Last update: July 12, 2017

1 Overview

You will implement variational Monte Carlo (VMC) for the He atom. The algorithm works by sampling the integral:

$$\langle E \rangle = \int d^{3N}\mathbf{R} \frac{\hat{H}\Psi(\mathbf{R},\mathbf{P})}{\Psi(\mathbf{R},\mathbf{P})} |\Psi(\mathbf{R},\mathbf{P})|^2 = \lim_{M\to\infty} \frac{1}{M} \sum_{\mathbf{R}_i} \frac{\hat{H}\Psi(\mathbf{R}_i,\mathbf{P})}{\Psi(\mathbf{R}_i,\mathbf{P})} \tag{1}$$

where the samples of electron positions, $\{\mathbf{R}_i\}_{i=1}^M$, are sampled with probability $|\Psi(\mathbf{R}_i,\mathbf{P})|^2$. Here, $\mathbf{P}$ are the variational parameters. Once you can compute $\langle E \rangle$, you can simply explore the values of $\mathbf{P}$ to find the lowest energy. For the He atom, you'll find that a simple guess for the wave function can get quite close to the exact ground state energy.

1.1 Learning objectives

- How to compute wave functions and expectation value.
- How to sample many-body wave functions using the Metropolis algorithm.
- The interplay between interactions in the Hamiltonian and correlations in the wave function.

2 Implementing the pieces

Looking at the equation for $\langle E \rangle$ above, you'll see that in order to implement a variational Monte Carlo (VMC) program, we need:

- $\Psi(\mathbf{R},\mathbf{P})$, $\nabla\Psi(\mathbf{R},\mathbf{P})$, and $\nabla^2\Psi(\mathbf{R},\mathbf{P})$ (slaterwf.py)
- A way to compute $\frac{H\Psi(\mathbf{R},\mathbf{P})}{\Psi(\mathbf{R},\mathbf{P})}$ (hamiltonian.py)
- A function to generate samples with probability proportional to $|\Psi|^2$ (metropolis.py)

Each of these files has a test built in that you can use to check your implementation.

2.1 Wave function object (slaterwf.py)

A good starting point for a trial wavefunction is a Slater determinant, but because there's only one electron of each spin, it's just a product of single particle orbitals. For this exercise, we'll assume these orbitals are exponentials.

$$\Psi(\mathbf{R},\mathbf{P}) = \Psi((r_1,r_2),\alpha) = \exp(-\alpha r_1)\exp(-\alpha r_2) \tag{2}$$

Here, r_1 and r_2 are the positions of an up and down electron relative to the nucleus, and the variational parameter is α. In order to calculate the energy and other useful properties of the wave function, you'll need to implement `value(self,pos)`, `gradient(self,pos)`, and `laplacian(self,pos)` that return $\Psi(\mathbf{R},\mathbf{P})$, $\nabla\Psi(\mathbf{R},\mathbf{P})/\Psi(\mathbf{R},\mathbf{P})$, and $\nabla^2\Psi(\mathbf{R},\mathbf{P})/\Psi(\mathbf{R},\mathbf{P})$, respectively. Here, `self` contains the parameters, $\mathbf{P}$, and `pos` represents an array of the positions of all the samples of both electrons. Luckily, since we know the form of the wave function, we can compute these derivatives analytically, and avoid finite-difference error.

You'll find details of how this is defined in the skeleton of `slaterwf.py` we've written for you. There is a `pass` keyword wherever code is missing that you should implement. Start by implementing the `value(self,pos)` and `gradient(self,pos)`, then read the next section about testing.

Useful python tools:

- `np.sum(array,axis=i)` will sum over the `i`-th index.
- `array3d + array2d[np.newaxis,:,:]` duplicates the values of `array2d` over the indices of the first axis to make the sum valid.

2.1.1 Testing

Try running the file by executing `python slaterwf.py`. It should print a table with the headers `delta`, `derivative err`, and `laplacian err`. This is produced by the code in the `if __name__=="__main__":` section, which computes the derivatives numerically with finite difference, and compares it to your implementation. The `delta` controls the accuracy of finite difference, and thus the `err` columns should be small ($\sim 10^{-3}$) and grow very small as `delta` shrinks. If not, go back and check your code! Once all your functions are passing their tests, you can move on to the next section.

