Lab 2: Fourier Transform Approximation, More General PAM

1 Introduction

The rate $F_B = 1/T_B$ at which discrete time (DT) information symbols are transmitted over a communication channel is called the **baud rate** (in honor of Émile Baudot, 1845-1903) or the **symbol rate**. If the symbols can only take on one of two values, e.g., 0 and A (**unipolar** binary signaling), or $-A$ and $+A$ (**polar** binary signaling), then the baud rate is equal to the bit rate.

In order to either accomodate as many simultaneous transmissions as possible (e.g., using time division multiplexing (TDM)), or to obtain the fastest rate for a single transmission over a bandlimited channel, it is important to know how much bandwidth a given signaling method requires. A good initial guess is that a higher baud rate will require more bandwidth and more simultaneous transmissions also require more bandwidth. But what is the theoretical minimum needed and how close can one get in practice?

As a prerequisite to answering this question it is necessary to be able to express signals in both the time and the frequency domains, using Fourier transforms in the latter case. Then, looking at PAM (pulse amplitude modulation) signals in the frequency domain and examining their different components, the bandwidth requirements of different signal classes can be determined.

1.1 Different Fourier Transforms

Fourier transforms are used to convert between time and frequency domain representations of signals. Both time and frequency domain representations can be either continuous or discrete in their respective variables. This results in a total of 4 different Fourier transform variants as outlined in the table below.

	Continuous Frequency (CF)	Discrete Frequency (DF)
Continuous Time (CT)	Fourier Transform FT	Fourier Series FS
Discrete Time (DT)	Discrete Time Fourier Transform DTFT	Discrete Fourier Transform Fast Fourier Transform DFT/FFT

The transform that is most general and easiest to work with analytically is the Fourier transform (FT) with continuous time and frequency domains. The Fourier series (FS) can be obtained from the FT by sampling in the frequency domain. The discrete time Fourier transform (DTFT) is the dual of the FS and is obtained from the FT by sampling in the time domain. Sampling in one domain implies periodicity in the other domain and thus time domain signals for the FS are periodic with period T_1, where $f_1 = 1/T_1$ is the fundamental frequency. Similarly, DTFT frequency domain represenations are periodic with period F_s, where F_s is the sampling rate. Finally, the discrete Fourier transform (DFT) and its computationally fast implementation, the fast Fourier transform (FFT), can be derived from the FT by sampling in both the time and frequency domains. In this case, since the representations in both domains are sampled, they are also both periodic in the blocklength N of the DFT. One of the key features of the DFT is that it can be computed easily (for moderate blocklengths at least) numerically for arbitrary signals.

In all 4 cases the frequency domain expressions are complex-valued in general (even if the time domain signals are real). Thus, it is necessary to display Fourier transforms in the form of two graphs, e.g., one for the **magnitude** and one for the **phase**.

1.2 Fourier Transform

Definition: The Fourier transform (FT) of a continuous time (CT) signal $x(t)$ is defined as

$$X(f) = \int_{-\infty}^{\infty} x(t)\, e^{-j2\pi ft} dt \, ,$$

where f is frequency in Hz (sec^{-1}).

Theorem: Inverse FT. A CT signal $x(t)$ can be recovered uniquely from its FT $X(f)$ by

$$x(t) = \int_{-\infty}^{\infty} X(f)\, e^{j2\pi ft} df \, .$$

Time-Shift Property: Let $x(t) \Leftrightarrow X(f)$ be a FT pair. Then, using the inverse FT,

$$x(t-t_0) = \int_{-\infty}^{\infty} X(f)\, e^{j2\pi f(t-t_0)} df = \int_{-\infty}^{\infty} X(f)\, e^{-j2\pi f t_0}\, e^{j2\pi ft} df \, ,$$

so that $x(t-t_0) \Leftrightarrow X(f)\, e^{-j2\pi f t_0}$, where $x(t-t_0)$ is $x(t)$ shifted to the right by t_0.

Frequency-Shift Property: Let $x(t) \Leftrightarrow X(f)$ be a FT pair. Then, using the definition of the FT,

$$X(f-f_0) = \int_{-\infty}^{\infty} x(t)\, e^{-j2\pi (f-f_0)t} dt = \int_{-\infty}^{\infty} x(t)\, e^{j2\pi f_0 t}\, e^{-j2\pi ft} dt \, ,$$

so that $X(f-f_0) \Leftrightarrow x(t)\, e^{j2\pi f_0 t}$, where $X(f-f_0)$ is $X(f)$ shifted to the right by f_0.

Example: Rectangular Pulse. Let

$$x(t) = \begin{cases} 1\,, & -\tau/2 \le t < \tau/2\,, \\ 0\,, & \text{otherwise}\,. \end{cases}$$

Then

$$X(f) = \int_{-\tau/2}^{\tau/2} e^{-j2\pi ft} dt = \frac{e^{-j2\pi ft}}{-j2\pi f}\Big|_{-\tau/2}^{\tau/2} = \frac{e^{-j\pi f\tau} - e^{j\pi f\tau}}{-j2\pi f} = \frac{\sin \pi f\tau}{\pi f}\,,$$

i.e., the FT of a rectangular pulse of width τ and amplitude 1 is a "sinc" pulse of amplitude τ and main lobe of width $2/\tau$.

Example: Ideal LPF. Let

$$H(f) = \begin{cases} 1\,, & -f_L \le f < f_L\,, \\ 0\,, & \text{otherwise}\,. \end{cases}$$

Then

$$h(t) = \int_{-f_L}^{f_L} e^{j2\pi ft} df = \frac{e^{j2\pi ft}}{j2\pi t}\Big|_{-f_L}^{f_L} = \frac{e^{j2\pi f_L t} - e^{-j2\pi f_L t}}{j2\pi t} = \frac{\sin 2\pi f_L t}{\pi t}\,.$$

Thus, the unit impulse response of an ideal LPF with gain 1 and cutoff frequency f_L is a "sinc" pulse with amplitude $2f_L$ and main lobe of width $1/f_L$.

1.3 Fourier Series

Definition: The Fourier series (FS) of a periodic CT signal $x(t)$ with period T_1 is defined as

$$X_k = \frac{1}{T_1} \int_{T_1} x(t)\, e^{-j2\pi kt/T_1} dt\,, \quad k = 0, \pm 1, \pm 2, \ldots\,,$$

where the integration is taken over any contiguous interval of length T_1. The FS coefficients X_k correspond to frequency components at $f_k = k/T_1$. Frequency $f_1 = 1/T_1$ is called the fundamental frequency, $f_2 = 2/T_1$ is called the 2'nd harmonic, $f_3 = 3/T_1$ is called the 3'rd harmonic, etc.

Theorem: Inverse FS. A periodic CT signal $x(t)$ can be recovered uniquely from its FS coefficients X_k by

$$x(t) = \sum_{k=-\infty}^{\infty} X_k\, e^{j2\pi kt/T_1}\,,$$

where T_1 is the period of $x(t)$.

FT of Periodic $x(t)$. A periodic CT signal $x(t)$ with period T_1 can also be characterized in the frequency domain in terms of a (continuous frequency) FT. Starting from its FS coefficients X_k, $x(t)$ can be expressed as

$$x(t) = \sum_{k=-\infty}^{\infty} X_k\, e^{j2\pi kt/T_1}\,.$$

Then, noting that the FT is a linear transformation, $X(f)$ can be computed as

$$X(f) = \mathcal{F}\{x(t)\} = \mathcal{F}\{\sum_{k=-\infty}^{\infty} X_k\, e^{j2\pi kt/T_1}\} = \sum_{k=-\infty}^{\infty} X_k\, \mathcal{F}\{e^{j2\pi kt/T_1}\} = \sum_{k=-\infty}^{\infty} X_k\, \delta(f - k/T_1)\,,$$

where $\delta(f)$ denotes a unit impulse in the frequency domain.

1.4 Discrete Time Fourier Transform

Definition: The discrete-time Fourier transform (DTFT) of a discrete time (DT) signal x_n, $n = 0, \pm 1, \pm 2, \ldots$, is defined as

$$X(\phi) = \sum_{n=-\infty}^{\infty} x_n\, e^{-j2\pi\phi n}\,,$$

where ϕ is a normalized (dimensionless) frequency. If $F_s = 1/T_s$ is the sampling rate of the sequence x_n, then $\phi = f/F_s$, where f and F_s are frequencies in Hz. Note that $X(\phi)$ is periodic in ϕ with period 1.

Theorem: Inverse DTFT. A DT signal x_n, $n = 0, \pm 1, \pm 2, \ldots$, can be recovered uniquely from its DTFT $X(\phi)$ by

$$x_n = \int_1 X(\phi)\, e^{j2\pi\phi n} d\phi\,,$$

where the integration is taken over any contiguous interval of length 1.

