Lab 1: Python Intro, Flat-Top PAM, PCM

1 Introduction

1.1 Baseband Signaling

A simple communication scenario is the transmission of a text string over a waveform channel such as a pair of twisted wires or a wireless radio frequency (RF) link. Starting from simple baseband signaling over an ideal noiseless channel and later proceeding to more complicated situations, many important aspects of the theory and practice of communication engineering can be studied in this context. One goal of this lab is to introduce Python as a high-level tool for signal processing and, in particular, for the generation and reception of communication signals. Another goal is to be able to transmit and receive ASCII text strings in Python using flat-top PAM (pulse amplitude modulation) which is a simple baseband signaling technique. An extension of this is to use pulse code modulation (PCM) to also transmit analog message signals.

Communication signals can be broadly divided into **baseband signals** and into **bandpass signals** (or passband signals). One way to define baseband and bandpass signals is as follows.

Definition: A communication signal is called a **baseband signal** if the filter of smallest bandwidth (BW) that passes the signal essentially undistorted is a LPF (lowpass filter) with passband $0 \leq f < f_L$. Conversely, if this filter of smallest BW is a BPF (bandpass filter) with passband $0 < f_1 < f < f_2$, then the corresponding signal is called a **bandpass signal**.

1.2 ASCII Code

The ASCII (American Standard Code for Information Interchange) code is a 7-bit code for encoding upper/lower case characters from the English alphabet, as well as numbers, punctuation marks, and control characters. The following table shows all the possible 128 code sequences of the original ASCII code.

7-Bit ASCII (American Standard Code for Information Interchange)								
	000...	001...	010...	011...	100...	101...	110...	111...
..0000	*NUL*	*DLE*	*SP*	0	@	P	`	p
..0001	*SOH*	*DC1*	!	1	A	Q	a	q
..0010	*STX*	*DC2*	"	2	B	R	b	r
..0011	*ETX*	*DC3*	#	3	C	S	c	s
..0100	*EOT*	*DC4*	$	4	D	T	d	t
..0101	*ENQ*	*NAK*	%	5	E	U	e	u
..0110	*ACK*	*SYN*	&	6	F	V	f	v
..0111	*BEL*	*ETB*	'	7	G	W	g	w
..1000	*BS*	*CAN*	(	8	H	X	h	x
..1001	*HT*	*EM*	)	9	I	Y	i	y
..1010	*LF*	*SUB*	*	:	J	Z	j	z
..1011	*VT*	*ESC*	+	;	K	[	k	{
..1100	*FF*	*FS*	,	<	L	\	l	|
..1101	*CR*	*GS*	-	=	M	]	m	}
..1110	*SO*	*RS*	.	>	N	^	n	~
..1111	*SI*	*US*	/	?	O	_	o	*DEL*

The first two columns consist of non-printable control characters, e.g., *CR* stands for carriage return and *LF* stands for line feed. The space character (ASCII code 0100000) is shown as *SP* in the table. In the teletype days transmissions typically started with a few synchronization characters (*SYN*), followed by a start of text character (*STX*). After that the actual message would be sent and at the end terminated with an end of text character (*ETX*). The main purpose of characters like *SYN*, *STX*, and *ETX* was to synchronize the transmitting and the receiving teletype machines.

The binary codes in the ASCII table above are shown with the MSB (most significant bit) on the left and the LSB (least significant bit) on the right. Because the wordlengths used in computers are typically powers of 2, it is quite common to extend the ASCII code by adding a zero to the left of the MSB to make 8-bit codewords as shown in the following example.

"Test" in Extended (8-bit) ASCII	
Character	**Extended ASCII Code**
`T`	`01010100`
`e`	`01100101`
`s`	`01110011`
`t`	`01110100`

1.3 Parallel to Serial Conversion

Using the extended 8-bit ASCII code, the letter "T", for instance, is encoded in binary as `01010100`. To transmit this 8-bit quantity in binary over a communication channel, each bit must be read out and transmitted indivdually, according to a specific scheme that is known to both the transmitter and receiver. The two most common schemes are to either read the bits out from left to right and therefore transmit **MSB-first**, or to read from right to left and in this case transmit **LSB-first**. In either case the codeword for the first character of a message is sent first, followed by the codeword of the second character, etc.

The following two tables show the serial data sequence d_n, $n \geq 0$, for the string "`Test`" using MSB-first and LSB-first parallel (8-bit) to serial conversion.

MSB-first Bit Sequence for "Test" (Extended 8-bit ASCII)
$d_n =$ `01010100 01100101 01110011 01110100` $\rightarrow$ Index n increases from left to right $\rightarrow$
d_0=0, d_1=1, d_2=0, d_3=1, d_4=0, d_5=1, d_6=0, d_7=0, d_8=0, d_9=1, d_{10}=1, ...

LSB-first Bit Sequence for "Test" (Extended 8-bit ASCII)
$d_n =$ `00101010 10100110 11001110 00101110` $\rightarrow$ Index n increases from left to right $\rightarrow$
d_0=0, d_1=0, d_2=1, d_3=0, d_4=1, d_5=0, d_6=1, d_7=0, d_8=1, d_9=0, d_{10}=1, ...

The spaces after every 8 bits are shown only for ease of reading. In an actual data transmission all 32 bits would be sent consecutively, without any spaces or pauses.

In practice, the LSB-first scheme is the one that is most commonly used for the transmission of ASCII text.

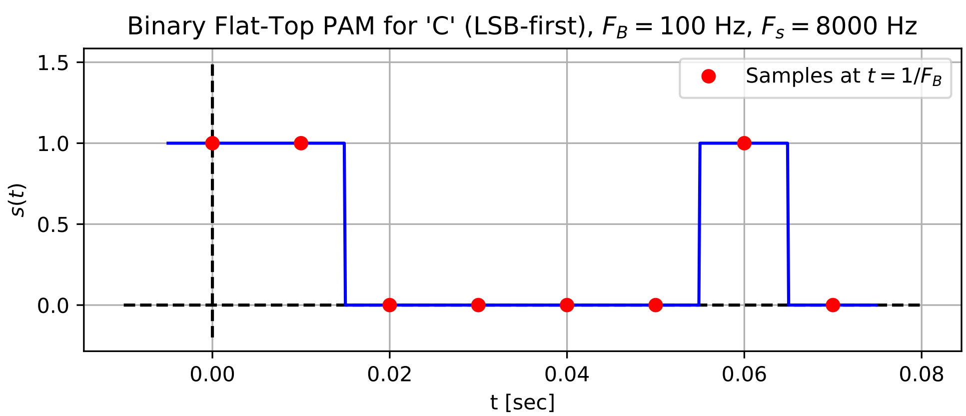
1.4 Flat-Top PAM

Most physical communication channels, like twisted-pair wire, coaxial cable, free-space radio frequency (RF) transmission, etc, are analog in nature. This implies in particular that they require a continuous-time (CT) signal or waveform at the input. A discrete-time (DT) sequence, like d_n in the ASCII examples of the previous section, therefore needs to be converted to a CT signal $s(t)$ before it can be transmitted over a waveform channel. A simple and straightforward rule, informally called “flat-top” PAM (pulse amplitude modulation), to obtain a waveform $s(t)$ from d_n with bit rate $F_b = 1/T_b$ is to set

$$s(t) = d_n \,, \qquad (n-1/2)\,T_b \le t < (n+1/2)\,T_b \,.$$

In blockdiagram form, the function of a flat-top PAM transmitter is represented as shown in the following figure.

The graph below shows the letter “`C`” (ASCII code `01000011`) converted (LSB-first) to a binary flat-top PAM signal $s(t)$.



