ELEC 421
Digital Signal and Image Processing

Siamak Najarian, Ph.D., P.Eng.,

Professor of Biomedical Engineering (retired),

Electrical and Computer Engineering Department,

University of British Columbia

Course Roadmap for DSP

Lecture	Title
Lecture 0	Introduction to DSP and DIP
Lecture 1	Signals
Lecture 2	Linear Time-Invariant System
Lecture 3	Convolution and its Properties
Lecture 4	The Fourier Series
Lecture 5	The Fourier Transform
Lecture 6	Frequency Response
Lecture 7	Discrete-Time Fourier Transform
Lecture 8	Introduction to the z-Transform
Lecture 9	Inverse z-Transform; Poles and Zeros
Lecture 10	The Discrete Fourier Transform
Lecture 11	Radix-2 Fast Fourier Transforms
Lecture 12	The Cooley-Tukey and Good-Thomas FFTs
Lecture 13	The Sampling Theorem
Lecture 14	Continuous-Time Filtering with Digital Systems; Upsampling and Downsampling
Lecture 15	MATLAB Implementation of Filter Design

Lecture 14:
Continuous-Time Filtering with Digital Systems; Upsampling and Downsampling

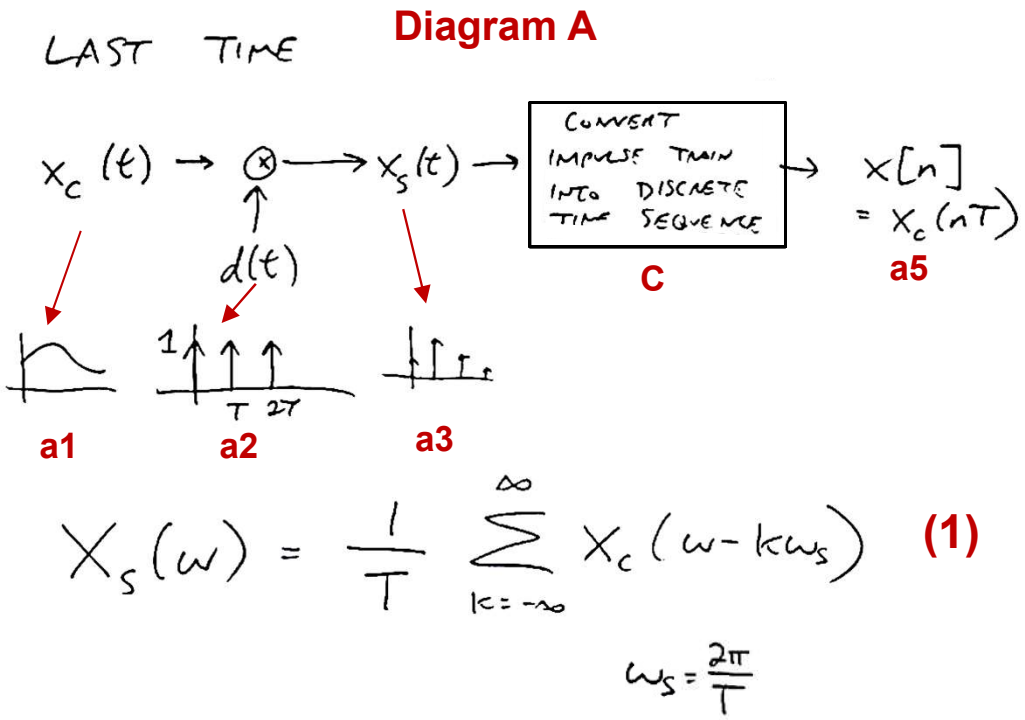
Table of Contents

- Review of sampling and reconstruction
- How copies appear in the CTFT vs. the DTFT
- Discrete-time processing of continuous-time signals
- For a given sampling rate, how should the middle discrete-time system be chosen?
- The effective continuous-time frequency response
- $H_{eff}(\omega)$, effective frequency response in the continuous-time world
- Cutoffs in discrete vs. continuous time
- How are the impulse responses related?
- Changing the sampling rate
- Downsampling by an integer factor
- Upsampling by an integer factor
- Ideal reconstruction of the missing samples via lowpass filtering
- Time-domain interpolation
- Upsampling with Interpolation: summary and conclusion
- Multirate Signal Processing
- Polyphase representations
- Wavelet analysis
- Use of stochastic processes in DSP

Review of sampling and reconstruction

- Earlier, we talked about sampling theorem and the relationship between what will be the Fourier transform of a continuous time signal and the Fourier transform of a signal that has been sampled. Also, we discussed how that could go wrong, and how aliasing can occur if we do not sample fast enough.
- In this lecture, we want to talk about two topics. First, if we wanted to design a certain continuous time filter, how would we design an equivalent filter in discrete time? Second, we want to talk about how do we do changing of the sampling rate? So, suppose that we sampled a signal at some rate, and now after the fact, we want to change the sampling rate to something else. This kind of **sampling rate conversion** happens a lot in real world systems, where one system is expecting data at 10 kilohertz and the other system is expected at 12 kilohertz. And, the question is how do we make them talk to each other?

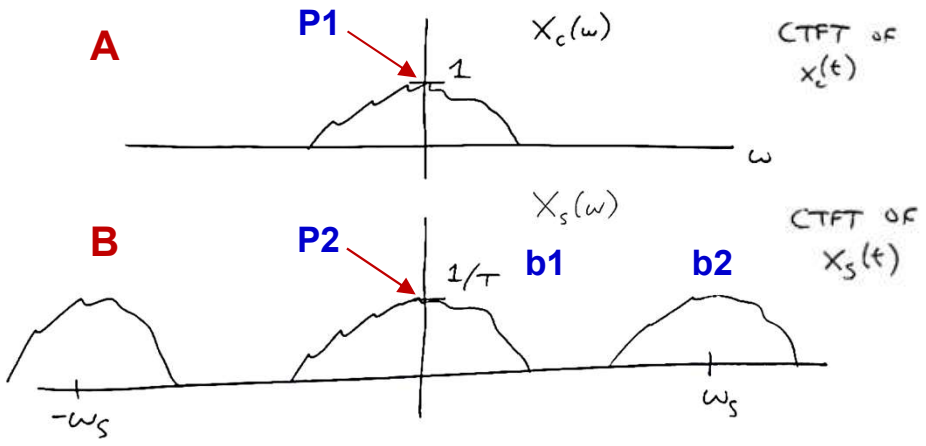
Review of sampling and reconstruction



- Last time, we talked about the operation in which we have a continuous time signal, $\mathbf{x_c(t)}$, we multiply that by **d(t)**, which is basically a set of delta functions that occurs every **T** units, and we get some sort of a continuous time sampled signal, $\mathbf{x_s(t)}$. This is shown in **Diagram A**.
- So, basically, what we are doing is taking the original signal, **a1**, multiplying it by **a2**, and then getting **a3**, which is the sampled impulses. And then, in practice, we would convert this impulse train, $\mathbf{x_s(t)}$, shown in block **C**, into discrete time sequence, **x[n]**. Here, under the hood, **x[n]** is really an approximation of $\mathbf{x_c(nT)}$, which means it is taking every **T** sample of the continuous time signal.
- Also, we showed that the Fourier transform of the impulse train sampled signal, $\mathbf{x_s(t)}$ or **a3**, was equal to the $\mathbf{X_s(\omega)}$, which is the formula shown in **(1)**. This formula means is that we get a bunch of copies of the signal, $\mathbf{X_c(\omega)}$, and those copies are centred at multiples of sampling frequency, $\mathbf{k\omega_s}$, and that they are **1/T** high as things used to be.

