Autocorrelation functions

Date: $Date: 2012-04-28 23:56:18 +0200 (sam. 28 avril 2012) $
Revision:$Revision: 113 $
Author: Thibauld Nion

> **Hint**
>
> This file has been generated from a Python script. This script can be executed, being its own illustration. But it can also be imported so that the implemented functions can be directly used in other scripts.

> **Note**
>
> This file displays several equations, if you experience any problem in the way they are displayed please refer to the troubleshooting instructions.

Introduction

Purpose

After giving a short presentation about the classical definitions and uses of the correlation functions, this document will focus on autocovariance and autocorrelation functions.

More precisely, what is commonly considered as implementation "details" will be described and illustrated in the specific (but quite common) case when one wants to use the Fast Fourier Transforms.

Notations

Let's have a quick look at the notations that will be used in this document (some of them will be explained in more details later):

$\star$
: cross-correlation *operator*

$*$
: convolution

$conj(f)$
: complex conjugate of f

f_{sym}
: symetrised f (around the origin)

f_{cycl}
: periodised f (on an infinite domain)

$\mathcal{F}$
: Fourier transform

Implementations

In order to provide sample code for the implementation for the various definitions and also to generate illustrations, we will use the Python language and two well known libraries:

- Matplotlib's pylab interface : http://matplotlib.sourceforge.net
- Numpy : http://numpy.scipy.org/

TABLE OF CO

Autocorrelation func
Introduction
Bestiary
Estimations
Implementation by
Illustrations
References

```
import pylab as pl
import numpy as np
```

Bestiary

In this section we will pick a few of the numerous definitions that relate to the concepts of covariance and correlation. Along the way we will give a few words of each definition's possible uses.

Please also consider reading the referenced documents as they often give better and more precise definitions than what's stated here – which is at best a rephrasal of those documents, anyway.

Statistics

Covariance

When dealing with random variables, one may define the covariance between two such variables, with the following formula:

$$cov(f, g) = E[(f - E(f)) \cdot (g - E(g))]$$

Where $E(f)$ is the *expected value* of f, commonly estimated by its average value.

> **Note**
>
> The action of subtracting the expected value to a random variable is called "centering" and $f - E(f)$ may be called a centered variable (because it's expected value is 0)

From which we can retrieve the formula for the variance, simply by applying the formula to a random variable and ... itself:

$$var(f) = cov(f, f) = E\left[(f - E(f))^2\right]$$

Without going into further details for these well known statistics we can say that the variance is quite helpful to know how much we can rely on the expected value (aka "mean") of a random variable, and can also be used to fit a gaussian model on the distribution of this random variable.

Correlation coefficient

The covariance is used to get an idea of how two random variables are linked together, but the prefered way to do so is with the correlation coefficient, also known as Pearson's coefficient [WikiPrsCf]:

$$corr(f, g) = \frac{cov(f,g)}{\sqrt{(var(f) \cdot var(g))}}$$

> **Note**
>
> An interesting feature of Pearson's coefficient is its normalisation: the values it can take are all between -1 and 1, 1 meaning that either variable can be almost perfectly described as a linear function of the other one.

Digital Signal Processing

In digital signal processing (DSP), all the work is done (obviously) on signals which can be for instance functions of time or space. Usually they are defined on a discrete space and their values can be indexed by integers: for a function f the ith value is noted f_i.

Cross-correlation operator

In DSP, there is an operation called either cross-correlation or cross-covariance, defined by the following formula (see also [Weisst.XC], [WikiXCorr]):

$$(f \star g)_i = \sum_j (conj(f_j) \cdot g_{i+j})$$

In the above formula i is an offset, that can correspond to a time shift for instance. This makes it possible to express the above operation by using the well known convolution operator:

$$f \star g = conj(f_{sym}) * g$$

Where f_{sym} is so that $f_{sym}(t) = f(-t)$.

Broadly speaking this correlation operator is used to detect similarities between two signals. Which can be understood by the idea that f is basically being translated over g and, for every possible shift, "compared" to g. This is so that the correlation operator helps to detect the presence of one signal *inside* another one (see also an illustration of that in DSPGuide [SmithDSPG]).

When applied to the same signal (e.g. considering $f \star f$), this helps detecting self-similarities in the signal, indicating the distances between consecutive repetitions of a same pattern and, by extension, giving some insight about the typical scales characterizing multiscale signals.

> **Note**
>
> If we compare these definitions with the statistical ones, it is interesting to note that there is no notion of *centering*.

Bridging the gap

Concerning the "statistical" definitions we began with, the big difference between them and the one for DSP is that they don't explicitly take into account any concept of time or space. However they are all but incompatible and that's what we are going to see in this section, starting with the following two statements.

When we consider f and g as random variables we can draw values for them, each value being called a *realisation* of the random variable.

When we consider f and g as discrete signals, their values at each time step can be considered as realisations of an underlying random variable which is a basic way to bridge the gap between both sets of definitions.

Autocovariance

The link can then be made with the statistical notion of *autocovariance* for temporal series. Such a series (f_i) can be considered as part of a random process f where, for any i, f_i is a random variable (see also [WikiStoch]).

Studying the structure of such a process can then be done by comparing all couples of random variables f_i and f_j by the mean of the covariance (except that since both variables are part of the same random process, we call that an **autocovariance**):

$$autocov_f(i,j) = E\Big[(f_i - \mu_i)(f_j - \mu_j)\Big]$$

where μ_i and μ_j are the expectations for f_i and f_j.

If the process is *second order stationnary* then things are getting interesting since it basically implies that, as far as the covariance measure is concerned, the only visible feature is the separation between f_i and f_j (ie $h = j - i$) and that for a given h we have the identity:

$autocov_f(i, i+h) = autocov_f(j, j+h)$ for any i, j

The formula then simplies to:

$$autocov_f(h) = E\big[(f_i - \mu)(f_{i+h} - \mu)\big]$$

> **Note**
>
> In the later case the difference between the indices i and j can be interpreted as a time shift (or "lag") or a space offset which is the root idea behind most of the practical application of this measure.

Autocorrelation

Wikipedia has a very nice page about autocorrelation [WikiACorr] where the statistical definition is clearly explained, and where the link between the autocorrelation and the DSP definition we've talked about is explained too.

As a consequence there is no need to go into further details here. Suffices to say that I stick to the idea that *autocovariance* and *autocorrelation* are two different things (the Wikipedia article judiciously points at the fact that both terms are sometimes used for the same formula...).

The autocorrelation formula expressed for a *second order stationnary* series (f_i) is then:

$$autocorr_f(i,j) = \frac{E\left[(f_i-\mu)(f_j-\mu)\right]}{var(f)}$$

Estimations

Let's now consider the computing side of the problem. For the sake of simplicity only the *autocovariance* is discussed here.

> **Note**
>
> Obtaining the autocorrelation from the autocovariance is usually just a matter of dividing the later by its value in 0 (considering that $autocov_f(0) = var(f)$).

Another important simplification is made, in that we will assume that the signals are already *centered*, meaning that their mean is assumed to be 0.

> **Note**
>
> There is an obvious way to get a centered signal from any signal by estimating its expected value (ie computing the average) and subtracting it from the signal.

> **Important**
>
> This basic centering method will be enough for the illustrations we intend to provide here but one should be aware that numerical instabilities can araise from such a procedure (see [WikiAlVar] for a discution on a similar topic).

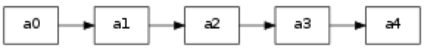
Standard formula

There is actually two well known formula to perform a direct computation of the autocovariance.