A DT sequence d_n, $n \ge 0$, can also be expressed as

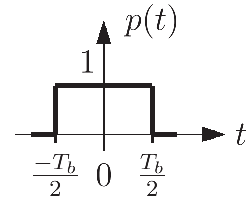
$$\{d_n\} = \{d_0\,\delta_n + d_1\,\delta_{n-1} + d_2\,\delta_{n-2} + \ldots\} \,,$$

where $\delta_k = 1$ if $k = 0$ and $\delta_k = 0$ otherwise. Each $d_m\,\delta_{n-m}$ gets converted to a piece, from $(m-1/2)\,T_b$ to $(m+1/2)\,T_b$, of the signal $s(t)$. Thus, another way to describe flat-top PAM mathematically is

$$s(t) = \sum_{n=0}^{\infty} d_n\, p(t - nT_b) \,,$$

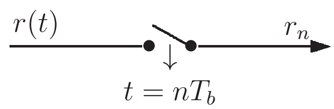
where d_n is a binary sequence with bitrate $F_b = 1/T_b$, and $p(t)$ is the rectangular pulse shown in the following figure.

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



1.5 Flat-Top PAM Receiver

In any communication system the task of the receiver is to reproduce the transmitted information as faithfully as possible. For flat-top PAM, the receiver initially simply consists of a sampler that samples the received signal $r(t)$ at times $t = nT_b$, so that $r_n = r(nT_b)$ as shown in the following blockdiagram.



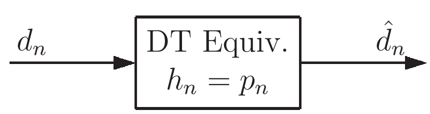
In the absence of noise and other channel degradations, $r(t) \approx s(t)$ and thus $r_n = \hat{d}_n$, the estimate of the received sequence d_n. In fact, setting $r(t) = s(t)$,

$$\hat{d}_n = r_n = s(t)\Big|_{t=nT_b} = \Big[\sum_{m=0}^{\infty} d_m p(t - mT_b)\Big]\Big|_{t=nT_b} = \sum_{m=0}^{\infty} d_m p(nT_b - mT_b)\,.$$

Defining the sampled version of $p(t)$ as

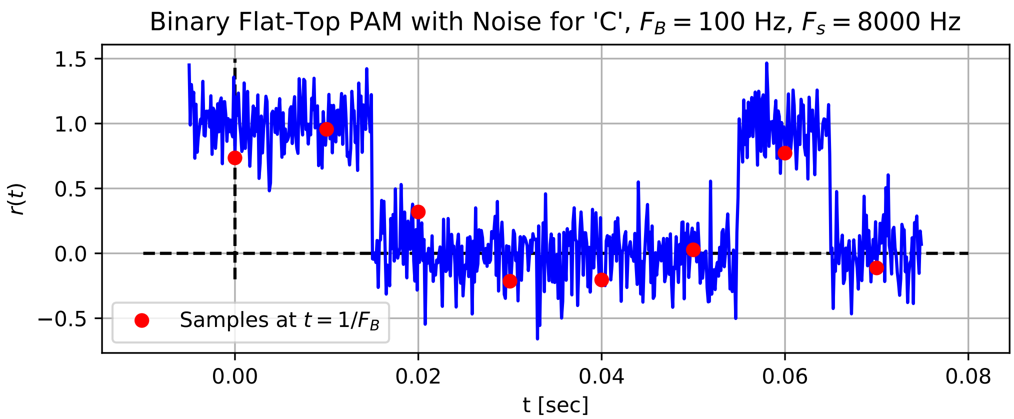
$$p_n = p(nT_b) \quad \Longrightarrow \quad \hat{d}_n = \sum_{m=0}^{\infty} d_m\, p_{n-m} = d_n * p_n\,.$$

Thus, the DT equivalent of PAM followed by sampling at the receiver is convolution between the input sequence d_n and the pulse samples p_n, as shown in the next blockdiagram.

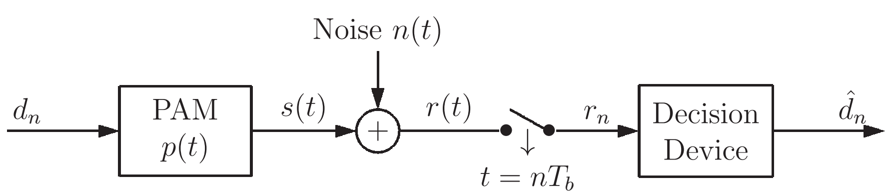


The rectangular pulse $p(t)$ that is used for flat-top PAM satisfies $p_n = p(nT_b) = \delta_n$ and thus the DT convolution is trivial. But, as you will see later, the same DT equivalent blockdiagram can be used for PAM with more general $p(t)$ and thus more general p_n.

But suppose now that noise gets added to $s(t)$ during the transmission and the received signal becomes $r(t) = s(t) + n(t)$, where $n(t)$ is noise. An example of how $r(t)$ might look for the letter "`C`" (8-bit ASCII, LSB-first) is shown in the graph below.

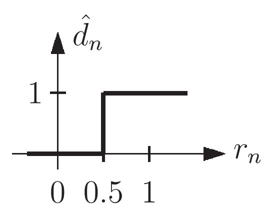


Now just sampling the received signal at the right time and converting it back to ASCII text will not work because r_n is a real number and not just 0 or 1. Thus, a decision device needs to be introduced after the sampler as shown in the blockdiagram below.



The characteristic of the decision device is shown below. Assuming that the statistics of positive and negative noise levels are similar and that the noise is independent of the transmitted data, the threshold of the decision device is chosen halfway between the “legal” values of 0 and 1.

$$\hat{d}_n = \begin{cases} 0\,, & r_n < 1/2\,, \\ 1\,, & r_n \geq 1/2\,. \end{cases}$$

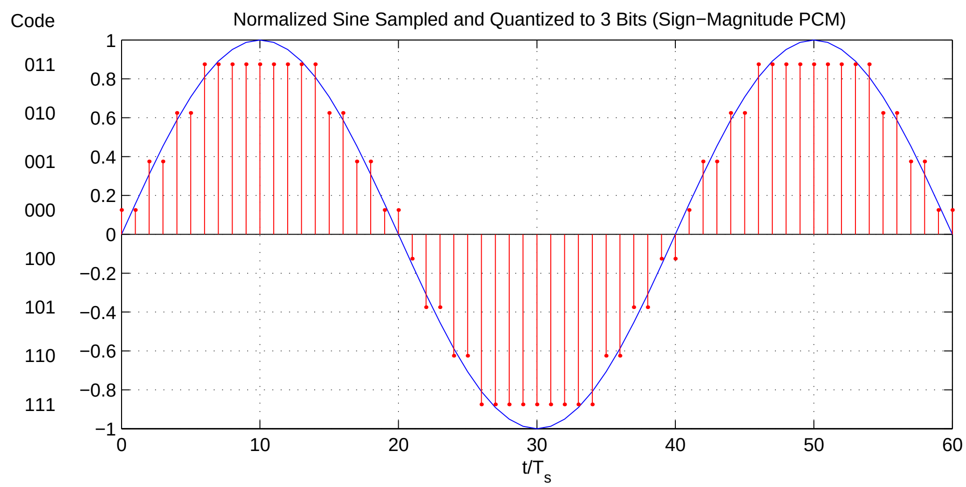


Note that, no matter how the threshold is chosen, errors will occur as the noise gets stronger and the signal-to-noise ratio (SNR) decreases. The structure of an optimum receiver that maximizes the SNR before using the decision device will be derived later on.