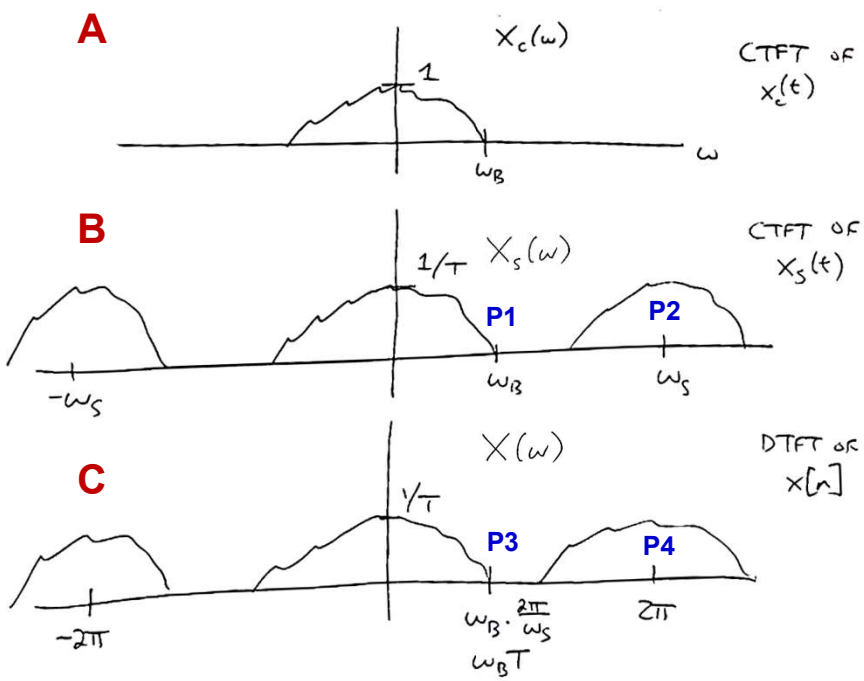
How copies appear in the CTFT vs. the DTFT

$$X_s(\omega) = \frac{1}{T}\sum_{k=-\infty}^{\infty} X_c(\omega - k\omega_s) \qquad \omega_s = \frac{2\pi}{T} \qquad (1)$$



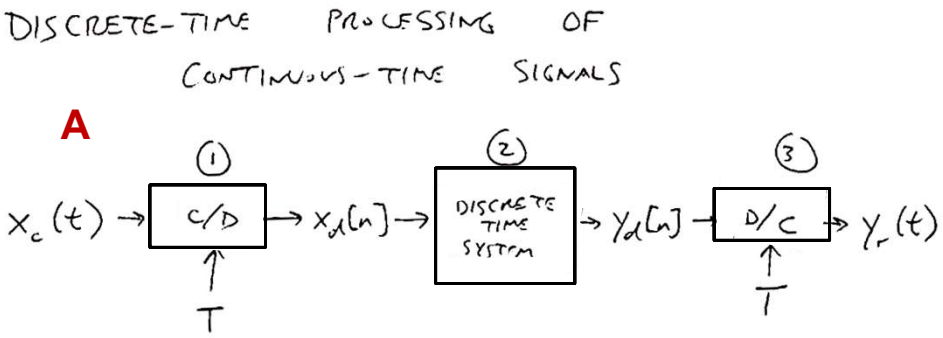
- To make the process we just discussed a little bit more understandable graphically, let us draw a sketch. In **A**, if $\mathbf{X_c(\omega)}$ is our continuous time FT of $\mathbf{x_c(t)}$ or **CTFT** of $\mathbf{x_c(t)}$, when we look at the sampled signal, $\mathbf{X_s(\omega)}$ in **B**, what we get is a bunch of copies (replicas), i.e., the **CTFT** of $\mathbf{x_s(t)}$. Additionally, if point **P1** in **A** had a height of **1** before, now it becomes **P2** with a height of **1/T**. Also, now, we have copies that are centered at multiples of the sampling frequency, $\boldsymbol{\omega_s}$.
- The upshot of last time was that, as long as we sample fast enough, then the copies in **B** do not overlap. If the copies ***do*** overlap, that means we have **aliasing**. We concluded earlier that aliasing happens when we do not sample fast enough, and as result, in **B**, $\boldsymbol{\omega_s}$ shown in **b2** gets too close to the middle copy, **b1**. After that, things start to bleed in, in the middle.

How copies appear in the CTFT vs. the DTFT



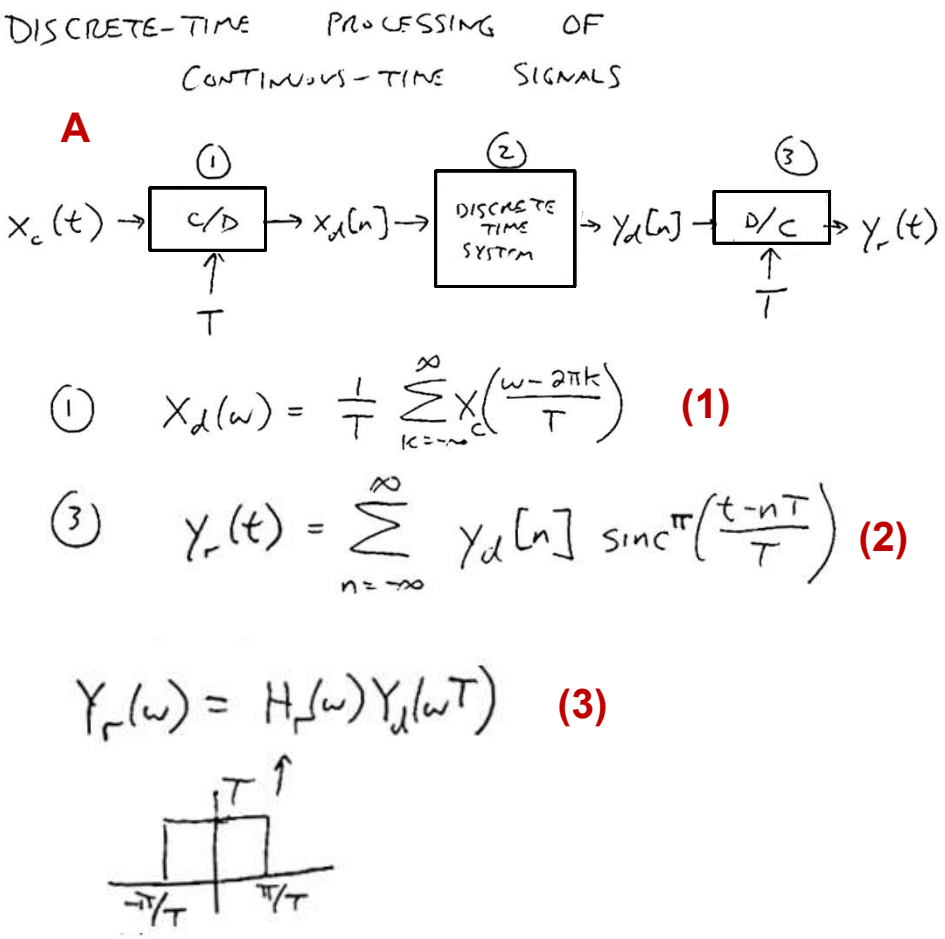
- **How is the DTFT of the discrete time signal, x[n], related to the signals shown in A, and B?** We know that the DTFT has to be **2π-periodic** (as shown in **C**). Fundamentally, all that we are doing is squishing our graph down, and also, we are scaling the **x**-axis so that the copy is reoccurring at multiples of **2π**. That is, instead of copies occurring at multiples of sampling frequency, ω_s, that we see in **B** for CTFT of $x_s(t)$, we get DTFT of **x[n]** or $X(\omega)$ at multiples of **2π**. Note that the bandlimit in both **A** and **B**, is ω_B.
- Here, what used to be ω_s in **B**, **P2**, now becomes **2π** in **C**, **P4**, and what used to be ω_B in **B**, **P1**, becomes ω_B times $2\pi/\omega_s$ or $\omega_B.T$, **P3** in **C**. Here, **T** is the sampling period.

