Week 3 January 18-22: Building a Variational Monte Carlo program

Morten Hjorth-Jensen Email morten.hjorth-jensen@fys.uio.no[1,2]

[1]Department of Physics and Center fo Computing in Science Education, University of Oslo, Oslo, Norway
[2]Department of Physics and Astronomy and Facility for Rare Ion Beams, Michigan State University, East Lansing, Michigan, USA

Dec 14, 2020

Overview of week 3

Topics.

- Variational Monte Carlo methods, Metropolis Algorithm, statistics and Markov Chain theory
- How to structure the VMC code

Teaching Material, videos and written material.

- Overview video
- Lecture notes and reading assignments
- Recommended background literature

Introduction

Structure and Aims. These notebooks serve the aim of linking traditional variational Monte Carlo VMC calculations methods with recent progress on solving many-particle problems using Machine Learning algorithms.

Furthermore, when linking with Machine Learning algorithms, in particular so-called Boltzmann Machines, there are interesting connections between these algorithms and so-called Shadow Wave functions (SWFs) (and references therein). The implications of the latter have been explored in various Monte Carlo calculations.

In total there are three notebooks:

1. the one you are reading now on Variational Monte Carlo methods,

2. notebook 2 on Machine Learning and quantum mechanical problems and in particular on Boltzmann Machines,

3. and finally notebook 3 on the link between Boltzmann machines and SWFs.

This notebook. In this notebook the aim is to give you an introduction as well as an understanding of the basic elements that are needed in order to develop a professional variational Monte Carlo code. We will focus on a simple system of two particles in an oscillator trap (or alternatively two fermions moving in a Coulombic potential). The particles can interact via a repulsive or an attrative force.

The advantage of these systems is that for two particles (boson or fermions) we have analytical solutions for the eigenpairs of the non-interacting case. Furthermore, for a two- or three-dimensional system of two electrons moving in a harmonic oscillator trap, we have analytical solutions for the interacting case as well.

Having analytical eigenpairs is an invaluable feature that allows us to assess the physical relevance of the trial wave functions, be these either from a standard VMC procedure, from Boltzmann Machines or from Shadow Wave functions.

In this notebook we start with the basics of a VMC calculation and introduce concepts like Markov Chain Monte Carlo methods and the Metropolis algorithm, importance sampling and Metropolis-Hastings algorithm, resampling methods to obtain better estimates of the statistical errors and minimization of the expectation values of the energy and the variance. The latter is done in order to obtain the best possible variational parameters. Furthermore it will define the so-called **cost** function, a commonly encountered quantity in Machine Learning algorithms. Minimizing the latter is the one which leads to the determination of the optimal parameters in basically all Machine Learning algorithms. For our purposes, it will serve as the first link between VMC methods and Machine Learning methods.

Topics like Markov Chain Monte Carlo and various resampling techniques are also central to Machine Learning methods. Presenting them in the context of VMC approaches leads hopefully to an easier starting point for the understanding of these methods.

Finally, the reader may ask what do we actually want to achieve with complicating life with Machine Learning methods when we can easily study interacting systems with standard Monte Carlo approaches. Our hope is that by adding additional degrees of freedom via Machine Learning algorithms, we can let the algorithms we employ learn the parameters of the model via a given optimization algorithm. In standard Monte Carlo calculations the practitioners end up with fine tuning the trial wave function using all possible insights about the system understudy. This may not always lead to the best possible ansatz and can in the long run be rather time-consuming. In fields like nuclear many-body physics with complicated interaction terms, guessing an analytical form for the trial wave fuction can be difficult. Letting the machine learn the form of the

trial function or find the optimal parameters may lead to insights about the problem which cannot be obtained by selecting various trial wave functions.

The emerging and rapidly expanding fields of Machine Learning and Quantum Computing hold also great promise in tackling the dimensionality problems (the so-called dimensionality curse in many-body problems) we encounter when studying complicated many-body problems. The approach to Machine Learning we will focus on is inspired by the idea of representing the wave function with a restricted Boltzmann machine (RBM), presented recently by G. Carleo and M. Troyer, Science **355**, Issue 6325, pp. 602-606 (2017). They named such a wave function/network a *neural network quantum state* (NQS). In their article they apply it to the quantum mechanical spin lattice systems of the Ising model and Heisenberg model, with encouraging results.

Machine learning (ML) is an extremely rich field, in spite of its young age. The increases we have seen during the last three decades in computational capabilities have been followed by developments of methods and techniques for analyzing and handling large data sets, relying heavily on statistics, computer science and mathematics. The field is rather new and developing rapidly. Machine learning is the science of giving computers the ability to learn without being explicitly programmed. The idea is that there exist generic algorithms which can be used to find patterns in a broad class of data sets without having to write code specifically for each problem. The algorithm will build its own logic based on the data.

Machine learning is a subfield of computer science, and is closely related to computational statistics. It evolved from the study of pattern recognition in artificial intelligence (AI) research, and has made contributions to AI tasks like computer vision, natural language processing and speech recognition. It has also, especially in later years, found applications in a wide variety of other areas, including bioinformatics, economy, physics, finance and marketing. An excellent reference we will come to back to is Mehta *et al.*, arXiv:1803.08823.

Our focus will first be on the basics of VMC calculations.

Basic Quantum Monte Carlo

We start with the variational principle. Given a hamiltonian H and a trial wave function $\Psi_T(\boldsymbol{R};\boldsymbol{\alpha})$, the variational principle states that the expectation value of $\mathcal{E}[\mathcal{H}]$, defined through

$$\mathcal{E}[\mathcal{H}] = \frac{\int \lceil \mathcal{R} \ominus \mathcal{T}^*(\mathcal{R};\boldsymbol{\alpha}) \mathcal{H}(\mathcal{R}) \ominus \mathcal{T}(\mathcal{R};\boldsymbol{\alpha})}{\int \lceil \mathcal{R} \ominus \mathcal{T}^*(\mathcal{R};\boldsymbol{\alpha}) \ominus \mathcal{T}(\mathcal{R};\boldsymbol{\alpha})},$$

is an upper bound to the ground state energy E_0 of the hamiltonian H, that is

$$E_0 \leq \mathcal{E}[H].$$

In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. Traditional integration methods such as

Gauss-Legendre quadrature will not be adequate for say the computation of the energy of a many-body system.

