Exploring Aliasing and the Sampling Theorem

Margherita Tonon

April 2025

1 Introduction

2 Background

In signal processing, two types of signals exist: analog signals and digital signals. An analog signal refers to a signal that varies continuously over time. The complexity of analog signal processing, their susceptibility to noise and signal degradation over time, as well as their limited reproductibility and scalability makes them inconvenient to work with in practice. Therefore, digital signals are used – signals that vary discretely over time and can take only a finite number of distinct values.

Sampling refers to the process of converting an analog signal into a digital signal. If we let $x(t)$ be a continuous time signal, the sampled signal $x[n]$ is defined as

$$x[n] = x(nT_s)$$

where n represents discrete time sampling points and T_s represents the sampling period, such that the sampling frequency $f_s = \frac{1}{T_s}$.

Sampling can also be represented as

$$x_s(t) = x(t) \sum_{-\infty}^{\infty} \delta(t - nT_s)$$

where $x_s(t)$ is the sampled points, and δ is the dirac delta distribution, taking value 1 if $t = nT_s$, and 0 otherwise. Therefore, it is clear that when $t = nT_s$, $x_s(t) = x(t)$; otherwise, $x_s(t) = 0$.

In practice, sampling allows us to discretize a continuous input, facilitating the handling of signals. After sampling is done, it is natural that the signal must be reconstructed in order to recover the original (time continuous) signal. However, recovery is not always perfect – the Shannon-Nyquist condition must be met.

The Shannon-Nyquist theorem states that a signal can be perfectly reconstructed if the sampling frequency is greater than two times the maximum frequency B of the continuous signal:

$$f_s > 2B$$

For example, if we consider a sine wave with frequency 5 Hertz, $\sin(2\pi \cdot 5 \cdot t)$, once sampled it can be perfectly reconstructed if the sampling frequency is greater than 10.

In the time domain, a sampled signal looks like discrete spikes (Figure 1). The Fourier Transform allows us to transition from the time domain into the frequency domain, and therefore, to visualize what sampling looks like in the frequency domain, we make use of the Inverse Convolution Theorem.

The Inverse Convolution Theorem states that if we have two functions $x(t)$ and $h(t)$ whose Fourier Transforms $\mathcal{F}(x(t))$ and $\mathcal{F}(h(t))$ are absolutely integrable in the frequency domain, then

$$\mathcal{F}^{-1}\left(X(f) * H(f)\right) = x(t) \cdot h(t)$$

where $*$ is the operation which represents a convolution. Consequently,

$$X(f) * H(f) = \mathcal{F}\left(x(t) \cdot h(t)\right)$$

Therefore, because we can express sampling as $x_s(t) = x(t) \cdot \sum_{-\infty}^{\infty} \delta(t - nT_s)$ in the time domain, the Inverse Convolution Theorem tells us that

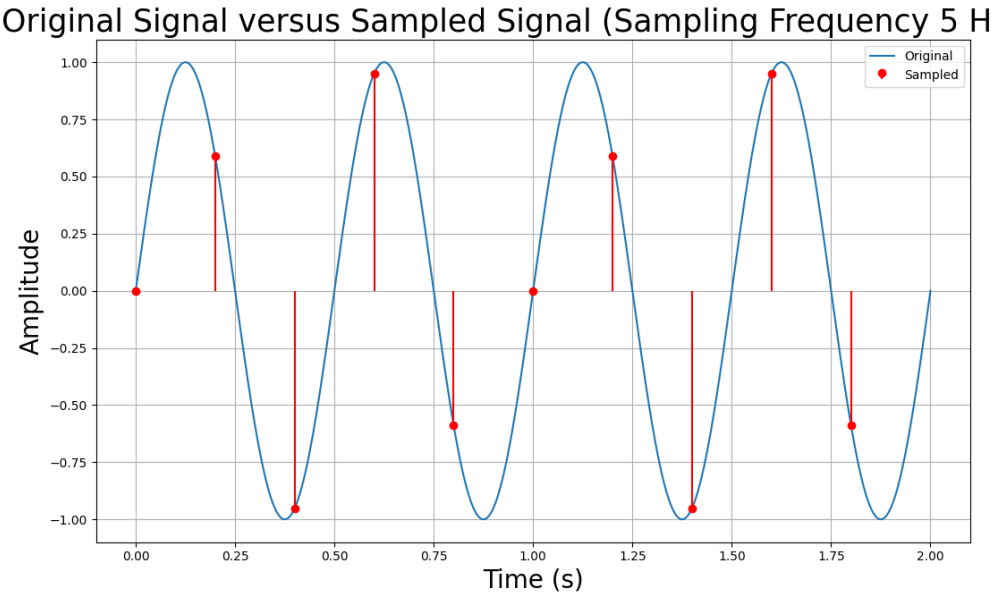

Figure 1: Representation of sampling of $\sin(4\pi t)$ in the time domain, sampled at a frequency of 5 Hertz

$$\mathcal{F}\left(x(t)\cdot\sum_{-\infty}^{\infty}\delta(t-nT_s)\right)=\mathcal{F}(x(t))*\mathcal{F}\left(\sum_{-\infty}^{\infty}\delta(t-nT_s)\right)$$