Discrete-time processing of continuous-time signals



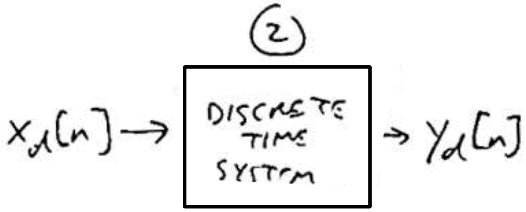
- **Discrete-time processing of continuous-time signals:** In practice, what we want to do under the hood is process some continuous-time signals. But, all we have are discrete-time systems. So, we need to do discrete-time processing of continuous-time signals. The underlying model for what we want to do is that we have some continuous-time signal that comes in, $\mathbf{x_c(t)}$, as shown in **A**, we do some sort of a continuous to discrete or A/D or a C/D conversion using A/D or C/D converter, where we sampled the signal by some sampling rate. This is shown by block **1**. We know that the sampling rate, **T**, is also an input to block **1**. Now, we get some sort of a discrete-time signal, $\mathbf{x_d[n]}$. Next, as shown in block **2**, we can process $\mathbf{x_d[n]}$ with some sort of a discrete-time system to get some discrete output, $\mathbf{y_d[n]}$. Then we turn $\mathbf{y_d[n]}$ back into a continuous-time signal with a D/C or D/A converter. Again, we know the sampling rate, **T**, is here. Our output signal is $\mathbf{y_r(t)}$.
- Now, we have all the pieces in place to understand how we should design the discrete time system in block **2** to emulate an entire continuous-time system. So, if we wanted to have the ***effective frequency response*** in the continuous-time world of this whole thing to be some certain frequency response, how would we do that?

Discrete-time processing of continuous-time signals



- For block **1** and equation **(1)**, we know that the DTFT or $\mathbf{X_d(\omega)}$ is equal to the sum of the copies of the scaled frequency response of the continuous-time signal of $\mathbf{x_c(t)}$. So, it is basically saying where the copies occur, which we already know they occur every **2π** places.
- How do we reconstruct the signal? The reconstruction filter, shown by block **3**, and in equation **(2)** says that if we want to go from D to A (or D to C), it is a combination of the original values, $\mathbf{y_d[n]}$ and a whole bunch of sinc functions. We talked about this last time and pointed out that this function is an ideal filter. The thing that takes us from spectrum of $\mathbf{y_d[n]}$ to spectrum of $\mathbf{y_r(t)}$ is just a **lowpass filter**. We showed that what we want to do in the frequency domain is notch out the middle copy. Once we have notched out the middle copy, we can think of that as something we can take the inverse continuous-time FT of. Here, $\mathbf{H_r(\omega)}$ is a lowpass filter, as shown in **(3)**. And, this lowpass filter has to be of height **T** in order to compensate for the fact that the copies are of height **1/T**.

For a given sampling rate, how should the middle discrete-time system be chosen?



(2) $Y_d(\omega) = H(\omega) X_d(\omega)$ (1)

↑ FREQ RESP OF D.T. SYSTEM

↙ RECONSTRUCTION FILTER (PULSE)

$Y_r(\omega) = H_r(\omega) Y_d(\omega T)$ (2)

$= H_r(\omega) H(\omega T) X_d(\omega T)$ (3) →

$Y_r(\omega) = H_r(\omega) H(\omega T) \frac{1}{T} \sum_{k=-\infty}^{\infty} X_c\left(\omega - \frac{2\pi k}{T}\right)$ (4)

- Block **2** is saying that we have a DTFT relationship here, **(1)**, where **H(ω)** is the frequency response of the discrete time system. Now, we are going to put all these things together to figure out what is the **effective continuous time frequency response** that we would get.
- Let us work backwards. What we know is that our CTFT output, $\mathbf{Y_r(\omega)}$ in **(2)**, is the reconstruction filter, $\mathbf{H_r(\omega)}$, which is the pulse, multiplied by whatever our discrete time output is, $\mathbf{Y_d(\omega T)}$. Now, we are going to fill in what we know about $\mathbf{Y_d(\omega T)}$ from earlier. Using **(1)**, we get **(3)**. We continue working backwards and also substitute for $\mathbf{X_d(\omega T)}$ and eventually we will have **(4)**. The spectrum of the original input, $\mathbf{X_d(\omega T)}$, is the sum term in **(4)**.
- In **(4)**, the continuous time FT, $\mathbf{Y_r(\omega)}$, is basically a combination of the lowpass reconstruction filter, $\mathbf{H_r(\omega)}$, whatever the original discrete time system was doing, **H(ωT)**, and the copies of the frequency response, the sum term.

The effective continuous-time frequency response

$$Y_r(\omega) = H_r(\omega) H(\omega T) \frac{1}{T} \sum_{k=-\infty}^{\infty} X_c\left(\omega - \frac{2\pi k}{T}\right) \quad (1)$$

IF $X_c(\omega) = 0$ FOR $|\omega| > \frac{\pi}{T}$, THEN

$$(2) \quad Y_r(\omega) = \begin{cases} H(\omega T) X_c(\omega) & |\omega| < \frac{\pi}{T} \\ 0 & \text{ELSE} \end{cases}$$

- Now, starting from **(1)**, what we are going to do is put the assumption that says if the continuous time signal, $\mathbf{X_c(\omega)}$, is equal to **0**, for **ω** outside of this region, $|\boldsymbol{\omega}| > \frac{\pi}{T}$, i.e., this is the bandlimit, then ***effectively*** our output frequency response, $\mathbf{Y_r(\omega)}$, is going to be equal to whatever the discrete time filter, **H(ωT)**, was doing inside the bandlimit, i.e., $\mathbf{Y_r(\omega) = H(\omega T).X_c(\omega)}$, and it is going to be equal to **0** outside. That is exactly the effect of the reconstruction filter, $\mathbf{H_r(\omega)}$. Here, the filter is just notching out the lowpass part, taking the middle copy.

The effective continuous-time frequency response

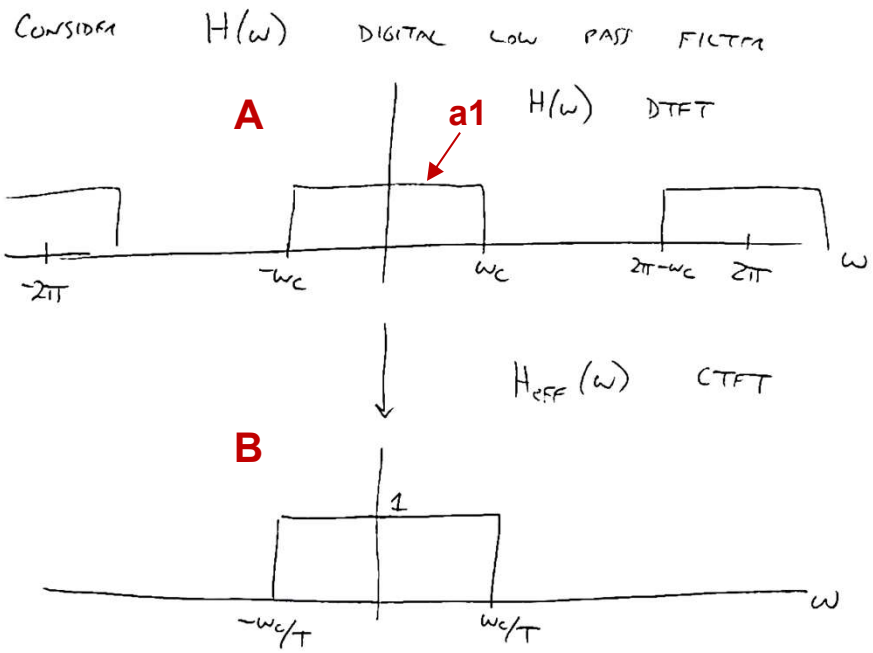
So the effective continuous-time frequency response is:

(1) $Y_r(\omega) = H_{eff}(\omega) X_c(\omega)$ (CTFT)

(2) $$H_{eff}(\omega) = \begin{cases} H(\omega T) & |\omega| < \pi/T \\ 0 & \text{else} \end{cases}$$

