Data Analysis and Machine Learning: Elements of Probability Theory and Statistical Data Analysis

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Sep 20, 2020

To do list

- add math about MVN and define MLE and other quantities
- rewrite about covariance matrix
- add KL theorem

Domains and probabilities

Consider the following simple example, namely the tossing of two dice, resulting in the following possible values

$$\{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}.$$

These values are called the *domain*. To this domain we have the corresponding *probabilities*

$$\{1/36, 2/36/, 3/36, 4/36, 5/36, 6/36, 5/36, 4/36, 3/36, 2/36, 1/36\}.$$

Tossing the dice

The numbers in the domain are the outcomes of the physical process of tossing say two dice. We cannot tell beforehand whether the outcome is 3 or 5 or any other number in this domain. This defines the randomness of the outcome, or unexpectedness or any other synonimous word which encompasses the uncertitude of the final outcome.

The only thing we can tell beforehand is that say the outcome 2 has a certain probability. If our favorite hobby is to spend an hour every evening throwing dice and registering the sequence of outcomes, we will note that the numbers in the above domain

$$\{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\},$$

appear in a random order. After 11 throws the results may look like

$$\{10, 8, 6, 3, 6, 9, 11, 8, 12, 4, 5\}.$$

Stochastic variables

Random variables are characterized by a domain which contains all possible values that the random value may take. This domain has a corresponding probability distribution function(PDF).

Stochastic variables and the main concepts, the discrete case

There are two main concepts associated with a stochastic variable. The *domain* is the set $\mathbb{D} = \{x\}$ of all accessible values the variable can assume, so that $X \in \mathbb{D}$. An example of a discrete domain is the set of six different numbers that we may get by throwing of a dice, $x \in \{1, 2, 3, 4, 5, 6\}$.

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

$$p(x) = \text{Prob}(X = x).$$

Stochastic variables and the main concepts, the continuous case

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.

The cumulative probability

Of interest to us is the *cumulative probability distribution function* (**CDF**), $P(x)$, which is just the probability for a stochastic variable X to assume any value less than x

$$P(x) = \text{Prob}(X \leq x) = \int_{-\infty}^{x} p(x')dx'.$$

The relation between a CDF and its corresponding PDF is then

$$p(x) = \frac{d}{dx}P(x).$$

Properties of PDFs

There are two properties that all PDFs must satisfy. The first one is positivity (assuming that the PDF is normalized)

$$0 \leq p(x) \leq 1.$$

Naturally, it would be nonsensical for any of the values of the domain to occur with a probability greater than 1 or less than 0. Also, the PDF must be normalized. That is, all the probabilities must add up to unity. The probability of "anything" to happen is always unity. For both discrete and continuous PDFs, this condition is

$$\sum_{x_i \in \mathbb{D}} p(x_i) = 1,$$

$$\int_{x \in \mathbb{D}} p(x)\, dx = 1.$$

Important distributions, the uniform distribution

The first one is the most basic PDF; namely the uniform distribution

$$p(x) = \frac{1}{b-a}\theta(x-a)\theta(b-x). \tag{1}$$

For $a = 0$ and $b = 1$ we have

$$p(x)dx = dx \quad \in [0, 1].$$

The latter distribution is used to generate random numbers. For other PDFs, one needs normally a mapping from this distribution to say for example the exponential distribution.

Gaussian distribution

The second one is the Gaussian Distribution

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp{(-\frac{(x-\mu)^2}{2\sigma^2})},$$

with mean value μ and standard deviation σ. If $\mu = 0$ and $\sigma = 1$, it is normally called the **standard normal distribution**

$$p(x) = \frac{1}{\sqrt{2\pi}} \exp{(-\frac{x^2}{2})},$$

The following simple Python code plots the above distribution for different values of μ and σ.

```
import numpy as np
from math import acos, exp, sqrt
from  matplotlib import pyplot as plt
from matplotlib import rc, rcParams
import matplotlib.units as units
import matplotlib.ticker as ticker
rc('text',usetex=True)
rc('font',**{'family':'serif','serif':['Gaussian distribution']})
font = {'family' : 'serif',
        'color'  : 'darkred',
        'weight' : 'normal',
        'size'   : 16,
        }
pi = acos(-1.0)
mu0 = 0.0
sigma0 = 1.0
mu1= 1.0
sigma1 = 2.0
mu2 = 2.0
sigma2 = 4.0

x = np.linspace(-20.0, 20.0)
v0 = np.exp(-(x*x-2*x*mu0+mu0*mu0)/(2*sigma0*sigma0))/sqrt(2*pi*sigma0*sigma0)
v1 = np.exp(-(x*x-2*x*mu1+mu1*mu1)/(2*sigma1*sigma1))/sqrt(2*pi*sigma1*sigma1)
v2 = np.exp(-(x*x-2*x*mu2+mu2*mu2)/(2*sigma2*sigma2))/sqrt(2*pi*sigma2*sigma2)
plt.plot(x, v0, 'b-', x, v1, 'r-', x, v2, 'g-')
plt.title(r'{\bf Gaussian distributions}', fontsize=20)
plt.text(-19, 0.3, r'Parameters: $\mu = 0$, $\sigma = 1$', fontdict=font)
plt.text(-19, 0.18, r'Parameters: $\mu = 1$, $\sigma = 2$', fontdict=font)
plt.text(-19, 0.08, r'Parameters: $\mu = 2$, $\sigma = 4$', fontdict=font)
plt.xlabel(r'$x$',fontsize=20)
plt.ylabel(r'$p(x)$ [MeV]',fontsize=20)

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.savefig('gaussian.pdf', format='pdf')
plt.show()
```

Exponential distribution

Another important distribution in science is the exponential distribution

$$p(x) = \alpha \exp{-(\alpha x)}.$$

Expectation values

Let $h(x)$ be an arbitrary continuous function on the domain of the stochastic variable X whose PDF is $p(x)$. We define the *expectation value* of h with respect to p as follows

$$\langle h \rangle_X \equiv \int h(x)p(x)\,dx \tag{2}$$

Whenever the PDF is known implicitly, like in this case, we will drop the index X for clarity. A particularly useful class of special expectation values are the *moments*. The n-th moment of the PDF p is defined as follows

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

Stochastic variables and the main concepts, mean values

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

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

for a continuous distribution and

$$\langle x \rangle = \mu \equiv \sum_{i=1}^{N} x_i p(x_i),$$

for a discrete distribution. Qualitatively it represents the centroid or the average value of the PDF and is therefore simply called the expectation value of $p(x)$.

