XXXX

LNP 1000

July 2022

Springer

Contents

Chapter 1
From Linear Algebra to Machine Learning: Ten Central Algorithms for Studying Quantum Mechanical Many-Particle Problems

Morten Hjorth-Jensen

Variational Monte Carlo methods

Quantum Monte Carlo Motivation

We start with the variational principle. Given a hamiltonian H and a trial wave function Ψ_T, the variational principle states that the expectation value of $\langle H \rangle$, defined through

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

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

$$E_0 \leq \langle H \rangle.$$

In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. Traditional integration methods such as the Gauss-Legendre will not be adequate for say the computation of the energy of a many-body system.

The trial wave function can be expanded in the eigenstates of the hamiltonian since they form a complete set, viz.,

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

and assuming the set of eigenfunctions to be 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 state of 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

Morten Hjorth-Jensen
Department of Physics and Astronomy and Facility for Rare Isotope Beams, Michigan State University, East Lansing, Michigan, USA and Department of Physics, University of Oslo and Center for Computing in Science Education, Oslo, Norway, e-mail: hjensen@msu.edu

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). DMC 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 recipe in a VMC calculation consists of the following elements:

- 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 $\boldsymbol{\alpha}=(\alpha_1,\dots,\alpha_M)$.
- Then we evaluate the expectation value of the hamiltonian H

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

- Thereafter we vary α according to some minimization algorithm and return to the first step.

With a trial wave function $\psi_T(\boldsymbol{R})$ we can in turn construct the quantum mechanical probability distribution

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

This is our new probability distribution function (PDF). The approximation to the expectation value of the Hamiltonian is now

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

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

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

with N being the number of Monte Carlo samples.

The Algorithm for performing a variational Monte Carlo calculations runs thus 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. Need importance sampling to get more relevant sampling, see lectures below.

Quantum Monte Carlo: hydrogen atom.

The radial Schroedinger equation for the hydrogen atom can be written as

$$-\frac{^2}{2m}\frac{\partial^2 u(r)}{\partial r^2}-\left(\frac{ke^2}{r}-\frac{^2 l(l+1)}{2mr^2}\right)u(r)=Eu(r),$$

or with dimensionless variables

$$-\frac{1}{2}\frac{\partial^2 u(\rho)}{\partial \rho^2}-\frac{u(\rho)}{\rho}+\frac{l(l+1)}{2\rho^2}u(\rho)-\lambda u(\rho)=0,$$

with the hamiltonian

$$H=-\frac{1}{2}\frac{\partial^2}{\partial \rho^2}-\frac{1}{\rho}+\frac{l(l+1)}{2\rho^2}.$$

Use variational parameter α in the trial wave function

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

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

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

A simple variational Monte Carlo calculation results in

α	$\langle H\rangle$	σ^2	$\sigma/\sqrt{N}$
7.00000E-01	-4.57759E-01	4.51201E-02	6.71715E-04
8.00000E-01	-4.81461E-01	3.05736E-02	5.52934E-04
9.00000E-01	-4.95899E-01	8.20497E-03	2.86443E-04
1.00000E-00	-5.00000E-01	0.00000E+00	0.00000E+00
1.10000E+00	-4.93738E-01	1.16989E-02	3.42036E-04
1.20000E+00	-4.75563E-01	8.85899E-02	9.41222E-04
1.30000E+00	-4.54341E-01	1.45171E-01	1.20487E-03

We note that at $\alpha=1$ we obtain the exact result, and the variance is zero, as it should. The reason 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! Variation is then performed by minimizing both the energy and the variance.

For bosons in a harmonic oscillator-like trap we will use is a spherical (S) or an elliptical (E) harmonic trap in one, two and finally three dimensions, with the latter given by

$$V_{ext}(\mathbf{r})=\begin{cases}\frac{1}{2}m\omega_{ho}^2 r^2 & (S)\\ \frac{1}{2}m[\omega_{ho}^2(x^2+y^2)+\omega_z^2 z^2] & (E)\end{cases} \tag{1.1}$$

where (S) stands for symmetric and

$$\hat{H} = \sum_{i}^{N} \left(\frac{-^2}{2m} \bigtriangledown_i^2 + V_{ext}(\mathbf{r}_i) \right) + \sum_{i<j}^{N} V_{int}(\mathbf{r}_i, \mathbf{r}_j), \tag{1.2}$$

as the two-body Hamiltonian of the system.

We will represent the inter-boson interaction by a pairwise, repulsive potential

$$V_{int}(|\mathbf{r}_i - \mathbf{r}_j|) = \begin{cases} \infty & |\mathbf{r}_i - \mathbf{r}_j| \leq a \\ 0 & |\mathbf{r}_i - \mathbf{r}_j| > a \end{cases} \tag{1.3}$$

where a is the so-called hard-core diameter of the bosons. Clearly, $V_{int}(|\mathbf{r}_i - \mathbf{r}_j|)$ is zero if the bosons are separated by a distance $|\mathbf{r}_i - \mathbf{r}_j|$ greater than a but infinite if they attempt to come within a distance $|\mathbf{r}_i - \mathbf{r}_j| \leq a$.

Our trial wave function for the ground state with N atoms is given by

$$\Psi_T(\mathbf{R}) = \Psi_T(\mathbf{r}_1, \mathbf{r}_2, \ldots \mathbf{r}_N, \alpha, \beta) = \prod_i g(\alpha, \beta, \mathbf{r}_i) \prod_{i<j} f(a, |\mathbf{r}_i - \mathbf{r}_j|), \tag{1.4}$$

where α and β are variational parameters. The single-particle wave function is proportional to the harmonic oscillator function for the ground state

$$g(\alpha, \beta, \mathbf{r}_i) = \exp\left[-\alpha(x_i^2 + y_i^2 + \beta z_i^2)\right]. \tag{1.5}$$

For spherical traps we have $\beta = 1$ and for non-interacting bosons ($a = 0$) we have $\alpha = 1/2a_{ho}^2$. The correlation wave function is

$$f(a, |\mathbf{r}_i - \mathbf{r}_j|) = \begin{cases} 0 & |\mathbf{r}_i - \mathbf{r}_j| \leq a \\ (1 - \frac{a}{|\mathbf{r}_i - \mathbf{r}_j|}) & |\mathbf{r}_i - \mathbf{r}_j| > a. \end{cases} \tag{1.6}$$

A simple Python code that solves the two-boson or two-fermion case in two-dimensions.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python
Importing various packages from math import exp, sqrt from random import random, seed import numpy as np import matplotlib.pyplot as plt from mpl$_{t}oolkits.mplot3dimportAxes3Dfrommatplotlibimportcmfrommatplotli$

Trial wave function for quantum dots 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 quantum dots 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-1.0/r12)

The Monte Carlo sampling with the Metropolis algo def MonteCarloSampling():

NumberMCcycles= 100000 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 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: PositionOld = PositionNew.copy() 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 return Energies, AlphaValues, BetaValues

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()

Prepare for plots fig = plt.figure() ax = fig.gca(projection='3d') Plot the surface. X, Y = np.meshgrid(AlphaValues, BetaValues) surf = ax.plot$_surface(X,Y,Energies,cmap=cm.coolwarm,linewidth=0,antialiased=False)Customizethezaxis.zmin=np.matrix(Energies).min()zmax=np.matrix(Energies).max()ax.set_zlim(zmin,zmax)ax.set_xlab$ ax.set$_ylabel(r'\beta')$ ax.set$_zlabel(r'\langle E\rangle')$ ax.zaxis.set$_majorlocator(LinearLocator(10))ax.zaxis.set_majorformatter(FormatStrFormatter('A$ $0.5,aspect=5)plt.show()$

Quantum Monte Carlo: the helium atom

The helium atom consists of two electrons and a nucleus with charge $Z = 2$. The contribution to the potential energy due to the attraction from the nucleus is

$$-\frac{2ke^2}{r_1}-\frac{2ke^2}{r_2},$$

and if we add the repulsion arising from the two interacting electrons, we obtain the potential energy

$$V(r_1,r_2) = -\frac{2ke^2}{r_1}-\frac{2ke^2}{r_2}+\frac{ke^2}{r_{12}},$$

with the electrons separated at a distance $r_{12} = |\boldsymbol{r}_1-\boldsymbol{r}_2|$.

The hamiltonian becomes then

$$\hat{H} = -\frac{^2\nabla_1^2}{2m}-\frac{^2\nabla_2^2}{2m}-\frac{2ke^2}{r_1}-\frac{2ke^2}{r_2}+\frac{ke^2}{r_{12}},$$

and Schroedingers equation reads

$$\hat{H}\psi = E\psi.$$

All observables are evaluated with respect to the probability distribution

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

generated by the trial wave function. The trial wave function must approximate an exact eigenstate in order that accurate results are to be obtained.

Choice of trial wave function for Helium: Assume $r_1 \to 0$.

$$E_L(\boldsymbol{R}) = \frac{1}{\psi_T(\boldsymbol{R})}H\psi_T(\boldsymbol{R}) = \frac{1}{\psi_T(\boldsymbol{R})}\left(-\frac{1}{2}\nabla_1^2-\frac{Z}{r_1}\right)\psi_T(\boldsymbol{R})+\text{finite terms.}$$

$$E_L(R) = \frac{1}{\mathbf{R}_T(r_1)}\left(-\frac{1}{2}\frac{d^2}{dr_1^2}-\frac{1}{r_1}\frac{d}{dr_1}-\frac{Z}{r_1}\right)\mathbf{R}_T(r_1)+\text{finite terms}$$

For small values of r_1, the terms which dominate are

$$\lim_{r_1\to 0} E_L(R) = \frac{1}{\mathbf{R}_T(r_1)}\left(-\frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1}\right)\mathbf{R}_T(r_1),$$

since the second derivative does not diverge due to the finiteness of Ψ at the origin.

This results in

$$\frac{1}{\mathbf{R}_T(r_1)}\frac{d\mathbf{R}_T(r_1)}{dr_1} = -Z,$$

and

$$\mathbf{R}_T(r_1) \propto e^{-Zr_1}.$$

A similar condition applies to electron 2 as well. For orbital momenta $l > 0$ we have

$$\frac{1}{\mathbf{R}_T(r)}\frac{d\mathbf{R}_T(r)}{dr} = -\frac{Z}{l+1}.$$

Similarly, studying the case $r_{12} \to 0$ we can write a possible trial wave function as

$$\psi_T(\boldsymbol{R}) = e^{-\alpha(r_1+r_2)}e^{\beta r_{12}}.$$

The last equation can be generalized to

$$\psi_T(\boldsymbol{R}) = \phi(\boldsymbol{r}_1)\phi(\boldsymbol{r}_2)\ldots\phi(\boldsymbol{r}_N)\prod_{i<j} f(r_{ij}),$$

for a system with N electrons or particles.

During the development of our code we need to make several checks. It is also very instructive to compute a closed form expression for the local energy. Since our wave function is rather simple it is straightforward to find an analytic expressions. Consider first the case of the simple helium function

$$\Psi_T(\boldsymbol{r}_1,\boldsymbol{r}_2) = e^{-\alpha(r_1+r_2)}$$

The local energy is for this case

$$E_{L1} = (\alpha - Z)\left(\frac{1}{r_1}+\frac{1}{r_2}\right) + \frac{1}{r_{12}} - \alpha^2$$

which gives an expectation value for the local energy given by

$$\langle E_{L1}\rangle = \alpha^2 - 2\alpha\left(Z - \frac{5}{16}\right)$$

With closed form formulae we can speed up the computation of the correlation. In our case we write it as

$$\Psi_C = \exp\left\{\sum_{i<j}\frac{ar_{ij}}{1+\beta r_{ij}}\right\},$$

which means that the gradient needed for the so-called quantum force and local energy can be calculated analytically. This will speed up your code since the computation of the correlation part and the Slater determinant are the most time consuming parts in your code.

We will refer to this correlation function as Ψ_C or the *linear Pade-Jastrow*.

We can test this by computing the local energy for our helium wave function

$$\psi_T(\boldsymbol{r}_1,\boldsymbol{r}_2) = \exp\left(-\alpha(r_1+r_2)\right)\exp\left(\frac{r_{12}}{2(1+\beta r_{12})}\right),$$

with α and β as variational parameters.

The local energy is for this case

$$E_{L2} = E_{L1} + \frac{1}{2(1+\beta r_{12})^2}\left\{\frac{\alpha(r_1+r_2)}{r_{12}}(1-\frac{\boldsymbol{r}_1\boldsymbol{r}_2}{r_1 r_2}) - \frac{1}{2(1+\beta r_{12})^2} - \frac{2}{r_{12}} + \frac{2\beta}{1+\beta r_{12}}\right\}$$

It is very useful to test your code against these expressions. It means also that you don't need to compute a derivative numerically as discussed in the code example below.

For the computation of various derivatives with different types of wave functions, you will find it useful to use python with symbolic python, that is sympy, see online manual. Using sympy allows you autogenerate both Latex code as well c++, python or Fortran codes. Here you will find some simple examples. We choose the $2s$ hydrogen-orbital (not normalized) as an example

$$\phi_{2s}(\boldsymbol{r}) = (Zr-2)\exp-(\frac{1}{2}Zr),$$

with $r^2 = x^2+y^2+z^2$.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python from sympy import symbols, diff, exp, sqrt x, y, z, Z = symbols('x y z Z') r = sqrt(x*x + y*y + z*z) r phi = (Z*r - 2)*exp(-Z*r/2) phi diff(phi, x)

This doesn't look very nice, but sympy provides several functions that allow for improving and simplifying the output.

We can improve our output by factorizing and substituting expressions

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python from sympy import symbols, diff, exp, sqrt, factor, Symbol, printing x, y, z, Z = symbols('x y z Z') r = sqrt(x*x + y*y + z*z) phi = (Z*r - 2)*exp(-Z*r/2) R = Symbol('r') Creates a symbolic equivalent of r print latex and c++ code print printing.latex(diff(phi, x).factor().subs(r, R)) print printing.ccode(diff(phi, x).factor().subs(r, R))

We can in turn look at second derivatives

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python from sympy import symbols, diff, exp, sqrt, factor, Symbol, printing x, y, z, Z = symbols('x y z Z') r = sqrt(x*x + y*y + z*z) phi = (Z*r - 2)*exp(-Z*r/2) R = Symbol('r') Creates a symbolic equivalent of r (diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).factor().subs(r, R) Collect the Z values (diff(diff(phi, x), x) + diff(diff(phi, y), y) +diff(diff(phi, z), z)).factor().collect(Z).subs(r, R) Factorize also the r**2 terms (diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).factor().collect(Z).subs(r, R).subs(r**2, R**2).factor() print printing.ccode((diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).factor().collect(Z).subs(r, R).subs(r**2, R**2).factor())

With some practice this allows one to be able to check one's own calculation and translate automatically into code lines.

The Metropolis algorithm

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 $\mathbf{P}_i^{(n\to\infty)} \to p_i$ 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:

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

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

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

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 $A_{i\to k} \leq 1$, the elements of the matrix satisfy $M_{ij} \geq 0$. The matrix M is therefore a stochastic matrix.

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

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.

Importance sampling

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

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

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 as well as Hasting's article,

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

Importance sampling, program elements

The general derivative formula of the Jastrow factor is (the subscript C stands for Correlation)

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_k}$$

However, with our written in way which can be reused later as

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \exp\left\{\sum_{i<j} f(r_{ij})\right\},$$

the gradient needed for the quantum force and local energy is easy to compute. The function $f(r_{ij})$ will depends on the system under study. In the equations below we will keep this general form.

In the Metropolis/Hasting algorithm, the *acceptance ratio* determines the probability for a particle to be accepted at a new position. The ratio of the trial wave functions evaluated at the new and current positions is given by (*OB* for the onebody part)

$$R \equiv \frac{\Psi_T^{new}}{\Psi_T^{old}} = \frac{\Psi_{OB}^{new}}{\Psi_{OB}^{old}}\frac{\Psi_C^{new}}{\Psi_C^{old}}$$

Here Ψ_{OB} is our onebody part (Slater determinant or product of boson single-particle states) while Ψ_C is our correlation function, or Jastrow factor. We need to optimize the $\nabla\Psi_T/\Psi_T$ ratio and the second derivative as well, that is the $\nabla^2\Psi_T/\Psi_T$ ratio. The first is needed when we

compute the so-called quantum force in importance sampling. The second is needed when we compute the kinetic energy term of the local energy.

$$\frac{\nabla\Psi}{\Psi} = \frac{\nabla(\Psi_{OB}\Psi_C)}{\Psi_{OB}\Psi_C} = \frac{\Psi_C\nabla\Psi_{OB} + \Psi_{OB}\nabla\Psi_C}{\Psi_{OB}\Psi_C} = \frac{\nabla\Psi_{OB}}{\Psi_{OB}} + \frac{\nabla\Psi_C}{\Psi_C}$$

The expectation value of the kinetic energy expressed in atomic units for electron i is

$$\langle\hat{K}_i\rangle = -\frac{1}{2}\frac{\langle\Psi|\nabla_i^2|\Psi\rangle}{\langle\Psi|\Psi\rangle},$$

$$\hat{K}_i = -\frac{1}{2}\frac{\nabla_i^2\Psi}{\Psi}.$$

The second derivative which enters the definition of the local energy is

$$\frac{\nabla^2\Psi}{\Psi} = \frac{\nabla^2\Psi_{OB}}{\Psi_{OB}} + \frac{\nabla^2\Psi_C}{\Psi_C} + 2\frac{\nabla\Psi_{OB}}{\Psi_{OB}}\cdot\frac{\nabla\Psi_C}{\Psi_C}$$

We discuss here how to calculate these quantities in an optimal way,

We have defined the correlated function as

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \prod_{i<j}^{N} g(r_{ij}) = \prod_{i=1}^{N}\prod_{j=i+1}^{N} g(r_{ij}),$$

with $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j| = \sqrt{(x_i-x_j)^2+(y_i-y_j)^2+(z_i-z_j)^2}$ in three dimensions or $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j| = \sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$ if we work with two-dimensional systems.

In our particular case we have

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \exp\left\{\sum_{i<j} f(r_{ij})\right\}.$$

The total number of different relative distances r_{ij} is $N(N-1)/2$. In a matrix storage format, the relative distances form a strictly upper triangular matrix

$$\mathbf{r} \equiv \begin{pmatrix} 0 & r_{1,2} & r_{1,3} & \cdots & r_{1,N} \\ \vdots & 0 & r_{2,3} & \cdots & r_{2,N} \\ \vdots & \vdots & 0 & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & r_{N-1,N} \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}.$$

This applies to $\mathbf{g} = \mathbf{g}(r_{ij})$ as well.

In our algorithm we will move one particle at the time, say the kth-particle. This sampling will be seen to be particularly efficient when we are going to compute a Slater determinant.

We have that the ratio between Jastrow factors R_C is given by

$$R_C = \frac{\Psi_C^{\text{new}}}{\Psi_C^{\text{cur}}} = \prod_{i=1}^{k-1}\frac{g_{ik}^{\text{new}}}{g_{ik}^{\text{cur}}}\prod_{i=k+1}^{N}\frac{g_{ki}^{\text{new}}}{g_{ki}^{\text{cur}}}.$$

For the Pade-Jastrow form

$$R_C = \frac{\Psi_C^{\text{new}}}{\Psi_C^{\text{cur}}} = \frac{\exp U_{new}}{\exp U_{cur}} = \exp\Delta U,$$

where

$$\Delta U = \sum_{i=1}^{k-1} \left(f_{ik}^{\mathrm{new}} - f_{ik}^{\mathrm{cur}}\right) + \sum_{i=k+1}^{N} \left(f_{ki}^{\mathrm{new}} - f_{ki}^{\mathrm{cur}}\right)$$

One needs to develop a special algorithm that runs only through the elements of the upper triangular matrix $\mathbf{g}$ and have k as an index.

The expression to be derived in the following is of interest when computing the quantum force and the kinetic energy. It has the form

$$\frac{\nabla_i \Psi_C}{\Psi_C} = \frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_i},$$

for all dimensions and with i running over all particles.

For the first derivative only $N-1$ terms survive the ratio because the g-terms that are not differentiated cancel with their corresponding ones in the denominator. Then,

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1} \frac{1}{g_{ik}}\frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N} \frac{1}{g_{ki}}\frac{\partial g_{ki}}{\partial x_k}.$$

An equivalent equation is obtained for the exponential form after replacing g_{ij} by $\exp(f_{ij})$, yielding:

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1} \frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N} \frac{\partial g_{ki}}{\partial x_k},$$

with both expressions scaling as $\mathcal{O}(N)$.

Using the identity

$$\frac{\partial}{\partial x_i} g_{ij} = -\frac{\partial}{\partial x_j} g_{ij},$$

we get expressions where all the derivatives acting on the particle are represented by the *second* index of g:

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1} \frac{1}{g_{ik}}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N} \frac{1}{g_{ki}}\frac{\partial g_{ki}}{\partial x_i},$$

and for the exponential case:

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1} \frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N} \frac{\partial g_{ki}}{\partial x_i}.$$

For correlation forms depending only on the scalar distances r_{ij} we can use the chain rule. Noting that

$$\frac{\partial g_{ij}}{\partial x_j} = \frac{\partial g_{ij}}{\partial r_{ij}}\frac{\partial r_{ij}}{\partial x_j} = \frac{x_j - x_i}{r_{ij}}\frac{\partial g_{ij}}{\partial r_{ij}},$$

we arrive at

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1} \frac{1}{g_{ik}}\frac{\mathbf{r_{ik}}}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N} \frac{1}{g_{ki}}\frac{\mathbf{r_{ki}}}{r_{ki}}\frac{\partial g_{ki}}{\partial r_{ki}}.$$

Note that for the Pade-Jastrow form we can set $g_{ij} \equiv g(r_{ij}) = e^{f(r_{ij})} = e^{f_{ij}}$ and

$$\frac{\partial g_{ij}}{\partial r_{ij}} = g_{ij}\frac{\partial f_{ij}}{\partial r_{ij}}.$$

Therefore,

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1} \frac{\mathbf{r_{ik}}}{r_{ik}}\frac{\partial f_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N} \frac{\mathbf{r_{ki}}}{r_{ki}}\frac{\partial f_{ki}}{\partial r_{ki}},$$

where

$$\mathbf{r}_{ij} = |\mathbf{r}_j - \mathbf{r}_i| = (x_j - x_i)\mathbf{e}_1 + (y_j - y_i)\mathbf{e}_2 + (z_j - z_i)\mathbf{e}_3$$

is the relative distance.