Here we have defined the vector $\boldsymbol{R} = [\boldsymbol{r}_1, \boldsymbol{r}_2, \dots, \boldsymbol{r}_n]$ as an array that contains the positions of all particles n while the vector $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \dots, \alpha_m]$ contains the variational parameters of the model, m in total.

The trial wave function can be expanded in the eigenstates $\Psi_i(\boldsymbol{R})$ of the hamiltonian since they form a complete set, viz.,

$$\Psi_T(\boldsymbol{R};\boldsymbol{\alpha}) = \sum_i a_i \Psi_i(\boldsymbol{R}),$$

and assuming that the set of eigenfunctions are normalized, one obtains

$$\frac{\sum_{nm} a_m^* a_n \int d\boldsymbol{R} \Psi_m^*(\boldsymbol{R}) H(\boldsymbol{R}) \Psi_n(\boldsymbol{R})}{\sum_{nm} a_m^* a_n \int d\boldsymbol{R} \Psi_m^*(\boldsymbol{R}) \Psi_n(\boldsymbol{R})} = \frac{\sum_n a_n^2 E_n}{\sum_n a_n^2} \geq E_0,$$

where we used that $H(\boldsymbol{R})\Psi_n(\boldsymbol{R}) = E_n \Psi_n(\boldsymbol{R})$. In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. The variational principle yields the lowest energy of states with a given symmetry.

In most cases, a wave function has only small values in large parts of configuration space, and a straightforward procedure which uses homogenously distributed random points in configuration space will most likely lead to poor results. This may suggest that some kind of importance sampling combined with e.g., the Metropolis algorithm may be a more efficient way of obtaining the ground state energy. The hope is then that those regions of configurations space where the wave function assumes appreciable values are sampled more efficiently.

The tedious part in a VMC calculation is the search for the variational minimum. A good knowledge of the system is required in order to carry out reasonable VMC calculations. This is not always the case, and often VMC calculations serve rather as the starting point for so-called diffusion Monte Carlo calculations (DMC). Diffusion Monte Carlo is a way of solving exactly the many-body Schroedinger equation by means of a stochastic procedure. A good guess on the binding energy and its wave function is however necessary. A carefully performed VMC calculation can aid in this context.

The basic procedure of a Variational Monte Carlo calculations consists thus of

1. Construct first a trial wave function $\psi_T(\boldsymbol{R};\boldsymbol{\alpha})$, for a many-body system consisting of n particles located at positions $\boldsymbol{R} = (\boldsymbol{R}_1, \dots, \boldsymbol{R}_n)$. The trial wave function depends on α variational parameters $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_M)$.

2. Then we evaluate the expectation value of the hamiltonian H

$$\overline{E}[\boldsymbol{\alpha}] = \frac{\int d\boldsymbol{R} \Psi_T^*(\boldsymbol{R},\boldsymbol{\alpha}) H(\boldsymbol{R}) \Psi_T(\boldsymbol{R},\boldsymbol{\alpha})}{\int d\boldsymbol{R} \Psi_T^*(\boldsymbol{R},\boldsymbol{\alpha}) \Psi_T(\boldsymbol{R},\boldsymbol{\alpha})}.$$

1. Thereafter we vary $\boldsymbol{\alpha}$ according to some minimization algorithm and return eventually to the first step if we are not satisfied with the results.

Here we have used the notation $\overline{E}$ to label the expectation value of the energy.

Linking with standard statistical expressions for expectation values. In order to bring in the Monte Carlo machinery, we define first a likelihood distribution, or probability density distribution (PDF). Using our ansatz for the trial wave function $\psi_T(\boldsymbol{R};\boldsymbol{\alpha})$ we define a PDF

$$P(\boldsymbol{R}) = \frac{|\psi_T(\boldsymbol{R};\boldsymbol{\alpha})|^2}{\int |\psi_T(\boldsymbol{R};\boldsymbol{\alpha})|^2 d\boldsymbol{R}}.$$

This is our model for probability distribution function. The approximation to the expectation value of the Hamiltonian is now

$$\overline{E}[\boldsymbol{\alpha}] = \frac{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R};\boldsymbol{\alpha})H(\boldsymbol{R})\Psi_T(\boldsymbol{R};\boldsymbol{\alpha})}{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R};\boldsymbol{\alpha})\Psi_T(\boldsymbol{R};\boldsymbol{\alpha})}.$$

We define a new quantity

$$E_L(\boldsymbol{R};\boldsymbol{\alpha}) = \frac{1}{\psi_T(\boldsymbol{R};\boldsymbol{\alpha})}H\psi_T(\boldsymbol{R};\boldsymbol{\alpha}),$$

called the local energy, which, together with our trial PDF yields a new expression (and which look simlar to the the expressions for moments in statistics)

$$\overline{E}[\boldsymbol{\alpha}] = \int P(\boldsymbol{R})E_L(\boldsymbol{R};\boldsymbol{\alpha})d\boldsymbol{R} \approx \frac{1}{N}\sum_{i=1}^{N} E_L(\boldsymbol{R_i};\boldsymbol{\alpha})$$

with N being the number of Monte Carlo samples. The expression on the right hand side follows from Bernoulli's law of large numbers, which states that the sample mean, in the limit $N \to \infty$ approaches the true mean

The Algorithm for performing a variational Monte Carlo calculations runs as this

- Initialisation: Fix the number of Monte Carlo steps. Choose an initial $\boldsymbol{R}$ and variational parameters α and calculate $|\psi_T^{\alpha}(\boldsymbol{R})|^2$.
- Initialise the energy and the variance and start the Monte Carlo calculation.
  - Calculate a trial position $\boldsymbol{R}_p = \boldsymbol{R} + r * step$ where r is a random variable $r \in [0,1]$.
  - Metropolis algorithm to accept or reject this move $w = P(\boldsymbol{R}_p)/P(\boldsymbol{R})$.
  - If the step is accepted, then we set $\boldsymbol{R} = \boldsymbol{R}_p$.
  - Update averages
- Finish and compute final averages.

Observe that the jumping in space is governed by the variable *step*. This is called brute-force sampling and is normally replaced by what is called **importance sampling**, discussed in more detail below here.

