Data Fundamentals (H)

Week 10: Digital Signals and Time Series

Time series, sampling, interpolation, filtering

DF3 - University of Glasgow - John H. Williamson - 2017

Summary

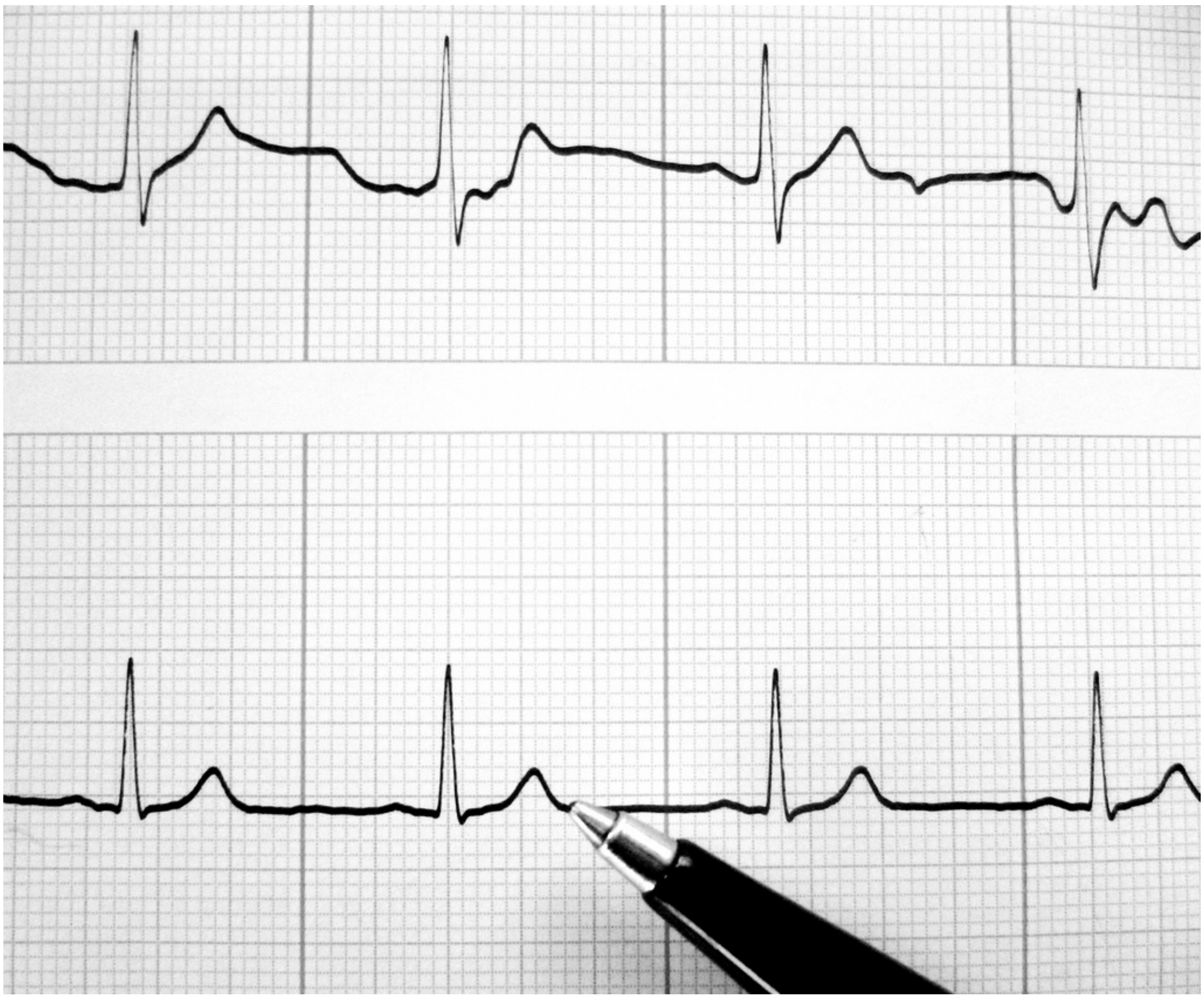
By the end of this unit you should know:

- what sampling is, for 1D and 2D signals
- how amplitude quantization affects signals
- what the sampling rate and Nyquist limit are
- what aliasing is, and how and when it manifests itself
- why interpolation is needed
- how to interpolate and resample digital signals correctly
- how to interpolate irregular data into a digital signal format

- the purpose of filtering signals
- how moving averages work
- how median filtering and order filters work
- what the frequency domain and time domain are and what they represent
- how the Fourier transform relates the time domain to the frequency domain
- what convolution is and how the Fourier transform can be used to predict its behaviour
- how convolution is implemented
- standard types of convolution, including smoothing

Example: an ECG

Electrocardiograms (ECGs) are used to measure the electrical behaviour of the heart, and are essential diagnostic tools in cardiology.



/>

[Image (https://flickr.com/photos/rvoegtli/5343361247) by Rosmarie Voegtli (https://flickr.com/people/rvoegtli) shared CC BY (https://creativecommons.org/licenses/by/2.0/)]

The ECG is captured via voltage measurements at the skin surface. If you had to build a computer system to support an ECG device, there are various questions you might need to be able to answer:

- How can we represent this signal as an `ndarray`?
- How can we efficiently transform it (e.g. scaling it or removing offsets)?
- How can we be sure we have represented it accurately?
- How can we smooth out random fluctuations in the signal while keeping the diagnostically relevant aspects?
- How can we align this with other signals we have measured, like pulse rate or oximetry data?
- How can we capture the periodic, repetitive nature of the signal and analyse it in those terms?

We will answer these questions today.

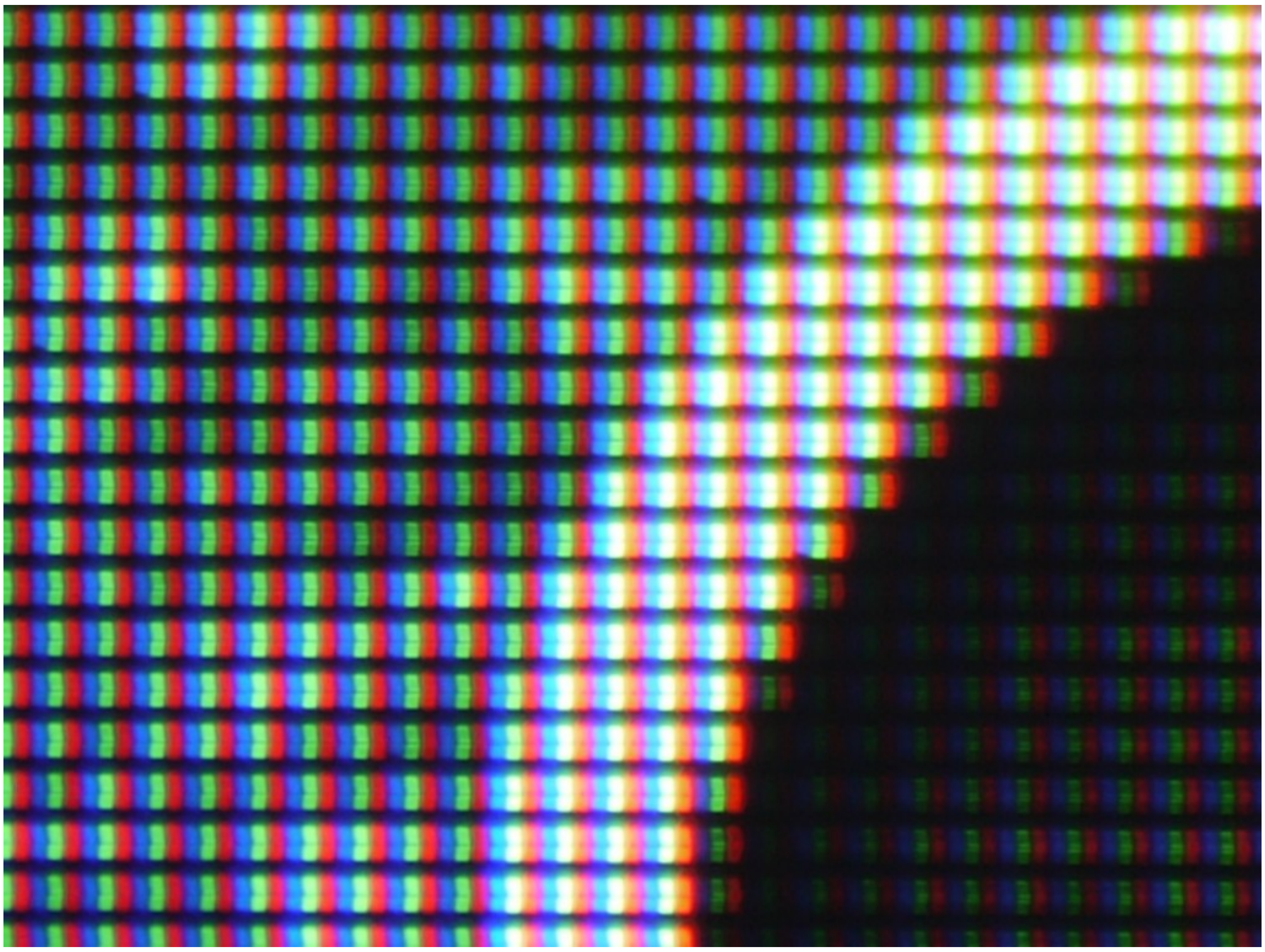
```
Out[2]:
```

Show Code

LaTeX commands

$$\mathbf{x}\mathbb{R}$$

Time series and signals

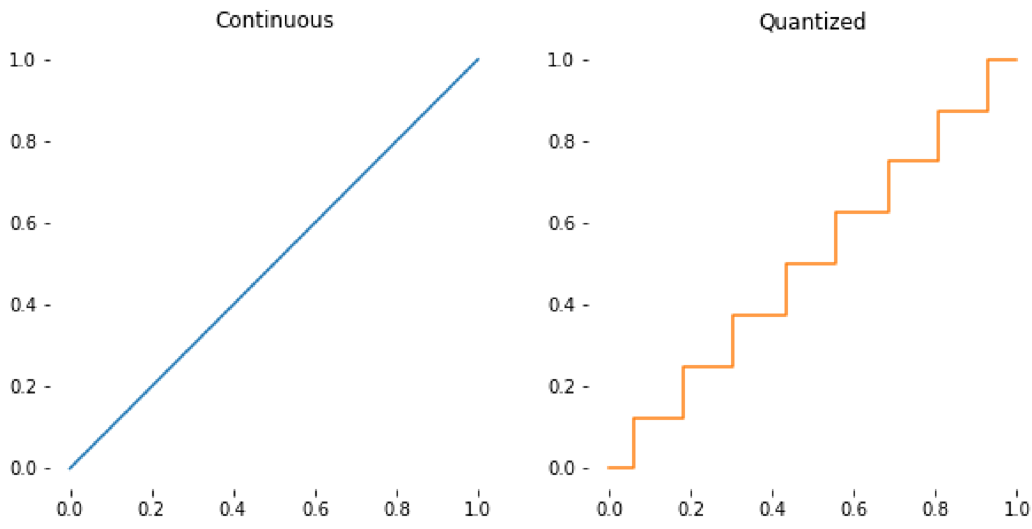

/>

[Image (https://flickr.com/photos/56164003@N02/8419208409) by citoki0815 (https://flickr.com/people/56164003@N02) shared CC BY (https://creativecommons.org/licenses/by/2.0/)]

Real world signals like sounds, light patterns and temperatures are continuous in both value and time/space. We can imagine them as real functions of real values:

$$x_t = f(t)$$

We have to **sample** them to make them amenable to digital signal processing. This involves quantising in both value and time/space. **Quantisation** means to force something to a discrete set of values (e.g. integers in range [-128, 127]).



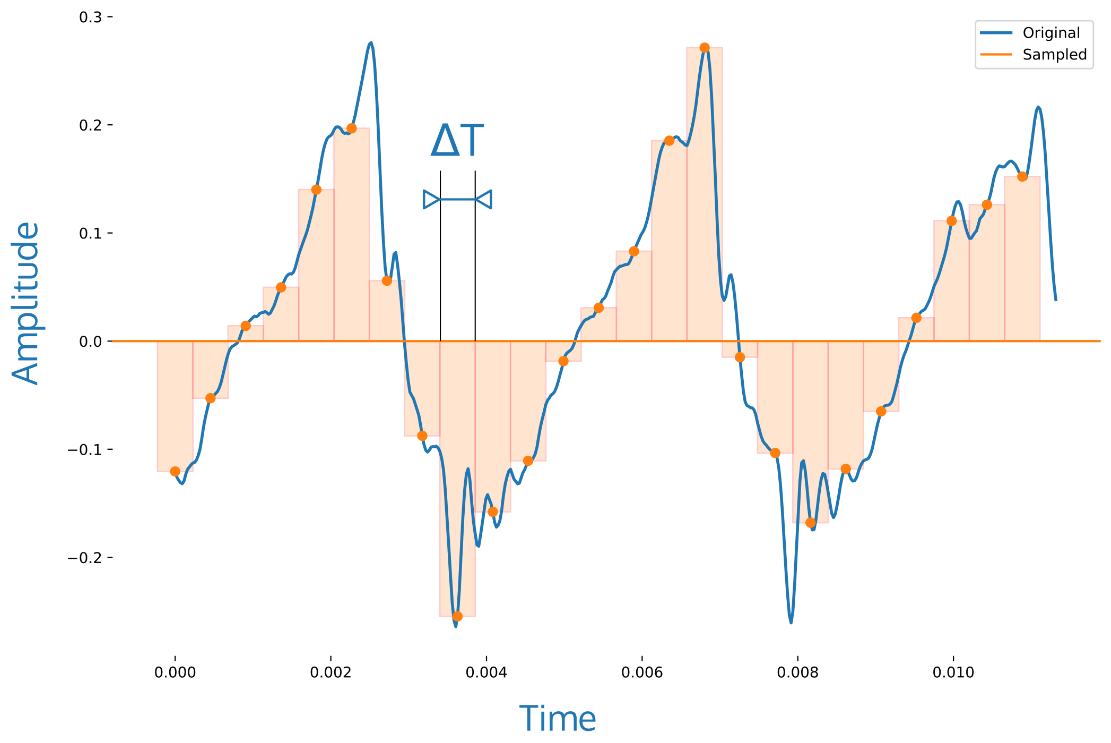
/>

We assume we have some continuous signal

$$x_t = f(t),$$

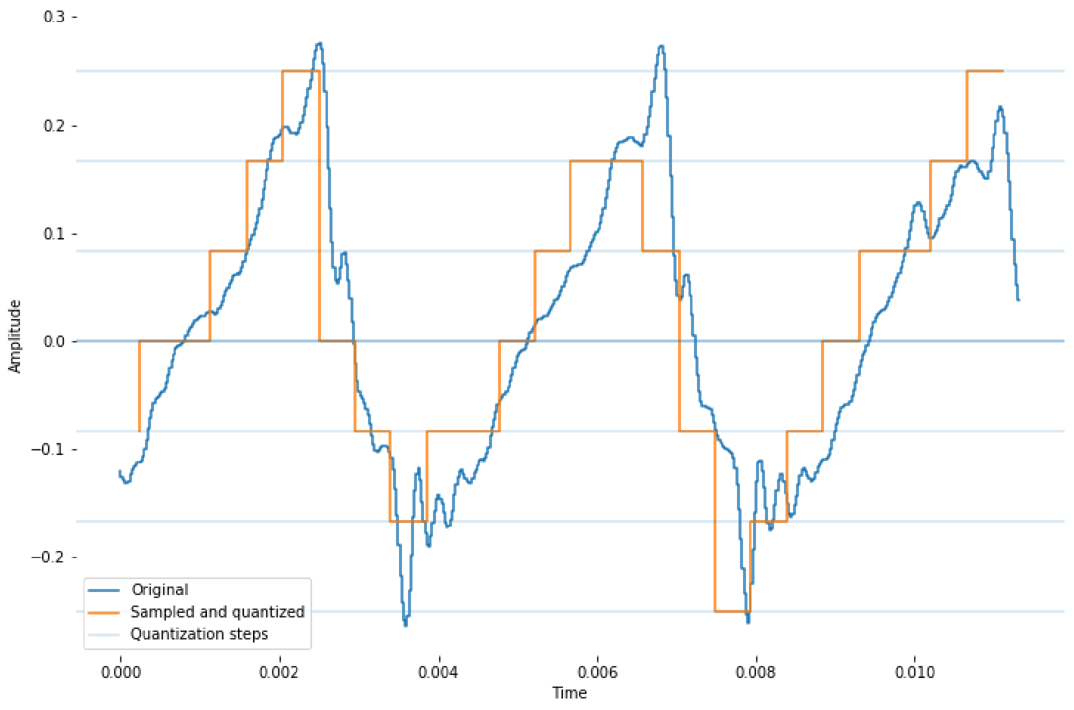
where x_t and t are continuous; perhaps $x_t \in \mathbb{R}^N$ and $t \in \mathbb{R}$. x_t may be vector valued $\mathbf{x_t}$, such as the temperature recorded at 10 different weather stations at a particular moment t; but we'll deal with scalar x_t for the moment.

We **sample** it *regularly* to obtain a sequence of measurements $x[t] = [x_1, x_2, \dots, x_n]$. This sampling is done by making measurements with precisely fixed time intervals between them. Each measurement records the value of $f(t)$ at that instant. This is *time quantization*.



/>

Each sample is itself quantized to a fixed set of values, so that it can be represented as fixed-length number in memory (e.g. an `int8` or a `uint16` or a `float64`). This is *amplitude quantization*.



/>

The result of this process is that we capture the essence of the continuous time varying function $f(t)$ as a 1D `ndarray` of numbers (or a 2D `ndarray`, if $\mathbf{x_t}$ was vector valued).

Note that this is the same *data structure* as we used for vectors, but its mathematical interpretation is quite different for signals.

Sampled sequences: throwing away time

This sampling process is extremely useful, because we can avoid explicitly storing time. Say we have a set of measurements over time, like the *wheat pricing data* from way back Lecture 1. This has the price of a quarter bushel of wheat, measured every year.

```
 Year(t)   Price (x_t)
1570  45.0
1575  42.0
1580  49.0
1585  41.5
1590  47.0
1595  64.0
1600  27.0
1605  33.0
1610  32.0
1615  33.0
1620  35.0
1625  33.0
1630  45.0
```

each being a (t, x_t) pair. A **sampled signal** is stored as a single vector of $[x_0, x_1, \dots, x_T]$, with the assumption that time starts at 0 in the first element and increases by a (precisely) fixed amount for every successive element. So the wheat data would become:

```
 [45.0, 42.0, 49.0, 41.5, 64.0, 27.0, 33.0, 32.0, 33.
0, 35.0, 33.0, 45.0]
```

Note that t is dropped entirely. The value of t is *implied* by the regular sampling process. We can use the index of the vector to work out the real world time at any point.

Sampling rate

We do need to store the **sampling rate** of the signal, which is how frequently we have sampled the original data. For example, if the measurement is sampled over time, the sampling rate would tell us the number of times a second (or a week, or a year) that the measurement was taken. In the wheat example above, the sampling rate would be every 5 years. This sample rate is called f_s.

