Week 8 February 22-26: Gradient Methods, and start Resampling Techniques

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

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

Feb 24, 2021

Overview of week 8, February 22-26

Topics.

- Start discussion of Resampling Techniques and statistics

Teaching Material, videos and written material.

- These lecture notes

- Video on the Conjugate Gradient methods

- Recommended background literature, Convex Optimization by Boyd and Vandenberghe. Their lecture slides are very useful (warning, these are some 300 pages).

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

$$\hat{J} = \left(\begin{array}{cc} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{array} \right),$$

we can rephrase Newton's method as

$$\left(\begin{array}{c} x_1^{n+1} \\ x_2^{n+1} \end{array} \right) = \left(\begin{array}{c} x_1^n \\ x_2^n \end{array} \right) + \left(\begin{array}{c} h_1^n \\ h_2^n \end{array} \right),$$

where we have defined

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

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 $(1 - t)x + ty$ 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

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

Code examples for steepest descent

Simple codes for steepest descent and conjugate gradient using a 2×2 matrix, in c++, Python code to come

```cpp
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#include "vectormatrixclass.h"
using namespace std;
//   Main function begins here
int main(int  argc, char * argv[]){
  int dim = 2;
  Vector x(dim),xsd(dim), b(dim),x0(dim);
  Matrix A(dim,dim);

  // Set our initial guess
  x0(0) = x0(1) = 0;
  // Set the matrix
  A(0,0) =  3;    A(1,0) =  2;   A(0,1) =  2;   A(1,1) =  6;
  b(0) = 2; b(1) = -8;
  cout << "The Matrix A that we are using: " << endl;
  A.Print();
  cout << endl;
  xsd = SteepestDescent(A,b,x0);
  cout << "The approximate solution using Steepest Descent is: " << endl;
  xsd.Print();
  cout << endl;
}
```

The routine for the steepest descent method

```cpp
Vector SteepestDescent(Matrix A, Vector b, Vector x0){
  int IterMax, i;
  int dim = x0.Dimension();
  const double tolerance = 1.0e-14;
  Vector x(dim),f(dim),z(dim);
  double c,alpha,d;
  IterMax = 30;
  x = x0;
  r = A*x-b;
  i = 0;
  while (i <= IterMax){
    z = A*r;
    c = dot(r,r);
    alpha = c/dot(r,z);
    x = x - alpha*r;
    r =   A*x-b;
    if(sqrt(dot(r,r)) < tolerance) break;
    i++;
  }
  return x;
}
```

Steepest descent example

```python
import numpy as np
import numpy.linalg as la

import scipy.optimize as sopt

import matplotlib.pyplot as pt
from mpl_toolkits.mplot3d import axes3d

def f(x):
    return 0.5*x[0]**2 + 2.5*x[1]**2

def df(x):
    return np.array([x[0], 5*x[1]])

fig = pt.figure()
ax = fig.gca(projection="3d")

xmesh, ymesh = np.mgrid[-2:2:50j,-2:2:50j]
fmesh = f(np.array([xmesh, ymesh]))
ax.plot_surface(xmesh, ymesh, fmesh)
```

And then as countor plot

```python
pt.axis("equal")
pt.contour(xmesh, ymesh, fmesh)
guesses = [np.array([2, 2./5])]
```

Find guesses

```python
x = guesses[-1]
s = -df(x)
```

Run it!

```python
def f1d(alpha):
    return f(x + alpha*s)

alpha_opt = sopt.golden(f1d)
next_guess = x + alpha_opt * s
guesses.append(next_guess)
print(next_guess)
```

What happened?

```python
pt.axis("equal")
pt.contour(xmesh, ymesh, fmesh, 50)
it_array = np.array(guesses)
pt.plot(it_array.T[0], it_array.T[1], "x-")
```