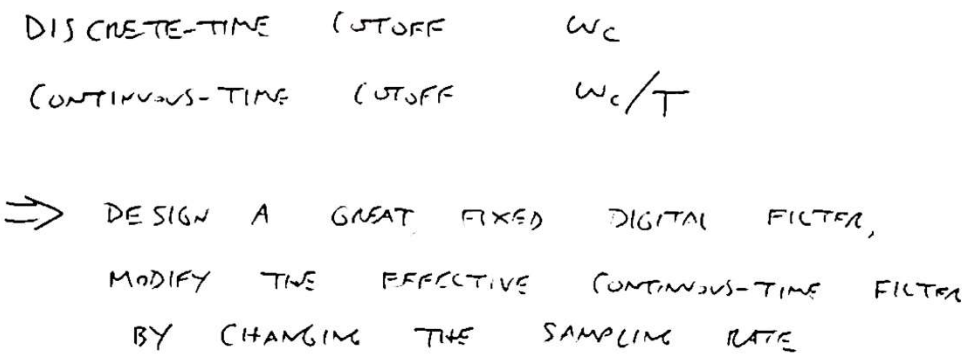
- The effective continuous-time frequency response is as shown in **(1)**. What is happening here is that in continuous-time, we have some ***effective filter***, $\mathbf{H_{eff}(\omega)}$, that is applied to our original continuous time input, $\mathbf{X_c(\omega)}$, in order to give us our output frequency response, $\mathbf{Y_r(\omega)}$. Note that all these signals are CTFT.
- Here, our effective continuous-time filter, $\mathbf{H_{eff}(\omega)}$, is just given by the discrete-time filter, $\mathbf{H(\omega T)}$, that is scaled by a factor of **T**, as shown in **(2)**.

$H_{eff}(\omega)$, effective frequency response in the continuous-time world



- Consider **H(ω)** as a digital lowpass filter. That means that it is periodic in the frequency domain, as shown in **A**. Here, the copies reoccur at multiples of **2π** and we have some sort of a cut off frequency, ω_c.
- Graphically, $H_{eff}(\omega)$ (effective frequency response in the continuous-time world) is simply the middle copy of **H(ω)**, **a1**, scaled. This is shown in **B**. So, if we want to get $H_{eff}(\omega)$ in the continuous-time world, we should design a lowpass filter in the discrete-world, **H(ω)**, that has a cut off, ω_c, but to get $H_{eff}(\omega)$, we basically multiply ω_c by **1/T**.

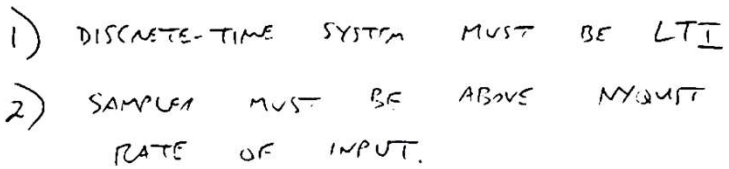
Cutoffs in discrete vs. continuous time



- **Practical Implication of ω_c/T:** The idea is that if we had a discrete-time cut off, ω_c, then we have an effective continuous-time cut off, $\omega_{eff} = \omega_c/T$. This suggests that we can accomplish many different continuous-time filters simply by designing one really good digital filter with a certain fixed cut off, and then running it through sampling systems with ***different sampling rates***. Here, a fixed digital filter means that ω_c remains constant.
- So, if we want to vary the continuous-time cut off, all we need to do is ***change the sampling rate of the input*** and then run it through this single great digital filter that we have designed. That is, we design a great fixed digital filter, and then we modify the effective continuous-time cut off of the filter, ω_{eff}, by changing the sampling rate. This is a neat trick! This means that by choosing a sampling rate, we can do whatever we want to make the effective system do what we want it to do. So, we choose a sampling rate so that we can make the continuous-time effective signal does the right thing.
- **Conclusion:** This technique leverages the power of well-designed digital filters and the flexibility of sampling rate to achieve a variety of continuous-time filtering behaviors. It is a clever way to achieve efficient and adaptable filtering in DSP applications.

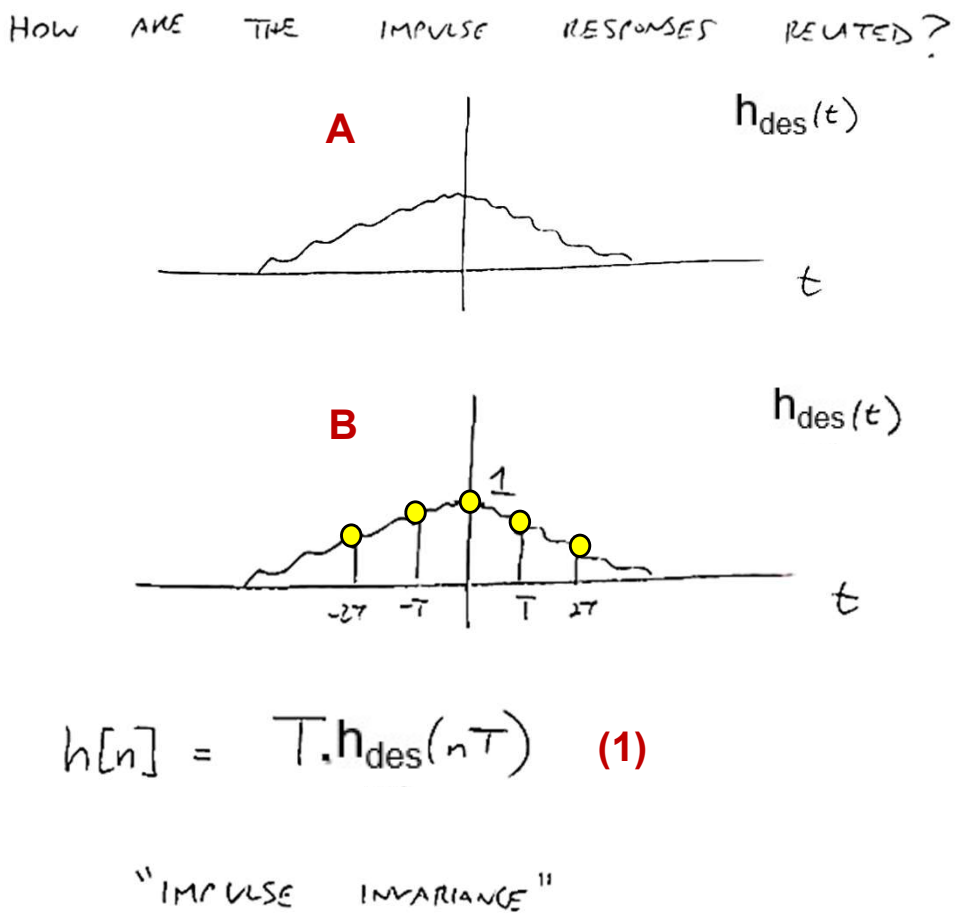
Cutoffs in discrete vs. continuous time

Caveats of $\omega_{eff} = \omega_c/T$ Method:



- **Caveats of $\omega_{eff} = \omega_c/T$ Method:** There are a bunch of caveats in this process. One is that we need to make sure that the discrete-time system is **LTI**. The other one is that the sampler must be above the **Nyquist Rate** of input. If the input is not sampled fast enough, then everything goes wrong. This is because we can have all these overlaps, and hence, the effective thing that we get is not actually what we thought we were going to get due to the **aliasing**. So, in summary, not being LTI and not being able to sample fast enough mean that we might not be able to actually get this method to work.

