Week 6 February 6-10: Metropolis Algoritm and Markov Chains, Importance Sampling, Fokker-Planck and Langevin equations

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

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

February 6-10

Overview of week 6, February 6-10

Topics.

- Markov Chain Monte Carlo
- Metropolis-Hastings sampling and Importance Sampling

Teaching Material, videos and written material.

- Overview video on Metropolis algoritm
- Video of lecture
- Handwritten notes
- See also Lectures from FYS3150/4150 on the Metropolis Algorithm

Basics of the Metropolis Algorithm

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

The basic of the Metropolis Algorithm

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

More on the Metropolis

The dynamical equation for $\mathcal{P}_i^{(n)}$ 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:

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

Metropolis algorithm, setting it up

Since the probability of making some transition must be 1, $\sum_j T_{i\rightarrow j} = 1$, and Eq. (1) becomes

$$\mathcal{P}_i^{(n)} = \mathcal{P}_i^{(n-1)} + \sum_j \left[\mathcal{P}_j^{(n-1)}T_{j\rightarrow i}A_{j\rightarrow i} - \mathcal{P}_i^{(n-1)}T_{i\rightarrow j}A_{i\rightarrow j}\right] . \tag{2}$$

Metropolis continues

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

$$\sum_j \left[p_jT_{j\rightarrow i}A_{j\rightarrow i} - p_iT_{i\rightarrow j}A_{i\rightarrow j}\right] = 0, \tag{3}$$

Detailed Balance

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

$$\frac{A_{j\to i}}{A_{i\to j}} = \frac{p_i T_{i\to j}}{p_j T_{j\to i}} \,. \tag{4}$$

More on Detailed Balance

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

$$A_{j\to i} = \min\left(1, \frac{p_i T_{i\to j}}{p_j T_{j\to i}}\right) . \tag{5}$$

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

Having chosen the acceptance probabilities, we have guaranteed that if the $\mathcal{P}_i^{(n)}$ 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.

Dynamical Equation

The dynamical equation can be written as

$$\mathcal{P}_i^{(n)} = \sum_j M_{ij} \mathcal{P}_j^{(n-1)} \tag{6}$$

with the matrix M given by

$$M_{ij} = \delta_{ij}\left[1 - \sum_k T_{i\to k} A_{i\to k}\right] + T_{j\to i} A_{j\to i} \,. \tag{7}$$

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

Interpreting the Metropolis Algorithm

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.

Even a defective matrix has at least one left and right eigenvector for each eigenvalue. An example of a defective matrix is

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} ,$$

with two zero eigenvalues, only one right eigenvector

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

and only one left eigenvector (0 1).

Gershgorin bounds and Metropolis

The Gershgorin bounds for the eigenvalues can be derived by multiplying on the left with the eigenvector with the maximum and minimum eigenvalues,

$$\begin{aligned} \sum_i \psi_i^{\max} M_{ij} &= \lambda_{\max} \psi_j^{\max} \\ \sum_i \psi_i^{\min} M_{ij} &= \lambda_{\min} \psi_j^{\min} \end{aligned} \tag{8}$$

Normalizing the Eigenvectors

Next we choose the normalization of these eigenvectors so that the largest element (or one of the equally largest elements) has value 1. Let's call this element k, and we can therefore bound the magnitude of the other elements to be less than or equal to 1. This leads to the inequalities, using the property that $M_{ij} \geq 0$,

$$\begin{aligned} \sum_i M_{ik} &\leq \lambda_{\max} \\ M_{kk} - \sum_{i \neq k} M_{ik} &\geq \lambda_{\min} \end{aligned} \tag{9}$$

where the equality from the maximum will occur only if the eigenvector takes the value 1 for all values of i where $M_{ik} \neq 0$, and the equality for the minimum will occur only if the eigenvector takes the value -1 for all values of $i \neq k$ where $M_{ik} \neq 0$.

More Metropolis analysis

That the maximum eigenvalue is 1 follows immediately from the property that $\sum_i M_{ik} = 1$. Similarly the minimum eigenvalue can be -1, but only if $M_{kk} = 0$ and the magnitude of all the other elements $\psi_i^{\min}$ of the eigenvector that multiply nonzero elements M_{ik} are -1.