To visualize sampling in the frequency domain we must therefore convolve the Fourier Transform of the input signal with the Fourier Transform of $\sum_{-\infty}^{\infty}\delta(t-nT_s)$, known as the Dirac comb. Using properties of the Fourier Transform, Dirac Delta distribution, ___, and the Poisson summation formula, we obtain that

$$\mathcal{F}\left(\sum_{-\infty}^{\infty}\delta(t-nT_s)\right)=\frac{1}{T_s}\sum_{n=-\infty}^{\infty}\delta\left(f-\frac{n}{T_s}\right)$$

We therefore have

$$\mathcal{F}(x(t))*\mathcal{F}\left(\sum_{-\infty}^{\infty}\delta(t-nT_s)\right)=X(f)*\frac{1}{T_s}\sum_{n=-\infty}^{\infty}\delta\left(f-\frac{n}{T_s}\right)$$

Convolution of a function with a delta function shifts the function by the shift factor, and therefore

$$\mathcal{F}(x(t))*\mathcal{F}\left(\sum_{-\infty}^{\infty}\delta(t-nT_s)\right)=\frac{1}{T_s}X\left(f-\frac{n}{T_s}\right)$$

We can consequently observe how the representation of a sampled signal in the frequency domain is simply the Fourier Transform of the signal, duplicated and shifted to multiples of the sampling frequency.

To recover the original signal from the sampled signal, we apply an ideal band pass filter,

$$H(f)=$$

which selects the frequencies between f_l and f_h Hertz. We then perform an inverse Fourier Transform on the signal and we will have recovered the original signal.

Nonetheless, the frequency domain representation of the sampled signal highlights the need for the Shannon-Nyquist theorem. When our sampling frequency is "large enough", the process of copying and shifting the frequencies occurs at larger intervals, meaning the copied frequencies are more spread out. It is therefore possible to apply an ideal filter to recover the original frequencies. However, if the sampling frequency decreases, the interval between frequencies is not as large and they may overlap. It is therefore no longer possible to apply an ideal filter to recover the original frequencies, as the original signal and the duplicated frequencies overlap. This is known as aliasing: the act of different frequency components becoming indistinguishable in the sampled signal due to overlapping spectra. As long as we set $f_s > 2B$, however, the signal is able to be perfectly recovered.

3 Implementation

To implement the process of sampling in Python, I begin by defining a functionn `generate_signal` which generates a sine wave (a continuous-time signal) of desired frequency.

```
def generate_signal(t_end, frequency, plot = False):
    """
    Generates a sine wave from t = 0 to t=t_end with frequency defined by the frequency parame
    If plot = True, plots the generated sine wave.
    Returns the time array t and the x(t) sinusoid.
    """
    t = np.linspace(0, t_end, 10000)
    x = np.sin(2 * np.pi * frequency * t)
    if plot == True:
        plt.plot(t, x)
        plt.title(f"Continuous Time Signal - Sine Wave With Frequency {frequency} Hertz")
        plt.xlabel("Time")
        plt.ylabel("Amplitude")
        plt.grid(True)
        plt.show()
    return t, x
```

Here, I simply create a time array with 10000 elements, and then apply the sine function with the given frequency to the 10000 time values. Users can set the "plot" parameter to "True" if they wish a plot of the original continuous-time signal to be displayed.

Next, I define the "sample_signal" function, which samples the continuous signal at a specified sampling frequency. Users can set "plot_one" to "True" if they wish to visualize the original signal together with the sampled points, and "plot_two" to True if they wish to just visualize the sampled points.

Thirdly, I create the "sampled_fourier_transform" function, which takes as inputs the sampled amplitudes and the sampling frequency, and performs the Fast Fourier Transform of the sampled amplitudes using the "scipy" module. Users can again set the plot parameter to True if they wish to observe the frequencies of the sampled signal in the frequency domain.

Finally, I define the "reconstruction" function, which

The sinc function, defined as $\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$, is utilized as it is the time domain representation of the ideal band pass filter. The sinc function is obtained by taking the inverse Fourier Transform of the ideal band pass filter:

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{2\pi i f t} df$$

$H(f)$ is even and real and therefore $e^{2\pi i f t} = \cos(2\pi f t) + i \sin(2\pi f t) = \cos(2\pi f t)$ as the imaginary part is 0.

$$h(t) = \int_{-\infty}^{\infty} H(f) \cos(2\pi f t) df$$

Evaluating this integral,

$$h(t) = \frac{1}{\pi t} \left[\sin(2\pi f_h t) - \sin(2\pi f_l t)\right]$$

This is the difference of two sinc-like functions. The signal $x(t)$ can therefore be recovered in the following way:

$$x(t) = \sum_{-\infty}^{\infty} x(nT) \text{sinc}\left(\frac{t}{T} - n\right)$$

which is the approach implemented in the "reconstruction" function.

4 Discussion

5 Conclusion

References

[1] https://realpython.com/python-scipy-fft/

6 Appendix

1. Access the GitHub repository with all code here.