  double e_local_x, e_local_y, tmp_e_local_y, wf_x, wf_y, branching,
    diffusion;

  static QickArray fq_x(particles, dimensions);

  // pointer-to-reference
  Func &local_e    = *local_e_ptr;
  Func &wf         = *wf_ptr;
  Func &q_force    = *q_force_ptr;
  Func &hamilton   = *hamilton_ptr;
  Func &greens     = *greens_ptr;
  Func &potential  = *potential_ptr;
```

oneMonteCarloStep() cont'd

```
// initial info for this walker
e_local_x = walkers[k].getLocalEnergy(wf, local_e, hamilton,
                                      potential);
wf_x      = walkers[k].getWaveFunction(wf);
fq_x      = walkers[k].getQuantumForce(q_force, wf);

// move each electron
double pos;
for(int i=0; i!=particles; i++)
  for(int j=0; j!=dimensions; j++){
    pos=walkers[k].getParticlePosition(i,j);
    walkers[k].setParticlePosition(i,j,pos+D*tau*fq_x(i,j)+
                                   ran.gran(1,0)*sqrt(tau));
  }
```

oneMonteCarloStep() cont'd

```
// new info for this walker
wf_y = walkers[k].getWaveFunction(wf);

diffusion = 0;
double pos_x, pos_y;

double w = walkers[k].getGreensFunction(greens, q_force, wf,
                                        D, tau)/
  walkers[k].getNewGreensFunction(greens, q_force, wf, D, tau);

w *= sqr(wf_y/wf_x);

trials++;
```

oneMonteCarloStep() cont'd

```
// metropolis:
if(w>ran.ran1()){
  for(int i=0; i!=particles; i++)
    walkers[k].updateParticlePosition(i);
  accept++;
}
else{
  for(int i=0; i!=particles; i++)
    walkers[k].resetParticlePosition(i);
  return; // skip branching if not accepted
}
```

oneMonteCarloStep() cont'd

```
e_local_y = walkers[k].getLocalEnergy(wf, local_e, hamilton,
                                      potential);
branching = -(.5*(e_local_x+e_local_y)-e_trial)*tau;
branching = exp(branching);

// random integer with average value equal to the branching
int MB = int(branching);
if(branching-MB > ran.ran1())
  ++MB;
// add MB-1 copies at end of list
// if MB=0, mark this config as dead
for(int n=0; n<MB-1; n++){
  // copy this walker to the end of the walker list
  copy_walker(k,no_of_walkers);
  no_of_walkers++;
}
```

oneMonteCarloStep() cont'd

```
    if (MB == 0){
      walkers[k].killWalker();
    }

}
```

oneTimeStep()

```
void DMC::oneTimeStep(Random &ran, int i_step){

  Func &local_e    = *local_e_ptr;
  Func &wf         = *wf_ptr;
  Func &q_force    = *q_force_ptr;
  Func &hamilton   = *hamilton_ptr;
  Func &greens     = *greens_ptr;
  Func &potential  = *potential_ptr;

  int M = no_of_walkers;

  // let the walkers march
  for(int k=0; k!=M; k++)
    oneMonteCarloStep(ran, k);
```

oneTimeStep() cont'd

```
// bring out the dead
for(int i=0; i!=M; i++){
  // if dead walker, put in last walker
  if(walkers[i].isDead()){
    no_of_walkers--;
    copy_walker(no_of_walkers,i);
  }
}

// adjust trial energy (and no. of walkers)
```

oneTimeStep() cont'd

```
double nrg=0;
for(int i=0; i!=no_of_walkers; i++)
  nrg += walkers[i].getLocalEnergy(wf, local_e, hamilton, potentia

nrg /= double(no_of_walkers);
energy  += nrg;
energy2 += sqr(nrg);

e_trial = nrg;// + log(desired_walkers/double(no_of_walkers))/10;

e_tau[i_step] = nrg;

e_g  += e_trial;
```

diffMC()

```
}

void DMC::diffMC(){

  // object containing various random generators
  Random ran(idum);

  // set initial position
  for(int i=0; i!=particles; i++)
    for(int j=0; j!=dimensions; j++)
      for(int k=0; k!=no_of_walkers; k++){
        walkers[k].setParticlePosition(i,j,
                                       step_length/params(0)*
                                       ran.gran(1,0));
```

diffMC() cont'd

```
      walkers[k].updateParticlePosition(i);
    }

accept=trials=0;
int adjustInterval = int(0.1 * termalization) + 1;

e_g=0, e_g2=0, energy=0, energy2=0;

for(int i_step=0; i_step!=termalization; i_step++){
  oneTimeStep(ran, i_step);

  if ( (i_step+1) % adjustInterval == 0 ) {
    tau *= accept / (1.0 * trials);
    cerr << accept << " " << trials << endl;
```

diffMC() cont'd

```
} // end for(i_step)
cerr << "adjustet time step: " << tau << endl;

e_g=0, e_g2=0, energy=0, energy2=0;

for(int i_step=0; i_step!=steps; i_step++)
  oneTimeStep(ran, i_step);

// find energy and dump to screen
energy  /= double(steps);
energy2 /= double(steps);
energy2 -= sqr(energy);
```

diffMC() cont'd

```
        << sqrt(energy2/double(steps)) << endl << "sigma= "
        << energy2 << endl;

e_g  /= double(steps);
e_g2 /= double(steps);
e_g2 -= sqr(e_g);

cout << "e_g= " << e_g << " +/-"
```

Results

Hydrogen

Target no. of walkers 1000
Time steps 400
Thermalization 100

energy= −0.5 +/−6.00921e−09
sigma= 1.44442e−14
e_g= −0.442508 +/−0.0742968
sigma_g= 2.20801

Results

- Helium

Target no. of walkers 1000
Time steps 4000
Thermalization 1000

energy= −2.90505 +/−0.000231644
sigma= 0.000214636
e_g= −2.86388 +/−0.0192971
sigma_g= 1.48951

Problems

- Not optimized but very slow.
- Moves all particles for each metropolis test.
- Standard deviation one tenth of what it should be. (It's lying!)