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

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

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

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

Simple implementation of the Conjugate gradient algorithm

```
Vector ConjugateGradient(Matrix A, Vector b, Vector x0){
int dim = x0.Dimension();
const double tolerance = 1.0e-14;
Vector x(dim),r(dim),v(dim),z(dim);
double c,t,d;

x = x0;
r = b - A*x;
v = r;
c = dot(r,r);
int i = 0; IterMax = dim;
while(i <= IterMax){
  z = A*v;
  t = c/dot(v,z);
  x = x + t*v;
  r = r - t*z;
  d = dot(r,r);
  if(sqrt(d) < tolerance)
    break;
  v = r + (d/c)*v;
  c = d;  i++;
}
  return x;
}
```

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

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

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

```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 gradient using the data in minibatch Bk
        #Compute new suggestion for
        j += 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.

```python
import numpy as np

def step_length(t,t0,t1):
    return t0/(t+t1)

n = 100 #100 datapoints
M = 5    #size of each minibatch
m = int(n/M) #number of minibatches
n_epochs = 500 #number of epochs
t0 = 1.0
t1 = 10

gamma_j = t0/t1
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 gradient using the data in minibatch Bk
        #Compute new suggestion for beta
        t = epoch*m+i
        gamma_j = step_length(t,t0,t1)
        j += 1

print("gamma_j after %d epochs: %g" % (n_epochs,gamma_j))
```

Program for stochastic gradient

```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("Own inversion")
print(theta_linreg)
sgdreg = SGDRegressor(n_iter = 50, penalty=None, eta0=0.1)
sgdreg.fit(x,y.ravel())
print("sgdreg from scikit")
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")
```

```python
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_epochs = 50
t0, t1 = 5, 50
m = 100
def learning_schedule(t):
    return t0/(t+t1)

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

for epoch in range(n_epochs):
    for i in range(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*gradients
print("theta from own sdg")
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

```cpp
//   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.

```cpp
//   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 methods

Resampling methods are an indispensable tool in modern statistics. They involve repeatedly drawing samples from a training set and refitting a model of interest on each sample in order to obtain additional information about the fitted model. For example, in order to estimate the variability of a linear regression fit, we can repeatedly draw different samples from the training data, fit a linear regression to each new sample, and then examine the extent to which the resulting fits differ. Such an approach may allow us to obtain information that would not be available from fitting the model only once using the original training sample.

Resampling approaches can be computationally expensive

Resampling approaches can be computationally expensive, because they involve fitting the same statistical method multiple times using different subsets of the training data. However, due to recent advances in computing power, the computational requirements of resampling methods generally are not prohibitive. In this chapter, we discuss two of the most commonly used resampling methods, cross-validation and the bootstrap. Both methods are important tools in the practical application of many statistical learning procedures. For example, cross-validation can be used to estimate the test error associated with a given statistical learning method in order to evaluate its performance, or to select the appropriate level of flexibility. The process of evaluating a model's performance is known as model assessment, whereas the process of selecting the proper level of flexibility for a model is known as model selection. The bootstrap is widely used.

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

The *probability distribution function (PDF)* is a function $p(x)$ on the domain which, in the discrete case, gives us the probability or relative frequency with which these values of X occur:

$$p(x) = \text{prob}(X = x)$$

In the continuous case, the PDF does not directly depict the actual probability. Instead we define the probability for the stochastic variable to assume any value on an infinitesimal interval around x to be $p(x)dx$. The continuous function $p(x)$ then gives us the *density* of the probability rather than the probability itself. The probability for a stochastic variable to assume any value on a non-infinitesimal interval $[a, b]$ is then just the integral:

$$\text{prob}(a \leq X \leq b) = \int_a^b p(x)dx$$

Qualitatively speaking, a stochastic variable represents the values of numbers chosen as if by chance from some specified PDF so that the selection of a large set of these numbers reproduces this PDF.