A sampling rate of f_s means that the spacing between samples ΔT is $1/f_s$. For example, sampling at $f_s = 100$Hz is the same saying we take a measurement once every $\Delta T = 0.01$ seconds. In imaging, the same concept applies, and the sampling rate would be the number of measurements per spatial subdivision (e.g. 72 pixels per inch). Since this is fixed for the whole sequence of samples, we only need one sample rate for the entire sequence.

Why sample signals?

This is a compact way to represent an approximation to a continuously varying function as an array of numbers. This saves storage space over explicitly storing time and it allows for a range of very efficient algorithms to be applied to signals. It allows us to work computationally with approximations of *analogue* signals like sounds and images, which are continuous in space and time. It also means we can perform any standard array operation on our samples. For example:

- mixing two signals is equivalent to (weighted) sum of their sample arrays.
- correlation between signals can be computed using elementwise multiplication
- selecting regions of signals can be done via slicing
- smoothing and regression can be applied to signal arrays

These are efficient and compact ways of manipulating discrete approximate representations of continuous signals.

Sampling theory

If we sample a signal with enough points -- frequently enough -- we can *perfectly* reconstruct any given continuous signal, as long as that signal does not have too much *high frequency content* (we will be precise about what this means shortly; for now we can loosely define it as a signal that is not too "rough").

This surprising result is the foundation of **digital signal processing**. It is what allows us to manipulate sounds, images, videos, ECG traces, and so on, as ndarrays with confidence that we are performing operations that are meaningful in terms of the original signals.

Nyquist limit

Specifically, for a given sampling rate f_s, we can recover an original signal $x(t)$ *perfectly* if the signal contains frequencies at most

$$f_n = \frac{f_s}{2}$$

.

This limit $\frac{f_s}{2}$ is called the **Nyquist limit**, f_n.

For example, audio is often recorded with a sample rate of 44.1KHz, because human hearing extends to about 20KHz. So to represent all frequencies that someone can hear, we need to have a Nyquist frequency of 20Khz, and thus a sample rate f_s of at least 40Khz (the "extra" 4.1Khz are for a complicated combination of engineering and legal reasons).

To make this work, audio signals are first *filtered* (by hardware) to remove all frequencies above 20KHz before sampling is performed.

Aliasing

If we don't follow this rule and we don't sample often enough, the result is **aliasing**. Aliasing results in unpleasant artifacts, where high-frequency components are "folded over". If we sample a signal component with frequency $f_q > f_n$, we will observe an artificial component at $f_n - (f_q \mod f_n)$ in the sampled signal. This creates phantom behaviour in the signals, which doesn't correspond to any real-world changes in the signal value.

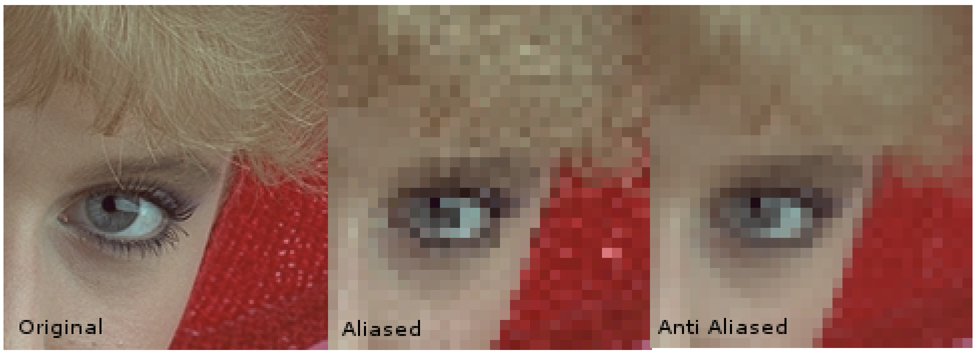
Video

In video, this often manifests itself as the **wagon wheel** effect, where fast moving objects appear to stop and then reverse their direction as they speed up. This occurs precisely when the frequency of movement exceeds half the frame rate of the video/film.

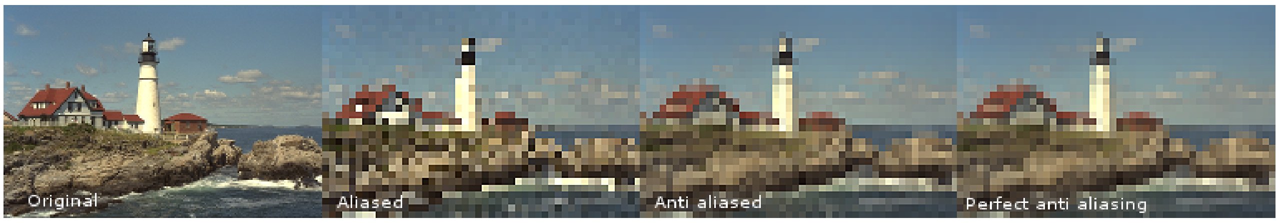
/>

Images

When an images is reduced in resolution, it must be **filtered** so as to remove high-frequency components that cannot be represented at a lower resolution, or serious aliasing occurs (i.e. spurious low-frequency components appear from high frequencies being "wrapped down"), which looks awful.



/>



/>

When the resolution of an image is reduced, the sample spacing gets larger, and so the signal must have less high-frequency content if it is to be represented accurately.

Audio

Aliasing is clearly audible when signals are inappropriately sampled. Correctly synthesizing signals that sounds "as if" they were generated in the real-world and then sampled without introducing aliasing is a major topic in the digital synthesis of music (it's very hard!).

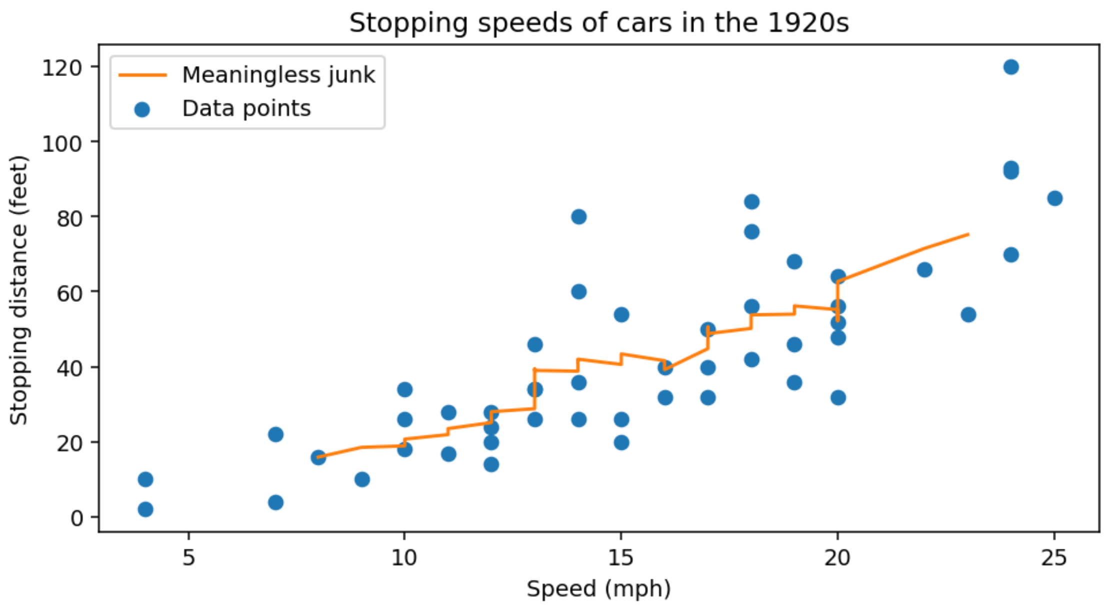
0:00 / 0:04

0:00 / 0:04

Irregular data and timing

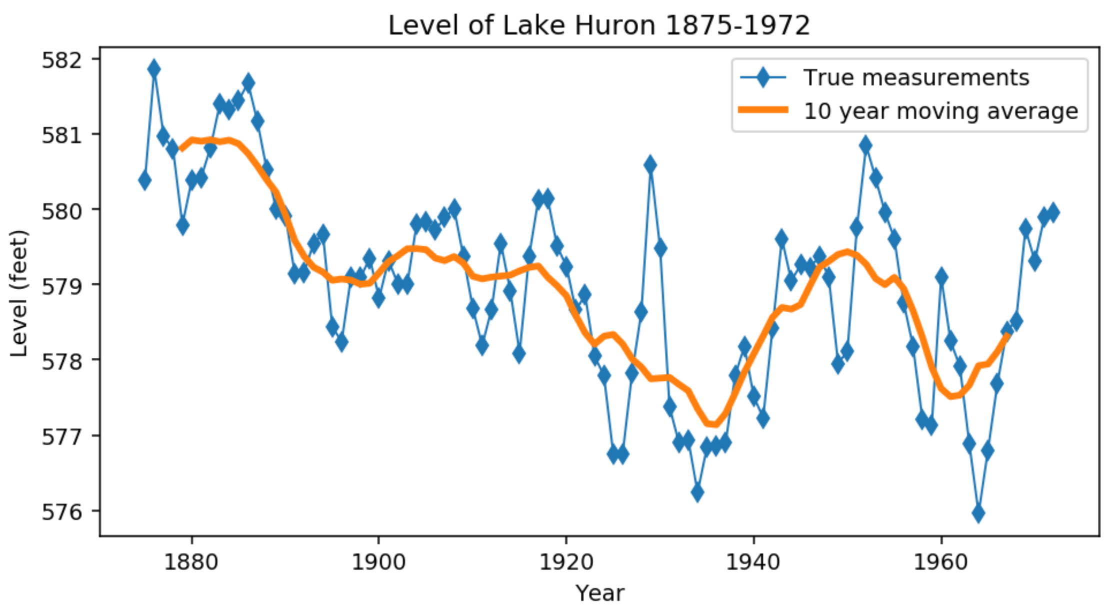
Remember the car stopping data from Unit 3's lab, where the stopping distance of 1920's cars was plotted against the speed of the vehicle. Some people plotted the moving average of the stopping distances. While this is a good idea in *theory*, it doesn't make sense, because the moving average requires a regularly sampled signal.

```
Out[6]:   Text(0,0.5,'Stopping distance (feet)')
```



Why couldn't we just plot a moving average of this data? We could do this just fine for the Lake Huron data:

```
Out[7]:  Text(0,0.5,'Level (feet)')
```



But it doesn't do anything meaningful for the car stopping data.

The problem is that moving averages are meaningless *unless* the data represent a regularly sampled signal. The car stopping data isn't regularly sampled -- it just has whatever measurements were collected, with arbitrary gaps and multiple readings at one point. We will see shortly that moving averages are a special type of a general operation called **convolution**, which is a key operation on sampled signals. When we compute a moving average, we must average like with like; the values we average must be evenly spaced in time. How can we solve this problem for the car data? How can we "smooth out" the data?

There are two things we can do:

- do **regression** (like linear regression) and find a continuous function which approximates the data according to some measure of goodness (e.g. least squares). Optimisation or Bayesian inference could help us do that. We could then use this function directly to answer questions.
- convert the data to a regularly sampled form, and use signal processing operations like a moving average on the regularly sampled signal.

The second solution can be achieved by a two step process:

- **interpolation**, where we find a continuous function that closely approximates the data.
- **(re)sampling**, where we sample the continuous function according to the requirements of a sampled signal (i.e. evenly spaced data ponts).

This two step process is sometimes called **gridding** because it forces irregularly spaced data onto a regular grid.

Gridding: reinterpolation onto regular grids

Interpolation

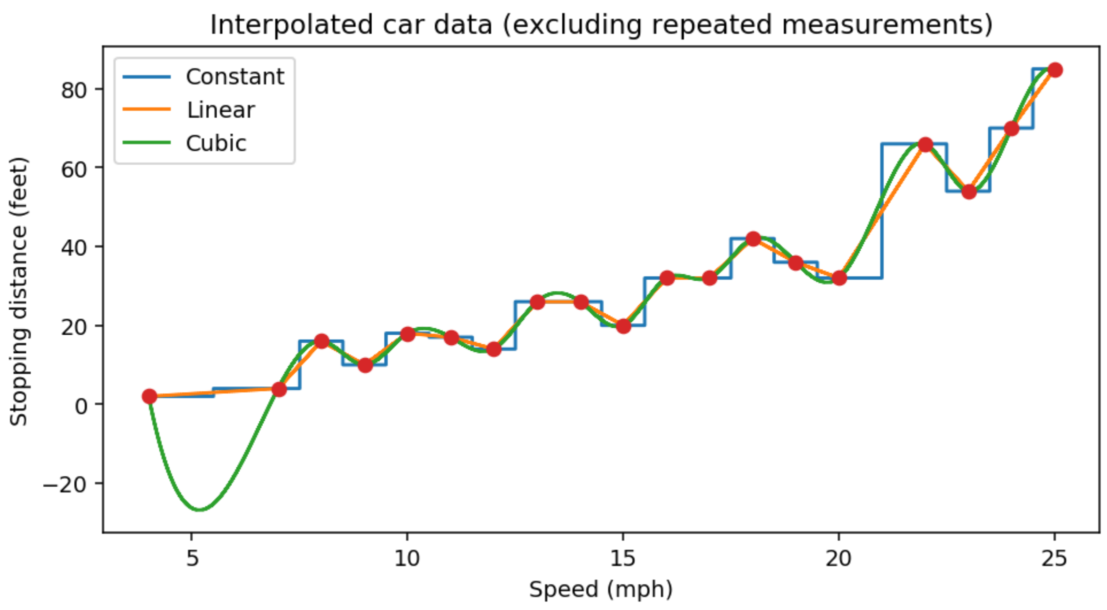
Interpolation means estimating a value between some known measurements. An **interpolation function** is a function which can produce an estimate for the value of a function that is represented by the data points observed.

There are many choices for interpolation algorithm:

- **constant** or **nearest-neighbour** interpolation assumes that data is unchanging between data points
- **linear** interpolation assumes we have straight line changes between data points
- **polynomial** interpolation fits a low-order polynomial (quadratic, cubic) through data points to find smoother approximations.

There are many others we will not discuss here. These are typically applied **piecewise**, so that the function is built up of "chunks" or "pieces" which are often just the span between two data points.

```
Out[8]:
Text(0.5,1,'Interpolated car data (excluding repeated mea
surements)')
```



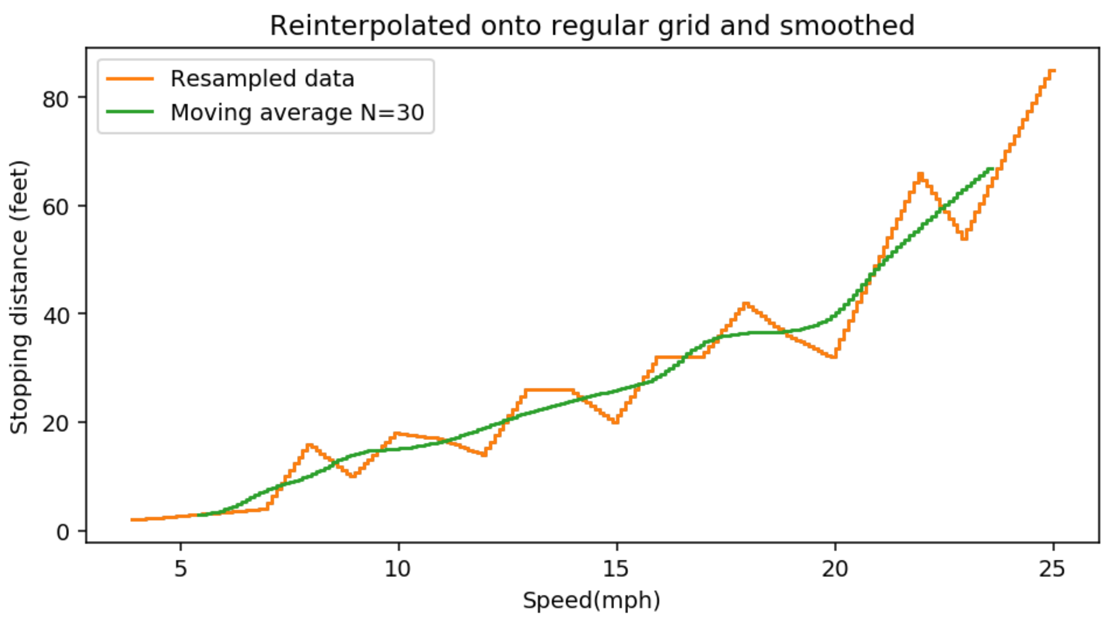
Resampling

The resampling step is trivial. Interpolation gives us a function $x(t)$ we can evaluate for any t within the bounds of the original data.

We simply evaluate $x(0\Delta T), x(1\Delta T), x(2\Delta T), \dots, x(n\Delta T)$, where $\Delta T = 1.0/f_s$, the distance between each sample at the sampling rate f_s we want to have. This gives us a regularly sampled signal that we can do any standard signal processing, like moving averages, on.

Out[9]:

```
<matplotlib.legend.Legend at 0x23eed9fb7b8>
```



Resampling to align multiple signals

What if we want to do an operation on multiple signals? For example, we might measure a number of physiological signals, sampled at different rates:

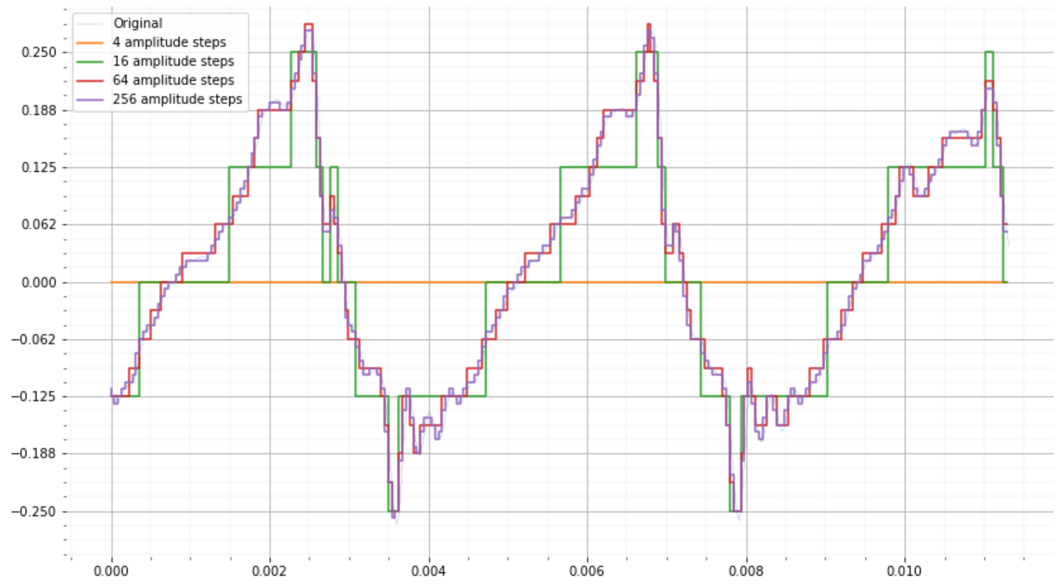
- ECG [100Hz]
- Blood oxygenation (SpO2) [1Hz]
- Lung inflation (via RIP bands) [20Hz]
- Exhalation/inhalation flow rate (via nasal cannulae) [500Hz]
- Skin conductivity [8Hz]

These might all be sampled by hardware at different sampling rates, and we might want to be able to compare ECG activity with exhalation rate. These signals cannot be directly compared. Instead, we can resample all of the signals to a common rate, perhaps 100Hz, and then manipulate the signals in this common timebase.