Truncated estimator

The first one is an adaptation of an unbiased estimator, and will be refered to as the **truncated estimator** in this document (this name probably does not hold outside this document by the way):

$$autocov_f(h) = \frac{1}{N-h}\sum_{i=0}^{N-h}(f(i)f(i+h))$$

> **Example**
>
> To get an idea, let's apply it on the following small signal.
>
> 
>
>
> - $autocov_f(0) = \frac{a_0 \cdot a_0 + a_1 \cdot a_1 + a_2 \cdot a_2 + a_3 \cdot a_3 + a_4 \cdot a_4}{5}$
> - $autocov_f(1) = \frac{a_0 \cdot a_1 + a_1 \cdot a_2 + a_2 \cdot a_3 + a_3 \cdot a_4}{4}$
> - $autocov_f(2) = \frac{a_0 \cdot a_2 + a_1 \cdot a_3 + a_2 \cdot a_4}{3}$
> - $autocov_f(3) = \frac{a_0 \cdot a_3 + a_1 \cdot a_4}{2}$
> - $autocov_f(4) = \frac{a_0 \cdot a_4}{1}$

This formula has a very basic advantage over the other ones: it only works on the signal itself without adding anything to it. However it has a couple of shortcommings:

- for $h >= 1$ only a part of the signal is taken into account a part of the signal. It is likely that the parts taken into account have their average values different from 0, thus introducing a bias with respect to our initial assumption.
- the closer h is to N the fewer samples are taken into account (eg. for $h = 1$ only one term is "summed up"), thus making the computed values less and less trustworthy.

Cyclic estimator

Another estimation can be performed via the **cyclic estimator** than can be expressed as folows:

$$autocov_f(h) = \frac{1}{N} \sum_{i=1}^{N} f_{cycl}(i) f_{cycl}(i+h)$$

Where $f_{cycl}(i) = f(i \; modulo \; N)$.

> **Example**
>
> To get an idea, let's apply it on the following small signal where the repetition of the signal is represented by a backward directed arrow.
>
> 
>
>
> - $autocov_f(0) = \frac{a_0 \cdot a_0 + a_1 \cdot a_1 + a_2 \cdot a_2 + a_3 \cdot a_3 + a_4 \cdot a_4}{5}$
> - $autocov_f(1) = \frac{a_0 \cdot a_1 + a_1 \cdot a_2 + a_2 \cdot a_3 + a_3 \cdot a_4 + a_4 \cdot a_0}{5}$
> - $autocov_f(2) = \frac{a_0 \cdot a_2 + a_1 \cdot a_3 + a_2 \cdot a_4 + a_3 \cdot a_0 + a_4 \cdot a_1}{5}$
> - $autocov_f(3) = \frac{a_0 \cdot a_3 + a_1 \cdot a_4 + a_2 \cdot a_0 + a_3 \cdot a_1 + a_4 \cdot a_2}{5}$
> - $autocov_f(4) = \frac{a_0 \cdot a_4 + a_1 \cdot a_0 + a_2 \cdot a_1 + a_3 \cdot a_2 + a_4 \cdot a_3}{5}$

Clearly the two big biases of the "truncated estimator" are circumvented by this formula. However it does have an annoying shortcoming: since it basically consists in sticking a copy of the signal head to tail with the signal itself, it may change the structure of the signal.

The most obvious artefact is the introduction of an apparent periodicity (the period being the size of the signal), but modifications in the signal structure do occur as soon as $h >= 1$.

> **Note**
>
> For both methods it is clear that the bigger h is the more $autocov_f(h)$ is biased. As a rule of thumb it is usually best to limit oneself to study the range $[0, N/2]$ when considering the autocovariance measure of a signal of length N.

Implementation by FFT

Convolution and correlaton

As explained in [SmithDSPG], we can use a simple relation between the convolution of two functions and the product of their Fourier transforms to get the following formula for what we called earlier the "DSP cross-correlation operator".

$$f \star g = \mathcal{F}^{-1}\left[\overline{\mathcal{F}(f)} \cdot \mathcal{F}(g)\right]$$

This formula is already suitable for a lot of applications of the correlation operator. However some work remains to be done to get a proper estimation of the autocovariance and autocorrelation function.

But, before going any further, let's point out at another trivia. When applied to the same signal, the formula simplifies itself into the following one.

$$f \star f = \mathcal{F}^{-1}\left[|\mathcal{F}(f)|^2\right]$$

Which explains why the autocorrelation is also commonly linked to a signal's power spectral density [WikiSpecD].

Implicit periodisation and cyclic estimator

Implementing Fourier transform on discrete signals usually induces an implicit periodisation of the signal, much like what we saw with the "cyclic estimator". Actually, the link between both is easy to see for a real signal:

$$autocov_f(h) = \frac{1}{N}\sum_{i=1}^{N} f_{cycl}(i) f_{cycl}(i+h)$$
$$autocov_f(h) = \frac{1}{N}\sum_{i=1}^{N} f_{cycl} \star f_{cycl}$$
$$autocov_f(h) = \frac{1}{N}\mathcal{F}^{-1}\left[|\mathcal{F}(f)|^2\right]$$

So if the biases of the cyclic estimator are acceptable for one's applications, the above formula is suitable to compute the autocovariance of a signal via the Fourier transform.

We can then propose the following function, computing the cyclic estimation of a signal via FFT.

fftCyclicAutocovariance1D

```
def fftCyclicAutocovariance1D(signal):
    """
    Given a 1D signal, return an estimation of its autocovariance
    function.

    The estimation is made by considering that the input signal
    actually desribes a full period of a wider, cyclic signal. The
    estimation is then the autocovariance of this wider signal.

    Uses the Fast Fourier Transform internally.
    """
```

Get a centered version of the signal.

```
centered_signal = signal - np.mean(signal)
```

Then the Fourier transform is computed using the FFT

```
ft_signal = np.fft.fft(centered_signal)
```

We get the autocovariance by taking the inverse transform of the power spectral density.

```
powerSpectralDensity = np.abs(ft_signal)**2
autocovariance = np.fft.ifft(powerSpectralDensity) / len(centered_signal)
```

All values of the autocovariance function are pure real numbers, we can get rid of the "complex" representation resulting from the use of the FFT.

```
return np.real(autocovariance)
```

fftCyclicAutocorrelation1D

```
def fftCyclicAutocorrelation1D(signal):
    """
    Given a 1D signal, return an estimation of its autocorrelation
    function.

    The autocorrelation is obtained by normalizing the autocovariance
    function computed by fftCyclicAutocovariance1D.
    """
```

Get the autocovariance function from the previously defined function.

```
autocovariance = fftCyclicAutocovariance1D(signal)
```

The normalization is made by the variance of the signal, which corresponds to the very first value of the autocovariance.

```
variance = autocovariance.flat[0]
```

Taking care of the case when the signal has zero variance (when it is constant nearly everywhere), we can then proceed to the normalisation.

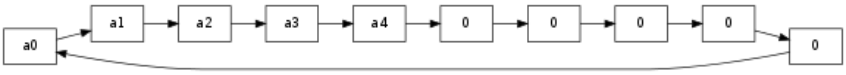
```
if variance==0.:
    return np.zeros(autocovariance.shape)
else:
    return (autocovariance / variance)
```

Padding and truncated estimator

Unfortunately speeding up the computation of a Fourier transform quite often poses some requirement on the size of the signal to process. This is due to the fact that speedy algorithms for the Fast Fourier Transform require the length of a signal to a power of 2 (some others are specialised in evenly sized signals, or sizes that are multiples of 3, 5 etc).

To match these contraints, padding the signal with a bunch of zeros is quite common (and easy to do). However, it induces an additional annoying bias to the measure we've just implemented.

Happily there is a fairly simple way to use a padding and an FFT algorithm to get the expected result of the "truncated estimator" for the autocovariance.

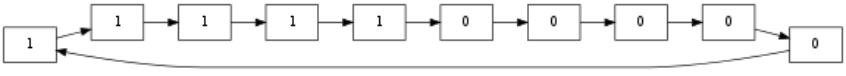
> **Example**
>
> Let's see what the FFT-based "DSP correlation" operator will get us on a padded signal.
>
> 
>
>
> - $(f \star f)(0) = a_0 \cdot a_0 + a_1 \cdot a_1 + a_2 \cdot a_2 + a_3 \cdot a_3 + a_4 \cdot a_4$
> - $(f \star f)(1) = a_0 \cdot a_1 + a_1 \cdot a_2 + a_2 \cdot a_3 + a_3 \cdot a_4$
> - $(f \star f)(2) = a_0 \cdot a_2 + a_1 \cdot a_3 + a_2 \cdot a_4$
> - $(f \star f)(3) = a_0 \cdot a_3 + a_1 \cdot a_4$
> - $(f \star f)(4) = a_0 \cdot a_4$

From the previous example we can see that, **when the signal is padded**, we have the following relation with the truncated estimator (for $h < N$):

$$autocov_f(h) = \frac{f \star f}{N-h}$$

> **Note**
>
> This is true for the previous example because we padded the signal with as many zeroes as initial samples (thus doubling the signal length). When the padding is smaller, the formula only holds for h lower than the padding's size.