Stochastic variables and the main concepts, central moments, the variance

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 $\mathrm{Var}(X)$

$$\begin{aligned}
\sigma_X^2 = \mathrm{Var}(X) &= \langle (x - \langle x \rangle)^2 \rangle = \int (x - \langle x \rangle)^2 p(x)dx \\
&= \int \left(x^2 - 2x\langle x \rangle^2 + \langle x \rangle^2 \right) p(x)dx \\
&= \langle x^2 \rangle - 2\langle x \rangle \langle x \rangle + \langle x \rangle^2 \\
&= \langle x^2 \rangle - \langle x \rangle^2
\end{aligned}$$

The square root of the variance, $\sigma = \sqrt{\langle (x - \langle x \rangle)^2 \rangle}$ is called the **standard deviation** of p. It is 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.

Probability Distribution Functions

The following table collects properties of probability distribution functions. In our notation we reserve the label $p(x)$ for the probability of a certain event, while $P(x)$ is the cumulative probability.

	Discrete PDF	Continuous PDF
Domain	$\{x_1, x_2, x_3, \ldots, x_N\}$	$[a, b]$
Probability	$p(x_i)$	$p(x)dx$
Cumulative	$P_i = \sum_{l=1}^{i} p(x_l)$	$P(x) = \int_a^x p(t)dt$
Positivity	$0 \leq p(x_i) \leq 1$	$p(x) \geq 0$
Positivity	$0 \leq P_i \leq 1$	$0 \leq P(x) \leq 1$
Monotonic	$P_i \geq P_j$ if $x_i \geq x_j$	$P(x_i) \geq P(x_j)$ if $x_i \geq x_j$
Normalization	$P_N = 1$	$P(b) = 1$

Probability Distribution Functions

With a PDF we can compute expectation values of selected quantities such as

$$\langle x^k \rangle = \sum_{i=1}^{N} x_i^k p(x_i),$$

if we have a discrete PDF or

$$\langle x^k \rangle = \int_a^b x^k p(x) dx,$$

in the case of a continuous PDF. We have already defined the mean value μ and the variance σ^2.

The three famous Probability Distribution Functions

There are at least three PDFs which one may encounter. These are the

Uniform distribution

$$p(x) = \frac{1}{b-a}\Theta(x-a)\Theta(b-x),$$

yielding probabilities different from zero in the interval $[a, b]$.

The exponential distribution

$$p(x) = \alpha \exp(-\alpha x),$$

yielding probabilities different from zero in the interval $[0,\infty)$ and with mean value

$$\mu = \int_0^\infty xp(x)dx = \int_0^\infty x\alpha\exp\left(-\alpha x\right)dx = \frac{1}{\alpha},$$

with variance

$$\sigma^2 = \int_0^\infty x^2p(x)dx - \mu^2 = \frac{1}{\alpha^2}.$$

Probability Distribution Functions, the normal distribution

Finally, we have the so-called univariate normal distribution, or just the **normal distribution**

$$p(x) = \frac{1}{b\sqrt{2\pi}}\exp\left(-\frac{(x-a)^2}{2b^2}\right)$$

with probabilities different from zero in the interval $(-\infty,\infty)$. The integral $\int_{-\infty}^{\infty}\exp\left(-(x^2\right)dx$ appears in many calculations, its value is $\sqrt{\pi}$, a result we will need when we compute the mean value and the variance. The mean value is

$$\mu = \int_0^\infty xp(x)dx = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty}x\exp\left(-\frac{(x-a)^2}{2b^2}\right)dx,$$

which becomes with a suitable change of variables

$$\mu = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty}b\sqrt{2}(a+b\sqrt{2}y)\exp -y^2dy = a.$$

Probability Distribution Functions, the normal distribution

Similarly, the variance becomes

$$\sigma^2 = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty}(x-\mu)^2\exp\left(-\frac{(x-a)^2}{2b^2}\right)dx,$$

and inserting the mean value and performing a variable change we obtain

$$\sigma^2 = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty}b\sqrt{2}(b\sqrt{2}y)^2\exp\left(-y^2\right)dy = \frac{2b^2}{\sqrt{\pi}}\int_{-\infty}^{\infty}y^2\exp\left(-y^2\right)dy,$$

and performing a final integration by parts we obtain the well-known result $\sigma^2 = b^2$. It is useful to introduce the standard normal distribution as well, defined by $\mu = a = 0$, viz. a distribution centered around zero and with a variance $\sigma^2 = 1$, leading to

$$p(x) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{x^2}{2}\right). \tag{3}$$

Probability Distribution Functions, the cumulative distribution

The exponential and uniform distributions have simple cumulative functions, whereas the normal distribution does not, being proportional to the so-called error function $erf(x)$, given by

$$P(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(-\frac{t^2}{2}\right) dt,$$

which is difficult to evaluate in a quick way.

Probability Distribution Functions, other important distribution

Some other PDFs which one encounters often in the natural sciences are the binomial distribution

$$p(x) = \left(\begin{array}{c} n \\ x \end{array} \right) y^x (1-y)^{n-x} \qquad x = 0, 1, \ldots, n,$$

where y is the probability for a specific event, such as the tossing of a coin or moving left or right in case of a random walker. Note that x is a discrete stochastic variable.

The sequence of binomial trials is characterized by the following definitions

- Every experiment is thought to consist of N independent trials.
- In every independent trial one registers if a specific situation happens or not, such as the jump to the left or right of a random walker.
- The probability for every outcome in a single trial has the same value, for example the outcome of tossing (either heads or tails) a coin is always 1/2.