The second derivative of the Jastrow factor divided by the Jastrow factor (the way it enters the kinetic energy) is

$$\left[\frac{\nabla^2\Psi_C}{\Psi_C}\right]_x = 2\sum_{k=1}^{N}\sum_{i=1}^{k-1}\frac{\partial^2 g_{ik}}{\partial x_k^2} + \sum_{k=1}^{N}\left(\sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}\right)^2$$

But we have a simple form for the function, namely

$$\Psi_C = \prod_{i<j}\exp f(r_{ij}),$$

and it is easy to see that for particle k we have

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k - \mathbf{r}_i)(\mathbf{r}_k - \mathbf{r}_j)}{r_{ki}r_{kj}}f'(r_{ki})f'(r_{kj}) + \sum_{j\neq k}\left(f''(r_{kj}) + \frac{2}{r_{kj}}f'(r_{kj})\right)$$

Importance sampling, Fokker-Planck and Langevin equations

A stochastic process is simply a function of two variables, one is the time, the other is a stochastic variable X, defined by specifying

- the set $\{x\}$ of possible values for X;
- the probability distribution, $w_X(x)$, over this set, or briefly $w(x)$

The set of values $\{x\}$ for X may be discrete, or continuous. If the set of values is continuous, then $w_X(x)$ is a probability density so that $w_X(x)dx$ is the probability that one finds the stochastic variable X to have values in the range $[x, x+dx]$.

An arbitrary number of other stochastic variables may be derived from X. For example, any Y given by a mapping of X, is also a stochastic variable. The mapping may also be time-dependent, that is, the mapping depends on an additional variable t

$$Y_X(t) = f(X,t).$$

The quantity $Y_X(t)$ is called a random function, or, since t often is time, a stochastic process. A stochastic process is a function of two variables, one is the time, the other is a stochastic variable X. Let x be one of the possible values of X then

$$y(t) = f(x,t),$$

is a function of t, called a sample function or realization of the process. In physics one considers the stochastic process to be an ensemble of such sample functions.

For many physical systems initial distributions of a stochastic variable y tend to equilibrium distributions: $w(y,t) \to w_0(y)$ as $t \to \infty$. In equilibrium detailed balance constrains the transition rates

$$W(y \to y')w(y) = W(y' \to y)w_0(y),$$

where $W(y' \to y)$ is the probability, per unit time, that the system changes from a state $|y\rangle$, characterized by the value y for the stochastic variable Y , to a state $|y'\rangle$.

Note that for a system in equilibrium the transition rate $W(y' \rightarrow y)$ and the reverse $W(y \rightarrow y')$ may be very different.

Consider, for instance, a simple system that has only two energy levels $\varepsilon_0 = 0$ and $\varepsilon_1 = \Delta E$.

For a system governed by the Boltzmann distribution we find (the partition function has been taken out)

$$W(0 \rightarrow 1) \exp -(\varepsilon_0/kT) = W(1 \rightarrow 0) \exp -(\varepsilon_1/kT)$$

We get then

$$\frac{W(1 \rightarrow 0)}{W(0 \rightarrow 1)} = \exp -(\Delta E/kT),$$

which goes to zero when T tends to zero.

If we assume a discrete set of events, our initial probability distribution function can be given by

$$w_i(0) = \delta_{i,0},$$

and its time-development after a given time step $\Delta t = \varepsilon$ is

$$w_i(t) = \sum_j W(j \rightarrow i) w_j(t = 0).$$

The continuous analog to $w_i(0)$ is

$$w(\mathbf{x}) \rightarrow \delta(\mathbf{x}),$$

where we now have generalized the one-dimensional position x to a generic-dimensional vector $\mathbf{x}$. The Kroenecker δ function is replaced by the δ distribution function $\delta(\mathbf{x})$ at $t = 0$.

The transition from a state j to a state i is now replaced by a transition to a state with position $\mathbf{y}$ from a state with position $\mathbf{x}$. The discrete sum of transition probabilities can then be replaced by an integral and we obtain the new distribution at a time $t + \Delta t$ as

$$w(\mathbf{y}, t + \Delta t) = \int W(\mathbf{y}, t + \Delta t | \mathbf{x}, t) w(\mathbf{x}, t) d\mathbf{x},$$

and after m time steps we have

$$w(\mathbf{y}, t + m\Delta t) = \int W(\mathbf{y}, t + m\Delta t | \mathbf{x}, t) w(\mathbf{x}, t) d\mathbf{x}.$$

When equilibrium is reached we have

$$w(\mathbf{y}) = \int W(\mathbf{y} | \mathbf{x}, t) w(\mathbf{x}) d\mathbf{x},$$

that is no time-dependence. Note our change of notation for W

We can solve the equation for $w(\mathbf{y}, t)$ by making a Fourier transform to momentum space. The PDF $w(\mathbf{x}, t)$ is related to its Fourier transform $\tilde{w}(\mathbf{k}, t)$ through

$$w(\mathbf{x}, t) = \int_{-\infty}^{\infty} d\mathbf{k} \exp(i\mathbf{k}\mathbf{x}) \tilde{w}(\mathbf{k}, t),$$

and using the definition of the δ-function

$$\delta(\mathbf{x}) = \frac{1}{2\pi} \int_{-\infty}^{\infty} d\mathbf{k} \exp(i\mathbf{k}\mathbf{x}),$$

we see that

$$\tilde{w}(\mathbf{k}, 0) = 1/2\pi.$$

We can then use the Fourier-transformed diffusion equation

$$\frac{\partial \tilde{w}(\mathbf{k},t)}{\partial t} = -D\mathbf{k}^2\tilde{w}(\mathbf{k},t),$$

with the obvious solution

$$\tilde{w}(\mathbf{k},t) = \tilde{w}(\mathbf{k},0)\exp\left[-(D\mathbf{k}^2t)\right) = \frac{1}{2\pi}\exp\left[-(D\mathbf{k}^2t)\right].$$

With the Fourier transform we obtain

$$w(\mathbf{x},t) = \int_{-\infty}^{\infty} d\mathbf{k}\exp\left[i\mathbf{k}\mathbf{x}\right]\frac{1}{2\pi}\exp\left[-(D\mathbf{k}^2t)\right] = \frac{1}{\sqrt{4\pi Dt}}\exp\left[-(\mathbf{x}^2/4Dt)\right],$$

with the normalization condition

$$\int_{-\infty}^{\infty} w(\mathbf{x},t)d\mathbf{x} = 1.$$

The solution represents the probability of finding our random walker at position $\mathbf{x}$ at time t if the initial distribution was placed at $\mathbf{x} = 0$ at $t = 0$.

There is another interesting feature worth observing. The discrete transition probability W itself is given by a binomial distribution. The results from the central limit theorem state that transition probability in the limit $n \to \infty$ converges to the normal distribution. It is then possible to show that

$$W(il - jl, n\varepsilon) \to W(\mathbf{y},t+\Delta t|\mathbf{x},t) = \frac{1}{\sqrt{4\pi D\Delta t}}\exp\left[-((\mathbf{y}-\mathbf{x})^2/4D\Delta t)\right],$$

and that it satisfies the normalization condition and is itself a solution to the diffusion equation.

Let us now assume that we have three PDFs for times $t_0 < t' < t$, that is $w(\mathbf{x}_0,t_0)$, $w(\mathbf{x}',t')$ and $w(\mathbf{x},t)$. We have then

$$w(\mathbf{x},t) = \int_{-\infty}^{\infty} W(\mathbf{x}.t|\mathbf{x}'.t')w(\mathbf{x}',t')d\mathbf{x}',$$

and

$$w(\mathbf{x},t) = \int_{-\infty}^{\infty} W(\mathbf{x}.t|\mathbf{x}_0.t_0)w(\mathbf{x}_0,t_0)d\mathbf{x}_0,$$

and

$$w(\mathbf{x}',t') = \int_{-\infty}^{\infty} W(\mathbf{x}'.t'|\mathbf{x}_0,t_0)w(\mathbf{x}_0,t_0)d\mathbf{x}_0.$$

We can combine these equations and arrive at the famous Einstein-Smoluchenski-Kolmogorov-Chapman (ESKC) relation

$$W(\mathbf{x}t|\mathbf{x}_0t_0) = \int_{-\infty}^{\infty} W(\mathbf{x},t|\mathbf{x}',t')W(\mathbf{x}',t'|\mathbf{x}_0,t_0)d\mathbf{x}'.$$

We can replace the spatial dependence with a dependence upon say the velocity (or momentum), that is we have

$$W(\mathbf{v},t|\mathbf{v}_0,t_0) = \int_{-\infty}^{\infty} W(\mathbf{v},t|\mathbf{v}',t')W(\mathbf{v}',t'|\mathbf{v}_0,t_0)d\mathbf{x}'.$$

We will now derive the Fokker-Planck equation. We start from the ESKC equation

$$W(\mathbf{x},t|\mathbf{x}_0,t_0) = \int_{-\infty}^{\infty} W(\mathbf{x},t|\mathbf{x}',t')W(\mathbf{x}',t'|\mathbf{x}_0,t_0)d\mathbf{x}'.$$

Define $s = t' - t_0$, $\tau = t - t'$ and $t - t_0 = s + \tau$. We have then

$$W(\mathbf{x},s+\tau|\mathbf{x}_0)=\int_{-\infty}^{\infty}W(\mathbf{x},\tau|\mathbf{x}')W(\mathbf{x}',s|\mathbf{x}_0)d\mathbf{x}'.$$

Assume now that τ is very small so that we can make an expansion in terms of a small step *xi*, with $\mathbf{x}'=\mathbf{x}-\xi$, that is

$$W(\mathbf{x},s|\mathbf{x}_0)+\frac{\partial W}{\partial s}\tau+O(\tau^2)=\int_{-\infty}^{\infty}W(\mathbf{x},\tau|\mathbf{x}-\xi)W(\mathbf{x}-\xi,s|\mathbf{x}_0)d\mathbf{x}'.$$

We assume that $W(\mathbf{x},\tau|\mathbf{x}-\xi)$ takes non-negligible values only when ξ is small. This is just another way of stating the Master equation!!

We say thus that $\mathbf{x}$ changes only by a small amount in the time interval τ. This means that we can make a Taylor expansion in terms of ξ, that is we expand

$$W(\mathbf{x},\tau|\mathbf{x}-\xi)W(\mathbf{x}-\xi,s|\mathbf{x}_0)=\sum_{n=0}^{\infty}\frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x}+\xi,\tau|\mathbf{x})W(\mathbf{x},s|\mathbf{x}_0)\right].$$

We can then rewrite the ESKC equation as

$$\frac{\partial W}{\partial s}\tau=-W(\mathbf{x},s|\mathbf{x}_0)+\sum_{n=0}^{\infty}\frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x},s|\mathbf{x}_0)\int_{-\infty}^{\infty}\xi^nW(\mathbf{x}+\xi,\tau|\mathbf{x})d\xi\right].$$

We have neglected higher powers of τ and have used that for $n=0$ we get simply $W(\mathbf{x},s|\mathbf{x}_0)$ due to normalization.

We say thus that $\mathbf{x}$ changes only by a small amount in the time interval τ. This means that we can make a Taylor expansion in terms of ξ, that is we expand

$$W(\mathbf{x},\tau|\mathbf{x}-\xi)W(\mathbf{x}-\xi,s|\mathbf{x}_0)=\sum_{n=0}^{\infty}\frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x}+\xi,\tau|\mathbf{x})W(\mathbf{x},s|\mathbf{x}_0)\right].$$

We can then rewrite the ESKC equation as

$$\frac{\partial W(\mathbf{x},s|\mathbf{x}_0)}{\partial s}\tau=-W(\mathbf{x},s|\mathbf{x}_0)+\sum_{n=0}^{\infty}\frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x},s|\mathbf{x}_0)\int_{-\infty}^{\infty}\xi^nW(\mathbf{x}+\xi,\tau|\mathbf{x})d\xi\right].$$

We have neglected higher powers of τ and have used that for $n=0$ we get simply $W(\mathbf{x},s|\mathbf{x}_0)$ due to normalization.

We simplify the above by introducing the moments

$$M_n=\frac{1}{\tau}\int_{-\infty}^{\infty}\xi^nW(\mathbf{x}+\xi,\tau|\mathbf{x})d\xi=\frac{\langle[\Delta x(\tau)]^n\rangle}{\tau},$$

resulting in

$$\frac{\partial W(\mathbf{x},s|\mathbf{x}_0)}{\partial s}=\sum_{n=1}^{\infty}\frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x},s|\mathbf{x}_0)M_n\right].$$

When $\tau\to 0$ we assume that $\langle[\Delta x(\tau)]^n\rangle\to 0$ more rapidly than τ itself if $n>2$. When τ is much larger than the standard correlation time of system then M_n for $n>2$ can normally be neglected. This means that fluctuations become negligible at large time scales.

If we neglect such terms we can rewrite the ESKC equation as

$$\frac{\partial W(\mathbf{x},s|\mathbf{x}_0)}{\partial s}=-\frac{\partial M_1W(\mathbf{x},s|\mathbf{x}_0)}{\partial x}+\frac{1}{2}\frac{\partial^2M_2W(\mathbf{x},s|\mathbf{x}_0)}{\partial x^2}.$$

In a more compact form we have

$$\frac{\partial W}{\partial s} = -\frac{\partial M_1 W}{\partial x} + \frac{1}{2}\frac{\partial^2 M_2 W}{\partial x^2},$$

which is the Fokker-Planck equation! It is trivial to replace position with velocity (momentum).

Consider a particle suspended in a liquid. On its path through the liquid it will continuously collide with the liquid molecules. Because on average the particle will collide more often on the front side than on the back side, it will experience a systematic force proportional with its velocity, and directed opposite to its velocity. Besides this systematic force the particle will experience a stochastic force $\mathbf{F}(t)$. The equations of motion are

- $\frac{d\mathbf{r}}{dt} = \mathbf{v}$ and
- $\frac{d\mathbf{v}}{dt} = -\xi\mathbf{v} + \mathbf{F}$.

From hydrodynamics we know that the friction constant ξ is given by

$$\xi = 6\pi\eta a/m$$

where η is the viscosity of the solvent and a is the radius of the particle .

Solving the second equation in the previous slide we get

$$\mathbf{v}(t) = \mathbf{v}_0 e^{-\xi t} + \int_0^t d\tau e^{-\xi(t-\tau)}\mathbf{F}(\tau).$$

If we want to get some useful information out of this, we have to average over all possible realizations of $\mathbf{F}(t)$, with the initial velocity as a condition. A useful quantity for example is

$$\langle \mathbf{v}(t)\cdot\mathbf{v}(t)\rangle_{\mathbf{v}_0} = v_0^{-\xi 2t} + 2\int_0^t d\tau e^{-\xi(2t-\tau)}\mathbf{v}_0\cdot\langle\mathbf{F}(\tau)\rangle_{\mathbf{v}_0}$$

$$+ \int_0^t d\tau' \int_0^t d\tau e^{-\xi(2t-\tau-\tau')}\langle\mathbf{F}(\tau)\cdot\mathbf{F}(\tau')\rangle_{\mathbf{v}_0}.$$

In order to continue we have to make some assumptions about the conditional averages of the stochastic forces. In view of the chaotic character of the stochastic forces the following assumptions seem to be appropriate

$$\langle\mathbf{F}(t)\rangle = 0,$$

and

$$\langle\mathbf{F}(t)\cdot\mathbf{F}(t')\rangle_{\mathbf{v}_0} = C_{\mathbf{v}_0}\delta(t-t').$$

We omit the subscript $\mathbf{v}_0$, when the quantity of interest turns out to be independent of $\mathbf{v}_0$. Using the last three equations we get

$$\langle \mathbf{v}(t)\cdot\mathbf{v}(t)\rangle_{\mathbf{v}_0} = v_0^2 e^{-2\xi t} + \frac{C_{\mathbf{v}_0}}{2\xi}(1-e^{-2\xi t}).$$

For large t this should be equal to 3kT/m, from which it follows that

$$\langle\mathbf{F}(t)\cdot\mathbf{F}(t')\rangle = 6\frac{kT}{m}\xi\delta(t-t').$$

This result is called the fluctuation-dissipation theorem .

Integrating

$$\mathbf{v}(t) = \mathbf{v}_0 e^{-\xi t} + \int_0^t d\tau e^{-\xi(t-\tau)}\mathbf{F}(\tau),$$

we get

$$\mathbf{r}(t) = \mathbf{r}_0 + \mathbf{v}_0\frac{1}{\xi}(1-e^{-\xi t}) + \int_0^t d\tau\int_0^\tau \tau' e^{-\xi(\tau-\tau')}\mathbf{F}(\tau'),$$

from which we calculate the mean square displacement

$$\langle(\mathbf{r}(t)-\mathbf{r}_0)^2\rangle_{\mathbf{v}_0}=\frac{v_0^2}{\xi}(1-e^{-\xi t})^2+\frac{3kT}{m\xi^2}(2\xi t-3+4e^{-\xi t}-e^{-2\xi t}).$$

For very large t this becomes

$$\langle(\mathbf{r}(t)-\mathbf{r}_0)^2\rangle=\frac{6kT}{m\xi}t$$

from which we get the Einstein relation

$$D=\frac{kT}{m\xi}$$

where we have used $\langle(\mathbf{r}(t)-\mathbf{r}_0)^2\rangle=6Dt$.

Code example for two electrons in a quantum dots

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python 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$_t oolkits.mplot3dimportAxes3Dfrommatplotlibimportcmfrommatplotlib.tickerimportLinearLocator,Form$

Read name of output file from command line if len(sys.argv) == 2: outfilename = sys.argv[1] else: print(': Name of output file must be given as command line argument.') outfile = open(outfilename,'w')

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-1.0/r12)

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 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('return Energies, AlphaValues, BetaValues

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$_{surface}(X,Y,Energies,cmap=cm.coolwarm,linewidth=0,antialiased=False)Customizethezaxis.zmin=np.matrix(Energies).min()zmax=np.matrix(Energies).max()ax.set_zlim(zmin,zmax)ax.set_xlabel(r'\alpha')$ ax.set$_{y}label(r'\beta')$ ax.set$_{z}label(r'\langle E \rangle')$ ax.zaxis.set$_{m}ajor_{l}ocator(LinearLocator(10))ax.zaxis.set_{m}ajor_{f}ormatter(FormatStrFormatter('Addacolorbarwhichm$ $0.5,aspect=5)plt.show()$

Bringing the gradient optmization.

The simple one-particle case in a harmonic oscillator trap

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python Gradient descent stepping with analytical derivative import numpy as np from scipy.optimize import minimize def DerivativeE(x): return x-1.0/(4*x*x*x);

def Energy(x): return x*x*0.5+1.0/(8*x*x); x0 = 1.0 eta = 0.1 Niterations = 100

for iter in range(Niterations): gradients = DerivativeE(x0) x0 -= eta*gradients

print(x0)

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python 2-electron VMC code for 2dim quantum dot with importance sampling Using gaussian rng for new positions and Metropolis- Hastings from math import exp, sqrt from random import random, seed, normalvariate import numpy as np import matplotlib.pyplot as plt from mpl$_{toolkits.mplot3dimportAxes3Dfrommatplotlibimportcmfrommatplotlib.tickerimportLinearLocator,FormatStrFormatterimportsysfromm$

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

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

r1 = (r[0,0]**2 + r[0,1]**2) r2 = (r[1,0]**2 + r[1,1]**2) return 0.5*(1-alpha*alpha)*(r1 + r2) +2.0*alpha

Derivate of wave function ansatz as function of variational parameters def DerivativeWFansatz(r,alpha):

r1 = (r[0,0]**2 + r[0,1]**2) r2 = (r[1,0]**2 + r[1,1]**2) WfDer = -(r1+r2) return WfDer

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

qforce = np.zeros((NumberParticles,Dimension), np.double) qforce[0,:] = -2*r[0,:]*alpha qforce[1,:] = -2*r[1,:]*alpha return qforce

Computing the derivative of the energy and the energy jit decorator tells Numba to compile this function. The argument types will be inferred by Numba when function is called. @jit def EnergyMinimization(alpha):

NumberMCcycles= 1000 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() energy = 0.0 DeltaE = 0.0 EnergyDer = 0.0 DeltaPsi = 0.0 DerivativePsiE = 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) QuantumForceOld = QuantumForce(PositionOld,alpha)

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) QuantumForceNew = QuantumForce(PositionNew,alpha) 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) DeltaPsi = DerivativeWFansatz(PositionOld,alpha) energy += DeltaE DerivativePsiE += DeltaPsi*DeltaE

We calculate mean, variance and error (no blocking applied) energy /= NumberMCcycles DerivativePsiE /= NumberMCcycles DeltaPsi /= NumberMCcycles EnergyDer = 2*(DerivativePsiE- DeltaPsi*energy) return energy, EnergyDer

Here starts the main program with variable declarations NumberParticles = 2 Dimension = 2 guess for variational parameters x0 = 1.5 Set up iteration using stochastic gradient method Energy =0 ; EnergyDer = 0 Energy, EnergyDer = EnergyMinimization(x0) print(Energy, EnergyDer)

eta = 0.01 Niterations = 100

for iter in range(Niterations): gradients = EnergyDer x0 -= eta*gradients Energy, EnergyDer = EnergyMinimization(x0)

print(x0)

VMC for fermions: Efficient calculation of Slater determinants

The potentially most time-consuming part is the evaluation of the gradient and the Laplacian of an N-particle Slater determinant.

We have to differentiate the determinant with respect to all spatial coordinates of all particles. A brute force differentiation would involve $N \cdot d$ evaluations of the entire determinant which would even worsen the already undesirable time scaling, making it $Nd \cdot O(N^3) \sim O(d \cdot N^4)$.

This poses serious hindrances to the overall efficiency of our code.

The efficiency can be improved however if we move only one electron at the time. The Slater determinant matrix $\hat{D}$ is defined by the matrix elements

$$d_{ij} = \phi_j(x_i)$$

where $\phi_j(\mathbf{r}_i)$ is a single particle wave function. The columns correspond to the position of a given particle while the rows stand for the various quantum numbers.

What we need to realize is that when differentiating a Slater determinant with respect to some given coordinate, only one row of the corresponding Slater matrix is changed.

Therefore, by recalculating the whole determinant we risk producing redundant information. The solution turns out to be an algorithm that requires to keep track of the *inverse* of the Slater matrix.

Let the current position in phase space be represented by the $(N \cdot d)$-element vector $\mathbf{r}^{\text{old}}$ and the new suggested position by the vector $\mathbf{r}^{\text{new}}$.

The inverse of $\hat{D}$ can be expressed in terms of its cofactors C_{ij} and its determinant (this our notation for a determinant) $|\hat{D}|$:

$$d_{ij}^{-1} = \frac{C_{ji}}{|\hat{D}|} \tag{1.7}$$

Notice that the interchanged indices indicate that the matrix of cofactors is to be transposed.