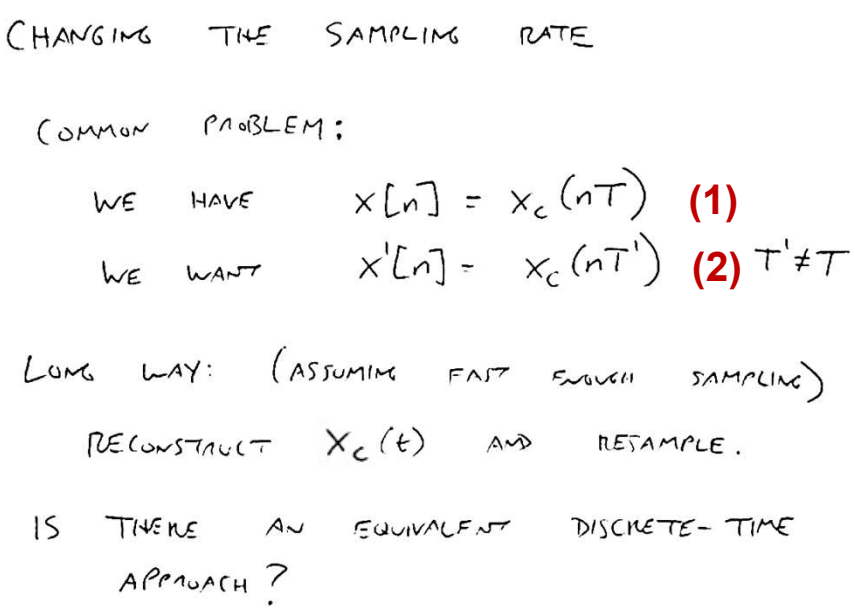
How are the impulse responses related?



How are the impulse responses related, "impulse invariance"? Let us say we have a time-domain ***desired*** impulse response, $\mathbf{h_{des}(t)}$, and it looks like **A**. Since we are in continuous-time, this desired impulse response is a continuous function. Assuming that everything is above the Nyquist, then the digital filter, **h[n]**, that corresponds to this desired response, $\mathbf{h_{des}(t)}$, is given by $\mathbf{h[n] = T.h_{des}(nT)}$, as shown in **(1)**.

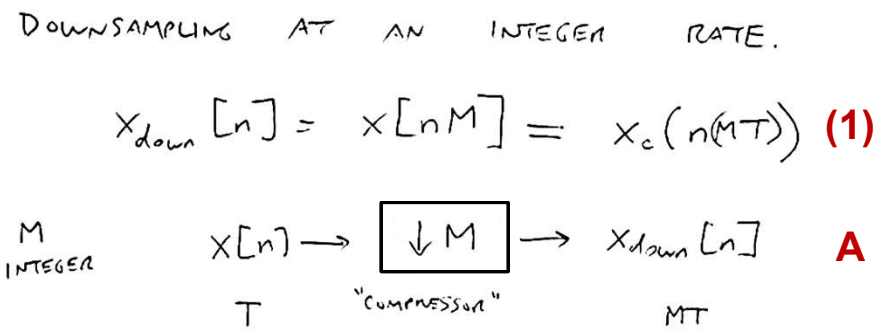
- It means to get our digital filter samples, **h[n]**, that correspond to $\mathbf{h_{des}(t)}$, we should sample $\mathbf{h_{des}(t)}$ every **T** units and then multiply those by **T**. This is shown in **B**.
- Graph **B** tells us that if we know what our ideal continuous-time impulse response is, we can immediately pick off what the corresponding values are. This property is called **impulse invariance**.
- The core idea behind impulse invariance is that under certain conditions, if the continuous-time filter's impulse response is perfectly preserved during sampling (without aliasing or distortion), the resulting discrete-time filter will have a frequency response that closely resembles the scaled frequency response of the original continuous-time filter.

Changing the sampling rate



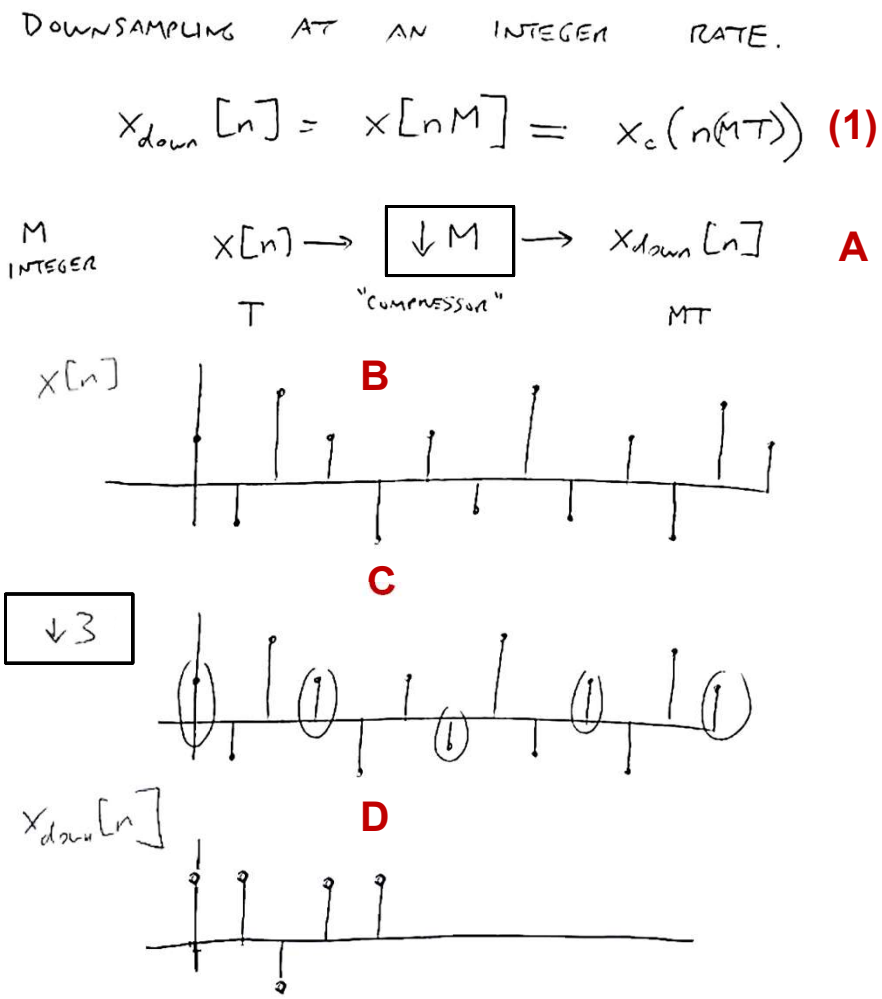
- **Changing the Sampling Rate:** Changing the sampling rate is something that happens a lot in practice. A common problem is that we have some discrete-time signal, **x[n]**, that came from sampling a continuous-time signal, $\mathbf{x_c(nT)}$ every **n** units. That is, $\mathbf{x[n] = x_c(nT)}$, as shown by (1). And, what we want, for some other reason, is a discrete-time signal, **x'[n]**, that came from sampling $\mathbf{x_c(nT')}$, at some different rate. That is, $\mathbf{x'[n] = x_c(nT')}$, as shown by (2). Here, **T' # T**, i.e., the periods (or sampling rates) are not equal to each other.
- **How do we go from x[n] to x'[n]?** Assuming everything is sampled above the Nyquist rate, in theory, there is the ***long way*** around. What we could do is to take our samples, **x[n]**, ***reconstruct*** the original continuous-time signal, $\mathbf{x_c(t)}$, and ***resample*** it to get back to the digital signal. That is true! We could do that, but that seems needless. This is because what we really want is a fully digital operation that goes from the original samples to the new samples. We may not be able to create analog signals inside our processor anyway. So, the long way (or the theoretical way), assuming fast enough sampling, is to ***reconstruct*** the continuous-time signal and ***resample***. We really do not want to do that if we can help it. So, **is there an equivalent discrete-time approach?**