Probability Distribution Functions, the binomial distribution

In order to compute the mean and variance we need to recall Newton's binomial formula

$$(a+b)^m = \sum_{n=0}^{m} \left(\begin{array}{c} m \\ n \end{array} \right) a^n b^{m-n},$$

which can be used to show that

$$\sum_{x=0}^{n} \left(\begin{array}{c} n \\ x \end{array} \right) y^x (1-y)^{n-x} = (y+1-y)^n = 1,$$

the PDF is normalized to one. The mean value is

$$\mu = \sum_{x=0}^{n} x \left(\begin{array}{c} n \\ x \end{array} \right) y^x (1-y)^{n-x} = \sum_{x=0}^{n} x \frac{n!}{x!(n-x)!} y^x (1-y)^{n-x},$$

resulting in

$$\mu = \sum_{x=0}^{n} x \frac{(n-1)!}{(x-1)!(n-1-(x-1))!} y^{x-1} (1-y)^{n-1-(x-1)},$$

which we rewrite as

$$\mu = ny \sum_{\nu=0}^{n} \left(\begin{array}{c} n-1 \\ \nu \end{array} \right) y^{\nu} (1-y)^{n-1-\nu} = ny(y+1-y)^{n-1} = ny.$$

The variance is slightly trickier to get. It reads $\sigma^2 = ny(1-y)$.

Probability Distribution Functions, Poisson's distribution

Another important distribution with discrete stochastic variables x is the Poisson model, which resembles the exponential distribution and reads

$$p(x) = \frac{\lambda^x}{x!} e^{-\lambda} \qquad x = 0, 1, \ldots, ; \lambda > 0.$$

In this case both the mean value and the variance are easier to calculate,

$$\mu = \sum_{x=0}^{\infty} x \frac{\lambda^x}{x!} e^{-\lambda} = \lambda e^{-\lambda} \sum_{x=1}^{\infty} \frac{\lambda^{x-1}}{(x-1)!} = \lambda,$$

and the variance is $\sigma^2 = \lambda$.

Probability Distribution Functions, Poisson's distribution

An example of applications of the Poisson distribution could be the counting of the number of α-particles emitted from a radioactive source in a given time interval. In the limit of $n \to \infty$ and for small probabilities y, the binomial distribution approaches the Poisson distribution. Setting $\lambda = ny$, with y the probability for an event in the binomial distribution we can show that

$$\lim_{n \to \infty} \left(\begin{array}{c} n \\ x \end{array} \right) y^x (1-y)^{n-x} e^{-\lambda} = \sum_{x=1}^{\infty} \frac{\lambda^x}{x!} e^{-\lambda}.$$

Meet the covariance!

An important quantity in a statistical analysis is the so-called covariance.

Consider the set $\{X_i\}$ of n stochastic variables (not necessarily uncorrelated) with the multivariate PDF $P(x_1, \dots, x_n)$. The *covariance* of two of the stochastic variables, X_i and X_j, is defined as follows

$$\mathrm{Cov}(X_i,\, X_j) = \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \tag{4}$$

$$= \int \cdots \int (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) P(x_1, \dots, x_n)\, dx_1 \dots dx_n, \tag{5}$$

with

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

Meet the covariance in matrix disguise

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.

Covariance

```
# Importing various packages
from math import exp, sqrt
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt

def covariance(x, y, n):
    sum = 0.0
    mean_x = np.mean(x)
    mean_y = np.mean(y)
    for i in range(0, n):
        sum += (x[(i)]-mean_x)*(y[i]-mean_y)
    return  sum/n

n = 10

x=np.random.normal(size=n)
y = 4+3*x+np.random.normal(size=n)
covxy = covariance(x,y,n)
print(covxy)
z = np.vstack((x, y))
c = np.cov(z.T)

print(c)
```

Meet the covariance, uncorrelated events

Consider the stochastic variables X_i and X_j, $(i \neq j)$. We have

$$\begin{aligned} Cov(X_i,\, X_j) &= \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \\ &= \langle x_i x_j - x_i \langle x_j \rangle - \langle x_i \rangle x_j + \langle x_i \rangle \langle x_j \rangle \rangle \\ &= \langle x_i x_j \rangle - \langle x_i \langle x_j \rangle \rangle - \langle \langle x_i \rangle x_j \rangle + \langle \langle x_i \rangle \langle x_j \rangle \rangle \\ &= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle - \langle x_i \rangle \langle x_j \rangle + \langle x_i \rangle \langle x_j \rangle \\ &= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle \end{aligned}$$

If X_i and X_j are independent (assuming $i \neq j$), we have that

$$\langle x_i x_j \rangle = \langle x_i \rangle \langle x_j \rangle,$$

leading to

$$Cov(X_i, X_j) = 0 \ (i \neq j).$$

Numerical experiments and the covariance

Now that we have constructed an idealized mathematical framework, let us try to apply it to empirical observations. Examples of relevant physical phenomena may be spontaneous decays of nuclei, or a purely mathematical set of numbers produced by some deterministic mechanism. It is the latter we will deal with, using so-called pseudo-random number generators. In general our observations will contain only a limited set of observables. We remind the reader that a *stochastic process* is a process that produces sequentially a chain of values

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

Numerical experiments and the covariance

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.

Numerical experiments and the covariance, actual situations

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

as follows

$$\langle x_\alpha \rangle \equiv \frac{1}{n}\sum_{k=1}^{n} x_{\alpha,k}.$$

The *sample variance* is:

$$\mathrm{Var}(x) \equiv \frac{1}{n}\sum_{k=1}^{n}(x_{\alpha,k} - \langle x_\alpha \rangle)^2,$$

with its square root being the *standard deviation of the sample*.