Simple example, the hydrogen atom. The radial Schroedinger equation for the hydrogen atom can be written as (when we have gotten rid of the first derivative term in the kinetic energy and used $rR(r) = u(r)$)

$$-\frac{\hbar^2}{2m}\frac{d^2u(r)}{dr^2} - \left(\frac{ke^2}{r} - \frac{\hbar^2 l(l+1)}{2mr^2}\right)u(r) = Eu(r).$$

We will specialize to the case with $l = 0$ and end up with

$$-\frac{\hbar^2}{2m}\frac{d^2u(r)}{dr^2} - \left(\frac{ke^2}{r}\right)u(r) = Eu(r).$$

Then we introduce a dimensionless variable $\rho = r/a$ where a is a constant with dimension length. Multiplying with $ma^2/\hbar^2$ we can rewrite our equations as

$$-\frac{1}{2}\frac{d^2u(\rho)}{d\rho^2} - \frac{ke^2ma}{\hbar^2}\frac{u(\rho)}{\rho} - \lambda u(\rho) = 0.$$

Since a is just a parameter we choose to set

$$\frac{ke^2ma}{\hbar^2} = 1,$$

which leads to $a = \hbar^2/mke^2$, better known as the Bohr radius with value 0.053 nm. Scaling the equations this way does not only render our numerical treatment simpler since we avoid carrying with us all physical parameters, but we obtain also a **natural** length scale. We will see this again and again. In our discussions below with a harmonic oscillator trap, the **natural** lentgh scale with be determined by the oscillator frequency, the mass of the particle and $\hbar$. We have also defined a dimensionless 'energy' $\lambda = Ema^2/\hbar^2$. With the rescaled quantities, the ground state energy of the hydrogen atom is 1/2. The equation we want to solve is now defined by the Hamiltonian

$$H = -\frac{1}{2}\frac{d^2}{d\rho^2} - \frac{1}{\rho}.$$

As trial wave function we peep now into the analytical solution for the hydrogen atom and use (with α as a variational parameter)

$$u_T^{\alpha}(\rho) = \alpha\rho\exp -(\alpha\rho).$$

Inserting this wave function into the expression for the local energy E_L gives

$$E_L(\rho) = -\frac{1}{\rho} - \frac{\alpha}{2}\left(\alpha - \frac{2}{\rho}\right).$$

To have analytical local energies saves us from computing numerically the second derivative, a feature which often increases our numerical expenditure with a factor of three or more. Integratng up the local energy (recall to bring back the PDF in the integration) gives $\overline{E}[\boldsymbol{\alpha}] = \alpha(\alpha/2 - 1)$.

Second example, the harmonic oscillator in one dimension. We present here another well-known example, the harmonic oscillator in one dimension for one particle. This will also serve the aim of introducing our next model, namely that of interacting electrons in a harmonic oscillator trap.

Here as well, we do have analytical solutions and the energy of the ground state, with $\hbar = 1$, is $1/2\omega$, with ω being the oscillator frequency. We use the following trial wave function

$$\psi_T(x;\alpha) = \exp -(\frac{1}{2}\alpha^2 x^2),$$

which results in a local energy

$$\frac{1}{2}\left(\alpha^2 + x^2(1-\alpha^4)\right).$$

We can compare our numerically calculated energies with the exact energy as function of α

$$\overline{E}[\alpha] = \frac{1}{4}\left(\alpha^2 + \frac{1}{\alpha^2}\right).$$

Similarly, with the above ansatz, we can also compute the exact variance which reads

$$\sigma^2[\alpha] = \frac{1}{4}\left(1 + (1-\alpha^4)^2\frac{3}{4\alpha^4}\right) - \overline{E}.$$

Our code for computing the energy of the ground state of the harmonic oscillator follows here. We start by defining directories where we store various outputs.

```
# Common imports
import os

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "Results/VMCHarmonic"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')

outfile = open(data_path("VMCHarmonic.dat"),'w')
```

We proceed with the implementation of the Monte Carlo algorithm but list first the ansatz for the wave function and the expression for the local energy

```
# VMC for the one-dimensional harmonic oscillator
# Brute force Metropolis, no importance sampling and no energy minimization
from math import exp, sqrt
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
from decimal import *
# Trial wave function for the Harmonic oscillator in one dimension
def WaveFunction(r,alpha):
    return exp(-0.5*alpha*alpha*r*r)

# Local energy  for the Harmonic oscillator in one dimension
def LocalEnergy(r,alpha):
    return 0.5*r*r*(1-alpha**4) + 0.5*alpha*alpha
```

Note that in the Metropolis algorithm there is no need to compute the trial wave function, mainly since we are just taking the ratio of two exponentials. It is then from a computational point view, more convenient to compute the argument from the ratio and then calculate the exponential. Here we have refrained from this purely of pedagogical reasons.

```
# The Monte Carlo sampling with the Metropolis algo
# The jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when the function is called.
@jit
def MonteCarloSampling():

    NumberMCcycles= 100000
    StepSize = 1.0
    # positions
    PositionOld = 0.0
    PositionNew = 0.0

    # seed for rng generator
    seed()
    # start variational parameter
    alpha = 0.4
    for ia in range(MaxVariations):
        alpha += .05
        AlphaValues[ia] = alpha
        energy = energy2 = 0.0
        #Initial position
        PositionOld = StepSize * (random() - .5)
        wfold = WaveFunction(PositionOld,alpha)
        #Loop over MC MCcycles
        for MCcycle in range(NumberMCcycles):
            #Trial position
            PositionNew = PositionOld + StepSize*(random() - .5)
            wfnew = WaveFunction(PositionNew,alpha)
            #Metropolis test to see whether we accept the move
            if random() <= wfnew**2 / wfold**2:
                PositionOld = PositionNew
                wfold = wfnew
            DeltaE = LocalEnergy(PositionOld,alpha)
            energy += DeltaE
```

```
            energy2 += DeltaE**2
        #We calculate mean, variance and error
        energy /= NumberMCcycles
        energy2 /= NumberMCcycles
        variance = energy2 - energy**2
        error = sqrt(variance/NumberMCcycles)
        Energies[ia] = energy
        Variances[ia] = variance
        outfile.write('%f %f %f %f \n' %(alpha,energy,variance,error))
    return Energies, AlphaValues, Variances
```