If $\hat{D}$ is invertible, then we must obviously have $\hat{D}^{-1}\hat{D} = \mathbf{1}$, or explicitly in terms of the individual elements of $\hat{D}$ and $\hat{D}^{-1}$:

$$\sum_{k=1}^{N} d_{ik} d_{kj}^{-1} = \delta_{ij} \tag{1.8}$$

Consider the ratio, which we shall call R, between $|\hat{D}(\mathbf{r}^{\text{new}})|$ and $|\hat{D}(\mathbf{r}^{\text{old}})|$. By definition, each of these determinants can individually be expressed in terms of the i-th row of its cofactor matrix

$$R \equiv \frac{|\hat{D}(\mathbf{r}^{\text{new}})|}{|\hat{D}(\mathbf{r}^{\text{old}})|} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}}) C_{ij}(\mathbf{r}^{\text{new}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}}) C_{ij}(\mathbf{r}^{\text{old}})} \tag{1.9}$$

Suppose now that we move only one particle at a time, meaning that $\mathbf{r}^{\text{new}}$ differs from $\mathbf{r}^{\text{old}}$ by the position of only one, say the i-th, particle . This means that $\hat{D}(\mathbf{r}^{\text{new}})$ and $\hat{D}(\mathbf{r}^{\text{old}})$ differ only by the entries of the i-th row. Recall also that the i-th row of a cofactor matrix $\hat{C}$ is independent of the entries of the i-th row of its corresponding matrix $\hat{D}$. In this particular case we therefore get that the i-th row of $\hat{C}(\mathbf{r}^{\text{new}})$ and $\hat{C}(\mathbf{r}^{\text{old}})$ must be equal. Explicitly, we have:

$$C_{ij}(\mathbf{r}^{\text{new}}) = C_{ij}(\mathbf{r}^{\text{old}}) \quad \forall\ j \in \{1, \ldots, N\} \tag{1.10}$$

Inserting this into the numerator of eq. (**??**) and using eq. (**??**) to substitute the cofactors with the elements of the inverse matrix, we get:

$$R = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}}) C_{ij}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}}) C_{ij}(\mathbf{r}^{\text{old}})} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}})} \tag{1.11}$$

Now by eq. (**??**) the denominator of the rightmost expression must be unity, so that we finally arrive at:

$$R = \sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}}) = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}}) \tag{1.12}$$

What this means is that in order to get the ratio when only the i-th particle has been moved, we only need to calculate the dot product of the vector $(\phi_1(\mathbf{r}_i^{\text{new}}), \ldots, \phi_N(\mathbf{r}_i^{\text{new}}))$ of single particle wave functions evaluated at this new position with the i-th column of the inverse matrix $\hat{D}^{-1}$

evaluated at the original position. Such an operation has a time scaling of $O(N)$. The only extra thing we need to do is to maintain the inverse matrix $\hat{D}^{-1}(\mathbf{x}^{\rm old})$.

If the new position $\mathbf{r}^{\rm new}$ is accepted, then the inverse matrix can by suitably updated by an algorithm having a time scaling of $O(N^2)$. This algorithm goes as follows. First we update all but the i-th column of $\hat{D}^{-1}$. For each column $j \neq i$, we first calculate the quantity:

$$S_j = (\hat{D}(\mathbf{r}^{\rm new}) \times \hat{D}^{-1}(\mathbf{r}^{\rm old}))_{ij} = \sum_{l=1}^{N} d_{il}(\mathbf{r}^{\rm new})\, d_{lj}^{-1}(\mathbf{r}^{\rm old}) \tag{1.13}$$

The new elements of the j-th column of $\hat{D}^{-1}$ are then given by:

$$d_{kj}^{-1}(\mathbf{r}^{\rm new}) = d_{kj}^{-1}(\mathbf{r}^{\rm old}) - \frac{S_j}{R}\, d_{ki}^{-1}(\mathbf{r}^{\rm old}) \quad \begin{array}{l} \forall \;\; k \in \{1,\ldots,N\} \\ j \neq i \end{array} \tag{1.14}$$

Finally the elements of the i-th column of $\hat{D}^{-1}$ are updated simply as follows:

$$d_{ki}^{-1}(\mathbf{r}^{\rm new}) = \frac{1}{R}\, d_{ki}^{-1}(\mathbf{r}^{\rm old}) \quad \forall \;\; k \in \{1,\ldots,N\} \tag{1.15}$$

We see from these formulas that the time scaling of an update of $\hat{D}^{-1}$ after changing one row of $\hat{D}$ is $O(N^2)$.

The scheme is also applicable for the calculation of the ratios involving derivatives. It turns out that differentiating the Slater determinant with respect to the coordinates of a single particle $\mathbf{r}_i$ changes only the i-th row of the corresponding Slater matrix.

The gradient and the Laplacian.

The gradient and the Laplacian can therefore be calculated as follows:

$$\frac{\vec{\nabla}_i |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \vec{\nabla}_i d_{ij}(\mathbf{r}) d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \vec{\nabla}_i \phi_j(\mathbf{r}_i) d_{ji}^{-1}(\mathbf{r})$$

and

$$\frac{\nabla_i^2 |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \nabla_i^2 d_{ij}(\mathbf{r}) d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \nabla_i^2 \phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r})$$

Thus, to calculate all the derivatives of the Slater determinant, we only need the derivatives of the single particle wave functions ($\vec{\nabla}_i \phi_j(\mathbf{r}_i)$ and $\nabla_i^2 \phi_j(\mathbf{r}_i)$) and the elements of the corresponding inverse Slater matrix ($\hat{D}^{-1}(\mathbf{r}_i)$). A calculation of a single derivative is by the above result an $O(N)$ operation. Since there are $d \cdot N$ derivatives, the time scaling of the total evaluation becomes $O(d \cdot N^2)$. With an $O(N^2)$ updating algorithm for the inverse matrix, the total scaling is no worse, which is far better than the brute force approach yielding $O(d \cdot N^4)$.

Important note: In most cases you end with closed form expressions for the single-particle wave functions. It is then useful to calculate the various derivatives and make separate functions for them.

The Slater determinant takes the form

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) = \frac{1}{\sqrt{4!}} \begin{vmatrix} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) & \psi_{100\uparrow}(\mathbf{r}_3) & \psi_{100\uparrow}(\mathbf{r}_4) \\ \psi_{100\downarrow}(\mathbf{r}_1) & \psi_{100\downarrow}(\mathbf{r}_2) & \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) & \psi_{200\uparrow}(\mathbf{r}_3) & \psi_{200\uparrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_1) & \psi_{200\downarrow}(\mathbf{r}_2) & \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{vmatrix}.$$

The Slater determinant as written is zero since the spatial wave functions for the spin up and spin down states are equal. But we can rewrite it as the product of two Slater determinants, one for spin up and one for spin down.

We can rewrite it as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) = \det\uparrow(1,2)\det\downarrow(3,4) - \det\uparrow(1,3)\det\downarrow(2,4)$$
$$-\det\uparrow(1,4)\det\downarrow(3,2) + \det\uparrow(2,3)\det\downarrow(1,4) - \det\uparrow(2,4)\det\downarrow(1,3)$$
$$+\det\uparrow(3,4)\det\downarrow(1,2),$$

where we have defined

$$\det\uparrow(1,2) = \frac{1}{\sqrt{2}}\begin{vmatrix}\psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2)\\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2)\end{vmatrix},$$

and

$$\det\downarrow(3,4) = \frac{1}{\sqrt{2}}\begin{vmatrix}\psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4)\\ \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4)\end{vmatrix}.$$

We want to avoid to sum over spin variables, in particular when the interaction does not depend on spin.

It can be shown, see for example Moskowitz and Kalos, Int. J. Quantum Chem. **20** 1107 (1981), that for the variational energy we can approximate the Slater determinant as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) \propto \det\uparrow(1,2)\det\downarrow(3,4),$$

or more generally as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\dots\mathbf{r}_N) \propto \det\uparrow\det\downarrow,$$

where we have the Slater determinant as the product of a spin up part involving the number of electrons with spin up only (2 for beryllium and 5 for neon) and a spin down part involving the electrons with spin down.

This ansatz is not antisymmetric under the exchange of electrons with opposite spins but it can be shown (show this) that it gives the same expectation value for the energy as the full Slater determinant.

As long as the Hamiltonian is spin independent, the above is correct. It is rather straightforward to see this if you go back to the equations for the energy discussed earlier this semester.

We will thus factorize the full determinant $|\hat{D}|$ into two smaller ones, where each can be identified with $\uparrow$ and $\downarrow$ respectively:

$$|\hat{D}| = |\hat{D}|_\uparrow \cdot |\hat{D}|_\downarrow$$

The combined dimensionality of the two smaller determinants equals the dimensionality of the full determinant. Such a factorization is advantageous in that it makes it possible to perform the calculation of the ratio R and the updating of the inverse matrix separately for $|\hat{D}|_\uparrow$ and $|\hat{D}|_\downarrow$:

$$\frac{|\hat{D}|^{\text{new}}}{|\hat{D}|^{\text{old}}} = \frac{|\hat{D}|^{\text{new}}_\uparrow}{|\hat{D}|^{\text{old}}_\uparrow} \cdot \frac{|\hat{D}|^{\text{new}}_\downarrow}{|\hat{D}|^{\text{old}}_\downarrow}$$

This reduces the calculation time by a constant factor. The maximal time reduction happens in a system of equal numbers of $\uparrow$ and $\downarrow$ particles, so that the two factorized determinants are half the size of the original one.

Consider the case of moving only one particle at a time which originally had the following time scaling for one transition:

$$O_R(N) + O_{\text{inverse}}(N^2)$$

For the factorized determinants one of the two determinants is obviously unaffected by the change so that it cancels from the ratio R.

Therefore, only one determinant of size $N/2$ is involved in each calculation of R and update of the inverse matrix. The scaling of each transition then becomes:

$$O_R(N/2)+O_{\text{inverse}}(N^2/4)$$

and the time scaling when the transitions for all N particles are put together:

$$O_R(N^2/2)+O_{\text{inverse}}(N^3/4)$$

which gives the same reduction as in the case of moving all particles at once.

Computing the ratios discussed above requires that we maintain the inverse of the Slater matrix evaluated at the current position. Each time a trial position is accepted, the row number i of the Slater matrix changes and updating its inverse has to be carried out. Getting the inverse of an $N\times N$ matrix by Gaussian elimination has a complexity of order of $\mathcal{O}(N^3)$ operations, a luxury that we cannot afford for each time a particle move is accepted. We will use the expression

$$d_{kj}^{-1}(\mathbf{x}^{\mathbf{new}})=\begin{cases} d_{kj}^{-1}(\mathbf{x}^{\mathbf{old}})-\frac{d_{ki}^{-1}(\mathbf{x}^{\mathbf{old}})}{R}\sum_{l=1}^{N}d_{il}(\mathbf{x}^{\mathbf{new}})d_{lj}^{-1}(\mathbf{x}^{\mathbf{old}}) & \text{if } j\neq i \\ \frac{d_{ki}^{-1}(\mathbf{x}^{\mathbf{old}})}{R}\sum_{l=1}^{N}d_{il}(\mathbf{x}^{\mathbf{old}})d_{lj}^{-1}(\mathbf{x}^{\mathbf{old}}) & \text{if } j=i \end{cases}$$

This equation scales as $O(N^2)$. The evaluation of the determinant of an $N\times N$ matrix by standard Gaussian elimination requires $\mathbf{O}(N^3)$ calculations. As there are Nd independent coordinates we need to evaluate Nd Slater determinants for the gradient (quantum force) and Nd for the Laplacian (kinetic energy). With the updating algorithm we need only to invert the Slater determinant matrix once. This can be done by standard LU decomposition methods.

Expectation value of the kinetic energy.

The expectation value of the kinetic energy expressed in atomic units for electron i is

$$\langle\hat{K}_i\rangle=-\frac{1}{2}\frac{\langle\Psi|\nabla_i^2|\Psi\rangle}{\langle\Psi|\Psi\rangle},$$

$$K_i=-\frac{1}{2}\frac{\nabla_i^2\Psi}{\Psi}. \tag{1.16}$$

$$\begin{aligned}\frac{\nabla^2\Psi}{\Psi}&=\frac{\nabla^2(\Psi_D\Psi_C)}{\Psi_D\Psi_C}=\frac{\nabla\cdot[\nabla(\Psi_D\Psi_C)]}{\Psi_D\Psi_C}=\frac{\nabla\cdot[\Psi_C\nabla\Psi_D+\Psi_D\nabla\Psi_C]}{\Psi_D\Psi_C}\\ &=\frac{\nabla\Psi_C\cdot\nabla\Psi_D+\Psi_C\nabla^2\Psi_D+\nabla\Psi_D\cdot\nabla\Psi_C+\Psi_D\nabla^2\Psi_C}{\Psi_D\Psi_C}\end{aligned} \tag{1.17}$$

$$\frac{\nabla^2\Psi}{\Psi}=\frac{\nabla^2\Psi_D}{\Psi_D}+\frac{\nabla^2\Psi_C}{\Psi_C}+2\frac{\nabla\Psi_D}{\Psi_D}\cdot\frac{\nabla\Psi_C}{\Psi_C} \tag{1.18}$$

The second derivative of the Jastrow factor divided by the Jastrow factor (the way it enters the kinetic energy) is

$$\left[\frac{\nabla^2\Psi_C}{\Psi_C}\right]_x = 2\sum_{k=1}^{N}\sum_{i=1}^{k-1}\frac{\partial^2 g_{ik}}{\partial x_k^2} + \sum_{k=1}^{N}\left(\sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}\right)^2$$

But we have a simple form for the function, namely

$$\Psi_C = \prod_{i<j}\exp f(r_{ij}) = \exp\left\{\sum_{i<j}\frac{ar_{ij}}{1+\beta r_{ij}}\right\},$$

and it is easy to see that for particle k we have

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}f'(r_{ki})f'(r_{kj}) + \sum_{j\neq k}\left(f''(r_{kj}) + \frac{2}{r_{kj}}f'(r_{kj})\right)$$

Using

$$f(r_{ij}) = \frac{ar_{ij}}{1+\beta r_{ij}},$$

and $g'(r_{kj}) = dg(r_{kj})/dr_{kj}$ and $g''(r_{kj}) = d^2g(r_{kj})/dr_{kj}^2$ we find that for particle k we have

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}\frac{a}{(1+\beta r_{ki})^2}\frac{a}{(1+\beta r_{kj})^2} + \sum_{j\neq k}\left(\frac{2a}{r_{kj}(1+\beta r_{kj})^2} - \frac{2a\beta}{(1+\beta r_{kj})^3}\right)$$

The gradient and Laplacian can be calculated as follows:

$$\frac{\nabla_i|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N}\vec{\nabla}_i d_{ij}(\mathbf{r})\,d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N}\vec{\nabla}_i\phi_j(\mathbf{r}_i)\,d_{ji}^{-1}(\mathbf{r})$$

and

$$\frac{\nabla_i^2|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N}\nabla_i^2 d_{ij}(\mathbf{r})\,d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N}\nabla_i^2\phi_j(\mathbf{r}_i)\,d_{ji}^{-1}(\mathbf{r})$$

The gradient for the determinant is

$$\frac{\nabla_i|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N}\nabla_i d_{ij}(\mathbf{r})\,d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N}\nabla_i\phi_j(\mathbf{r}_i)\,d_{ji}^{-1}(\mathbf{r}).$$

We have

$$\Psi_C = \prod_{i<j}g(r_{ij}) = \exp\left\{\sum_{i<j}\frac{ar_{ij}}{1+\beta r_{ij}}\right\},$$

the gradient needed for the quantum force and local energy is easy to compute. We get for particle k

$$\frac{\nabla_k\Psi_C}{\Psi_C} = \sum_{j\neq k}\frac{\mathbf{r}_{kj}}{r_{kj}}\frac{a}{(1+\beta r_{kj})^2},$$

which is rather easy to code. Remember to sum over all particles when you compute the local energy.

We need to compute the ratio between wave functions, in particular for the Slater determinants.

$$R = \sum_{j=1}^{N}d_{ij}(\mathbf{r}^{\text{new}})\,d_{ji}^{-1}(\mathbf{r}^{\text{old}}) = \sum_{j=1}^{N}\phi_j(\mathbf{r}_i^{\text{new}})\,d_{ji}^{-1}(\mathbf{r}^{\text{old}})$$

What this means is that in order to get the ratio when only the i-th particle has been moved, we only need to calculate the dot product of the vector $(\phi_1(\mathbf{r}_i^{\text{new}}),\ldots,\phi_N(\mathbf{r}_i^{\text{new}}))$ of single particle wave functions evaluated at this new position with the i-th column of the inverse matrix $\hat{D}^{-1}$

evaluated at the original position. Such an operation has a time scaling of $O(N)$. The only extra thing we need to do is to maintain the inverse matrix $\hat{D}^{-1}(\mathbf{x}^{\mathrm{old}})$.

Gradient Methods

Top-down start

- We will start with a top-down view, with a simple harmonic oscillator problem in one dimension as case.
- Thereafter we continue with implementing the simplest possible steepest descent approach to our two-electron problem with an electrostatic (Coulomb) interaction. Our code includes also importance sampling. The simple Python code here illustrates the basic elements which need to be included in our own code.
- Then we move on to the mathematical description of various gradient methods.

Motivation

Our aim with this part of the project is to be able to

- find an optimal value for the variational parameters using only some few Monte Carlo cycles
- use these optimal values for the variational parameters to perform a large-scale Monte Carlo calculation

To achieve this will look at methods like *Steepest descent* and the *conjugate gradient method*. Both these methods allow us to find the minima of a multivariable function like our energy (function of several variational parameters). Alternatively, you can always use Newton's method. In particular, since we will normally have one variational parameter, Newton's method can be easily used in finding the minimum of the local energy.

Simple example and demonstration

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

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

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

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

Simple example and demonstration

The derivative of the energy with respect to α gives

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

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

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

The condition

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

gives the optimal $\alpha = 1$, as expected.

*Exercise : Find the local energy for the harmonic oscillator

a) Derive the local energy for the harmonic oscillator in one dimension and find its expectation value.

b) Show also that the optimal value of optimal $\alpha = 1$

c) Repeat the above steps in two dimensions for N bosons or electrons. What is the optimal value of α?

Variance in the simple model

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

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

which yields a second derivative which is always positive.

Computing the derivatives

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. How can we can then obtain the derivatives of the energy with respect to the variational parameters without having to resort to expensive numerical derivations?

Expressions for finding the derivatives of the local energy

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

$$\bar{E}_\alpha = \frac{d\langle E_L[\alpha]\rangle}{d\alpha}.$$

as the derivative of the energy with respect to the variational parameter α (we limit ourselves to one parameter only). In the above example this was easy and we obtain a simple expression for the derivative. We define also the derivative of the trial function (skipping the subindex T) as

$$\bar{\psi}_\alpha = \frac{d\psi[\alpha]\rangle}{d\alpha}.$$

Derivatives of the local energy

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

$$\bar{E}_\alpha = 2\left(\langle\frac{\bar{\psi}_\alpha}{\psi[\alpha]}E_L[\alpha]\rangle - \langle\frac{\bar{\psi}_\alpha}{\psi[\alpha]}\rangle\langle E_L[\alpha]\rangle\right).$$

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

$$\langle\frac{\bar{\psi}_\alpha}{\psi[\alpha]}E_L[\alpha]\rangle,$$

and

$$\langle\frac{\bar{\psi}_\alpha}{\psi[\alpha]}\rangle\langle E_L[\alpha]\rangle$$

*Exercise : General expression for the derivative of the energy

a) Show that

$$\bar{E}_\alpha = 2\left(\langle\frac{\bar{\psi}_\alpha}{\psi[\alpha]}E_L[\alpha]\rangle - \langle\frac{\bar{\psi}_\alpha}{\psi[\alpha]}\rangle\langle E_L[\alpha]\rangle\right).$$

b) Find the corresponding expression for the variance.

Python program for 2-electrons in 2 dimensions

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python 2-electron VMC code for 2dim quantum dot with importance sampling Using gaussian rng for new positions and Metropolis- Hastings Added energy minimization with gradient descent using fixed step size To do: replace with optimization codes from scipy and/or use stochastic gradient descent from math import exp, sqrt from random import random, seed, normalvariate import numpy as np import matplotlib.pyplot as plt from mpl$_t oolkits.mplot3dimportAxes3Dfrommatplotlibimportcmfrommatplotlib.tickerimportLinearLocator, FormatStrFormatterimportsys$

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-1.0/r12)

Derivate of wave function ansatz as function of variational parameters def DerivativeWFansatz(r,alpha,beta):

WfDer = np.zeros((2), np.double) 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 WfDer[0] = -0.5*(r1+r2) WfDer[1] = -r12*r12*deno2 return WfDer

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

Computing the derivative of the energy and the energy def EnergyMinimization(alpha, beta):

NumberMCcycles= 10000 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() energy = 0.0 DeltaE = 0.0 EnergyDer = np.zeros((2), np.double) DeltaPsi = np.zeros((2), np.double) DerivativePsiE = np.zeros((2), np.double) 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) DerPsi = DerivativeWFansatz(PositionOld,alpha,beta) DeltaPsi += DerPsi energy += DeltaE DerivativePsiE += DerPsi*DeltaE

We calculate mean values energy /= NumberMCcycles DerivativePsiE /= NumberMCcycles DeltaPsi /= NumberMCcycles EnergyDer = 2*(DerivativePsiE-DeltaPsi*energy) return energy, EnergyDer

Here starts the main program with variable declarations NumberParticles = 2 Dimension = 2 guess for variational parameters alpha = 0.9 beta = 0.2 Set up iteration using gradient descent method Energy = 0 EDerivative = np.zeros((2), np.double) eta = 0.01 Niterations = 50 for iter in range(Niterations): Energy, EDerivative = EnergyMinimization(alpha,beta)

alphagradient = EDerivative[0] betagradient = EDerivative[1] alpha -= eta*alphagradient beta -= eta*betagradient

print(alpha, beta) print(Energy, EDerivative[0], EDerivative[1])

Using Broyden's algorithm in scipy

The following function uses the above described BFGS algorithm. Here we have defined a function which calculates the energy and a function which computes the first derivative.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python 2-electron VMC code for 2dim quantum dot with importance sampling Using gaussian rng for new positions and Metropolis- Hastings Added energy minimization using the BFGS algorithm, see p. 136 of https://www.springer.com/it/book/9780387303031 from math import exp, sqrt from random import random, seed, normalvariate import numpy as np import matplotlib.pyplot as plt from mpl*toolkits.mplot3dimportAxes3Dfrommatplotlibimportcmfrommatplotlib.tickerimportLinearLocator,Forma

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-1.0/r12)

Derivate of wave function ansatz as function of variational parameters def DerivativeWFansatz(r,alpha,beta):

WfDer = np.zeros((2), np.double) 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 WfDer[0] = -0.5*(r1+r2) WfDer[1] = -r12*r12*deno2 return WfDer

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

Computing the derivative of the energy and the energy def EnergyDerivative(x0):

Parameters in the Fokker-Planck simulation of the quantum force D = 0.5 TimeStep = 0.05 NumberMCcycles= 10000 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)

energy = 0.0 DeltaE = 0.0 alpha = x0[0] beta = x0[1] EnergyDer = 0.0 DeltaPsi = 0.0 DerivativePsiE = 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) DerPsi = DerivativeWFansatz(PositionOld,alpha,beta) DeltaPsi += DerPsi energy += DeltaE DerivativePsiE += DerPsi*DeltaE

We calculate mean values energy /= NumberMCcycles DerivativePsiE /= NumberMCcycles DeltaPsi /= NumberMCcycles EnergyDer = 2*(DerivativePsiE-DeltaPsi*energy) return EnergyDer

Computing the expectation value of the local energy def Energy(x0): 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)

energy = 0.0 DeltaE = 0.0 alpha = x0[0] beta = x0[1] NumberMCcycles= 10000 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

We calculate mean values energy /= NumberMCcycles return energy

Here starts the main program with variable declarations NumberParticles = 2 Dimension = 2 seed for rng generator seed() guess for variational parameters x0 = np.array([0.9,0.2]) Using Broydens method res = minimize(Energy, x0, method='BFGS', jac=EnergyDerivative, options='gtol': 1e-4,'disp': True) print(res.x)

Note that the **minimize** function returns the finale values for the variable $\alpha = x0[0]$ and $\beta = x0[1]$ in the array x.