2.2 Hamiltonian object (hamiltonian.py)

Next you need to be able to compute the local energy of the wave function, defined as:

$$E(\mathbf{R}) \equiv \frac{\hat{H}\Psi(\mathbf{R})}{\Psi(\mathbf{R})} = -\frac{1}{2}\sum_i \frac{\nabla_i^2\Psi(\mathbf{R})}{\Psi(\mathbf{R})} - \sum_i \frac{2}{r_i} + \frac{1}{r_{12}}, \tag{3}$$

where $r_{12} = |\mathbf{r}_1 - \mathbf{r}_2|$ is the distance between the two electrons. You already have the first part from `slaterwf.py`, now we just need the potential energies. You'll implement these in `hamiltonian.py`. Like before, there are `pass` where there is missing code. Implement this before continuing.

2.2.1 Testing

You can check your code the same way as before. This time, we have computed the potential energy for a few configurations by hand. The errors you should see should be around 10^{-9}. Check that your errors are this small before continuing.

2.3 Metropolis algorithm (metropolis.py)

The next step is to draw $\mathbf{R}$ from the distribution $|\Psi(\mathbf{R})|^2$. The Metropolis-Hastings algorithm does this by the following process:

1. Start with an initial configuration $\mathbf{R}_0$.
2. Propose a new configuration $\mathbf{R}' = \mathbf{R}_0 + \sqrt{\tau}\chi$, where χ is a Gaussian random number.
3. Compute the acceptance probability $a = \frac{\Psi^2(\mathbf{R}')}{\Psi^2(\mathbf{R}_0)}$
4. Generate a uniform random number u between 0 and 1.
5. If $u < a$, then set $\mathbf{R}_1 = \mathbf{R}'$. Otherwise, set $\mathbf{R}_1 = \mathbf{R}_0$.

The τ parameter is adjusted to maintain an efficient acceptance ratio. An acceptance ratio between 0.3 and 0.7 is roughly desirable for VMC; for this first implementation, if the acceptance is too high or too low, it's a signal that the samples haven't been able to move too far from their starting point. You can play around with τ to see how it affects acceptance and the accuracy.

Once again the missing parts are marked by `pass`, so fill these parts in before moving on. Useful python tools are:

- `np.random.randn()`
- `np.random.random()`
- Conditional slices in numpy: `R[:,:,u<a] = Rprime[:,:,u<a]`

2.3.1 Testing

The test for this function will metropolis sample a Slater wave function and check it against exact solutions. This test code is carrying out the integration of Eq. (1).

Can you figure out why these are the exact answers? *Hint*: Check out how the total energy of the wave function is being evaluated and have a look at the form of our trial wave function.

This should also tell you why the total energy is correct even before the Metropolis algorithm is correctly implemented, or when the acceptance is zero (meaning the sample is completely random). What happens when you change `alpha` in that case?

3 Optimizing one parameter

At this point we have a code that can evaluate the energy of a wave function. Now using your code, you can explore the following:

- For what α is the energy minimized if we don't include electron-electron interaction? Notice anything about the errors at that point? If you understood the tests in the last section, you should know what to expect, but try using your code to see how it manifests. What should the ground state energy be in this case?
- What happens to the minimum when interactions are included? Errors? The exact ground state energy is 2.901 Ha. How close are you?
- What is the behavior of the kinetic and potential energies as a function of α? Do they make sense?

Some code to guide your investigation is provided in `he_optimization.py`. Once again, fill in the `pass` sections. It's usually good practice to produce data, write it to disk, then load and plot it separately.

3.1 Testing

Now it's your turn to write a test! Can you think of a good way to check this code? Think about what cases you know the answer already.

4 Append a Jastrow factor

In the previous section, you should have noticed that we no longer have an eigenstate with our simple Slater wave function when interactions are turned on. This is because it is a product state with no correlations. Now you'll put correlation into the wave function using a Jastrow factor.