Finally, the results are presented here with the exact energies and variances as well.

```
#Here starts the main program with variable declarations
MaxVariations = 20
Energies = np.zeros((MaxVariations))
ExactEnergies = np.zeros((MaxVariations))
ExactVariance = np.zeros((MaxVariations))
Variances = np.zeros((MaxVariations))
AlphaValues = np.zeros(MaxVariations)
(Energies, AlphaValues, Variances) = MonteCarloSampling()
outfile.close()
ExactEnergies = 0.25*(AlphaValues*AlphaValues+1.0/(AlphaValues*AlphaValues))
ExactVariance = 0.25*(1.0+((1.0-AlphaValues**4)**2)*3.0/(4*(AlphaValues**4)))-ExactEnergies*ExactE

#simple subplot
plt.subplot(2, 1, 1)
plt.plot(AlphaValues, Energies, 'o-',AlphaValues, ExactEnergies,'r-')
plt.title('Energy and variance')
plt.ylabel('Dimensionless energy')
plt.subplot(2, 1, 2)
plt.plot(AlphaValues, Variances, '.-',AlphaValues, ExactVariance,'r-')
plt.xlabel(r'$\alpha$', fontsize=15)
plt.ylabel('Variance')
save_fig("VMCHarmonic")
plt.show()
#nice printout with Pandas
import pandas as pd
from pandas import DataFrame
data ={'Alpha':AlphaValues, 'Energy':Energies,'Exact Energy':ExactEnergies,'Variance':Variances,'E
frame = pd.DataFrame(data)
print(frame)
```

For $\alpha = 1$ we have the exact eigenpairs, as can be deduced from the table here. With $\omega = 1$, the exact energy is $1/2$ a.u. with zero variance, as it should. We see also that our computed variance follows rather well the exact variance. Increasing the number of Monte Carlo cycles will improve our statistics (try to increase the number of Monte Carlo cycles).

The fact that the variance is exactly equal to zero when $\alpha = 1$ is that we then have the exact wave function, and the action of the hamiltionan on the wave function

$$H\psi = \text{constant} \times \psi,$$

yields just a constant. The integral which defines various expectation values involving moments of the hamiltonian becomes then

$$\langle H^n \rangle = \frac{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R})H^n(\boldsymbol{R})\Psi_T(\boldsymbol{R})}{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R})\Psi_T(\boldsymbol{R})} = \text{constant}\times\frac{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R})\Psi_T(\boldsymbol{R})}{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R})\Psi_T(\boldsymbol{R})} = \text{constant}.$$

This gives an important information: the exact wave function leads to zero variance! As we will see below, many practitioners perform a minimization on both the energy and the variance.

The Metropolis algorithm

Till now we have not yet discussed the derivation of the Metropolis algorithm. We assume the reader has some familiarity with the mathematics of Markov chains.

The Metropolis algorithm , see the original article, was invented by Metropolis et. al and is often simply called the Metropolis algorithm. It is a method to sample a normalized probability distribution by a stochastic process. We define $\mathbf{P}_i^{(n)}$ to be the probability for finding the system in the state i at step n. The algorithm is then

- Sample a possible new state j with some probability $T_{i\rightarrow j}$.
- Accept the new state j with probability $A_{i\rightarrow j}$ and use it as the next sample. With probability $1-A_{i\rightarrow j}$ the move is rejected and the original state i is used again as a sample.

We wish to derive the required properties of T and A such that $\mathbf{P}_i^{(n\rightarrow\infty)} \rightarrow p_i$ so that starting from any distribution, the method converges to the correct distribution. Note that the description here is for a discrete probability distribution. Replacing probabilities p_i with expressions like $p(x_i)dx_i$ will take all of these over to the corresponding continuum expressions.

The dynamical equation for $\mathbf{P}_i^{(n)}$ can be written directly from the description above. The probability of being in the state i at step n is given by the probability of being in any state j at the previous step, and making an accepted transition to i added to the probability of being in the state i, making a transition to any state j and rejecting the move:

$$\mathbf{P}_i^{(n)} = \sum_j \left[\mathbf{P}_j^{(n-1)}T_{j\rightarrow i}A_{j\rightarrow i} + \mathbf{P}_i^{(n-1)}T_{i\rightarrow j}\left(1-A_{i\rightarrow j}\right)\right].$$

Since the probability of making some transition must be 1, $\sum_j T_{i\rightarrow j} = 1$, and the above equation becomes

$$\mathbf{P}_i^{(n)} = \mathbf{P}_i^{(n-1)} + \sum_j \left[\mathbf{P}_j^{(n-1)}T_{j\rightarrow i}A_{j\rightarrow i} - \mathbf{P}_i^{(n-1)}T_{i\rightarrow j}A_{i\rightarrow j}\right].$$

For large n we require that $\mathbf{P}_i^{(n\to\infty)} = p_i$, the desired probability distribution. Taking this limit, gives the balance requirement

$$\sum_j \left[p_j T_{j\to i} A_{j\to i} - p_i T_{i\to j} A_{i\to j}\right] = 0\,.$$

The balance requirement is very weak. Typically the much stronger detailed balance requirement is enforced, that is rather than the sum being set to zero, we set each term separately to zero and use this to determine the acceptance probabilities. Rearranging, the result is

$$\frac{A_{j\to i}}{A_{i\to j}} = \frac{p_i T_{i\to j}}{p_j T_{j\to i}}\,.$$

The Metropolis choice is to maximize the A values, that is

$$A_{j\to i} = \min\left(1, \frac{p_i T_{i\to j}}{p_j T_{j\to i}}\right).$$

Other choices are possible, but they all correspond to multilplying $A_{i\to j}$ and $A_{j\to i}$ by the same constant smaller than unity.[1]

Having chosen the acceptance probabilities, we have guaranteed that if the $\mathbf{P}_i^{(n)}$ has equilibrated, that is if it is equal to p_i, it will remain equilibrated. Next we need to find the circumstances for convergence to equilibrium.

The dynamical equation can be written as

$$\mathbf{P}_i^{(n)} = \sum_j M_{ij} \mathbf{P}_j^{(n-1)}$$

with the matrix M given by

$$M_{ij} = \delta_{ij}\left[1 - \sum_k T_{i\to k} A_{i\to k}\right] + T_{j\to i} A_{j\to i}\,.$$

Summing over i shows that $\sum_i M_{ij} = 1$, and since $\sum_k T_{i\to k} = 1$, and $A_{i\to k} \le 1$, the elements of the matrix satisfy $M_{ij} \ge 0$. The matrix M is therefore a stochastic matrix.