Brief reminder on Newton-Raphson's method

Let us quickly remind ourselves how we derive the above method.

Perhaps the most celebrated of all one-dimensional root-finding routines is Newton's method, also called the Newton-Raphson method. This method requires the evaluation of both the function f and its derivative f' at arbitrary points. If you can only calculate the derivative numerically and/or your function is not of the smooth type, we normally discourage the use of this method.

The equations

The Newton-Raphson formula consists geometrically of extending the tangent line at a current point until it crosses zero, then setting the next guess to the abscissa of that zero-crossing. The mathematics behind this method is rather simple. Employing a Taylor expansion for x sufficiently close to the solution s, we have

$$f(s)=0=f(x)+(s-x)f'(x)+\frac{(s-x)^2}{2}f''(x)+\dots.$$

For small enough values of the function and for well-behaved functions, the terms beyond linear are unimportant, hence we obtain

$$f(x)+(s-x)f'(x)\approx 0,$$

yielding

$$s\approx x-\frac{f(x)}{f'(x)}.$$

Having in mind an iterative procedure, it is natural to start iterating with

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}.$$

Simple geometric interpretation

The above is Newton-Raphson's method. It has a simple geometric interpretation, namely x_{n+1} is the point where the tangent from $(x_n,f(x_n))$ crosses the x-axis. Close to the solution, Newton-Raphson converges fast to the desired result. However, if we are far from a root, where the higher-order terms in the series are important, the Newton-Raphson formula can give grossly inaccurate results. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function. If an iteration places a trial guess near such a local extremum, so that the first derivative nearly vanishes, then Newton-Raphson may fail totally

Extending to more than one variable

Newton's method can be generalized to systems of several non-linear equations and variables. Consider the case with two equations

$$\begin{aligned}f_1(x_1,x_2) &=0\\ f_2(x_1,x_2) &=0,\end{aligned}$$

which we Taylor expand to obtain

$$\begin{aligned}0&=f_1(x_1+h_1,x_2+h_2)=f_1(x_1,x_2)+h_1\partial f_1/\partial x_1+h_2\partial f_1/\partial x_2+\dots\\ 0&=f_2(x_1+h_1,x_2+h_2)=f_2(x_1,x_2)+h_1\partial f_2/\partial x_1+h_2\partial f_2/\partial x_2+\dots.\end{aligned}$$

Defining the Jacobian matrix $\hat{J}$ we have

$$\hat{J}=\begin{pmatrix}\partial f_1/\partial x_1 & \partial f_1/\partial x_2\\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2\end{pmatrix},$$

we can rephrase Newton's method as

$$\begin{pmatrix} x_1^{n+1} \\ x_2^{n+1} \end{pmatrix} = \begin{pmatrix} x_1^{n} \\ x_2^{n} \end{pmatrix} + \begin{pmatrix} h_1^{n} \\ h_2^{n} \end{pmatrix},$$

where we have defined

$$\begin{pmatrix} h_1^{n} \\ h_2^{n} \end{pmatrix} = -\hat{\mathbf{J}}^{-1} \begin{pmatrix} f_1(x_1^n, x_2^n) \\ f_2(x_1^n, x_2^n) \end{pmatrix}.$$

We need thus to compute the inverse of the Jacobian matrix and it is to understand that difficulties may arise in case $\hat{J}$ is nearly singular.

It is rather straightforward to extend the above scheme to systems of more than two non-linear equations. In our case, the Jacobian matrix is given by the Hessian that represents the second derivative of cost function.

Steepest descent

The basic idea of gradient descent is that a function $F(\mathbf{x})$, $\mathbf{x} \equiv (x_1, \cdots, x_n)$, decreases fastest if one goes from $\mathbf{x}$ in the direction of the negative gradient $-\nabla F(\mathbf{x})$.

It can be shown that if

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k),$$

with $\gamma_k > 0$.

For γ_k small enough, then $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$. This means that for a sufficiently small γ_k we are always moving towards smaller function values, i.e a minimum.

More on Steepest descent

The previous observation is the basis of the method of steepest descent, which is also referred to as just gradient descent (GD). One starts with an initial guess $\mathbf{x}_0$ for a minimum of F and computes new approximations according to

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k), \quad k \geq 0.$$

The parameter γ_k is often referred to as the step length or the learning rate within the context of Machine Learning.

The ideal

Ideally the sequence $\{\mathbf{x}_k\}_{k=0}$ converges to a global minimum of the function F. In general we do not know if we are in a global or local minimum. In the special case when F is a convex function, all local minima are also global minima, so in this case gradient descent can converge to the global solution. The advantage of this scheme is that it is conceptually simple and straightforward to implement. However the method in this form has some severe limitations:

In machine learing we are often faced with non-convex high dimensional cost functions with many local minima. Since GD is deterministic we will get stuck in a local minimum, if

the method converges, unless we have a very good intial guess. This also implies that the scheme is sensitive to the chosen initial condition.

Note that the gradient is a function of $\mathbf{x} = (x_1, \cdots, x_n)$ which makes it expensive to compute numerically.

The sensitiveness of the gradient descent

The gradient descent method is sensitive to the choice of learning rate γ_k. This is due to the fact that we are only guaranteed that $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$ for sufficiently small γ_k. The problem is to determine an optimal learning rate. If the learning rate is chosen too small the method will take a long time to converge and if it is too large we can experience erratic behavior.

Many of these shortcomings can be alleviated by introducing randomness. One such method is that of Stochastic Gradient Descent (SGD), see below.

Convex functions

Ideally we want our cost/loss function to be convex(concave).

First we give the definition of a convex set: A set C in $\mathbb{R}^n$ is said to be convex if, for all x and y in C and all $t \in (0,1)$, the point $(1t)x + ty$ also belongs to C. Geometrically this means that every point on the line segment connecting x and y is in C as discussed below.

The convex subsets of $\mathbb{R}$ are the intervals of $\mathbb{R}$. Examples of convex sets of $\mathbb{R}^2$ are the regular polygons (triangles, rectangles, pentagons, etc...).

Convex function

Convex function: Let $X \subset \mathbb{R}^n$ be a convex set. Assume that the function $f : X \to \mathbb{R}$ is continuous, then f is said to be convex if

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

for all $x_1, x_2 \in X$ and for all $t \in [0,1]$. If $\leq$ is replaced with a strict inequaltiy in the definition, we demand $x_1 \neq x_2$ and $t \in (0,1)$ then f is said to be strictly convex. For a single variable function, convexity means that if you draw a straight line connecting $f(x_1)$ and $f(x_2)$, the value of the function on the interval $[x_1, x_2]$ is always below the line as illustrated below.

Conditions on convex functions

In the following we state first and second-order conditions which ensures convexity of a function f. We write D_f to denote the domain of f, i.e the subset of R^n where f is defined. For more details and proofs we refer to: S. Boyd and L. Vandenberghe. Convex Optimization. Cambridge University Press.

First order condition.

Suppose f is differentiable (i.e $\nabla f(x)$ is well defined for all x in the domain of f). Then f is convex if and only if D_f is a convex set and

$$f(y) \geq f(x) + \nabla f(x)^T (y - x)$$

holds for all $x, y \in D_f$. This condition means that for a convex function the first order Taylor expansion (right hand side above) at any point a global under estimator of the function. To convince yourself you can make a drawing of $f(x) = x^2 + 1$ and draw the tangent line to $f(x)$ and note that it is always below the graph.

Second order condition.

Assume that f is twice differentiable, i.e the Hessian matrix exists at each point in D_f. Then f is convex if and only if D_f is a convex set and its Hessian is positive semi-definite for all $x \in D_f$. For a single-variable function this reduces to $f''(x) \geq 0$. Geometrically this means that f has nonnegative curvature everywhere.

This condition is particularly useful since it gives us an procedure for determining if the function under consideration is convex, apart from using the definition.

More on convex functions

The next result is of great importance to us and the reason why we are going on about convex functions. In machine learning we frequently have to minimize a loss/cost function in order to find the best parameters for the model we are considering.

Ideally we want the global minimum (for high-dimensional models it is hard to know if we have local or global minimum). However, if the cost/loss function is convex the following result provides invaluable information:

Any minimum is global for convex functions.

Consider the problem of finding $x \in \mathbb{R}^n$ such that $f(x)$ is minimal, where f is convex and differentiable. Then, any point x^* that satisfies $\nabla f(x^*) = 0$ is a global minimum.

This result means that if we know that the cost/loss function is convex and we are able to find a minimum, we are guaranteed that it is a global minimum.

Some simple problems

1. Show that $f(x) = x^2$ is convex for $x \in \mathbb{R}$ using the definition of convexity. Hint: If you re-write the definition, f is convex if the following holds for all $x, y \in D_f$ and any $\lambda \in [0,1]$ $\lambda f(x) + (1-\lambda) f(y) - f(\lambda x + (1-\lambda) y) \geq 0$.
2. Using the second order condition show that the following functions are convex on the specified domain.
   - $f(x) = e^x$ is convex for $x \in \mathbb{R}$.
   - $g(x) = -\ln(x)$ is convex for $x \in (0, \infty)$.

3. Let $f(x)=x^2$ and $g(x)=e^x$. Show that $f(g(x))$ and $g(f(x))$ is convex for $x\in\mathbb{R}$. Also show that if $f(x)$ is any convex function than $h(x)=e^{f(x)}$ is convex.
4. A norm is any function that satisfy the following properties

   - $f(\alpha x)=|\alpha|f(x)$ for all $\alpha\in\mathbb{R}$.
   - $f(x+y)\leq f(x)+f(y)$
   - $f(x)\leq 0$ for all $x\in\mathbb{R}^n$ with equality if and only if $x=0$

Using the definition of convexity, try to show that a function satisfying the properties above is convex (the third condition is not needed to show this).

Standard steepest descent

Before we proceed, we would like to discuss the approach called the **standard Steepest descent**, which again leads to us having to be able to compute a matrix. It belongs to the class of Conjugate Gradient methods (CG).

The success of the CG method for finding solutions of non-linear problems is based on the theory of conjugate gradients for linear systems of equations. It belongs to the class of iterative methods for solving problems from linear algebra of the type

$$\hat{A}\hat{x}=\hat{b}.$$

In the iterative process we end up with a problem like

$$\hat{r}=\hat{b}-\hat{A}\hat{x},$$

where $\hat{r}$ is the so-called residual or error in the iterative process.

When we have found the exact solution, $\hat{r}=0$.

Gradient method

The residual is zero when we reach the minimum of the quadratic equation

$$P(\hat{x})=\frac{1}{2}\hat{x}^T\hat{A}\hat{x}-\hat{x}^T\hat{b},$$

with the constraint that the matrix $\hat{A}$ is positive definite and symmetric. This defines also the Hessian and we want it to be positive definite.

Steepest descent method

We denote the initial guess for $\hat{x}$ as $\hat{x}_0$. We can assume without loss of generality that

$$\hat{x}_0=0,$$

or consider the system

$$\hat{A}\hat{z}=\hat{b}-\hat{A}\hat{x}_0,$$

instead.

Steepest descent method

One can show that the solution $\hat{x}$ is also the unique minimizer of the quadratic form

$$f(\hat{x}) = \frac{1}{2}\hat{x}^T\hat{A}\hat{x} - \hat{x}^T\hat{x}, \quad \hat{x} \in \mathbf{R}^n.$$

This suggests taking the first basis vector $\hat{r}_1$ (see below for definition) to be the gradient of f at $\hat{x} = \hat{x}_0$, which equals

$$\hat{A}\hat{x}_0 - \hat{b},$$

and $\hat{x}_0 = 0$ it is equal $-\hat{b}$.

Final expressions

We can compute the residual iteratively as

$$\hat{r}_{k+1} = \hat{b} - \hat{A}\hat{x}_{k+1},$$

which equals

$$\hat{b} - \hat{A}(\hat{x}_k + \alpha_k\hat{r}_k),$$

or

$$(\hat{b} - \hat{A}\hat{x}_k) - \alpha_k\hat{A}\hat{r}_k,$$

which gives

$$\alpha_k = \frac{\hat{r}_k^T\hat{r}_k}{\hat{r}_k^T\hat{A}\hat{r}_k}$$

leading to the iterative scheme

$$\hat{x}_{k+1} = \hat{x}_k - \alpha_k\hat{r}_k,$$

Conjugate gradient method

In the CG method we define so-called conjugate directions and two vectors $\hat{s}$ and $\hat{t}$ are said to be conjugate if

$$\hat{s}^T\hat{A}\hat{t} = 0.$$

The philosophy of the CG method is to perform searches in various conjugate directions of our vectors $\hat{x}_i$ obeying the above criterion, namely

$$\hat{x}_i^T\hat{A}\hat{x}_j = 0.$$

Two vectors are conjugate if they are orthogonal with respect to this inner product. Being conjugate is a symmetric relation: if $\hat{s}$ is conjugate to $\hat{t}$, then $\hat{t}$ is conjugate to $\hat{s}$.

Conjugate gradient method

An example is given by the eigenvectors of the matrix

$$\hat{v}_i^T\hat{A}\hat{v}_j = \lambda\hat{v}_i^T\hat{v}_j,$$

which is zero unless $i = j$.

Conjugate gradient method

Assume now that we have a symmetric positive-definite matrix $\hat{A}$ of size $n\times n$. At each iteration $i+1$ we obtain the conjugate direction of a vector

$$\hat{x}_{i+1} = \hat{x}_i + \alpha_i\hat{p}_i.$$

We assume that $\hat{p}_i$ is a sequence of n mutually conjugate directions. Then the $\hat{p}_i$ form a basis of R^n and we can expand the solution $\hat{A}\hat{x} = \hat{b}$ in this basis, namely

$$\hat{x} = \sum_{i=1}^{n}\alpha_i\hat{p}_i.$$

Conjugate gradient method

The coefficients are given by

$$\mathbf{Ax} = \sum_{i=1}^{n}\alpha_i\mathbf{Ap}_i = \mathbf{b}.$$

Multiplying with $\hat{p}_k^T$ from the left gives

$$\hat{p}_k^T\hat{A}\hat{x} = \sum_{i=1}^{n}\alpha_i\hat{p}_k^T\hat{A}\hat{p}_i = \hat{p}_k^T\hat{b},$$

and we can define the coefficients α_k as

$$\alpha_k = \frac{\hat{p}_k^T\hat{b}}{\hat{p}_k^T\hat{A}\hat{p}_k}$$

Conjugate gradient method and iterations

If we choose the conjugate vectors $\hat{p}_k$ carefully, then we may not need all of them to obtain a good approximation to the solution $\hat{x}$. We want to regard the conjugate gradient method as an iterative method. This will us to solve systems where n is so large that the direct method would take too much time.

We denote the initial guess for $\hat{x}$ as $\hat{x}_0$. We can assume without loss of generality that

$$\hat{x}_0 = 0,$$

or consider the system

$$\hat{A}\hat{z} = \hat{b} - \hat{A}\hat{x}_0,$$

instead.

Conjugate gradient method

One can show that the solution $\hat{x}$ is also the unique minimizer of the quadratic form

$$f(\hat{x}) = \frac{1}{2}\hat{x}^T\hat{A}\hat{x} - \hat{x}^T\hat{x}, \quad \hat{x} \in \mathbf{R}^n.$$

This suggests taking the first basis vector $\hat{p}_1$ to be the gradient of f at $\hat{x} = \hat{x}_0$, which equals

$$\hat{A}\hat{x}_0 - \hat{b},$$

and $\hat{x}_0 = 0$ it is equal $-\hat{b}$. The other vectors in the basis will be conjugate to the gradient, hence the name conjugate gradient method.

Conjugate gradient method

Let $\hat{r}_k$ be the residual at the k-th step:

$$\hat{r}_k = \hat{b} - \hat{A}\hat{x}_k.$$

Note that $\hat{r}_k$ is the negative gradient of f at $\hat{x} = \hat{x}_k$, so the gradient descent method would be to move in the direction $\hat{r}_k$. Here, we insist that the directions $\hat{p}_k$ are conjugate to each other, so we take the direction closest to the gradient $\hat{r}_k$ under the conjugacy constraint. This gives the following expression

$$\hat{p}_{k+1} = \hat{r}_k - \frac{\hat{p}_k^T\hat{A}\hat{r}_k}{\hat{p}_k^T\hat{A}\hat{p}_k}\hat{p}_k.$$

Conjugate gradient method

We can also compute the residual iteratively as

$$\hat{r}_{k+1} = \hat{b} - \hat{A}\hat{x}_{k+1},$$

which equals

$$\hat{b} - \hat{A}(\hat{x}_k + \alpha_k \hat{p}_k),$$

or

$$(\hat{b} - \hat{A}\hat{x}_k) - \alpha_k \hat{A}\hat{p}_k,$$

which gives

$$\hat{r}_{k+1} = \hat{r}_k - \hat{A}\hat{p}_k,$$

Broyden–Fletcher–Goldfarb–Shanno algorithm

The optimization problem is to minimize $f(\mathbf{x})$ where $\mathbf{x}$ is a vector in R^n, and f is a differentiable scalar function. There are no constraints on the values that $\mathbf{x}$ can take.

The algorithm begins at an initial estimate for the optimal value $\mathbf{x}_0$ and proceeds iteratively to get a better estimate at each stage.

The search direction p_k at stage k is given by the solution of the analogue of the Newton equation

$$B_k \mathbf{p}_k = -\nabla f(\mathbf{x}_k),$$

where B_k is an approximation to the Hessian matrix, which is updated iteratively at each stage, and $\nabla f(\mathbf{x}_k)$ is the gradient of the function evaluated at x_k. A line search in the direction p_k is then used to find the next point x_{k+1} by minimising

$$f(\mathbf{x}_k + \alpha \mathbf{p}_k),$$

over the scalar $\alpha > 0$.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) and variants thereof address some of the shortcomings of the Gradient descent method discussed above.

The underlying idea of SGD comes from the observation that a given function, which we want to minimize, can almost always be written as a sum over n data points $\{\mathbf{x}_i\}_{i=1}^n$,

$$C(\beta) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \beta).$$

Computation of gradients

This in turn means that the gradient can be computed as a sum over i-gradients

$$\nabla_\beta C(\beta) = \sum_i^n \nabla_\beta c_i(\mathbf{x}_i, \beta).$$

Stochasticity/randomness is introduced by only taking the gradient on a subset of the data called minibatches. If there are n data points and the size of each minibatch is M, there will be n/M minibatches. We denote these minibatches by B_k where $k = 1, \cdots, n/M$.

SGD example

As an example, suppose we have 10 data points $(\mathbf{x}_1, \cdots, \mathbf{x}_{10})$ and we choose to have $M = 5$ minibathces, then each minibatch contains two data points. In particular we have $B_1 = (\mathbf{x}_1, \mathbf{x}_2), \cdots, B_5 = (\mathbf{x}_9, \mathbf{x}_{10})$. Note that if you choose $M = 1$ you have only a single batch with all data points and on the other extreme, you may choose $M = n$ resulting in a minibatch for each datapoint, i.e $B_k = \mathbf{x}_k$.

The idea is now to approximate the gradient by replacing the sum over all data points with a sum over the data points in one the minibatches picked at random in each gradient descent step

$$\nabla_\beta C(\beta) = \sum_{i=1}^n \nabla_\beta c_i(\mathbf{x}_i, \beta) \rightarrow \sum_{i \in B_k}^n \nabla_\beta c_i(\mathbf{x}_i, \beta).$$

The gradient step

Thus a gradient descent step now looks like

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^n \nabla_\beta c_i(\mathbf{x}_i, \beta)$$

where k is picked at random with equal probability from $[1, n/M]$. An iteration over the number of minibathces (n/M) is commonly referred to as an epoch. Thus it is typical to choose a number of epochs and for each epoch iterate over the number of minibatches, as exemplified in the code below.

Simple example code

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python
import numpy as np

n = 100 100 datapoints M = 5 size of each minibatch m = int(n/M) number of minibatches
n$_epochs = 10 number of epochs$

j = 0 for epoch in range(1,n$_epochs+1) : for i in range(m) : k = np.random.randint(m) Pick the k-th minibatch at random Compute the grad$
1

Taking the gradient only on a subset of the data has two important benefits. First, it introduces randomness which decreases the chance that our opmization scheme gets stuck in

a local minima. Second, if the size of the minibatches are small relative to the number of datapoints ($M < n$), the computation of the gradient is much cheaper since we sum over the datapoints in the $k-th$ minibatch and not all n datapoints.

When do we stop?

A natural question is when do we stop the search for a new minimum? One possibility is to compute the full gradient after a given number of epochs and check if the norm of the gradient is smaller than some threshold and stop if true. However, the condition that the gradient is zero is valid also for local minima, so this would only tell us that we are close to a local/global minimum. However, we could also evaluate the cost function at this point, store the result and continue the search. If the test kicks in at a later stage we can compare the values of the cost function and keep the β that gave the lowest value.

Slightly different approach

Another approach is to let the step length γ_j depend on the number of epochs in such a way that it becomes very small after a reasonable time such that we do not move at all.

As an example, let $e = 0,1,2,3,\cdots$ denote the current epoch and let $t_0, t_1 > 0$ be two fixed numbers. Furthermore, let $t = e \cdot m + i$ where m is the number of minibatches and $i = 0, \cdots, m-1$. Then the function

$$\gamma_j(t; t_0, t_1) = \frac{t_0}{t + t_1}$$

goes to zero as the number of epochs gets large. I.e. we start with a step length $\gamma_j(0; t_0, t_1) = t_0/t_1$ which decays in *time* t.