Sampling: Amplitude quantization

Sampling requires that we change a signal which varies continuously $x(t)$ against another value which changes continuously t. Time quantisation in sampling makes t discrete. **Amplitude quantisation** makes $x(t)$ discrete, by reducing it to a number of distinct values, typically evenly spaced. The number of levels used is often quoted in bits:

- 6 bits=64 levels
- 8 bits=256 levels
- 10 bits=1024 levels
- 16 bits=65536 levels
- and so on.



/>[Image: a signal can be discretised (quantised) at different levels of granualarity. This is amplitude quantisation.]

Time quantisation can introduce aliasing. Amplitude quantisation introduces **noise** to signals. It may not seem like it at first glance, but because the difference between the value of a signal and the *quantisation levels* is random, the effect is an increase in the noise present in the signal.

This can be clearly heard in audio, for example.

```
c:\conda3\lib\site-packages\scipy\io\wavfile.py:273: WavFi
leWarning: Chunk (non-data) not understood, skipping it.
  WavFileWarning)
```

0:00 / 0:02

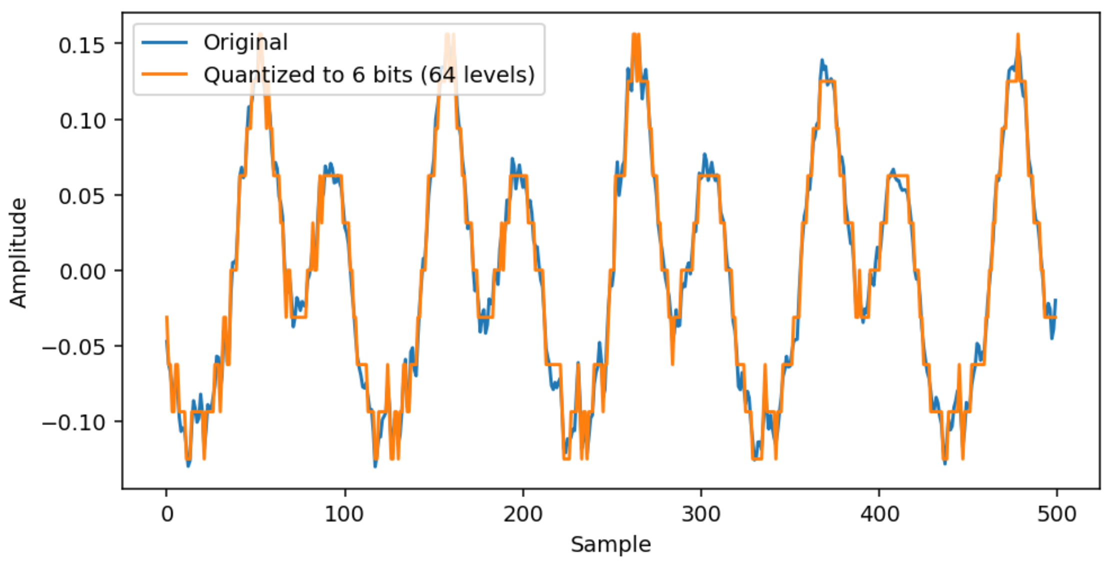
0:00 / 0:02

0:00 / 0:02

We can plot the **residual** between a high amplitude resolution signal (e.g. 16 bits=65536 values) and a low resolution signal (e.g. 6 bits=64 values). We can see the residual looks random and unstructured.

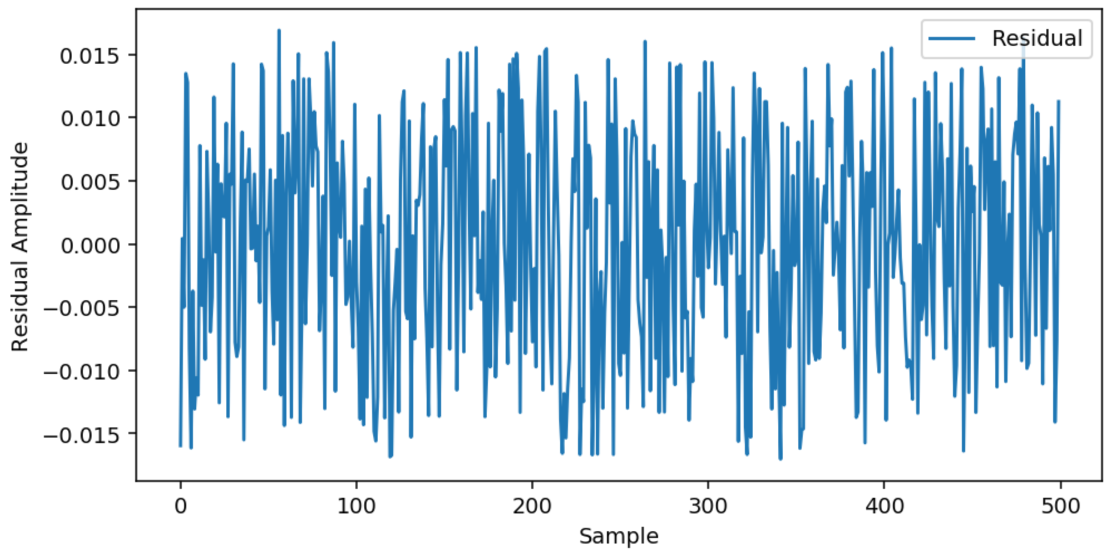
```
Out[13]:
```

```
Text(0,0.5,'Amplitude')
```



We can plot the residual (difference) of the 6 bit quantized signal from the original (well, floating-point) signal. This is how much error the quantisation has introduced. As can be seen, the result is pretty much just random noise, with no structure remaining. Quantisation adds measurement noise.

```
Out[14]:
          Text(0,0.5,'Residual Amplitude')
```



Noise

All measurements of the world introduce noise. Measurement can never perfectly capture the state of the world in all of its detail. This means that time series of signals have some level of noise when they are recorded. Quantisation to digital form, as we saw above, also introduces noise.

Going back to the example of wheat prices measured over time, we can imagine the signal $x(t)$ has two components: $y(t)$ which is the *real* measurement we want (e.g. the real price of wheat) and $\epsilon(t)$ which is a random fluctuation signal (e.g. the price adjustment made by the market trader you happened to ask)

$$x(t) = y(t) + \epsilon(t)$$

We can state how much of the signal we expect to be true signal and how much to be noise as the **signal to noise ratio**, which is just the ratio of the amplitude of the signal component $y(t)$ to the noise component $\epsilon(t)$. We write:

$$\text{SNR} = \frac{S}{N},$$

where S measures the amplitude of $y(t)$ and N measures the amplitude of $\epsilon(t)$. We typically represent this logarithmically, using **decibels**, which are just

$$\text{SNR}_{dB} = 10 \log_{10}\left(\frac{S}{N}\right)$$

An increase of 10 decibels in the SNR makes the signal 10 times louder relative to the noise.

[Side note: quantising a signal to N bits has a base signal-to-noise ratio of $(6.02N + 1.76)\text{dB}$*; we can compute exactly what the SNR for the quantisation error will be.]*

Removing noise

Unfortunately $\epsilon(t)$ is random, so we can't just subtract it out of the equation and find $y(t)$; we don't know which part of the signal is noise and which part is real. However, if we make **assumptions** about how $y(t)$ should look, we can try and separate out parts of the signal that could never be part of $y(t)$. For example, we might assume that the true wheat price doesn't change very fast, and will be similar to what it was last year. If the measurement of the signal seems to change very quickly, we can discount those rapid changes as being implausible.

This is **filtering**; removing elements of a signal based on a **model** that encodes our *assumptions* about how the signal should really behave. All filtering requires making assumptions, and the assumptions have to be formulated into a model that can be used to separate the signal from the noise. If the assumptions are wrong, we will destroy parts of the true signal we want to measure, or fail to eliminate the noise we wanted to suppress.

Filtering and smoothing

Why filter?

Filtering takes advantage of the fact that we we have **temporal structure**. Real signals cannot have arbitrary changes. At least some portion is usually predictable, and we can code that predictive model as a filter to clean up signals. A very simple assumption is that the signal we observe is some "true" process that is corrupted by noise, and that the noise is independent at each time step. For example, each measurement might be $x[t] = y[t] + \mathcal{N}(0, \sigma)$; a true signal $y[t]$ with some random fluctuation that doesn't depend on the previous time step.

In this case, we can average out the contribution of the noise by averaging over multiple time steps. This will also average out the true signal, but if we know that the true signal is not changing quickly (i.e. it *does* have a strong dependence on the state at the previous timestep) then we won't damage that signal too much.

The bad mouse

Imagine we have a new mouse-like pointing device, which measures where an object is, but it has significant electrical noise. The cursor position produced is very disturbed by these random fluctuations.

- The position is sampled at 100Hz. 100 times per second.
- Human movements don't typically exceed 4Hz. True changes in position don't oscillate more than four times a second.

It should be able to separate the noise (changing 100 times per second) from the intentional cursor movement (changing smoothly, 4 times per second).

```
None
```

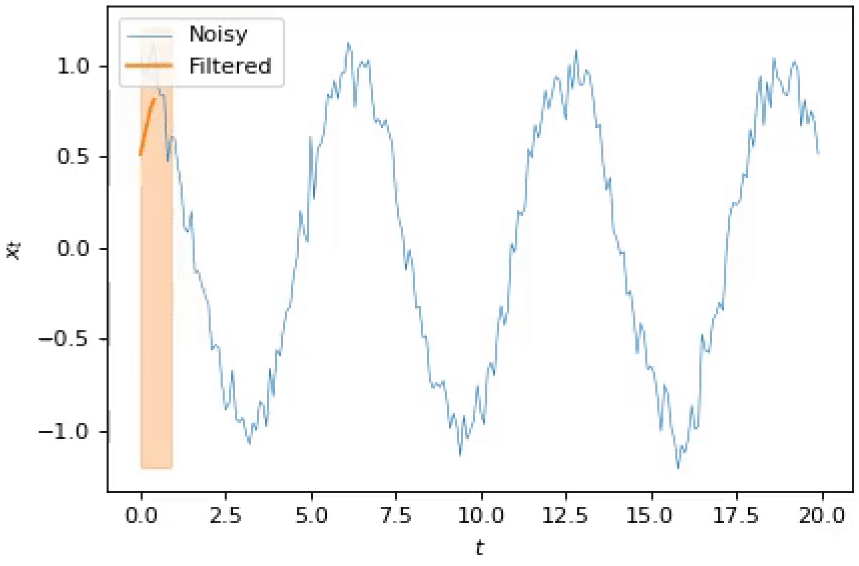
```
<function make_moving_filter.<locals>.moving_filter at 0x0
000023EEDE79268>
```

A simple filter: Moving averages

A very simple predictive assumption is this:

> Tomorrow will be the same as today.

In other words, we expect (true) signals to change slowly, and noise that we are not interested in components which change quickly. We can apply this model using a **moving average**, where we take a **sliding window** of samples and compute their mean. We then slide the window along by one sample, and take the mean of the next window, until we reach the end of the signal.



/>

$$y[t] = \frac{1}{N}\sum_{i=0}^{n-1} x[t+i-N/2]$$

Sliding window

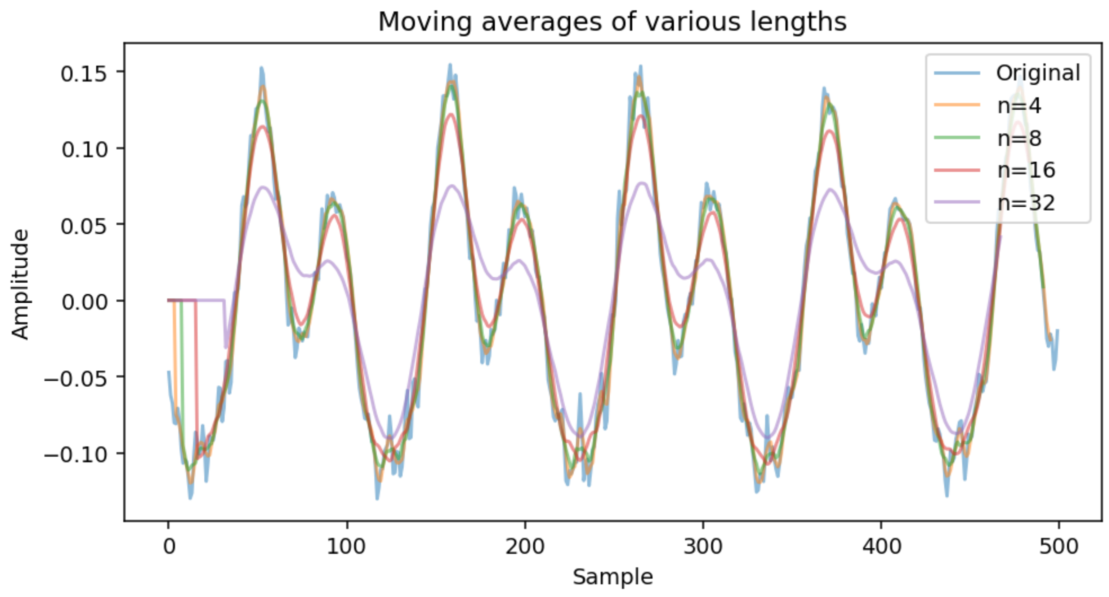
The idea of a **sliding window** is critical in digital signal processing. A sliding window takes a signal of unbounded length, and reduces it to a collection of fixed length vectors. We can perform any standard vector operation on these vectors, including matrix operations, norms, vector quantisation, and so on. This makes signals tractable with the tools we already understand for working on vectors and on arrays in general.

Using the moving average

The moving average filter has one parameter: the length N of the window.

The result of this operation is a new sampled signal with N fewer samples, as we don't have enough information to compute the samples at the start and the end of the signal.

```
Out[18]:
         Text(0.5,1,'Moving averages of various lengths')
```



Another simple filter: Exponential smooth

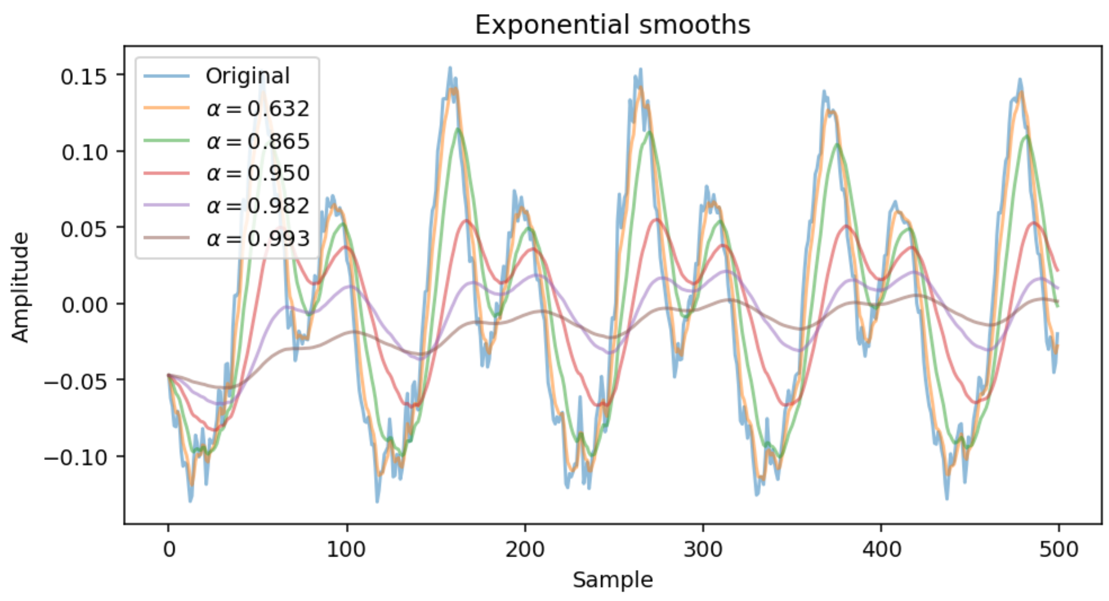
An alterative approach is to use **feedback**. We can predict the next sample from the previous sample, update with the new value of the signal, then use that prediction as the prediction for the next step.

$$y[t] = \alpha y[t-1] + (1-\alpha)x[t]$$

This is the **exponential smooth** (or a one-pole filter, or an order-1 IIR filter, or a first-order autoregressive model) and is often used because it requires very little memory and very little calculation (just two multiplies and add, and a couple of memory locations) but can smooth signals very heavily. It has one parameter, $\alpha,\ 0<\alpha<1$. If $\alpha=1$ the signal is smoothed to be completely flat and unchanging. If $\alpha=0$, the signal is not smoothed at all. As $\alpha\rightarrow 1$, the signal gets rapidly smoother.