Numerical experiments and the covariance, our observables

You can think of the above observables as a set of quantities which define a given experiment. This experiment is then repeated several times, say m times. The total average is then

$$\langle X_m \rangle = \frac{1}{m}\sum_{\alpha=1}^{m} x_\alpha = \frac{1}{mn}\sum_{\alpha,k} x_{\alpha,k}, \tag{6}$$

where the last sums end at m and n. The total variance is

$$\sigma_m^2 = \frac{1}{mn^2}\sum_{\alpha=1}^{m}(\langle x_\alpha \rangle - \langle X_m \rangle)^2,$$

which we rewrite as

$$\sigma_m^2 = \frac{1}{m}\sum_{\alpha=1}^{m}\sum_{kl=1}^{n}(x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,l} - \langle X_m \rangle). \tag{7}$$

Numerical experiments and the covariance, the sample variance

We define also the sample variance σ^2 of all mn individual experiments as

$$\sigma^2 = \frac{1}{mn}\sum_{\alpha=1}^{m}\sum_{k=1}^{n}(x_{\alpha,k} - \langle X_m \rangle)^2. \tag{8}$$

These quantities, being known experimental values or the results from our calculations, may differ, in some cases significantly, from the similarly named exact values for the mean value μ_X, the variance $\mathrm{Var}(X)$ and the covariance $\mathrm{Cov}(X,Y)$.

Numerical experiments and the covariance, central limit theorem

The central limit theorem states that the PDF $\tilde{p}(z)$ of the average of m random values corresponding to a PDF $p(x)$ is a normal distribution whose mean is the mean value of the PDF $p(x)$ and whose variance is the variance of the PDF $p(x)$ divided by m, the number of values used to compute z.

The central limit theorem leads then to the well-known expression for the standard deviation, given by

$$\sigma_m = \frac{\sigma}{\sqrt{m}}.$$

In many cases the above estimate for the standard deviation, in particular if correlations are strong, may be too simplistic. We need therefore a more precise defintion of the error and the variance in our results.

Definition of Correlation Functions and Standard Deviation

Our estimate of the true average μ_X is the sample mean $\langle X_m \rangle$

$$\mu_X \approx X_m = \frac{1}{mn}\sum_{\alpha=1}^{m}\sum_{k=1}^{n} x_{\alpha,k}.$$

We can then use Eq. (7)

$$\sigma_m^2 = \frac{1}{mn^2}\sum_{\alpha=1}^{m}\sum_{kl=1}^{n}(x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,l} - \langle X_m \rangle),$$

and rewrite it as

$$\sigma_m^2 = \frac{\sigma^2}{n} + \frac{2}{mn^2}\sum_{\alpha=1}^{m}\sum_{k<l}^{n}(x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,l} - \langle X_m \rangle),$$

where the first term is the sample variance of all mn experiments divided by n and the last term is nothing but the covariance which arises when $k \neq l$.

Definition of Correlation Functions and Standard Deviation

Our estimate of the true average μ_X is the sample mean $\langle X_m \rangle$

If the observables are uncorrelated, then the covariance is zero and we obtain a total variance which agrees with the central limit theorem. Correlations may often be present in our data set, resulting in a non-zero covariance. The first term is normally called the uncorrelated contribution. Computationally the uncorrelated first term is much easier to treat efficiently than the second. 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.

Definition of Correlation Functions and Standard Deviation

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

$$f_d = \frac{1}{nm}\sum_{\alpha=1}^{m}\sum_{k=1}^{n-d}(x_{\alpha,k} - \langle X_m\rangle)(x_{\alpha,k+d} - \langle X_m\rangle),$$

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

$$\frac{2}{mn^2}\sum_{\alpha=1}^{m}\sum_{k<l}^{n}(x_{\alpha,k} - \langle X_m\rangle)(x_{\alpha,l} - \langle X_m\rangle) = \frac{2}{n}\sum_{d=1}^{n-1} f_d$$

Definition of Correlation Functions and Standard Deviation

The value of f_d reflects the correlation between measurements separated by the distance d in the samples. Notice that for $d = 0$, f is just the sample variance, σ^2. If we divide f_d by σ^2, we arrive at the so called **autocorrelation function**

$$\kappa_d = \frac{f_d}{\sigma^2} \tag{9}$$

which gives us a useful measure of the correlation pair correlation starting always at 1 for $d = 0$.

Definition of Correlation Functions and Standard Deviation, sample variance

The sample variance of the mn experiments can now be written in terms of the autocorrelation function

$$\sigma_m^2 = \frac{\sigma^2}{n} + \frac{2}{n}\cdot\sigma^2\sum_{d=1}^{n-1}\frac{f_d}{\sigma^2} = \left(1 + 2\sum_{d=1}^{n-1}\kappa_d\right)\frac{1}{n}\sigma^2 = \frac{\tau}{n}\cdot\sigma^2 \tag{10}$$

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

$$\tau = 1 + 2\sum_{d=1}^{n-1}\kappa_d \tag{11}$$

For a correlation free experiment, τ equals 1.

Definition of Correlation Functions and Standard Deviation

From the point of view of Eq. (10) 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

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

To neglect the autocorrelation time τ will always cause our simple uncorrelated estimate of $\sigma_m^2 \approx \sigma^2/n$ 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. The solution to this problem is given by more practically oriented methods like the blocking technique.

Code to compute the Covariance matrix and the Covariance

```
# Importing various packages
from math import exp, sqrt
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt

# Sample covariance, note the factor 1/(n-1)
def covariance(x, y, n):
    sum = 0.0
    mean_x = np.mean(x)
    mean_y = np.mean(y)
    for i in range(0, n):
        sum += (x[(i)]-mean_x)*(y[i]-mean_y)
    return  sum/(n-1.)

n = 100
x = np.random.normal(size=n)
print(np.mean(x))
y = 4+3*x+np.random.normal(size=n)
print(np.mean(y))
z = x**3+np.random.normal(size=n)
print(np.mean(z))
covxx = covariance(x,x,n)
covyy = covariance(y,y,n)
covzz = covariance(z,z,n)
covxy = covariance(x,y,n)
covxz = covariance(x,z,n)
covyz = covariance(y,z,n)
print(covxx,covyy, covzz)
print(covxy,covxz, covyz)
w = np.vstack((x, y, z))
#print(w)
c = np.cov(w)
print(c)
#eigen = np.zeros(n)
Eigvals, Eigvecs = np.linalg.eig(c)
print(Eigvals)
```

Random Numbers

Uniform deviates are just random numbers that lie within a specified range (typically 0 to 1), with any one number in the range just as likely as any other. They are, in other words, what you probably think random numbers are. However, we want to distinguish uniform deviates from other sorts of random numbers, for example numbers drawn from a normal (Gaussian) distribution of specified mean and standard deviation. These other sorts of deviates are almost always generated by performing appropriate operations on one or more uniform deviates, as we will see in subsequent sections. So, a reliable source of random uniform deviates, the subject of this section, is an essential building block for any sort of stochastic modeling or Monte Carlo computer work.