In this way we can fix the number of epochs, compute β and evaluate the cost function at the end. Repeating the computation will give a different result since the scheme is random by design. Then we pick the final β that gives the lowest value of the cost function.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python
import numpy as np

def step$_length(t,t0,t1) : returnt0/(t+t1)$

n = 100 100 datapoints M = 5 size of each minibatch m = int(n/M) number of minibatches
n$_epochs = 500numberofepochst0 = 1.0t1 = 10$

gamma$_j = t0/t1 j = 0 forepochinrange(1, n_epochs+1) : foriinrange(m) : k = np.random.randint(m) Pickthek-thminibatchatrandomComputethegradientusingthedatainminibatchBkComputenewsuggestionforbetat = epoch*m+igamma_j = step_length(t,t0,t1) j+ = 1$

print("gamma$_jafter$

Program for stochastic gradient

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python
Importing various packages from math import exp, sqrt from random import random, seed
import numpy as np import matplotlib.pyplot as plt from sklearn.linear$_model import SGDRegressor$

x = 2*np.random.rand(100,1) y = 4+3*x+np.random.randn(100,1)

xb = np.c$_[np.ones((100,1)), x] theta_linreg = np.linalg.inv(xb.T.dot(xb)).dot(xb.T).dot(y) print("Owninversion") print(theta_linreg) sgdreg = SGDRegressor(n_iter = 50, penalty = None, eta0 = 0.1) sgdreg.fit(x, y.ravel()) print("sgdregfromscikit") print(sgdreg.intercept_, sgdreg.coef_)$

theta = np.random.randn(2,1)

eta = 0.1 Niterations = 1000 m = 100

for iter in range(Niterations): gradients = 2.0/m*xb.T.dot(xb.dot(theta)-y) theta -= eta*gradients print("theta frm own gd") print(theta)

xnew = np.array([[0],[2]]) xbnew = np.c$_{[}$$np.ones((2,1)), xnew]ypredict = xbnew.dot(theta)ypredict2 = xbnew.dot(theta_linreg)$

n$_e$$pochs = 50t0, t1 = 5, 50m = 100def learning_schedule(t) : returnt0/(t + t1)$

theta = np.random.randn(2,1)

for epoch in range(n$_e$$pochs) : foriinrange(m) : random_index = np.random.randint(m)xi = xb[random_index : random_index + 1]yi = y[random_index : random_index + 1]gradients = 2 * xi.T.dot(xi.dot(theta) - yi)eta = learning_schedule(epoch * m + i)theta = theta - eta * gradientsprint("thetafromownsdg")print(theta)$

plt.plot(xnew, ypredict, "r-") plt.plot(xnew, ypredict2, "b-") plt.plot(x, y ,'ro') plt.axis([0,2.0,0, 15.0]) plt.xlabel(r'x') plt.ylabel(r'y') plt.title(r'Random numbers ') plt.show()

Using gradient descent methods, limitations

- **Gradient descent (GD) finds local minima of our function**. Since the GD algorithm is deterministic, if it converges, it will converge to a local minimum of our energy function. Because in ML we are often dealing with extremely rugged landscapes with many local minima, this can lead to poor performance.
- **GD is sensitive to initial conditions**. One consequence of the local nature of GD is that initial conditions matter. Depending on where one starts, one will end up at a different local minima. Therefore, it is very important to think about how one initializes the training process. This is true for GD as well as more complicated variants of GD.
- **Gradients are computationally expensive to calculate for large datasets**. In many cases in statistics and ML, the energy function is a sum of terms, with one term for each data point. For example, in linear regression, $E \propto \sum_{i=1}^{n} (y_i - \mathbf{w}^T \cdot \mathbf{x}_i)^2$; for logistic regression, the square error is replaced by the cross entropy. To calculate the gradient we have to sum over *all* n data points. Doing this at every GD step becomes extremely computationally expensive. An ingenious solution to this, is to calculate the gradients using small subsets of the data called "mini batches". This has the added benefit of introducing stochasticity into our algorithm.
- **GD is very sensitive to choices of learning rates**. GD is extremely sensitive to the choice of learning rates. If the learning rate is very small, the training process take an extremely long time. For larger learning rates, GD can diverge and give poor results. Furthermore, depending on what the local landscape looks like, we have to modify the learning rates to ensure convergence. Ideally, we would *adaptively* choose the learning rates to match the landscape.
- **GD treats all directions in parameter space uniformly.** Another major drawback of GD is that unlike Newton's method, the learning rate for GD is the same in all directions in parameter space. For this reason, the maximum learning rate is set by the behavior of the steepest direction and this can significantly slow down training. Ideally, we would like to take large steps in flat directions and small steps in steep directions. Since we are exploring rugged landscapes where curvatures change, this requires us to keep track of not only the gradient but second derivatives. The ideal scenario would be to calculate the Hessian but this proves to be too computationally expensive.
- GD can take exponential time to escape saddle points, even with random initialization. As we mentioned, GD is extremely sensitive to initial condition since it determines the particular local minimum GD would eventually reach. However, even with a good initialization

scheme, through the introduction of randomness, GD can still take exponential time to escape saddle points.

Codes from numerical recipes

You can however use codes we have adapted from the text Numerical Recipes in C++, see chapter 10.7. Here we present a program, which you also can find at the webpage of the course we use the functions **dfpmin** and **lnsrch**. This is a variant of the Broyden et al algorithm discussed in the previous slide.

- The program uses the harmonic oscillator in one dimensions as example.
- The program does not use armadillo to handle vectors and matrices, but employs rather my own vector-matrix class. These auxiliary functions, and the main program *model.cpp* can all be found under the program link here.

Below we show only excerpts from the main program. For the full program, see the above link.

Finding the minimum of the harmonic oscillator model in one dimension

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python // Main function begins here int main() int n, iter; double gtol, fret; double alpha; n = 1; // reserve space in memory for vectors containing the variational // parameters Vector g(n), p(n); cout « "Read in guess for alpha" « endl; cin » alpha; gtol = 1.0e-5; // now call dfmin and compute the minimum p(0) = alpha; dfpmin(p, n, gtol, iter, fret, Efunction, dEfunction); cout « "Value of energy minimum = " « fret « endl; cout « "Number of iterations = " « iter « endl; cout « "Value of alpha at minimum = " « p(0) « endl; return 0; // end of main program

Functions to observe

The functions **Efunction** and **dEfunction** compute the expectation value of the energy and its derivative. They use the the quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS) It uses the first derivatives only. The BFGS algorithm has proven good performance even for non-smooth optimizations. These functions need to be changed when you want to your own derivatives.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python // this function defines the expectation value of the local energy double Efunction(Vector x) double value = x(0)*x(0)*0.5+1.0/(8*x(0)*x(0)); return value; // end of function to evaluate

// this function defines the derivative of the energy void dEfunction(Vector x, Vector g) g(0) = x(0)-1.0/(4*x(0)*x(0)*x(0)); // end of function to evaluate

You need to change these functions in order to compute the local energy for your system. I used 1000 cycles per call to get a new value of $\langle E_L[\alpha]\rangle$. When I compute the local energy I also compute its derivative. After roughly 10-20 iterations I got a converged result in terms of α.

Resampling Techniques, Bootstrap and Blocking

Why resampling methods ?

Statistical analysis.

- Our simulations can be treated as *computer experiments*. This is particularly the case for Monte Carlo methods
- The results can be analysed with the same statistical tools as we would use analysing experimental data.
- As in all experiments, we are looking for expectation values and an estimate of how accurate they are, i.e., possible sources for errors.

Statistical analysis

- As in other experiments, many numerical experiments have two classes of errors:
  - Statistical errors
  - Systematical errors
- Statistical errors can be estimated using standard tools from statistics
- Systematical errors are method specific and must be treated differently from case to case.

Statistics, wrapping up from last week

Let us analyze the problem by splitting up the correlation term into partial sums of the form:

$$f_d = \frac{1}{n-d}\sum_{k=1}^{n-d}(x_k - \bar{x}_n)(x_{k+d} - \bar{x}_n)$$

The correlation term of the error can now be rewritten in terms of f_d

$$\frac{2}{n}\sum_{k<l}(x_k - \bar{x}_n)(x_l - \bar{x}_n) = 2\sum_{d=1}^{n-1} f_d$$

The value of f_d reflects the correlation between measurements separated by the distance d in the sample samples. Notice that for $d = 0$, f is just the sample variance, $\mathrm{var}(x)$. If we divide f_d by $\mathrm{var}(x)$, we arrive at the so called *autocorrelation function*

$$\kappa_d = \frac{f_d}{\mathrm{var}(x)}$$

which gives us a useful measure of pairwise correlations starting always at 1 for $d = 0$.

Statistics, final expression

The sample error can now be written in terms of the autocorrelation function:

$$\begin{aligned} \mathrm{err}_X^2 &= \frac{1}{n}\mathrm{var}(x) + \frac{2}{n}\cdot\mathrm{var}(x)\sum_{d=1}^{n-1}\frac{f_d}{\mathrm{var}(x)} \\ &= \left(1+2\sum_{d=1}^{n-1}\kappa_d\right)\frac{1}{n}\mathrm{var}(x) \\ &= \frac{\tau}{n}\cdot\mathrm{var}(x) \end{aligned} \tag{1.19}$$

and we see that err_X can be expressed in terms the uncorrelated sample variance times a correction factor τ which accounts for the correlation between measurements. We call this correction factor the *autocorrelation time*:

$$\tau = 1+2\sum_{d=1}^{n-1}\kappa_d \tag{1.20}$$

Statistics, effective number of correlations

For a correlation free experiment, τ equals 1.

We can interpret a sequential correlation as an effective reduction of the number of measurements by a factor τ. The effective number of measurements becomes:

$$n_{\mathrm{eff}} = \frac{n}{\tau}$$

To neglect the autocorrelation time τ will always cause our simple uncorrelated estimate of $\mathrm{err}_X^2 \approx \mathrm{var}(x)/n$ to be less than the true sample error. The estimate of the error will be too *good*. On the other hand, the calculation of the full autocorrelation time poses an efficiency problem if the set of measurements is very large.

Can we understand this? Time Auto-correlation Function

The so-called time-displacement autocorrelation $\phi(t)$ for a quantity $\mathbf{M}$ is given by

$$\phi(t) = \int dt' \left[\mathbf{M}(t') - \langle \mathbf{M} \rangle\right] \left[\mathbf{M}(t'+t) - \langle \mathbf{M} \rangle\right],$$

which can be rewritten as

$$\phi(t) = \int dt' \left[\mathbf{M}(t')\mathbf{M}(t'+t) - \langle \mathbf{M} \rangle^2\right],$$

where $\langle \mathbf{M} \rangle$ is the average value and $\mathbf{M}(t)$ its instantaneous value. We can discretize this function as follows, where we used our set of computed values $\mathbf{M}(t)$ for a set of discretized times (our Monte Carlo cycles corresponding to moving all electrons?)

$$\phi(t) = \frac{1}{t_{\max}-t}\sum_{t'=0}^{t_{\max}-t} \mathbf{M}(t')\mathbf{M}(t'+t) - \frac{1}{t_{\max}-t}\sum_{t'=0}^{t_{\max}-t} \mathbf{M}(t') \times \frac{1}{t_{\max}-t}\sum_{t'=0}^{t_{\max}-t} \mathbf{M}(t'+t).$$

Time Auto-correlation Function

One should be careful with times close to $t_{\max}$, the upper limit of the sums becomes small and we end up integrating over a rather small time interval. This means that the statistical error in $\phi(t)$ due to the random nature of the fluctuations in $\mathbf{M}(t)$ can become large.

One should therefore choose $t \ll t_{\max}$.

Note that the variable $\mathbf{M}$ can be any expectation values of interest.

The time-correlation function gives a measure of the correlation between the various values of the variable at a time t' and a time $t'+t$. If we multiply the values of $\mathbf{M}$ at these two different times, we will get a positive contribution if they are fluctuating in the same direction, or a negative value if they fluctuate in the opposite direction. If we then integrate over time, or use the discretized version of, the time correlation function $\phi(t)$ should take a non-zero value if the fluctuations are correlated, else it should gradually go to zero. For times a long way apart the different values of $\mathbf{M}$ are most likely uncorrelated and $\phi(t)$ should be zero.

Time Auto-correlation Function

We can derive the correlation time by observing that our Metropolis algorithm is based on a random walk in the space of all possible spin configurations. Our probability distribution function $\hat{\mathbf{w}}(t)$ after a given number of time steps t could be written as

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^{\mathbf{t}}\hat{\mathbf{w}}(0),$$

with $\hat{\mathbf{w}}(0)$ the distribution at $t = 0$ and $\hat{\mathbf{W}}$ representing the transition probability matrix. We can always expand $\hat{\mathbf{w}}(0)$ in terms of the right eigenvectors of $\hat{\mathbf{v}}$ of $\hat{\mathbf{W}}$ as

$$\hat{\mathbf{w}}(0) = \sum_i \alpha_i \hat{\mathbf{v}}_i,$$

resulting in

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^t \hat{\mathbf{w}}(0) = \hat{\mathbf{W}}^t \sum_i \alpha_i \hat{\mathbf{v}}_i = \sum_i \lambda_i^t \alpha_i \hat{\mathbf{v}}_i,$$

with λ_i the i^{th} eigenvalue corresponding to the eigenvector $\hat{\mathbf{v}}_i$.

Time Auto-correlation Function

If we assume that λ_0 is the largest eigenvector we see that in the limit $t \to \infty$, $\hat{\mathbf{w}}(t)$ becomes proportional to the corresponding eigenvector $\hat{\mathbf{v}}_0$. This is our steady state or final distribution.

We can relate this property to an observable like the mean energy. With the probabilty $\hat{\mathbf{w}}(t)$ (which in our case is the squared trial wave function) we can write the expectation values as

$$\langle \mathbf{M}(t) \rangle = \sum_\mu \hat{\mathbf{w}}(t)_\mu \mathbf{M}_\mu,$$

or as the scalar of a vector product

$$\langle \mathbf{M}(t) \rangle = \hat{\mathbf{w}}(t)\mathbf{m},$$

with $\mathbf{m}$ being the vector whose elements are the values of $\mathbf{M}_\mu$ in its various microstates μ.

Time Auto-correlation Function

We rewrite this relation as

$$\langle \mathbf{M}(t) \rangle = \hat{\mathbf{w}}(t)\mathbf{m} = \sum_i \lambda_i^t \alpha_i \hat{\mathbf{v}}_i \mathbf{m}_i.$$

If we define $m_i = \hat{\mathbf{v}}_i \mathbf{m}_i$ as the expectation value of $\mathbf{M}$ in the i^{th} eigenstate we can rewrite the last equation as

$$\langle \mathbf{M}(t) \rangle = \sum_i \lambda_i^t \alpha_i m_i.$$

Since we have that in the limit $t \to \infty$ the mean value is dominated by the the largest eigenvalue λ_0, we can rewrite the last equation as

$$\langle \mathbf{M}(t) \rangle = \langle \mathbf{M}(\infty) \rangle + \sum_{i \neq 0} \lambda_i^t \alpha_i m_i.$$

We define the quantity

$$\tau_i = -\frac{1}{log \lambda_i},$$

and rewrite the last expectation value as

$$\langle \mathbf{M}(t) \rangle = \langle \mathbf{M}(\infty) \rangle + \sum_{i \neq 0} \alpha_i m_i e^{-t/\tau_i}.$$

Time Auto-correlation Function

The quantities τ_i are the correlation times for the system. They control also the auto-correlation function discussed above. The longest correlation time is obviously given by the second largest eigenvalue τ_1, which normally defines the correlation time discussed above. For large times, this is the only correlation time that survives. If higher eigenvalues of the transition matrix are well separated from λ_1 and we simulate long enough, τ_1 may well define the correlation time. In other cases we may not be able to extract a reliable result for τ_1. Coming back to the time correlation function $\phi(t)$ we can present a more general definition in terms of the mean magnetizations $\langle \mathbf{M}(t) \rangle$. Recalling that the mean value is equal to $\langle \mathbf{M}(\infty) \rangle$ we arrive at the expectation values

$$\phi(t) = \langle \mathbf{M}(0) - \mathbf{M}(\infty) \rangle \langle \mathbf{M}(t) - \mathbf{M}(\infty) \rangle,$$

resulting in

$$\phi(t) = \sum_{i,j \neq 0} m_i \alpha_i m_j \alpha_j e^{-t/\tau_i},$$

which is appropriate for all times.

Correlation Time

If the correlation function decays exponentially

$$\phi(t) \sim \exp(-t/\tau)$$

then the exponential correlation time can be computed as the average

$$\tau_{\exp} = -\langle \frac{t}{log|\frac{\phi(t)}{\phi(0)}|} \rangle.$$

If the decay is exponential, then

$$\int_0^\infty dt \phi(t) = \int_0^\infty dt \phi(0) \exp(-t/\tau) = \tau \phi(0),$$

which suggests another measure of correlation

$$\tau_{\text{int}} = \sum_k \frac{\phi(k)}{\phi(0)},$$

called the integrated correlation time.

Resampling methods: Jackknife and Bootstrap

Two famous resampling methods are the **independent bootstrap** and **the jackknife**.

The jackknife is a special case of the independent bootstrap. Still, the jackknife was made popular prior to the independent bootstrap. And as the popularity of the independent bootstrap soared, new variants, such as **the dependent bootstrap**.

The Jackknife and independent bootstrap work for independent, identically distributed random variables. If these conditions are not satisfied, the methods will fail. Yet, it should be said that if the data are independent, identically distributed, and we only want to estimate the variance of $\overline{X}$ (which often is the case), then there is no need for bootstrapping.

Resampling methods: Jackknife

The Jackknife works by making many replicas of the estimator $\widehat{\theta}$. The jackknife is a resampling method, we explained that this happens by scrambling the data in some way. When using the jackknife, this is done by systematically leaving out one observation from the vector of observed values $\hat{x} = (x_1, x_2, \cdots, X_n)$. Let $\hat{x}_i$ denote the vector

$$\hat{x}_i = (x_1, x_2, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n),$$

which equals the vector $\hat{x}$ with the exception that observation number i is left out. Using this notation, define $\widehat{\theta}_i$ to be the estimator $\widehat{\theta}$ computed using $\vec{X}_i$.

Resampling methods: Jackknife estimator

To get an estimate for the bias and standard error of $\widehat{\theta}$, use the following estimators for each component of $\widehat{\theta}$

$$\widehat{\mathrm{Bias}}(\widehat{\theta}, \theta) = (n-1)\left(-\widehat{\theta} + \frac{1}{n}\sum_{i=1}^{n}\widehat{\theta}_i\right) \qquad \text{and} \qquad \widehat{\sigma}^2_{\widehat{\theta}} = \frac{n-1}{n}\sum_{i=1}^{n}(\widehat{\theta}_i - \frac{1}{n}\sum_{j=1}^{n}\widehat{\theta}_j)^2.$$

Jackknife code example

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python from numpy import * from numpy.random import randint, randn from time import time

def jackknife(data, stat): n = len(data);t = zeros(n); inds = arange(n); t0 = time() 'jackknifing' by leaving out an observation for each i for i in range(n): t[i] = stat(delete(data,i))

analysis print("Runtime: print("original bias std. error") print("

return t

Returns mean of data samples def stat(data): return mean(data)

mu, sigma = 100, 15 datapoints = 10000 x = mu + sigma*random.randn(datapoints) jackknife returns the data sample t = jackknife(x, stat)

Resampling methods: Bootstrap

Bootstrapping is a nonparametric approach to statistical inference that substitutes computation for more traditional distributional assumptions and asymptotic results. Bootstrapping offers a number of advantages:

1. The bootstrap is quite general, although there are some cases in which it fails.
2. Because it does not require distributional assumptions (such as normally distributed errors), the bootstrap can provide more accurate inferences when the data are not well behaved or when the sample size is small.
3. It is possible to apply the bootstrap to statistics with sampling distributions that are difficult to derive, even asymptotically.
4. It is relatively simple to apply the bootstrap to complex data-collection plans (such as stratified and clustered samples).

Resampling methods: Bootstrap background

Since $\widehat{\theta} = \widehat{\theta}(\hat{X})$ is a function of random variables, $\widehat{\theta}$ itself must be a random variable. Thus it has a pdf, call this function $p(\hat{t})$. The aim of the bootstrap is to estimate $p(\hat{t})$ by the relative frequency of $\widehat{\theta}$. You can think of this as using a histogram in the place of $p(\hat{t})$. If the relative frequency closely resembles $p(\vec{t})$, then using numerics, it is straight forward to estimate all the interesting parameters of $p(\hat{t})$ using point estimators.

Resampling methods: More Bootstrap background

In the case that $\widehat{\theta}$ has more than one component, and the components are independent, we use the same estimator on each component separately. If the probability density function of X_i, $p(x)$, had been known, then it would have been straight forward to do this by:

1. Drawing lots of numbers from $p(x)$, suppose we call one such set of numbers $(X_1^*, X_2^*, \cdots, X_n^*)$.
2. Then using these numbers, we could compute a replica of $\widehat{\theta}$ called $\widehat{\theta}^*$.

By repeated use of (1) and (2), many estimates of $\widehat{\theta}$ could have been obtained. The idea is to use the relative frequency of $\widehat{\theta}^*$ (think of a histogram) as an estimate of $p(\hat{t})$.

Resampling methods: Bootstrap approach

But unless there is enough information available about the process that generated $X_1, X_2, \cdots, X_n$, $p(x)$ is in general unknown. Therefore, Efron in 1979 asked the question: What if we replace $p(x)$ by the relative frequency of the observation X_i; if we draw observations in accordance with the relative frequency of the observations, will we obtain the same result in some asymptotic sense? The answer is yes.

Instead of generating the histogram for the relative frequency of the observation X_i, just draw the values $(X_1^*, X_2^*, \cdots, X_n^*)$ with replacement from the vector $\hat{X}$.

Resampling methods: Bootstrap steps

The independent bootstrap works like this:

1. Draw with replacement n numbers for the observed variables $\hat{x} = (x_1, x_2, \cdots, x_n)$.
2. Define a vector $\hat{x}^*$ containing the values which were drawn from $\hat{x}$.
3. Using the vector $\hat{x}^*$ compute $\widehat{\theta}^*$ by evaluating $\widehat{\theta}$ under the observations $\hat{x}^*$.
4. Repeat this process k times.

When you are done, you can draw a histogram of the relative frequency of $\widehat{\theta}^*$. This is your estimate of the probability distribution $p(t)$. Using this probability distribution you can estimate any statistics thereof. In principle you never draw the histogram of the relative frequency of $\widehat{\theta}^*$. Instead you use the estimators corresponding to the statistic of interest. For example, if you are interested in estimating the variance of $\widehat{\theta}$, apply the etsimator $\widehat{\sigma}^2$ to the values $\widehat{\theta}^*$.