Time-Shift Property: Let $x_n \Leftrightarrow X(\phi)$ be a DTFT pair. Using the inverse DTFT

$$x_{n-m} = \int_1 X(\phi)\, e^{j2\pi\phi(n-m)}\, d\phi = \int_1 X(\phi)\, e^{-j2\pi\phi m}\, e^{j2\pi\phi n}\, d\phi\,.$$

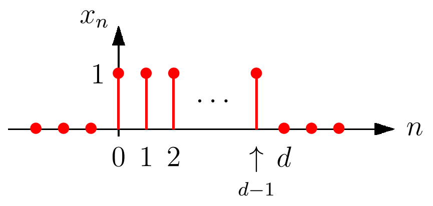
Therefore, $x_{n-m} \Leftrightarrow X(\phi)\, e^{-j2\pi\phi m}$, where x_{n-m} is x_n shifted to the right by m.

Frequency-Shift Property: Let $x_n \Leftrightarrow X(\phi)$ be a DTFT pair. Using the definition of the DTFT

$$X(\phi - \phi_0) = \sum_{n=-\infty}^{\infty} x_n\, e^{-j2\pi(\phi-\phi_0)n} = \sum_{n=-\infty}^{\infty} x_n\, e^{j2\pi\phi_0 n}\, e^{-j2\pi\phi n}\,.$$

Thus, $X(\phi - \phi_0) \Leftrightarrow x_n\, e^{j2\pi\phi_0 n}$, where $X(\phi - \phi_0)$ is $X(\phi)$ shifted to the right by ϕ_0.

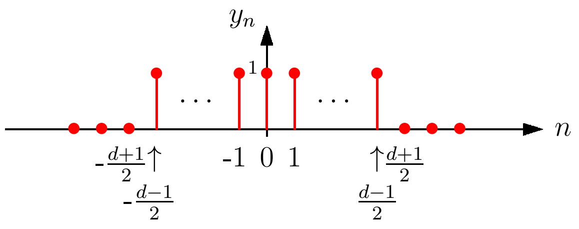
Example: Rectangular DT Pulse. Let x_n be the rectangular DT pulse of width d samples and amplitde 1 shown in the following graph.



The DTFT of x_n is

$$X(\phi) = \sum_{n=0}^{d-1} e^{-j2\pi\phi n} = \frac{1-e^{-j2\pi\phi d}}{1-e^{-j2\pi\phi}} = \frac{e^{j\pi\phi d}-e^{-j\pi\phi d}}{e^{j\pi\phi}-e^{-j\pi\phi}}\,\frac{e^{-j\pi\phi d}}{e^{-j\pi\phi}} = \frac{\sin\pi\phi d}{\sin\pi\phi}\,e^{-j\pi\phi(d-1)}\,.$$

If d is an odd integer, then $y_n = x[n+\frac{d-1}{2}]$ is the (symmetric around n=0) rectangular pulse of width d shown below.



Using the time shift property, its DTFT is

$$Y(\phi) = X(\phi)\,e^{j2\pi\phi(d-1)/2} = \frac{\sin\pi\phi d}{\sin\pi\phi}\,.$$

1.5 Discrete Fourier Transform

Definition: The discrete Fourier transform (DFT) of a DT signal x_n, $n = 0, 1, \ldots, N-1$ (mod N), that is periodic with period N, is defined as

$$X_k = \sum_{n=0}^{N-1} x_n\, e^{-j2\pi kn/N}\,, \quad k = 0, 1, \ldots, N-1 \pmod N\,.$$

Note that the sum can be taken over any N consecutive indexes. The term FFT (fast Fourier transform) refers to a fast algorithm for computing the DFT for composite N and, very often, for the case when N is a power of 2.

Theorem: Inverse DFT/FFT. A periodic DT signal x_n with period N can be recovered uniquely from the DFT coefficients X_k, $k = 0, 1, \ldots N-1$ (mod N), by

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k\, e^{j2\pi kn/N}\,, \quad n = 0, 1, \ldots, N-1 \pmod N\,.$$

Note that the sum can again be taken over any N consecutive indexes. The term inverse FFT refers to a fast algorithm for computing the inverse DFT when N is composite, most often when N is a power of 2.

1.6 Approximation of FT using DFT/FFT

The Fourier transform (FT) of the CT signal $x(t)$,

$$X(f) = \int_{-\infty}^{\infty} x(t)\, e^{-j2\pi f t} dt,$$

can be approximated as follows by sampling $x(t)$ with rate $F_s = 1/T_s$ at $t = nT_s$ and replacing the integral by a sum over rectangular areas of height $x_n = x(nT_s)$ and width T_s

$$\underbrace{X(f)}_{\text{FT}} \approx \sum_{n=-\infty}^{\infty} \underbrace{x(nT_S)}_{= x_n}\, e^{-j2\pi f T_s n}\, T_s = T_s \sum_{n=-\infty}^{\infty} x_n\, e^{-j2\pi f T_s n} = T_s \underbrace{X(fT_s)}_{\text{DTFT}} = \frac{X(f/F_s)}{F_s} \,,$$

where $X(fT_s) = X(f/F_s)$ is the DTFT of the sequence $x_n = x(nT_s) = x(n/F_s)$ with normalized frequency $\phi = f/F_s$. Note that, because of the sampling in the time domain, $X(f/F_s)$ is periodic in f/F_s with period 1.

Now suppose $x(t)$ was sampled at rate $F_s = 1/T_s$ yielding $x_n = x(nT_s)$ and a total number N of samples, e.g., $\{x_n\}_{n=0}^{N-1}$, are available to compute an approximation to the FT of $x(t)$. Using the above approximation one obtains

$$X(f) \approx \frac{1}{F_s} \sum_{n=0}^{N-1} x_n\, e^{-j2\pi f T_s n} \,,$$

where the last expression has the form of a DFT. Setting $fT_s = k/N$ and thus $f = kF_s/N$ yields

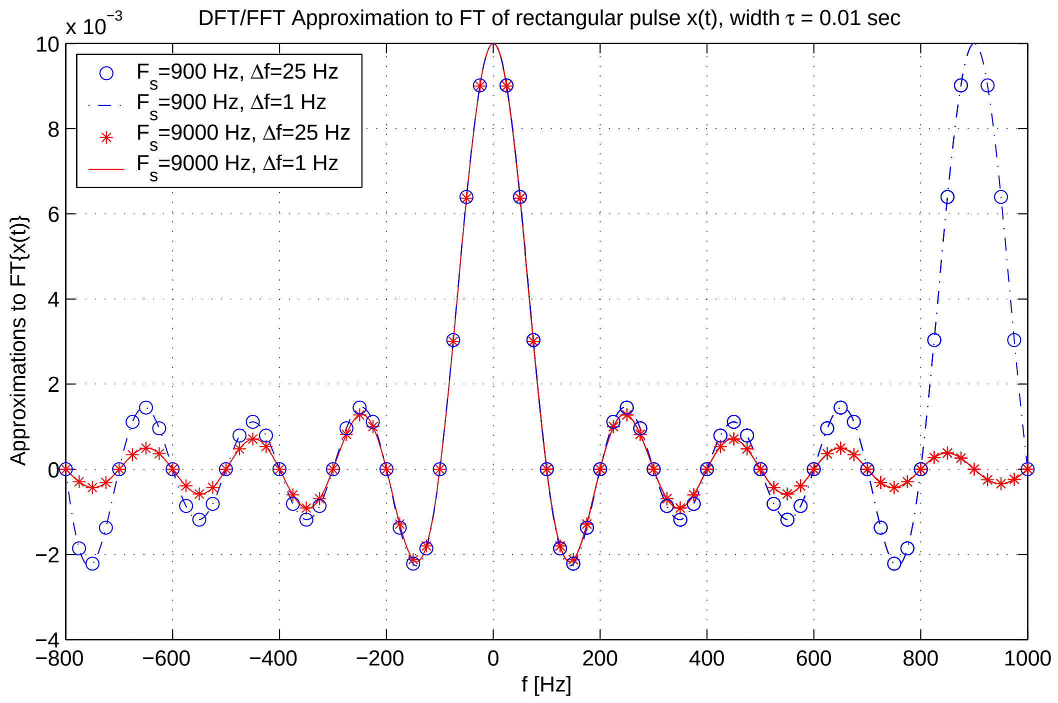
$$X\big(\frac{kF_s}{N}\big) \approx \frac{X_k}{F_s} \,,$$

where $X(kF_s/N)$ is the FT of $x(t)$ at $f = kF_s/N$ and X_k is the DFT (or FFT) of $x_n = x(nT_s)$, $n = 0, 1, \ldots, N-1$. Note that, since k has to be an integer, the frequency resolution of this approximation to $X(f)$ is $\Delta f = F_s/N$.