Statistics, moments

A particularly useful class of special expectation values are the *moments*. The n-th moment of the PDF p is defined as follows:

$$\langle x^n \rangle \equiv \int x^n p(x)\, dx$$

The zero-th moment $\langle 1 \rangle$ is just the normalization condition of p. The first moment, $\langle x \rangle$, is called the *mean* of p and often denoted by the letter μ:

$$\langle x \rangle = \mu \equiv \int x p(x)\, dx$$

Statistics, central moments

A special version of the moments is the set of *central moments*, the n-th central moment defined as:

$$\langle (x - \langle x \rangle)^n \rangle \equiv \int (x - \langle x \rangle)^n p(x)\, dx$$

The zero-th and first central moments are both trivial, equal 1 and 0, respectively. But the second central moment, known as the *variance* of p, is of particular interest. For the stochastic variable X, the variance is denoted as σ_X^2 or $\mathrm{var}(X)$:

$$\sigma_X^2 \;\; = \;\; \mathrm{var}(X) = \langle (x - \langle x \rangle)^2 \rangle = \int (x - \langle x \rangle)^2 p(x)\, dx \tag{1}$$

$$= \int \left(x^2 - 2x\langle x \rangle^2 + \langle x \rangle^2 \right) p(x)\, dx \tag{2}$$

$$= \langle x^2 \rangle - 2\langle x \rangle \langle x \rangle + \langle x \rangle^2 \tag{3}$$

$$= \langle x^2 \rangle - \langle x \rangle^2 \tag{4}$$

The square root of the variance, $\sigma = \sqrt{\langle (x - \langle x \rangle)^2 \rangle}$ is called the *standard deviation* of p. It is clearly just the RMS (root-mean-square) value of the deviation of the PDF from its mean value, interpreted qualitatively as the *spread* of p around its mean.

Statistics, covariance

Another important quantity is the so called covariance, a variant of the above defined variance. Consider again the set $\{X_i\}$ of n stochastic variables (not necessarily uncorrelated) with the multivariate PDF $P(x_1, \ldots, x_n)$. The *covariance* of two of the stochastic variables, X_i and X_j, is defined as follows:

$$\mathrm{cov}(X_i,\, X_j) \equiv \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle$$
$$= \int \cdots \int (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle)\, P(x_1, \ldots, x_n)\, dx_1 \ldots dx_n \tag{5}$$

with

$$\langle x_i \rangle = \int \cdots \int x_i\, P(x_1, \ldots, x_n)\, dx_1 \ldots dx_n$$

Statistics, more covariance

If we consider the above covariance as a matrix $C_{ij} = \mathrm{cov}(X_i,\, X_j)$, then the diagonal elements are just the familiar variances, $C_{ii} = \mathrm{cov}(X_i,\, X_i) = \mathrm{var}(X_i)$. It turns out that all the off-diagonal elements are zero if the stochastic variables are uncorrelated. This is easy to show, keeping in mind the linearity of the expectation value. Consider the stochastic variables X_i and X_j, $(i \neq j)$:

$$\mathrm{cov}(X_i,\, X_j) = \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \tag{6}$$

$$= \langle x_i x_j - x_i \langle x_j \rangle - \langle x_i \rangle x_j + \langle x_i \rangle \langle x_j \rangle \rangle \tag{7}$$

$$= \langle x_i x_j \rangle - \langle x_i \langle x_j \rangle \rangle - \langle \langle x_i \rangle x_j \rangle + \langle \langle x_i \rangle \langle x_j \rangle \rangle \tag{8}$$

$$= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle - \langle x_i \rangle \langle x_j \rangle + \langle x_i \rangle \langle x_j \rangle \tag{9}$$

$$= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle \tag{10}$$

Statistics, independent variables

If X_i and X_j are independent, we get $\langle x_i x_j \rangle = \langle x_i \rangle \langle x_j \rangle$, resulting in $\text{cov}(X_i, X_j) = 0 \ \ (i \neq j)$.