Code example for the Bootstrap method

The following code starts with a Gaussian distribution with mean value $\mu = 100$ and variance $\sigma = 15$. We use this to generate the data used in the bootstrap analysis. The bootstrap analysis returns a data set after a given number of bootstrap operations (as many as we have data points). This data set consists of estimated mean values for each bootstrap operation. The histogram generated by the bootstrap method shows that the distribution for these mean values is also a Gaussian, centered around the mean value $\mu = 100$ but with standard deviation $\sigma/\sqrt{n}$, where n is the number of bootstrap samples (in this case the same as the number of original data points). The value of the standard deviation is what we expect from the central limit theorem.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python
from numpy import * from numpy.random import randint, randn from time import time from scipy.stats import norm import matplotlib.pyplot as plt

Returns mean of bootstrap samples def stat(data): return mean(data)

Bootstrap algorithm def bootstrap(data, statistic, R): t = zeros(R); n = len(data); inds = arange(n); t0 = time()

non-parametric bootstrap for i in range(R): t[i] = statistic(data[randint(0,n,n)])

analysis print("Runtime: print("original bias std. error") print("mean(t), std(t))) return t

mu, sigma = 100, 15 datapoints = 10000 x = mu + sigma*random.randn(datapoints) bootstrap returns the data sample t = bootstrap(x, stat, datapoints) the histogram of the bootstrapped data t = bootstrap(x, stat, datapoints) the histogram of the bootstrapped data n, binsboot, patches = plt.hist(t, bins=50, density='true',histtype='bar', color='red', alpha=0.75)

add a 'best fit' line y = norm.pdf(binsboot, mean(t), std(t)) lt = plt.plot(binsboot, y, 'r–', linewidth=1) plt.xlabel('Smarts') plt.ylabel('Probability') plt.axis([99.5, 100.6, 0, 3.0]) plt.grid(True)

plt.show()

Resampling methods: Blocking

The blocking method was made popular by Flyvbjerg and Pedersen (1989) and has become one of the standard ways to estimate $V(\widehat{\theta})$ for exactly one $\widehat{\theta}$, namely $\widehat{\theta} = \overline{X}$.

Assume $n = 2^d$ for some integer $d > 1$ and $X_1, X_2, \cdots, X_n$ is a stationary time series to begin with. Moreover, assume that the time series is asymptotically uncorrelated. We switch to vector notation by arranging $X_1, X_2, \cdots, X_n$ in an n-tuple. Define:

$$\hat{X} = (X_1, X_2, \cdots, X_n).$$

The strength of the blocking method is when the number of observations, n is large. For large n, the complexity of dependent bootstrapping scales poorly, but the blocking method does not, moreover, it becomes more accurate the larger n is.

Blocking Transformations

We now define blocking transformations. The idea is to take the mean of subsequent pair of elements from $\vec{X}$ and form a new vector $\vec{X}_1$. Continuing in the same way by taking the mean of subsequent pairs of elements of $\vec{X}_1$ we obtain $\vec{X}_2$, and so on. Define $\vec{X}_i$ recursively by:

$$\begin{aligned} (\vec{X}_0)_k &\equiv (\vec{X})_k \\ (\vec{X}_{i+1})_k &\equiv \frac{1}{2}\Big((\vec{X}_i)_{2k-1} + (\vec{X}_i)_{2k}\Big) \qquad \text{for all} \qquad 1 \leq i \leq d-1 \end{aligned} \tag{1.21}$$

The quantity $\vec{X}_k$ is subject to k **blocking transformations**. We now have d vectors $\vec{X}_0, \vec{X}_1, \cdots, \vec{X}_{d-1}$ containing the subsequent averages of observations. It turns out that if the components of $\vec{X}$ is a stationary time series, then the components of $\vec{X}_i$ is a stationary time series for all $0 \leq i \leq d-1$

We can then compute the autocovariance, the variance, sample mean, and number of observations for each i. Let $\gamma_i, \sigma_i^2, \overline{X}_i$ denote the autocovariance, variance and average of the elements of $\vec{X}_i$ and let n_i be the number of elements of $\vec{X}_i$. It follows by induction that $n_i = n/2^i$.

Blocking Transformations

Using the definition of the blocking transformation and the distributive property of the covariance, it is clear that since $h = |i-j|$ we can define

$$\begin{aligned} \gamma_{k+1}(h) &= cov\left((X_{k+1})_i, (X_{k+1})_j\right) \\ &= \frac{1}{4} cov\left((X_k)_{2i-1} + (X_k)_{2i}, (X_k)_{2j-1} + (X_k)_{2j}\right) \\ &= \frac{1}{2}\gamma_k(2h) + \frac{1}{2}\gamma_k(2h+1) \text{ h} = 0 \qquad (1.22) \\ &= \frac{1}{4}\gamma_k(2h-1) + \frac{1}{2}\gamma_k(2h) + \frac{1}{4}\gamma_k(2h+1) \quad \text{else} \qquad (1.23) \end{aligned}$$

The quantity $\hat{X}$ is asymptotic uncorrelated by assumption, $\hat{X}_k$ is also asymptotic uncorrelated. Let's turn our attention to the variance of the sample mean $V(\overline{X})$.

Blocking Transformations, getting there

We have

$$V(\overline{X}_k)=\frac{\sigma_k^2}{n_k}+\underbrace{\frac{2}{n_k}\sum_{h=1}^{n_k-1}\left(1-\frac{h}{n_k}\right)\gamma_k(h)}_{\equiv e_k}=\frac{\sigma_k^2}{n_k}+e_k \quad \text{if} \quad \gamma_k(0)=\sigma_k^2. \tag{1.24}$$

The term e_k is called the **truncation error**:

$$e_k=\frac{2}{n_k}\sum_{h=1}^{n_k-1}\left(1-\frac{h}{n_k}\right)\gamma_k(h). \tag{1.25}$$

We can show that $V(\overline{X}_i)=V(\overline{X}_j)$ for all $0\leq i\leq d-1$ and $0\leq j\leq d-1$.

Blocking Transformations, final expressions

We can then wrap up

$$\begin{aligned} n_{j+1}\overline{X}_{j+1}&=\sum_{i=1}^{n_{j+1}}(\hat{X}_{j+1})_i=\frac{1}{2}\sum_{i=1}^{n_j/2}(\hat{X}_j)_{2i-1}+(\hat{X}_j)_{2i}\\ &=\frac{1}{2}\left[(\hat{X}_j)_1+(\hat{X}_j)_2+\cdots+(\hat{X}_j)_{n_j}\right]=\underbrace{\frac{n_j}{2}}_{=n_{j+1}}\overline{X}_j=n_{j+1}\overline{X}_j. \end{aligned} \tag{1.26}$$

By repeated use of this equation we get $V(\overline{X}_i)=V(\overline{X}_0)=V(\overline{X})$ for all $0\leq i\leq d-1$. This has the consequence that

$$V(\overline{X})=\frac{\sigma_k^2}{n_k}+e_k \qquad \text{for all} \qquad 0\leq k\leq d-1. \tag{1.27}$$

Flyvbjerg and Petersen demonstrated that the sequence $\{e_k\}_{k=0}^{d-1}$ is decreasing, and conjecture that the term e_k can be made as small as we would like by making k (and hence d) sufficiently large. The sequence is decreasing (Master of Science thesis by Marius Jonsson, UiO 2018). It means we can apply blocking transformations until e_k is sufficiently small, and then estimate $V(\overline{X})$ by $\widehat{\sigma}_k^2/n_k$.

For an elegant solution and proof of the blocking method, see the recent article of Marius Jonsson (former MSc student of the Computational Physics group).

Boltzmann Machines

Why use a generative model rather than the more well known discriminative deep neural networks (DNN)?

- Discriminitave methods have several limitations: They are mainly supervised learning methods, thus requiring labeled data. And there are tasks they cannot accomplish, like drawing new examples from an unknown probability distribution.

- A generative model can learn to represent and sample from a probability distribution. The core idea is to learn a parametric model of the probability distribution from which the training data was drawn. As an example
  1. A model for images could learn to draw new examples of cats and dogs, given a training dataset of images of cats and dogs.
  2. Generate a sample of an ordered or disordered Ising model phase, having been given samples of such phases.
  3. Model the trial function for Monte Carlo calculations
- Both use gradient-descent based learning procedures for minimizing cost functions
- Energy based models don't use backpropagation and automatic differentiation for computing gradients, instead turning to Markov Chain Monte Carlo methods.
- DNNs often have several hidden layers. A restricted Boltzmann machine has only one hidden layer, however several RBMs can be stacked to make up Deep Belief Networks, of which they constitute the building blocks.

History: The RBM was developed by amongst others Geoffrey Hinton, called by some the "Godfather of Deep Learning", working with the University of Toronto and Google.

A BM is what we would call an undirected probabilistic graphical model with stochastic continuous or discrete units.

It is interpreted as a stochastic recurrent neural network where the state of each unit(neurons/nodes) depends on the units it is connected to. The weights in the network represent thus the strength of the interaction between various units/nodes.

It turns into a Hopfield network if we choose deterministic rather than stochastic units. In contrast to a Hopfield network, a BM is a so-called generative model. It allows us to generate new samples from the learned distribution.

A standard BM network is divided into a set of observable and visible units $\hat{x}$ and a set of unknown hidden units/nodes $\hat{h}$.

Additionally there can be bias nodes for the hidden and visible layers. These biases are normally set to 1.

BMs are stackable, meaning they cwe can train a BM which serves as input to another BM. We can construct deep networks for learning complex PDFs. The layers can be trained one after another, a feature which makes them popular in deep learning

However, they are often hard to train. This leads to the introduction of so-called restricted BMs, or RBMS. Here we take away all lateral connections between nodes in the visible layer as well as connections between nodes in the hidden layer. The network is illustrated in the figure below.

The network

The network layers:

1. A function $\mathbf{x}$ that represents the visible layer, a vector of M elements (nodes). This layer represents both what the RBM might be given as training input, and what we want it to be able to reconstruct. This might for example be the pixels of an image, the spin values of the Ising model, or coefficients representing speech.
2. The function $\mathbf{h}$ represents the hidden, or latent, layer. A vector of N elements (nodes). Also called "feature detectors".

The goal of the hidden layer is to increase the model's expressive power. We encode complex interactions between visible variables by introducing additional, hidden variables that interact with visible degrees of freedom in a simple manner, yet still reproduce the complex correlations between visible degrees in the data once marginalized over (integrated out).

Examples of this trick being employed in physics:

1. The Hubbard-Stratonovich transformation
2. The introduction of ghost fields in gauge theory
3. Shadow wave functions in Quantum Monte Carlo simulations

The network parameters, to be optimized/learned:

1. $\mathbf{a}$ represents the visible bias, a vector of same length as $\mathbf{x}$.
2. $\mathbf{b}$ represents the hidden bias, a vector of same lenght as $\mathbf{h}$.
3. W represents the interaction weights, a matrix of size $M \times N$.

Joint distribution.

The restricted Boltzmann machine is described by a Bolztmann distribution

$$P_{rbm}(\mathbf{x},\mathbf{h}) = \frac{1}{Z} e^{-\frac{1}{T_0}E(\mathbf{x},\mathbf{h})}, \tag{1.28}$$

where Z is the normalization constant or partition function, defined as

$$Z = \int\int e^{-\frac{1}{T_0}E(\mathbf{x},\mathbf{h})} d\mathbf{x} d\mathbf{h}. \tag{1.29}$$

It is common to ignore T_0 by setting it to one.

Network Elements, the energy function.

The function $E(\mathbf{x},\mathbf{h})$ gives the **energy** of a configuration (pair of vectors) $(\mathbf{x},\mathbf{h})$. The lower the energy of a configuration, the higher the probability of it. This function also depends on the parameters **a**, **b** and W. Thus, when we adjust them during the learning procedure, we are adjusting the energy function to best fit our problem.

Defining different types of RBMs.

There are different variants of RBMs, and the differences lie in the types of visible and hidden units we choose as well as in the implementation of the energy function $E(\mathbf{x},\mathbf{h})$. The connection between the nodes in the two layers is given by the weights w_{ij}.

Binary-Binary RBM:

RBMs were first developed using binary units in both the visible and hidden layer. The corresponding energy function is defined as follows:

$$E(\mathbf{x},\mathbf{h}) = -\sum_{i}^{M} x_i a_i - \sum_{j}^{N} b_j h_j - \sum_{i,j}^{M,N} x_i w_{ij} h_j, \tag{1.30}$$

where the binary values taken on by the nodes are most commonly 0 and 1.

Gaussian-Binary RBM:

Another varient is the RBM where the visible units are Gaussian while the hidden units remain binary:

$$E(\mathbf{x},\mathbf{h}) = \sum_{i}^{M} \frac{(x_i - a_i)^2}{2\sigma_i^2} - \sum_{j}^{N} b_j h_j - \sum_{i,j}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2}. \tag{1.31}$$

1. RBMs are Useful when we model continuous data (i.e., we wish **x** to be continuous)
2. Requires a smaller learning rate, since there's no upper bound to the value a component might take in the reconstruction

Other types of units include:

1. Softmax and multinomial units
2. Gaussian visible and hidden units
3. Binomial units
4. Rectified linear units

Cost function.

When working with a training dataset, the most common training approach is maximizing the log-likelihood of the training data. The log likelihood characterizes the log-probability of generating the observed data using our generative model. Using this method our cost function is chosen as the negative log-likelihood. The learning then consists of trying to find parameters

that maximize the probability of the dataset, and is known as Maximum Likelihood Estimation (MLE). Denoting the parameters as $\boldsymbol{\theta} = a_1,...,a_M,b_1,...,b_N,w_{11},...,w_{MN}$, the log-likelihood is given by

$$\mathscr{L}(\{\theta_i\}) = \langle \log P_\theta(\boldsymbol{x})\rangle_{data} \tag{1.32}$$

$$= -\langle E(\boldsymbol{x};\{\theta_i\})\rangle_{data} - \log Z(\{\theta_i\}), \tag{1.33}$$

where we used that the normalization constant does not depend on the data, $\langle \log Z(\{\theta_i\})\rangle = \log Z(\{\theta_i\})$ Our cost function is the negative log-likelihood, $\mathscr{C}(\{\theta_i\}) = -\mathscr{L}(\{\theta_i\})$

Optimization / Training.

The training procedure of choice often is Stochastic Gradient Descent (SGD). It consists of a series of iterations where we update the parameters according to the equation

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla \mathscr{C}(\boldsymbol{\theta}_k) \tag{1.34}$$

at each k-th iteration. There are a range of variants of the algorithm which aim at making the learning rate η more adaptive so the method might be more efficient while remaining stable.

We now need the gradient of the cost function in order to minimize it. We find that

$$\frac{\partial \mathscr{C}(\{\theta_i\})}{\partial \theta_i} = \langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i}\rangle_{data} + \frac{\partial \log Z(\{\theta_i\})}{\partial \theta_i} \tag{1.35}$$

$$= \langle O_i(\boldsymbol{x})\rangle_{data} - \langle O_i(\boldsymbol{x})\rangle_{model}, \tag{1.36}$$

where in order to simplify notation we defined the "operator"

$$O_i(\boldsymbol{x}) = \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i}, \tag{1.37}$$

and used the statistical mechanics relationship between expectation values and the log-partition function:

$$\langle O_i(\boldsymbol{x})\rangle_{model} = \mathrm{Tr} P_\theta(\boldsymbol{x}) O_i(\boldsymbol{x}) = -\frac{\partial \log Z(\{\theta_i\})}{\partial \theta_i}. \tag{1.38}$$

The data-dependent term in the gradient is known as the positive phase of the gradient, while the model-dependent term is known as the negative phase of the gradient. The aim of the training is to lower the energy of configurations that are near observed data points (increasing their probability), and raising the energy of configurations that are far from observed data points (decreasing their probability).

The gradient of the negative log-likelihood cost function of a Binary-Binary RBM is then

$$\frac{\partial \mathscr{C}(w_{ij},a_i,b_j)}{\partial w_{ij}} = \langle x_i h_j\rangle_{data} - \langle x_i h_j\rangle_{model} \tag{1.39}$$

$$\frac{\partial \mathscr{C}(w_{ij},a_i,b_j)}{\partial a_{ij}} = \langle x_i\rangle_{data} - \langle x_i\rangle_{model} \tag{1.40}$$

$$\frac{\partial \mathscr{C}(w_{ij},a_i,b_j)}{\partial b_{ij}} = \langle h_i\rangle_{data} - \langle h_i\rangle_{model}. \tag{1.41}$$

$$\tag{1.42}$$

To get the expectation values with respect to the *data*, we set the visible units to each of the observed samples in the training data, then update the hidden units according to the conditional probability found before. We then average over all samples in the training data to calculate expectation values with respect to the data.

Kullback-Leibler relative entropy.

When the goal of the training is to approximate a probability distribution, as it is in generative modeling, another relevant measure is the **Kullback-Leibler divergence**, also known as the relative entropy or Shannon entropy. It is a non-symmetric measure of the dissimilarity between two probability density functions p and q. If p is the unkown probability which we approximate with q, we can measure the difference by

$$\mathrm{KL}(p||q) = \int_{-\infty}^{\infty} p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} d\boldsymbol{x}. \tag{1.43}$$

Thus, the Kullback-Leibler divergence between the distribution of the training data $f(\boldsymbol{x})$ and the model distribution $p(\boldsymbol{x}|\boldsymbol{\theta})$ is

$$\begin{aligned}
\mathrm{KL}(f(\boldsymbol{x})||p(\boldsymbol{x}|\boldsymbol{\theta})) &= \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log \frac{f(\boldsymbol{x})}{p(\boldsymbol{x}|\boldsymbol{\theta})} d\boldsymbol{x} && (1.44)\\
&= \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log f(\boldsymbol{x}) d\boldsymbol{x} - \int_{-\infty}^{\infty} f(\boldsymbol{x}) \log p(\boldsymbol{x}|\boldsymbol{\theta}) d\boldsymbol{x} && (1.45)\\
&= \langle \log f(\boldsymbol{x}) \rangle_{f(\boldsymbol{x})} - \langle \log p(\boldsymbol{x}|\boldsymbol{\theta}) \rangle_{f(\boldsymbol{x})} && (1.46)\\
&= \langle \log f(\boldsymbol{x}) \rangle_{data} + \langle E(\boldsymbol{x}) \rangle_{data} + \log Z && (1.47)\\
&= \langle \log f(\boldsymbol{x}) \rangle_{data} + \mathscr{C}_{LL}. && (1.48)
\end{aligned}$$

The first term is constant with respect to $\boldsymbol{\theta}$ since $f(\boldsymbol{x})$ is independent of $\boldsymbol{\theta}$. Thus the Kullback-Leibler Divergence is minimal when the second term is minimal. The second term is the log-likelihood cost function, hence minimizing the Kullback-Leibler divergence is equivalent to maximizing the log-likelihood.

To further understand generative models it is useful to study the gradient of the cost function which is needed in order to minimize it using methods like stochastic gradient descent.

The partition function is the generating function of expectation values, in particular there are mathematical relationships between expectation values and the log-partition function. In this case we have

$$\langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \rangle_{model} = \int p(\boldsymbol{x}|\boldsymbol{\theta}) \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} d\boldsymbol{x} = -\frac{\partial \log Z(\theta_i)}{\partial \theta_i}. \tag{1.49}$$

Here $\langle \cdot \rangle_{model}$ is the expectation value over the model probability distribution $p(\boldsymbol{x}|\boldsymbol{\theta})$.

Setting up for gradient descent calculations

Using the previous relationship we can express the gradient of the cost function as

$$\frac{\partial \mathscr{C}_{LL}}{\partial \theta_i} = \langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \rangle_{data} + \frac{\partial \log Z(\theta_i)}{\partial \theta_i} \tag{1.50}$$

$$= \langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \rangle_{data} - \langle \frac{\partial E(\boldsymbol{x};\theta_i)}{\partial \theta_i} \rangle_{model} \tag{1.51}$$

$$\tag{1.52}$$

This expression shows that the gradient of the log-likelihood cost function is a **difference of moments**, with one calculated from the data and one calculated from the model. The data-dependent term is called the **positive phase** and the model-dependent term is called the **negative phase** of the gradient. We see now that minimizing the cost function results in lowering the energy of configurations $\boldsymbol{x}$ near points in the training data and increasing the energy of configurations not observed in the training data. That means we increase the model's probability of configurations similar to those in the training data.

The gradient of the cost function also demonstrates why gradients of unsupervised, generative models must be computed differently from for those of for example FNNs. While the data-dependent expectation value is easily calculated based on the samples $\boldsymbol{x}_i$ in the training data, we must sample from the model in order to generate samples from which to caclulate the model-dependent term. We sample from the model by using MCMC-based methods. We can not sample from the model directly because the partition function Z is generally intractable.

As in supervised machine learning problems, the goal is also here to perform well on **unseen** data, that is to have good generalization from the training data. The distribution $f(x)$ we approximate is not the **true** distribution we wish to estimate, it is limited to the training data. Hence, in unsupervised training as well it is important to prevent overfitting to the training data. Thus it is common to add regularizers to the cost function in the same manner as we discussed for say linear regression.

RBMs for the quantum many body problem

The idea of applying RBMs to quantum many body problems was presented by G. Carleo and M. Troyer, working with ETH Zurich and Microsoft Research.

Some of their motivation included

- The wave function Ψ is a monolithic mathematical quantity that contains all the information on a quantum state, be it a single particle or a complex molecule. In principle, an exponential amount of information is needed to fully encode a generic many-body quantum state.
- There are still interesting open problems, including fundamental questions ranging from the dynamical properties of high-dimensional systems to the exact ground-state properties of strongly interacting fermions.
- The difficulty lies in finding a general strategy to reduce the exponential complexity of the full many-body wave function down to its most essential features. That is

  1. Dimensional reduction
  2. Feature extraction

- Among the most successful techniques to attack these challenges, artifical neural networks play a prominent role.
- Want to understand whether an artifical neural network may adapt to describe a quantum system.

Carleo and Troyer applied the RBM to the quantum mechanical spin lattice systems of the Ising model and Heisenberg model, with encouraging results. Our goal is to test the method on systems of moving particles. For the spin lattice systems it was natural to use a binary-binary RBM, with the nodes taking values of 1 and -1. For moving particles, on the other hand, we want the visible nodes to be continuous, representing position coordinates. Thus, we start by choosing a Gaussian-binary RBM, where the visible nodes are continuous and hidden nodes take on values of 0 or 1. If eventually we would like the hidden nodes to be continuous as well the rectified linear units seem like the most relevant choice.

