Physics 468: Computing Project 4

January 24, 2020

Diffusion Monte-Carlo

Last week we used monte-carlo integration to estimate the interaction energy between the two electrons in the ground state of Helium. The clear advantage of this approach is that the computer has to do most of the work ;-).

The next project is to find the ground state wavefunction of an arbitrary potential using the technique that has become known as "Diffusion Monte Carlo" (see ref. 1, which this outline follows closely). It turns out that the Schrödinger equation can be twisted around to look like the diffusion equation (with a source term) if you simply transform t to $\tau = it$, so $t = -i\tau$. Start with the Schrödinger Eq.:

$$i\hbar\frac{\partial}{\partial t}\Psi(x,t) = -\frac{\hbar^2}{2m}\nabla^2\Psi(x,t) + V(r)\Psi(x,t) = \hat{H}\Psi(x,t) \tag{1}$$

Replacing t with $-it$ gives:

$$i\hbar\frac{\partial}{-i\partial\tau}\Psi(x,\tau) = -\frac{\hbar^2}{2m}\nabla^2\Psi(x,\tau) + V(r)\Psi(x,\tau) \tag{2}$$

Clearing signs gives:

$$\hbar\frac{\partial}{\partial\tau}\Psi(x,\tau) = \frac{\hbar^2}{2m}\nabla^2\Psi(x,\tau) - V(r)\Psi(x,\tau) = -\hat{H}\Psi(x,\tau) \tag{3}$$

Notice that this is nothing other than the diffusion equation (or heat equation) with the ∇^2 transport term and $-V(r)$ as a "source" term. We can adjust the energy by a constant amount to get:

$$\hbar\frac{\partial}{\partial t}\Psi(x,t) = \frac{\hbar^2}{2m}\nabla^2\Psi(x,t) + (E_r - V(r))\Psi(x,t) \tag{4}$$

This just shifts all energies (eigenvalues) by a constant amount.. but doesn't change the *eigenfunctions*. Note that if you have a time independent eigenfunction of $\hat{H}$

$$\hat{H}\,|\psi_n\rangle = E_n\,|\psi_n\rangle \tag{5}$$

You can think of adding an offset as:

$$(\hat{H} + \hat{I}E_r)\,|\psi_n\rangle = (E_n + E_r)\,|\psi_n\rangle \tag{6}$$

So all the eigenvalues are shifted by E_r, but the eigenvectors are unchanged. Note that for our transformed eigenvalue problem we have:

$$\hbar\,|\dot{\Psi}\rangle = (-\hat{H} + \hat{I}E_r)\,|\Psi\rangle \tag{7}$$

Now.. let's assume Ψ is a superposition of time evolved eigenkets of $\hat{H}$:

$$|\Psi\rangle = \sum c_n\,|\psi_n\rangle\,e^{-i\omega_n t} = \sum c_n\,|\psi_n\rangle\,e^{-\omega_n \tau} \tag{8}$$

Where ω_n are the *shifted* eigenvalues of the *shifted* Hamiltonian.

$$\omega_n = \frac{E_n - E_r}{\hbar} \tag{9}$$

Notice that if $E_n > E_r$ for a particular component, then that component simply decays exponentially with time. If $E_n < E_r$ for some component, then that component *grows* exponentially with time.

Monte-Carlo

The basic idea here is to use a monte-carlo approach to *simulate* the diffusion process and thereby get a numerical estimate of a solution. If we carefully adjust the value of E_r so that the solution reaches a steady state function over a long period of time, then all the other states will have decayed away and the only remaining term will be the ground state wavefunction.

