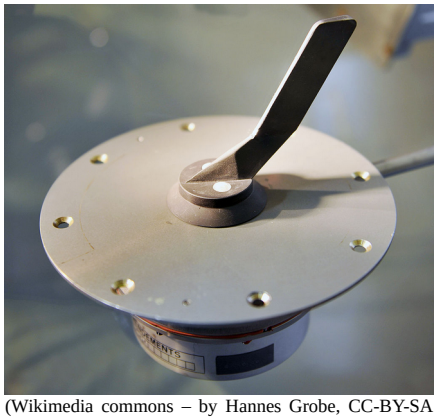
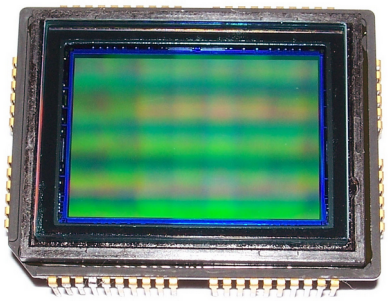
Sensors, Signals, and Noise

Sensors are a fundamental component of automated systems, they provide information about the environment and allow the software to determine appropriate actions to achieve its operational goals. Sensors are imperfect, noisy, and sometimes unreliable, so in order to build a reliable system we must learn how sensors work, and how to deal with noise, sensor failures, and other problems that may arise while a system is running.

What is a sensor?

- It's a ***device*** (doesn't have to be electronic! It can be mechanical, chemical, or something else) whose purpose is ***to measure a specific physical quantity***.
- The measured quantity is converted in a ***signal*** that can be ***read or interpreted*** by an ***observer***.

 (Wikimedia commons – by Brian Snelson, CC-2.0)	 (Wikimedia commons – by Hannes Grobe, CC-BY-SA 3.0)	 (Wikimedia commons – by Andrzej_w_k_2 , CC-BY-4.0)
A speedometer which converts the speed of a moving vehicle (physical quantity) to a signal (the red indicator line) that can be read by an observer (you or I)	An angle-of-attack sensor, measuring the angle of a plane with regard to the wind. The resulting (electronic) signal is read by a computer (observer)	A CCD photosensor used in a digital camera. Each photodetector measures incoming light. The resulting signal is read by a processing element and converted into an image

There are many, many types of sensors, developed for specific applications. Examples include

* Acoustic sensors for measuring sound (e.g. microphones)
* Pressure (e.g. a scale to measure wight, an atmospheric pressure sensor)
* Distance (e.g. ultrasonic, laser rangefinders, radar)
* Temperature (e.g. infrared thermometers)
* Acceleration (e.g. the accelerometers in your phone to detect/track motion)
* Rotation (e.g. gyroscopes, also in your phone)
* Light (e.g. photosensors in cameras, photo detectors in automated light switches)

* Voltage, Current, and/or Power (e.g. the battery level sensor in any electronic device)
* Chemical quantities (e.g. acidity (ph), salinity, concentration of specific substances)

These are just a few examples. Chances are if you can think of a physical quantity there will be a type of sensor designed to measure this quantity.

Properties that a good sensor should have

- It must be sensitive to the quantity being measured. For instance, we may try using a GPS to measure the speed of a car, but the measurements we get from this will not be as accurate as those from a speedometer – the GPS is not sensitive to velocity, it is only sensitive to position.

- It must not be sensitive to other physical properties encountered during use: For example, an infrared thermometer should not be too sensitive to the amount of visible light in the room where it's being used.

- It should not change the value of the physical quantity being measured by a significant amount. All measurements will affect the quantity being measured, but the effect should be negligible otherwise the sensor is useless (because right after we measure it, the quantity has changed)

For practical purposes, we will be concerned with sensors that produce an electric signal which is then read by a computer to provide an automated system with information about the environment. Let's see how we would model and understand such a sensor.

Sensor model

In an ideal world, a sensor's ***response*** (the output or signal it produces as a result of the quantity it is measuring) should be a ***simple function*** of the quantity being measured:

$$r(x) = f(x)$$

Here ***x*** is a physical quantity, ***r(x)*** is the sensor's response to this quantity, and ***f(x)*** is a simple function such as a ***linear*** or ***logarithmic*** function. That the response should correspond to a simple function is important since we ideally want to recover the value of ***x*** from the measured ***r(x)***.