The Metropolis method is simply the power method for computing the right eigenvector of M with the largest magnitude eigenvalue. By construction, the correct probability distribution is a right eigenvector with eigenvalue 1. Therefore, for the Metropolis method to converge to this result, we must show that M has only one eigenvalue with this magnitude, and all other eigenvalues are smaller.

[1] The penalty function method uses just such a factor to compensate for p_i that are evaluated stochastically and are therefore noisy.

The system: two electrons in a harmonic oscillator trap in two dimensions

The Hamiltonian of the quantum dot is given by

$$\hat{H} = \hat{H}_0 + \hat{V},$$

where $\hat{H}_0$ is the many-body HO Hamiltonian, and $\hat{V}$ is the inter-electron Coulomb interactions. In dimensionless units,

$$\hat{V} = \sum_{i<j}^{N} \frac{1}{r_{ij}},$$

with $r_{ij} = \sqrt{\mathbf{r}_i^2 - \mathbf{r}_j^2}$.

This leads to the separable Hamiltonian, with the relative motion part given by ($r_{ij} = r$)

$$\hat{H}_r = -\nabla_r^2 + \frac{1}{4}\omega^2 r^2 + \frac{1}{r},$$

plus a standard Harmonic Oscillator problem for the center-of-mass motion. This system has analytical solutions in two and three dimensions (M. Taut 1993 and 1994).

We want to perform a Variational Monte Carlo calculation of the ground state of two electrons in a quantum dot well with different oscillator energies, assuming total spin $S = 0$. Our trial wave function has the following form

$$\psi_T(\boldsymbol{r}_1, \boldsymbol{r}_2) = C \exp\left(-\alpha_1 \omega (r_1^2 + r_2^2)/2\right) \exp\left(\frac{r_{12}}{(1 + \alpha_2 r_{12})}\right), \tag{1}$$

where the αs represent our variational parameters, two in this case.

Why does the trial function look like this? How did we get there? **This will be one of our main motivations** for switching to Machine Learning later.

To find an ansatz for the correlated part of the wave function, it is useful to rewrite the two-particle local energy in terms of the relative and center-of-mass motion. Let us denote the distance between the two electrons as r_{12}. We omit the center-of-mass motion since we are only interested in the case when $r_{12} \to 0$. The contribution from the center-of-mass (CoM) variable $\boldsymbol{R}_{\rm CoM}$ gives only a finite contribution. We focus only on the terms that are relevant for r_{12} and for three dimensions.

The relevant local energy becomes then

$$\lim_{r_{12}\to 0} E_L(R) = \frac{1}{_T(r_{12})}\left(2\frac{d^2}{dr_{ij}^2} + \frac{4}{r_{ij}}\frac{d}{dr_{ij}} + \frac{2}{r_{ij}} - \frac{l(l+1)}{r_{ij}^2} + 2E\right)\mathcal{R}_T(r_{12}) = 0.$$

Set $l = 0$ and we have the so-called **cusp** condition

$$\frac{d\mathcal{R}_T(r_{12})}{dr_{12}} = -\frac{1}{2(l+1)}\mathcal{R}_T(r_{12}) \qquad r_{12} \to 0$$

The above results in

$$\mathcal{R}_T \propto \exp\left(r_{ij}/2\right),$$

for anti-parallel spins and

$$\mathcal{R}_T \propto \exp\left(r_{ij}/4\right),$$

for anti-parallel spins. This is the so-called cusp condition for the relative motion, resulting in a minimal requirement for the correlation part of the wave fuction. For general systems containing more than say two electrons, we have this condition for each electron pair ij.

First code attempt for the two-electron case. First, as with the hydrogen case, we declare where to store files.

```
# Common imports
import os

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "Results/VMCQdotMetropolis"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')

outfile = open(data_path("VMCQdotMetropolis.dat"),'w')
```

Thereafter we set up the analytical expressions for the wave functions and the local energy

```
# 2-electron VMC for quantum dot system in two dimensions
# Brute force Metropolis, no importance sampling and no energy minimization
from math import exp, sqrt
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import sys
from numba import jit
```

```
# Trial wave function for the 2-electron quantum dot in two dims
def WaveFunction(r,alpha,beta):
    r1 = r[0,0]**2 + r[0,1]**2
    r2 = r[1,0]**2 + r[1,1]**2
    r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2)
    deno = r12/(1+beta*r12)
    return exp(-0.5*alpha*(r1+r2)+deno)

# Local energy  for the 2-electron quantum dot in two dims, using analytical local energy
def LocalEnergy(r,alpha,beta):

    r1 = (r[0,0]**2 + r[0,1]**2)
    r2 = (r[1,0]**2 + r[1,1]**2)
    r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2)
    deno = 1.0/(1+beta*r12)
    deno2 = deno*deno
    return 0.5*(1-alpha*alpha)*(r1 + r2) +2.0*alpha + 1.0/r12+deno2*(alpha*r12-deno2+2*beta*deno-1
```

The Monte Carlo sampling without importance sampling is set up here.

```
# The Monte Carlo sampling with the Metropolis algo
# The jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when the function is called.
@jit
def MonteCarloSampling():

    NumberMCcycles= 10000
    StepSize = 1.0
    # positions
    PositionOld = np.zeros((NumberParticles,Dimension), np.double)
    PositionNew = np.zeros((NumberParticles,Dimension), np.double)

    # seed for rng generator
    seed()
    # start variational parameter
    alpha = 0.9
    for ia in range(MaxVariations):
        alpha += .025
        AlphaValues[ia] = alpha
        beta = 0.2
        for jb in range(MaxVariations):
            beta += .01
            BetaValues[jb] = beta
            energy = energy2 = 0.0
            DeltaE = 0.0
            #Initial position
            for i in range(NumberParticles):
                for j in range(Dimension):
                    PositionOld[i,j] = StepSize * (random() - .5)
            wfold = WaveFunction(PositionOld,alpha,beta)

            #Loop over MC MCcycles
            for MCcycle in range(NumberMCcycles):
                #Trial position moving one particle at the time
                for i in range(NumberParticles):
                    for j in range(Dimension):
                        PositionNew[i,j] = PositionOld[i,j] + StepSize * (random() - .5)
                    wfnew = WaveFunction(PositionNew,alpha,beta)

                    #Metropolis test to see whether we accept the move
                    if random() < wfnew**2 / wfold**2:
```

```
                        for j in range(Dimension):
                            PositionOld[i,j] = PositionNew[i,j]
                        wfold = wfnew
                    DeltaE = LocalEnergy(PositionOld,alpha,beta)
                    energy += DeltaE
                    energy2 += DeltaE**2
                #We calculate mean, variance and error ...
                energy /= NumberMCcycles
                energy2 /= NumberMCcycles
                variance = energy2 - energy**2
                error = sqrt(variance/NumberMCcycles)
                Energies[ia,jb] = energy
                Variances[ia,jb] = variance
                outfile.write('%f %f %f %f %f\n' %(alpha,beta,energy,variance,error))
    return Energies, Variances, AlphaValues, BetaValues
```