Representing the wave function

The wavefunction should be a probability amplitude depending on $\boldsymbol{x}$. The RBM model is given by the joint distribution of $\boldsymbol{x}$ and $\boldsymbol{h}$

$$F_{rbm}(\mathbf{x},\mathbf{h}) = \frac{1}{Z} e^{-\frac{1}{T_0}E(\mathbf{x},\mathbf{h})}. \tag{1.53}$$

To find the marginal distribution of $\boldsymbol{x}$ we set:

$$F_{rbm}(\mathbf{x}) = \sum_{\mathbf{h}} F_{rbm}(\mathbf{x},\mathbf{h}) \tag{1.54}$$

$$= \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{x},\mathbf{h})}. \tag{1.55}$$

Now this is what we use to represent the wave function, calling it a neural-network quantum state (NQS)

$$\Psi(\mathbf{X}) = F_{rbm}(\mathbf{x}) \tag{1.56}$$

$$= \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\mathbf{x},\mathbf{h})} \tag{1.57}$$

$$= \frac{1}{Z} \sum_{\{h_j\}} e^{-\sum_i^M \frac{(x_i-a_i)^2}{2\sigma^2} + \sum_j^N b_j h_j + \sum_{i,j^{M,N}} \frac{x_i w_{ij} h_j}{\sigma^2}} \tag{1.58}$$

$$= \frac{1}{Z} e^{-\sum_i^M \frac{(x_i-a_i)^2}{2\sigma^2}} \prod_j^N (1 + e^{b_j + \sum_i^M \frac{x_i w_{ij}}{\sigma^2}}). \tag{1.59}$$

$$\tag{1.60}$$

Choose the cost function

Now we don't necessarily have training data (unless we generate it by using some other method). However, what we do have is the variational principle which allows us to obtain the ground state wave function by minimizing the expectation value of the energy of a trial wavefunction (corresponding to the untrained NQS). Similarly to the traditional variational Monte Carlo method then, it is the local energy we wish to minimize. The gradient to use for the stochastic gradient descent procedure is

$$C_i = \frac{\partial \langle E_L \rangle}{\partial \theta_i} = 2(\langle E_L \frac{1}{\Psi}\frac{\partial \Psi}{\partial \theta_i}\rangle - \langle E_L \rangle \langle \frac{1}{\Psi}\frac{\partial \Psi}{\partial \theta_i}\rangle), \tag{1.61}$$

where the local energy is given by

$$E_L = \frac{1}{\Psi}\hat{\mathbf{H}}\Psi. \tag{1.62}$$

Mathematical details.

Because we are restricted to potential functions which are positive it is convenient to express them as exponentials, so that

$$\phi_C(\boldsymbol{x}_C) = e^{-E_C(\boldsymbol{x}_C)} \tag{1.63}$$

where $E(\boldsymbol{x}_C)$ is called an *energy function*, and the exponential representation is the *Boltzmann distribution*. The joint distribution is defined as the product of potentials.

The joint distribution of the random variables is then

$$\begin{aligned} p(\boldsymbol{x}) &= \frac{1}{Z}\prod_C \phi_C(\boldsymbol{x}_C) \\ &= \frac{1}{Z}\prod_C e^{-E_C(\boldsymbol{x}_C)} \\ &= \frac{1}{Z}e^{-\sum_C E_C(\boldsymbol{x}_C)} \\ &= \frac{1}{Z}e^{-E(\boldsymbol{x})}. \end{aligned} \tag{1.64}$$

$$p_{BM}(\boldsymbol{x},\boldsymbol{h}) = \frac{1}{Z_{BM}}e^{-\frac{1}{T}E_{BM}(\boldsymbol{x},\boldsymbol{h})}, \tag{1.65}$$

with the partition function

$$Z_{BM} = \int\int e^{-\frac{1}{T}E_{BM}(\tilde{\boldsymbol{x}},\tilde{\boldsymbol{h}})}d\tilde{\boldsymbol{x}}d\tilde{\boldsymbol{h}}. \tag{1.66}$$

T is a physics-inspired parameter named temperature and will be assumed to be 1 unless otherwise stated. The energy function of the Boltzmann machine determines the interactions between the nodes and is defined

$$\begin{aligned} E_{BM}(\boldsymbol{x},\boldsymbol{h}) = &-\sum_{i,k}^{M,K} a_i^k \alpha_i^k(x_i) - \sum_{j,l}^{N,L} b_j^l \beta_j^l(h_j) - \sum_{i,j,k,l}^{M,N,K,L} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(h_j) \\ &- \sum_{i,m=i+1,k}^{M,M,K} \alpha_i^k(x_i) v_{im}^k \alpha_m^k(x_m) - \sum_{j,n=j+1,l}^{N,N,L} \beta_j^l(h_j) u_{jn}^l \beta_n^l(h_n). \end{aligned} \tag{1.67}$$

Here $\alpha_i^k(x_i)$ and $\beta_j^l(h_j)$ are one-dimensional transfer functions or mappings from the given input value to the desired feature value. They can be arbitrary functions of the input variables and are independent of the parameterization (parameters referring to weight and biases), meaning they are not affected by training of the model. The indices k and l indicate that there can be multiple transfer functions per variable. Furthermore, a_i^k and b_j^l are the visible and

hidden bias. w_{ij}^{kl} are weights of the **inter-layer** connection terms which connect visible and hidden units. v_{im}^{k} and u_{jn}^{l} are weights of the **intra-layer** connection terms which connect the visible units to each other and the hidden units to each other, respectively.

We remove the intra-layer connections by setting v_{im} and u_{jn} to zero. The expression for the energy of the RBM is then

$$E_{RBM}(\boldsymbol{x},\boldsymbol{h}) = -\sum_{i,k}^{M,K} a_i^k \alpha_i^k(x_i) - \sum_{j,l}^{N,L} b_j^l \beta_j^l(h_j) - \sum_{i,j,k,l}^{M,N,K,L} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(h_j). \tag{1.68}$$

resulting in

$$\begin{aligned}
P_{RBM}(\boldsymbol{x}) &= \int P_{RBM}(\boldsymbol{x},\tilde{\boldsymbol{h}}) d\tilde{\boldsymbol{h}} \\
&= \frac{1}{Z_{RBM}} \int e^{-E_{RBM}(\boldsymbol{x},\tilde{\boldsymbol{h}})} d\tilde{\boldsymbol{h}} \\
&= \frac{1}{Z_{RBM}} \int e^{\sum_{i,k} a_i^k \alpha_i^k(x_i) + \sum_{j,l} b_j^l \beta_j^l(\tilde{h}_j) + \sum_{i,j,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(\tilde{h}_j)} d\tilde{\boldsymbol{h}} \\
&= \frac{1}{Z_{RBM}} e^{\sum_{i,k} a_i^k \alpha_i^k(x_i)} \int \prod_j^N e^{\sum_l b_j^l \beta_j^l(\tilde{h}_j) + \sum_{i,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(\tilde{h}_j)} d\tilde{\boldsymbol{h}} \\
&= \frac{1}{Z_{RBM}} e^{\sum_{i,k} a_i^k \alpha_i^k(x_i)} \left(\int e^{\sum_l b_1^l \beta_1^l(\tilde{h}_1) + \sum_{i,k,l} \alpha_i^k(x_i) w_{i1}^{kl} \beta_1^l(\tilde{h}_1)} d\tilde{h}_1 \right. \\
&\quad \times \int e^{\sum_l b_2^l \beta_2^l(\tilde{h}_2) + \sum_{i,k,l} \alpha_i^k(x_i) w_{i2}^{kl} \beta_2^l(\tilde{h}_2)} d\tilde{h}_2 \\
&\quad \times \ldots \\
&\quad \left. \times \int e^{\sum_l b_N^l \beta_N^l(\tilde{h}_N) + \sum_{i,k,l} \alpha_i^k(x_i) w_{iN}^{kl} \beta_N^l(\tilde{h}_N)} d\tilde{h}_N \right) \\
&= \frac{1}{Z_{RBM}} e^{\sum_{i,k} a_i^k \alpha_i^k(x_i)} \prod_j^N \int e^{\sum_l b_j^l \beta_j^l(\tilde{h}_j) + \sum_{i,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(\tilde{h}_j)} d\tilde{h}_j
\end{aligned} \tag{1.69}$$

Similarly

$$\begin{aligned}
P_{RBM}(\boldsymbol{h}) &= \frac{1}{Z_{RBM}} \int e^{-E_{RBM}(\tilde{\mathbf{x}},\boldsymbol{h})} d\tilde{\boldsymbol{x}} \\
&= \frac{1}{Z_{RBM}} e^{\sum_{j,l} b_j^l \beta_j^l(h_j)} \prod_i^M \int e^{\sum_k a_i^k \alpha_i^k(\tilde{x}_i) + \sum_{j,k,l} \alpha_i^k(\tilde{x}_i) w_{ij}^{kl} \beta_j^l(h_j)} d\tilde{x}_i
\end{aligned} \tag{1.70}$$

Using Bayes theorem

$$\begin{aligned}
P_{RBM}(\boldsymbol{h}|\boldsymbol{x}) &= \frac{P_{RBM}(\boldsymbol{x},\boldsymbol{h})}{P_{RBM}(\boldsymbol{x})} \\
&= \frac{\frac{1}{Z_{RBM}} e^{\sum_{i,k} a_i^k \alpha_i^k(x_i) + \sum_{j,l} b_j^l \beta_j^l(h_j) + \sum_{i,j,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(h_j)}}{\frac{1}{Z_{RBM}} e^{\sum_{i,k} a_i^k \alpha_i^k(x_i)} \prod_j^N \int e^{\sum_l b_j^l \beta_j^l(\tilde{h}_j) + \sum_{i,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(\tilde{h}_j)} d\tilde{h}_j} \\
&= \prod_j^N \frac{e^{\sum_l b_j^l \beta_j^l(h_j) + \sum_{i,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(h_j)}}{\int e^{\sum_l b_j^l \beta_j^l(\tilde{h}_j) + \sum_{i,k,l} \alpha_i^k(x_i) w_{ij}^{kl} \beta_j^l(\tilde{h}_j)} d\tilde{h}_j}
\end{aligned} \tag{1.71}$$

Similarly

$$
\begin{aligned}
P_{RBM}(\boldsymbol{x}|\boldsymbol{h}) =& \frac{P_{RBM}(\boldsymbol{x},\boldsymbol{h})}{P_{RBM}(\boldsymbol{h})} \\
=& \prod_i^M \frac{e^{\sum_k a_i^k \alpha_i^k(x_i)+\sum_{j,k,l}\alpha_i^k(x_i)w_{ij}^{kl}\beta_j^l(h_j)}}{\int e^{\sum_k a_i^k \alpha_i^k(\tilde{x}_i)+\sum_{j,k,l}\alpha_i^k(\tilde{x}_i)w_{ij}^{kl}\beta_j^l(h_j)}d\tilde{x}_i}
\end{aligned} \tag{1.72}
$$

The original RBM had binary visible and hidden nodes. They were showned to be universal approximators of discrete distributions. It was also shown that adding hidden units yields strictly improved modelling power. The common choice of binary values are 0 and 1. However, in some physics applications, -1 and 1 might be a more natural choice. We will here use 0 and 1.

$$
E_{BB}(\boldsymbol{x},\mathbf{h}) = -\sum_i^M x_i a_i - \sum_j^N b_j h_j - \sum_{i,j}^{M,N} x_i w_{ij} h_j. \tag{1.73}
$$

$$
p_{BB}(\boldsymbol{x},\boldsymbol{h}) = \frac{1}{Z_{BB}} e^{\sum_i^M a_i x_i + \sum_j^N b_j h_j + \sum_{ij}^{M,N} x_i w_{ij} h_j} \tag{1.74}
$$

$$
= \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T\boldsymbol{a}+\boldsymbol{b}^T\boldsymbol{h}+\boldsymbol{x}^T\boldsymbol{W}\boldsymbol{h}} \tag{1.75}
$$

with the partition function

$$
Z_{BB} = \sum_{\boldsymbol{x},\boldsymbol{h}} e^{\boldsymbol{x}^T\boldsymbol{a}+\boldsymbol{b}^T\boldsymbol{h}+\boldsymbol{x}^T\boldsymbol{W}\boldsymbol{h}}. \tag{1.76}
$$

Marginal Probability Density Functions.

In order to find the probability of any configuration of the visible units we derive the marginal probability density function.

$$
\begin{aligned}
p_{BB}(\boldsymbol{x}) &= \sum_{\boldsymbol{h}} p_{BB}(\boldsymbol{x},\boldsymbol{h}) \qquad (1.77)\\
&= \frac{1}{Z_{BB}} \sum_{\boldsymbol{h}} e^{\boldsymbol{x}^T\boldsymbol{a}+\boldsymbol{b}^T\boldsymbol{h}+\boldsymbol{x}^T\boldsymbol{W}\boldsymbol{h}}\\
&= \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T\boldsymbol{a}} \sum_{\boldsymbol{h}} e^{\sum_j^N (b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})h_j}\\
&= \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T\boldsymbol{a}} \sum_{\boldsymbol{h}} \prod_j^N e^{(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})h_j}\\
&= \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T\boldsymbol{a}} \left(\sum_{h_1} e^{(b_1+\boldsymbol{x}^T\boldsymbol{w}_{*1})h_1} \times \sum_{h_2} e^{(b_2+\boldsymbol{x}^T\boldsymbol{w}_{*2})h_2} \times \right.\\
&\quad \left. \ldots \times \sum_{h_2} e^{(b_N+\boldsymbol{x}^T\boldsymbol{w}_{*N})h_N} \right)\\
&= \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T\boldsymbol{a}} \prod_j^N \sum_{h_j} e^{(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})h_j}\\
&= \frac{1}{Z_{BB}} e^{\boldsymbol{x}^T\boldsymbol{a}} \prod_j^N (1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}). \qquad (1.78)
\end{aligned}
$$

A similar derivation yields the marginal probability of the hidden units

$$p_{BB}(\boldsymbol{h}) = \frac{1}{Z_{BB}} e^{\boldsymbol{b}^T\boldsymbol{h}} \prod_i^M (1+e^{a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h}}). \qquad (1.79)$$

Conditional Probability Density Functions.

We derive the probability of the hidden units given the visible units using Bayes' rule

$$
\begin{aligned}
p_{BB}(\boldsymbol{h}|\boldsymbol{x}) &= \frac{p_{BB}(\boldsymbol{x},\boldsymbol{h})}{p_{BB}(\boldsymbol{x})}\\
&= \frac{\frac{1}{Z_{BB}} e^{\boldsymbol{x}^T\boldsymbol{a}+\boldsymbol{b}^T\boldsymbol{h}+\boldsymbol{x}^T\boldsymbol{W}\boldsymbol{h}}}{\frac{1}{Z_{BB}} e^{\boldsymbol{x}^T\boldsymbol{a}} \prod_j^N (1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}})}\\
&= \frac{e^{\boldsymbol{x}^T\boldsymbol{a}} e^{\sum_j^N (b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})h_j}}{e^{\boldsymbol{x}^T\boldsymbol{a}} \prod_j^N (1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}})}\\
&= \prod_j^N \frac{e^{(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})h_j}}{1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}}\\
&= \prod_j^N p_{BB}(h_j|\boldsymbol{x}). \qquad (1.80)
\end{aligned}
$$

From this we find the probability of a hidden unit being "on" or "off":

$$p_{BB}(h_j=1|\boldsymbol{x}) = \frac{e^{(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})h_j}}{1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}} \tag{1.81}$$

$$= \frac{e^{(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})}}{1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}} \tag{1.82}$$

$$= \frac{1}{1+e^{-(b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j})}}, \tag{1.83}$$

and

$$p_{BB}(h_j=0|\boldsymbol{x}) = \frac{1}{1+e^{b_j+\boldsymbol{x}^T\boldsymbol{w}_{*j}}}. \tag{1.84}$$

Similarly we have that the conditional probability of the visible units given the hidden are

$$p_{BB}(\boldsymbol{x}|\boldsymbol{h}) = \prod_i^M \frac{e^{(a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h})x_i}}{1+e^{a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h}}} \tag{1.85}$$

$$= \prod_i^M p_{BB}(x_i|\boldsymbol{h}). \tag{1.86}$$

$$p_{BB}(x_i=1|\boldsymbol{h}) = \frac{1}{1+e^{-(a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h})}} \tag{1.87}$$

$$p_{BB}(x_i=0|\boldsymbol{h}) = \frac{1}{1+e^{a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h}}}. \tag{1.88}$$

Gaussian-Binary Restricted Boltzmann Machines.

Inserting into the expression for $E_{RBM}(\boldsymbol{x},\boldsymbol{h})$ in equation results in the energy

$$E_{GB}(\boldsymbol{x},\boldsymbol{h}) = \sum_i^M \frac{(x_i-a_i)^2}{2\sigma_i^2} - \sum_j^N b_j h_j - \sum_{ij}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2}$$

$$= ||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 - \boldsymbol{b}^T\boldsymbol{h} - (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T\boldsymbol{W}\boldsymbol{h}. \tag{1.89}$$

Joint Probability Density Function.

$$p_{GB}(\boldsymbol{x},\boldsymbol{h}) = \frac{1}{Z_{GB}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 + \boldsymbol{b}^T\boldsymbol{h} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T\boldsymbol{W}\boldsymbol{h}}$$

$$= \frac{1}{Z_{GB}} e^{-\sum_i^M \frac{(x_i-a_i)^2}{2\sigma_i^2} + \sum_j^N b_j h_j + \sum_{ij}^{M,N} \frac{x_i w_{ij} h_j}{\sigma_i^2}}$$

$$= \frac{1}{Z_{GB}} \prod_{ij}^{M,N} e^{-\frac{(x_i-a_i)^2}{2\sigma_i^2} + b_j h_j + \frac{x_i w_{ij} h_j}{\sigma_i^2}}, \tag{1.90}$$

with the partition function given by

$$Z_{GB} = \int \sum_{\tilde{\boldsymbol{h}}}^{\tilde{\boldsymbol{H}}} e^{-||\frac{\tilde{\boldsymbol{x}}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 + \boldsymbol{b}^T\tilde{\boldsymbol{h}} + (\frac{\tilde{\boldsymbol{x}}}{\boldsymbol{\sigma}^2})^T\boldsymbol{W}\tilde{\boldsymbol{h}}} d\tilde{\boldsymbol{x}}. \tag{1.91}$$

Marginal Probability Density Functions.

We proceed to find the marginal probability densitites of the Gaussian-binary RBM. We first marginalize over the binary hidden units to find $p_{GB}(\boldsymbol{x})$

$$\begin{aligned} p_{GB}(\boldsymbol{x}) =& \sum_{\tilde{\boldsymbol{h}}}^{\tilde{\boldsymbol{H}}} p_{GB}(\boldsymbol{x}, \tilde{\boldsymbol{h}}) \\ =& \frac{1}{Z_{GB}} \sum_{\tilde{\boldsymbol{h}}}^{\tilde{\boldsymbol{H}}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 + \boldsymbol{b}^T\tilde{\boldsymbol{h}} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T\boldsymbol{W}\tilde{\boldsymbol{h}}} \\ =& \frac{1}{Z_{GB}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2} \prod_j^N (1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T\boldsymbol{w}_{*j}}). \end{aligned} \tag{1.92}$$

We next marginalize over the visible units. This is the first time we marginalize over continuous values. We rewrite the exponential factor dependent on $\boldsymbol{x}$ as a Gaussian function before we integrate in the last step.

$$
\begin{aligned}
p_{GB}(\boldsymbol{h}) &= \int p_{GB}(\tilde{\boldsymbol{x}},\boldsymbol{h})d\tilde{\boldsymbol{x}} \\
&= \frac{1}{Z_{GB}}\int e^{-||\frac{\tilde{\boldsymbol{x}}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2+\boldsymbol{b}^T\boldsymbol{h}+(\frac{\tilde{\boldsymbol{x}}}{\boldsymbol{\sigma}^2})^T\boldsymbol{W}\boldsymbol{h}}d\tilde{\boldsymbol{x}} \\
&= \frac{1}{Z_{GB}}e^{\boldsymbol{b}^T\boldsymbol{h}}\int\prod_i^M e^{-\frac{(\tilde{x}_i-a_i)^2}{2\sigma_i^2}+\frac{\tilde{x}_i\boldsymbol{w}_{i*}^T\boldsymbol{h}}{\sigma_i^2}}d\tilde{\boldsymbol{x}} \\
&= \frac{1}{Z_{GB}}e^{\boldsymbol{b}^T\boldsymbol{h}}\Big(\int e^{-\frac{(\tilde{x}_1-a_1)^2}{2\sigma_1^2}+\frac{\tilde{x}_1\boldsymbol{w}_{1*}^T\boldsymbol{h}}{\sigma_1^2}}d\tilde{x}_1 \\
&\quad\times\int e^{-\frac{(\tilde{x}_2-a_2)^2}{2\sigma_2^2}+\frac{\tilde{x}_2\boldsymbol{w}_{2*}^T\boldsymbol{h}}{\sigma_2^2}}d\tilde{x}_2 \\
&\quad\times\ldots \\
&\quad\times\int e^{-\frac{(\tilde{x}_M-a_M)^2}{2\sigma_M^2}+\frac{\tilde{x}_M\boldsymbol{w}_{M*}^T\boldsymbol{h}}{\sigma_M^2}}d\tilde{x}_M\Big) \\
&= \frac{1}{Z_{GB}}e^{\boldsymbol{b}^T\boldsymbol{h}}\prod_i^M\int e^{-\frac{(\tilde{x}_i-a_i)^2-2\tilde{x}_i\boldsymbol{w}_{i*}^T\boldsymbol{h}}{2\sigma_i^2}}d\tilde{x}_i \\
&= \frac{1}{Z_{GB}}e^{\boldsymbol{b}^T\boldsymbol{h}}\prod_i^M\int e^{-\frac{\tilde{x}_i^2-2\tilde{x}_i(a_i+\tilde{x}_i\boldsymbol{w}_{i*}^T\boldsymbol{h})+a_i^2}{2\sigma_i^2}}d\tilde{x}_i \\
&= \frac{1}{Z_{GB}}e^{\boldsymbol{b}^T\boldsymbol{h}}\prod_i^M\int e^{-\frac{\tilde{x}_i^2-2\tilde{x}_i(a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h})+(a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h})^2-(a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h})^2+a_i^2}{2\sigma_i^2}}d\tilde{x}_i \\
&= \frac{1}{Z_{GB}}e^{\boldsymbol{b}^T\boldsymbol{h}}\prod_i^M\int e^{-\frac{(\tilde{x}_i-(a_i+\boldsymbol{w}_{i*}^T\boldsymbol{h}))^2-a_i^2-2a_i\boldsymbol{w}_{i*}^T\boldsymbol{h}-(\boldsymbol{w}_{i*}^T\boldsymbol{h})^2+a_i^2}{2\sigma_i^2}}d\tilde{x}_i \\
&= \frac{1}{Z_{GB}}e^{\boldsymbol{b}^T\boldsymbol{h}}\prod_i^M e^{\frac{2a_i\boldsymbol{w}_{i*}^T\boldsymbol{h}+(\boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}}\int e^{-\frac{(\tilde{x}_i-a_i-\boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}}d\tilde{x}_i \\
&= \frac{1}{Z_{GB}}e^{\boldsymbol{b}^T\boldsymbol{h}}\prod_i^M\sqrt{2\pi\sigma_i^2}e^{\frac{2a_i\boldsymbol{w}_{i*}^T\boldsymbol{h}+(\boldsymbol{w}_{i*}^T\boldsymbol{h})^2}{2\sigma_i^2}}.
\end{aligned}
\tag{1.93}
$$

Conditional Probability Density Functions.