Downsampling by an integer factor



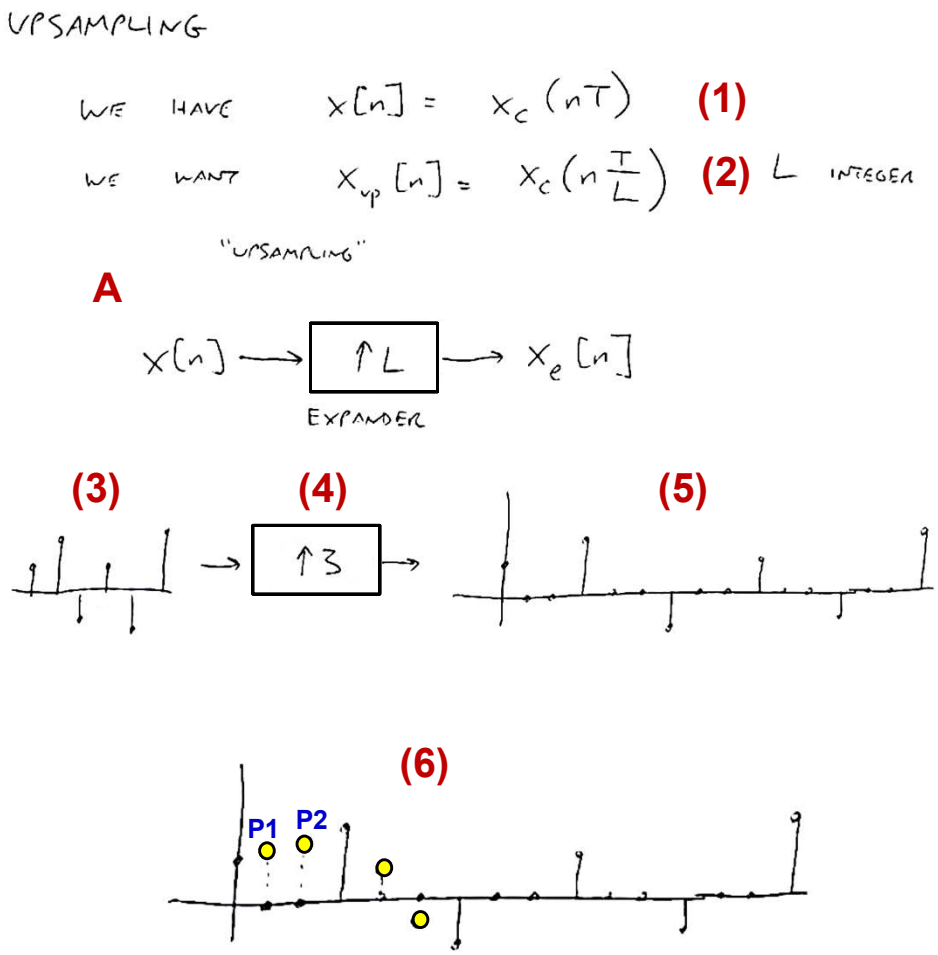
- Next, we are going to talk about two different equivalent discrete-time ways of going from **x[n]** to **x'[n]**. These are called **upsampling** and **downsampling**.
- **Downsampling at an Integer Rate:** This means that the downsampled signal, $\mathbf{x_{down}[n]}$, is the original digital signal, **x[nM]**, taken every **M** units. This is the same thing as if we had sampled the continuous-time signal, $\mathbf{x_c(n(MT))}$, every **MT** units. That is, $\mathbf{x_{down}[n] = x[nM] = x_c(n(MT))}$, as shown in **(1)**.
- Sometimes, as shown in **A**, we represent the above process in the form of a block that says, downsample by a factor of **M** ($\downarrow \mathbf{M}$). We use a **down arrow** in the block. For the original signal, **x[n]**, the sampling rate was **T**, and for $\mathbf{x_{down}[n]}$, the sampling rate is **MT**. Sometimes, the block is called a **compressor**, i.e., the block of $\downarrow \mathbf{M}$.

Downsampling by an integer factor



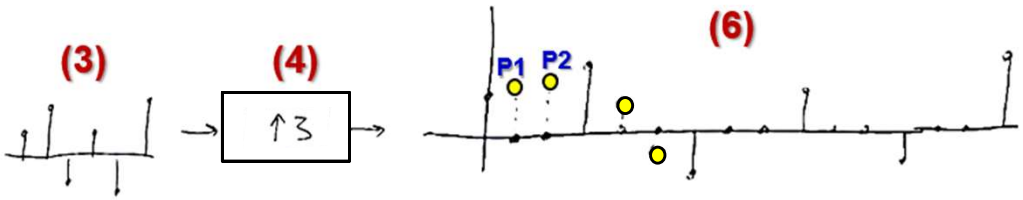
- **How would we do this downsampling?** We need to take every **M** sample of our original discrete time signal, **x[n]**, in **B**. Here, **M** has got to be an integer. As an example, say **M = 3**, then in **C**, it is like saying we are taking every **3**rd samples, i.e., the encircled samples are chosen. After downsampling by **3**, we get our new samples, $\mathbf{x_{down}[n]}$, as shown in **D**.
- It is definitely true that if our original signal was sampled fast enough, i.e., if it was sampled at least **M** times the Nyquist rate, then we should still be able to reconstruct the original signal from the samples in $\mathbf{x_{down}[n]}$. However, it is possible that in the process of downsampling, we remove some critical information that we would need to reconstruct the signal.

Upsampling by an integer factor



- **Upsampling at an Integer Rate:** Upsampling is the process by which we increase the sampling rate. We get more samples than we had before. We have **x[n]**, which is some set of samples like $\mathbf{x_c(nT)}$, and we want $\mathbf{x_{up}[n]}$ to be equal to the continuous-time signal sampled at some faster rate, $\mathbf{x_c(nT/L)}$. These two equations are shown in **(1)** and **(2)**.
- In **A**, there is a block inside which we have an **up arrow**. This is called an **expander**. **What does this expander do?** This expander is going to insert zeros between our original samples. So, for example, if we have a set of original samples, in **(3)**, and we put it through an upsampler **3** (**L = 3**), shown in **(4)**, we will get **(5)**. In **(5)**, we get the original samples and then between every pair of samples, we get a couple of zeros. Graph **(5)** is not quite what we want. What we want is something that instead of getting just zeros, we get like connecting the dots, as shown in **(6)**. That is, we want our signal to hallucinate what should have happened between these samples, such as **P1** and **P2**.

Upsampling by an integer factor



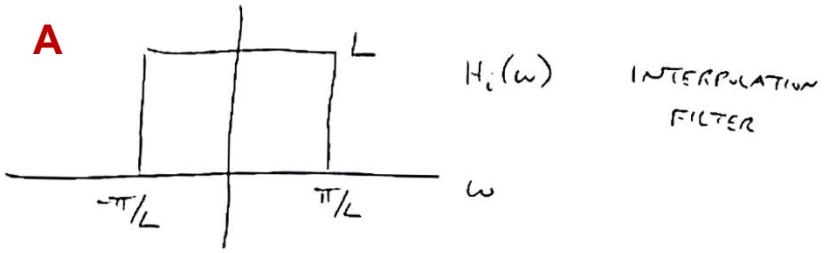
- **Going from (3) to (6):** To get from (3) to (6) is not that hard. The reason for that is, assuming that we have already sampled the signal fast enough, we know that we could always bump up to the original continuous-time signal and sample it as fast as we wanted. So, all the information to make more samples is implicit in our original set by assumption. If we sample above the Nyquist rate, then we have got all the samples we needed from the get go. So, we should be able to generate as dense of a sampling as we want from now.

Ideal reconstruction of the missing samples via lowpass filtering

$$x_e[n] = \begin{cases} x[n/L] & n = 0, \pm L, \pm 2L, \ldots \\ 0 & \text{ELSE.} \end{cases} \quad (1)$$

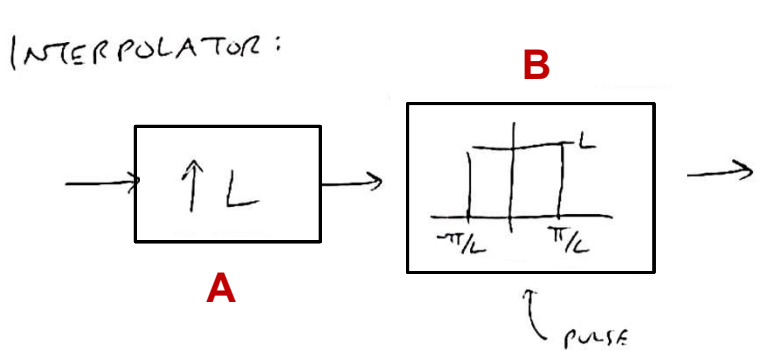
How to create $x_{up}[n]$ from $x_e[n]$?