The above relation can be described as the correction of a "mask effect" because it boils down to dividing $f \star f$ by the "DSP autocorrelation" of a signal equal to 1 on the original domain of f and equal to 0 on the padding's domain. The example below should make it clearer:

> **Example**
>
> Let's see what the FFT-based "DSP correlation" operator will get us on a mask signal indicating which sample corresponds to the original signal and which belongs to the padding.
>
> 
>

- $(mask \star mask)(0) = 5$
- $(mask \star mask)(1) = 4$
- $(mask \star mask)(2) = 3$
- $(mask \star mask)(3) = 2$
- $(mask \star mask)(4) = 1$

fftAutocovariance1D

```
def fftAutocovariance1D(signal):
    """
    Compute the autocovariance of the input 1D signal.

    Consider the input signal to be a representative sample of a wider
    signal that has no other pattern that those present on the sample
    (this is what "representative" stands for) and especially no
    pattern whose scale is higer or equal to the input signal's size
    (this is for the difference with fftCyclicAutocovariance1D).

    The autocovariance is computed by a FFT and with a zero padding
    made in order to double the size of the signal. However the
    returned function is of the same size as the signal.
    """
```

Get a centered version of the signal.

```
centered_signal = signal - np.mean(signal)
```

Pad the centered signal with zeros, in such a way that the padded signal is twice as big as the input.

```
zero_padding = np.zeros_like(centered_signal)
padded_signal = np.concatenate(( centered_signal,
                                 zero_padding    ))
```

Then the Fourier transform is computed using the FFT.

```
ft_signal = np.fft.fft(padded_signal)
```

We get an erroneous autocovariance by taking the inverse transform of the power spectral density.

```
pseudo_powerSpectralDensity = np.abs(ft_signal)**2
pseudo_autocovariance = np.fft.ifft(pseudo_powerSpectralDensity)
```

We repeat the same process (except for centering) on a 'mask' signal, in order to estimate the error made on the previous computation.

```
input_domain  = np.ones_like(centered_signal)
mask = np.concatenate(( input_domain, zero_padding ))
ft_mask = np.fft.fft(mask)
mask_correction_factors = np.fft.ifft(np.abs(ft_mask)**2)
```

The "error" made can now be easily corrected by an element-wise division.

```
autocovariance = pseudo_autocovariance / mask_correction_factors
```

All values of the autocovariance function are pure real numbers, we can get rid of the "complex" representation resulting from the use of the FFT. Also we make sure that we return a signal of the same size as the input signal.

```
return np.real(autocovariance[0:len(signal)])
```

fftAutocorrelation1D

```
def fftAutocorrelation1D(signal):
    """
    Given a 1D signal, return an estimation of its autocorrelation
    function.

    The autocorrelation is obtained by normalizing the autocovariance
    function computed by fftAutocovariance1D.
    """
```

Get the autocovariance function from the previously defined function.

```
autocovariance = fftAutocovariance1D(signal)
```

The normalization is made by the variance of the signal, which corresponds to the very first value of the autocovariance.

```
variance = autocovariance[0]
```

Taking care of the case when the signal has zero variance (when it is constant nearly everywhere), we can then proceed to the normalisation.

```
if variance==0.:
    return np.zeros(autocovariance.shape)
else:
    return (autocovariance / variance)
```

> **Note**
>
> It can seem computationally expensive to compute the correction factors as the "DSP autocorrelation", and it is, indeed !
>
> However, if you deal with several signals of the same size, the result can be saved for multiple reuse, and it has a hidden advantage: FFT algorithms and libraries uses various kinds of "normalisation" factor (be it 2π, simply 2 or anything else).
>
> Computing the correlations of both the signal and the mask and dividing the former by the later will just make it invisibile to you what normalisation convention was used to compute the FFT.

Autocovariance for sets

There exists another definition of the autocovariance function for binary signals that differs from all the previous ones by a major point: it is computed **without centering**.

This measure actually corresponds to the "DSP cross-correlation operator".

It is quite interesting when you consider the binary signal as a discrete representation of "sets" (that would be simple segments in 1D, but might be more interesting shapes in higher dimensions), because its values can be directly interpreted in terms of probability of intersections between thoses sets and their translated images.

This specific definition can actually be implemented in the same way as before, by just skipping the centering step.

fftSetAutocovariance1D

```
def fftSetAutocovariance1D(binarySignal):
    """
    Compute an autocovariance function without any centering, on a
    given binary signal considered as a discrete description of a set.

    The computed autocovariance will represent, for each translation
    vector h, the probability that a point belongs both to the initial
    set and its translated image.

    The measure computed by FFT, using a zero padding as with
    fftAutocovariance1D.
    """
```

Pad the binary signal with zeros, in such a way that the padded signal is twice as big as the input.

```
zero_padding = np.zeros_like(binarySignal)
padded_signal = np.concatenate(( binarySignal,
                                 zero_padding    ))
```

Then the Fourier transform is computed using the FFT.

```
ft_signal = np.fft.fft(padded_signal)
```

We get an erroneous autocovariance by taking the inverse transform of the power spectral density.

```
pseudo_powerSpectralDensity = np.abs(ft_signal)**2
pseudo_autocovariance = np.fft.ifft(pseudo_powerSpectralDensity)
```

We repeat the same process on a 'mask' signal, in order to estimate the error made on the previous computation.

```
input_domain  = np.ones_like(centered_signal)
mask = np.concatenate(( input_domain, zero_padding ))
ft_mask = np.fft.fft(mask)
mask_correction_factors = np.fft.ifft(np.abs(ft_mask)**2)
```

The "error" made can now be easily corrected by an element-wise division.

```
autocovariance = pseudo_autocovariance / mask_correction_factors
```

All values of the autocovariance function are pure real numbers, we can get rid of the "complex" representation resulting from the use of the FFT. Also we make sure that we return a signal of the same size as the input signal.

```
return np.real(autocovariance[0:len(binarySignal)])
```

Higher dimensions

As usual, it is theoretically trivial to extend the previous definitions and algorithms to higher dimensions.

More interestingly though, and provided that you have a good FFT library at your disposal, this is also trivial in practice :) .

fftCyclicAutocovariance

```
def fftCyclicAutocovariance(signal):
    """
    Given a n-dimensional signal, return an estimation of its
    autocovariance function.

    The estimation is made by considering that the input signal
    actually desribes a full period of a wider, cyclic signal. The
    estimation is then the autocovariance of this wider signal.

    Uses the Fast Fourier Transform internally.
    """
```

Get a centered version of the signal.

```
centered_signal = signal - np.mean(signal)
```

Then the Fourier transform is computed using the FFT

```
ft_signal = np.fft.fftn(centered_signal)
```

We get the autocovariance by taking the inverse transform of the power spectral density.

```
    powerSpectralDensity = np.abs(ft_signal)**2
    autocovariance = np.fft.ifftn(powerSpectralDensity) / len(centered_signal)
```

All values of the autocovariance function are pure real numbers, we can get rid of the "complex" representation resulting from the use of the FFT.

```
    return np.real(autocovariance)
```

fftCyclicAutocorrelation

```
def fftCyclicAutocorrelation(signal):
    """
    Given a n-dimensional signal, return an estimation of its
    autocorrelation function.

    The autocorrelation is obtained by normalizing the autocovariance
    function computed by fftCyclicAutocovariance.
    """
```

Get the autocovariance function from the previously defined function.

```
    autocovariance = fftCyclicAutocovariance(signal)
```

The normalization is made by the variance of the signal, which corresponds to the very first value of the autocovariance.

```
    variance = autocovariance.flat[0]
```