We finish by deriving the conditional probabilities.

$$
\begin{aligned}
p_{GB}(\boldsymbol{h}|\boldsymbol{x}) &= \frac{p_{GB}(\boldsymbol{x},\boldsymbol{h})}{p_{GB}(\boldsymbol{x})} \\
&= \frac{\frac{1}{Z_{GB}}e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2+\boldsymbol{b}^T\boldsymbol{h}+(\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T\boldsymbol{W}\boldsymbol{h}}}{\frac{1}{Z_{GB}}e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2}\prod_j^N(1+e^{b_j+(\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T\boldsymbol{w}_{*j}})} \\
&= \prod_j^N\frac{e^{(b_j+(\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T\boldsymbol{w}_{*j})h_j}}{1+e^{b_j+(\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T\boldsymbol{w}_{*j}}} \\
&= \prod_j^N p_{GB}(h_j|\boldsymbol{x}).
\end{aligned}
\tag{1.94}
$$

The conditional probability of a binary hidden unit h_j being on or off again takes the form of a sigmoid function

$$p_{GB}(h_j = 1|\boldsymbol{x}) = \frac{e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}}{1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}} = \frac{1}{1 + e^{-b_j - (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}} \tag{1.95}$$

$$p_{GB}(h_j = 0|\boldsymbol{x}) = \frac{1}{1 + e^{b_j + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{w}_{*j}}}. \tag{1.96}$$

The conditional probability of the continuous $\boldsymbol{x}$ now has another form, however.

$$\begin{aligned}
p_{GB}(\boldsymbol{x}|\boldsymbol{h}) &= \frac{p_{GB}(\boldsymbol{x}, \boldsymbol{h})}{p_{GB}(\boldsymbol{h})} \\
&= \frac{\frac{1}{Z_{GB}} e^{-||\frac{\boldsymbol{x}-\boldsymbol{a}}{2\boldsymbol{\sigma}}||^2 + \boldsymbol{b}^T \boldsymbol{h} + (\frac{\boldsymbol{x}}{\boldsymbol{\sigma}^2})^T \boldsymbol{W} \boldsymbol{h}}}{\frac{1}{Z_{GB}} e^{\boldsymbol{b}^T \boldsymbol{h}} \prod_i^M \sqrt{2\pi\sigma_i^2} e^{\frac{2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}}} \\
&= \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} \frac{e^{-\frac{(x_i - a_i)^2}{2\sigma_i^2} + \frac{x_i \boldsymbol{w}_{i*}^T \boldsymbol{h}}{2\sigma_i^2}}}{e^{\frac{2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}}} \\
&= \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} \frac{e^{-\frac{x_i^2 - 2a_i x_i + a_i^2 - 2x_i \boldsymbol{w}_{i*}^T \boldsymbol{h}}{2\sigma_i^2}}}{e^{\frac{2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}}} \\
&= \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{x_i^2 - 2a_i x_i + a_i^2 - 2x_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + 2a_i \boldsymbol{w}_{i*}^T \boldsymbol{h} + (\boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}} \\
&= \prod_i^M \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(x_i - b_i - \boldsymbol{w}_{i*}^T \boldsymbol{h})^2}{2\sigma_i^2}} \\
&= \prod_i^M \mathcal{N}(x_i | b_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}, \sigma_i^2)
\end{aligned} \tag{1.97}$$

$$\Rightarrow p_{GB}(x_i|\boldsymbol{h}) = \mathcal{N}(x_i | b_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}, \sigma_i^2). \tag{1.98}$$

The form of these conditional probabilities explains the name "Gaussian" and the form of the Gaussian-binary energy function. We see that the conditional probability of x_i given $\boldsymbol{h}$ is a normal distribution with mean $b_i + \boldsymbol{w}_{i*}^T \boldsymbol{h}$ and variance σ_i^2.

Neural Quantum States

The wavefunction should be a probability amplitude depending on $\boldsymbol{x}$. The RBM model is given by the joint distribution of $\boldsymbol{x}$ and $\boldsymbol{h}$

$$F_{rbm}(\boldsymbol{x}, \mathbf{h}) = \frac{1}{Z} e^{-\frac{1}{T_0} E(\boldsymbol{x}, \mathbf{h})} \tag{1.99}$$

To find the marginal distribution of $\boldsymbol{x}$ we set:

$$F_{rbm}(\mathbf{x}) = \sum_{\mathbf{h}} F_{rbm}(\mathbf{x},\mathbf{h}) \tag{1.100}$$

$$= \frac{1}{Z}\sum_{\mathbf{h}} e^{-E(\mathbf{x},\mathbf{h})} \tag{1.101}$$

Now this is what we use to represent the wave function, calling it a neural-network quantum state (NQS)

$$\Psi(\mathbf{X}) = F_{rbm}(\mathbf{x}) \tag{1.102}$$

$$= \frac{1}{Z}\sum_{\boldsymbol{h}} e^{-E(\mathbf{x},\mathbf{h})} \tag{1.103}$$

$$= \frac{1}{Z}\sum_{\{h_j\}} e^{-\sum_i^M \frac{(x_i-a_i)^2}{2\sigma^2}+\sum_j^N b_j h_j+\sum_{i,j}^{M,N}\frac{x_i w_{ij} h_j}{\sigma^2}} \tag{1.104}$$

$$= \frac{1}{Z} e^{-\sum_i^M \frac{(x_i-a_i)^2}{2\sigma^2}} \prod_j^N (1+e^{b_j+\sum_i^M \frac{x_i w_{ij}}{\sigma^2}}) \tag{1.105}$$

$$\tag{1.106}$$

The above wavefunction is the most general one because it allows for complex valued wavefunctions. However it fundamentally changes the probabilistic foundation of the RBM, because what is usually a probability in the RBM framework is now a an amplitude. This means that a lot of the theoretical framework usually used to interpret the model, i.e. graphical models, conditional probabilities, and Markov random fields, breaks down. If we assume the wavefunction to be postive definite, however, we can use the RBM to represent the squared wavefunction, and thereby a probability. This also makes it possible to sample from the model using Gibbs sampling, because we can obtain the conditional probabilities.

$$|\Psi(\mathbf{X})|^2 = F_{rbm}(\mathbf{X}) \tag{1.107}$$

$$\Rightarrow \Psi(\mathbf{X}) = \sqrt{F_{rbm}(\mathbf{X})} \tag{1.108}$$

$$= \frac{1}{\sqrt{Z}}\sqrt{\sum_{\{h_j\}} e^{-E(\mathbf{X},\mathbf{h})}} \tag{1.109}$$

$$= \frac{1}{\sqrt{Z}}\sqrt{\sum_{\{h_j\}} e^{-\sum_i^M \frac{(X_i-a_i)^2}{2\sigma^2}+\sum_j^N b_j h_j+\sum_{i,j}^{M,N}\frac{X_i w_{ij} h_j}{\sigma^2}}} \tag{1.110}$$

$$= \frac{1}{\sqrt{Z}}e^{-\sum_i^M \frac{(X_i-a_i)^2}{4\sigma^2}}\sqrt{\sum_{\{h_j\}}\prod_j^N e^{b_j h_j+\sum_i^M \frac{X_i w_{ij} h_j}{\sigma^2}}} \tag{1.111}$$

$$= \frac{1}{\sqrt{Z}}e^{-\sum_i^M \frac{(X_i-a_i)^2}{4\sigma^2}}\sqrt{\prod_j^N\sum_{h_j} e^{b_j h_j+\sum_i^M \frac{X_i w_{ij} h_j}{\sigma^2}}} \tag{1.112}$$

$$= \frac{1}{\sqrt{Z}}e^{-\sum_i^M \frac{(X_i-a_i)^2}{4\sigma^2}}\prod_j^N\sqrt{e^0+e^{b_j+\sum_i^M \frac{X_i w_{ij}}{\sigma^2}}} \tag{1.113}$$

$$= \frac{1}{\sqrt{Z}}e^{-\sum_i^M \frac{(X_i-a_i)^2}{4\sigma^2}}\prod_j^N\sqrt{1+e^{b_j+\sum_i^M \frac{X_i w_{ij}}{\sigma^2}}} \tag{1.114}$$

$$\tag{1.115}$$

Cost function.

This is where we deviate from what is common in machine learning. Rather than defining a cost function based on some dataset, our cost function is the energy of the quantum mechanical system. From the variational principle we know that minizing this energy should lead to the ground state wavefunction. As stated previously the local energy is given by

$$E_L = \frac{1}{\Psi}\hat{\mathbf{H}}\Psi, \tag{1.116}$$

and the gradient is

$$G_i = \frac{\partial\langle E_L\rangle}{\partial\alpha_i} = 2(\langle E_L\frac{1}{\Psi}\frac{\partial\Psi}{\partial\alpha_i}\rangle - \langle E_L\rangle\langle\frac{1}{\Psi}\frac{\partial\Psi}{\partial\alpha_i}\rangle), \tag{1.117}$$

where $\alpha_i = a_1,...,a_M,b_1,...,b_N,w_{11},...,w_{MN}$.

We use that $\frac{1}{\Psi}\frac{\partial\Psi}{\partial\alpha_i} = \frac{\partial\ln\Psi}{\partial\alpha_i}$, and find

$$\ln\Psi(\mathbf{X}) = -\ln Z - \sum_m^M\frac{(X_m-a_m)^2}{2\sigma^2} + \sum_n^N\ln(1+e^{b_n+\sum_i^M\frac{X_i w_{in}}{\sigma^2}}). \tag{1.118}$$

This gives

$$\frac{\partial}{\partial a_m}\ln\Psi = \frac{1}{\sigma^2}(X_m - a_m) \tag{1.119}$$

$$\frac{\partial}{\partial b_n}\ln\Psi = \frac{1}{e^{-b_n-\frac{1}{\sigma^2}\sum_i^M X_i w_{in}}+1} \tag{1.120}$$

$$\frac{\partial}{\partial w_{mn}}\ln\Psi = \frac{X_m}{\sigma^2(e^{-b_n-\frac{1}{\sigma^2}\sum_i^M X_i w_{in}}+1)}. \tag{1.121}$$

If $\Psi = \sqrt{F_{rbm}}$ we have

$$\ln\Psi(\mathbf{X}) = -\frac{1}{2}\ln Z - \sum_m^M \frac{(X_m - a_m)^2}{4\sigma^2} + \frac{1}{2}\sum_n^N \ln(1+e^{b_n+\sum_i^M \frac{X_i w_{in}}{\sigma^2}}), \tag{1.122}$$

which results in

$$\frac{\partial}{\partial a_m}\ln\Psi = \frac{1}{2\sigma^2}(X_m - a_m) \tag{1.123}$$

$$\frac{\partial}{\partial b_n}\ln\Psi = \frac{1}{2(e^{-b_n-\frac{1}{\sigma^2}\sum_i^M X_i w_{in}}+1)} \tag{1.124}$$

$$\frac{\partial}{\partial w_{mn}}\ln\Psi = \frac{X_m}{2\sigma^2(e^{-b_n-\frac{1}{\sigma^2}\sum_i^M X_i w_{in}}+1)}. \tag{1.125}$$

Let us assume again that our Hamiltonian is

$$\hat{\mathbf{H}} = \sum_p^P(-\frac{1}{2}\nabla_p^2 + \frac{1}{2}\omega^2 r_p^2) + \sum_{p<q}\frac{1}{r_{pq}}, \tag{1.126}$$

where the first summation term represents the standard harmonic oscillator part and the latter the repulsive interaction between two electrons. Natural units ($= c = e = m_e = 1$) are used, and P is the number of particles. This gives us the following expression for the local energy (D being the number of dimensions)

$$E_L = \frac{1}{\Psi}\mathbf{H}\Psi \tag{1.127}$$

$$= \frac{1}{\Psi}(\sum_p^P(-\frac{1}{2}\nabla_p^2 + \frac{1}{2}\omega^2 r_p^2) + \sum_{p<q}\frac{1}{r_{pq}})\Psi \tag{1.128}$$

$$= -\frac{1}{2}\frac{1}{\Psi}\sum_p^P \nabla_p^2\Psi + \frac{1}{2}\omega^2\sum_p^P r_p^2 + \sum_{p<q}\frac{1}{r_{pq}} \tag{1.129}$$

$$= -\frac{1}{2}\frac{1}{\Psi}\sum_p^P\sum_d^D \frac{\partial^2\Psi}{\partial x_{pd}^2} + \frac{1}{2}\omega^2\sum_p^P r_p^2 + \sum_{p<q}\frac{1}{r_{pq}} \tag{1.130}$$

$$= \frac{1}{2}\sum_p^P\sum_d^D(-(\frac{\partial}{\partial x_{pd}}\ln\Psi)^2 - \frac{\partial^2}{\partial x_{pd}^2}\ln\Psi + \omega^2 x_{pd}^2) + \sum_{p<q}\frac{1}{r_{pq}}. \tag{1.131}$$

$$\tag{1.132}$$

Letting each visible node in the Boltzmann machine represent one coordinate of one particle, we obtain

$$E_L = \frac{1}{2}\sum_m^M(-(\frac{\partial}{\partial v_m}\ln\Psi)^2 - \frac{\partial^2}{\partial v_m^2}\ln\Psi + \omega^2 v_m^2) + \sum_{p<q}\frac{1}{r_{pq}}, \tag{1.133}$$

where we have that

$$\frac{\partial}{\partial x_m}\ln\Psi = -\frac{1}{\sigma^2}(x_m - a_m) + \frac{1}{\sigma^2}\sum_n^N \frac{w_{mn}}{e^{-b_n - \frac{1}{\sigma^2}\sum_i^M x_i w_{in}} + 1} \tag{1.134}$$

$$\frac{\partial^2}{\partial x_m^2}\ln\Psi = -\frac{1}{\sigma^2} + \frac{1}{\sigma^4}\sum_n^N \omega_{mn}^2 \frac{e^{b_n + \frac{1}{\sigma^2}\sum_i^M x_i w_{in}}}{(e^{b_n + \frac{1}{\sigma^2}\sum_i^M x_i w_{in}} + 1)^2}. \tag{1.135}$$

We now have all the expressions neeeded to calculate the gradient of the expected local energy with respect to the RBM parameters $\frac{\partial\langle E_L\rangle}{\partial\alpha_i}$.

If we use $\Psi = \sqrt{F_{rbm}}$ we obtain

$$\frac{\partial}{\partial x_m}\ln\Psi = -\frac{1}{2\sigma^2}(x_m - a_m) + \frac{1}{2\sigma^2}\sum_n^N \frac{w_{mn}}{e^{-b_n - \frac{1}{\sigma^2}\sum_i^M x_i w_{in}} + 1} \tag{1.136}$$

$$\frac{\partial^2}{\partial x_m^2}\ln\Psi = -\frac{1}{2\sigma^2} + \frac{1}{2\sigma^4}\sum_n^N \omega_{mn}^2 \frac{e^{b_n + \frac{1}{\sigma^2}\sum_i^M x_i w_{in}}}{(e^{b_n + \frac{1}{\sigma^2}\sum_i^M x_i w_{in}} + 1)^2}. \tag{1.137}$$

The difference between this equation and the previous one is that we multiply by a factor $1/2$.

Python version for the two non-interacting particles

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python
2-electron VMC code for 2dim quantum dot with importance sampling Using gaussian rng
for new positions and Metropolis- Hastings Added restricted boltzmann machine method for
dealing with the wavefunction RBM code based heavily off of: https://github.com/CompPhysics/ComputationalPhysi
pages/doc/Programs/BoltzmannMachines/MLcpp/src/Pycode/ob from math import exp, sqrt
from random import random, seed, normalvariate import numpy as np import matplotlib.pyplot
as plt from mpl$_t$oolkits.mplot3dimportAxes3Dfrommatplotlibimportcmfrommatplotlib.tickerimportLinearLocator,FormatStrFormatterim

Trial wave function for the 2-electron quantum dot in two dims def WaveFunction(r,a,b,w):
sigma=1.0 sig2 = sigma**2 Psi1 = 0.0 Psi2 = 1.0 Q = Qfac(r,b,w)

for iq in range(NumberParticles): for ix in range(Dimension): Psi1 += (r[iq,ix]-a[iq,ix])**2

for ih in range(NumberHidden): Psi2 *= (1.0 + np.exp(Q[ih]))

Psi1 = np.exp(-Psi1/(2*sig2))

return Psi1*Psi2

Local energy for the 2-electron quantum dot in two dims, using analytical local energy def
LocalEnergy(r,a,b,w): sigma=1.0 sig2 = sigma**2 locenergy = 0.0

Q = Qfac(r,b,w)

for iq in range(NumberParticles): for ix in range(Dimension): sum1 = 0.0 sum2 = 0.0 for
ih in range(NumberHidden): sum1 += w[iq,ix,ih]/(1+np.exp(-Q[ih])) sum2 += w[iq,ix,ih]**2
* np.exp(Q[ih]) / (1.0 + np.exp(Q[ih]))**2

dlnpsi1 = -(r[iq,ix] - a[iq,ix]) /sig2 + sum1/sig2 dlnpsi2 = -1/sig2 + sum2/sig2**2 locenergy
+= 0.5*(-dlnpsi1*dlnpsi1 - dlnpsi2 + r[iq,ix]**2)

if(interaction==True): for iq1 in range(NumberParticles): for iq2 in range(iq1): distance =
0.0 for ix in range(Dimension): distance += (r[iq1,ix] - r[iq2,ix])**2

locenergy += 1/sqrt(distance)

return locenergy

Derivate of wave function ansatz as function of variational parameters def DerivativeWFansatz(r,a,b,w):

sigma=1.0 sig2 = sigma**2

Q = Qfac(r,b,w)

WfDer = np.empty((3,),dtype=object) WfDer = [np.copy(a),np.copy(b),np.copy(w)]

WfDer[0] = (r-a)/sig2 WfDer[1] = 1 / (1 + np.exp(-Q))

for ih in range(NumberHidden): WfDer[2][:,:,ih] = w[:,:,ih] / (sig2*(1+np.exp(-Q[ih])))

return WfDer

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

sigma=1.0 sig2 = sigma**2

qforce = np.zeros((NumberParticles,Dimension), np.double) sum1 = np.zeros((NumberParticles,Dimension), np.double)

Q = Qfac(r,b,w)

for ih in range(NumberHidden): sum1 += w[:,:,ih]/(1+np.exp(-Q[ih]))

qforce = 2*(-(r-a)/sig2 + sum1/sig2)

return qforce

def Qfac(r,b,w): Q = np.zeros((NumberHidden), np.double) temp = np.zeros((NumberHidden), np.double)

for ih in range(NumberHidden): temp[ih] = (r*w[:,:,ih]).sum()

Q = b + temp

return Q

Computing the derivative of the energy and the energy def EnergyMinimization(a,b,w):

NumberMCcycles= 10000 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() energy = 0.0 DeltaE = 0.0

EnergyDer = np.empty((3,),dtype=object) DeltaPsi = np.empty((3,),dtype=object) DerivativePsiE = np.empty((3,),dtype=object) EnergyDer = [np.copy(a),np.copy(b),np.copy(w)] DeltaPsi = [np.copy(a),np.copy(b),np.copy(w)] DerivativePsiE = [np.copy(a),np.copy(b),np.copy(w)] for i in range(3): EnergyDer[i].fill(0.0) for i in range(3): DeltaPsi[i].fill(0.0) for i in range(3): DerivativePsiE[i].fill(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,a,b,w) QuantumForceOld = QuantumForce(PositionOld,a,b,w)

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,a,b,w) QuantumForceNew = QuantumForce(PositionNew,a,b,w)

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 print("wf new: ", wfnew) print("force on 1 new:", QuantumForceNew[0,:]) print("pos of 1 new: ", PositionNew[0,:]) print("force on 2 new:", Quantum-

ForceNew[1,:]) print("pos of 2 new: ", PositionNew[1,:]) DeltaE = LocalEnergy(PositionOld,a,b,w) DerPsi = DerivativeWFansatz(PositionOld,a,b,w)

DeltaPsi[0] += DerPsi[0] DeltaPsi[1] += DerPsi[1] DeltaPsi[2] += DerPsi[2]

energy += DeltaE

DerivativePsiE[0] += DerPsi[0]*DeltaE DerivativePsiE[1] += DerPsi[1]*DeltaE DerivativePsiE[2] += DerPsi[2]*DeltaE

We calculate mean values energy /= NumberMCcycles DerivativePsiE[0] /= NumberMCcycles DerivativePsiE[1] /= NumberMCcycles DerivativePsiE[2] /= NumberMCcycles DeltaPsi[0] /= NumberMCcycles DeltaPsi[1] /= NumberMCcycles DeltaPsi[2] /= NumberMCcycles EnergyDer[0] = 2*(DerivativePsiE[0]-DeltaPsi[0]*energy) EnergyDer[1] = 2*(DerivativePsiE[1]-DeltaPsi[1]*energy) EnergyDer[2] = 2*(DerivativePsiE[2]-DeltaPsi[2]*energy) return energy, EnergyDer

Here starts the main program with variable declarations NumberParticles = 2 Dimension = 2 NumberHidden = 2

interaction=False

guess for parameters a=np.random.normal(loc=0.0, scale=0.001, size=(NumberParticles,Dimension)) b=np.random.normal(loc=0.0, scale=0.001, size=(NumberHidden)) w=np.random.normal(loc=0.0, scale=0.001, size=(NumberParticles,Dimension,NumberHidden)) Set up iteration using stochastic gradient method Energy = 0 EDerivative = np.empty((3,),dtype=object) EDerivative = [np.copy(a),np.copy(b),np.copy(w)] Learning rate eta, max iterations, need to change to adaptive learning rate eta = 0.001 MaxIterations = 50 iter = 0 np.seterr(invalid='raise') Energies = np.zeros(MaxIterations) EnergyDerivatives1 = np.zeros(MaxIterations) EnergyDerivatives2 = np.zeros(MaxIterations)

while iter < MaxIterations: Energy, EDerivative = EnergyMinimization(a,b,w) agradient = EDerivative[0] bgradient = EDerivative[1] wgradient = EDerivative[2] a -= eta*agradient b -= eta*bgradient w -= eta*wgradient Energies[iter] = Energy print("Energy:",Energy) EnergyDerivatives1[iter] = EDerivative[0] EnergyDerivatives2[iter] = EDerivative[1] EnergyDerivatives3[iter] = EDerivative[2]

iter += 1

nice printout with Pandas import pandas as pd from pandas import DataFrame pd.set$_{option}('max_columns', 6) data = {'Energy' : Energies,'ADerivative' : EnergyDerivatives1,'BDerivative' : EnergyDerivatives2,'WeightsDerivative' : EnergyDerivatives3}$

frame = pd.DataFrame(data) print(frame)