Random Numbers, better name: pseudo random numbers

A disclaimer is however appropriate. It should be fairly obvious that something as deterministic as a computer cannot generate purely random numbers.

Numbers generated by any of the standard algorithms are in reality pseudo random numbers, hopefully abiding to the following criteria:

- they produce a uniform distribution in the interval [0,1].
- correlations between random numbers are negligible
- the period before the same sequence of random numbers is repeated is as large as possible and finally
- the algorithm should be fast.

Random number generator RNG

The most common random number generators are based on so-called Linear congruential relations of the type

$$N_i = (aN_{i-1} + c)\text{MOD}(M),$$

which yield a number in the interval [0,1] through

$$x_i = N_i/M$$

The number M is called the period and it should be as large as possible and N_0 is the starting value, or seed. The function MOD means the remainder, that is if we were to evaluate (13)MOD(9), the outcome is the remainder of the division 13/9, namely 4.

Random number generator RNG and periodic outputs

The problem with such generators is that their outputs are periodic; they will start to repeat themselves with a period that is at most M. If however the parameters a and c are badly chosen, the period may be even shorter.

Consider the following example

$$N_i = (6N_{i-1} + 7)\text{MOD}(5),$$

with a seed $N_0 = 2$. This generator produces the sequence $4, 1, 3, 0, 2, 4, 1, 3, 0, 2, \ldots\ldots$, i.e., a sequence with period 5. However, increasing M may not guarantee a larger period as the following example shows

$$N_i = (27N_{i-1} + 11)\text{MOD}(54),$$

which still, with $N_0 = 2$, results in $11, 38, 11, 38, 11, 38, \ldots$, a period of just 2.

Random number generator RNG and its period

Typical periods for the random generators provided in the program library are of the order of $\sim 10^9$ or larger. Other random number generators which have become increasingly popular are so-called shift-register generators. In these generators each successive number depends on many preceding values (rather than the last values as in the linear congruential generator). For example, you could make a shift register generator whose lth number is the sum of the $l - i$th and $l - j$th values with modulo M,

$$N_l = (aN_{l-i} + cN_{l-j})\text{MOD}(M).$$

Random number generator RNG, other examples

Such a generator again produces a sequence of pseudorandom numbers but this time with a period much larger than M. It is also possible to construct more elaborate algorithms by including more than two past terms in the sum of each iteration. One example is the generator of Marsaglia and Zaman which consists of two congruential relations

$$N_l = (N_{l-3} - N_{l-1})\text{MOD}(2^{31} - 69), \tag{12}$$

followed by

$$N_l = (69069N_{l-1} + 1013904243)\text{MOD}(2^{32}), \tag{13}$$

which according to the authors has a period larger than 2^{94}.

Random number generator RNG, other examples

Instead of using modular addition, we could use the bitwise exclusive-OR ($\oplus$) operation so that

$$N_l = (N_{l-i}) \oplus (N_{l-j})$$

where the bitwise action of $\oplus$ means that if $N_{l-i} = N_{l-j}$ the result is 0 whereas if $N_{l-i} \neq N_{l-j}$ the result is 1. As an example, consider the case where $N_{l-i} = 6$ and $N_{l-j} = 11$. The first one has a bit representation (using 4 bits only) which reads 0110 whereas the second number is 1011. Employing the $\oplus$ operator yields 1101, or $2^3 + 2^2 + 2^0 = 13$.

In Fortran90, the bitwise $\oplus$ operation is coded through the intrinsic function IEOR(m, n) where m and n are the input numbers, while in C it is given by $m \wedge n$.

Random number generator RNG, RAN0

We show here how the linear congruential algorithm can be implemented, namely

$$N_i = (aN_{i-1})\text{MOD}(M).$$

However, since a and N_{i-1} are integers and their multiplication could become greater than the standard 32 bit integer, there is a trick via Schrage's algorithm which approximates the multiplication of large integers through the factorization

$$M = aq + r,$$

where we have defined

$$q = [M/a],$$

and

$$r = M \text{ MOD } a.$$

where the brackets denote integer division. In the code below the numbers q and r are chosen so that $r < q$.

Random number generator RNG, RAN0

To see how this works we note first that

$$(aN_{i-1})\text{MOD}(M) = (aN_{i-1} - [N_{i-1}/q]M)\text{MOD}(M), \tag{14}$$

since we can add or subtract any integer multiple of M from aN_{i-1}. The last term $[N_{i-1}/q]M\text{MOD}(M)$ is zero since the integer division $[N_{i-1}/q]$ just yields a constant which is multiplied with M.

Random number generator RNG, RAN0

We can now rewrite Eq. (14) as

$$(aN_{i-1})\text{MOD}(M) = (aN_{i-1} - [N_{i-1}/q](aq + r))\text{MOD}(M), \tag{15}$$

which results in

$$(aN_{i-1})\text{MOD}(M) = (a(N_{i-1} - [N_{i-1}/q]q) - [N_{i-1}/q]r))\,\text{MOD}(M), \tag{16}$$

yielding

$$(aN_{i-1})\text{MOD}(M) = (a(N_{i-1}\text{MOD}(q)) - [N_{i-1}/q]r))\,\text{MOD}(M). \tag{17}$$

Random number generator RNG, RAN0

The term $[N_{i-1}/q]r$ is always smaller or equal $N_{i-1}(r/q)$ and with $r < q$ we obtain always a number smaller than N_{i-1}, which is smaller than M. And since the number $N_{i-1}\text{MOD}(q)$ is between zero and $q-1$ then $a(N_{i-1}\text{MOD}(q)) < aq$. Combined with our definition of $q = [M/a]$ ensures that this term is also smaller than M meaning that both terms fit into a 32-bit signed integer. None of these two terms can be negative, but their difference could. The algorithm below adds M if their difference is negative. Note that the program uses the bitwise $\oplus$ operator to generate the starting point for each generation of a random number. The period of $ran0$ is $\sim 2.1 \times 10^9$. A special feature of this algorithm is that is should never be called with the initial seed set to 0.