Example: Approximation to $X(f)$ for the rectangular pulse

$$x(t) = \begin{cases} 1\,, & \tau/2 \le t < \tau/2\,, \\ 0\,, & \text{otherwise} \end{cases} \qquad \Longleftrightarrow \qquad X(f) = \frac{\sin \pi f \tau}{\pi f} \,,$$

of amplitude 1 and width τ, centered at $t = 0$ (so that $X(f)$ is all real). The following figure shows the DFT/FFT approximation to $X(f)$ for different values and combinations of F_s and Δf.



The worst approximation is the one with $F_s = 900$ Hz and $\Delta f = 25$ Hz (resulting in a block length $N = F_s/\Delta f = 36$). Leaving F_s fixed but setting $\Delta f = 1$ Hz ($\Rightarrow N = 900$) improves the resolution (dash-dot blue line), but does not conceal the fact that the FT approximation is periodic in f with period F_s. To take care of this, F_s needs to be increased. Using $F_s = 9000$ Hz and $\Delta f = 25$ Hz ($\Rightarrow N = 360$) essentially removes errors due to aliasing for $|f| < 1000$ Hz, but does not give very good resolution along the f axis. Finally, using $F_s = 9000$ Hz and $\Delta f = 1$ Hz ($\Rightarrow N = 9000$) yields a rather close approximation (red line) to $X(f)$ for $|f| < 1000$ Hz.

1.7 Fourier Transform of PAM Signals

Let

$$s(t) = \sum_{n=-\infty}^{\infty} a_n \, p(t - nT_B) ,$$

be the PAM signal, based on a pulse $p(t)$, corresponding to the DT message sequence a_n with baud rate (or symbol rate) $F_B = 1/T_B$. The FT of $s(t)$ can be computed as

$$S(f) = \int_{-\infty}^{\infty} \sum_{n=-\infty}^{\infty} a_n \, p(t - nT_B) \, e^{-j2\pi ft} \, dt = \sum_{n=-\infty}^{\infty} a_n \int_{-\infty}^{\infty} p(t - nT_B) \, e^{-j2\pi ft} \, dt$$

$$= \sum_{n=-\infty}^{\infty} a_n P(f) \, e^{-j2\pi f T_B n} = P(f) \sum_{n=-\infty}^{\infty} a_n \, e^{-j2\pi f T_B n} = A(fT_B) \, P(f) \, .$$

where $A(fT_B)$ is the DTFT of a_n with normalized frequency $\phi = fT_B = f/F_B$, and $P(f)$ is the FT of $p(t)$.

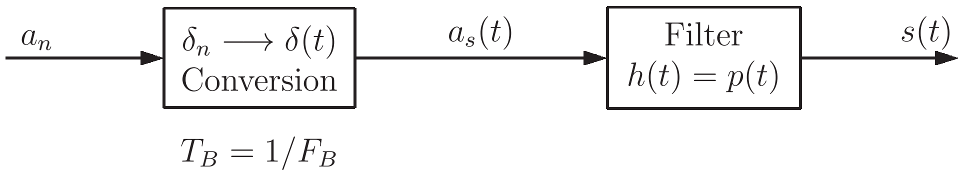
A convenient way to implement PAM with general $p(t)$ can be derived as follows. Define

$$a_s(t) = \sum_{n=-\infty}^{\infty} a_n\, \delta(t - nT_B) \qquad \Longleftrightarrow \qquad A_s(f) = A(fT_B)$$

where $A_s(f)$ is the FT of the (sampled) CT signal $a_s(t)$ and $A(fT_B)$ is the DTFT of the DT sequence a_n. To obtain $S(f)$, multiply $A_s(f)$ with $P(f)$ in the frequency domain or, equivalently, convolve $a_s(t)$ and $p(t)$ in the time domain. Thus

$$s(t) = a_s(t) * p(t) = \Big(\sum_{n=-\infty}^{\infty} a_n\, \delta(t - nT_B)\Big) * p(t) = \sum_{n=-\infty}^{\infty} a_n\, p(t - nT_B)\ .$$

The following figure shows this in the form of a blockdiagram.



The first block converts between δ_n and $\delta(t)$ by converting the amplitudes of the DT impulses to areas under the CT impulses. At the same time, the indexes n of the DT impulses are converted to times $t = nT_B$ where $F_B = 1/T_B$ is the baud rate of the DT signal a_n. The second block then merely consists of a CT shaping filter with impulse response $h(t) = p(t)$. Thus, to change the PAM pulse $p(t)$ that is used to generate $s(t)$, only the impulse response of the shaping filter needs to be modified.

1.8 PAM with General $p(t)$

Let a_n be a DT sequence with baud rate F_B and consider the PAM signal

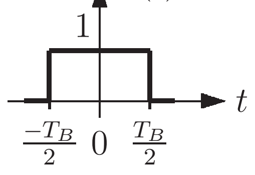
$$s(t) = \sum_{n=-\infty}^{\infty} a_n\, p(t - nT_B)\ ,$$

that is obtained by using a pulse $p(t)$ to convert the DT data sequence a_n into the CT signal $s(t)$ for transmission over a waveform channel.

Common choices for $p(t)$ are:

(1) "Cheap and Dirty" Waveform Generation ("Flat-Top" PAM): DT sequence to CT waveform conversion using the rectangular pulse

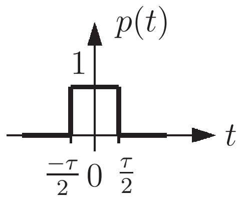
$$p(t) = \begin{cases} 1\ , & -T_B/2 \le t < T_B/2\ , \\ 0\ , & \text{otherwise}\ . \end{cases}$$



Very often this is used for unipolar binary sequences with $a_n \in \{0,1\}$ or for polar binary sequences with $a_n \in \{-1,1\}$, and then $s(t)$ can directly be taken from the output of some digital logic circuitry. This is inexpensive to implement, but is not bandwidth efficient.

(2) Simple Waveform Generation for TDM: DT sequence to CT waveform conversion such that TDM (time division multiplexing) can be used to share a communication channel among U users. A simple solution is to use the narrow rectangular pulse of width τ

$$p(t) = \begin{cases} 1\,, & -\tau/2 \le t < \tau/2\,, \\ 0\,, & \text{otherwise}\,. \end{cases}$$



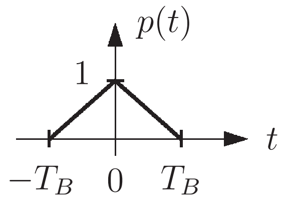
where τ must satisfy $\tau \le T_B/U$. In this case each of the U DT sequences $a_i[n]$, $i = 1, 2, \ldots, U$, is converted to a CT waveform using $p(t)$, then staggered in time by an offset of $(i-1)T_B/U$ for the i-th sequence, and finally added up to form the TDM signal $s(t)$. For the $U = 2$ case, for instance, one uses

$$s(t) = \sum_{m=-\infty}^{\infty} a_1[m]\, p(t - mT_B) + \sum_{m=-\infty}^{\infty} a_2[m]\, p(t - mT_B - T_B/2)\,.$$

This is an inexpensive way to share a communication channel, but it requires even more bandwidth than case (1).

(3) Linear Interpolation between DT Samples: Linear interpolation (i.e., straight lines) between the samples of the DT sequence a_n is obtained using the triangular pulse

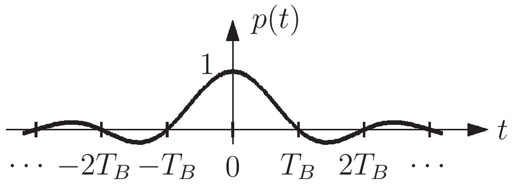
$$p(t) = \begin{cases} 1 + t/T_B\,, & -T_B \le t < 0\,, \\ 1 - t/T_B\,, & 0 \le t < T_B\,, \\ 0\,, & \text{otherwise}\,. \end{cases}$$



Straight line interpolation is a good compromise between bandlimitation of the CT signal (the main lobe of $P(f)$ contains 99.7% of the pulse energy) and ease of implementation.