Taking care of the case when the signal has zero variance (when it is constant nearly everywhere), we can then proceed to the normalisation.

```
    if variance==0.:
        return np.zeros(autocovariance.shape)
    else:
        return (autocovariance / variance)
```

fftAutocovariance

```
def fftAutocovariance(signal):
    """
    Compute the autocovariance of the input n-dimensional signal.

    Consider the input signal to be a representative sample of a wider
    signal that has no other pattern that those present on the sample
    (this is what "representative" stands for) and especially no
    pattern whose scale is higer or equal to the input signal's size
    on each of its dimensions (this is for the difference with
    fftCyclicAutocovariance).

    The autocovariance is computed by a FFT and with a zero padding
    made in such a way that the padded signal is `2**n` bigger than
    the input one (where n is the dimension). However the returned
    function is of the same size as the signal on every dimension.
    """
```

Get a centered version of the signal.

```
    centered_signal = signal - np.mean(signal)
```

For code brevity, we will use a convenient option of the fft library we're using, thanks to which the padding will be automatically handled by the fft function itself.

```
    padded_shape = [2*s+1 for s in centered_signal.shape]
    ft_signal = np.fft.fftn(centered_signal, padded_shape)
```

We get an erroneous autocovariance by taking the inverse transform of the power spectral density.

```
pseudo_powerSpectralDensity = np.abs(ft_signal)**2
pseudo_autocovariance = np.fft.ifftn(pseudo_powerSpectralDensity)
```

We repeat the same process (except for centering) on a 'mask' signal, in order to estimate the error made on the previous computation.

```
input_domain  = np.ones_like(centered_signal)
ft_mask = np.fft.fftn(input_domain, padded_shape)
mask_correction_factors = np.fft.ifftn(np.abs(ft_mask)**2)
```

The "error" made can now be easily corrected by an element-wise division.

```
autocovariance = pseudo_autocovariance / mask_correction_factors
```

All values of the autocovariance function are pure real numbers, we can get rid of the "complex" representation resulting from the use of the FFT. Also we make sure that we return a signal of the same size as the input signal, on each dimension.

```
crop_slices = [slice(i) for i in signal.shape]
return np.real(autocovariance[crop_slices])
```

fftAutocorrelation

```
def fftAutocorrelation(signal):
    """
    Given a n-dimensional signal, return an estimation of its
    autocorrelation function.

    The autocorrelation is obtained by normalizing the autocovariance
    function computed by fftAutocovariance.
    """
```

Get the autocovariance function from the previously defined function.

```
autocovariance = fftAutocovariance(signal)
```

The normalization is made by the variance of the signal, which corresponds to the very first value of the autocovariance.

```
variance = autocovariance.flat[0]
```

Taking care of the case when the signal has zero variance (when it is constant nearly everywhere), we can then proceed to the normalisation.

```
if variance==0.:
    return np.zeros(autocovariance.shape)
else:
    return (autocovariance / variance)
```

fftSetAutocovariance

```
def fftSetAutocovariance(binary_signal):
    """
    Compute an autocovariance function without any centering, on a
    given binary signal considered as a discrete description of a set.

    The computed autocovariance will represent, for each translation
    vector h, the probability that a point belongs both to the initial
    set and its translated image.

    The measure computed by FFT, using a zero padding as with
```

```
    fftAutocovariance.
    """
```

Compute the FFT with zero padding.

```
padded_shape = [2*s+1 for s in binary_signal.shape]
ft_signal = np.fft.fftn(binary_signal, padded_shape)
```

We get an erroneous autocovariance by taking the inverse transform of the power spectral density.

```
pseudo_powerSpectralDensity = np.abs(ft_signal)**2
pseudo_autocovariance = np.fft.ifftn(pseudo_powerSpectralDensity)
```

We repeat the same process (except for centering) on a 'mask' signal, in order to estimate the error made on the previous computation.

```
input_domain  = np.ones_like(binary_signal)
ft_mask = np.fft.fftn(input_domain, padded_shape)
mask_correction_factors = np.fft.ifftn(np.abs(ft_mask)**2)
```

The "error" made can now be easily corrected by an element-wise division.

```
autocovariance = pseudo_autocovariance / mask_correction_factors
```

All values of the autocovariance function are pure real numbers, we can get rid of the "complex" representation resulting from the use of the FFT. Also we make sure that we return a signal of the same size as the input signal, on each dimension.

```
crop_slices = [slice(i) for i in binary_signal.shape]
return np.real(autocovariance[crop_slices])
```

Illustrations

In this section we take a few sample signals and try some of the above formulas to get a feeling of their uses and also to provide some code snippets !

> **Note**
>
> All graphical representation of the autocorrelation function will be displayed on a range that is half the length of the processed signal. This is related to the "rule of thumb" I explained in the section about Estimations, since I tend to consider that what's outside this range is quite unreliable.

```
if __name__=="__main__":
```

> **Warning**
>
> If you import this script the code below won't be executed, but if you execute this script directly, images will be automatically generated and saved on disk.

One dimensional case

Sample signals

All our signals are defined on the same domain (ie. for the same number of steps).

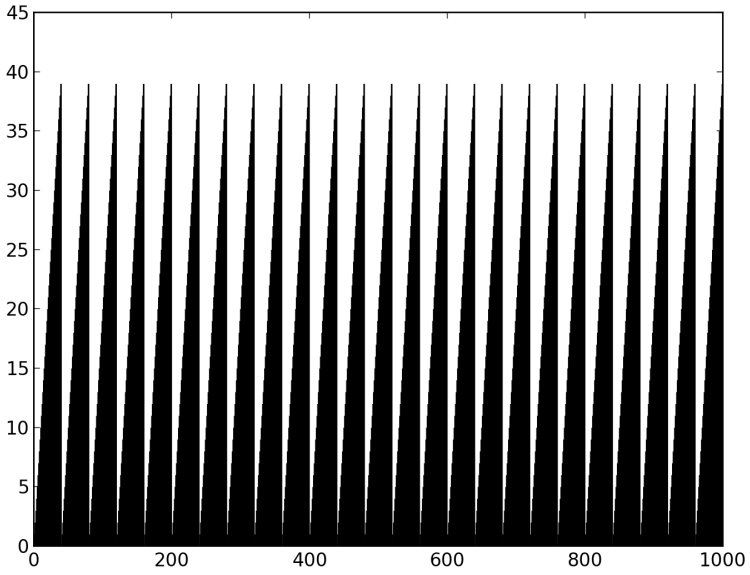
```
domain_length = 1000
domain = range(domain_length)
```

First let's define a periodic signal that will repeat every 40 steps.

```
signal = [ step%40 for step in domain]


pl.figure()
pl.vlines(domain, 0, signal, color="k", linewidth=1.)
```

```
pl.ylim(0, 45)
pl.savefig("signal.png", dpi=200)
```

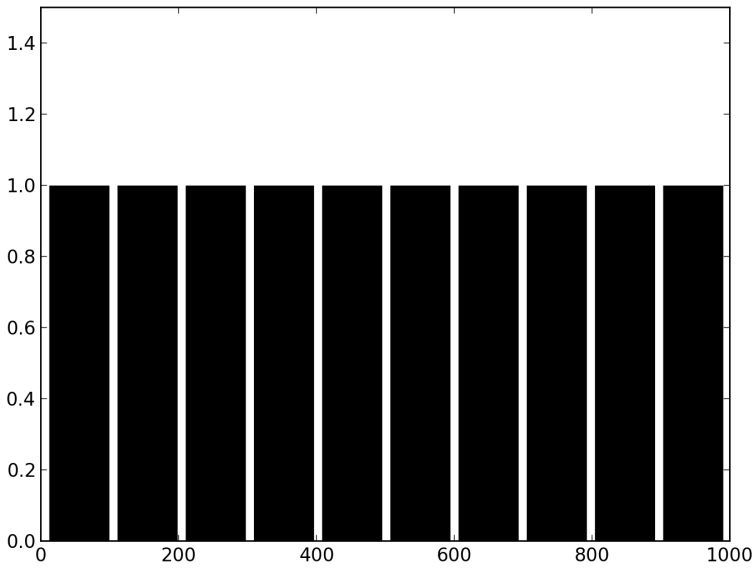
Fig.(signal.png): Cyclic signal repeating every 40 steps.