Perhaps the simplest model for sensors that we can use in practice is called the ***affine model***.

$$r(x) = ax + b,$$

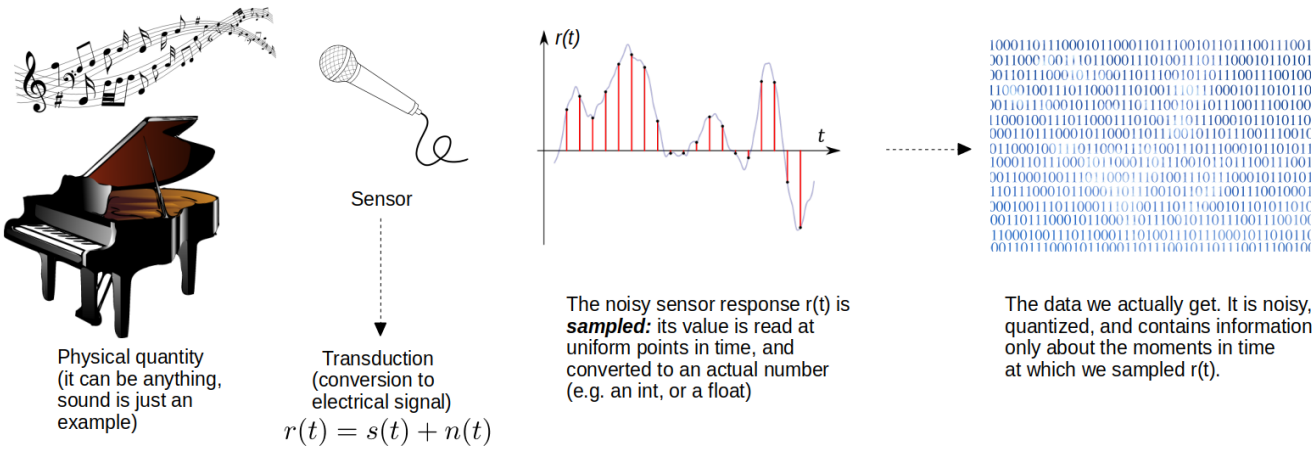
where ***a*** and ***b*** are suitable constants. This corresponds to a linear response function and allows us to recover the value of ***x*** as

$$x = \frac{r(x) - b}{a}$$

However, life is a little bit more interesting than that. The process of converting a physical quantity to an actual value available to software running on our system is much more complex than the response function above.

Converting physical quantities to data – the sampling process

The information our system eventually has access to is not a direct measurement of the physical quantity we are interested in. The actual process of converting a physical quantity into values inside a computer involves multiple steps, and each step inevitably introduces noise and distortion. An overview of the process is shown below



Sensing process – Step 1: Transduction

The physical quantity (sound in the example above) is sensed by an appropriate sensor (in this case a microphone). The sensor has to convert that physical quantity to an electrical signal (voltage or current) that can be processed by a computer. This process is called ***transduction***.

The process is noisy and imperfect. The response of the sensor will ***not*** be an exact function of the input signal. Sources of noise include:

- Limitations in the sensor – for weak signal values the sensor may nor respond at all. For very strong signal values the sensor ***saturates,*** that is, it reaches its maximum output value and any further increase in ***s(t)*** produces no change in ***r(t)***. You can see this in photography, where bright areas in the picture are completely white due to sensor saturation.

- Sensor non-linearity – No physical sensor has a perfect linear response.

- Ambient noise – The signal being measured exists in some environment that contains other unrelated signals (e.g. consider someone talking on the phone along a busy street – the microphone picks up noise from the street).

- Electrical, thermal, and RF noise – electrical circuits are affected by electromagnetic radiation, heat, and other electrical components within the sensor's casing.

The result of this is that our sensor response ***r(t)*** is the result of a possibly distorted function of ***s(t)***, and a noise component ***n(t)*** that is unpredictable, unavoidable, and possibly large.