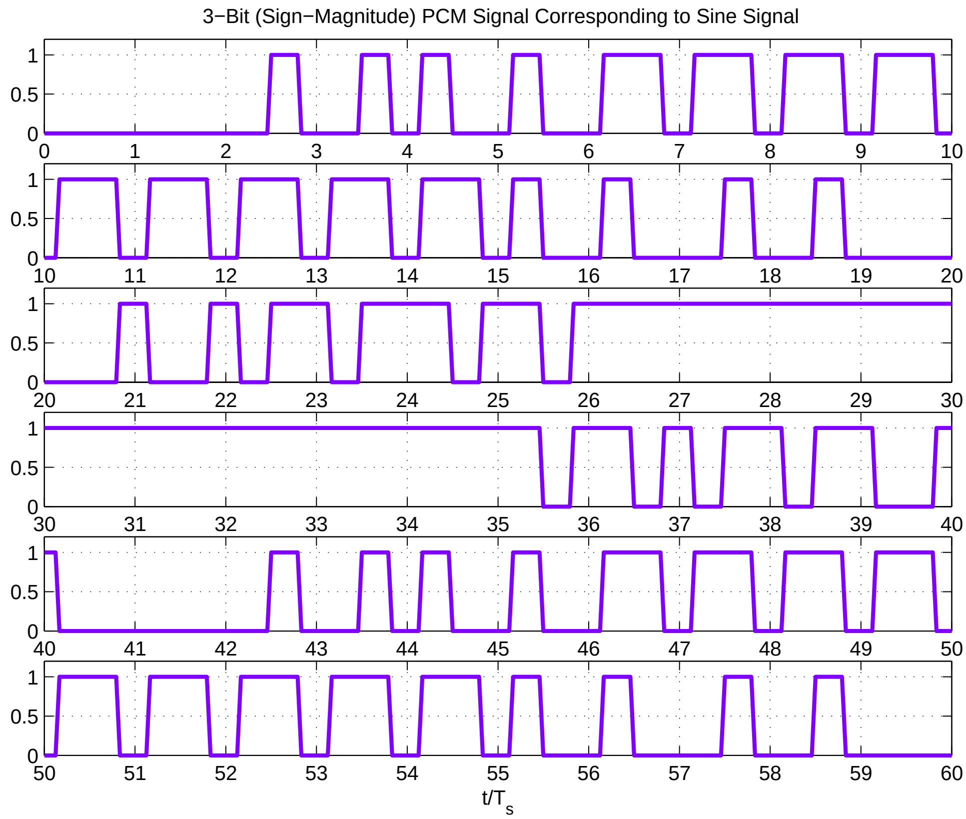
1.6 Pulse Code Modulation

Pulse code modulation (PCM) was invented by Alec Reeves in 1937 to obtain a digital representation of analog message signals $m(t)$. In essence, $m(t)$ is sampled at rate F_s samples per second and then each sample is quantized to b bits which are in turn transmitted serially, e.g., using flat-top PAM. In telephony $F_s = 8000$ samples per second and $b = 8$ are the most commonly used parameters for individual subscriber lines, resulting in a binary signal with bit rate $F_b = 64$ kbit/sec. One advantage of using PCM is that several signals can easily be multiplexed in time so that they can share a single communication channel. A T1 carrier, for example, is used in telephony to transmit 24 multiplexed PCM signals with a total rate of 1.544 Mbit/sec (this includes some overhead for synchronization). A second advantage is that repeaters that need to be used to compensate for losses over large distances can (within some limits) restore the signal perfectly because only two signal levels need to be distinguished.

Example: 3-Bit PCM for a Sinewave. The figure below shows a normalized sinewave (blue line) and its sampled and quantized version (red stem plot). A 3-bit quantizer with sign-magnitude output was used. The sign-magnitude format (rather than 2's complement) reduces the sensitivity for small amplitude values to sign errors that may occur during transmission.

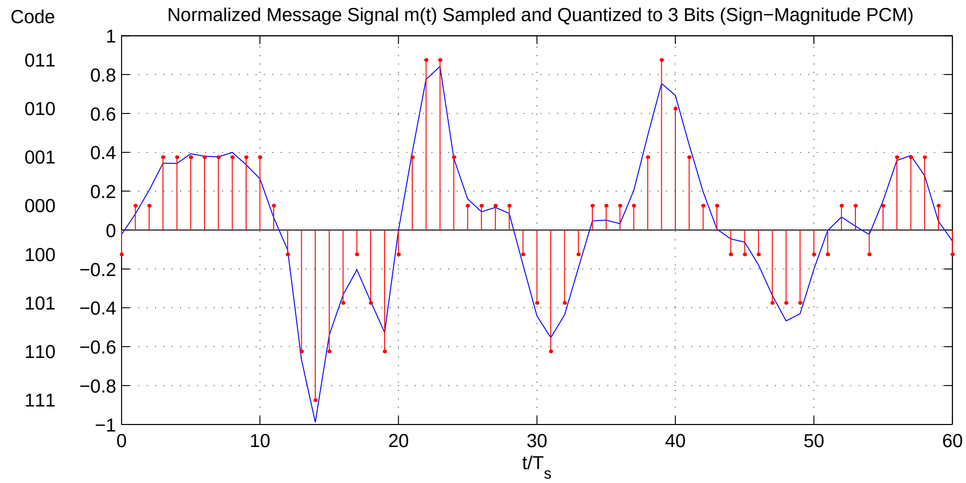


The 6 graphs below show how the quantized samples of the sine signal are transmitted serially using binary flat-top PAM. In this case the parallel to serial conversion is MSB first, i.e., the sign bit is transmitted first and the LSB of the magnitude representation is transmitted last.

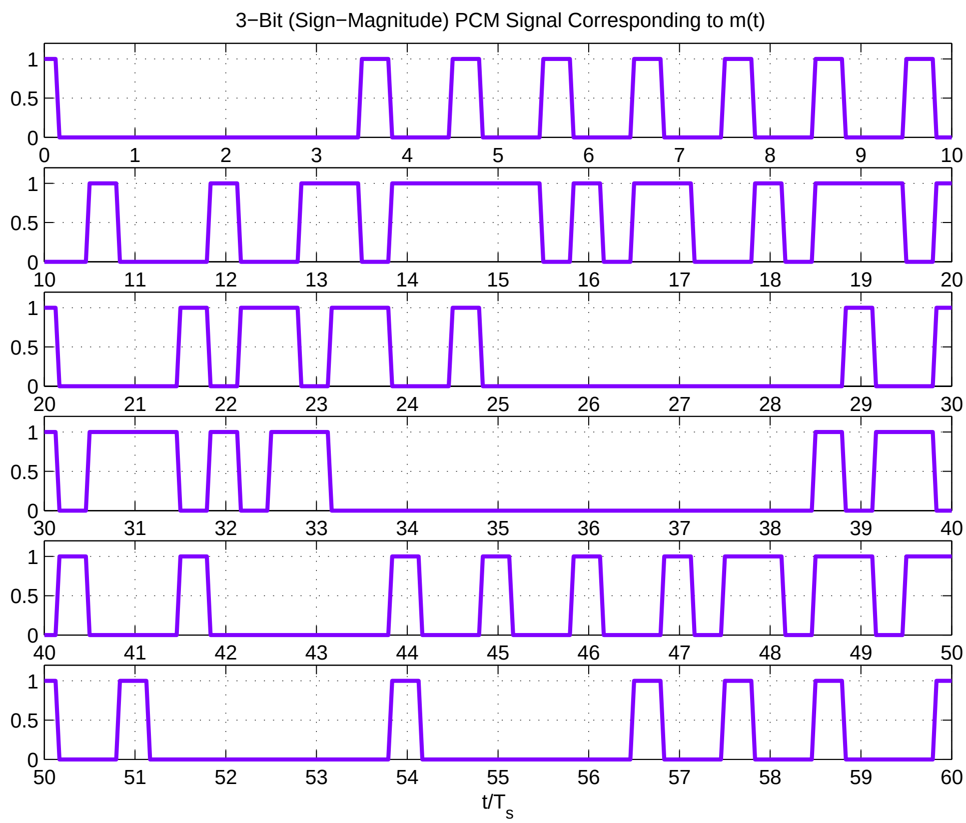


An interesting question is how to synchronize the receiver to the bit rate used at the transmitter when long strings of zeros or ones occur in the data. Several variations of the basic PCM scheme exist to deal with this problem, but their treatment is beyond the scope of this lab description.