We have implemented a Jastrow wave function and an object `MultiplyWF` which can construct a Slater-Jastrow wave function. These objects are analogous to the `SlaterWF` object, and can be used everywhere you used the `SlaterWF` object before. You can construct it by doing

```
wf=MultiplyWF(SlaterWF(alpha),JastrowWF(beta))
```

The functional form of this simple Jastrow factor is

$$\Psi_J(\mathbf{R}) = \exp(\beta r_{12}) \tag{4}$$

Optimize α and β with this new Jastrow, using the skeleton in `he_optimization.py`. How much closer are you to the ground state energy (2.901 Ha)? How have the errors changed? What happens to the optimal value of α? Why? If you get stuck on the "why", the next section might give a clue.

5 Computing other expectation values

So far we've computed the expectation of the energy, $\langle E \rangle$, but using this same technique we can evaluate any operator. Just replace the $\hat{H}$ with $\hat{O}$ where $\hat{O}$ is an operator that measures the observable.

One particularly simple $\hat{O}$ is the operator measuring the pair distribution function of the electrons. This operator is diagonal in the position basis: for each configuration, simply measure the distances between the electrons. In the special case of two electrons, the distribution is really just one number: the distance between the two electrons.

Once you write the code to compute this (no skeleton needed for this one!), use this to analyze the differences between our wave functions:

- Compare the Slater wave function to the Slater-Jastrow wave function with the same α. Which of the two has a greater average distance between electrons?
- Compare the optimized Slater wave function to the optimized Slater-Jastrow wave function. This should explain why optimized uncorrelated wave functions tend to over-localize; can you use these results to understand this? How does this relate to α changing after we appended the Jastrow?

6 Improving the sampling: biased moves (advanced)

The Metropolis algorithm has an arbitrary transition probability in it. So long as you choose the acceptance probability to satisfy detailed balance, the algorithm will still sample the correct distribution (see Wikipedia or ask me for detailed explanation). Our current set-up assumes a Gaussian transition probability, which is not the most efficient choice.

Because the acceptance is higher for transitioning to $\mathbf{R}$ where $|\Psi(\mathbf{R}, \mathbf{P})|^2$ is larger, it makes sense to bias the transition towards the direction that $\Psi(\mathbf{R}, \mathbf{P})$ is growing. The new proposed move will be

$$\mathbf{R}' = \mathbf{R}_0 + \sqrt{\tau}\chi + \tau \nabla \Psi(\mathbf{R}_0, \mathbf{P}) \tag{5}$$

where χ is still a Gaussian random number. The acceptance probability will now be

$$a = \frac{\Psi^2(\mathbf{R}')}{\Psi^2(\mathbf{R}_0)} \frac{T(\mathbf{R}' \to \mathbf{R}_0)}{T(\mathbf{R}_0 \to \mathbf{R}')} \tag{6}$$

$T(\mathbf{R}_0 \to \mathbf{R}')$ represents the probability of your transition move choosing to move from $\mathbf{R}$ to $\mathbf{R}'$, and $T(\mathbf{R}' \to \mathbf{R}_0)$ is the probability of the reverse move. For the simple moves these two are the same, but for biased moves they differ.

Now implement the new sampling function:

- What should the acceptance probability be in order to maintain detailed balance?
- The file `metropolis_drift.py` contains the skeleton for this approach, and as before, go ahead and fill in the `pass` sections. Check that the test succeeds.

- Compare the acceptance probabilities for the metropolis and biased metropolis methods. Do they produce the same energies (or other expectation values)? How do their acceptance probabilities compare?

Higher acceptance probabilities means we need to spend fewer steps to decorrelate the wave function.

7 Wrap up

It's been a few hours, so let's summarize what we've done this session:

1. Implemented several wave function objects that can also evaluate gradients and Laplacians of themselves.
2. Evaluate a He Hamiltonian with these objects.
3. Implemented a Metropolis sampling algorithm (or two, if you finished the discussion).
4. Compute expectation values using Monte Carlo.
5. Used the expectations to brute-force optimize the parameters of the trial wave function. This is a primitive VMC code.
6. Used this to explore the differences between correlated and uncorrelated wave functions.

The first four points will be useful for other QMC algorithms, so if you had trouble with these that didn't get worked out, try to fix them before the next session. If you are really stuck, come find me and I'll try and get you unstuck.