Sensing process – Step 2: Sampling

The sensor response r(t) is not usable by a computer system – it is continuous, and real valued. In order for us to have a quantity that we can represent, store, and manipulate inside a compute we need to sample the sensor response.

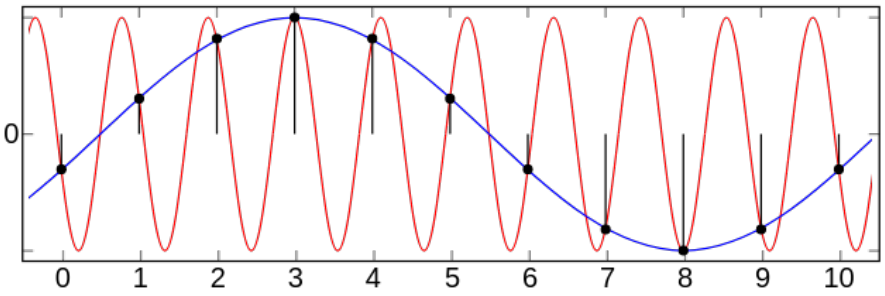
Sampling involves:

- Taking the value of the response r(t) at uniformly spaced intervals. For example, sound recordings typically use a sampling frequency of 44,100 Hz. This means that a microphone's output would be read 44,100 times for each second of sound recording. Or, what is the same – the response r(t) is read each 1/44,100 seconds. Another example, in a camera's photo-sensor, we have a regular grid of photo-sensors, and we can only capture information at these sites and nowhere else.

- The resulting sequence of sampled values s(k) represents our original signal and is no longer continuous, so we can work with it. However, for the values in this sequence to be meaningful, we have to be careful about how often we sample the original r(t).

The sampling process, by definition, does not preserve the original continous signal. If the sampling is not done carefully, information is lost.

The choice of interval for reading the sensor response (called the sampling period) has to be small enough to allow us to capture the fastest changes that can occur in the signal we are measuring. Otherwise, the resulting values will not allows us to determine what the original signal looked like. This is called aliasing and is illustrated below:



(Source Wikimedia Commons, Jarvisa, Moxyfire, CC-SA 3.0)

The original signal was sampled at points too far apart from each other, when we try to use these samples to determine what the original signal looked like, we obtain the blue curve – clearly something is not right!

A more complex signal will be distorted in non-trivial ways, the point is, the process of converting a continuous time signal into a sequence of samples is not trivial and has to be done with care.

Aside: Understanding sampling and aliasing – DFT and Frequency Analysis

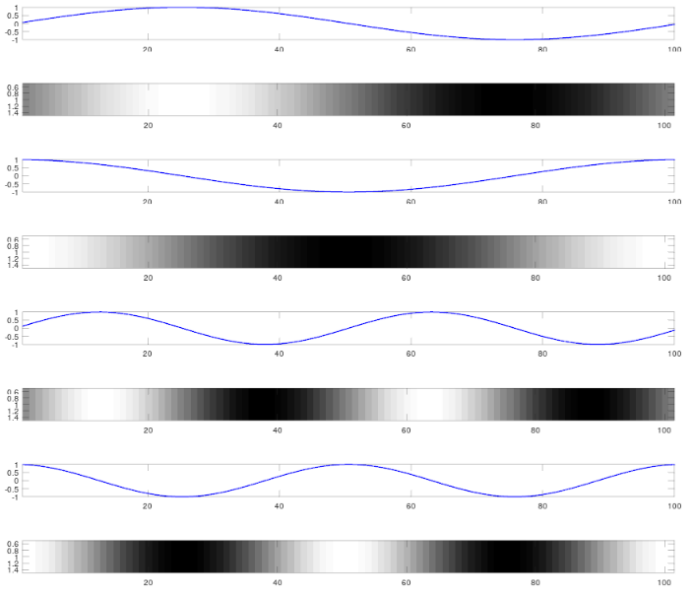
Linear Algebra provides us with an incredibly powerful tool for analyzing and understanding signals: The Discrete Fourier Transform (DFT). The key ideas you need to remember are:

- A sampled signal s(k) is just a sequence of values of some particular length n. Such a sequence of values can be stored in a vector of length n.

- A vector of length n can be fully expressed using any orthonormal basis with n vectors.