Example: 3-Bit PCM for a Message Signal. The graph below shows a small segment of a typical (normalized) speech message signal $m(t)$ (blue line) and its digitized version (red stem plot) using 3-bit sign-magnitude format. It is quite visible that the 3-bit quantization is too crude. In fact, since low amplitudes are much more common in speech signals than high amplitudes, practical PCM schemes often use a non-linear quantization scheme.



The conversion of the digitized message signal to a binary PCM signal is shown below. Again, the MSB-first rule was used for the parallel to serial conversion.



The name “pulse code modulation” may seem a bit confusing, since PCM is really mostly a technique for digitizing analog signals and making them into binary data streams. It has to be understood in the context of other techniques that were developed at about the same time, for example pulse width modulation (PWM) and pulse position modulation (PPM).

1.7 Python and Jupyter Installation

The easiest way to get started with Python and Jupyter is to download and install Anaconda from https://www.continuum.io/downloads. Anaconda installs Python, the Jupyter Notebook, and commonly used packages for scientific computing, such as numpy, scipy, and matplotlib. There are two different versions of Anaconda, one for Python 2.x and one for Python 3.x. For this course Python 3.x will work fine, but it is also possible to install both, the Python 2.x and the Python 3.x versions, on the same computer.

If you prefer to install Python and Jupyter separately, you can go to https://www.python.org/downloads/ for Python and afterwards to http://jupyter.readthedocs.io/en/latest/install.html (scroll down to "Installing Jupyter with pip"). If you already have Python installed you can directly go to the step of installing Jupyter. To be able to use vectors and matrices efficiently, use signal processing functions such as filtering and fft, and make plots, you will also need numpy, scipy, and matplotlib, see https://www.scipy.org/install.html for instructions to install the SciPy stack.

1.8 Python Crash Course

Let's use Jupyter notebook to generate 1 second of a sinusoidal waveform with sampling frequency F_s. We start by importing the `numpy` and `matplotlib.pyplot` modules:

```
import numpy as np
import matplotlib.pyplot as plt
```

To be able to display graphs interactively and set the figure size we use:

```
%matplotlib notebook
fsz = (7,5)  # figure size
fsz2 = (fsz[0],fsz[1]/2.0)  # half high figure
```

Setting parameters for the sinusoidal waveform:

```
# initial parameters
Fs = 8000   # sampling rate
fm = 1000   # frequency of sinusoid
tlen = 1.0  # length in seconds
```

Next, we generate a time axis and a sinewave:

```
# generate time axis
tt = np.arange(np.round(tlen*Fs))/float(Fs)
# generate sine
xt = np.sin(2*np.pi*fm*tt)
```

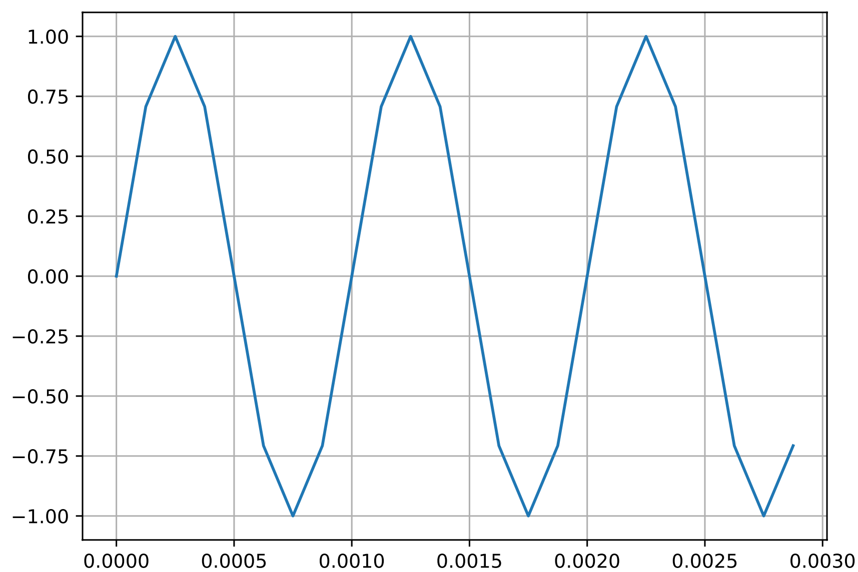
Printing the first few values of `xt` as a check:

```
# print the first 12 values of x(t)
print('xt = {}'.format(xt[:12]))

xt = [ 0.00000000e+00  7.07106781e-01  1.00000000e+00  7.07106781e-01
  1.22464680e-16 -7.07106781e-01 -1.00000000e+00 -7.07106781e-01
 -2.44929360e-16  7.07106781e-01  1.00000000e+00  7.07106781e-01]
```

Now we can plot the first few periods of `xt`:

```
plt.figure(1, figsize=fsz)
plt.plot(tt[:24], xt[:24])
plt.grid()
```

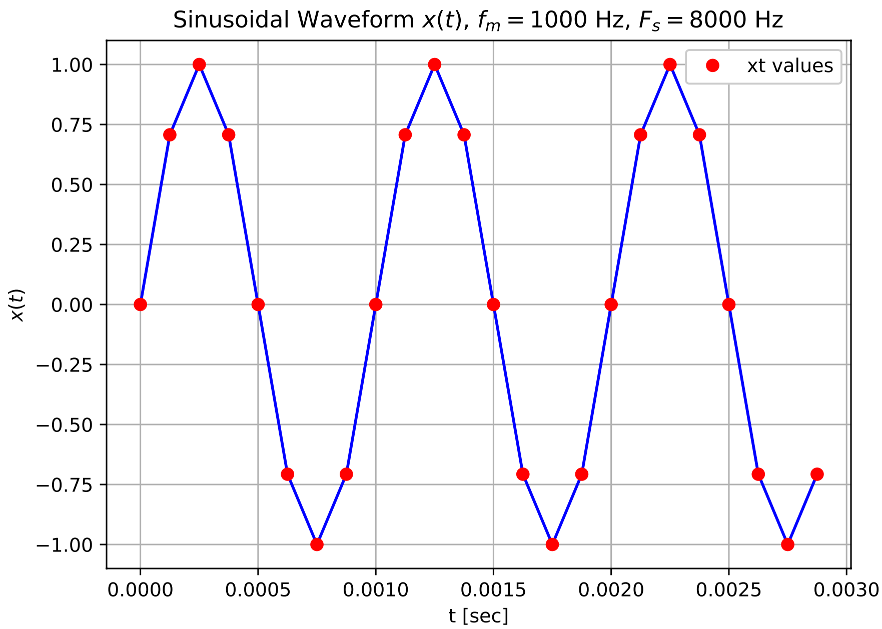
The resulting graph looks like this:



In general, graphs have to be labeled so that it can be understood what they are showing. Here is how to add an `xlabel`, a `ylabel`, a `title`, to specify colors, and to add a legend. Note that LaTeX statements can be included in strings enclosed by \$...\$ signs.

```
# create a labeled graph
plt.figure(2, figsize=fsz)
plt.plot(tt[:24], xt[:24], '-b')
plt.plot(tt[:24], xt[:24], 'or', label='xt values')
plt.ylabel('$x(t)$')
plt.xlabel('t [sec]')
strt2 = 'Sinusoidal Waveform $x(t)$'
strt2 = strt2 + ', $f_m={}$ Hz, $F_s={}$ Hz'.format(fm, Fs)
plt.title(strt2)
plt.legend()
plt.grid()
```