Also useful for us is the covariance of linear combinations of stochastic variables. Let $\{X_i\}$ and $\{Y_i\}$ be two sets of stochastic variables. Let also $\{a_i\}$ and $\{b_i\}$ be two sets of scalars. Consider the linear combination:

$$U = \sum_i a_i X_i \qquad V = \sum_j b_j Y_j$$

By the linearity of the expectation value

$$\text{cov}(U, V) = \sum_{i,j} a_i b_j \text{cov}(X_i, Y_j)$$

Statistics, more variance

Now, since the variance is just $\text{var}(X_i) = \text{cov}(X_i, X_i)$, we get the variance of the linear combination $U = \sum_i a_i X_i$:

$$\text{var}(U) = \sum_{i,j} a_i a_j \text{cov}(X_i, X_j) \tag{11}$$

And in the special case when the stochastic variables are uncorrelated, the off-diagonal elements of the covariance are as we know zero, resulting in:

$$\text{var}(U) = \sum_i a_i^2 \text{cov}(X_i, X_i) = \sum_i a_i^2 \text{var}(X_i)$$

$$\text{var}(\sum_i a_i X_i) = \sum_i a_i^2 \text{var}(X_i)$$

which will become very useful in our study of the error in the mean value of a set of measurements.

Statistics and stochastic processes

A *stochastic process* is a process that produces sequentially a chain of values:

$$\{x_1, x_2, \ldots x_k, \ldots\}.$$

We will call these values our *measurements* and the entire set as our measured *sample*. The action of measuring all the elements of a sample we will call a stochastic *experiment* since, operationally, they are often associated with results of empirical observation of some physical or mathematical phenomena; precisely an experiment. We assume that these values are distributed according to some PDF $p_X(x)$, where X is just the formal symbol for the stochastic variable whose PDF is $p_X(x)$. Instead of trying to determine the full distribution p we are often only interested in finding the few lowest moments, like the mean μ_X and the variance σ_X.

Statistics and sample variables

In practical situations a sample is always of finite size. Let that size be n. The expectation value of a sample, the *sample mean*, is then defined as follows:

$$\bar{x}_n \equiv \frac{1}{n} \sum_{k=1}^{n} x_k$$

The *sample variance* is:

$$\text{var}(x) \equiv \frac{1}{n} \sum_{k=1}^{n} (x_k - \bar{x}_n)^2$$

its square root being the *standard deviation of the sample*. The *sample covariance* is:

$$\text{cov}(x) \equiv \frac{1}{n} \sum_{kl} (x_k - \bar{x}_n)(x_l - \bar{x}_n)$$

Statistics, sample variance and covariance

Note that the sample variance is the sample covariance without the cross terms. In a similar manner as the covariance in Eq. (5) is a measure of the correlation between two stochastic variables, the above defined sample covariance is a measure of the sequential correlation between succeeding measurements of a sample.

These quantities, being known experimental values, differ significantly from and must not be confused with the similarly named quantities for stochastic variables, mean μ_X, variance $\text{var}(X)$ and covariance $\text{cov}(X, Y)$.

Statistics, law of large numbers

The law of large numbers states that as the size of our sample grows to infinity, the sample mean approaches the true mean μ_X of the chosen PDF:

$$\lim_{n \to \infty} \bar{x}_n = \mu_X$$

The sample mean $\bar{x}_n$ works therefore as an estimate of the true mean μ_X.

What we need to find out is how good an approximation $\bar{x}_n$ is to μ_X. In any stochastic measurement, an estimated mean is of no use to us without a measure of its error. A quantity that tells us how well we can reproduce it in another experiment. We are therefore interested in the PDF of the sample mean itself. Its standard deviation will be a measure of the spread of sample means, and we will simply call it the *error* of the sample mean, or just sample error, and denote it by err_X. In practice, we will only be able to produce an *estimate* of the sample error since the exact value would require the knowledge of the true PDFs behind, which we usually do not have.