(4) Minimum Bandwidth Interpolation: To obtain a minimum bandwidth CT waveform that interpolates between the samples of the DT sequence a_n, the "sin(x) over x" or "sinc" pulse

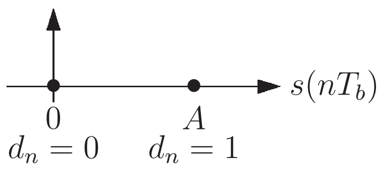
$$p(t) = \frac{\sin(\pi t/T_B)}{\pi t/T_B}$$



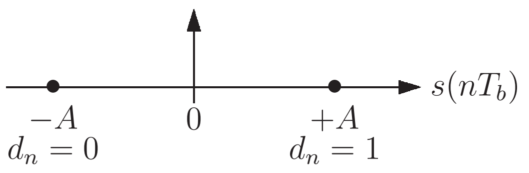
is used. Note that this pulse extends over the whole time axis from $-\infty$ to ∞. In practice the "tails" of this $p(t)$ need to be truncated and/or "windowed" to finite length, e.g., from $-kT_B$ to kT_B, with $k \approx 10 \ldots 20$, which destroys the property of strict bandlimitation to $1/(2T_B)$.

1.9 Unipolar and Polar Signaling

For a **unipolar** binary PAM signal $s(t)$ of amplitude A, the sampled values at times nT_b, where $F_b = 1/T_b$ is the bit rate, are $s(nT_b) \in \{0, A\}$ as shown graphically in the following figure.



For a **polar** binary PAM signal $s(t)$ of amplitude A, the sampled values at times nT_b are $s(nT_b) \in \{-A, +A\}$ as shown in the next figure.



If the channel is noisy, then the sampled values will deviate from their nominal values at either $\{0, A\}$ or $\{-A, +A\}$. How much noise can be tolerated before errors occur (e.g., a positive contribution from noise makes a 0 look more likely to be an A) depends on the distance between the nominal signal points, called the **minimum distance** of the signal set. Of course, this distance can be increased by making A larger, but that also requires more energy at the transmitter to send the signal.

For unipolar binary signaling the minimum distance is A. If both signal values are equally likely, then each bit sent requires energy proportional to $(0^2 + A^2)/2 = A^2/2$. Thus, the **normalized minimum distance** (normalized by dividing the distance by the square root of the average energy) is

$$d_{\min} = \frac{A}{\sqrt{A^2/2}} = \sqrt{2}\,, \qquad \text{unipolar binary signaling}\,.$$

For polar binary signaling the minimum distance is $2A$ and, assuming again equally likely signals, the energy required per bit at the transmitter is proportional to $(A^2+(-A)^2)/2 = A^2$. In this case the normalized minimum distance is

$$d_{\min} = \frac{2A}{\sqrt{A^2}} = 2\,, \qquad \text{polar binary signaling}\,.$$

To put it bluntly, a polar binary signal gives “more bang for the buck”, i.e., it gives more distance between the signal points per square root of joules of energy used at the transmitter. Thus, whenever possible, polar signaling is preferred over unipolar signaling.

1.10 Time Division Multiplexing

One method for sharing a communication channel among several users is **time-division multiplexing (TDM)**. This works by assigning one or more time slots to each user. The users then transmit during their assigned slot or slots, typically in round robin fashion from the first user to the last user and then starting over with the first user again.

Example: Suppose there are three users. User 1 has two time slots assigned and users 2 and 3 have one time slot each. Then they could share the channel using the following transmission pattern (1 means user 1 transmits, 2 means user 2 transmits, etc):

$$| \; 1 \; | \; 1 \; | \; 2 \; | \; 3 \; | \; 1 \; | \; 1 \; | \; 2 \; | \; 3 \; | \; 1 \; | \; 1 \; | \; 2 \; | \; 3 \; | \; \ldots$$

Alternatively, they could also use the pattern

$$| \; 1 \; | \; 2 \; | \; 1 \; | \; 3 \; | \; 1 \; | \; 2 \; | \; 1 \; | \; 3 \; | \; 1 \; | \; 2 \; | \; 1 \; | \; 3 \; | \; \ldots$$

The multiplexing for TDM can be done either in discrete or continuous time. For two users with data sequences $a_1[n]$ and $a_2[n]$ (of equal length or padded with zeros to make the lengths equal), both with baud rate $F_B = 1/T_B$ one of the following two methods could be used to produce a TDM signal $s(t)$:

1. **Two User DT Multiplexing.** In this case a DT sequence a_n is first obtained by multiplexing $a_1[n]$ and $a_2[n]$ as follows

$$\{a_n\} = \{a_1[0], a_2[0], a_1[1], a_2[1], a_1[2], a_2[2], a_1[3], a_2[3], \ldots\} \,.$$

   Note that a_n has baudrate $2F_B$. The CT signal $s(t)$ is then obtained as

$$s(t) = \sum_n a_n \, p(t - nT_B/2) \,,$$

   where $p(t)$ is a pulse designed for a PAM signal with symbol spacing $T_B/2$ in the time domain, i.e., if $p(t)$ is a rectangular pulse, its width is equal to $T_B/2$.

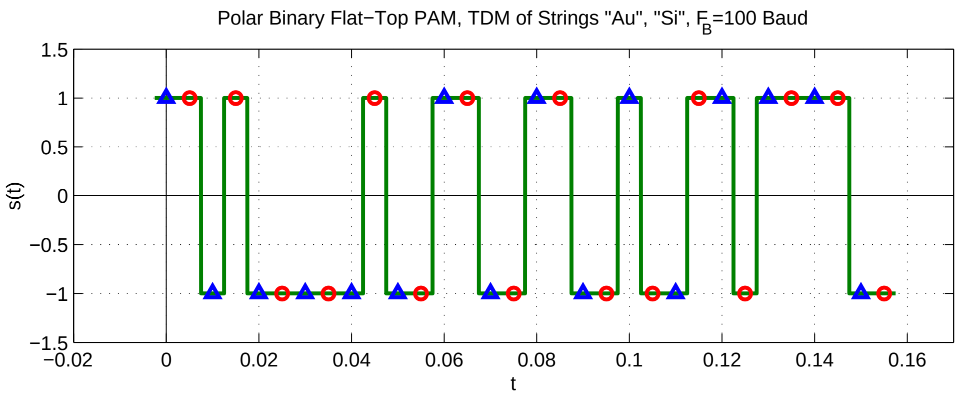
2. **Two User CT Multiplexing.** In this case two CT PAM signals

$$s_1(t) = \sum_n a_1[n] \, p(t - nT_B) \,, \qquad \text{and} \qquad s_2(t) = \sum_n a_2[n] \, p(t - nT_B) \,,$$

   each with baud rate $F_B = 1/T_B$, are first generated. The pulse $p(t)$, however, needs to be a pulse that is designed for a PAM signal with symbol spacing $T_B/2$ in the time domain, i.e., if $p(t)$ is a rectangular pulse, its width has to be $T_B/2$. The TDM signal $s(t)$ is then obtained by combining $s_1(t)$ and $s_2(t)$ as

$$s(t) = s_1(t) + s_2(t - T_B/2) \,.$$

The graph below shows an example of binary polar ($0 \to -1$, $1 \to +1$) 2-user TDM with rectangular $p(t)$. User 1 sends ASCII text "`Au`" (LSB-first: `10000010 10101110`) and user 2 sends ASCII text "`Si`" (LSB-first: `11001010 10010110`). The baud rate for each user is $F_B = 100$ bits/sec and the aggregate baud rate of both users is 200 bits/sec.



If the bitrate of each user stays fixed and more users are multiplexed, then the time slot for each transmitted symbol becomes more narrow. On a practical channel with bandlimitation, the number of users that can be time-division multiplexed at a fixed bitrate for each user is therefore limited.

1.11 Complex-Valued Time-Domain Signals

Symmetry Properties of FT for Real-Valued x(t). From the IFT

$$x(t) = \int_{-\infty}^{\infty} X(f)\, e^{j2\pi ft} df \;,$$

we obtain

$$x^*(t) = \Big(\int_{-\infty}^{\infty} X(f)\, e^{j2\pi ft} df\Big)^* = \int_{-\infty}^{\infty} X^*(f)\, e^{-j2\pi ft} df = \int_{-\infty}^{\infty} X^*(-f)\, e^{j2\pi ft} df \;,$$

where * denotes complex conjugate and $-f$ was substituted for f in the last equation. Thus, if $x(t)$ is real

$$x(t) = x^*(t) \quad \Longleftrightarrow \quad X(f) = X^*(-f) \;,$$

or

$$|X(f)|\, e^{j\angle X(f)} = |X(-f)|\, e^{-j\angle X(-f)} \;.$$

Therefore, if $x(t)$ is real-valued, its FT satisfies

$$|X(-f)| = |X(f)| \qquad \text{and} \qquad \angle X(-f) = -\angle X(f) \;,$$

i.e., the magnitude of $X(f)$ is an even function of f and the phase of $X(f)$ is an odd function of f.