This results in the following graph:



In communications we often work with complex-valued signals. Let's use Euler's relation

$$e^{j\alpha} = \cos\alpha + j\,\sin\alpha\,,$$

to generate a complex exponential waveform in Python that contains both a cosine and a sine waveform. Here is a Jupyter notebook implementation:

Generation of Complex Exponential

Using Euler's relation

$$e^{j\alpha} = \cos\alpha + j\,\sin\alpha.$$

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

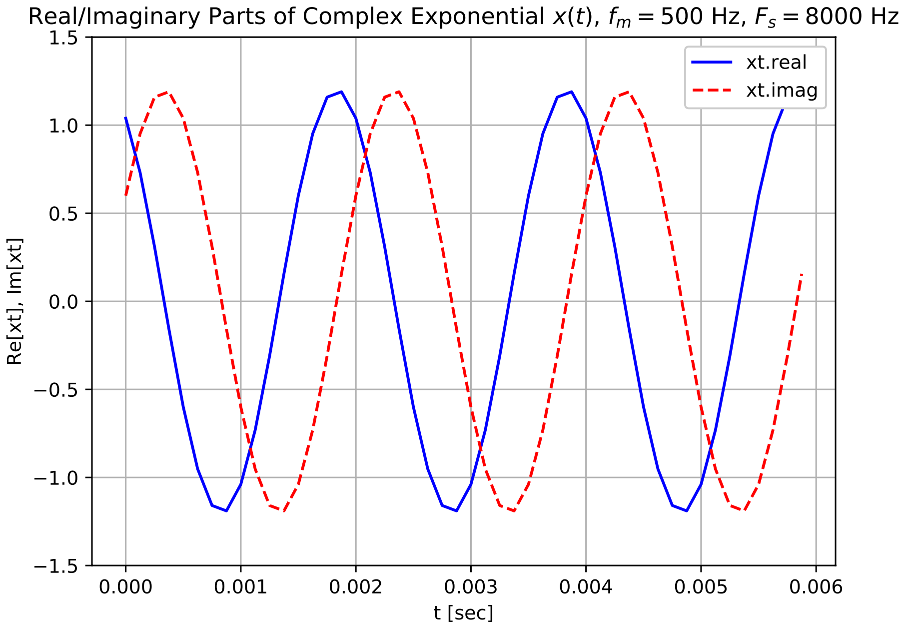
In [2]:
```
%matplotlib notebook
fsz = (7,5)   # figure size
fsz2 = (fsz[0],fsz[1]/2.0)   # half high figure
```

In [3]:
```
# parameters
Fs = 8000    # sampling rate
fm = 500     # frequency of sinusoid
phi = 30     # phase in degrees
A = 1.2      # amplitude
tlen = 1.0   # length in seconds
```

In [4]:
```
# time axis
tt = np.arange(np.round(tlen*Fs))/float(Fs)
# complex exponential
xt = A*np.exp(1j*(2*np.pi*fm*tt + np.pi/180*phi))
```

In [7]:
```
plt.figure(1, figsize=fsz)
plt.plot(tt[:48], xt[:48].real, '-b', label='xt.real')
plt.plot(tt[:48], xt[:48].imag, '--r', label='xt.imag')
plt.ylim([-1.5, 1.5])
plt.ylabel('Re[xt], Im[xt]')
plt.xlabel('t [sec]')
strt1 = 'Real/Imaginary Parts of Complex Exponential $x(t)$'
strt1 = strt1 + ', $f_m={}$ Hz, $F_s={}$ Hz'.format(fm, Fs)
plt.title(strt1)
plt.legend(loc=1)
plt.grid()
```

This produces the following graph:

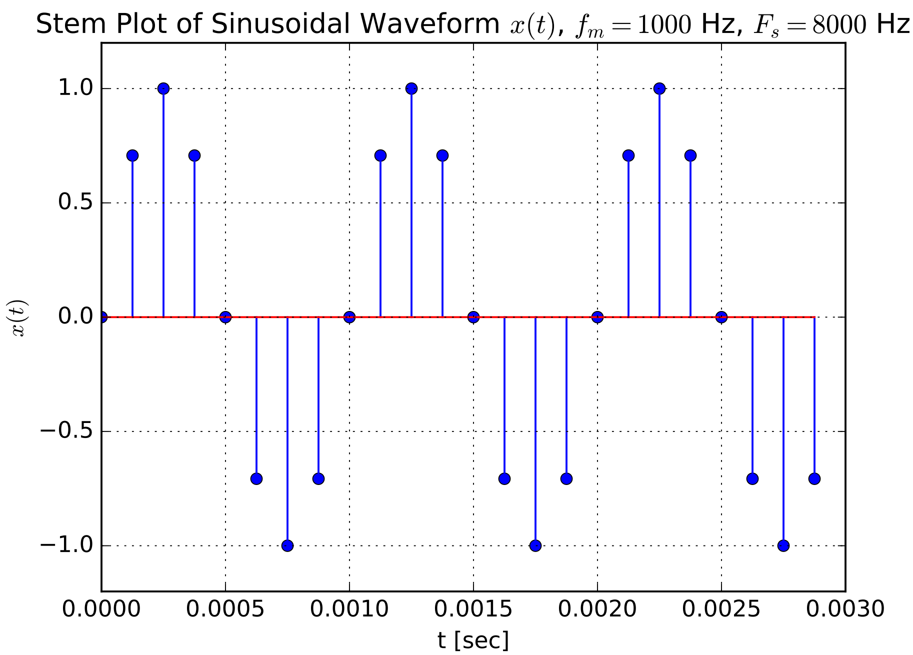


You may have noticed that the 1000 Hz sinusoidal waveform we generated earlier doesn't look like a "nice" sinusoid because there are only 8 samples per period. A problem that often occurs in digital signal processing is that the sampling rate of a signal needs to be either increased or decreased. To see how this can be done in Python, we will use the 1000 Hz sinusoidal signal generated earlier and increase its sampling frequency from $F_s = 8000$ Hz to $3\,F_s = F_{s3} = 24000$ Hz.

We start from making a stem plot of the original signal:

```
# make stem plot
plt.figure(1, figsize=fsz)
plt.stem(tt[:24], xt[:24])
plt.ylabel('$x(t)$')
plt.xlabel('t [sec]')
strt1 = 'Stem Plot of Sinusoidal Waveform $x(t)$'
strt1 = strt1 + ', $f_m={}$ Hz, $F_s={}$ Hz'.format(fm, Fs)
plt.title(strt1)
plt.grid()
```

The result looks like this:

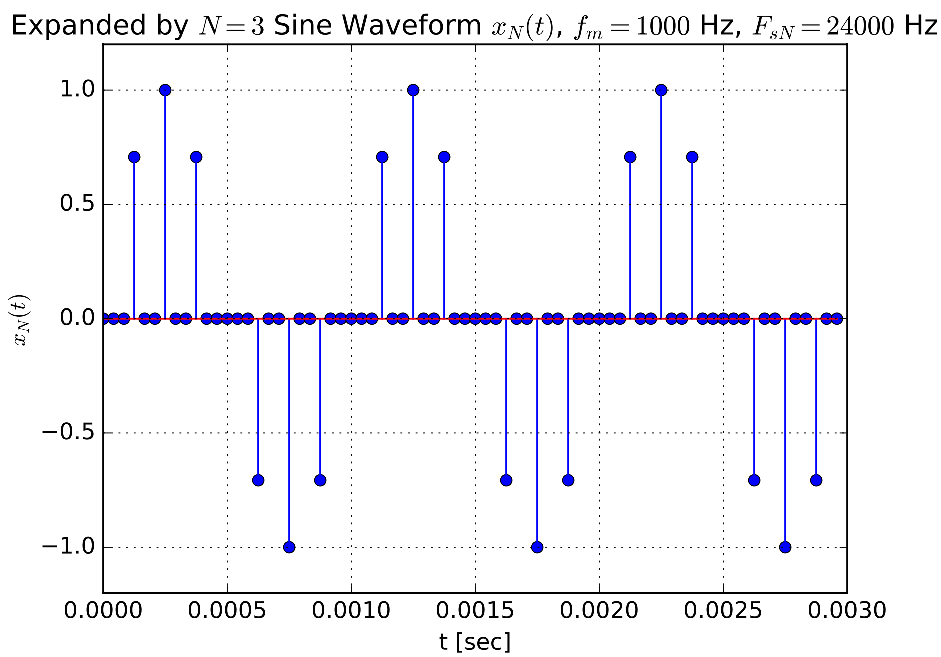


To expand the signal `xt` 3-fold, insert two zeros after each sample by first constructing a 2-dimensional array whose first row is `xt` and whose remaining two rows are all zeros. Then reshape this array into a 1-dimensional vector, reading it out columns first (`order='F'` where F stands for Fortran).

```
N = 3   # upsampling factor
xNt = np.vstack((xt, np.zeros((N-1, xt.size))))  # expand N times
xNt = np.reshape(xNt, -1, order='F')   # reshape into array
print(xNt[:24])   # check readout order

[  0.00000000e+00   0.00000000e+00   0.00000000e+00   7.07106781e-01
   0.00000000e+00   0.00000000e+00   1.00000000e+00   0.00000000e+00
   0.00000000e+00   7.07106781e-01   0.00000000e+00   0.00000000e+00
   1.22464680e-16   0.00000000e+00   0.00000000e+00  -7.07106781e-01
   0.00000000e+00   0.00000000e+00  -1.00000000e+00   0.00000000e+00
   0.00000000e+00  -7.07106781e-01   0.00000000e+00   0.00000000e+00]

FsN = N*Fs  # new sampling rate
ttN = np.arange(xNt.size)/float(FsN)  # new time axis
```

```
# new stem plot
plt.figure(2, figsize=fsz)
plt.stem(ttN[:N*24], xNt[:N*24])
plt.ylim([-1.2, 1.2])
plt.ylabel('$x_N(t)$')
plt.xlabel('t [sec]')
strt2 = 'Expanded by $N={}$ Sine Waveform $x_N(t)$'.format(N)
strt2 = strt2 + ', $f_m={}$ Hz, $F_{{sN}}={}$ Hz'.format(fm, FsN)
plt.title(strt2)
plt.grid()
```

Now the stem plot looks like this:



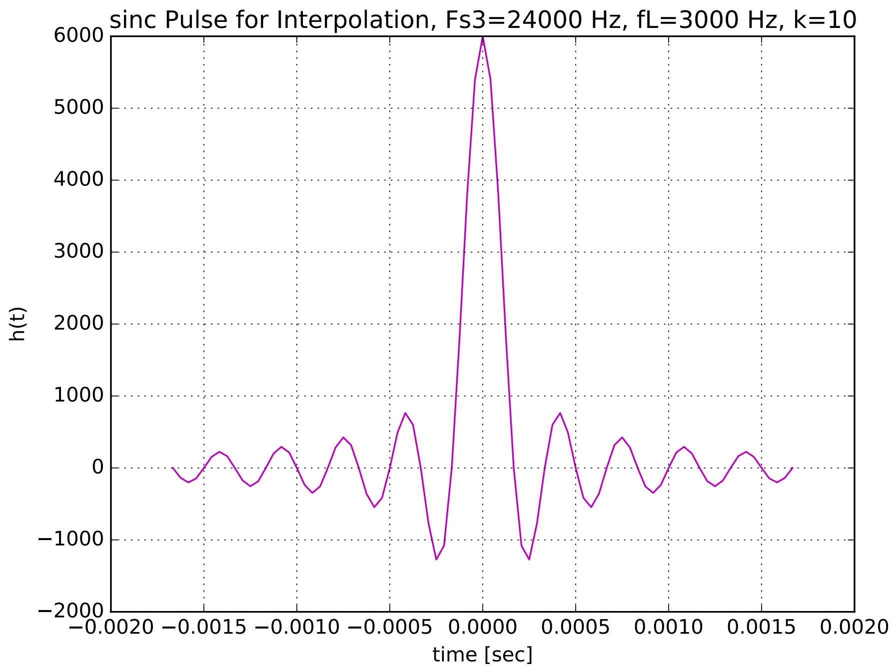
The next step is to interpolate between the nonzero samples in accordance with the assumption that the original signal is bandlimited to frequencies less than $F_s/2$. We will use a sinc pulse for the interpolation, corresponding to a lowpass filter with cutoff frequency f_L. The following function is used to generate the impulse response of the interpolation filter, truncated to $-k/(2f_L) \leq t \leq k/(2f_L)$.

```
def sinc_ipol(Fs, fL, k):
    """
    sinc interpolation function, cutoff frequency fL,
    taillength k/(2*fL) seconds
    >>>>> tth, ht = sinc_ipol(Fs, fL, k) <<<<<
    where Fs    sampling rate
          fL    cutoff frequency in Hz
          k     taillength in terms of zero crossings of sinc
          tth   time axis for h(t)
          ht    truncated sinc pulse h(t)
    """
    # create time axis
    ixk = int(np.round(Fs*k/float(2*fL)))
    tth = np.arange(-ixk, ixk+1)/float(Fs)
    # sinc pulse
    ht = 2*fL*np.sinc(2*fL*tth)
    return tth, ht
```

The code below generates the impulse response $h(t)$ of the interpolation filter for $f_L = 3000$ Hz and $k = 10$.

```
# plot of interpolation waveform
fL = 3000   # cutoff frequency
k = 10  # sinc pulse truncation
tth, ht = sinc_ipol(FsN, fL, k)
plt.figure(3, figsize=fsz)
plt.plot(tth, ht, '-m')
plt.ylabel('$h(t)$')
plt.xlabel('t [sec]')
strt3 = "'sinc' Pulse for Interpolation"
strt3 = strt3 + ', $F_s={}$ Hz, $f_L={}$ Hz, $k={}$'.format(FsN, fL, k)
plt.title(strt3)
plt.grid()
```

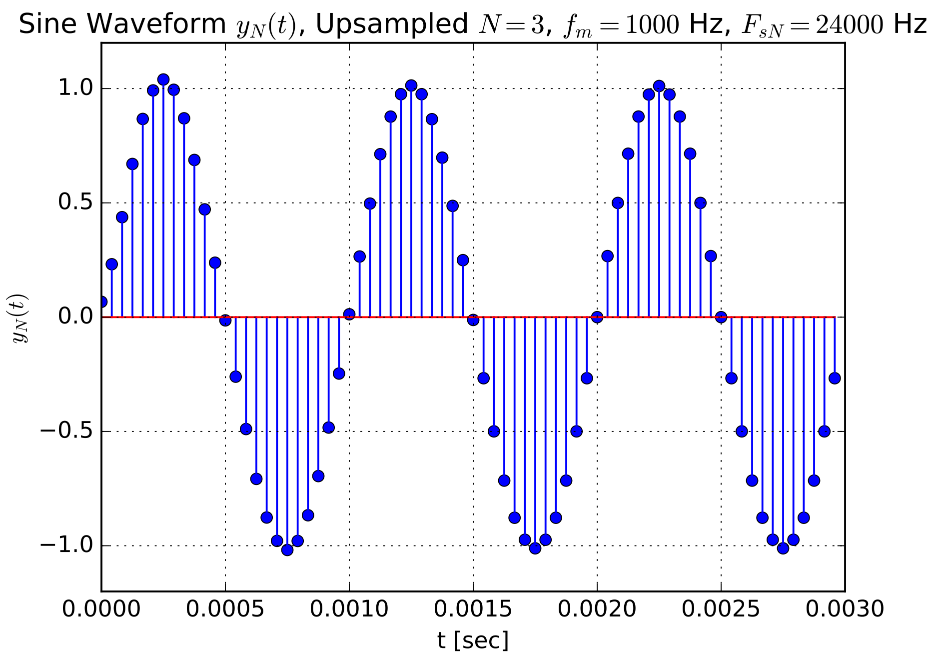
Here is what this sinc interpolation pulse looks like:



To obtain the interpolated sinewave it remains to convolve the sinc pulse with the expanded version `xNt` of the original sinewave.

```
# convolve expanded sine sequence with interpolation waveform to
# obtain upsampled (by factor N) sequence yNt with sampling rate FsN
yNt = np.convolve(xNt, ht, 'same')/float(Fs)

# stem plot of upsampled sequence
plt.figure(4, figsize=fsz)
plt.stem(ttN[:N*24], yNt[:N*24])
plt.ylim([-1.2, 1.2])
plt.ylabel('$y_N(t)$')
plt.xlabel('t [sec]')
strt4 = 'Sine Waveform $y_N(t)$, Upsampled $N={}$'.format(N)
strt4 = strt4 + ', $f_m={}$ Hz, $F_{{sN}}={}$ Hz'.format(fm, FsN)
plt.title(strt4)
plt.grid()
```

The result looks like this:



1.9 CT and DT Signals in Digital Signal Processing

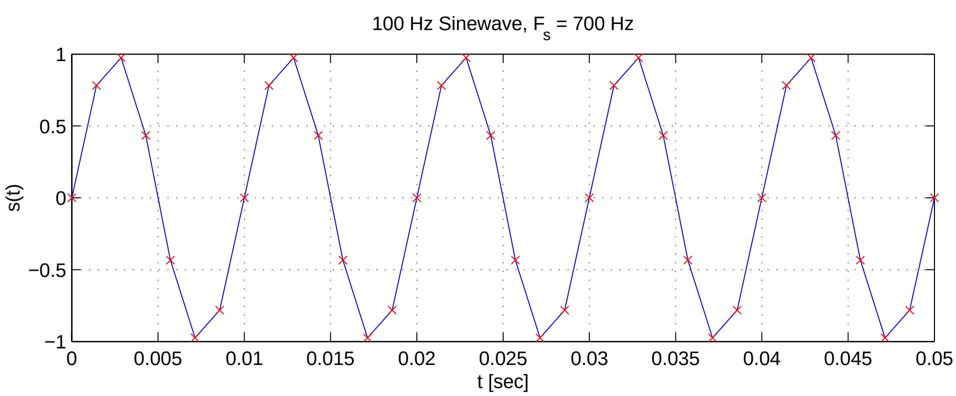
A continous time (CT) signal or waveform $s(t)$ is a signal whose amplitude is defined for any value of the time variable t. In a digital signal processing (DSP) environment, $s(t)$ has to be represented by a discrete time (DT) signal or sequence s_n that is related to $s(t)$ by

$$s_n = s(nT_s) = s(n/F_s)\ ,$$

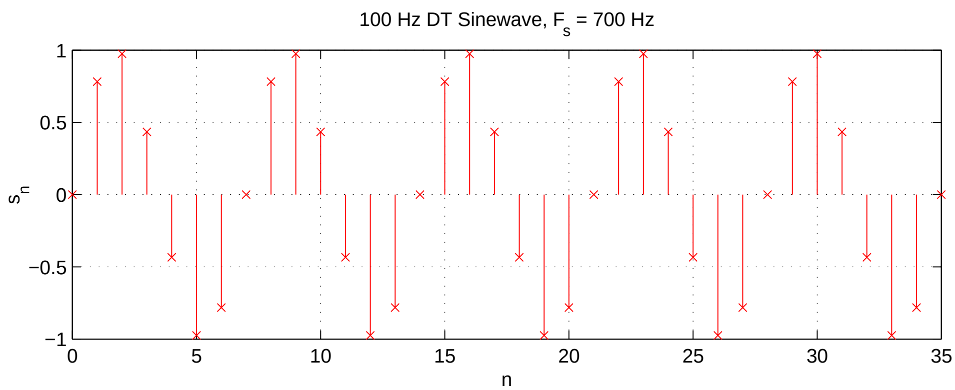
where $F_s = 1/T_s$ is the sampling rate in Hz. As an example, consider the sinusoid

$$s(t) = \sin 2\pi f_0 t\ .$$

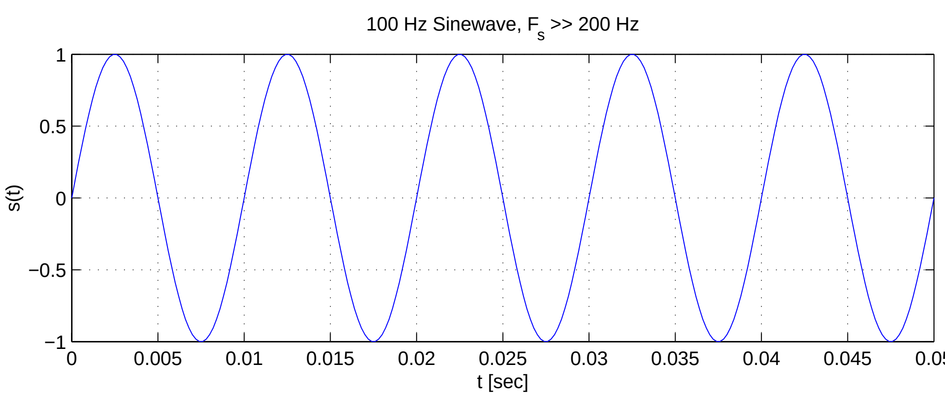
Setting $f_0 = 100$ Hz and $F_s = 700$ Hz and plotting s_n versus $t = n\,T_s$ yields the following graph.



Even though the sampling theorem is not violated ($F_s > 2f_0$), the plot (with straight lines between samples) is clearly not a very good approximation to $s(t)$. In such a case it is actually better to just simply plot the DT signal s_n versus n in the form of a stem plot as shown in the next figure.



However, if F_s is chosen much larger than $2f_0$, then a plot of s_n versus $t = nT_s$ is a very close approximation to plotting $s(t)$ versus t, as can be easily seen in the graph below (which uses $F_s = 5000$ Hz and does not explicitly show the sampling points).



In general, a signal vector `s` in a DSP environment with bandlimitation to f_B Hz will be considered a DT signal s_n if the sampling rate F_s (in Hz) is not much higher than the Nyquist rate $2f_B$, and it will be considered (a good approximation to) a CT signal $s(t)$ if $F_s \gg 2f_B$.

2 Lab Experiments

E1. Signals in Python. Assume that $s(t)$ is a bandlimited CT signal that is to be represented in Python by a DT sequence s_n with sampling rate $F_s = 1/T_s$. According to the sampling theorem, F_s must be at least twice the highest frequency in $s(t)$, but how should F_s be chosen so that a plot of s_n looks like $s(t)$? The goal of this experiment is to gain some intuitive experience for this question and at the same time practice some Python programming.