Random number generator RNG, RAN0 code

```
    /*
     ** The function
     **            ran0()
     ** is an "Minimal" random number generator of Park and Miller
     ** Set or reset the input value
     ** idum to any integer value (except the unlikely value MASK)
     ** to initialize the sequence; idum must not be altered between
     ** calls for sucessive deviates in a sequence.
     ** The function returns a uniform deviate between 0.0 and 1.0.
     */
double ran0(long &idum)
{
   const int a = 16807, m = 2147483647, q = 127773;
   const int r = 2836, MASK = 123459876;
   const double am = 1./m;
   long     k;
   double   ans;
   idum ^= MASK;
   k = (*idum)/q;
```

```
    idum = a*(idum - k*q) - r*k;
    // add m if negative difference
    if(idum < 0) idum += m;
    ans=am*(idum);
    idum ^= MASK;
    return ans;
} // End: function ran0()
```

Properties of Selected Random Number Generators

As mentioned previously, the underlying PDF for the generation of random numbers is the uniform distribution, meaning that the probability for finding a number x in the interval [0,1] is $p(x) = 1$.

A random number generator should produce numbers which are uniformly distributed in this interval. The table shows the distribution of $N = 10000$ random numbers generated by the functions in the program library. We note in this table that the number of points in the various intervals $0.0 - 0.1$, $0.1 - 0.2$ etc are fairly close to 1000, with some minor deviations.

Two additional measures are the standard deviation σ and the mean $\mu = \langle x \rangle$.

Properties of Selected Random Number Generators

For the uniform distribution, the mean value μ is then

$$\mu = \langle x \rangle = \frac{1}{2}$$

while the standard deviation is

$$\sigma = \sqrt{\langle x^2 \rangle - \mu^2} = \frac{1}{\sqrt{12}} = 0.2886.$$

Properties of Selected Random Number Generators

The various random number generators produce results which agree rather well with these limiting values.

x-bin	ran0	ran1	ran2	ran3
0.0-0.1	1013	991	938	1047
0.1-0.2	1002	1009	1040	1030
0.2-0.3	989	999	1030	993
0.3-0.4	939	960	1023	937
0.4-0.5	1038	1001	1002	992
0.5-0.6	1037	1047	1009	1009
0.6-0.7	1005	989	1003	989
0.7-0.8	986	962	985	954
0.8-0.9	1000	1027	1009	1023
0.9-1.0	991	1015	961	1026
μ	0.4997	0.5018	0.4992	0.4990
σ	0.2882	0.2892	0.2861	0.2915

Simple demonstration of RNGs using python

The following simple Python code plots the distribution of the produced random numbers using the linear congruential RNG employed by Python. The trend displayed in the previous table is seen rather clearly.

```
#!/usr/bin/env python
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import random

# initialize the rng with a seed
random.seed()
counts = 10000
values = np.zeros(counts)
for i in range (1, counts, 1):
    values[i] = random.random()

# the histogram of the data
n, bins, patches = plt.hist(values, 10, facecolor='green')

plt.xlabel('$x$')
plt.ylabel('Number of counts')
plt.title(r'Test of uniform distribution')
plt.axis([0, 1, 0, 1100])
plt.grid(True)
plt.show()
```

Properties of Selected Random Number Generators

Since our random numbers, which are typically generated via a linear congruential algorithm, are never fully independent, we can then define an important test which measures the degree of correlation, namely the so-called auto-correlation function defined previously, see again Eq. (9). We rewrite it here as

$$C_k = \frac{f_d}{\sigma^2},$$

with $C_0 = 1$. Recall that $\sigma^2 = \langle x_i^2 \rangle - \langle x_i \rangle^2$ and that

$$f_d = \frac{1}{nm} \sum_{\alpha=1}^{m} \sum_{k=1}^{n-d} (x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,k+d} - \langle X_m \rangle),$$

The non-vanishing of C_k for $k \neq 0$ means that the random numbers are not independent. The independence of the random numbers is crucial in the evaluation of other expectation values. If they are not independent, our assumption for approximating σ_N is no longer valid.

Autocorrelation function

This program computes the autocorrelation function as discussed in the equation on the previous slide for random numbers generated with the normal distribution $N(0, 1)$.

```
# Importing various packages
from math import exp, sqrt
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt

def autocovariance(x, n, k, mean_x):
    sum = 0.0
    for i in range(0, n-k):
        sum += (x[(i+k)]-mean_x)*(x[i]-mean_x)
    return  sum/n

n = 1000
x=np.random.normal(size=n)
autocor = np.zeros(n)
figaxis = np.zeros(n)
mean_x=np.mean(x)
var_x = np.var(x)
print(mean_x, var_x)
for i in range (0, n):
    figaxis[i] = i
    autocor[i]=(autocovariance(x, n, i, mean_x))/var_x

plt.plot(figaxis, autocor, "r-")
plt.axis([0,n,-0.1, 1.0])
plt.xlabel(r'$i$')
plt.ylabel(r'$\gamma_i$')
plt.title(r'Autocorrelation function')
plt.show()
```

As can be seen from the plot, the first point gives back the variance and a value of one. For the remaining values we notice that there are still non-zero values for the auto-correlation function.