Our second signal will be a black (0) and white (1), it will repeat every 99 steps and show 86-step-long white segments separated by 13-step-long white segments.

```
signal_bin = []
for step in domain:
    step_mod99 = step%99
    if step_mod99 < 13:
        signal_bin.append(0)
    else:
        signal_bin.append(1)

pl.figure()
pl.vlines(domain, 0, signal_bin, color="k", linewidth=1.)
pl.ylim(0,1.5)
pl.savefig("signal_bin.png", dpi=200)
```

Fig.(signal_bin.png): Binary signal repeating every 99 steps.



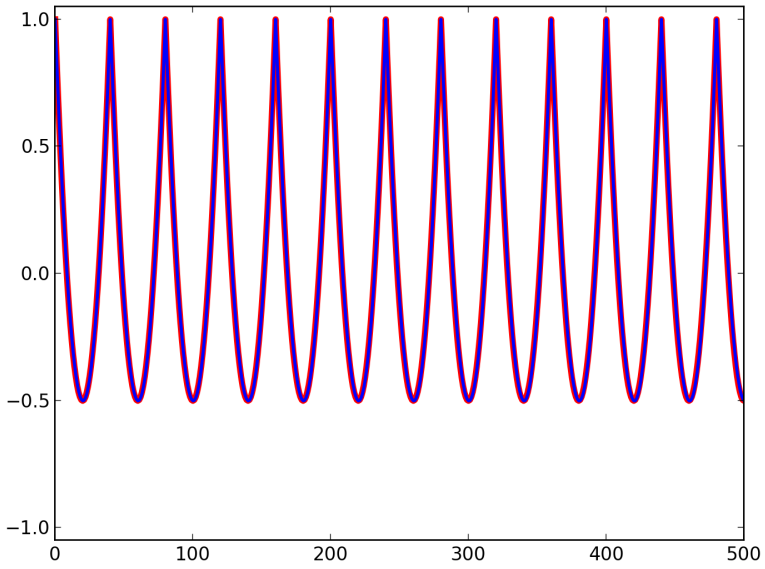
Comparing estimations

Let's have a look at the cyclic estimation and the so-called truncated one, side by side.

```
cyclic_autocorr = fftCyclicAutocorrelation1D(signal)

truncated_autocorr = fftAutocorrelation1D(signal)

pl.figure()
pl.plot(domain,cyclic_autocorr,color="red",linewidth=4)
pl.plot(domain,truncated_autocorr,color="blue",linewidth=2)
pl.ylim(-1.05, 1.05)
pl.xlim(0, domain_length/2)
pl.savefig("autocorr_comparison.png", dpi=200)
```

Fig.(autocorr_comparison.png): Comparing cyclic and "truncated" estimations of the autocorrelation.

In red (thick line) is the cyclic estimation, while the "truncated" one is in blue (thin line).



Conclusion

We can get approximately the same result wether we have to pad the signal or not. And both estimated functions make it visible that the signal repeats itself every 40 steps (this is the width between two peaks of the autocorrelation). However we will later visualize the Bias of the cyclic estimator .

Mask effect

The "truncated" estimation implies correcting a mask effect and this section illustrates how this effect apears on the autocovariance functions.

We can apply the same algorithm as the one in fftAutocovariance1D, but without the "mask effect correction" part:

Get a centered version of the signal and zero-pad it.

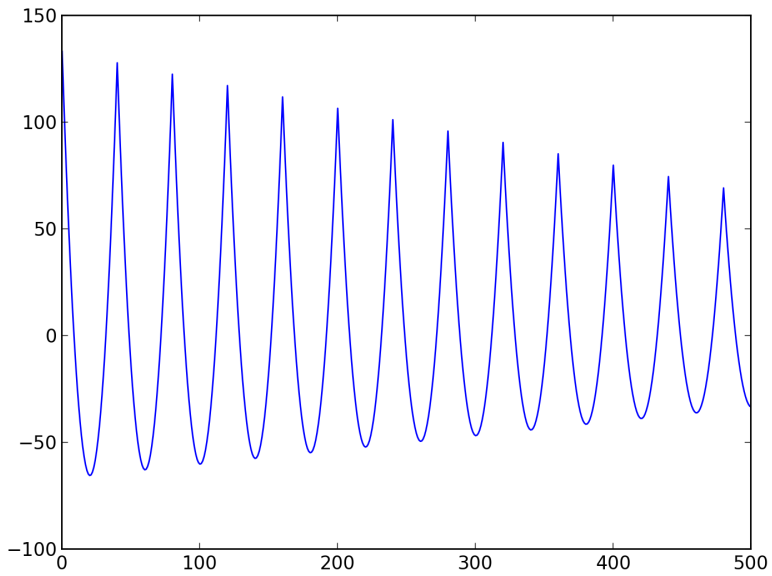
```
centered_signal = signal - np.mean(signal)
zero_padding = np.zeros_like(centered_signal)
padded_signal = np.concatenate(( centered_signal,
                                 zero_padding    ))
```

Compute its Fourier transform and get the pseudo covariance.

```
ft_signal = np.fft.fft(padded_signal)
pseudo_powerSpectralDensity = np.abs(ft_signal)**2
pseudo_autocovariance = np.fft.ifft(pseudo_powerSpectralDensity) / len(signal)
```

```
pl.figure()
pl.plot(domain,pseudo_autocovariance[0:len(domain)])
pl.xlim(0, domain_length/2)
pl.savefig("erroneous_autocov_signal.png", dpi=200)
```

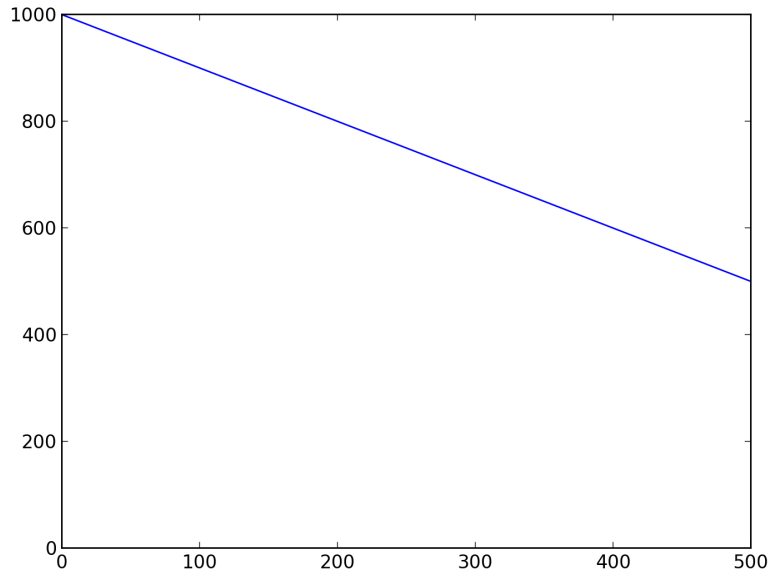
Fig.(erroneous_autocov_signal.png): Erroneous estimation of the autocovariance with the FFT on a padded signal.