Let's assume we have a collection of random *walkers* who just bump around at random. The density of such a group of walkers, on the average, would evolve according to the diffusion equation (Eq 3, without the source term). If we further transform our x and t to dimensionless units as $x \to Lx$ and $\tau \to T\tau$ and $E \to \mathcal{E}E$, then Eq. 3 becomes

$$\frac{\partial}{\partial\tau}\Psi(x,\tau) = \frac{\hbar T}{2mL^2}\nabla^2\Psi(x,\tau) + \frac{T\mathcal{E}}{\hbar}(E_r - V(r))\Psi(x,\tau) \tag{10}$$

If we pick $\frac{\hbar T}{mL^2} = 1$ and $\frac{T\mathcal{E}}{\hbar} = 1$ then this becomes:

$$\frac{\partial}{\partial \tau}\Psi(x,\tau) = \frac{1}{2}\nabla^2\Psi(x,\tau) + (E_r - V(r))\Psi(x,\tau) \tag{11}$$

Since there are two constraints and three variables, we are free to set any one (L, T or $\mathcal{E}$) and the other follow.

The first term describes walkers that just get a random bump of Δx in space every $\Delta\tau$ of time, where $\Delta\tau \propto \Delta x^2$. So, we can simply give our walkers a random bump of size Δx every time step. The second term describes a birth/death process. If the local potential is less than E_r, then more walkers are born there. On the other hand, if the local potential of a walker is greater than E_r, those walkers are removed (killed) from the simulation. As the system evolves E_r is continually adjusted to attempt to keep the *total* number of walkers constant.

0.1 Python Tricks

The implementation of this algorithm is shown below. There are a couple of python *tricks* (that you should learn!) that are used to make the code efficient.

```
def doMCSteps(mcsteps = 1000,   # number of monte-carlo steps
              ds=0.1,           # step size
              nWalkers=1000,    # target number of walkers
              minX=-3.0,        # minX to start walkers
              maxX=3.0,         # maxX to start walkers
              numBins=30,       # number of histogram bins
              V=SHO,            # potential. Default to Simple Harmonic Oscillator
              oldWalkers = None, # use these walkers..
              oldHist = None,   # use this histogram
              alpha = 1.0,      # the dn factor...
              ):
    """
    DMC algorithm

    Follows closely the paper "Introduction to the diffusion Monte Carlo method"

    Ioan Kisztin, Byron Faber and Klaus Schulten of U. Ill, Urban Champaign, 1995/9/25

    Start nWalkers out uniformly distributed between minX and maxX. On each time-step
    bump each walker by a small random amount +/- ds. After each bump, check the new
    potential energy for each walker relative to the current best estimate of the
    ground state energy of the system. Generally, if the walkers potential energy is
    higher than the ground state energy, the walker is deleted. If it's lower, more
    walkers are created. In this way, walkers diffuse from regions of low potential
    energy to regions of high potential energy with a density that, on the average
    is proportional to the ground state wavefunction. The estimate of the energy is
    adjusted to keep the total number of walkers constant.
    """
```

```
    vrefs = []                              # empty list of vrefs
    n0=nWalkers                             # target/initial number of walkers
    dt=ds**2                                # it's diffusion! dt goes like the square of ds

    if oldWalkers is not None:
        Walkers = oldWalkers                # use the walkers given...
        nw=len(Walkers)
    else:
        Walkers=linspace(minX,maxX,n0)      # start out with walkers distributed uniformly
        nw=n0

    if oldHist is not None:
        h=oldHist                           # use the existing histogram
    elif numBins>0:
        h=zeros(numBins)                    # histogram starts out with zeros
    else:
        h=None

    vref = V(Walkers).sum()/nw              # initialize the reference energy

    for i in xrange(mcsteps):
        Walkers += normal(size=nw)*ds           # bump walkers a bit...
        dv = V(Walkers) - vref                  # get change in V
        m_n = array(1.0 -dv*dt + rand(nw),int)  # m_n (Eq 2.28) birth/death parameter
        k=Walkers.take(flatnonzero(m_n>0))    # keep the walkers where m_n > 0
        d=Walkers.take(flatnonzero(m_n>1))    # duplicate those where m_n > 1
        Walkers=append(k,d)                   # rejoin the walkers
        nw = len(Walkers)                     # how many now?
        dn = nw-n0                            # how far are we from n0?
        vavg = V(Walkers).sum()/nw            # average potential energy of all walkers
        vref = vavg - alpha*(dn/(n0*dt))      # adjust vref (eq 2.34) based on dn

        if numBins > 0:                       # are we binning?
            vrefs.append(vref)                # yes! Do the binning
            hvals, binArray = histogram(Walkers, bins=numBins, range=(minX,maxX))
            h += hvals                        # sum the bins
        else:
            hvals, binArray = None, None      # we're not doing binning this time.

    return h, binArray, vrefs, Walkers, hvals
```