Unlike the moving average, which can only have N adjusted in integer steps, α varies continuously.

```
Out[20]:
          Text(0.5,1,'Exponential smooths')
```



```
<function make_exp_filter.<locals>.exp_smooth at 0x0000023
EF55352F0>
Exiting...
Hit 0/1 targets in 1.4 seconds
Hits per second: 0.000
Hit rate: 0.000
```

Nonlinear filtering

Both moving averages and the exponential smooth are **linear filters**, because their output is a weighted sum of previous inputs (moving average) or previous outputs (exponential smooth). A weighted sum is a linear operation, satisfying the conditions of linearity:

Definition of linearity

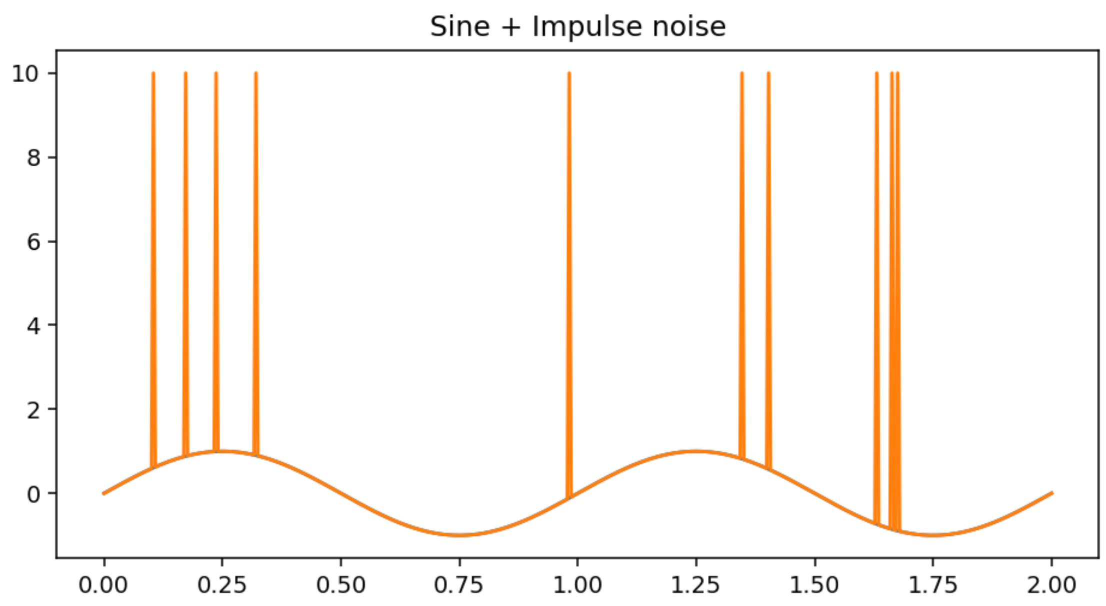
- $f(x+y) = f(x) + f(y)$
- $f(ax) = af(x)$
- $f(0) = 0$

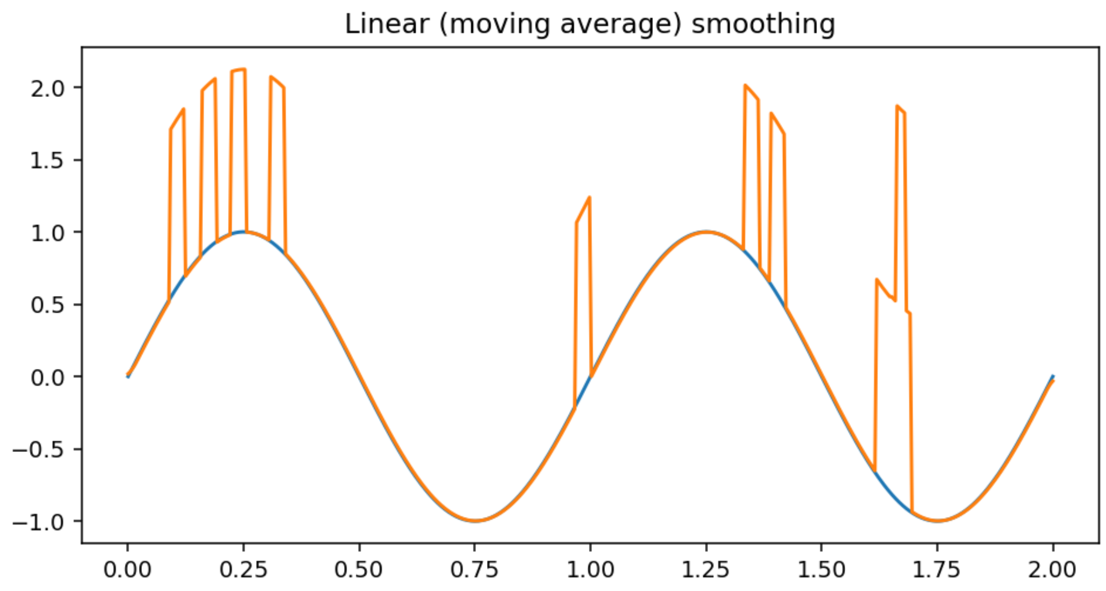
Median filtering

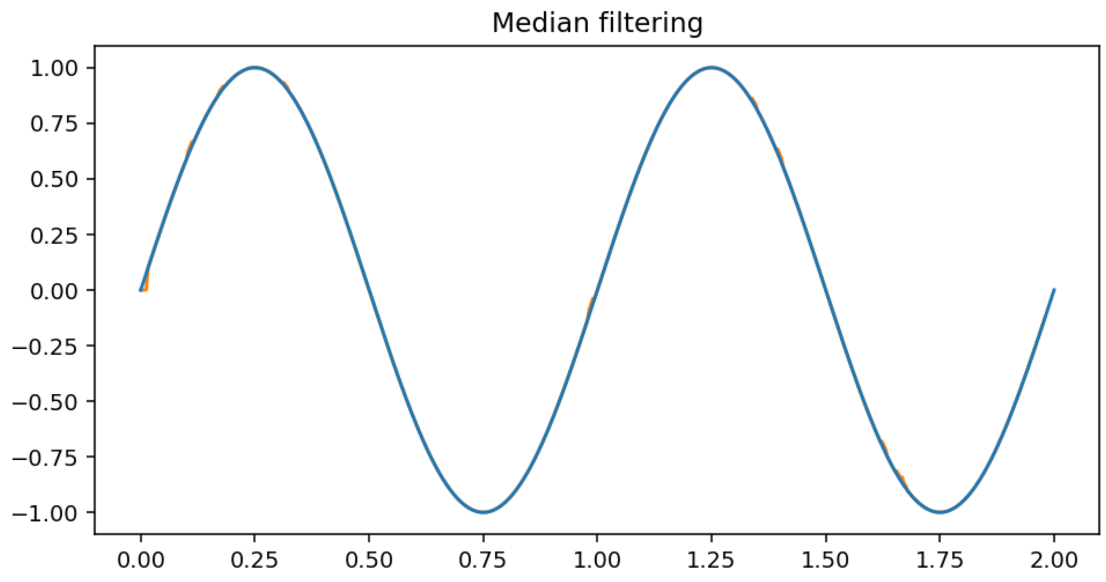
Any filter which is not a weighted sum is a **nonlinear filter**. The most common of these filters is the **median filter**, which is just like the moving average, but uses a median to estimate the central tendency of a window, as opposed to the mean. This is a filter based on a model of the world where most measurements are good, but a small minority may be arbitrarily corrupted (for example, a sensor might cut out entirely) and there is no guarantee these corruptions will be small.

This adds significant robustness to extreme values, as is always the benefit of the median over the mean. Median filters can be slow to compute, however, requiring sort operations or specialised median cascade algorithms.

```
Out[23]:
         [<matplotlib.lines.Line2D at 0x23ef560d6d8>]
```


Sine + Impulse noise
10
8
6
4
2
0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


Linear (moving average) smoothing
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


Median filtering
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

Order filters

Order filters are just the generalisation of median filters to use other *order statistics*. This includes maximum, minimum, and percentile filtering. These have specialised applications, such as "stretching out" local maxima in data.

```
Exiting...
Hit 0/1 targets in 0.9 seconds
Hits per second: 0.000
Hit rate: 0.000
```

Generalising moving averages: Convolution and linear filters

Linear filters

Many of the filters commonly used are **linear filters**. These have a *very* well developed theoretical background and their properties can be analysed in detail. A linear filter is any filter where the output is a weighted sum of neighbouring values (and the original value). No other functions can form part of a linear filter. Despite this, a very wide range of effects can be acheived using linear filtering.

Linear filters are often very efficient because CPUs (especially DSP-specific CPUs) often have a **multiply and accumulate** instruction, which multiplies a value by a constant and accumulates into a (high precision) register. This makes implementing linear filters very straightforward.

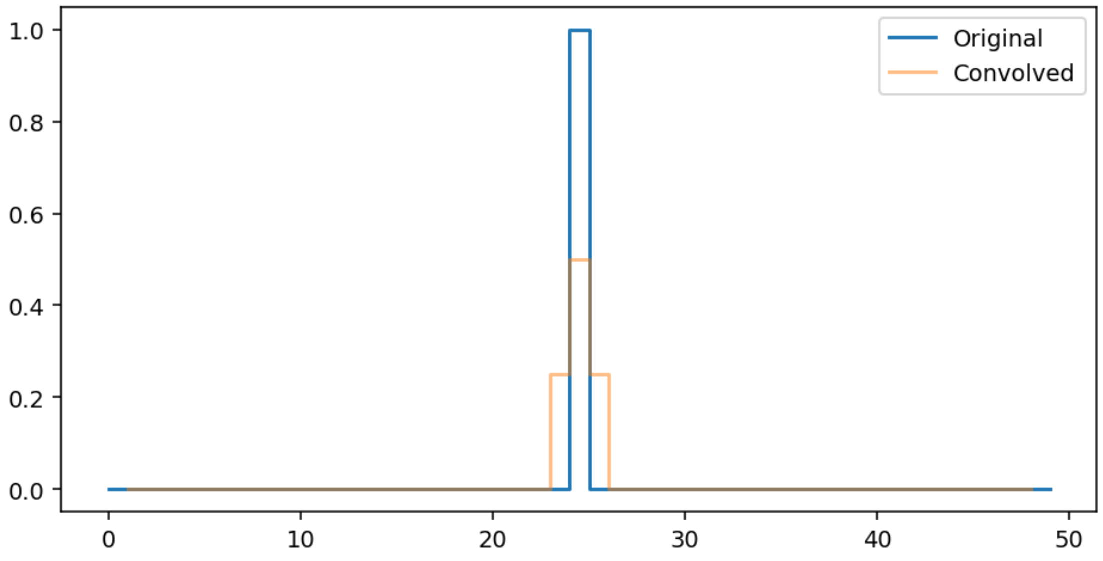
In 1D (e.g. a time series), this might be of the form:

$$f(x[t]) = 0.25x[t-1] + 0.5x[t] + 0.25x[t+1]$$

This is a weighted sum of three samples. The total value is still 1.0.

In the plot below, note that this "spreads out" the spike -- it is a smoothing operation.

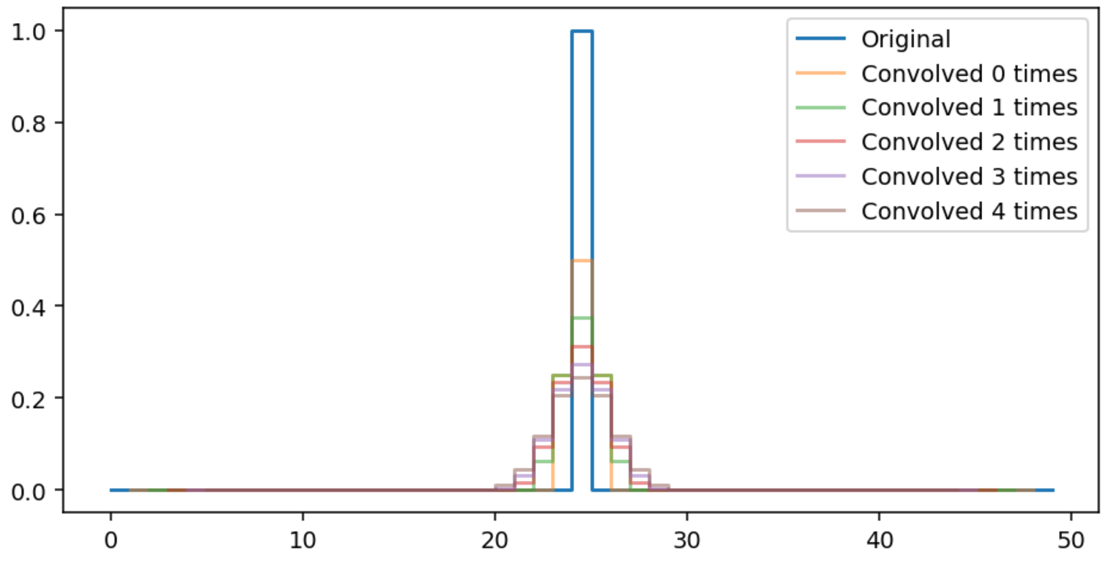
```
Out[25]:
<matplotlib.legend.Legend at 0x23ef54a8828>
```



We can repeat this operation several times. This will progressively "spread out" the spike, with each filtering operation spreading the signal without altering the overall amplitude.

Out[26]:

```
<matplotlib.legend.Legend at 0x23ef551e7b8>
```



Convolution

This process of taking weighted sums of neighbouring values is called **convolution**. It is a hugely important operation on signals.

Convolution is notated with a star $*$

$f * g$ is the convolution of f and g ($\otimes$ is sometimes used in other texts). Convolution is defined for continuous functions $f(x)$ and $g(x)$; we won't deal with convolution of continuous functions directly, but the convolution of two vectors or matrices is written in the same way.

For two vectors of length N and M, the definition of convolution is:

$$(x * y)[n] = \sum_{m=-M}^{M} x[n]y[n-m]$$

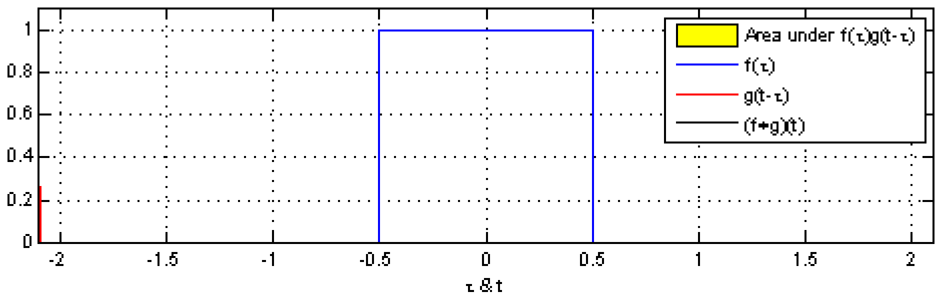
This is convolution of sampled signals.

Convolution is a very powerful operator for sampled data. It is extremely general, and effects like blurring and sharpening images, filtering audio and many forms of processing scientific data all boil down to applications of convolution. It is as important for signals as multiplication is for matrices.

The convolution kernel

In $x * y$, we can consider x to be the signal to be transformed, and y to be the operation to be performed. y is called the **convolution kernel** and is just an array like x. Typically, y is much smaller than x; it might only be a few elements long.

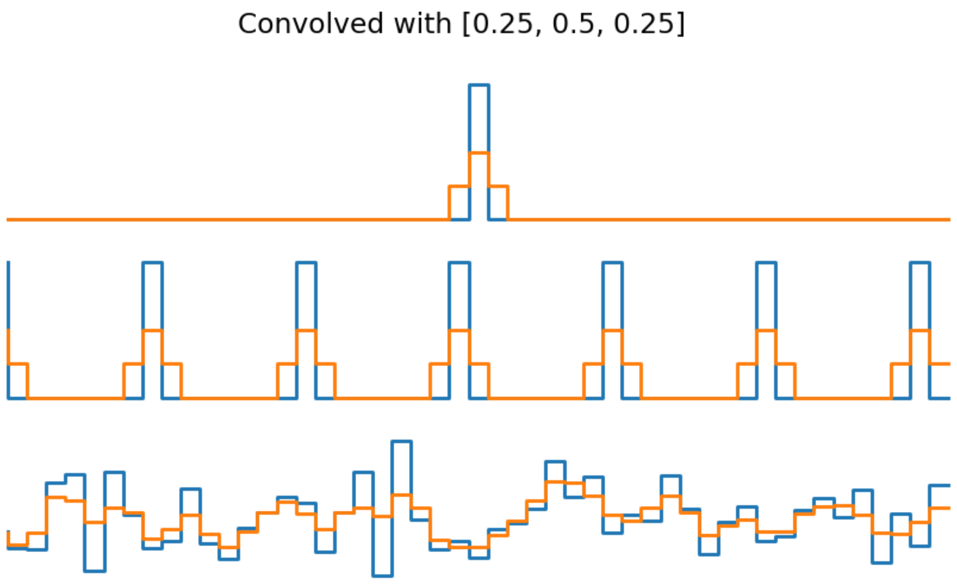
You can think of a convolution as a window *sliding* across a signal or image, summing up everything below the window:

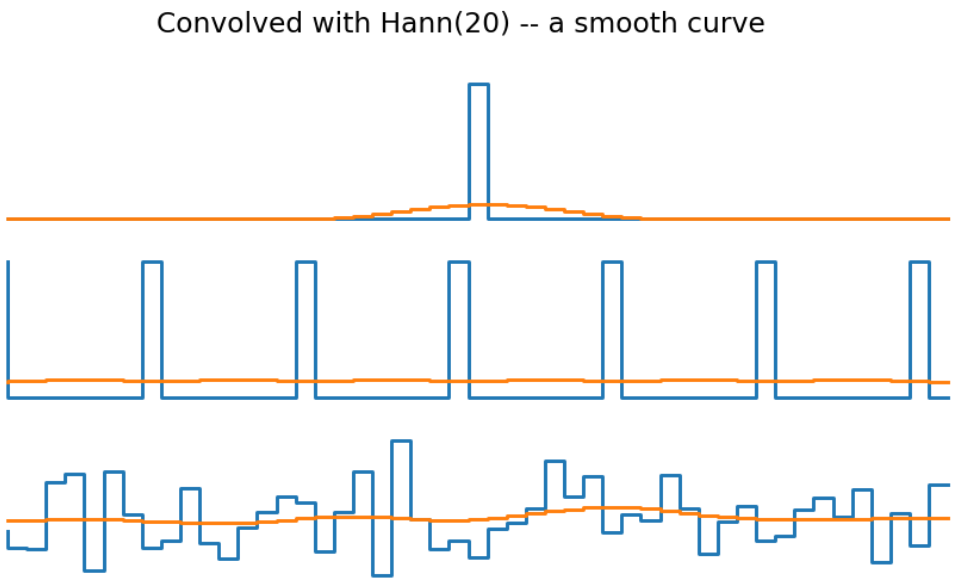


/>[Image credit: "Convolution of box signal with itself2" Brian Amber (https://commons.wikimedia.org/wiki/File:Convolution_of_box_signal_with_itself2.gif#/medi Licensed under CC BY-SA 3.0]

The window "shape" are the weights applied to form the weighted sum, which is computed at each possible offset of that window.

```
Exiting...
Hit 0/0 targets in 3.0 seconds
Hits per second: 0.000
Error in quit routine; exiting anyway
```



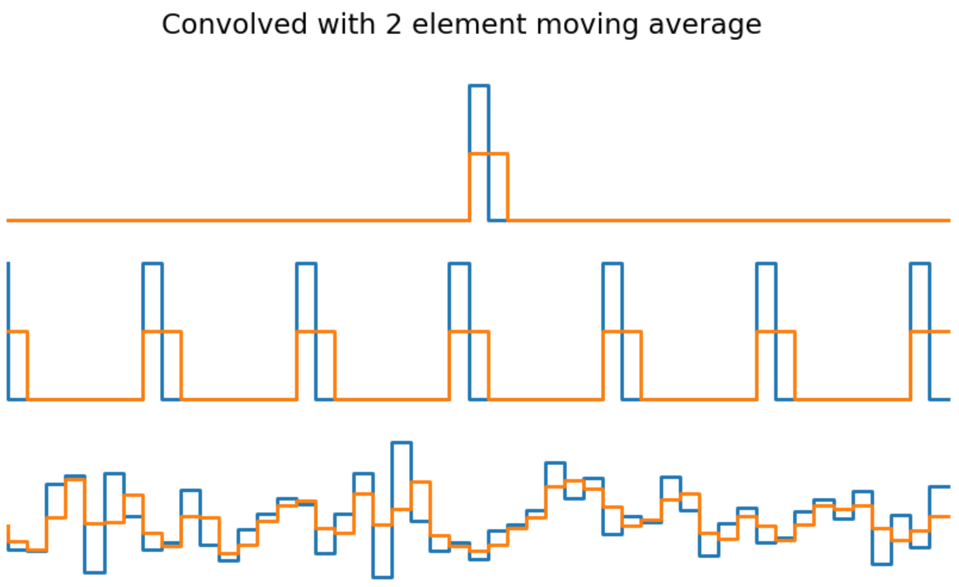


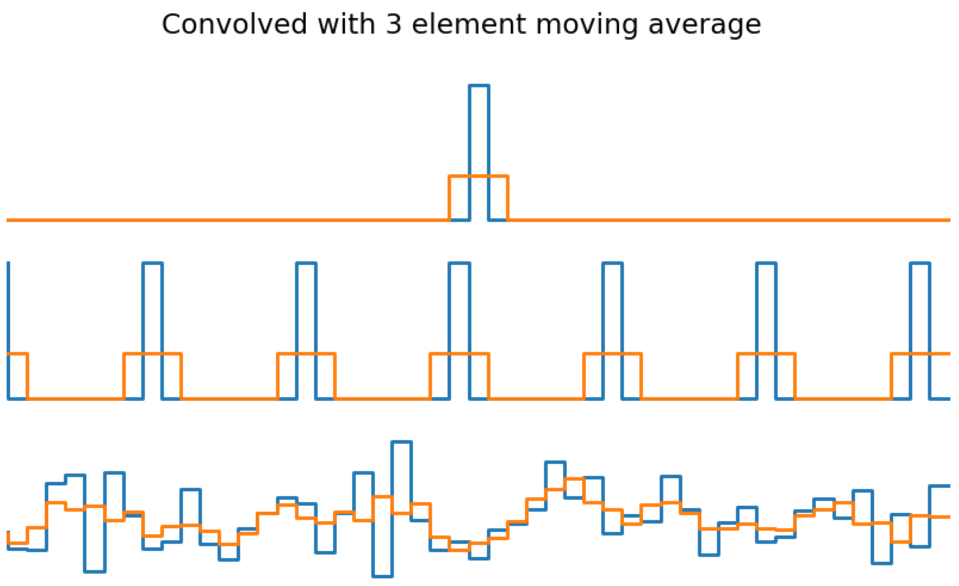
Moving average is a convolution

The moving average is a simple convolution, where there are N elements, each with value $1/N$, like:

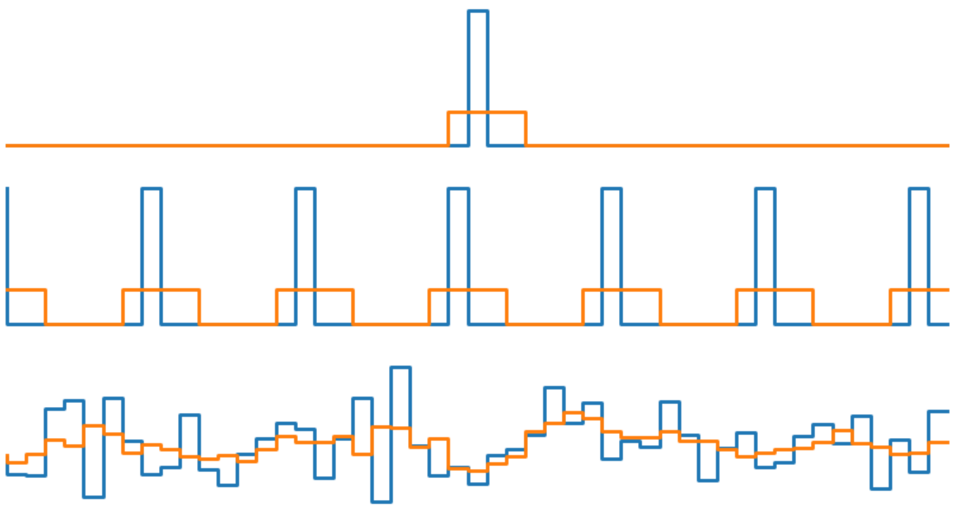
```
N=2 [0.5, 0.5]
N=4 [0.25, 0.25, 0.25, 0.25]
N=5 [0.2, 0.2, 0.2, 0.2, 0.2]
```

```
[ 0.5  0.5]
[ 0.33333333  0.33333333  0.33333333]
[ 0.25  0.25  0.25  0.25]
[ 0.2  0.2  0.2  0.2  0.2]
[ 0.16666667  0.16666667  0.16666667  0.16666667  0.166666
67  0.16666667]
[ 0.14285714  0.14285714  0.14285714  0.14285714  0.142857
14  0.14285714
  0.14285714]
```

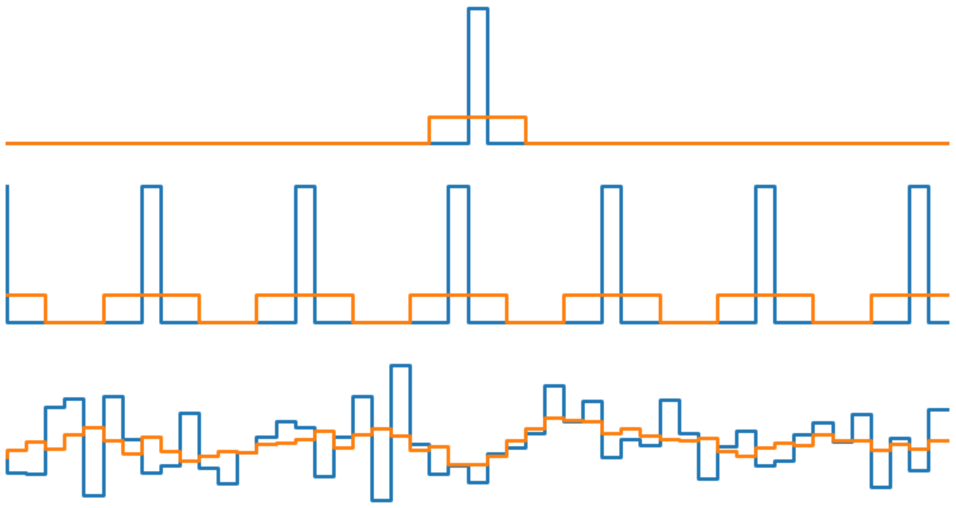




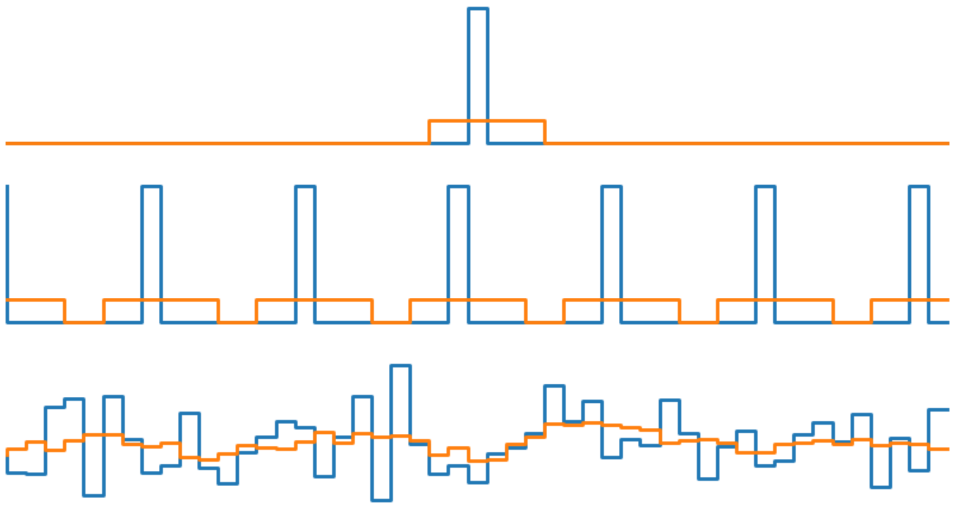
Convolved with 4 element moving average

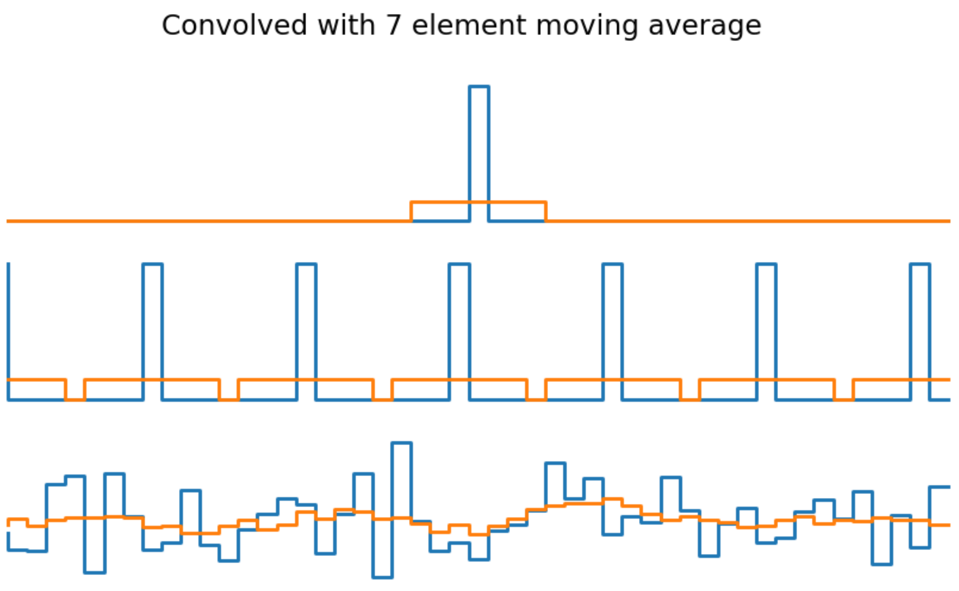


Convolved with 5 element moving average



Convolved with 6 element moving average





Quiz

Algebraic properties

The convolution operator **commutes** and **associates**:

$$f * g = g * f$$

$$f * (g * h) = (f * g) * h$$

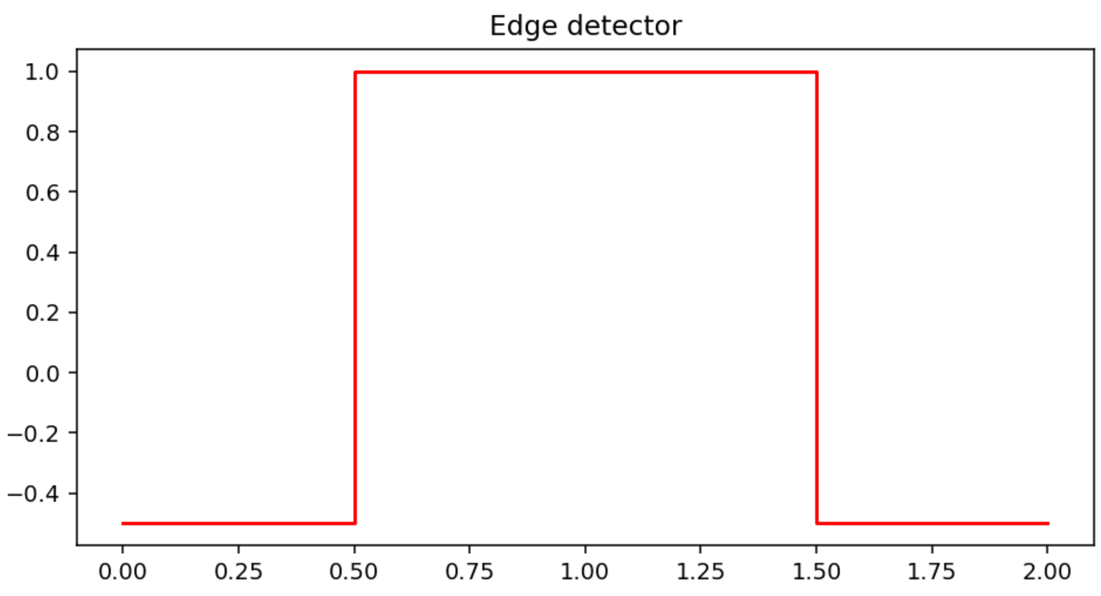
This is important, because if we have two filters -- that is two convolution kernels -- we can convolve the kernels and form a new kernel, which store and can apply to many signals. Just like the same property meant that we could combine many matrix operations into one single matrix, we can combine many convolutions into a single convolution.

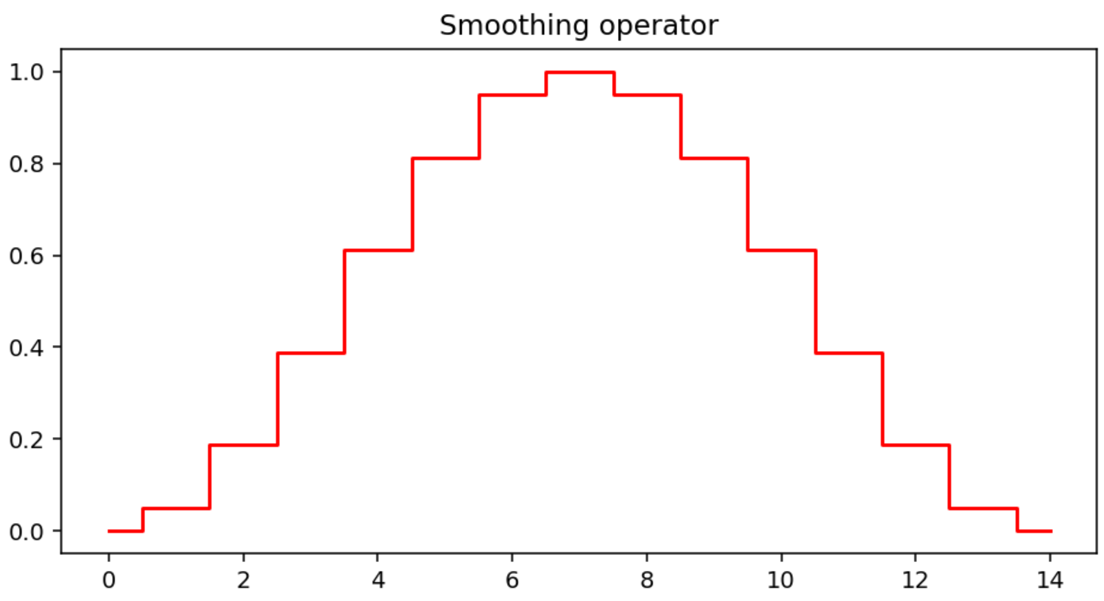
Because convolution is linear, convolution also **distributes** over addition (and thus also subtraction):

$$f * (g + h) = f * g + f * h$$

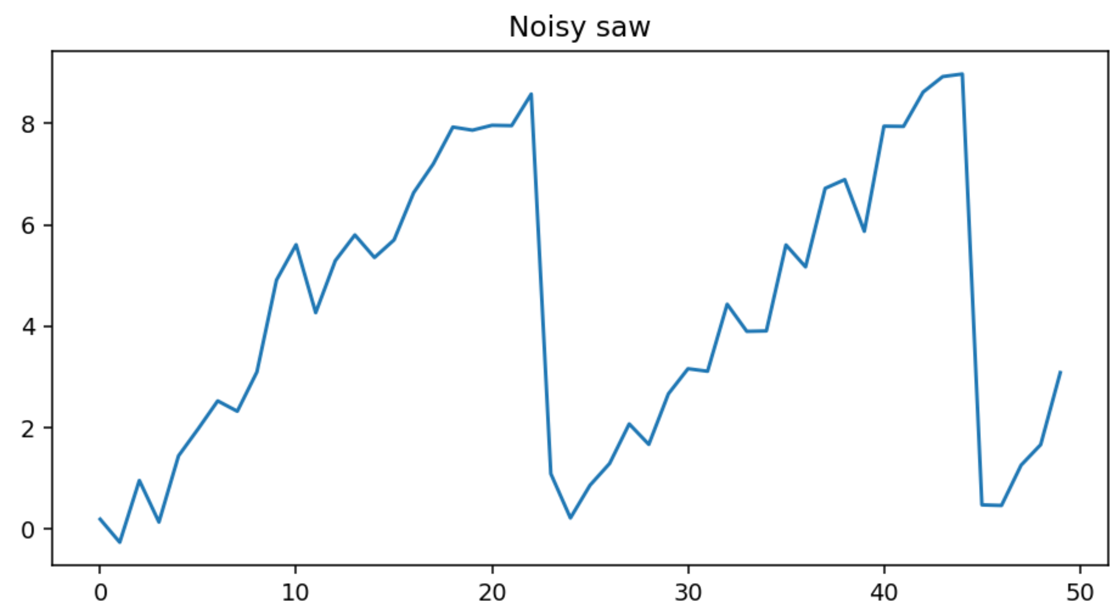
We can show this in an example. We can define two very simple convolutions: one that enhances "edges" or sharp transitions, and one that smooths out the signal. The convolution of these two kernels, produces a smoothing edge detecting kernel.

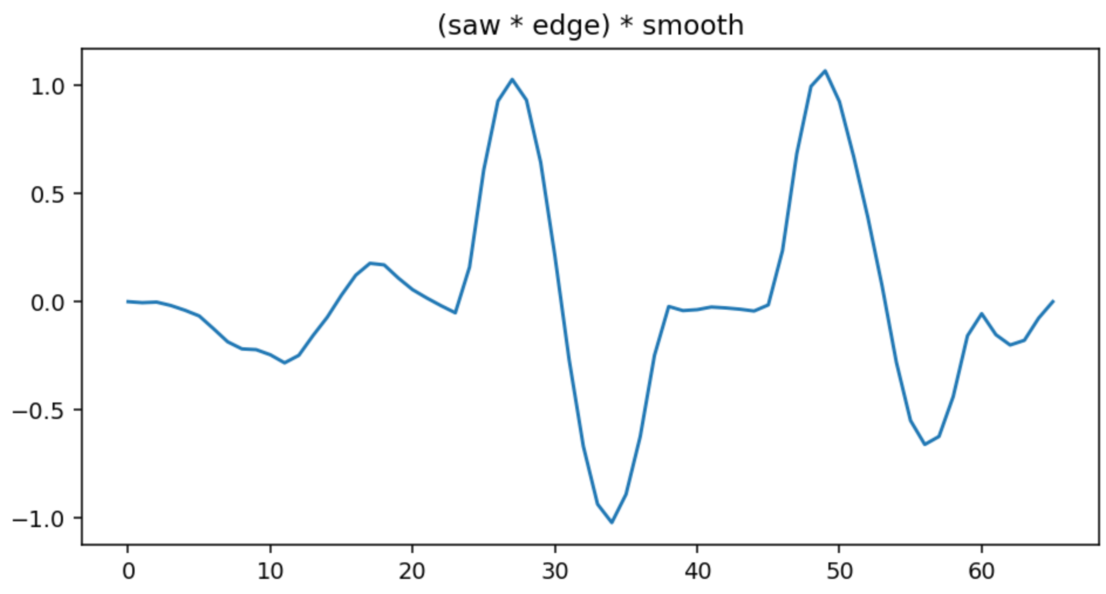
```
Out[29]:  Text(0.5,1,'Smoothing operator')
```