Correlation function and which random number generators should I use

The program here computes the correlation function for one of the standard functions included with the c++ compiler.

```
//  This function computes the autocorrelation function for
//  the standard c++ random number generator

#include <fstream>
#include <iomanip>
#include <iostream>
#include <cmath>
using namespace std;
// output file as global variable
ofstream ofile;

//     Main function begins here
int main(int argc, char* argv[])
{
     int n;
     char *outfilename;

     cin >> n;
```

```
      double MCint = 0.;      double MCintsqr2=0.;
      double invers_period = 1./RAND_MAX; // initialise the random number generator
      srand(time(NULL));  // This produces the so-called seed in MC jargon
      // Compute the variance and the mean value of the uniform distribution
      // Compute also the specific values x for each cycle in order to be able to
      // the covariance and the correlation function
      // Read in output file, abort if there are too few command-line arguments
      if( argc <= 2 ){
        cout << "Bad Usage: " << argv[0] <<
          " read also output file and number of cycles on same line" << endl;
        exit(1);
      }
      else{
        outfilename=argv[1];
      }
      ofile.open(outfilename);
      // Get  the number of Monte-Carlo samples
      n = atoi(argv[2]);
      double *X;
      X = new double[n];
      for (int i = 0;  i < n; i++){
            double x = double(rand())*invers_period;
            X[i] = x;
            MCint += x;
            MCintsqr2 += x*x;
      }
      double Mean = MCint/((double) n );
      MCintsqr2 = MCintsqr2/((double) n );
      double STDev = sqrt(MCintsqr2-Mean*Mean);
      double Variance = MCintsqr2-Mean*Mean;
//    Write mean value and standard deviation
      cout << " Standard deviation= " << STDev << " Integral = " << Mean << endl;

      // Now we compute the autocorrelation function
      double *autocor;  autocor = new double[n];
      for (int j = 0; j < n; j++){
        double sum = 0.0;
        for (int k = 0; k < (n-j); k++){
          sum  += (X[k]-Mean)*(X[k+j]-Mean);
        }
        autocor[j] = sum/Variance/((double) n );
        ofile << setiosflags(ios::showpoint | ios::uppercase);
        ofile << setw(15) << setprecision(8) << j;
        ofile << setw(15) << setprecision(8) << autocor[j] << endl;
      }
      ofile.close();  // close output file
      return 0;
}  // end of main program
```

Which RNG should I use?

- C++ has a class called **random**. The random class contains a large selection of RNGs and is highly recommended. Some of these RNGs have very large periods making it thereby very safe to use these RNGs in case

one is performing large calculations. In particular, the Mersenne twister random number engine has a period of 2^{19937}.

- Add RNGs in Python

How to use the Mersenne generator

The following part of a c++ code (from project 4) sets up the uniform distribution for $x \in [0, 1]$.

```
/*

//  You need this
#include <random>

// Initialize the seed and call the Mersienne algo
std::random_device rd;
std::mt19937_64 gen(rd());
// Set up the uniform distribution for x \in [[0, 1]
std::uniform_real_distribution<double> RandomNumberGenerator(0.0,1.0);

// Now use the RNG
int ix = (int) (RandomNumberGenerator(gen)*NSpins);
```

Why blocking?

Statistical analysis.

- Monte Carlo simulations can be treated as *computer experiments*
- 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.

A very good article which explains blocking is H. Flyvbjerg and H. G. Petersen, *Error estimates on averages of correlated data*, Journal of Chemical Physics 91, 461-466 (1989).

Why blocking?

Statistical analysis.

- As in other experiments, Monte Carlo 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. (In VMC a common source is the step length or time step in importance sampling)

Code to demonstrate the calculation of the autocorrelation function

The following code computes the autocorrelation function, the covariance and the standard deviation for standard RNG. The following file gives the code.

```
//  This function computes the autocorrelation function for
//  the Mersenne random number generator with a uniform distribution
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <random>
#include <armadillo>
#include <string>
#include <cmath>
using namespace  std;
using namespace arma;
// output file
ofstream ofile;

//     Main function begins here
int main(int argc, char* argv[])
{
  int MonteCarloCycles;
  string filename;
  if (argc > 1) {
    filename=argv[1];
    MonteCarloCycles = atoi(argv[2]);
    string fileout = filename;
    string argument = to_string(MonteCarloCycles);
    fileout.append(argument);
    ofile.open(fileout);
  }

  // Compute the variance and the mean value of the uniform distribution
  // Compute also the specific values x for each cycle in order to be able to
  // compute the covariance and the correlation function

  vec X  = zeros<vec>(MonteCarloCycles);
  double MCint = 0.;      double MCintsqr2=0.;
  std::random_device rd;
  std::mt19937_64 gen(rd());
  // Set up the uniform distribution for x \in [[0, 1]
  std::uniform_real_distribution<double> RandomNumberGenerator(0.0,1.0);
  for (int i = 0;  i < MonteCarloCycles; i++){
    double x =   RandomNumberGenerator(gen);
    X(i) = x;
    MCint += x;
    MCintsqr2 += x*x;
  }
  double Mean = MCint/((double) MonteCarloCycles );
  MCintsqr2 = MCintsqr2/((double) MonteCarloCycles );
```

```
  double STDev = sqrt(MCintsqr2-Mean*Mean);
  double Variance = MCintsqr2-Mean*Mean;
  //   Write mean value and variance
  cout << " Sample variance= " << Variance  << " Mean value = " << Mean << endl;
  // Now we compute the autocorrelation function
  vec autocorrelation = zeros<vec>(MonteCarloCycles);
  for (int j = 0; j < MonteCarloCycles; j++){
    double sum = 0.0;
    for (int k = 0; k < (MonteCarloCycles-j); k++){
      sum  += (X(k)-Mean)*(X(k+j)-Mean);
    }
    autocorrelation(j) = sum/Variance/((double) MonteCarloCycles );
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << setw(15) << setprecision(8) << j;
    ofile << setw(15) << setprecision(8) << autocorrelation(j) << endl;
  }
  // Now compute the exact covariance using the autocorrelation function
  double Covariance = 0.0;
  for (int j = 0; j < MonteCarloCycles; j++){
    Covariance  += autocorrelation(j);
  }
  Covariance *=  2.0/((double) MonteCarloCycles);
  // Compute now the total variance, including the covariance, and obtain the standard deviation
  double TotalVariance = (Variance/((double) MonteCarloCycles ))+Covariance;
  cout << "Covariance =" << Covariance << "Totalvariance= " << TotalVariance << "Sample Variance/n
  cout << " STD from sample variance= " << sqrt(Variance/((double) MonteCarloCycles )) << " STD w

  ofile.close();  // close output file
  return 0;
}  // end of main program
```

What is blocking?