The SHO potential function is defined like so:

```
def SHO(x):
    """
    SHO potential
    """
    return x**2/2
```

The `Walkers` array is just a standard numpy array object with walker positions. The function `normal` returns a normal distribution with a variance of 1.0 and a mean of zero. The `flatnonzero` function is probably the most strangely named function in all of

numpy ;-). It takes an array (of any dimension) and returns the indices of the *flattened* array that are non-zero. The `take` method of an array, returns a new array that consists of only the elements whose indices are supplied. Finally `append` is a function that joins two different arrays to form a new array.

We'll go through the algorithm in detail in class, but the basic idea is described in the documentation string of the `doMCSteps` function. There are arguments that permit you to do a few steps, plot the situation, then (reusing the old Walkers array and histogram data) do a few more steps. This is how we can animate the walker density and histogram results. If you find `cp4_starter.py` on the "K" drive and run it you'll see three different graphs. You can adjust the output via the code:

```
if __name__=='__main__':

    SHOW_WALKERS = 1
    SHOW_HIST = 0
```

By setting `SHOW_WALKERS` to 1 you'll see something like Fig. 1.

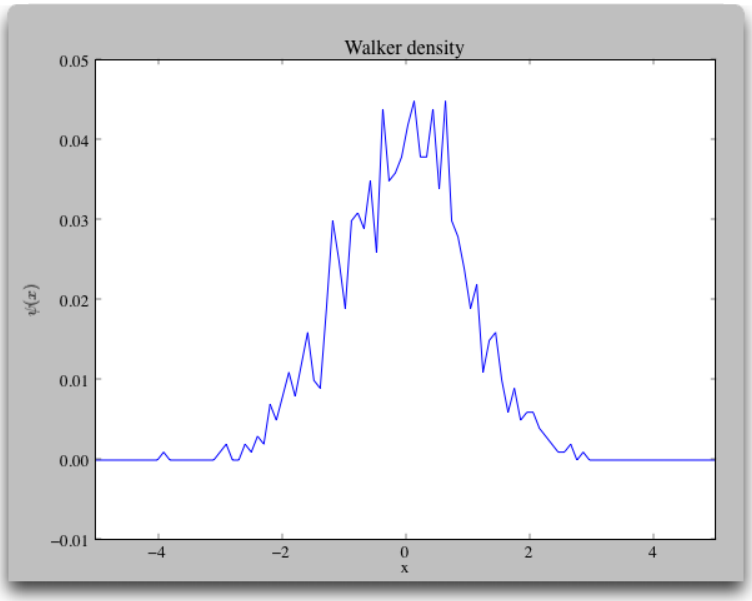

Figure 1: snapshot of walker density

By setting `SHOW_HIST` to 1 you'll see something like Fig. 2.

Finally, by setting them both to zero, you'll see a tabular output:

```
average E_r during thermalization: 3.34922332762

          i    currAvg     cumAvg      exact       diff
          0 1.71609074 1.71609074 0.50000000 1.21609074
          1 1.35883367 1.53746221 0.50000000 1.03746221
          2 0.90345090 1.32612510 0.50000000 0.82612510
          3 0.71431704 1.17317309 0.50000000 0.67317309
          4 0.59218963 1.05697640 0.50000000 0.55697640

.
.
.

        191 0.54166729 0.52138706 0.50000000 0.02138706
        192 0.45987345 0.52106833 0.50000000 0.02106833
        193 0.40347323 0.52046217 0.50000000 0.02046217
        194 0.52557984 0.52048842 0.50000000 0.02048842
        195 0.55790794 0.52067933 0.50000000 0.02067933
        196 0.43694421 0.52025428 0.50000000 0.02025428
        197 0.55379021 0.52042366 0.50000000 0.02042366
        198 0.56585716 0.52065197 0.50000000 0.02065197
        199 0.49564333 0.52052692 0.50000000 0.02052692
```

Along with the corresponding graph, Fig. 3.