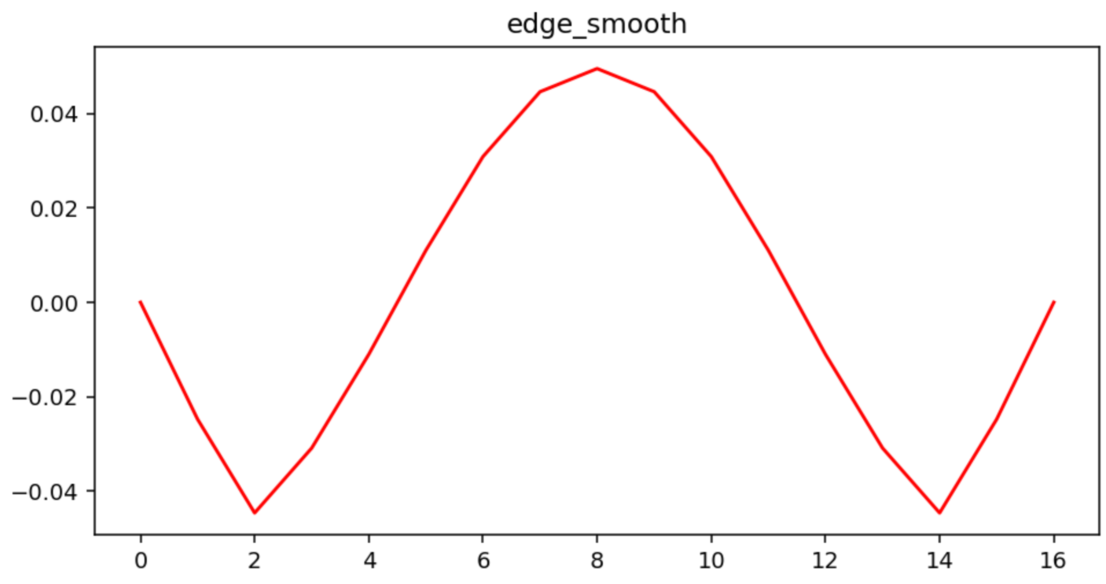


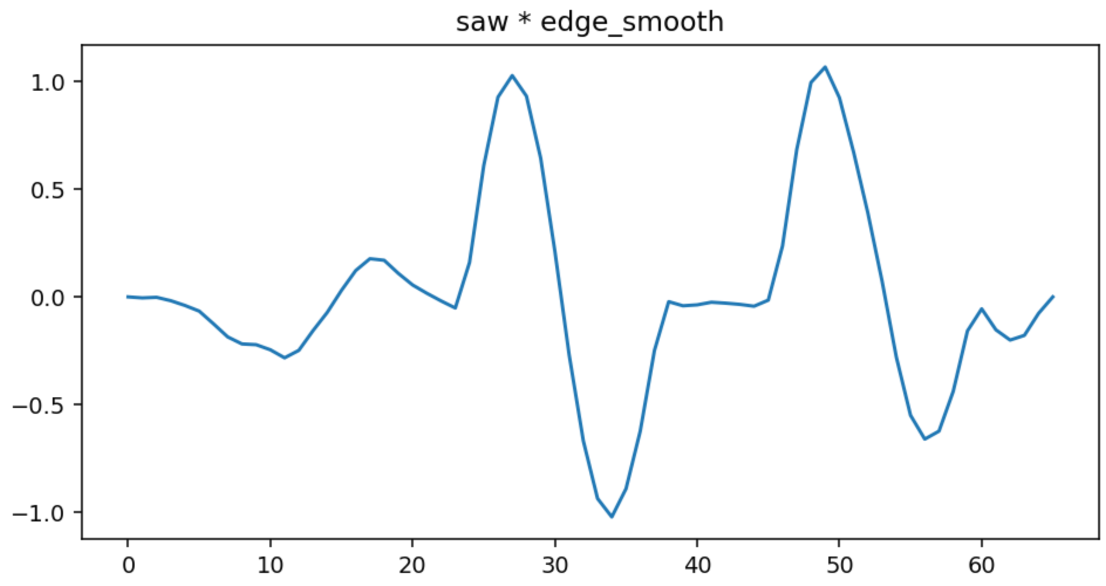


```
Out[30]:
         [<matplotlib.lines.Line2D at 0x23ef6716128>]
```


Noisy saw
8
6
4
2
0
0
10
20
30
40
50


(saw * edge) * smooth
1.0
0.5
0.0
−0.5
−1.0
0
10
20
30
40
50
60


edge_smooth
0.04
0.02
0.00
−0.02
−0.04
0
2
4
6
8
10
12
14
16



Simplest convolutions: dirac delta functions

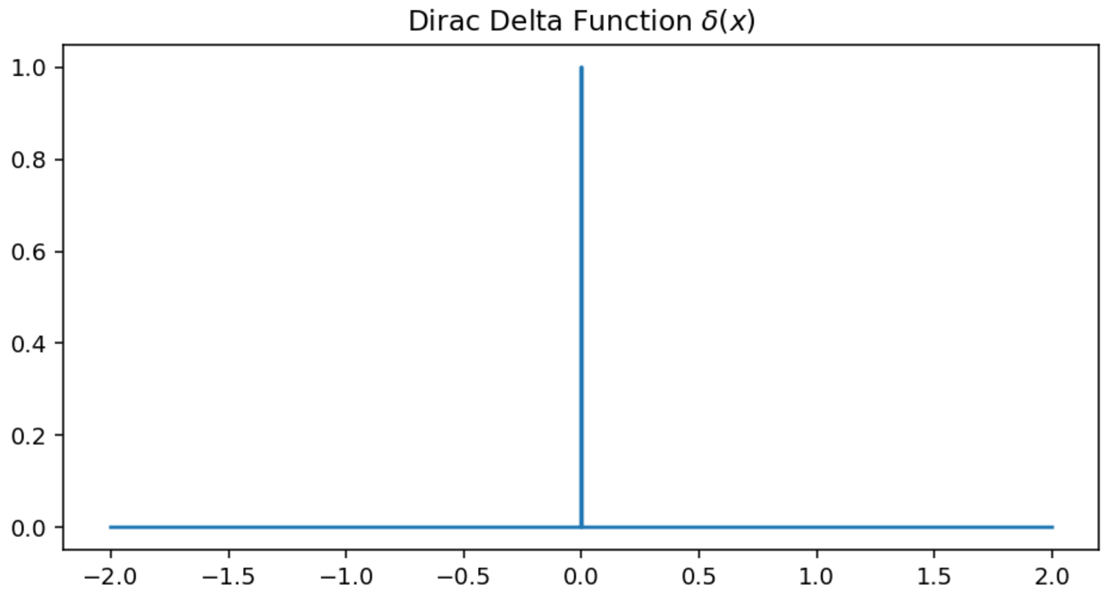
The Dirac **delta function** is a very useful tool in analysing system responses. It is a function that is zero everywhere except at 0, where it is 1. It satisfies:

$$\int_{-\infty}^{\infty} \delta(x) = 1$$

$$\delta(x) = \begin{cases} 0, x \neq 0 \\ 1, x = 0 \end{cases}$$

It is a perfect spike of height 1. Technically, it is not a proper function, but we can treat it like one for many purposes.

```
Out[31]:
         Text(0.5,1,'Dirac Delta Function $\\delta(x)$')
```



Convolutions with the delta function

The key property of the delta function is that:

$$f(x) * \delta(x) = f(x),$$

i.e. convolution of any function with the delta function **does not change the original function**. It is an *identity* element with respect to convolution.

Implication: if we can *stimulate* a system with the delta function, we can read back the convolution that represents the linear model of that system.

Of course, we don't have continuous functions, but discrete data. The discretised version of the delta function is just an array of zeros with a single 1; an **impulse**. If we feed a perfect impulse into a system we want to model, we can recover the convolution kernel. This is *linear system identification*.

There are two practical difficulties: producing a perfect impulse (i.e. the impulse itself must not be blurred or distorted); and dealing with noise in the observations. In the digital realm, these problems aren't directly relevant (we can always feed in noise-free perfect impulses), but are serious obstacles in measurements of physical systems.

Audio IR example

An interesting example for 1D convolutions is **impulse response recovery**. A vital part of sound is the **reverberation** (reverb) of the environment in which the sound is recorded; compare the sound a piano being played in a small room to in a concert hall -- same instrument, but very different sounds.

Reverberation is effectively a linear operation (the summation of lots of tiny echoes from the surroundings) and thus can *be represented as a convolution*. Simulating reverb is extremely important in creating life-like synthesised instruments, or blending together material from different sources (e.g. vocalist in isolation booth with the big band in the bandstand).

Impulse response recovery can extract the reverberation of a specific room or environment by apply a Dirac delta function in the audio domain. This is often a loud clap, balloon popping, or a starter pistol, which are close to perfect impulses. The recording of the sound of the room after the delta function sound is simply the recovered impulse response (IR); any sound can then be convolved with the IR to sound **exactly** like it was produced in that environment.

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-32-4b2f6d84a813> in <module>()
----> 1 playwav("res/ir_1_01.wav", time=2)
      2 playwav("res/ir_2_01.wav", time=4)
      3 playwav("res/ir_3_01.wav", time=1)

NameError: name 'playwav' is not defined
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-33-126b6e09aef5> in <module>()
----> 1 playwav("res/dry_1.wav")
      2 playwav("res/dry_2.wav")

NameError: name 'playwav' is not defined
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-34-facde80e11fd> in <module>()
----> 1 playwav("res/dry_1_ir_1.wav", time=4)
      2 playwav("res/dry_2_ir_1.wav", time=4)

NameError: name 'playwav' is not defined
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-35-b7fa6a1c171e> in <module>()
----> 1 playwav("res/dry_1_ir_2.wav", time=6)
      2 playwav("res/dry_2_ir_2.wav", time=5)

NameError: name 'playwav' is not defined
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-36-f0d24166313d> in <module>()
----> 1 playwav("res/dry_1_ir_3.wav", time=4)
      2 playwav("res/dry_2_ir_3.wav", time=4)

NameError: name 'playwav' is not defined
```

IIR filters are linear

It is possible to design filters with feedback (like the exponential smoother) which are exactly equivalent to convolutions but with *infinitely long convolution kernels*. The computation for these feedback filters -- called IIR or infinite impulse response -- can be very efficient indeed, and are a key component in audio and image processing.

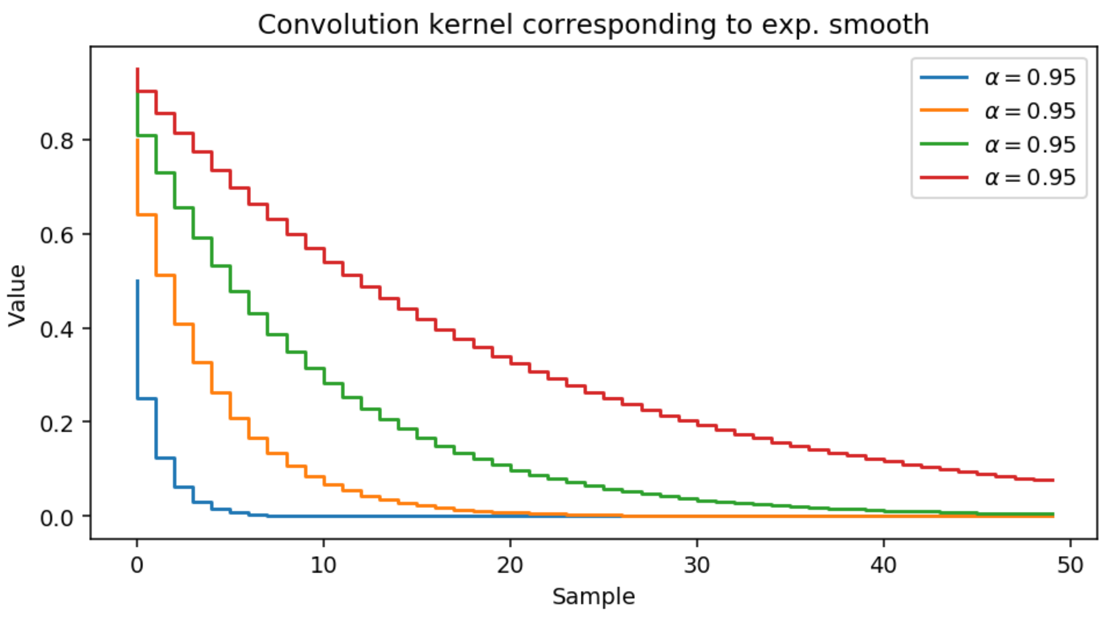
These feedback filters are still linear filters and all results relating to linear filters apply to them.

We can find the convolution kernel corresponding to the exponential smooth using the Dirac delta function approach: just pass an impulse through the process, and the output is the convolution kernel that simulates that process.

For example, the code below shows the impulse response of:

$$y[t] = \alpha y[t-1] + (1-\alpha)x[t]$$