The FT of a real-valued time function therefore exhibits redundancy in the sense that the spectrum for negative f can be derived unambiguously from the spectrum for positive f. Combining this observation with the frequency shift property of the FT, we see that we can obtain a real-valued bandpass signal from a complex-valued lowpass signal as follows. Let $x_L(t) \Leftrightarrow X_L(f)$ denote the complex-valued lowpass signal and its FT with frequencies limited to $|f| \leq f_L$. Then we can generate a complex-valued bandpass signal $x_u(t)$ at some frequency $f_c > f_L$ by writing

$$x_u(t) = x_L(t)\, e^{j2\pi f_c t} .$$

To convert this to a real-valued bandpass signal $x(t)$ we use

$$x(t) = Re\{x_u(t)\} = \frac{x_u(t) + x_u^*(t)}{2} \quad \Longleftrightarrow \quad \frac{X_u(f) + X_u^*(-f)}{2} = X(f) \, .$$

Note that since $f_c > f_L$ the spectra $X_u(f)$ and $X_u^*(-f)$ do not overlap and therefore $X_L(f)$ is not altered except for the translation in frequency by f_c. Since

$$X^*(-f) = \frac{X_u^*(-f) + X_u(f)}{2} = X(f) \, ,$$

the signal $x(t) \Leftrightarrow X(f)$ is therefore the real-valued bandpass signal corresponding to the complex-valued lowpass signal $x_L(t)$.

Example:

1.12 Sequences and Waveforms as Python Objects

Python has been an object-oriented language since it existed. Because of this, creating *classes* and using *objects*, which are members or *instances* of classes, is quite easy. Two kinds of objects that we repeatedly use in (digital) communications are *sequences* and (approximations to) *waveforms*. A sequence is characterized by its symbol (or Baud) rate F_B, its starting index n_0, and its length measured in symbols. A waveform is characterized by its sampling rate F_s, its starting time t_0, and its time length t_{len}. When we use object-oriented programming (OOP) we would like to create a sequence or a waveform as an object that has F_B or F_s and n_0 or t_0 as properties associated with each specific instance. We would also like to have *methods*, such as addition, or scaling, or concatenation, defined that are specific to a particular class of objects, like sequences and waveforms. Here is an example of a `class` definition `sigWave` for waveforms:

```
# Module: comsig.py
# Class and method definitions for COMmunication Signals
import numpy as np
import copy

class sigWave:
    """ Class for 'waveform' (CT) signals """
    type = 'waveform'
    def __init__(self, sig, Fs=8000, t0=0):
        """
        sig: real or complex-valued waveform samples
        Fs: sampling rate (default 8000 samples/sec)
        t0: start time of waveform in seconds (default 0)
        """
        self._sig = np.asanyarray(sig)
        self._Fs = Fs
        self._t0 = t0
        self._shape = np.shape(self._sig)
        if len(self._shape) > 1:
            self._Nsamp = len(self._sig[0])
        else:
            self._Nsamp = len(self._sig)
        self._tlen = self._Nsamp/float(self._Fs)
        self._tend = self._t0 + (self._Nsamp-1)/float(self._Fs)
```

A class is like a blueprint for creating objects. The keyword `__init__` represents a *constructor* in Python, i.e., the blueprint of how a new instance is constructed or initialized, e.g., by typing

```
import numpy as np
import comsig
Fs = 16000
f0 = 100
tlen = 0.1
tt = np.arange(np.round(tlen*Fs))/float(Fs)
# Instantiate waveform sig0 with sampling rate Fs and t0=-0.05
sig0 = comsig.sigWave(1.5*np.sin(2*np.pi*f0*tt),Fs,-0.05)
```

In general objects have properties and part of the class definition is usually concerned with getting and setting these properties. Here's what this looks like for the `sigWave` class (using the same indentation as `def __init__...`):

```
    # Properties
    def __len__(self):
        return self._Nsamp   # Returns length in samples
    def __str__(self):       # String representation of object
        return 'Fs={}, t0={}, tlen={}'.format(self._Fs,self._t0,self._tlen)
    __repr__ = __str__
    def get_shape(self):
        return self._shape   # Returns shape of signal array
    def get_Fs(self):
        return self._Fs      # Returns sampling rate
    def get_t0(self):
        return self._t0      # Returns start time
    def get_tlen(self):
        return self._tlen    # Returns length in seconds
    def get_avgpwr(self):    # Returns average power
        return np.mean(np.power(np.abs(self._sig),2.0))
    def get_tend(self):
        return self._tend    # Returns end time
    def set_t0(self, t0):
        self._t0 = t0        # Set new start time
        self._tend = self._t0 + (self._Nsamp-1)/float(self._Fs)
```

We also want to be able to use *methods* that change an object in a meaningful way, for example to scale a waveform or normalize it or raise it to some power `x`. Here are some simple methods that act on a single waveform from the `sigWave` class (using again the same indentation as `def __init__...`):

```
    # Methods
    def timeAxis(self):       # Generate time axis
        return self._t0 + np.arange(self._Nsamp)/float(self._Fs)
    def signal(self):         # Return the waveform
        return self._sig
    def copy(self):           # Make a copy of a sigWave object
        return copy.deepcopy(self)
    def normalized(self):     # Normalize the signal to -1,+1 range
        new_sig = 1.0/np.max(abs(self._sig))*self._sig
        return sigWave(new_sig, self._Fs, self._t0)
    def scale(self, factor):  # Make a scaled copy of a sigWave object
        return sigWave(factor*self._sig, self._Fs, self._t0)
    def pwrx(self, x):        # Raise the signal to power x
        return sigWave(np.power(self._sig, x), self._Fs, self._t0)
    def apwrx(self, x):       # Raise absolute value of signal to power x
        return sigWave(np.power(np.abs(self._sig), x), self._Fs, self._t0)
```

Some more complex methods like addition or concatenation that combine two waveforms from the `sigWave` class are shown next (using again the same indentation as `def __init__...`):

```
    # Methods (contd.)
    def __add__(self, other):
        """Add two sigWave signals, sample by sample"""
        if other == 0:
            return self
        assert self._Fs == other._Fs
        new_t0 = min(self._t0, other._t0)
        new_Nsamp = 1 + int(round(self._Fs*(max(self._tend,other._tend)-new_t0)))
        new_tend = new_t0 + (new_Nsamp-1)/float(self._Fs)
        if self._t0 == new_t0:
            new_self = np.hstack((self._sig,np.zeros(new_Nsamp-self._Nsamp)))
            new_other = np.hstack((np.zeros(new_Nsamp-other._Nsamp),other._sig))
        else:
            new_self = np.hstack((np.zeros(new_Nsamp-self._Nsamp),self._sig))
            new_other = np.hstack((other._sig,np.zeros(new_Nsamp-other._Nsamp)))
        new_sig = new_self + new_other
        return sigWave(new_sig, self._Fs, new_t0)
    def __or__(self, other):
        """Concatenate two waveforms"""
        assert self._Fs == other._Fs
        new_sig = np.hstack((self._sig,other._sig))
        return sigWave(new_sig, self._Fs, self._t0)
```

2 Lab Experiments

E1. "`showft`" to Approximate FT. Here is the header of a Python function, called `showft`, which computes and plots (a DFT/FFT approximation to) the FT of the CT signal $x(t)$ (after sampling with rate $F_s \Rightarrow x_n = x(n/F_s)$). The function `showft` is one of several "show" functions that we will develop and it is shown below as the first part of a Python module that we will call `showfun`.

```
# File: showfun.py
# "show" functions like showft, showpsd, etc
import numpy as np
import matplotlib.pyplot as plt

def showft(sig_xt, ff_lim):
    """
    Plot (DFT/FFT approximation to) Fourier transform of
    waveform x(t). Displays magnitude |X(f)| either linear
    and absolute or normalized (wrt to maximum value) in dB.
    Phase of X(f) is shown in degrees.
    >>>>> showft(sig_xt, ff_lim) <<<<<
    where  sig_xt: waveform from class sigWave
           sig_xt.timeAxis(): time axis for x(t)
           sig_xt.signal():   sampled CT signal x(t)
           ff_lim = [f1, f2, llim]
           f1:       lower frequency limit for display
           f2:       upper frequency limit for display
           llim = 0: display |X(f)| linear and absolute
           llim > 0: same as llim = 0 but phase is masked
                     (set to zero) for |X(f)| < llim
           llim < 0: display 20*log_{10}(|X(f)|/max(|X(f)|))
                     in dB with lower display limit llim dB,
                     phase is masked (set to zero) for f
                     with magnitude (dB, normalized) < llim
    """
```

The main function input is `sig_xt`, a waveform from class `sigWave`. The sampling rate of `sig_xt` is extracted using `sig_xt.get_Fs()`. A `sigWave` object also returns a time axis array starting at time `t0` using the method `sig_xt.timeAxis()` so that `showft` knows where $t = 0$ is located, which is necessary to display the phase of the FT correctly.