- The DFT uses an orthonormal basis composed of basis vectors that represent sine and cosine waves with different frequencies.

The figure below illustrates how we build an orthonormal basis using sine and cosine waves for a signal with 100 samples (i.e. n=100 in this particular example – so we will need 100 orthonormal basis vectors, each with length 100).



Lowest-frequency ***sine*** wave with the same length as our signal (blue) – this sine wave completes 1 cycle within the length of our input signal. The corresponding basis vector is obtained by sampling this sine wave into a vector with 100 entries.

Lowest-frequency ***cosine*** wave with the same length as our signal (blue), the corresponding basis vector is obtained by sampling this cosine wave into a vector of 100 entries.

Second lowest-frequency ***sine*** wave with the same length as our signal, this wave completes ***two cycles*** within the length of our input signal. The corresponding basis vector is obtained by sampling this sine wave into a vector of 100 entries.

Second lowest-frequency ***cosine*** wave with the same length as our signal. We obtain a basis vector from this cosine wave by sampling from it.

We obtain further basis vectors by sampling sine and cosine waves of increasing frequencies (3 cycles per signal length, then 4, then 5, and so on). Since we require n basis vectors, and each frequency provides 2 of them (one sine, one cosine), our orthonormal basis will contain sine and

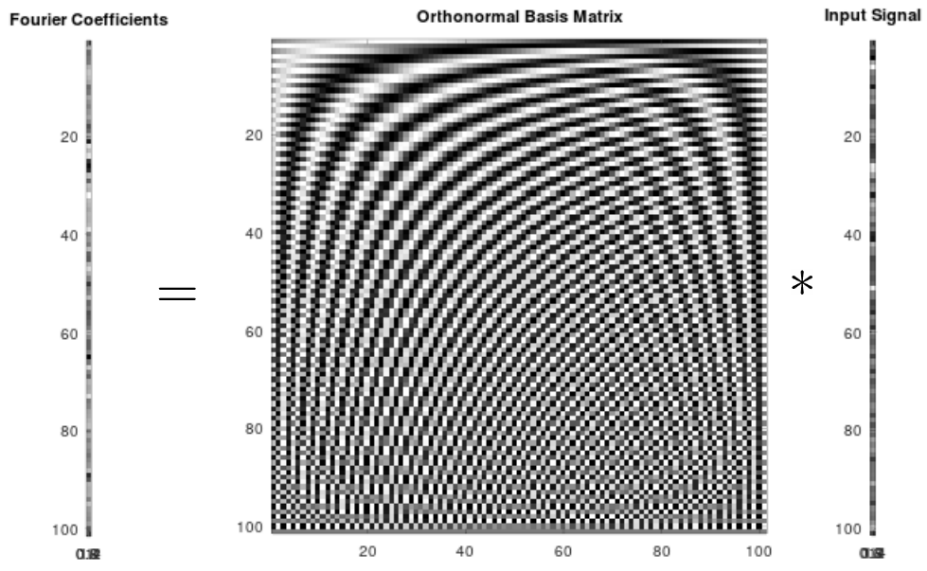
cosine waves with frequencies from 1 cycle per signal length to n/2 cycles per signal length. The resulting vectors must be normalized to unit length.

We note that sine and cosine waves with frequencies that are integer multiples of each other are mutually orthogonal. The dot product of any two of our basis vectors is zero.

The full set of orthonormal vectors can be stored in an n x n matrix with one basis vector per row. The DFT of our input signal s(k) is obtained by multiplying the vector containing our signal with the matrix containing our orthonormal basis

$$\vec{f} = F \cdot \vec{s}$$

You should recognize this operation as a projection of s(k) onto each of the vectors in our ourthonormal basis. This is illustrated in the figure below.



The entries in vector $\vec{f}$ are the coeefficients of the DFT for our signal, they correspond to the amount of each sine or cosine wave that is present in our input signal s(k). That is, f(1) tells us how much of a sine wave with 1-cycle per signal length is present in our input, f(2) tells us the same for the cosine wabe with 1-cycle per signal, and so on.

By adding up each of the sine and cosine waves in our orthonormal basis in the amounts given by the coefficients in $\vec{f}$ we can reconstruct exactly the input signal s(k).