```
Out[37]:
          <matplotlib.legend.Legend at 0x23efa0307b8>
```

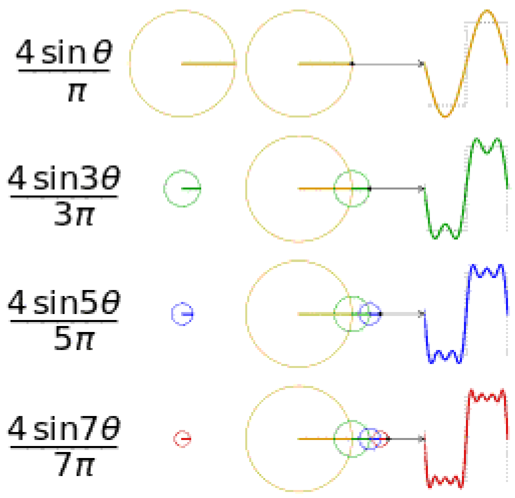


Frequency domain

We have used the term "frequency" loosely so far. We can precisely define what frequency is, and what concepts like having "lots of high frequency components" mean, and what the effect of filtering operations is.

We can view signals is two ways:

- as a sequence of amplitude measurements over time: **the time domain**
- as a sum of oscillations with different frequencies: **the frequency domain**



$\frac{4\sin\theta}{\pi}$
$\frac{4\sin 3\theta}{3\pi}$
$\frac{4\sin 5\theta}{5\pi}$
$\frac{4\sin 7\theta}{7\pi}$

/>

A more detailed Javascript toy: http://bl.ocks.org/jinroh/7524988 (http://bl.ocks.org/jinroh/7524988)

In this context, an pure oscillation is a **sine wave** $A\sin(2\pi\omega t + \theta)$. ω is the frequency of oscillation, θ is the **phase** of oscillation (offset to time) and A is the magnitude of the oscillation. When we talk about frequency, we are talking about the repetition period of a sinusoidal oscillation. The frequency domain representation represents signals as sums of oscillations at all possible frequencies, where each frequency has a phase (time offset) and magnitude (how large it is).

Definition

> A pure frequency is a sine wave with a specific period.

This corresponds to how our ears work; we hear sounds as a superposition of frequencies, where an isolated, pure frequency is the sound of a sine wave.

Fourier transform

Sine wave decomposition

The animation above shows how a "square wave" (a signal switching from full on to full off) can be approximated by the sum of sinsusoids. The **Fourier theorem** tells us that **any** repeating function can be decomposed into sine waves in this manner.

A sine wave has a single unique *frequency, amplitude and phase*:

$$f(x) = A\sin(\omega 2\pi t + \theta),$$

where A is the amplitude, ω is the frequency and θ is the phase. The Fourier transform lets us write **any** real signal as a sum of functions of this type. The phase is simply a shift of position of the sine wave. The Fourier transform is often presented from a complex analysis perspective, which isn't familiar to most computer scientists. We can instead approach it from a measure of similarity: the correlation between two signals.

Correlation between signals

Imagine we want to see "how alike" to signals $a[t]$ and $b[t]$ are. We could compute the elementwise product of a and b. When they are both the same sign, the product will be a large positive number; when they are opposite sign the product will be large negative number. We can sum this product over the whole length of the signals.

$$c = \sum_t a[t]b[t]$$

This is the (unnormalised) **correlation** between two signals.

- If the two signals are unrelated, $c \approx 0$, because the positive and negative products are equally likely and cancel each other out on average.
- if $a[t]$ and $b[t]$ are closely c will be large and positive
- if $a[t]$ and $b[t]$ are inverses (negatives) of each other c will be large and negative.

This lets us measure how similar two signals are.

Transform

Now, imagine we generate every *possible frequency* of sine wave as signals $a_1[t], a_2[t], a_3[t], \dots$ and correlate each with a test signal $b[t]$. For some $a[t]$ the response will be nearly zero, because the signals are unrelated. For others, it will be larger, because $b[t]$ has some oscillation at the test frequency.

This is the **amplitude** or **magnitude** of the response to that frequency. Now all we need is to work out how "shifted" the sine wave should be to line up best.

It turns out that all we need to do is compare with both a sine wave $a[t] = \sin(\omega x)$ and a cosine wave $a'[t] = \cos(\omega x) = \sin(\omega x + \frac{\pi}{2})$ to compute $c(\omega)$ and $c(\omega)'$ (the cosine wave version of $c(\omega)$) and we can work out the phase directly from the correlation with their value.

The phase is defined as the angle between these two values:

$$\theta = \tan^{-1}\left(\frac{c(\omega)}{c'(\omega)}\right)$$

To get the magnitude ignoring the phase, we can compute

$$A = \sqrt{c(\omega)^2 + (c'(\omega))^2}$$

We can build a little interactive tool to analyse this for simple signals:

Fourier transform equation

The Fourier transform allows us to write any (periodic) function $f(x)$ as an (infinite) sum of **sinusoids**; simple waves with distinct frequencies, amplitudes and phases.

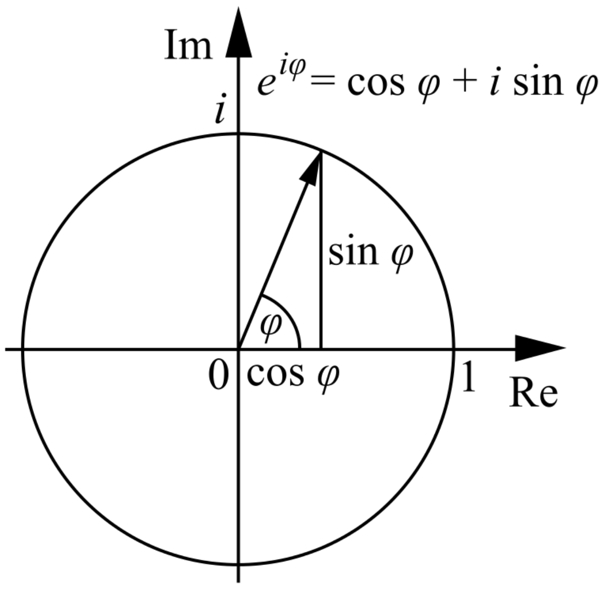
The Fourier transform is formally defined as:

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \omega} dx$$

It gives a *complex* value $\hat{f}(\omega)$ for each possible frequency ω.

Using Euler's identity for complex exponentials:

$$e^{2\pi i \theta} = \cos(2\pi\theta) + i\sin(2\pi\theta),$$



/>we can see this in terms of sums sine and cosine waves.

For **real signals** (i.e. all the ones we will deal with which have no imaginary part), we can deal with only the real part of the input signal, but we still have a complex output $\hat{f}(\omega)$.

Essentially, the Fourier transform "compares" a function with every possible frequency of sine and cosine wave, and returns how much of that frequency is present and what "phase" the sinusoidal wave is in. The sine part and the cosine part are returned as the real and imaginary components of this value.

$$\hat{f}(\omega)_{\text{real}} = \int_{-\infty}^{\infty} f(x)\cos(-2\pi x\omega) dx$$

$$\hat{f}(\omega)_{\text{imag}} = \int_{-\infty}^{\infty} f(x)\sin(-2\pi x\omega) dx$$

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(x)\sin(-2\pi x\omega) dx + i\int_{-\infty}^{\infty} f(x)\cos(-2\pi x\omega)$$

Inversion

Importantly, the Fourier transform can be **inverted**; a function decomposed into sinusoids can be reconstituted *exactly* into the original function by an almost identical process:

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\omega) e^{2\pi i x \omega} d\xi$$

This is the **inverse Fourier transform**. The Fourier transform is just *another way* of looking at a function. Instead of considering a value which varies in amplitude over time, we consider a value which varies in amplitude and phase over frequencies.

DFT

The Fourier transform applies to continuous functions $f(x)$. We need to work on discrete measurements, i.e. vectors of data $\x[t]$. The **discrete Fourier transform** is the discrete version of the Fourier transform for a vector of data $[x_0, \dots, x_{N-1}]$ is just:

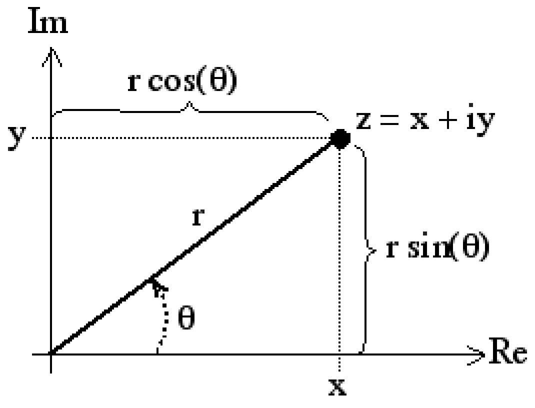
$$F[k] = \sum_{j=0}^{N-1} x[j]e^{-2\pi i \frac{j}{N}}$$

$$= \sum_{j=0}^{N-1} \left[x[j] \cos\left(-2\pi \frac{j}{N}\right) + ix[j] \sin\left(-2\pi \frac{j}{N}\right)\right]$$

for the k frequency components of x ($k = 0, 1, \dots, N-1$). The DFT has as many

Complex components: phase and magnitude

The result of the DFT is a sequence of **complex numbers** $F[k] = a + bi$. We can write any complex number as a magnitude A (length) and angle θ (direction), as in an **Argand diagram**.



/>

The angle θ is known as the **phase**, and the magnitude A is the **magnitude** of that component. The **frequency** of the component is given by the k index, and depends on the sampling rate. $X_0 = 0$, and $X_{N/2} = f_n$, where f_n is the **Nyquist rate**, half the original sampling rate. So for the kth component, the real frequency in terms of the original signal is:

$$\text{freq} = f_N k/N$$

This gives us an amplitude and phase for each component between 0 and f_N in evenly spaced subdivisions, with as many subdivisions as there were elements of $x[t]$.

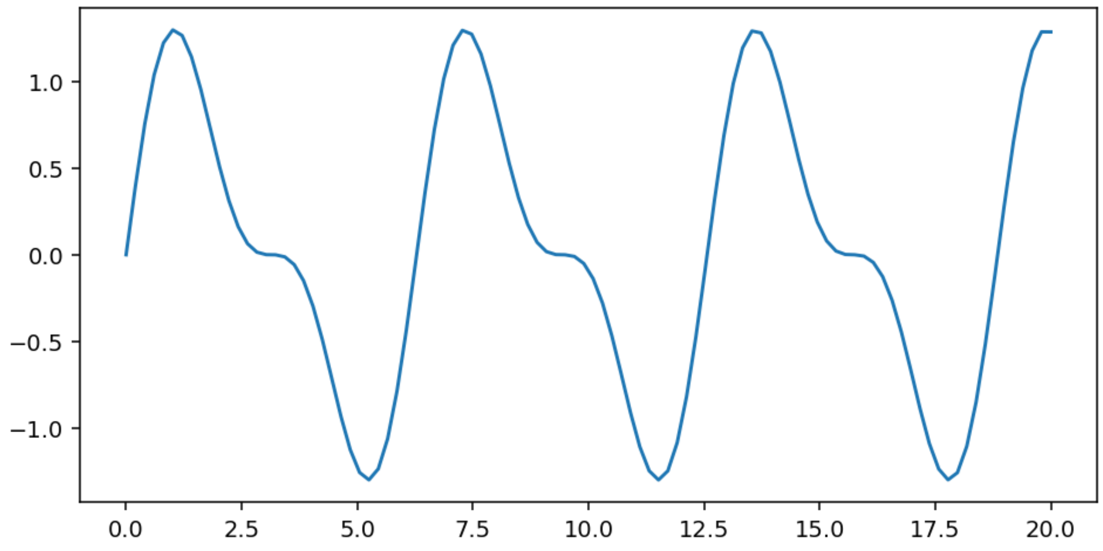
The DFT is trivial to implement in code:

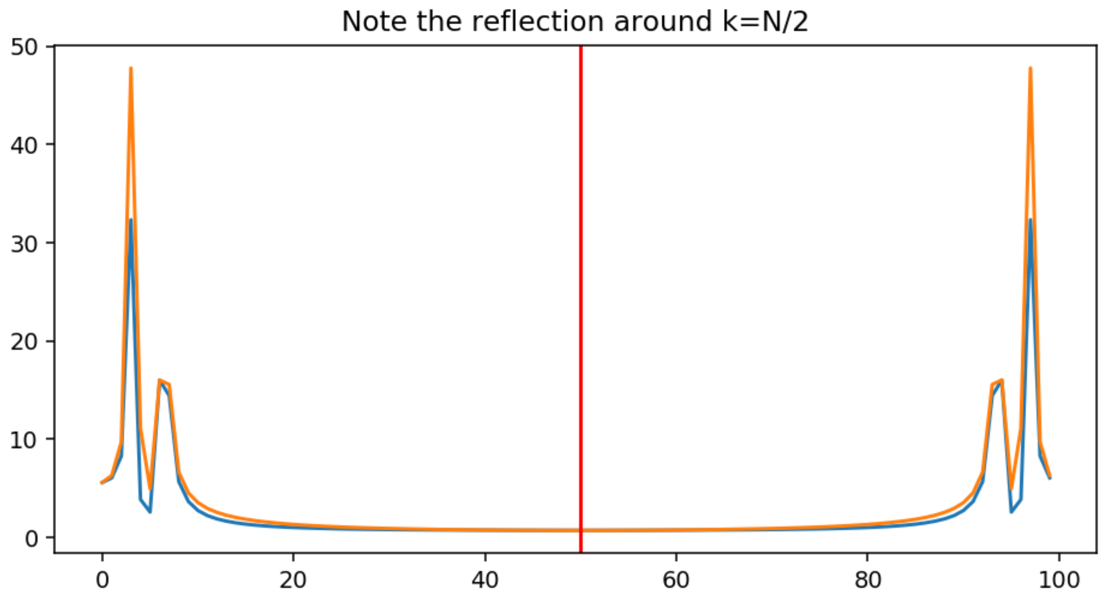
We can plot some examples:

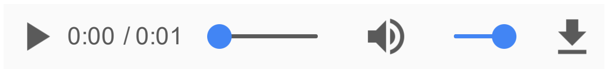
```
c:\conda3\lib\site-packages\ipykernel_launcher.py:22: Comp
lexWarning: Casting complex values to real discards the im
aginary part
```

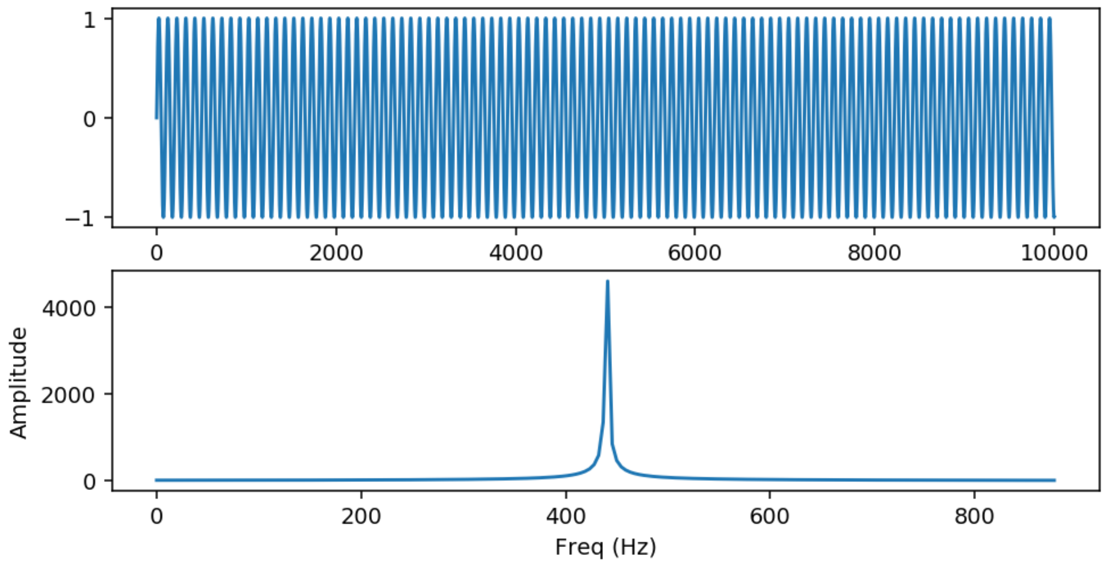
Out[41]:

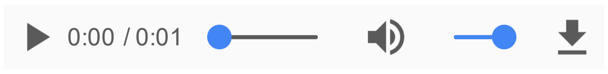
```
Text(0.5,1,'Note the reflection around k=N/2')
```



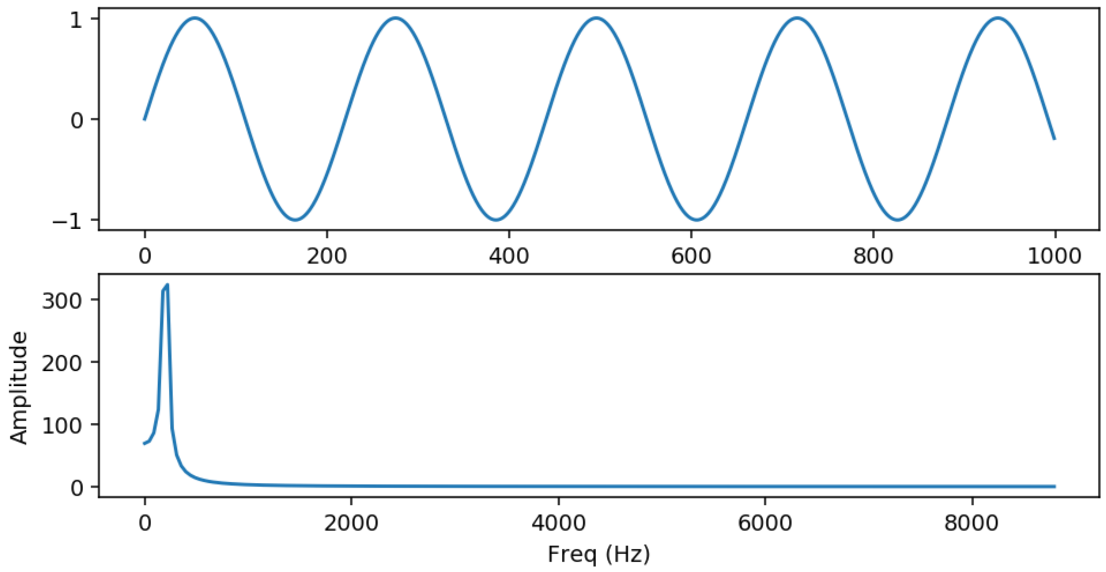


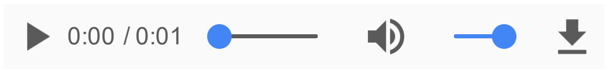

0:00 / 0:01


1
0
−1
0
2000
4000
6000
8000
10000
Amplitude
4000
2000
0
0
200
400
600
800
Freq (Hz)

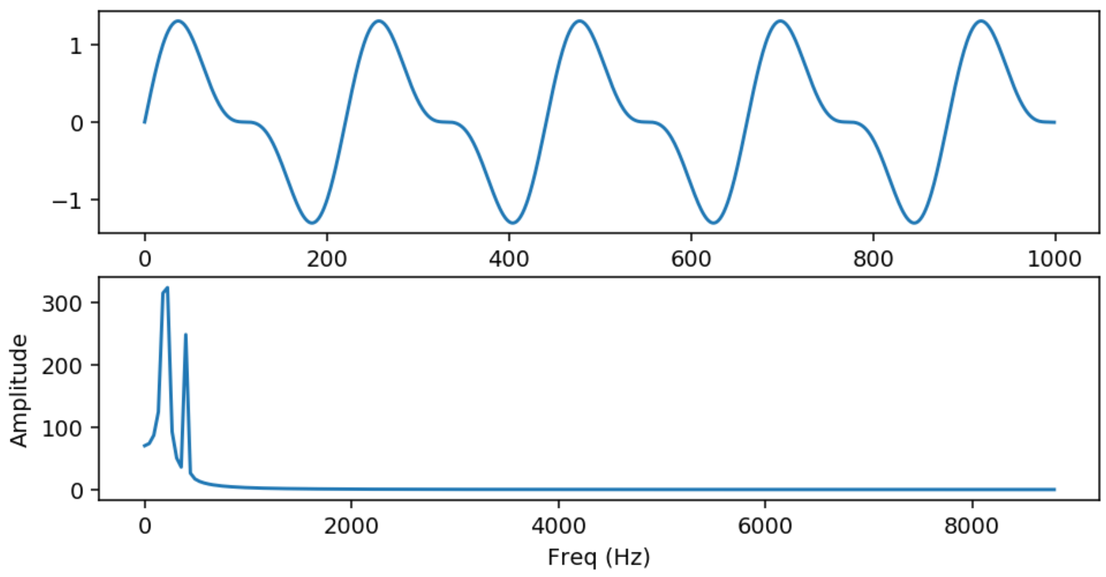

0:00 / 0:01

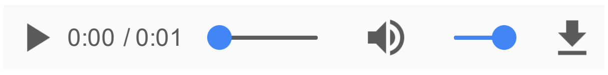
```
<matplotlib.figure.Figure at 0x23ef03adcf8>
```


1
0
−1
0
200
400
600
800
1000
Amplitude
300
200
100
0
0
2000
4000
6000
8000
Freq (Hz)

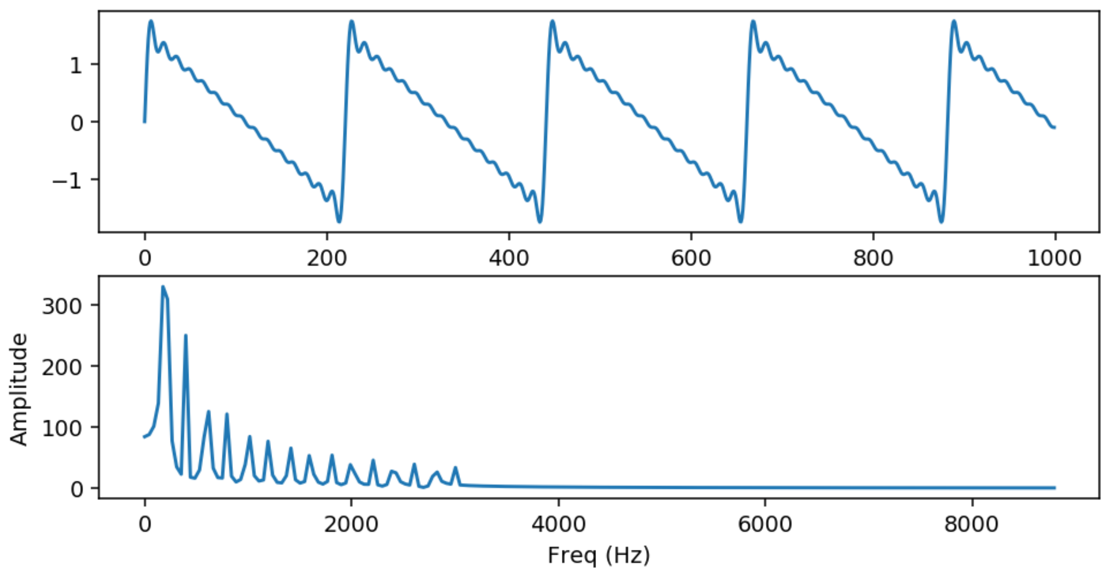


```
<matplotlib.figure.Figure at 0x23ef06fb128>
```





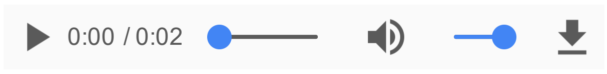
```
<matplotlib.figure.Figure at 0x23ef071f5f8>
```



Reconstruction of a signal

We can perform an FFT, decompose into components, and then reconstruct with the k most important (loudest) frequency components.

```
c:\conda3\lib\site-packages\scipy\io\wavfile.py:273: WavFi
leWarning: Chunk (non-data) not understood, skipping it.
  WavFileWarning)
```



0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

0:00 / 0:02

Out[49]:
```
[None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None]
```

FFT

The DFT is very expensive to compute ($O(N^2)$ with lots of expensive floating point operations like exponentation). Luckily, there is a **much** faster algorithm, which uses divide-and-conquer to speed up the operation. This is the **fast Fourier transform (FFT)**, developed at IBM in the 1960s. The FFT runs in $O(N \log(N))$ time in the best case.

The FFT only runs in $O(N \log(N))$ time if N is a power of two (because the standard FFT splits the signal recursively into two) and is $O(N^2)$ for non-power-of-2 inputs. Variations of the FFT run in $O(N \log(N))$ for vectors with length which is highly composite, but still $O(N^2)$ for prime N. There are very, very optimised FFT libraries which worry about all of this for you.

The convolution theorem

There is a very important property of convolutions which explains why we took this tangent through the Fourier transform.

The **convolution theorem** states that the Fourier transform of the convolution of two signals is equal to the (elementwise-)product of the Fourier transform of two signals.

$$\mathrm{FT}(f(x) * g(x)) = \mathrm{FT}(f(x))\mathrm{FT}(g(x))$$

This means we can compute the convolution by taking products in the frequency/spatial frequency domain, and then transforming back to the time or spatial domain.

$$f(x) * g(x) = \mathrm{IFT}[\mathrm{FT}(f(x))\mathrm{FT}(g(x))]$$

Convolution interchanges with elementwise-multiplication through the Fourier transform. These two operations (convolution and multiplication) are intimately linked, and the Fourier transform lets us see this relationship.

Frequency domain effects of filtering

Time domain convolution clearly affects the frequency spatial domain; the convolution theorem tells us exactly how. We can use this to **analyse** the effect of filters before we apply them to images. For example, if we want to downsize an image by some ratio r, we must ensure that there are no frequency components $\frac{f_n}{r}$ before dropping pixels, or we will **alias**. This means that the prefiltering kernel K must have $DFT(K)$ so that it has zero magnitude for all frequencies $|DFT(K)| = 0, > \frac{f_n}{r}$.

We can apply any frequency-based filter in the frequency domain simply by multiplying; but having to take the DFT of the entire image can be much slower and much more memory intensive than a small convolution -- the opposite is true for large convolutions.

Filter types

There are several common types of filter, named for the frequency domain effect that they have.

- A **smoothing filter** or **lowpass filter** reduces high frequencies.
- A **highpass filter** reduces low frequencies.
- A **bandpass filter** reduces frequencies outside of a certain band.
- A **notch filter** or **bandstop filter** reduces frequencies inside a certain band.

Although there are nonlinear versions of these filters, most filters that are in use for processing signals are linear filters implemented with a convolution (or as an IIR filter equivalent to a convolution).

Linear filters in the frequency domain

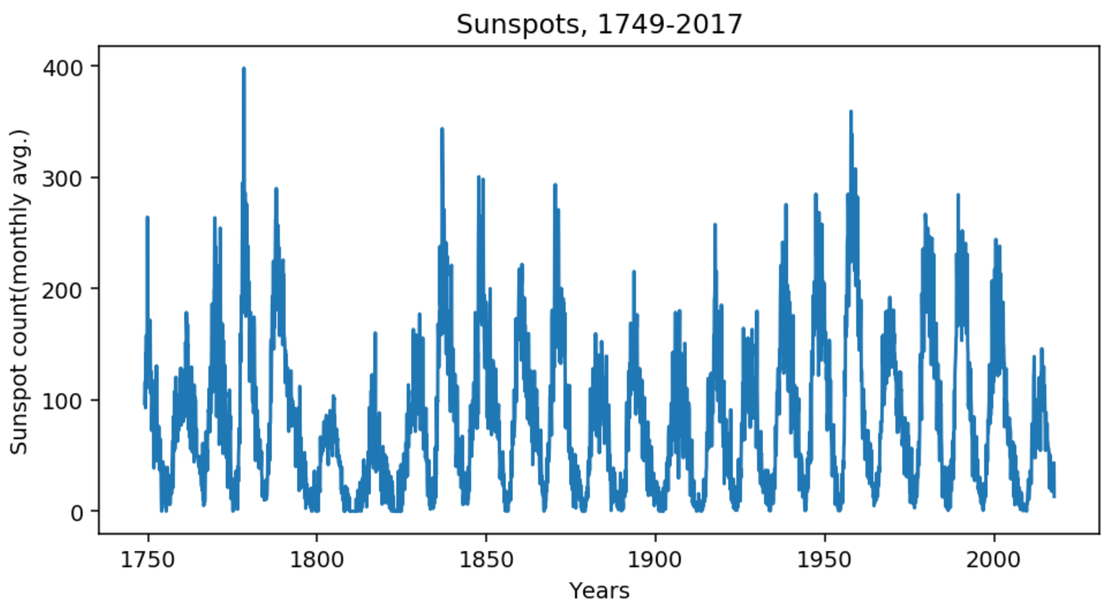
A **linear filter** can change the amplitude of frequency components -- but it can **never** introduce new frequencies. If a frequency is not present in a signal, no linear filter can make it appear. Only nonlinear filters can introduce new frequencies.

Proof: $FT(f * g) = FT(f)FT(g)$, so no matter what we make g, if any element of $FT(f)$ is zero, it will always stay zero.

An example: Sunspots

We saw the sunspot data way back in Unit 1. This is a count of the sunspots visible on the sun. Astronomers have tracked this for centuries, and have kept detailed records of the number of spots visible. This is a measure of the activity of processes within the sun, and it varies over time, in a way that appears to oscillate with a regular rhythm.