We just pass it through a lowpass filter



- **What does the expander do?** The expander, $x_e[n]$, is equal to $x[n/L]$ for $n = 0, \pm L, \pm 2L, \ldots,$ and **0** elsewhere, as shown by (1). Here, **n** is multiples of **L**.
- **How to create the upsampled version of n, $x_{up}[n]$, from the expanded version of n, $x_e[n]$?** The distinction is that the expanded version just has a bunch of zeros in the gaps. The upsampled version actually has the correct values inside the gaps. It turns out, we just pass $x_e[n]$ through a lowpass filter that looks like A. This is called **Interpolation Filter**. Interpolation is really another word for ***connecting the dots***. And, so it turns out that by simply filtering our expanded signal, $x_e[n]$, with this filter, $H_i(\omega)$, we get the right answer.

Time-domain interpolation



- **Interpolator:** The whole system of inserting the zeros and applying the lowpass filter is sometimes called an **interpolator**. What we need to do here is build a system that is combined of an expander block, **A**, and a lowpass filter block, **B**. This whole thing is all we need to upsample the signal by an integer rate. If we think about what is happening in the time-domain, **B** is actually a pulse.

Upsampling with Interpolation: summary and conclusion

Summary and Conclusion:

Upsampling with Interpolation:

- **Upsampling:** This process increases the sampling rate of a discrete-time signal by inserting additional samples between the existing ones. The goal is to create a smoother representation of the original continuous-time signal from which the samples were obtained.
- **Interpolation:** This refers to the techniques used to estimate the values of the missing samples during upsampling.

The Expander Block and Zeros:

- The expander block acts as an "inserter" that creates a new sequence by placing zeros between the original samples. This effectively increases the sampling rate by an integer factor, depending on the number of zeros inserted between each original sample.

Time-Domain Interpretation:

- In the time-domain, the expander block can be visualized as a pulse (often a rectangular pulse) that gets inserted between the original samples. The width of this pulse determines the number of zeros inserted.

Upsampling with Interpolation: summary and conclusion

The Lowpass Filter:

- The lowpass filter plays a crucial role in the interpolation process.
- The inserted zeros create spectral "images" of the original signal's frequency content at multiples of the upsampling rate. These images can distort the upsampled signal if left unaddressed.
- The lowpass filter removes these unwanted spectral images, ideally allowing only the original signal's spectrum to pass through. This filtering process helps reconstruct a smoother version of the original continuous-time signal from the upsampled discrete-time sequence.

Overall System:

- The combination of the expander block and the lowpass filter forms an **interpolator**. By inserting zeros and filtering the resulting sequence, this system aims to achieve the following:
  - Increase the sampling rate of the signal.
  - Minimize the distortion introduced by the upsampling process.
  - Create a more accurate representation of the original continuous-time signal.

Multirate signal processing

Multirate Signal Processing (MRSP):
Imagine we have a box that can change the sampling rate of a signal. That is essentially what Multirate Signal Processing (MRSP) is all about. It deals with processing signals that have different sampling rates or require a change in sampling rate.

Why would we want to change the sampling rate? Here are some reasons:

- **Matching devices:** Different devices might have different default sampling rates. MRSP helps us connect them and process the signals seamlessly.
- **Data compression:** We can reduce the amount of data by lowering the sampling rate for certain frequency components (if they do not contain important information).
- **Feature extraction:** Sometimes, analyzing a signal at a different sampling rate can reveal hidden patterns or features.

Multirate signal processing

Basic Operations in MRSP:
There are two fundamental operations in MRSP:

1. **Decimation (or Downsampling):** This reduces the sampling rate of a signal. Think of it as taking samples less frequently. We need to be careful not to lose information during this process (explained by the Nyquist criterion).
2. **Interpolation:** This increases the sampling rate of a signal. Think of it as inserting new samples between existing ones. We use interpolation techniques to estimate the missing values and avoid distortion.

Example:
Imagine we have a music signal originally sampled at 44.1 kHz (CD quality). We want to play it on a low-bandwidth device that can only handle 8 kHz sampling. Here, MRSP would use decimation to reduce the sampling rate from 44.1 kHz to 8 kHz. However, to avoid losing too much information, we need to ensure the original signal does not have any significant frequency components above 4 kHz (half of the new sampling rate according to the Nyquist criterion).

Multirate signal processing

Multirate Signal Processing: Decimation

Equation:
The main equation for decimation by a factor of **M** (reducing sampling rate by **M**) is:
y[n] = x[nM]
where:

- **x[n]**: Original signal at sampling rate $\mathbf{F_s}$
- **y[n]**: Decimated signal at sampling rate $\mathbf{F_s/M}$
- **n**: Integer index for the samples in the decimated signal

Example:
Let us say we have a sine wave signal **x(t) = sin(2πt)** sampled at 10 kHz ($\mathbf{F_s}$ = 10 kHz). We want to downsample it by a factor of 2 (**M** = 2) to obtain a signal at 5 kHz.
Here is how decimation works:

- We take only every other sample from the original signal: **x[0], x[2], x[4], ...**
- These become the samples of our decimated signal **y[n]**: **y[0] = x[0], y[1] = x[2], y[2] = x[4], ...**

Note:
Decimation by a factor **M** reduces the sampling rate but also introduces a constraint on the original signal's frequency content according to the Nyquist criterion. The highest frequency component in the original signal must be less than $\mathbf{F_s/(2M)}$ to avoid aliasing (distortion) after decimation.

Polyphase representations

Polyphase Representations in DSP:
Polyphase representations are a powerful tool used in MRSP. They offer a way to efficiently implement certain multirate filters. Here is the idea:

- Imagine we have a complex filter designed for a specific sampling rate.
- In the polyphase representation, we split this filter's impulse response (the filter's "fingerprint") into smaller, "phase-shifted" versions.
- These smaller versions, called polyphase components, depend on the upsampling or downsampling factor we want to achieve.

Benefits of Polyphase Representations:

- **Efficient filtering:** By operating on these smaller polyphase components, we can implement the original filter for different sampling rates without having to redesign it entirely.
- **Parallel processing:** Polyphase components can be processed in parallel, potentially leading to faster filtering operations.

Example:
Imagine we have a complex lowpass filter designed for a 1 kHz sampling rate. Now, we want to use this filter for a signal upsampled to 4 kHz. Using the polyphase representation, we can split the original filter's impulse response into 4 smaller components. Each component represents a specific phase shift of the filter's response. By processing these components and combining the results appropriately, we can achieve the same filtering effect on the upsampled signal (4 kHz) as the original filter had on the 1 kHz signal.

Polyphase representations

Polyphase Representation:

Concept:
Instead of directly applying a filter to an upsampled or downsampled signal, we can decompose the filter into its polyphase components. These components represent phase-shifted versions of the original filter's impulse response.

Equations:
The equations for polyphase decomposition depend on the upsampling or downsampling factor (**M**). Here is an example for upsampling by a factor of **L**:

- Define the original filter impulse response as **h[n]**.
- The polyphase components are denoted as $\mathbf{H_k[n]}$; **(k = 0, 1, ..., L-1)**.
- They are calculated as: $\mathbf{H_k[n] = h[nL + k]}$; (for $\mathbf{0 \le n \le (N-1)/L}$)
- where:
  - **N**: Length of the original filter impulse response
  - **k**: Index for the polyphase component (determines the phase shift)

Example:
Consider a moving average filter with impulse response **h[n] = {1/3, 1/3, 1/3}** (averages the past 3 samples). We want to use this filter on a signal upsampled by a factor of **L** = 2 (doubled sampling rate).