Back to understanding our signal - Here's what you should be thinking of:

- We can analyze any signal as a sum of sine and cosine waves of different frequencies. Each input signal has a different frequency content – that is, the amounts in which each sine/cosine wave contributes to the input signal is different – each different sine/cosine wave is a frequency component.

For any signal we care to analyze, there is a finite number of frequency components that can contribute to the overall signal.

- In order to properly represent a signal, the Nyquist-Shannon sampling theorem states our sampling rate must be at least twice the frequency of the highest frequency component that is present in our original signal. Sampling at a rate that is slower than this will unavoidably cause aliasing and loss of information.

Importantly – the discussion above is intended to explain how the DFT works, and how it captures frequency information. But you would not actually compute one with a matrix multiplication with sine/cosine vectors as shown above. The standard way to compute the DFT uses complex sinusoidal waves taking advantage of the identity

$$e^{ix} = cos(x) + isin(x)$$

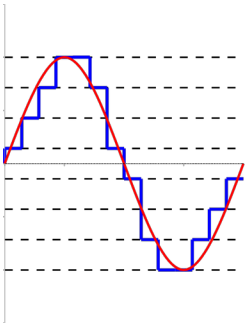
and you would use the Fast Fourier Transform (https://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/2001/pub/www/notes/fourier/fourier.pdf) algorithm to compute the coefficients for these complex sinusoidal functions efficiently.

Sensing process – Step 3: Discretization

The last part of the process, converting the sampled values of ***r(t)*** into values within a useful range also introduces distortion. Firstly, any values outside the allowed range will be clipped – that is, mapped towards the minimum, or maximum allowed values within the range of the sensor.

- The value of ***r(t)*** at each of these time instants must be discretized (converted into a number within a reasonable, pre-defined range). For instance, image brightness for digital pictures uses values in [0, 255] to represent different intensities from black to white.

Secondly, the resulting values are quantized – both integers and floating point values within a computer have limited, finite precision. That effectively means there is a fixed set of possible output values for the sampling process, and the real-valued ***r(t)*** has to be mapped to one of these. This is illustrated below:



(Wikimedia commons, Hyacinth, CC-SA 4.0)

The signal (red) is mapped to one of the allowed values (represented by dotted lines) – the resulting data is shown as a blue plot. You may think this is not a big deal, the range of values for the output is typically much larger than the example above shows. But the point is – the process itself introduces non-trivial, unavoidable, and hard-to-eliminate distortions that corrupt the information we get from the input signal ***s(k)***.

Dealing with sensor noise

From the discussion above, it should be clear to you that you can not simply take a sensor's reported value and take it as a true representation of the state of the world. You have to account for the noise present in the measurements available to you.

Of course not every application will require the same amount of care when handling sensor data, but you must consider for your specific application:

- What are the characteristics of the sensors you're working with

- How noisy are their measurements, and how much impact can the noise have on the system's

ability to perform its function properly

Based on the above, you may need to implement and use a suitable method for reducing sensor noise and for obtaining more reliable measurements.

Completely removing noise from a sampled signal is not possible

We saw above that noise is introduced at multiple points during the process of sampling a signal. If we bundle all of these into a single noise term we find that

$$s(k) = r(k) + n(k)$$

so our final, sampled signal value ***s(k)*** is the result of taking the correct (undistorted, noise free) signal value ***r(k)*** and adding a noise term ***n(k)*** that accounts for all the noise sources present in the signal acquisition process.

For each sampled value of ***s(k)*** we have two unknowns. There is no way to recover ***r(k)*** given the available information. So, noise removal, or more accurately ***noise reduction*** will require us to make some ***reasonable assumptions*** (of course!) about what ***r(k)*** may or may not be.

Reasonable assumptions:

- Noise values are decorrelated, that is, the noise component for ***s(j)*** does not depend in any way on (and can't be predicted from) the noise component for ***s(k)*** for $j \neq k$.

- Noise is zero-mean, that is, if we were able to take an average of all the noise values for the samples in our signal, the average of these values will approach zero.