```
Out[50]:
          Text(0.5,1,'Sunspots, 1749-2017')
```



Sunshine tunes

We can listen to the sound of the sun:

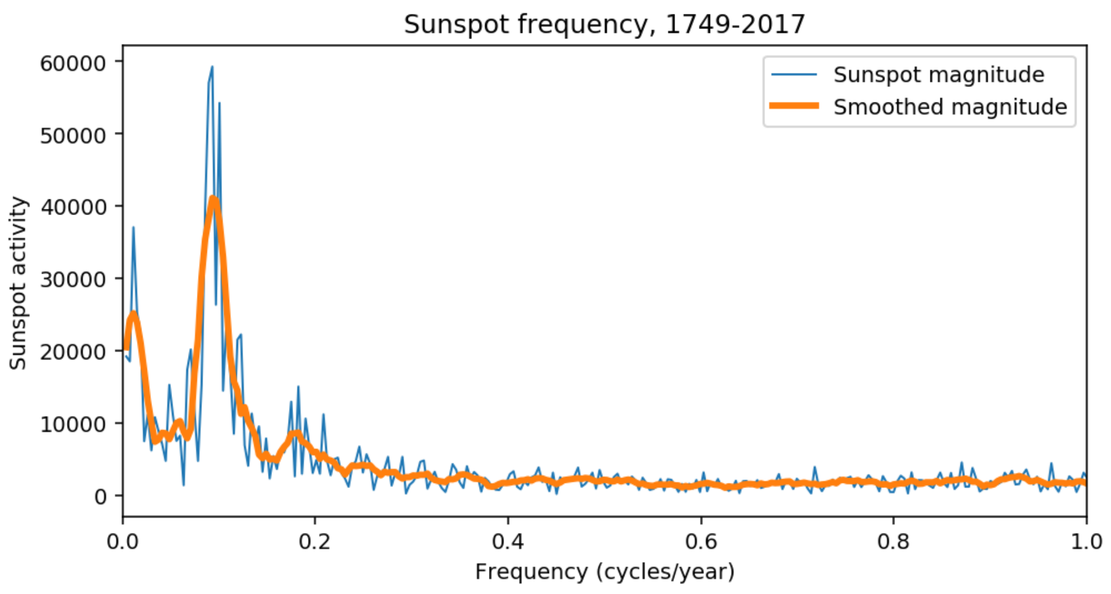
0:00 / 0:03

0:00 / 0:14

This looks and sounds like it is oscillating in some way. Let's compute the DFT, and plot the resulting magnitude spectrum:

Out[52]:

```
Text(0.5,1,'Sunspot frequency, 1749-2017')
```

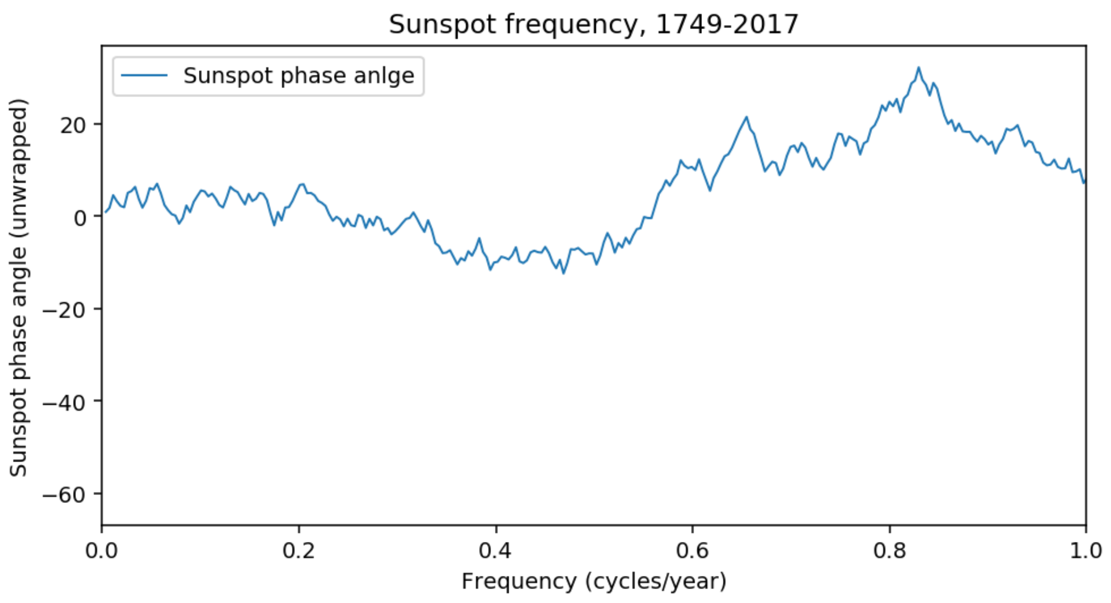


Phase plot

Every frequency has an amplitude (magnitude) and also a phase. We can plot this, but it isn't very helpful here. This tells us how "shifted" each oscillation is in the time domain, which isn't particularly interesting for the sunspot data.

Out[53]:

```
Text(0.5,1,'Sunspot frequency, 1749-2017')
```

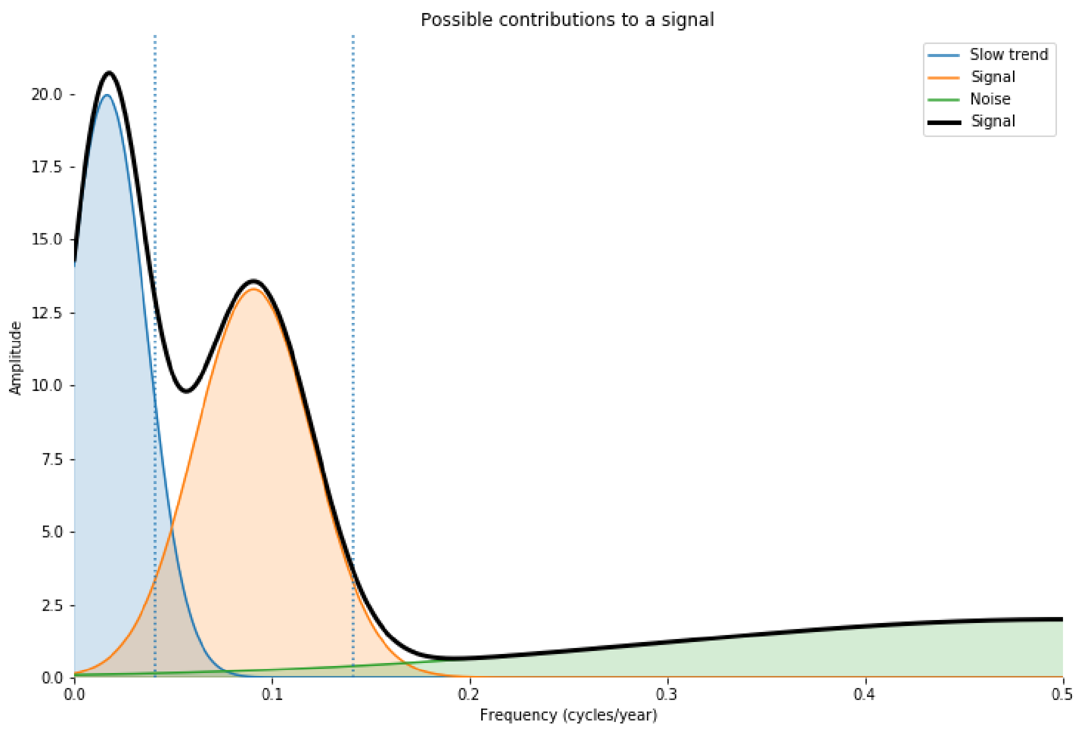


Analysing from a frequency perspective

We might assume that this signal is made up of many processes summed together.

- Some of this is random fluctuations, which vary from day to day; high-frequency noise
- Some part of it might be some long term trend (the slow death of the sun, for example); low-frequency oscillations
- Some part of it, however, might be interesting signal; the oscillation of the nuclear process within the sun.

These different components would occupy different parts of the frequency spectrum.



/>

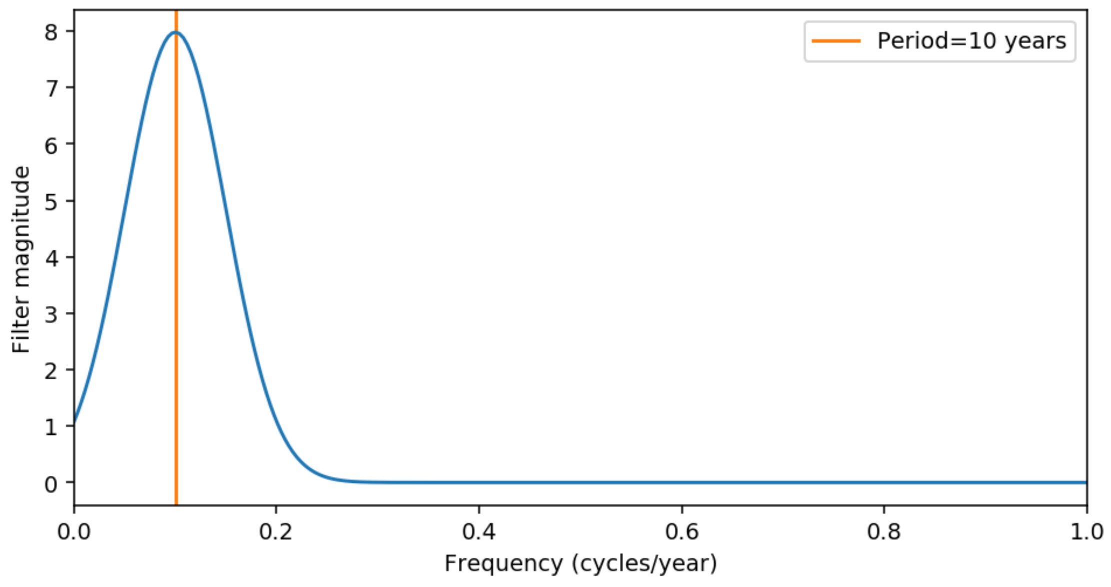
In this case, there is some kind of peak around about 0.1 cycles/year, or about every 10 years. This is visible on the magnitude spectrum plot.

By the convolution theorem, we can design a filter to select only these frequencies. We just define a mask in frequency space, convert to time domain using the inverse Fourier transform and convolve.

(we could actually just multiply in the frequency domain, then invert).

Here, we'll create a simple function in the frequency domain that will select the frequencies we are interested in, and mask out the remainder. A Gaussian function (used for the PDF of a normal distribution) is an effective mask to use, as it is nice and smooth.

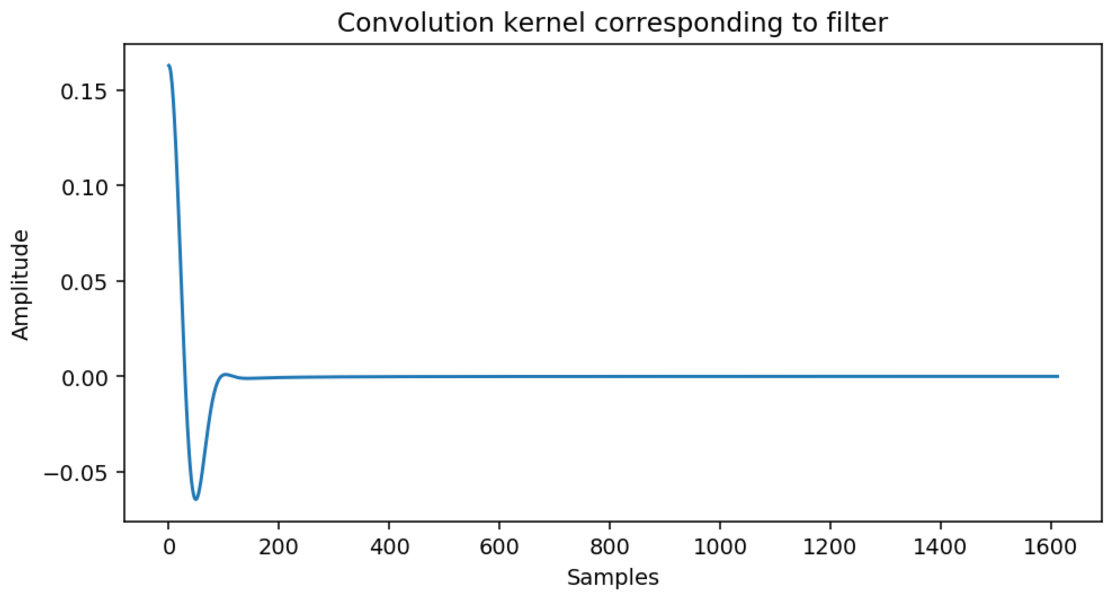
```
Out[54]:
         [<matplotlib.lines.Line2D at 0x23ef67736a0>]
```



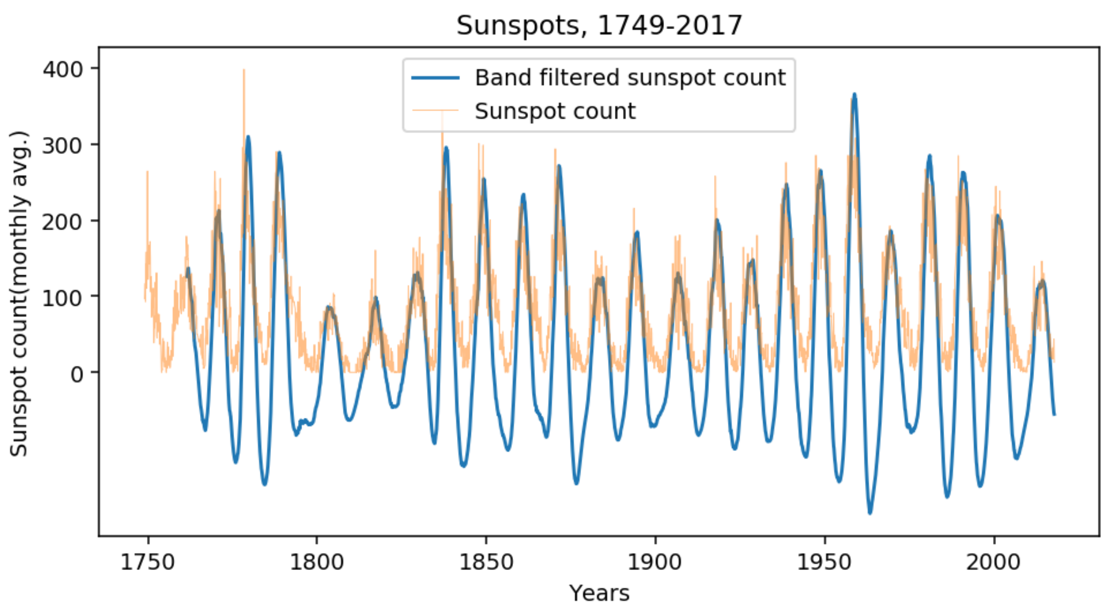
We can now take this frequency domain representation, and convert it to the time domain using the **inverse Fourier transform**, which will give us a **convolution kernel** we can apply to the signal.

Technically, we also need to specify the phase for each frequency as well as the magnitude. We can assume that we want a phase of zero for each frequency to make things simpler (there are better choices).

```
Out[55]:
         Text(0,0.5,'Amplitude')
```



```
Out[56]:
         Text(0.5,1,'Sunspots, 1749-2017')
```



Filter design

There is an enormous literature of **filter design** tools that can construct convolution kernels from a specified frequency properties (e.g. keep frequencies below some threshold, keep all frequencies in some band, and so on). We have seen the Fourier transform lets us design and analyse convolution kernels, but most design tools don't use the frequency domain directly.

These algorithms usually use iterative optimisation to find kernels that satisfy frequency domain requirements, in terms of expected magnitude effects and also phase shift effects. These are often used to design IIR filters (those with feedback, equivalent to convolution with an infinite kernel) with specific properties.

Resources

- **Sampling, Quantization and Encoding** http://kaushanim.blogspot.co.uk/2010/03/sampling-quantization-encoding.html (http://kaushanim.blogspot.co.uk/2010/03/sampling-quantization-encoding.html) (short introduction to sampling and quantization)
- **DSP for the Braindead** http://yehar.com/blog/?p=121 (http://yehar.com/blog/?p=121) (not actually for the braindead, in fact much more advanced than we cover here!)

Beyond this course

- **The Scientist and Engineer's Guide to Signal Processing** http://dspguide.com/ (http://dspguide.com/) (free, online book)
- **Digital Signal Processing: A Computer Science Perspective** Jonathan (Y) Stein (outrageously expensive, but an excellent coverage of the topics)