- Here, the polyphase components would be:
  - $\mathbf{H_0[n] = h[2n]}$ (even indexed samples)
  - $\mathbf{H_1[n] = h[2n+1]}$ (odd indexed samples)

In this case, $\mathbf{H_0[n] = \{h[0], h[2]\} = \{1/3, 1/3\}}$ (only even positions have a value) and $\mathbf{H_1[n] = \{h[1]\} = \{1/3\}}$ (only odd positions have a value).

Polyphase representations

Upsampling with Polyphase Filtering:

1. Upsample the original signal by inserting **L-1** zeros between each sample.
2. Apply the corresponding polyphase component ($\mathbf{H_k[n]}$) to each group of **L** samples in the upsampled signal.
3. Sum the outputs from all polyphase components for each group of **L** samples to get the filtered value.

This process achieves the same filtering effect as applying the original filter directly to the upsampled signal, but potentially with less computation.

Remember:

- MRSP is about manipulating sampling rates of signals.
- Polyphase representations are a tool within MRSP that helps implement multirate filters efficiently.

Wavelet analysis

Wavelet Analysis in DSP:

- Imagine a signal as a complex scene: We might have mountains (low frequencies) and birds chirping (high frequencies) all happening simultaneously. Fourier Transform, a popular signal analysis tool, is like looking at the scene with a single spotlight. It tells us the overall "brightness" (energy) at different colors (frequencies) but does not reveal details about where these colors appear (time).
- Wavelet analysis is like having a magnifying glass that zooms in on specific areas. It uses small wavelets (wave-like functions) that can be stretched or compressed in time (like zooming). By analyzing the signal with these wavelets at different scales and positions, we can understand how the different frequency components (mountains and birds) change over time.

The Equations:

The core of wavelet analysis involves the Discrete Wavelet Transform (DWT). Here is a breakdown of the algorithm:

1. **Filter Banks:** The signal is passed through high-pass and low-pass filters (like separating colors with filters).
2. **Decomposition:** The low-pass filter output captures the overall trend (mountains) - the approximation coefficients. The high-pass filter output captures the details (birds chirping) - the detail coefficients.
3. **Downsampling (Optional):** Sometimes, the detail coefficients are downsampled (fewer data points) to maintain the same number of coefficients as the original signal.
4. **Repeat:** Steps 1-3 are repeated with the approximation coefficients, essentially zooming in on the "mountains" with smaller wavelets to analyze their finer details. This creates multiple levels of detail coefficients.

Wavelet analysis

Numerical Example:

- Imagine a sine wave representing a single frequency. A wavelet analysis would decompose it into approximation coefficients (capturing the overall wave) and detail coefficients (potentially showing very little detail since the signal has only one frequency).

Advantages:

- **Time-Frequency Analysis:** Unlike Fourier Transform, wavelet analysis provides information about both frequency and time, allowing for a more detailed understanding of how the signal changes.
- **Sparsity:** For certain signals (like images with sharp edges), wavelet analysis can represent them efficiently with many zero coefficients (areas with no change).

Disadvantages:

- **Computational Cost:** Performing a wavelet transform can be computationally expensive compared to simpler methods like Fourier Transform.
- **Choosing the Right Wavelet:** Different wavelets are suitable for different types of signals. Choosing the wrong wavelet can lead to poor analysis results.

Conclusion:

- Wavelet analysis offers a powerful tool for analyzing signals in both the time and frequency domains. It is particularly useful for understanding non-stationary signals (signals whose frequencies change over time) and signals with sharp features. However, it requires more computational resources and careful selection of the wavelet function compared to some other methods.

Use of stochastic processes in DSP

Stochastic Process:

- Imagine flipping a coin repeatedly. The outcome (heads or tails) is random, but there might be underlying patterns or probabilities. A stochastic process is similar. It deals with signals that exhibit some randomness or uncertainty.
  - **Think of it as:** A constantly unfolding story where the next element depends on chance or some probability distribution.
  - **Examples:** Stock prices, weather patterns, audio with background noise, biological signals (biosignals), such as EEG.
  - **Applications:** Signal processing in finance, communication systems with noise, biosignal processing.

Stochastic processes play a significant role in DSP (Digital Signal Processing) because many real-world signals exhibit some degree of randomness or uncertainty. Here is how they are related:

Signals in the Real World:

- Unlike theoretical signals used in textbooks, most real-world signals are not perfectly predictable.
- They might contain noise from various sources, have inherent randomness due to their nature, or be influenced by external factors.

Stochastic Processes as a Modeling Tool:

- Stochastic processes provide a mathematical framework for modeling and analyzing signals with randomness.
- They allow us to describe the statistical properties of these signals, such as their probability distributions, correlations between samples, and spectral characteristics.

Use of stochastic processes in DSP

Applications of Stochastic Processes in DSP:

Here are some specific applications of stochastic processes in DSP:

- **Signal Modeling:** Stochastic models can be used to represent signals like:
  - Speech or other biosignals with background noise.
  - Financial data with fluctuations in prices.
  - Images with sensor noise.
- **Noise Reduction:** By understanding the statistical properties of noise as a stochastic process, we can design filters to remove it effectively.
- **Signal Compression:** Stochastic models can help compress signals by focusing on the essential information and discarding the random fluctuations.
- **Communication Systems:** In communication channels with noise, stochastic models are used to analyze signal transmission and design robust communication protocols.
- **Adaptive Signal Processing:** Algorithms designed to adjust their behavior based on the characteristics of the incoming signal often rely on stochastic process models to understand the changing environment.

Examples:

- Consider a music signal with background noise. The music itself can be modeled as a deterministic process, but the noise can be modeled as a stochastic process with a specific probability distribution.
- In image processing, sensor noise in a digital photograph can be modeled as a stochastic process to develop algorithms for image denoising.

Conclusion:

- Stochastic processes offer a powerful tool in DSP for dealing with the inherent randomness and uncertainty present in real-world signals. By understanding these statistical properties, we can design better algorithms for signal processing tasks like filtering, compression, communication, and adaptation.

Use of stochastic processes in DSP

Methods that use stochastic process:

1. Linear Prediction:

- This technique predicts the future value of a signal based on its past values. It is like trying to guess the next word in a sentence based on the previous ones.
  - **Think of it as**: An educated guess about the future based on what we have seen in the past.
  - **Process**: Analyzes a signal's past values to identify patterns and uses those patterns to estimate the next value.
- **Applications**: Signal compression, speech coding, noise reduction.

2. Auto-Regressive Model (AR Model):

- This is a specific type of linear prediction model where the predicted value is a linear combination of past values of the signal itself. It is like saying the next word depends only on the last few words we heard.
  - **Think of it as**: A simplified prediction model that assumes the future relies only on the signal's recent history.
  - **Equation (Simplified):** $\hat{y}(n) = a_1\hat{y}(n-1) + a_2\hat{y}(n-2) + \ldots + a_p\hat{y}(n-p) + e(n)$ ($\hat{y}$ is predicted value, a_k**'s** are coefficients, **e** is error)
  - **Applications**: Signal modeling, system identification, speech analysis.

Use of stochastic processes in DSP

3. Matched Filter:

- Imagine searching for a specific sound in a noisy recording. A matched filter is designed to maximize the signal-to-noise ratio (SNR) when detecting a particular known signal. It is like having a key that perfectly fits a specific lock.
  - **Think of it as**: A tool to find a hidden pattern (signal) buried in noise.
  - **Process**: The filter is designed based on the characteristics of the known signal to enhance its presence and suppress everything else.
  - **Applications**: Radar systems, communication signal detection, image feature extraction, biosignals and systems.

4. Wiener Filter:

- This filter aims to minimize the mean squared error between the desired signal and the filter's output. It is like trying to find the best possible fit for the desired signal, even if we do not have its exact form.
  - **Think of it as**: A filter that tries to remove all unwanted noise while preserving the original signal as much as possible.
  - **Process**: Analyzes the statistics of both the signal and noise to create a filter that optimizes the trade-off between noise reduction and signal distortion.
  - **Applications**: Noise cancellation in audio systems, image deblurring, channel equalization in communication systems.

End of Lecture 14