These assumptions are often not exactly true in reality – most real sensors introduce some amount of structured noise (not de-correlated) or have a non-zero-mean noise. However, in practice, the assumptions above will allow us to do a reasonably good job of reducing noise.

Do keep in mind: If you have information about the noise characteristics of your sensor (which you could obtain, for example, by calibration: measuring known signals to carefully build a profile of the noise behaviour of your sensor) then you should use this information to build a more accurate de-noising process for your signals.

Noise reduction from averaging multiple readings

A first, simple approach for senoising relies on using multiple measurements. This is possible if

- We have multiple, redundant sensors reading the same quantity. For instance, multiple microphones located in close proximity and recording the same sound source. Redundant sensors are also common in high-reliability, fault tolerant systems.

or

- We have a sensor whose read-rate is relatively fast with respect to the speed at which the signal we are measuring changes. This allows us to assume the signal value ***r(k)*** is fixed during the interval required to take multiple readings. For example, a thermostat can easily take hundreds of readings within a very short time span, during this time the temperature in a room will not change measurably.

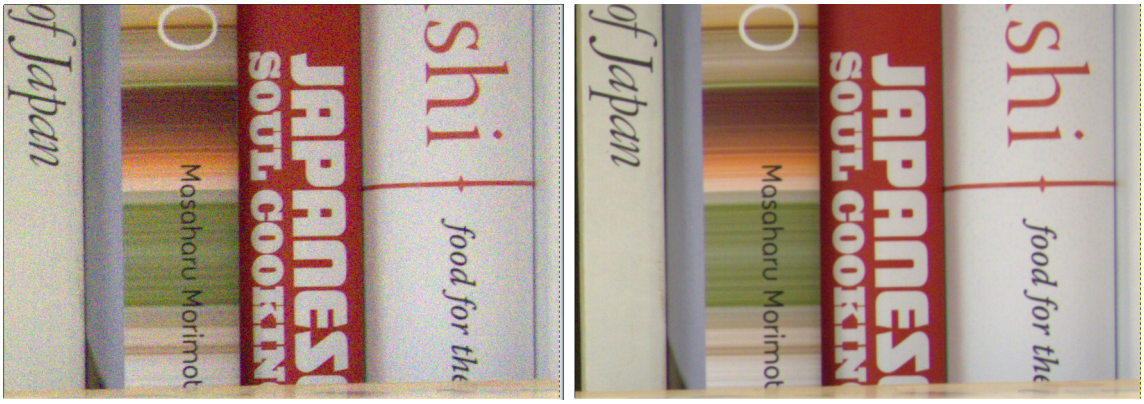
If either of the conditions above exists, we can obtain a denoised estimate of the signal simply by averaging the multiple measurements we have available.

This works because

$$
\begin{aligned}
s(k) &= r(k) + n(k) \\
\tfrac{1}{N}\textstyle\sum_{k=1}^{N} s(k) &= \tfrac{1}{N}\textstyle\sum_{k=1}^{N}(r(k) + n(k)) \\
\tfrac{1}{N}\textstyle\sum_{k=1}^{N} s(k) &= \tfrac{1}{N}\textstyle\sum_{k=1}^{N} r(k) + \tfrac{1}{N}\textstyle\sum_{k=1}^{N} n(k) \\
\tfrac{1}{N}\textstyle\sum_{k=1}^{N} s(k) &= \tfrac{1}{N} N * r(k) + \epsilon \\
\tfrac{1}{N}\textstyle\sum_{k=1}^{N} s(k) &\approx r(k)
\end{aligned}
$$

where ϵ approaches zero as we average more readings (recall that noise is zero-mean so it averages out).

The figure below shows an example of this process in the context of photography, where individual shots are noisy, but averaging over many shots of the same scene produces a clean result.



(Left: Original noisy image from a single shot. Right: average of 20 frames)

Denoising by averaging multiple measurements is very effective when either of the conditions mentioned above holds. However, it is often the case that we don't have redundant sensors, or it is not possible to sample the signal fast enough to assume that it hasn't changed during the time it takes to record multiple readings.