We can now visualize the mask effect.

```
mask = np.concatenate(( np.ones_like(centered_signal),
                        zero_padding                  ))
ft_mask = np.fft.fft(mask)
mask_correction_factors = np.fft.ifft(np.abs(ft_mask)**2)


pl.figure()
pl.plot(domain,mask_correction_factors[0:len(domain)])
pl.xlim(0, domain_length/2)
pl.savefig("autocov_mask.png", dpi=200)
```

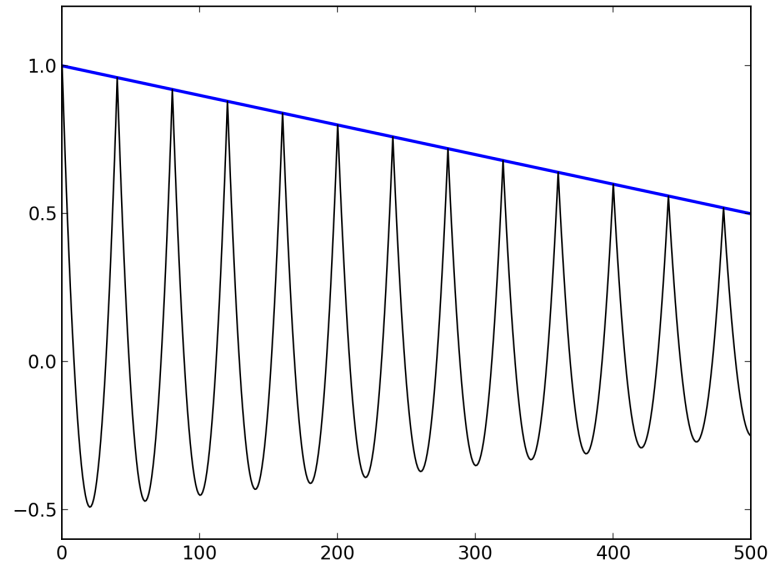
Fig.(autocov_mask.png): Mask correction factors.



And by normalizing both curves between -1 and 1, we can see how they relate to eachother and how the second curve can actually correct the errors made in the first one.

```
pl.figure()
pl.plot(domain,mask_correction_factors[0:len(domain)]/mask_correction_factors[0],
        color="b",linewidth=2)
pl.plot(domain,pseudo_autocovariance[0:len(domain)]/pseudo_autocovariance[0],
        color="k")
pl.xlim(0, domain_length/2)
pl.savefig("autocov_mask_hull.png", dpi=200)
```

Fig.(autocov_mask_hull.png): Relation between the mask effect and the hull of the "erroneous autocovariance".



Bias of the cyclic estimator

If we perform the cyclic and "truncated" measures on a signal whose period is not a divider of the domain's size, we can point at a big difference in the behaviours of both kind of evaluation that seemed to give the same result in the first example.

First we compute the autocorrelation with a cyclic estimator.

```
cyclic_autocorr_signal_bin = fftCyclicAutocorrelation1D(signal_bin)
```

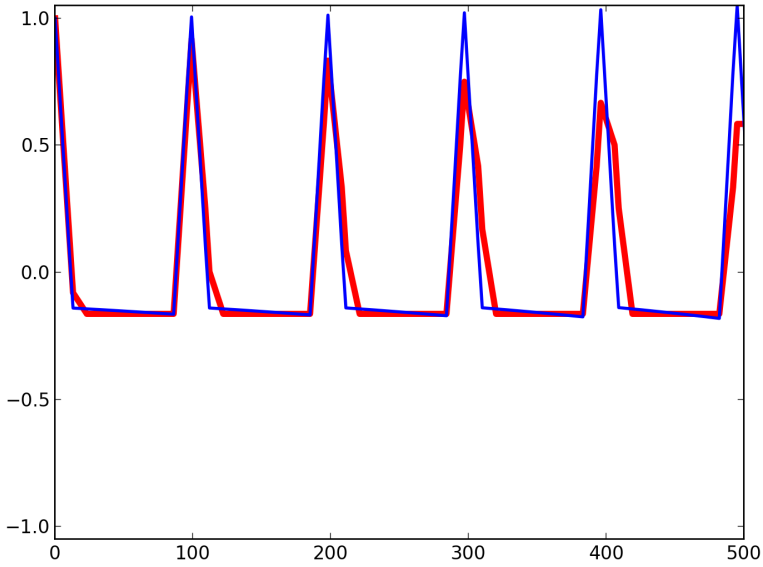
Then the "truncated" estimation is performed on a zero-padded version of the signal.

```
truncated_autocorr_signal_bin = fftAutocorrelation1D(signal_bin)
```

```
pl.figure()
pl.plot(domain,cyclic_autocorr_signal_bin, color="r", linewidth=4)
pl.plot(domain,truncated_autocorr_signal_bin, color="b", linewidth=2)
pl.ylim(-1.05, 1.05)
pl.xlim(0, domain_length/2)
pl.savefig("bias_of_cyclic_estimation.png", dpi=200)
```

Fig.(bias_of_cyclic_estimation.png): Comparison of the cyclic and truncated estimations for a signal whose period is not in sync with the implicit periodisation of the FFT.

In red (thick line) is the cyclic estimation while the "truncated" one is in blue (thin line).



There is an easy way to explain those differences by considering that the autocorrelation functions are actually *correct* but are not computed on the same functions. An idea that we will try to briefly explain in the following paragraphs.

As stated earlier the cyclic estimation computes the autocorrelation as if the signal was repeating itself with a maximal period of the size of the initial domain, and thus considers that if we were looking at the signal on a wider domain, we could define it as follows.

```
signal_bin_cyclic = []
for step in domain:
    step_mod99 = step%99
    if step_mod99 < 13:
        signal_bin_cyclic.append(0)
    else:
        signal_bin_cyclic.append(1)
# repeat the signal now
for step in domain:
    step_mod99 = step%99
    if step_mod99 < 13:
        signal_bin_cyclic.append(0)
    else:
        signal_bin_cyclic.append(1)
```

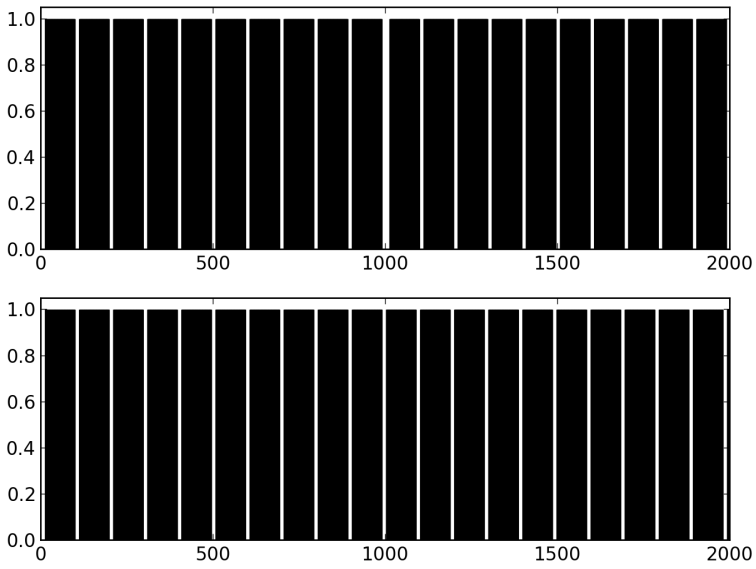
On the contrary the "truncated" estimation assumes that we are looking through a "window" at a signal that will keep the same structure "outside" the window and won't present any new pattern at larger scales. So that, if looked on a wide domain it could be described as the following.

```
domain_pad = range(0,2*domain_length)
signal_bin_stationary = []
for step in domain_pad:
    step_mod99 = step%99
    if step_mod99 < 13:
        signal_bin_stationary.append(0)
    else:
        signal_bin_stationary.append(1)
```

Now if we compare those two signals we can see that the difference is actually quite small, but big enough for the autocorrelation fuctions to be diffrent.

```
pl.figure()
pl.subplot(2,1,1)
pl.vlines(domain_pad, 0, signal_bin_cyclic, color="k", linewidth=1.)
pl.ylim(0,1.05)
pl.subplot(2,1,2)
pl.vlines(domain_pad, 0, signal_bin_stationary, color="k", linewidth=1.)
pl.ylim(0,1.05)
pl.savefig("signal_bin_cyclic.png", dpi=200)
```

Fig.(signal_bin_cyclic.png): The two implicit representations of the same signal according to both the cyclic (top) and "truncated" (bottom) estimators.



Conclusion

The cyclic and "truncated" estimations do not give the same results as soon as the studied signal is not synchronised with the implicit periodisation performed by the cyclic estimator. The later introduces a high scale periodical pattern that influences the correlation function even at much lower scales, and should then be used with care.