And finally comes the main part with the plots as well.

```
#Here starts the main program with variable declarations
NumberParticles = 2
Dimension = 2
MaxVariations = 10
Energies = np.zeros((MaxVariations,MaxVariations))
Variances = np.zeros((MaxVariations,MaxVariations))
AlphaValues = np.zeros(MaxVariations)
BetaValues = np.zeros(MaxVariations)
(Energies, Variances, AlphaValues, BetaValues) = MonteCarloSampling()
outfile.close()

# Prepare for plots
fig = plt.figure()
ax = fig.gca(projection='3d')
# Plot the surface.
X, Y = np.meshgrid(AlphaValues, BetaValues)
surf = ax.plot_surface(X, Y, Energies,cmap=cm.coolwarm,linewidth=0, antialiased=False)
# Customize the z axis.
zmin = np.matrix(Energies).min()
zmax = np.matrix(Energies).max()
ax.set_zlim(zmin, zmax)
ax.set_xlabel(r'$\alpha$')
ax.set_ylabel(r'$\beta$')
ax.set_zlabel(r'$\langle E \rangle$')
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)
save_fig("QdotMetropolis")
plt.show()
```

Importance sampling

The above way of performing a Monte Carlo calculation is not the most efficient one. We need to replace the brute force Metropolis algorithm with a walk in coordinate space biased by the trial wave function. This approach is based on the Fokker-Planck equation and the Langevin equation for generating a trajectory in coordinate space. The link between the Fokker-Planck equation and the Langevin equations are explained, only partly, in the slides below. An excellent reference on topics like Brownian motion, Markov chains, the Fokker-Planck

equation and the Langevin equation is the text by Van Kampen Here we will focus first on the implementation part first.

For a diffusion process characterized by a time-dependent probability density $P(x,t)$ in one dimension the Fokker-Planck equation reads (for one particle /walker)

$$\frac{\partial P}{\partial t} = D\frac{\partial}{\partial x}\left(\frac{\partial}{\partial x} - F\right)P(x,t),$$

where F is a drift term and D is the diffusion coefficient.

The new positions in coordinate space are given as the solutions of the Langevin equation using Euler's method, namely, we go from the Langevin equation

$$\frac{\partial x(t)}{\partial t} = DF(x(t)) + \eta,$$

with η a random variable, yielding a new position

$$y = x + DF(x)\Delta t + \xi\sqrt{\Delta t},$$

where ξ is gaussian random variable and Δt is a chosen time step. The quantity D is, in atomic units, equal to $1/2$ and comes from the factor $1/2$ in the kinetic energy operator. Note that Δt is to be viewed as a parameter. Values of $\Delta t \in [0.001, 0.01]$ yield in general rather stable values of the ground state energy.

The process of isotropic diffusion characterized by a time-dependent probability density $P(\mathbf{x},t)$ obeys (as an approximation) the so-called Fokker-Planck equation

$$\frac{\partial P}{\partial t} = \sum_i D\frac{\partial}{\partial \mathbf{x_i}}\left(\frac{\partial}{\partial \mathbf{x_i}} - \mathbf{F_i}\right)P(\mathbf{x},t),$$

where $\mathbf{F_i}$ is the i^{th} component of the drift term (drift velocity) caused by an external potential, and D is the diffusion coefficient. The convergence to a stationary probability density can be obtained by setting the left hand side to zero. The resulting equation will be satisfied if and only if all the terms of the sum are equal zero,

$$\frac{\partial^2 P}{\partial \mathbf{x_i}^2} = P\frac{\partial}{\partial \mathbf{x_i}}\mathbf{F_i} + \mathbf{F_i}\frac{\partial}{\partial \mathbf{x_i}}P.$$

The drift vector should be of the form $\mathbf{F} = g(\mathbf{x})\frac{\partial P}{\partial \mathbf{x}}$. Then,

$$\frac{\partial^2 P}{\partial \mathbf{x_i}^2} = P\frac{\partial g}{\partial P}\left(\frac{\partial P}{\partial \mathbf{x}_i}\right)^2 + Pg\frac{\partial^2 P}{\partial \mathbf{x}_i^2} + g\left(\frac{\partial P}{\partial \mathbf{x}_i}\right)^2.$$

The condition of stationary density means that the left hand side equals zero. In other words, the terms containing first and second derivatives have to cancel each other. It is possible only if $g = \frac{1}{P}$, which yields

$$\mathbf{F} = 2\frac{1}{\Psi_T}\nabla\Psi_T,$$

which is known as the so-called *quantum force*. This term is responsible for pushing the walker towards regions of configuration space where the trial wave function is large, increasing the efficiency of the simulation in contrast to the Metropolis algorithm where the walker has the same probability of moving in every direction.

The Fokker-Planck equation yields a (the solution to the equation) transition probability given by the Green's function

$$G(y,x,\Delta t) = \frac{1}{(4\pi D\Delta t)^{3N/2}}\exp\left(-(y-x-D\Delta tF(x))^2/4D\Delta t\right)$$

which in turn means that our brute force Metropolis algorithm

$$A(y,x) = \min(1,q(y,x))),$$

with $q(y,x) = |\Psi_T(y)|^2/|\Psi_T(x)|^2$ is now replaced by the Metropolis-Hastings algorithm. See also Hasting's original article,

$$q(y,x) = \frac{G(x,y,\Delta t)|\Psi_T(y)|^2}{G(y,x,\Delta t)|\Psi_T(x)|^2}$$

Code example for the interacting case with importance sampling. We are now ready to implement importance sampling. This is done here for the two-electron case with the Coulomb interaction, as in the previous example. We have two variational parameters α and β. After the set up of files

```
# Common imports
import os

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "Results/VMCQdotImportance"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')

outfile = open(data_path("VMCQdotImportance.dat"),'w')
```