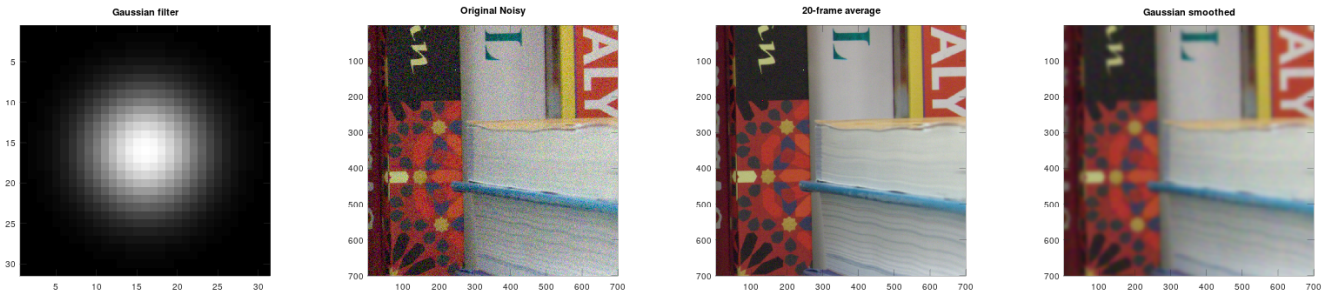
For fast-changing signals, we need to be careful about how the averaging process works.

Noise reduction by local smoothing

For signals that are fast changing, denoising is possible with filtering that carefully averages singal samples that are near each other. Proximity in this case may be in terms of time, as in the case of a sound recording where signal samples next to each other correspond to neighbouring time instants. It can also refer to spatial proximity as in the case of digital images, where pixels nearby pixels correspond to parts of an image that are spatially close.

The specific type of averaging that will yield the best results depends on the type of signal, and the characteristics and amount of noise present in the signal. But regardless of how it is done, this process will ***unavoidably*** also destroy some of the information present in the original signal.

A classic example of spatial filtering is the standard image smoothing Gaussian filter that is available in most image processing programs. The process is illustrated below.



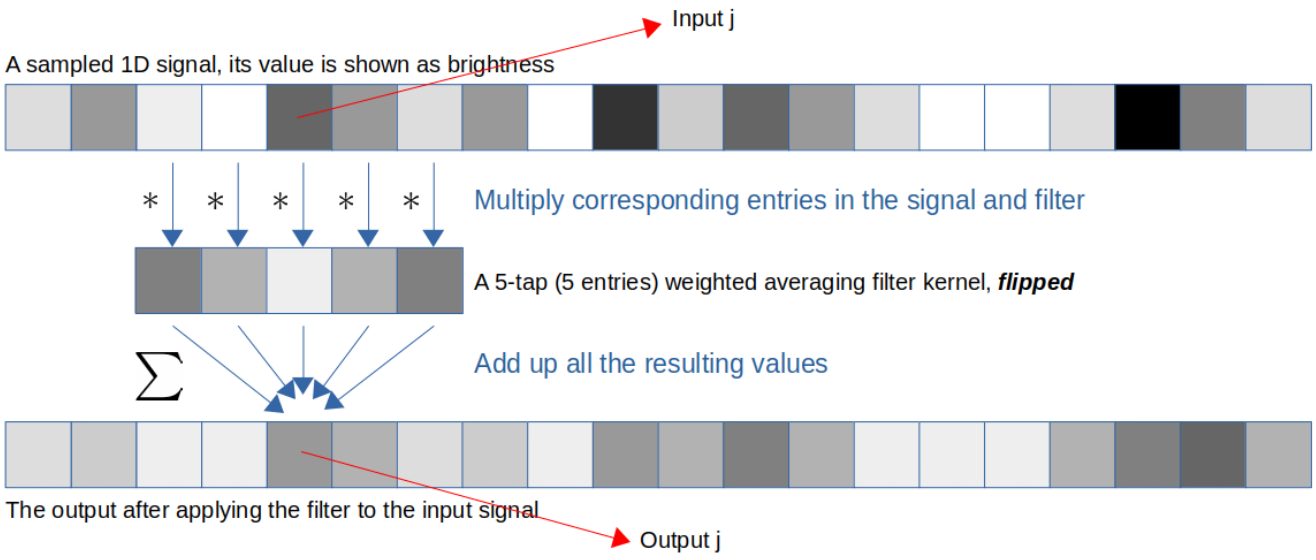
The Gaussian filter (on the lefmost picture) is applied, centered at each pixel, and averages the values of neighbouring pixels with a weight that decreases as we move away from the center (the weight is represented in the image by the brightness, white is highest, black is lowest). The goal is to mix neighbouring pixels giving more weight to those that are closest to the central one.