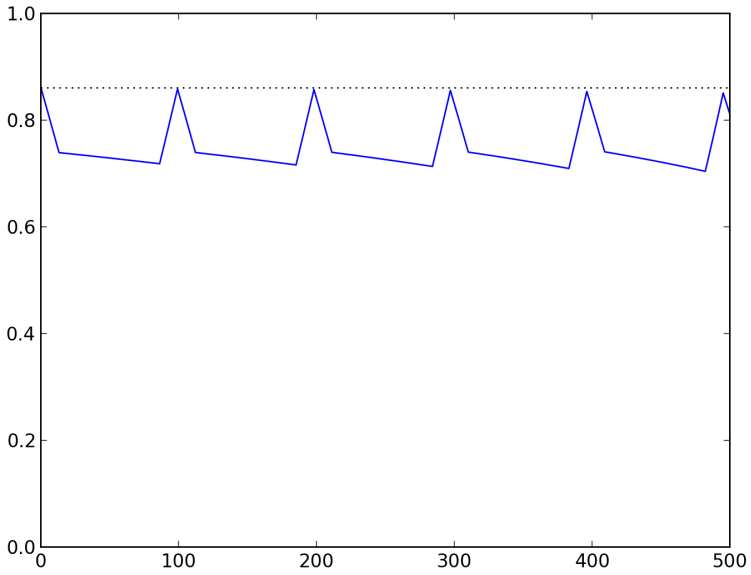
Autocovariance and centering for binary signals

We've mentioned earlier that there was a specific definition of the autocovariance that could be used for binary signals, let's visualize this measure for the binary signal we've just studied.

```
set_autocovariance = fftSetAutocovariance1D(signal_bin)

pl.figure()
# Display a horizontal line "y=mean(signal_bin)"
pl.plot(domain,[np.mean(signal_bin)]*domain_length,"k:")
# Display the autocovariance
pl.plot(set_autocovariance, color="b")
```

```
pl.ylim(0,1.)
pl.xlim(0, domain_length/2)
pl.savefig("set_autocov.png", dpi=200)
```

Fig.(set_autocov.png): "DSP-like" definition of the autocovariance for a binary signal.



Conclusion

While this looks very much like a diggression, I felt that it might be interesting to mention this specific definition of the autocovariance for binary signals, especially because its FFT based estimation can be performed by a small modification to the previous procedures.

Important

This definition of the autocovariance, is not suitable for computing the **autocorrelation** function. For the autocorrelation, the **centering must be done** anyway as shown in previous sections.

Two dimensional case

We will now see a few examples of the same measures on a 2d image.

Generating a random image

To change a little and stop looking at periodic signal we generate a slightly less boring (though quite common) boolean model of rectangles.

We start with a blank image.

```
img_width = 512
img_height = 512
img = np.zeros((img_width, img_height))
```

Then we draw the actual number of rectangles following a Poisson law.

```
num_rectangles = np.random.poisson(20,1)
```

Our rectangles will have a fixed size of 100 (along x) by 60 (along y) pixels.

```
rect_width  = 100
rect_height = 60
```

We will then draw those (filled) rectangle at random places over the image.

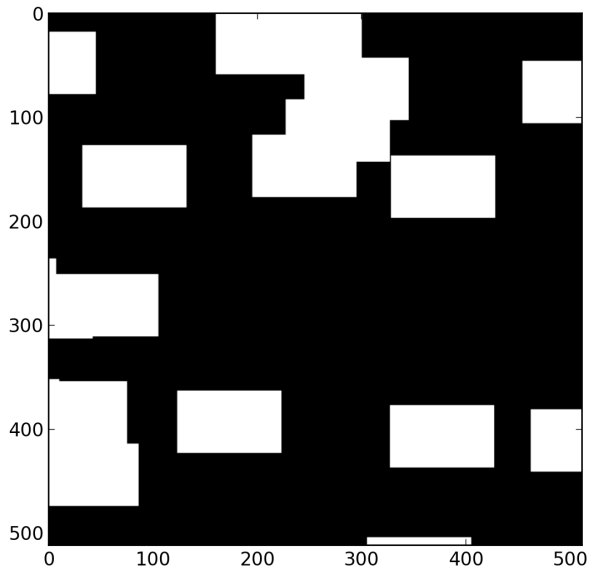
To do so we first determine the x and y coordinates where the centers of every rectangle should be. Those coordinates must be so that each rectangle will (at least) intersect the image's domain even if its center is outside the image.

```
x_coords = np.random.randint( -rect_width/2,
                              img_width + rect_width/2,
                              num_rectangles )

y_coords = np.random.randint( -rect_height/2,
                              img_height + rect_height/2,
                              num_rectangles )
```

Then, for each rectangle we compute the actual intersection of the rectangle with the image and fill the correspoding region with ones (ie we draw them in white).

```
for x,y in zip(x_coords,y_coords):
    rect_drawing_area_x = slice( max(         0, x-rect_width/2 ),
                                 min( img_width, x+rect_width/2 ) )
    rect_drawing_area_y = slice( max(         0, y-rect_height/2 ),
                                 min( img_height, y+rect_height/2 ) )
    img[(rect_drawing_area_x,rect_drawing_area_y)] = 1.


pl.figure()
pl.imshow(255.*img.T,cmap=pl.cm.gray)
pl.savefig("boolean_model_of_rectangles.png", dpi=200)
```

Fig.(boolean_model_of_rectangles.png): Boolean model of rectangles.



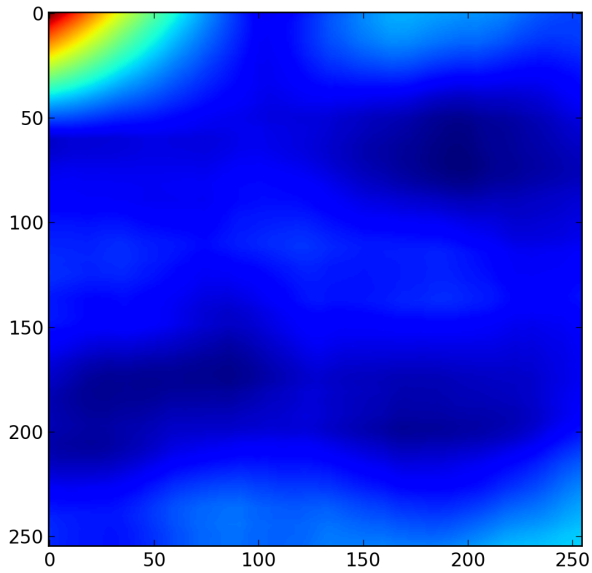
Comparing estimations

We can now compare the two implemented estimators for the autocorrelation of such an image.

The cyclic estimation first.

```
cyclic_autocorr_img    = fftCyclicAutocorrelation(img)

pl.figure()
pl.imshow(cyclic_autocorr_img[0:255,0:255].T,cmap=pl.cm.jet)
pl.savefig("cyclic_autocorr_img.png", dpi=200)
```

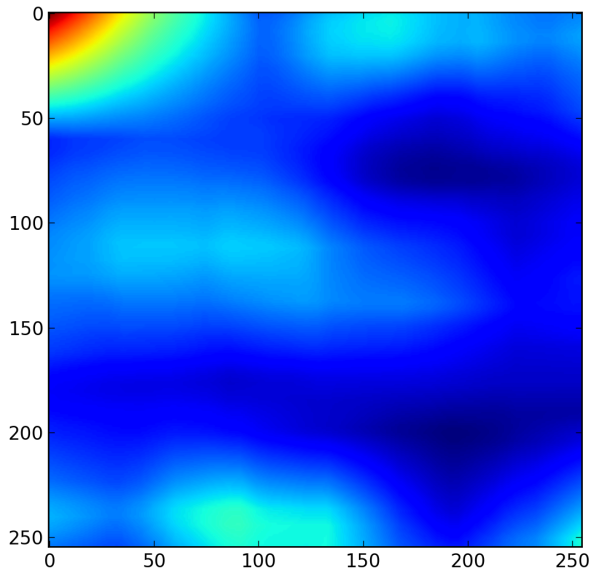
Fig.(cyclic_autocorr_img.png): Cyclic estimation of a 2D autocorrelation function.