Let's first see what the properties of M must be to eliminate any -1 eigenvalues. To have a -1 eigenvalue, the left eigenvector must contain only ± 1 and 0 values. Taking in turn each ± 1 value as the maximum, so that it corresponds to the index k, the nonzero M_{ik} values must correspond to i index values of the eigenvector which have opposite sign elements. That is, the M matrix must break up into sets of states that always make transitions from set A to set B ... back to set A.

In particular, there can be no rejections of these moves in the cycle since the -1 eigenvalue requires $M_{kk} = 0$. To guarantee no eigenvalues with eigenvalue -1, we simply have to make sure that there are no cycles among states. Notice that this is generally trivial since such cycles cannot have any rejections at any stage. An example of such a cycle is sampling a noninteracting Ising spin. If the transition is taken to flip the spin, and the energy difference is zero, the Boltzmann factor will not change and the move will always be accepted. The system will simply flip from up to down to up to down ad infinitum. Including a rejection probability or using a heat bath algorithm immediately fixes the problem.

Final Considerations I

Next we need to make sure that there is only one left eigenvector with eigenvalue 1. To get an eigenvalue 1, the left eigenvector must be constructed from only ones and zeroes. It is straightforward to see that a vector made up of ones and zeroes can only be an eigenvector with eigenvalue 1 if the matrix element $M_{ij} = 0$ for all cases where $\psi_i \neq \psi_j$. That is we can choose an index i and take $\psi_i = 1$. We require all elements ψ_j where $M_{ij} \neq 0$ to also have the value 1. Continuing we then require all elements ψ_ℓ $M_{j\ell}$ to have value 1. Only if the matrix M can be put into block diagonal form can there be more than one choice for the left eigenvector with eigenvalue 1. We therefore require that the transition matrix not be in block diagonal form. This simply means that we must choose the transition probability so that we can get from any allowed state to any other in a series of transitions.

Final Considerations II

Finally, we note that for a defective matrix, with more eigenvalues than independent eigenvectors for eigenvalue 1, the left and right eigenvectors of eigenvalue 1 would be orthogonal. Here the left eigenvector is all 1 except for states that can never be reached, and the right eigenvector is $p_i > 0$ except for states that give zero probability. We already require that we can reach all states that contribute to p_i. Therefore the left and right eigenvectors with eigenvalue 1 do not correspond to a defective sector of the matrix and they are unique. The Metropolis algorithm therefore converges exponentially to the desired distribution.

Final Considerations III

The requirements for the transition $T_{i\to j}$ are

- A series of transitions must let us to get from any allowed state to any other by a finite series of transitions.

- The transitions cannot be grouped into sets of states, A, B, C ,... such that transitions from A go to B, B to C etc and finally back to A. With condition (a) satisfied, this condition will always be satisfied if either $T_{i\to i} \neq 0$ or there are some rejected moves.

Importance Sampling: Overview of what needs to be coded

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.

Importance sampling

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.

Importance sampling

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,

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

Importance sampling

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.

Importance sampling

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

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

which in turn means that our brute force Metropolis algorithm

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

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

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

Code example for the interacting case with importance sampling

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

```
# Common imports
import os

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
```

```python
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "Results/VMCQdotImportance"

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

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

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

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

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

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

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

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

```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_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import sys
from numba import jit,njit


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

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

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

# Setting up the quantum force for the two-electron quantum dot, recall that it is a vector
```

```python
def QuantumForce(r,alpha,beta):

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

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

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

    NumberMCcycles= 100000
    # Parameters in the Fokker-Planck simulation of the quantum force
    D = 0.5
    TimeStep = 0.05
    # positions
    PositionOld = np.zeros((NumberParticles,Dimension), np.double)
    PositionNew = np.zeros((NumberParticles,Dimension), np.double)
    # Quantum force
    QuantumForceOld = np.zeros((NumberParticles,Dimension), np.double)
    QuantumForceNew = np.zeros((NumberParticles,Dimension), np.double)

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

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

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

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

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