we move on to the set up of the trial wave function, the analytical expression for the local energy and the analytical expression for the quantum force.

```
# 2-electron VMC code for 2dim quantum dot with importance sampling
# Using gaussian rng for new positions and Metropolis- Hastings
# No energy minimization
from math import exp, sqrt
from random import random, seed, normalvariate
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import sys
from numba import jit,njit


# Trial wave function for the 2-electron quantum dot in two dims
def WaveFunction(r,alpha,beta):
    r1 = r[0,0]**2 + r[0,1]**2
    r2 = r[1,0]**2 + r[1,1]**2
    r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2)
    deno = r12/(1+beta*r12)
    return exp(-0.5*alpha*(r1+r2)+deno)

# Local energy  for the 2-electron quantum dot in two dims, using analytical local energy
def LocalEnergy(r,alpha,beta):

    r1 = (r[0,0]**2 + r[0,1]**2)
    r2 = (r[1,0]**2 + r[1,1]**2)
    r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2)
    deno = 1.0/(1+beta*r12)
    deno2 = deno*deno
    return 0.5*(1-alpha*alpha)*(r1 + r2) +2.0*alpha + 1.0/r12+deno2*(alpha*r12-deno2+2*beta*deno-1

# Setting up the quantum force for the two-electron quantum dot, recall that it is a vector
def QuantumForce(r,alpha,beta):

    qforce = np.zeros((NumberParticles,Dimension), np.double)
    r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2)
    deno = 1.0/(1+beta*r12)
    qforce[0,:] = -2*r[0,:]*alpha*(r[0,:]-r[1,:])*deno*deno/r12
    qforce[1,:] = -2*r[1,:]*alpha*(r[1,:]-r[0,:])*deno*deno/r12
    return qforce
```

The Monte Carlo sampling includes now the Metropolis-Hastings algorithm, with the additional complication of having to evaluate the **quantum force** and the Green's function which is the solution of the Fokker-Planck equation.

```
# The Monte Carlo sampling with the Metropolis algo
# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit()
def MonteCarloSampling():

    NumberMCcycles= 100000
    # Parameters in the Fokker-Planck simulation of the quantum force
    D = 0.5
    TimeStep = 0.05
    # positions
```

```
PositionOld = np.zeros((NumberParticles,Dimension), np.double)
PositionNew = np.zeros((NumberParticles,Dimension), np.double)
# Quantum force
QuantumForceOld = np.zeros((NumberParticles,Dimension), np.double)
QuantumForceNew = np.zeros((NumberParticles,Dimension), np.double)

# seed for rng generator
seed()
# start variational parameter  loops, two parameters here
alpha = 0.9
for ia in range(MaxVariations):
    alpha += .025
    AlphaValues[ia] = alpha
    beta = 0.2
    for jb in range(MaxVariations):
        beta += .01
        BetaValues[jb] = beta
        energy = energy2 = 0.0
        DeltaE = 0.0
        #Initial position
        for i in range(NumberParticles):
            for j in range(Dimension):
                PositionOld[i,j] = normalvariate(0.0,1.0)*sqrt(TimeStep)
        wfold = WaveFunction(PositionOld,alpha,beta)
        QuantumForceOld = QuantumForce(PositionOld,alpha, beta)

        #Loop over MC MCcycles
        for MCcycle in range(NumberMCcycles):
            #Trial position moving one particle at the time
            for i in range(NumberParticles):
                for j in range(Dimension):
                    PositionNew[i,j] = PositionOld[i,j]+normalvariate(0.0,1.0)*sqrt(TimeStep)+
                                       QuantumForceOld[i,j]*TimeStep*D
                wfnew = WaveFunction(PositionNew,alpha,beta)
                QuantumForceNew = QuantumForce(PositionNew,alpha, beta)
                GreensFunction = 0.0
                for j in range(Dimension):
                    GreensFunction += 0.5*(QuantumForceOld[i,j]+QuantumForceNew[i,j])*\
                                  (D*TimeStep*0.5*(QuantumForceOld[i,j]-QuantumForceNew[i,j])-
                                  PositionNew[i,j]+PositionOld[i,j])

                GreensFunction = exp(GreensFunction)
                ProbabilityRatio = GreensFunction*wfnew**2/wfold**2
                #Metropolis-Hastings test to see whether we accept the move
                if random() <= ProbabilityRatio:
                   for j in range(Dimension):
                       PositionOld[i,j] = PositionNew[i,j]
                       QuantumForceOld[i,j] = QuantumForceNew[i,j]
                   wfold = wfnew
            DeltaE = LocalEnergy(PositionOld,alpha,beta)
            energy += DeltaE
            energy2 += DeltaE**2
        # We calculate mean, variance and error (no blocking applied)
        energy /= NumberMCcycles
        energy2 /= NumberMCcycles
        variance = energy2 - energy**2
        error = sqrt(variance/NumberMCcycles)
        Energies[ia,jb] = energy
        outfile.write('%f %f %f %f %f\n' %(alpha,beta,energy,variance,error))
return Energies, AlphaValues, BetaValues
```

The main part here contains the setup of the variational parameters, the energies and the variance.

```
#Here starts the main program with variable declarations
NumberParticles = 2
Dimension = 2
MaxVariations = 10
Energies = np.zeros((MaxVariations,MaxVariations))
AlphaValues = np.zeros(MaxVariations)
BetaValues = np.zeros(MaxVariations)
(Energies, AlphaValues, BetaValues) = MonteCarloSampling()
outfile.close()
# Prepare for plots
fig = plt.figure()
ax = fig.gca(projection='3d')
# Plot the surface.
X, Y = np.meshgrid(AlphaValues, BetaValues)
surf = ax.plot_surface(X, Y, Energies,cmap=cm.coolwarm,linewidth=0, antialiased=False)
# Customize the z axis.
zmin = np.matrix(Energies).min()
zmax = np.matrix(Energies).max()
ax.set_zlim(zmin, zmax)
ax.set_xlabel(r'$\alpha$')
ax.set_ylabel(r'$\beta$')
ax.set_zlabel(r'$\langle E \rangle$')
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)
save_fig("QdotImportance")
plt.show()
```

Technical aspects, improvements and how to define the cost function

The above procedure is also not the smartest one. Looping over all variational parameters becomes expensive and we see from the previous plot that the surface is not very smooth, indicating that we need many more Monte Carlo cycles in order to reliably define an energy minimum.