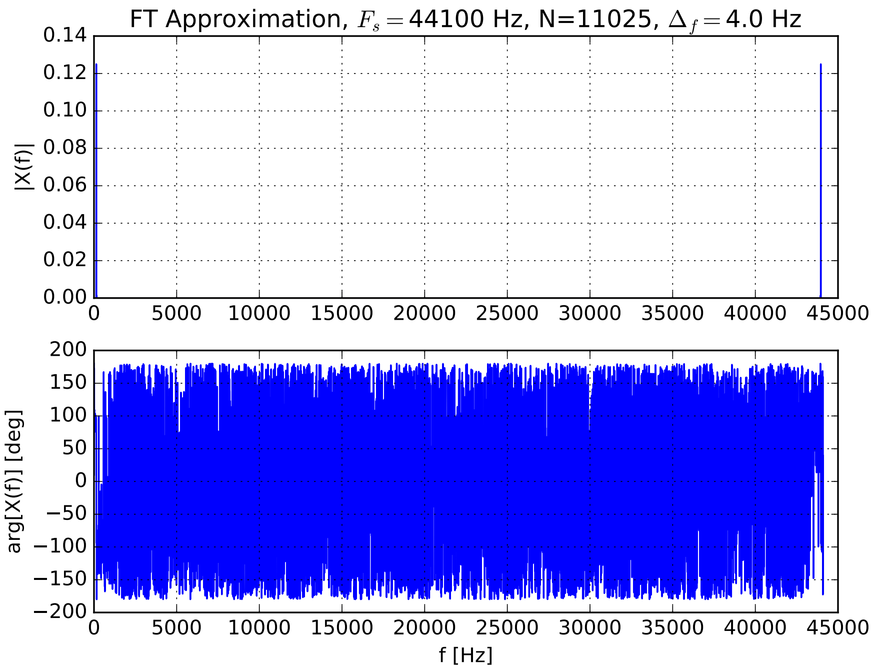
The following code should get you started with the basic functionality of `showft`.

```
    # ***** Prepare x(t), swap pos/neg parts of time axis *****
    N = len(sig_xt)           # Blocklength of DFT/FFT
    Fs = sig_xt.get_Fs()      # Sampling rate
    tt = sig_xt.timeAxis()    # Get time axis for x(t)
    ixp = np.where(tt>=0)[0]  # Indexes for t>=0
    ixn = np.where(tt<0)[0]   # Indexes for t<0
    xt = sig_xt.signal()      # Get x(t)
    xt = np.hstack((xt[ixp],xt[ixn]))
                              # Swap pos/neg time axis parts
    # ***** Compute X(f), make frequency axis *****
    Xf = np.fft.fft(xt)/float(Fs)    # DFT/FFT of x(t),
                              # scaled for X(f) approximation
    ff = Fs*np.array(np.arange(N),np.int64)/float(N)  # Frequency axis
    # ***** Compute |X(f)|, arg[X(f)] *****
    absXf = np.abs(Xf)        # Magnitude |X(f)|
    argXf = np.angle(Xf)      # Phase arg[X(f)]
    # ***** Plot magnitude/phase *****
    f1 = plt.figure()
    af11 = f1.add_subplot(211)
    af11.plot(ff,absXf)         # Plot magnitude
    af11.grid()
    af11.set_ylabel('$|X(f)|$')
    strgt = 'FT Approximation, $F_s=$' + str(Fs) + ' Hz'
    strgt = strgt + ', $N=$' + str(N)
    strgt = strgt + ', $\Delta_f$={0:3.2f}'.format(Fs/float(N)) + ' Hz'
    af11.set_title(strgt)
    af12 = f1.add_subplot(212)
    af12.plot(ff,180/np.pi*argXf)  # Plot phase in degrees
    af12.grid()
    af12.set_ylabel('$arg[X(f)]$ [deg]')
    af12.set_xlabel('$f$ [Hz]')
    plt.show()
```

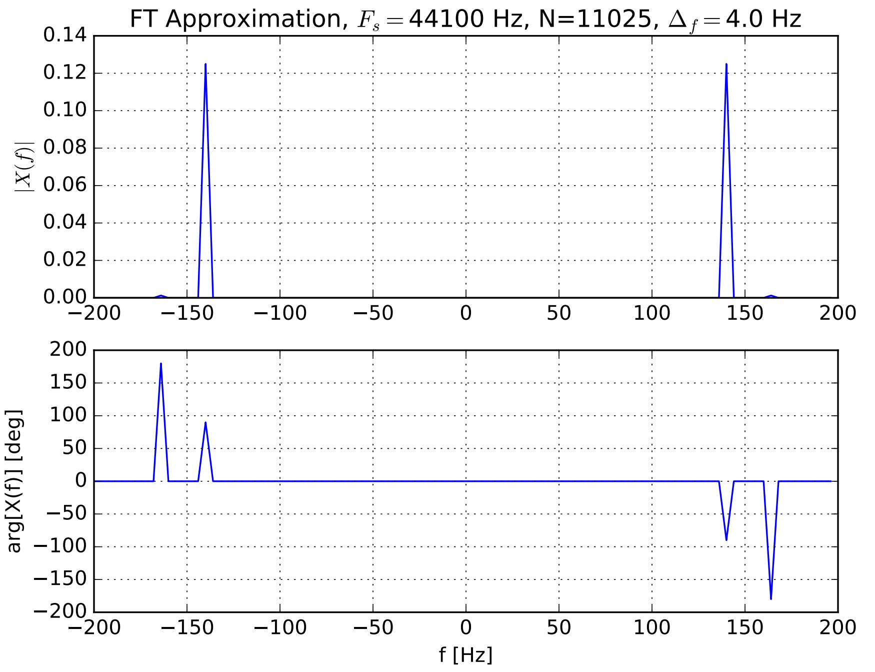
(a) To try out `showft`, run the following sequence of commands in Python:

```
import numpy as np
import comsig
import showfun                # Note: May have to change directory
                              # to point to showfun.py
Fs = 44100                    # Sampling rate
fa, fb = 140, 164             # Frequencies fa, fb
tlen = 0.25                   # Length of t-axis in sec
tt = np.arange(round(Fs*tlen))/float(Fs)  # Time axis
xt = np.sin(2*np.pi*fa*tt)-0.01*np.cos(2*np.pi*fb*tt)
                              # Linear combination of sinusoids
sig_xt = comsig.sigWave(xt, Fs, 0)  # Waveform from class sigWave
showfun.showft(sig_xt,[-200,200,1e-3])  # Display X(f), using ff_lim
```

Note that this generates a sine of amplitude 1 at frequency $f_a = 140$ Hz and a negative cosine of much smaller amplitude 0.01 at frequency $f_b = 164$ Hz. The resulting (disappointing) plot is shown below.



This is all correct in principle, but would be much easier to interpret if it looked like this:



To obtain this second set of graphs, you need to improve the code of the `showft` function as follows:

(i) The `ff_lim = [f1,f2,llim]` parameter set needs to control the horizontal display of the FT as follows. The frequency axis must be limited to $f_1 \leq f < f_2$. Note that the result of the FFT computation is a vector of DFT coefficients X_k for $k = 0, 1, \ldots N-1$. Since the DFT is periodic with period N, index N corresponds to frequency F_s in Hz and thus index k corresponds to frequency kF_s/N. If `f1` in `ff_lim` is negative, then the parts of the frequency axis for $f \geq F_s/2$ and $f < F_s/2$ (and the corresponding DFT coefficients in `Xf`) must be swapped. This comes from the fact that the DFT coefficients satisfy $X_{N-i} = X_{-i}$ and thus the negative frequency components for the FT approximation are obtained from $X_{-1} = X_{N-1}$, $X_{-2} = X_{N-2}$, etc.

(ii) The `ff_lim = [f1,f2,llim]` parameter set needs to control the vertical display of the FT as follows. If `llim>0`, the phase plot needs to be set to zero (masked) when the magnitude of `Xf` is less than `llim`. The reason for this is that any useful phase information may be buried in (numerical) noise if we compute `angle(Xf)` no matter how small the magnitude of `Xf` is.

After making these changes to `showft`, run the commands

```
import importlib
importlib.reload(showfun)            # Reload showfun after changes
showfun.showft(sig_xt,[-200,200,1e-3])  # Display X(f), using ff_lim
```

again and check that the result looks right (both magnitude and phase).

Question: The sine at $f_a = 140$ Hz in `xt` has amplitude 1, but the "spikes" of $|X(f)|$ at $f = \pm f_a$ only have amplitude 0.125. Is that right? **Hint:** For CT impulse functions $\delta(.)$ the area underneath the impulse determines the "size" of the impulse.