The result of denoising by averaging multiple readings is shown for comparison. The 20-frame average gives the best result, it has much reduced noise while preserving the detail present in the original image. The Gaussian smoothed result has eliminated most of the noise, but as expected, it also has removed some of the information in the image – this shows as blurryness where sharp edges occur in the original.

Weighted averaging should be considered whenever we are not able to provice multiple readings of a slow-changing or static signal. This is a very common situation in automated systems where we often have a single sensor, and this sensor is measuring a time-changing signal at a rate that is not fast enough to allow us to carry out simple averaging.

Linear Filtering and Convolution

Weighted averaging on a signal is typically done by applying a linear filter to the signal, the mathematical operation we perform is called a convolution between the signal and the filter. The figure below shows how the convolution process works for a Gaussian averaging filter on a 1-D signal.



To obtain the filtered output at position ***j***, we place the filter kernel centered at ***j***, multiply the corresponding overlapping entries from the signal and the filter, and add up the resulting values. We have to perform this operation for each entry in the output.

The array with the values used by the filter is called a ***kernel***, and it is ***flipped horizontally*** before we compute the output values. This is because mathematically, the convolution operation is defined as

$$O(j) = \sum_{k=-N/2}^{N/2} I(j+k) \cdot h(-k)$$

where ***O(), I(),*** and ***h()*** are the output, input, and filter kernel respectively, and the filter ***h()*** has ***N*** entries. Notice that the filter entries are applied in ***reversed order***. This is equivalent to multiplying corresponding overlapping entries between a signal and the flipped filter kernel. This is required to preserve an important property of convolution. By definition, if we applied filter ***h()*** to an input signal that consists of a unit impulse function

$$\delta(j) = \begin{cases} 1 & \text{if j=0} \\ 0 & \text{otherwise} \end{cases}$$

the output should be exactly the same filter kernel. If the filter is not flipped, the convolution would result in a flipped output. The corresponding operation ***without flipping the filter*** is called a ***correlation***. This is a very subtle distinction but it's important for you to remember.

In practice, we often use filters that are horizontally symmetric in which case the convolution and the correlation produce the same output, but this is not always the case. So be careful! There is an analogous definition for a 2D convolution (which is what was used above to smooth the noisy image using a 2D Gaussian filter kernel), and the operation can be generalized to higher-dimensions – convolution in higher dimensions works on ***tensors***.

Convolution with linear filters can be used for a variety of purposes – smoothing is only one of them. With the right filter kernel it is possible to detect specific patterns in a signal, enhance specific frequencies or signal content (as in an audio equalizer), and extract meaningful ***features*** from the input signal (for instance, to detect specific events in a time sequence).

What do we do about categorical or discrete measurements?

Some sensors return measurements that are discrete or categorical (i.e. they correspond to one of a small number of pre-defined categories). Averaging does not work in this case, as neither the signal nor the added noise will be real-valued.

However, we can still apply the idea of carrying out multiple measurements.

Under the assumption that the underlying signal is slow-changing or static compared to the rate at which we're sampling, we can use one of the following techniques:

- Obtain multiple measurements, take the median value

- Obtain multiple measurements, take the mode (the value that occurs most often)

Which of these is most appropriate depends, once again, on the type of signal, sensor, and noise. Often you will need to try different approaches and select the one that is most suitable to your particular problem. Regardless, what you should not forget whenever working with sensors is:

* All sensors return noisy values, some are better than others but noise is unavoirable
* You need to spend some thinking about whether or not, and how much denoising is needed for your specific application.
* You need to be able to implement simple denoising procedures based on averaging, or in the case of categorica/discrete signals, using the median or mode.
* You need to be able to determine whether simple denoising is enough, or whether you need to implement a stronger denoising method specific to your sensor/application.