Blocking.

- Say that we have a set of samples from a Monte Carlo experiment
- Assuming (wrongly) that our samples are uncorrelated our best estimate of the standard deviation of the mean $\langle \mathbf{M} \rangle$ is given by

$$\sigma = \sqrt{\frac{1}{n}\left(\langle \mathbf{M}^2 \rangle - \langle \mathbf{M} \rangle^2\right)}$$

- If the samples are correlated we can rewrite our results to show that

$$\sigma = \sqrt{\frac{1+2\tau/\Delta t}{n}\left(\langle \mathbf{M}^2 \rangle - \langle \mathbf{M} \rangle^2\right)}$$

where τ is the correlation time (the time between a sample and the next uncorrelated sample) and Δt is time between each sample

What is blocking?

Blocking.

- If $\Delta t \gg \tau$ our first estimate of σ still holds

- Much more common that $\Delta t < \tau$
- In the method of data blocking we divide the sequence of samples into blocks
- We then take the mean $\langle \mathbf{M}_i \rangle$ of block $i = 1 \ldots n_{blocks}$ to calculate the total mean and variance
- The size of each block must be so large that sample j of block i is not correlated with sample j of block $i + 1$
- The correlation time τ would be a good choice

What is blocking?

Blocking.

- Problem: We don't know τ or it is too expensive to compute
- Solution: Make a plot of std. dev. as a function of blocksize
- The estimate of std. dev. of correlated data is too low $\rightarrow$ the error will increase with increasing block size until the blocks are uncorrelated, where we reach a plateau
- When the std. dev. stops increasing the blocks are uncorrelated

Implementation

- Do a Monte Carlo simulation, storing all samples to file
- Do the statistical analysis on this file, independently of your Monte Carlo program
- Read the file into an array
- Loop over various block sizes
- For each block size n_b, loop over the array in steps of n_b taking the mean of elements $in_b, \ldots, (i + 1)n_b$
- Take the mean and variance of the resulting array
- Write the results for each block size to file for later analysis

Actual implementation with code, main function

When the file gets large, it can be useful to write your data in binary mode instead of ascii characters. The following python file reads data from file with the output from every Monte Carlo cycle.

```
    # Blocking
        @timeFunction
        def blocking(self, blockSizeMax = 500):
            blockSizeMin = 1

            self.blockSizes = []
            self.meanVec = []
            self.varVec = []

            for i in range(blockSizeMin, blockSizeMax):
                if(len(self.data) % i != 0):
                    pass#continue
                blockSize = i
                meanTempVec = []
                varTempVec = []
                startPoint = 0
                endPoint = blockSize

                while endPoint <= len(self.data):
                    meanTempVec.append(np.average(self.data[startPoint:endPoint]))
                    startPoint = endPoint
                    endPoint += blockSize
                mean, var = np.average(meanTempVec), np.var(meanTempVec)/len(meanTempVec)
                self.meanVec.append(mean)
                self.varVec.append(var)
                self.blockSizes.append(blockSize)

            self.blockingAvg = np.average(self.meanVec[-200:])
            self.blockingVar = (np.average(self.varVec[-200:]))
            self.blockingStd = np.sqrt(self.blockingVar)
```

The Bootstrap method

The Bootstrap resampling method is also very popular. It is very simple:

1. Start with your sample of measurements and compute the sample variance and the mean values

2. Then start again but pick in a random way the numbers in the sample and recalculate the mean and the sample variance.

3. Repeat this K times.

It can be shown, see the article by Efron that it produces the correct standard deviation.

This method is very useful for small ensembles of data points.

Bootstrapping

Given a set of N data, assume that we are interested in some observable θ which may be estimated from that set. This observable can also be for example the result of a fit based on all N raw data. Let us call the value of the observable obtained from the original data set $\hat{\theta}$. One recreates from the sample repeatedly other samples by choosing randomly N data out of the original set. This costs essentially nothing, since we just recycle the original data set for the building of new sets.

Bootstrapping, recipe

Let us assume we have done this K times and thus have K sets of N data values each. Of course some values will enter more than once in the new sets. For each of these sets one computes the observable θ resulting in values θ_k with $k = 1, ..., K$. Then one determines

$$\tilde{\theta} = \frac{1}{K} \sum_{k=1}^{K} \theta_k,$$

and

$$sigma_{\tilde{\theta}}^2 = \frac{1}{K} \sum_{k=1}^{K} \left(\theta_k - \tilde{\theta}\right)^2 .$$

These are estimators for $\angle\theta\rangle$ and its variance. They are not unbiased and therefore $\tilde{\theta} \neq \hat{\theta}$ for finite K.

The difference is called bias and gives an idea on how far away the result may be from the true $\angle\theta\rangle$. As final result for the observable one quotes $\angle\theta\rangle = \tilde{\theta} \pm \sigma_{\tilde{\theta}}$.

Bootstrapping, code

```
# Bootstrap
    @timeFunction
    def bootstrap(self, nBoots = 1000):
        bootVec = np.zeros(nBoots)
        for k in range(0,nBoots):
            bootVec[k] = np.average(np.random.choice(self.data, len(self.data)))
        self.bootAvg = np.average(bootVec)
        self.bootVar = np.var(bootVec)
        self.bootStd = np.std(bootVec)
```

Jackknife, code

```
# Jackknife
    @timeFunction
    def jackknife(self):
        jackknVec = np.zeros(len(self.data))
        for k in range(0,len(self.data)):
            jackknVec[k] = np.average(np.delete(self.data, k))
        self.jackknAvg = self.avg - (len(self.data) - 1) * (np.average(jackknVec) - self.avg)
```

```
self.jackknVar = float(len(self.data) - 1) * np.var(jackknVec)
self.jackknStd = np.sqrt(self.jackknVar)
```