(b) The signal

```
xt = np.sin(2*np.pi*fa*tt)-0.01*np.cos(2*np.pi*fb*tt)
```

that you generated and displayed in (a) is a linear combination of two sinusoids, but only one is visible in the magnitude plot of $X(f)$. If you try

```
x2t = np.sin(2*np.pi*fa*tt)+0.01*np.cos(2*np.pi*fb*tt)
sig_x2t = comsig.sigWave(x2t, Fs, 0)
showfun.showft(sig_xt2,[-200,200,1e-3])  # Display X(f), using ff_lim
```

then you won't even see from the graph (phase plot) that there are two sinusoids in $x(t)$. The reason for this is that the amplitude of the cosine (0.01) is too small to be seen alongside the amplitude of the sine (1) in a graph with linear vertical axis and since the phase of cosine is zero, it also does not show up in the phase plot. Therefore one more feature needs to be added to the `showft` function as follows:

(iii) When `llim<0` in `ff_lim = [f1,f2,llim]`, the display of the FT is modified as follows. The magnitude of $X(f)$ is displayed in decibels (dB) as

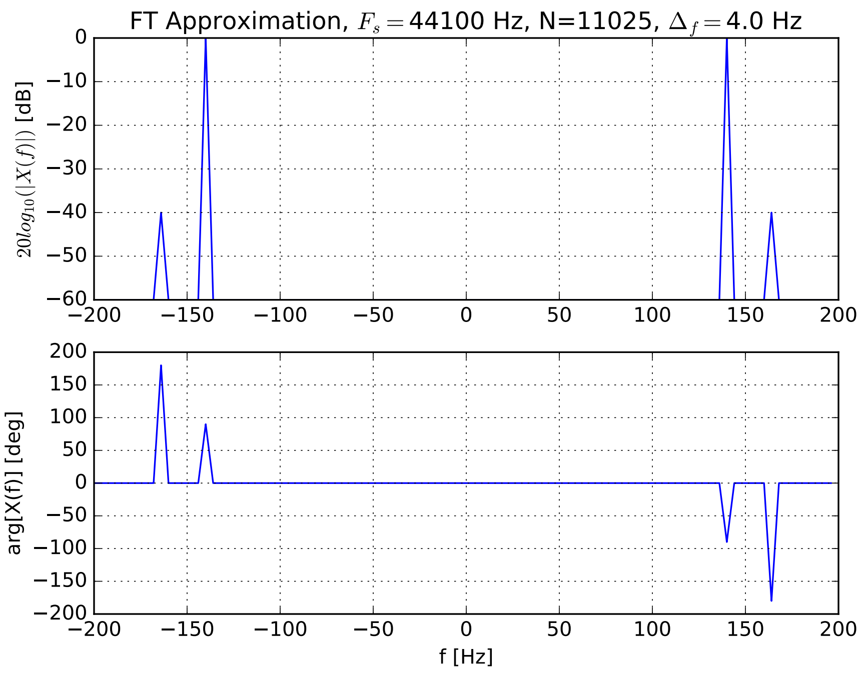
$$20\log_{10}\Big(\frac{|X(f)|}{\max_f(|X(f)|)}\Big) .$$

The division by $\max_f(|X(f)|)$ normalizes the display and sets the overall maximum at 0 dB. This makes it easy to determine, for instance, the -3dB bandwidth of a filter. Since $X(f)$ may have zero crossings and $\log 0 = -\infty$, a lower limit of value `llim` in dB must also be imposed on the magnitude for $X(f)$ to obtain a reasonably scaled display. The same limit is also used to mask the phase of $X(f)$ (i.e., set it to zero) when the magnitude of $X(f)$ is less than `llim` in dB.

Check your modifications of `showft` by running the commands

```
importlib.reload(showfun)       # Reload showfun after modifications
xt = np.sin(2*np.pi*fa*tt)-0.01*np.cos(2*np.pi*fb*tt)
sig_xt = comsig.sigWave(xt, Fs, 0)
showfun.showft(sig_xt,[-200,200,-60]) # Display X(f) in dB, using ff_lim
```

with `fa`, `fb`, and `tt` as defined previously. The result should look similar to the following magnitude and phase plots.



(c) Use your `showft` function to plot $P(f)$ (magnitude in normalized dB, `llim=-40`, phase in degrees) in the range -2000 Hz to 2000 Hz for a rectangular PAM pulse $p(t)$ with amplitude

1 and pulsewidth $T_B = 1/100$ sec, symmetric around $t = 0$. Choose F_s and N so that the frequency resolution is about 10 Hz and the effect of aliasing due to sampling at rate F_s is negligible in the frequency range of the plot. Check that your display looks right and apply corrections to `showft` and/or your parameters as necessary.

E2. General PAM Transmitter "`pam10`". First we define a new class called `sigSequ` for *sequences* as follows (this is part of the `comsig.py` module which imports `copy` and `numpy` as `np`):

```
class sigSequ:
    """ Class for 'sequence' (DT) signals """
    type = 'sequence'
    def __init__(self, sig, FB = 100, n0 = 0):
        """
        sig: real or complex-valued sequence values
        FB: symbol (or Baud) rate (default 100 Baud)
        n0: start index of sequence (default 0)
        """
        self._sig = np.asanyarray(sig)
        self._FB = FB
        self._n0 = n0

    # Properties
    def __len__(self):
        return len(self._sig)
    def get_size(self):
        return self._sig.size
    def get_shape(self):
        return self._sig.shape
    def get_FB(self):
        return self._FB
    def get_n0(self):
        return self._n0

    # Methods
    def indexAxis(self):
        return self._n0 + np.arange(len(self._sig))
    def signal(self):
        return self._sig
    def copy(self):
        return copy.deepcopy(self)
    def scale_offset(self, a, b = 0):
        """ x[n]_out = a*x[n]_in + b """
        return sigSequ(a*self._sig+b, self._FB, self._n0)
```

The PAM transmitter function `pam10` outlined below is based on the principle of first converting the DT sequence a_n in `sig_an` to the CT signal $a_s(t)$ and then passing the result through a shaping filter with $h(t) = p(t)$, as explained in the introduction, to generate a PAM signal $s(t)$ in `sig_st` with pulse shape $p(t)$. The header of this function, which is the first part of a Python module `pamfun.py` that we will generate in this and later labs and use for PAM modulation and demodulation functions, looks as follows.

```
# File: pamfun.py
# Functions for pulse amplitude modulation (PAM)
import numpy as np
import comsig

def pam10(sig_an, Fs, ptype, pparms=[]):
    """
    Pulse amplitude modulation: a_n -> s(t), (n0-1/2)*TB<=t<(N+n0-1/2)*TB,
    V1.0 for 'rect', 'sinc', and 'tri' pulse types.
    >>>>> sig_st = pam10(sig_an, Fs, ptype, pparms) <<<<<
    where  sig_an: sequence from class sigSequ
           sig_an.signal():  N-symbol DT input sequence a_n, n0<=n<N+n0
           sig_an.get_FB():  Baud rate of a_n, TB=1/FB
           sig_an.get_n0():  Start index
           Fs:    sampling rate of s(t)
           ptype: pulse type ('rect','sinc','tri')
           pparms not used for 'rect','tri'
           pparms = [k, beta]  for 'sinc'
           k:     "tail" truncation parameter for 'sinc'
                  (truncates p(t) to -k*TB <= t < k*TB)
           beta:  Kaiser window parameter for 'sinc'
           sig_st: waveform from class sigWave
           sig_st.timeAxis():  time axis for s(t), starts at (n0-1/2)*TB
           sig_st.signal():    CT output signal s(t),
                               (n0-1/2)*TB<=t<(N+n0-1/2)*TB,
                               with sampling rate Fs
    """
```