And then the "truncated" estimation.

```
truncated_autocorr_img = fftAutocorrelation(img)


pl.figure()
pl.imshow(truncated_autocorr_img[0:255,0:255].T,cmap=pl.cm.jet)
pl.savefig("truncated_autocorr_img.png", dpi=200)
```

Fig.(truncated_autocorr_img.png): "Truncated" estimation of a 2D autocorrelation function.

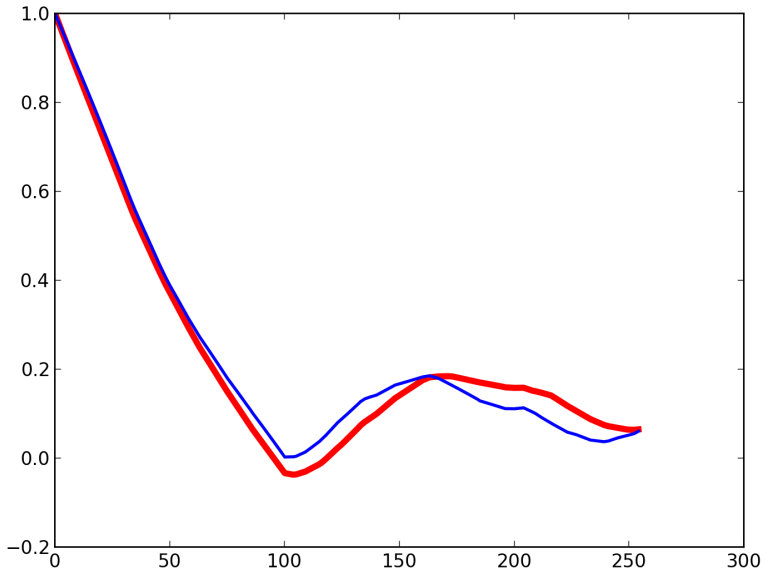


And when we take horizontal profiles (ie. along the x-axis) of the 2D autocorrelations, we get the following graph.

```
cyclic_autocorr_profile    = cyclic_autocorr_img[0:255,0]
truncated_autocorr_profile = truncated_autocorr_img[0:255,0]

pl.figure()
pl.plot(range(0,255), cyclic_autocorr_profile, color="r", linewidth=4)
pl.plot(range(0,255), truncated_autocorr_profile, color="b", linewidth=2)
pl.savefig("autocorr_img_profile_x.png", dpi=200)
```

Fig.(autocorr_img_profile_x.png): Profile of the autocorrelation functions along the x-axis.

In red (thick line) the cyclic estimation and in blue (thiner line) the "truncated" one.



Conclusion

As in the 1d case, both estimations are relatively close but show clear differences anyway. In particular, they seem to be very similar when close to the origin of the graph and the further they are from the origin the more different they look.

However, when we look at the horizontal profile, both estimations seem to bend at the exact same places, especially above the length 100 pixels which corresponds to the horizontal length of the white rectangles used to generate the image. Of course they also bend elsewhere due to the overall structure of the generated image, but the bend at 100 pixels is one of the most significant feature of the curve as well as the most reliable (it will remain even if we "draw" another realisation of this random image with the same algorithm)

"Set" autocovariance for binary images

Eventually, we can also look at the "set" autocovariance function for this binary image, as well as its profile along the x-axis.

```
set_autocovariance_img = fftSetAutocovariance(img)

pl.figure()
pl.imshow(set_autocovariance_img[0:255,0:255].T,cmap=pl.cm.jet)
pl.savefig("set_autocov_img.png", dpi=200)

pl.figure()
pl.plot(range(0,255), set_autocovariance_img[0:255,0])
# Display a horizontal line "y=mean(img)"
pl.plot(range(0,255),[np.mean(img)]*255,"k:")
pl.savefig("set_autocov_img_profile_x.png", dpi=200)
```

Fig.(set_autocov_img.png): "Set" autocovariance of the boolean model.

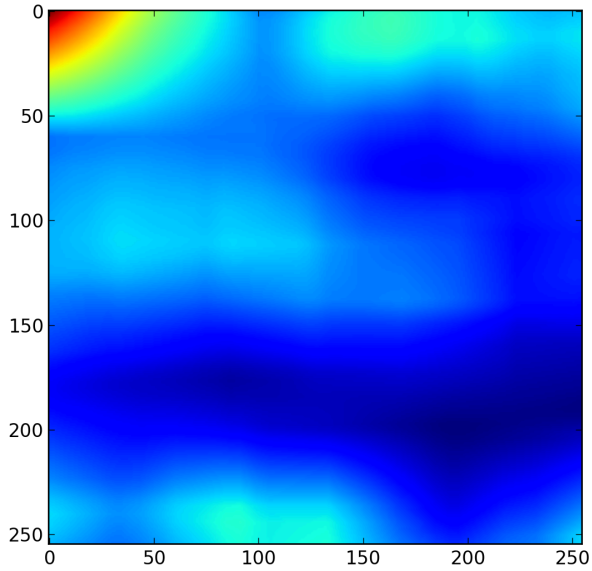
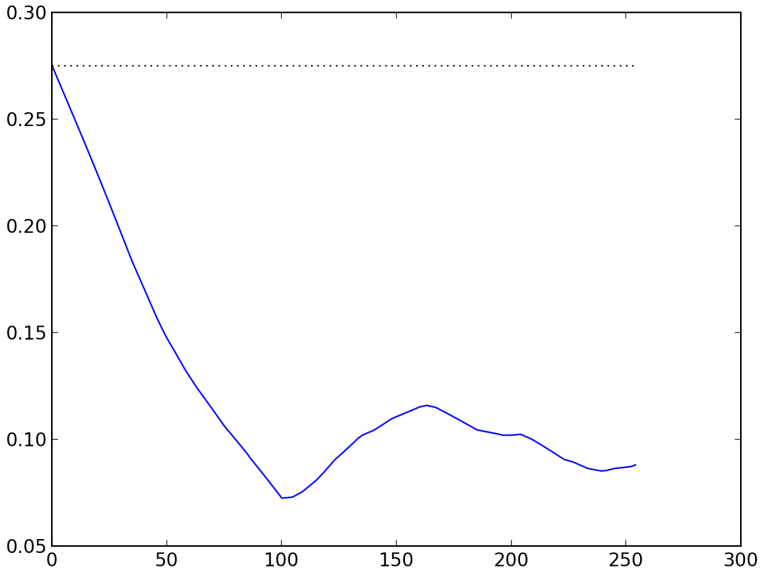


Fig.(set_autocov_img_profile_x.png): Profile of the "set" autocovariance, along the x-axis.



Conclusion

As in the 1d case, this measure is closely related to statistical features of the image: the value of the picture at length 0 pixel corresponds to the density of white pixels in the image, and on the horizontal profile, the bend at 100 pixels is still visible (just like for the previous two autocorrelation estimators). There is however much more to say about this measure, but this is clearly out of scopre of the current document.

References

[WikiPrsCf]"Pearson product-moment correlation coefficient" *in Wikipedia*; http://en.wikipedia.org/wiki/Pearson_coefficient

[SmithDSPG]*(1, 2)* Steve W. Smith, "Properties of Convolution" *in The Scientist and Engineer's Guide to Digital Signal Processing*; http://www.dspguide.com/ch7/3.htm

[Weisst.XC]Eric W. Weisstein, "Cross-Correlation" in *MathWorld A Wolfram Web Resource*; http://mathworld.wolfram.com/Cross-Correlation.html

[WikiXCorr]"Cross-correlation" *in Wikipedia*; http://en.wikipedia.org/wiki/Cross-correlation

[WikiAlVar] "Algorithms for calculating variance" *in* *Wikipedia*; http://en.wikipedia.org/wiki/Algorithms_for_calculating_variance
[WikiACorr] "Autocorrelation" *in* *Wikipedia*; http://en.wikipedia.org/wiki/Autocorrelation
[WikiStoch] "Stochastic process" *in* *Wikipedia*; http://en.wikipedia.org/wiki/Stochastic_process
[WikiSpecD] "Power Spectracl Density" *in* *Wikipedia*; http://en.wikipedia.org/wiki/Spectral_density

---