Statistics, more on sample error

Let us first take a look at what happens to the sample error as the size of the sample grows. In a sample, each of the measurements x_i can be associated with its own stochastic variable X_i. The stochastic variable $\overline{X}_n$ for the sample mean $\bar{x}_n$ is then just a linear combination, already familiar to us:

$$\overline{X}_n = \frac{1}{n} \sum_{i=1}^{n} X_i$$

All the coefficients are just equal $1/n$. The PDF of $\overline{X}_n$, denoted by $p_{\overline{X}_n}(x)$ is the desired PDF of the sample means.

Statistics

The probability density of obtaining a sample mean $\bar{x}_n$ is the product of probabilities of obtaining arbitrary values $x_1, x_2, \ldots, x_n$ with the constraint that the mean of the set $\{x_i\}$ is $\bar{x}_n$:

$$p_{\overline{X}_n}(x) = \int p_X(x_1) \cdots \int p_X(x_n) \, \delta\left(x - \frac{x_1 + x_2 + \cdots + x_n}{n}\right) dx_n \cdots dx_1$$

And in particular we are interested in its variance $\mathrm{var}(\overline{X}_n)$.

Statistics, central limit theorem

It is generally not possible to express $p_{\overline{X}_n}(x)$ in a closed form given an arbitrary PDF p_X and a number n. But for the limit $n \to \infty$ it is possible to make an approximation. The very important result is called *the central limit theorem*. It tells us that as n goes to infinity, $p_{\overline{X}_n}(x)$ approaches a Gaussian distribution whose mean and variance equal the true mean and variance, μ_X and σ_X^2, respectively:

$$\lim_{n \to \infty} p_{\overline{X}_n}(x) = \left(\frac{n}{2\pi \mathrm{var}(X)}\right)^{1/2} e^{-\frac{n(x - \bar{x}_n)^2}{2\mathrm{var}(X)}} \tag{12}$$

Statistics, more technicalities

The desired variance $\mathrm{var}(\overline{X}_n)$, i.e. the sample error squared err_X^2, is given by:

$$\mathrm{err}_X^2 = \mathrm{var}(\overline{X}_n) = \frac{1}{n^2} \sum_{ij} \mathrm{cov}(X_i, X_j) \tag{13}$$

We see now that in order to calculate the exact error of the sample with the above expression, we would need the true means μ_{X_i} of the stochastic variables X_i. To calculate these requires that we know the true multivariate PDF of all the X_i. But this PDF is unknown to us, we have only got the measurements of

one sample. The best we can do is to let the sample itself be an estimate of the PDF of each of the X_i, estimating all properties of X_i through the measurements of the sample.

Statistics

Our estimate of μ_{X_i} is then the sample mean $\bar{x}$ itself, in accordance with the the central limit theorem:

$$\mu_{X_i} = \langle x_i \rangle \approx \frac{1}{n} \sum_{k=1}^{n} x_k = \bar{x}$$

Using $\bar{x}$ in place of μ_{X_i} we can give an *estimate* of the covariance in Eq. (13)

$$\text{cov}(X_i, X_j) = \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \approx \langle (x_i - \bar{x})(x_j - \bar{x}) \rangle,$$

resulting in

$$\frac{1}{n} \sum_{l}^{n} \left(\frac{1}{n} \sum_{k}^{n} (x_k - \bar{x}_n)(x_l - \bar{x}_n) \right) = \frac{1}{n} \frac{1}{n} \sum_{kl} (x_k - \bar{x}_n)(x_l - \bar{x}_n) = \frac{1}{n} \text{cov}(x)$$

Statistics and sample variance

By the same procedure we can use the sample variance as an estimate of the variance of any of the stochastic variables X_i

$$\text{var}(X_i) = \langle x_i - \langle x_i \rangle \rangle \approx \langle x_i - \bar{x}_n \rangle,$$

which is approximated as

$$\text{var}(X_i) \approx \frac{1}{n} \sum_{k=1}^{n} (x_k - \bar{x}_n) = \text{var}(x) \tag{14}$$