The following commands implement `pam10` for rectangular $p(t)$.

```
    N = len(sig_an)            # Number of data symbols
    FB = sig_an.get_FB()       # Baud rate
    n0 = sig_an.get_n0()       # Starting index
    ixL = int(ceil(Fs*(n0-0.5)/float(FB)))    # Left index for time axis
    ixR = int(ceil(Fs*(N+n0-0.5)/float(FB)))  # Right index for time axis
    tt = np.arange(ixL,ixR)/float(Fs)  # Time axis for s(t)
    t0 = tt[0]                 # Start time for s(t)
    # ***** Conversion from DT a_n to CT a_s(t) *****
    an = sig_an.signal()       # Sequence a_n
    ast = np.zeros(len(tt))       # Initialize a_s(t)
    ix = np.array(np.around(Fs*(np.arange(0,N)+n0)/float(FB)),int)
                               # Symbol center indexes
    ast[ix-int(ixL)] = Fs*an   # delta_n -> delta(t) conversion
    # ***** Set up PAM pulse p(t) *****
    ptype = ptype.lower()      # Convert ptype to lowercase
                               # Set left/right limits for p(t)
    if (ptype=='rect'):
          kL = -0.5; kR = -kL
    else:
        kL = -1.0; kR = -kL    # Default left/right limits
    ixpL = int(ceil(Fs*kL/float(FB)))   # Left index for p(t) time axis
    ixpR = int(ceil(Fs*kR/float(FB)))   # Right index for p(t) time axis
    ttp = np.arange(ixpL,ixpR)/float(Fs)  # Time axis for p(t)
    pt = np.zeros(len(ttp))    # Initialize pulse p(t)
    if (ptype=='rect'):        # Rectangular p(t)
        ix = np.where(np.logical_and(ttp>=kL/float(FB), ttp<kR/float(FB)))[0]
        pt[ix] = np.ones(len(ix))
    else:
        print("ptype '%s' is not recognized" % ptype)
    # ***** Filter with h(t) = p(t) *****
    st = np.convolve(ast, pt)/float(Fs)  # s(t) = a_s(t)*p(t)
    st = st[-ixpL:ixR-ixL-ixpL]  # Trim after convolution
    return comsig.sigWave(st, Fs, t0)  # Return waveform from sigWave class
```

(a) Complete the `pam10` function so that it can also generate PAM signals based on triangular $p(t)$ and truncated and windowed "sinc" $p(t)$. A truncated "sinc" pulse with first zero crossing at T_B and taillength kT_B is defined as

$$p(t) = \begin{cases} \dfrac{\sin(\pi t/T_B)}{\pi t/T_B} = \text{sinc}(t/T_B)\,, & -kT_B \le t < kT_B\,, \\ 0\,, & \text{otherwise} \end{cases}$$

The abrupt truncation to kT_B destroys the property of strict bandlimitation and leads to a spectrum with "ringing" (Gibbs phenomenon) at the band edges. To reduce this effect a windowed version of $p(t)$ can be used, defined as

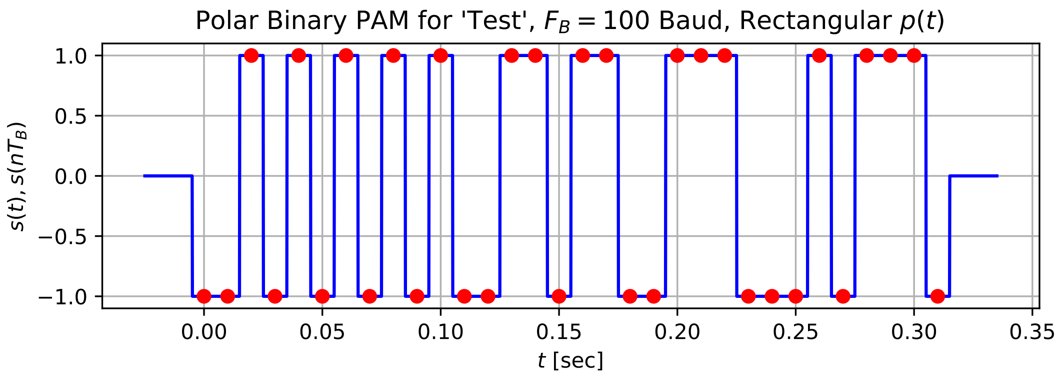
$$p_w(t) = p(t)\, w(t)\,,$$

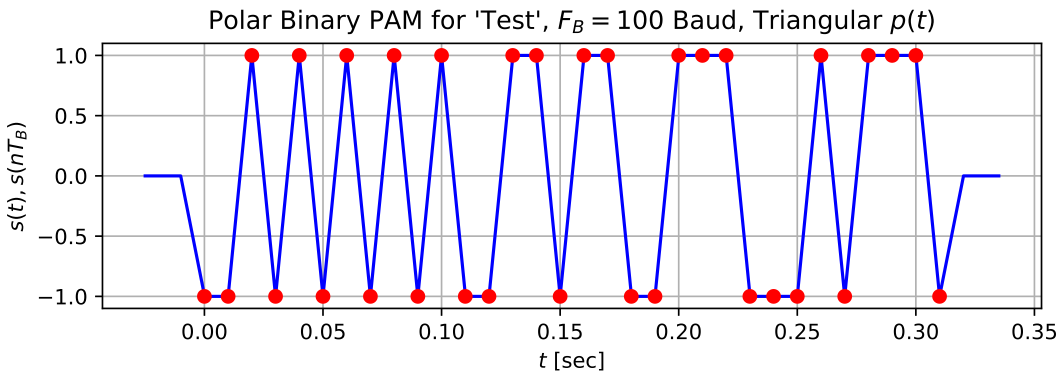
where $w(t)$ is a window function that extends from $t = -kT_B$ to $t = kT_B$. A good, universal window function is the Kaiser window with parameter β. Assuming that `pt` contains the truncated "sinc" $p(t)$, the Kaiser windowed version `pwt` is obtained in Python using the command

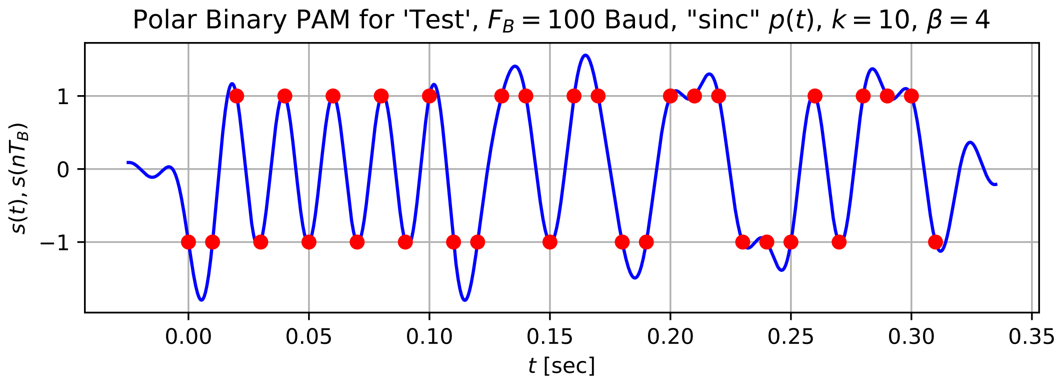
```
import numpy as np
pwt = pt*np.kaiser(len(pt),beta)    # Pulse p(t), Kaiser windowed
```

where `beta` is a parameter in the range of about 0 to 8, where 0 corresponds to a rectangular window.

Examples for $s(t)$ for rectangular, triangular, and "sinc" $p(t)$ are shown in the following graphs. Note that a_n was preceded by two 0's and padded with two 0's before $s(t)$ was generated.







(b) Generate a random polar binary message sequence of length approximately 0.5 sec using the Python commands

```
import numpy as np
import comsig
FB = 100                          # Baud rate
dn = np.random.rand(round(FB/2.0))  # random sequence, uniform in [0...1)
dn = np.array(np.floor(2*dn),int)   # unipolar binary DT sequence
an = 2*dn - 1                       # polar binary DT sequence
sig_an = comsig.sigSequ(an, FB, 0)  # Random polar binary 'sigSequ' sequence
```

Then use the `pam10` function to generate PAM signals $s(t)$ from `an` with rectangular, triangular, and "sinc" ($k = 10$, $\beta = 2$) $p(t)$ for $F_B = 100$ baud and $F_s = 44100$ Hz. Use `showft` to plot $S(f)$ (magnitude in dB, lower limit -60 dB) in the range $f = -1000$ to $f = 1000$ Hz for the rectangular and triangular $p(t)$ and in the range $f = -200$ to $f = 200$ Hz for the "sinc" $p(t)$. Compare the spectra of the three cases.

E3. Analysis of PAM Signals. (Experiment for ECEN 5002, optional for ECEN 4652)
(a) The PAM signals in `pamsig201.wav`, `pamsig202.wav`, and `pamsig203.wav` are polar binary PAM signals that were obtained from ASCII (8-bit, LSB first, $0 \rightarrow -1$, $1 \rightarrow +1$) texts. Look at the signals in the time and frequency domains and, if possible, determine the baud rates F_B, and the pulse shapes $p(t)$ that were used. All $p(t)$ are based on rectangular, triangular, or "sinc" pulses, but one signal uses a $p(t)$ that is a little different from the ones given in the introduction (Hint: It is a pulse with a zero dc component). After analyzing the PAM signals, try to recover the ASCII text using appropriate sampling parameters. Keep in mind that the `.wav` files start at $t = -T_B/2$ and the symbol interval lengths are T_B.

(b) Repeat (a) with appropriate modifications for the 2-user TDM signal in `pamsig204.wav`. Both users use the same baud rate F_B and the multiplexing is done on a bit-by-bit basis.