What we can do however is to perform some preliminary calculations with selected variational parameters (normally with less Monte Carlo cycles than those used in a full production calculation). For every step we evaluate the derivatives of the energy as functions of the variational parameters. When the derivatives disappear we have hopefully reached the global minimum.

At this point we have the optimal variational parameters and can start our large-scale production run. To find the optimal parameters entails the computation of the gradients of the energy and optimization algorithms like various **gradient descent** methods. This is an art by itself and is discussed for example in our lectures on optimization methods. We refer the reader to these notes for more details.

This part allows us also to link with the true working horse of every Machine Learning algorithm, namely the optimization part. This normally involves one of

the stochastic gradient descent algorithms discussed in the above lecture notes. We will come back to these topics in the second notebook.

In order to apply these optmization algortihms we anticipate partly what is to come in notebook 2 on Boltzmann machines. Our cost (or loss) function is here given by the expectation value of the energy as function of the variational parameters.

To find the derivatives of the local energy expectation value as function of the variational parameters, we can use the chain rule and the hermiticity of the Hamiltonian.

Let us define

$$\bar{E}_{\alpha_i} = \frac{d\langle E_L\rangle}{d\alpha_i}.$$

as the derivative of the energy with respect to the variational parameter α_i We define also the derivative of the trial function (skipping the subindex T) as

$$\bar{\Psi}_i = \frac{d\Psi}{d\alpha_i}.$$

The elements of the gradient of the local energy are then (using the chain rule and the hermiticity of the Hamiltonian)

$$\bar{E}_i = 2\left(\langle\frac{\bar{\Psi}_i}{\Psi}E_L\rangle - \langle\frac{\bar{\Psi}_i}{\Psi}\rangle\langle E_L\rangle\right).$$

From a computational point of view it means that we need to compute the expectation values of

$$\langle\frac{\bar{\Psi}_i}{\Psi}E_L\rangle,$$

and

$$\langle\frac{\bar{\Psi}_i}{\Psi}\rangle\langle E_L\rangle$$

These integrals are evaluted using MC intergration (with all its possible error sources). We can then use methods like stochastic gradient or other minimization methods to find the optimal variational parameters

As an alternative to the energy as cost function, we could use the variance as the cost function. As discussed earlier, if we have the exact wave function, the variance is exactly equal to zero. Suppose the trial function (our model) is the exact wave function.

The variance is defined as

$$\sigma_E = \langle E^2\rangle - \langle E\rangle^2.$$

Some practitioners perform Monte Carlo calculations by minimizing both the energy and the variance.

In order to minimize the variance we need the derivatives of

$$\sigma_E = \langle E^2\rangle - \langle E\rangle^2,$$

with respect to the variational parameters. The derivatives of the variance can then be used to defined the so-called Hessian matrix, which in turn allows us to use minimization methods like Newton's method or standard gradient methods.

This leads to however a more complicated expression, with obvious errors when evaluating many more integrals by Monte Carlo integration. It is normally less used, see however Filippi and Umrigar. The expression becomes complicated

$$\bar{E}_{ij} = 2\left[\langle(\frac{\bar{\Psi}_{ij}}{\Psi}+\frac{\bar{\Psi}_j}{\Psi}\frac{\bar{\Psi}_i}{\Psi})(E_L-\langle E\rangle)\rangle - \langle\frac{\bar{\Psi}_i}{\Psi}\rangle\bar{E}_j - \langle\frac{\bar{\Psi}_j}{\Psi}\rangle\bar{E}_i\right] + \langle\frac{\bar{\Psi}_i}{\Psi}E_{Lj}\rangle + \langle\frac{\bar{\Psi}_j}{\Psi}E_{Li}\rangle - \langle\frac{\bar{\Psi}_i}{\Psi}\rangle\langle E_{Lj}\rangle\langle\frac{\bar{\Psi}_j}{\Psi}\rangle\langle E_{Li}\rangle.$$

Evaluating the cost function means having to evaluate the above second derivative of the energy.

Before we proceed with code examples, let us look at some simple examples, here the one-particle harmonic oscillator in one dimension. This serves as a very useful check when developing a code. The first code discussed is the two-dimensional non-interacting harmonic oscillator.

Simple example. Let us illustrate what is needed in our calculations using a simple example, the harmonic oscillator in one dimension. For the harmonic oscillator in one-dimension we have a trial wave function and probability

$$\psi_T(x) = e^{-\alpha^2x^2} \qquad P_T(x)dx = \frac{e^{-2\alpha^2x^2}dx}{\int dx e^{-2\alpha^2x^2}}$$

with α being the variational parameter. We obtain then the following local energy

$$E_L[\alpha] = \alpha^2 + x^2\left(\frac{1}{2}-2\alpha^2\right),$$

which results in the expectation value for the local energy

$$\langle E_L[\alpha]\rangle = \frac{1}{2}\alpha^2 + \frac{1}{8\alpha^2}$$

The derivative of the energy with respect to α gives

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = \alpha - \frac{1}{4\alpha^3}$$

and a second derivative which is always positive (meaning that we find a minimum)

$$\frac{d^2\langle E_L[\alpha]\rangle}{d\alpha^2} = 1 + \frac{3}{4\alpha^4}$$

The condition

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = 0,$$

gives the optimal $\alpha = 1/\sqrt{2}$, as expected.

We can also minimize the variance. In our simple model the variance is

$$\sigma^2[\alpha] = \frac{1}{2}\alpha^4 - \frac{1}{4} + \frac{1}{32\alpha^4},$$

with first derivative

$$\frac{d\sigma^2[\alpha]}{d\alpha} = 2\alpha^3 - \frac{1}{8\alpha^5}$$

and a second derivative which is always positive (as expected for a convex function)

$$\frac{d^2\sigma^2[\alpha]}{d\alpha^2} = 6\alpha^2 + \frac{5}{8\alpha^6}$$

In general we end up computing the expectation value of the energy in terms of some parameters $\alpha_0, \alpha_1, \ldots, \alpha_n$ and we search for a minimum in this multi-variable parameter space. This leads to an energy minimization problem *where we need the derivative of the energy as a function of the variational parameters.*

In the above example this was easy and we were able to find the expression for the derivative by simple derivations. However, in our actual calculations the energy is represented by a multi-dimensional integral with several variational parameters.