Now we can calculate an estimate of the error err_X of the sample mean $\bar{x}_n$:

$$\text{err}_X^2 = \frac{1}{n^2} \sum_{ij} \text{cov}(X_i, X_j)$$

$$\approx \qquad\qquad \frac{1}{n^2} \sum_{ij} \frac{1}{n} \text{cov}(x) = \frac{1}{n^2} n^2 \frac{1}{n} \text{cov}(x)$$

$$= \frac{1}{n} \text{cov}(x) \tag{15}$$

which is nothing but the sample covariance divided by the number of measurements in the sample.

Statistics, uncorrelated results

In the special case that the measurements of the sample are uncorrelated (equivalently the stochastic variables X_i are uncorrelated) we have that the off-diagonal elements of the covariance are zero. This gives the following estimate of the sample error:

$$\mathrm{err}_X^2 = \frac{1}{n^2} \sum_{ij} \mathrm{cov}(X_i, X_j) = \frac{1}{n^2} \sum_i \mathrm{var}(X_i),$$

resulting in

$$\mathrm{err}_X^2 \approx \frac{1}{n^2} \sum_i \mathrm{var}(x) = \frac{1}{n}\mathrm{var}(x) \tag{16}$$

where in the second step we have used Eq. (14). The error of the sample is then just its standard deviation divided by the square root of the number of measurements the sample contains. This is a very useful formula which is easy to compute. It acts as a first approximation to the error, but in numerical experiments, we cannot overlook the always present correlations.

Statistics, computations

For computational purposes one usually splits up the estimate of err_X^2, given by Eq. (15), into two parts

$$\mathrm{err}_X^2 = \frac{1}{n}\mathrm{var}(x) + \frac{1}{n}(\mathrm{cov}(x) - \mathrm{var}(x)),$$

which equals

$$\frac{1}{n^2} \sum_{k=1}^{n} (x_k - \bar{x}_n)^2 + \frac{2}{n^2} \sum_{k<l} (x_k - \bar{x}_n)(x_l - \bar{x}_n) \tag{17}$$

The first term is the same as the error in the uncorrelated case, Eq. (16). This means that the second term accounts for the error correction due to correlation between the measurements. For uncorrelated measurements this second term is zero.

Statistics, more on computations of errors

Computationally the uncorrelated first term is much easier to treat efficiently than the second.

$$\mathrm{var}(x) = \frac{1}{n} \sum_{k=1}^{n} (x_k - \bar{x}_n)^2 = \left(\frac{1}{n}\sum_{k=1}^{n} x_k^2\right) - \bar{x}_n^2$$

We just accumulate separately the values x^2 and x for every measurement x we receive. The correlation term, though, has to be calculated at the end of the experiment since we need all the measurements to calculate the cross terms. Therefore, all measurements have to be stored throughout the experiment.

Statistics, wrapping up 1

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 (see eq. (17)) can now be written in terms of the autocorrelation function:

$$\mathrm{err}_X^2 = \frac{1}{n}\mathrm{var}(x) + \frac{2}{n} \cdot \mathrm{var}(x) \sum_{d=1}^{n-1} \frac{f_d}{\mathrm{var}(x)}$$

$$= \left(1 + 2 \sum_{d=1}^{n-1} \kappa_d\right) \frac{1}{n}\mathrm{var}(x)$$

$$= \frac{\tau}{n} \cdot \mathrm{var}(x) \tag{18}$$

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*:

$$\tau = 1 + 2 \sum_{d=1}^{n-1} \kappa_d \tag{19}$$

Statistics, effective number of correlations

For a correlation free experiment, τ equals 1. From the point of view of eq. (18) 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 $\text{err}_X^2 \approx \text{var}(x)/n$ 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 ith 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 ith 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

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

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

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

which suggests another measure of correlation

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

called the integrated correlation time.