So, what are we supposed to do?

Use the Diffusion Monte Carlo method to find the ground state wavefunction for either the Morse potential, or the Finite square well. Make a plot of the average energy vs. the number of diffusion steps (see example code in cp4_starter.py).

Questions

Please answer these questions at the end of your report.

1) Which potential did you choose to study? Why did you pick the potential you chose? For your potential, how did you compare the resulting wavefunction with the theoretically predicted wavefunction? How did the steady state energy compare to the theoretically predicted energy?

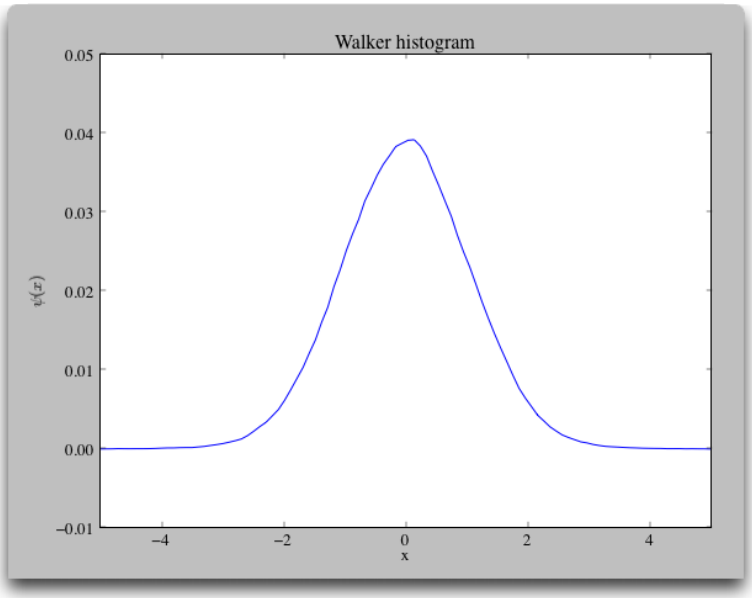

Figure 2: accumulating walker histogram

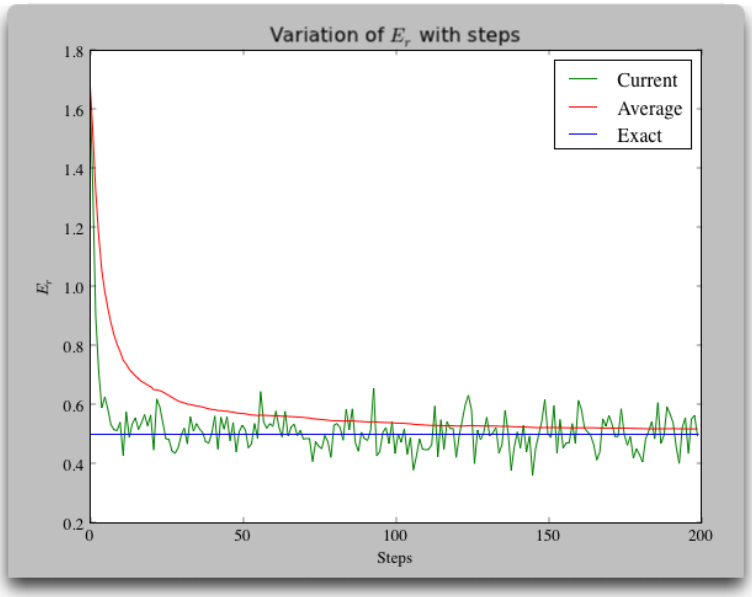

Figure 3: Energy evolution for 200 steps

2) What, if any, difficulties did you encounter in arriving at your solution? If you were asked to improve on your result, what would you propose as a strategy for achieving better results?

References

[1] *Introduction to the diffusion Monte Carlo method*, Ioan Kosztin, Byron Faber, and Klaus Schulten, Am. J. Phys. 64, 633 (1996), DOI:10.1119/1.18168