(a) Make a script file called `sine100.py`, using the following (numpy) Python commands:

```
# File: sine100.py
# Asks for sampling frequency Fs and then generates
# and plots 5 periods of a 100 Hz sinewave
import numpy as np
from ast import literal_eval

Fs = literal_eval(input('Enter sampling rate Fs in Hz: '))
f0 = 100           # Frequency of sine
tlen = 5e-2        # Signal duration in sec
tt = np.arange(0,np.round(tlen*Fs))/float(Fs)   # Time axis
st = np.sin(2*np.pi*f0*tt)   # Sinewave, frequency f0
```

This creates a discrete time axis `tt` of duration `tlen` with time instants spaced `1/Fs` seconds apart. The last statement generates the signal `st` which is a 100 Hz sine, sampled with rate Fs. Plot `st` versus `tt` for `Fs` equal to 200, 400, 800, 1600, 3200, etc. What is the smallest `Fs` that yields a "nice" and representative graph of a 100 Hz CT sinusoid? Look at the sample graphs shown in the introduction.

(b) After generating the sine in `st`, use the command

```
rt = np.sign(st)              # Sine -> rectangular
```

to generate a rectangular signal `rt` with frequency 100 Hz and sampling rate `Fs`. Note that `sign` stands for the **signum** function which is defined as

$$\mathrm{sgn}(x) = \begin{cases} +1\,, & x > 0\,, \\ 0\,, & x = 0\,, \\ -1\,, & x < 0\,. \end{cases}$$

Plot `rt` versus `tt` for `Fs` equal to 200, 400, 800, 1600, 3200, etc. What is the smallest `Fs` that yields a "nice" and representative graph of a 100 Hz rectangular CT signal?

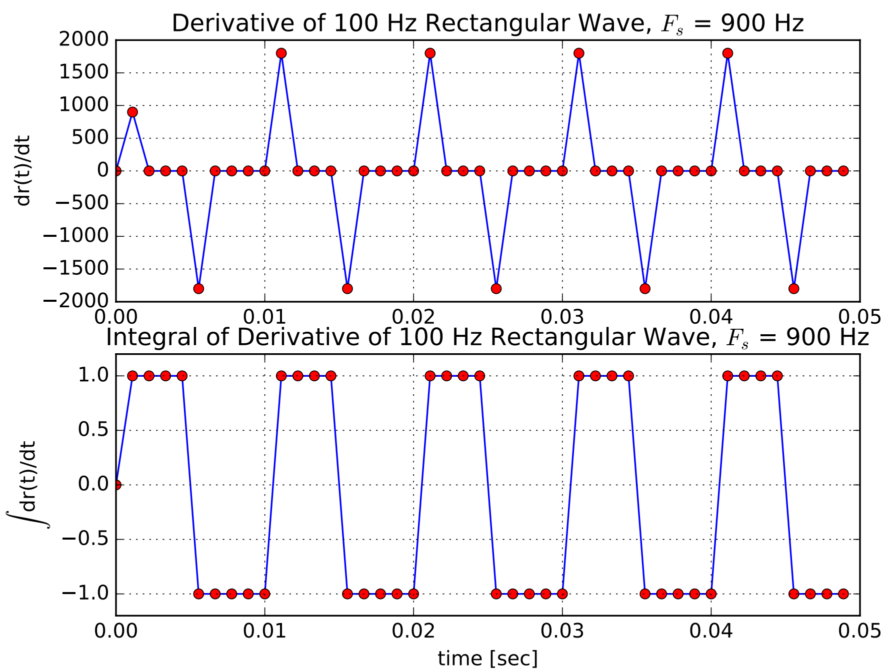
(c) DT approximations to CT signals in Python can be differentiated and integrated, which is often useful for the generation and anlysis of communication signals. Again, the goodness of the approximation of a CT signal by its sampled counterpart depends on the choice of the sampling frequency `Fs`. The following table shows the (numpy) Python commands for the DT approximations to differentiation and integration of a CT signal $x(t)$.

Derivative and Integral of Signals		
	CT Signal $x(t)$	**DT Signal** `x`
Derivative	$\dfrac{dx(t)}{dt}$	`np.diff(np.hstack((0,x)))*Fs`
Integral	$\int dx(t)\, dt$	`np.cumsum(x)/float(Fs)`

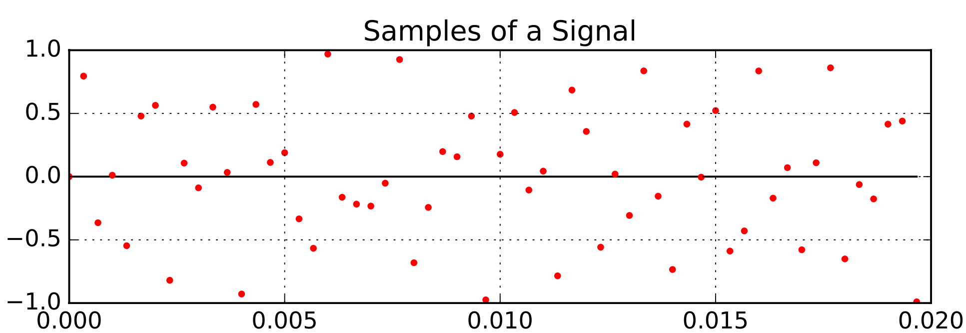
Note that since the sampling rate is `Fs`, "*dt*" is equal to `Ts=1/Fs`. Generate a rectangular 100 Hz signal in `rt` as in part (b) and then use the following commands to generate the "derivative" `rdt` of `rt` and the "integral" `rdit` of `rdt` which should be `rt` again.

```
rdt = np.diff(np.hstack((0,rt)))*Fs     # Derivative of rt
rdit = np.cumsum(rdt)/float(Fs)      # Integral of rdt
```

Make plots of `rdt` and `rdit` with `Fs` at the lower end of the useful range first so that you can see what `diff` and `cumsum` are doing. Then increase `Fs` until you obtain a plot that is a "nice" and representative approximation to differentiation and integration of CT signals. The following graphs show an example when `Fs=900` Hz.



(d) The graph below shows the samples of a signal called `sig01`. This signal is available in `sig01.wav`. Use the `wavread` function in the Appendix to read the .wav file.



Make a plot of the "CT version" of `sig01`, assuming that `sig01` is a baseband signal. Explain your approach and specify the parameters that you used.

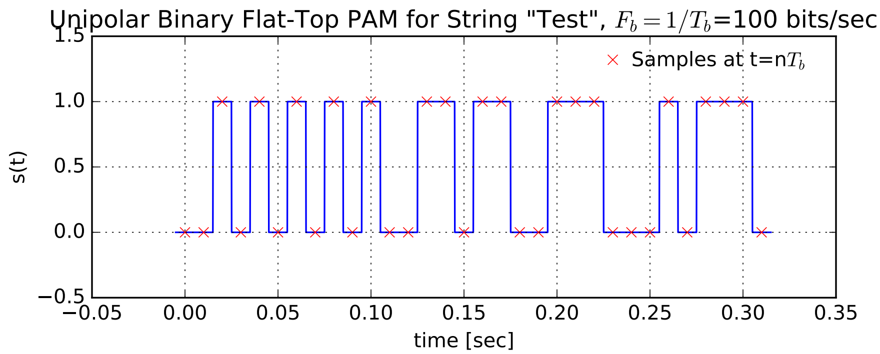
E2. Flat-Top PAM. (a) The goal of the first part of this experiment is to write a Python script file, e.g., called `ftpam01.py`, that accepts a text string as input and produces a flat-top PAM signal `st` as output. Here is an outline of `ftpam01.py`:

```
# File: ftpam01.py
# Script file that accepts an ASCII text string as input and
# produces a corresponding binary unipolar flat-top PAM signal
# s(t) with bit rate Fb and sampling rate Fs.
# The ASCII text string uses 8-bit encoding and LSB-first
# conversion to a bitstream dn. At every index value
# n=0,1,2,..., dn is either 0 or 1. To convert dn to a
# flat-top PAM CT signal approximation s(t), the formula
#   s(t) = dn,  (n-1/2)*Tb <= t < (n+1/2)*Tb,
# is used.
import numpy as np
import ascfun as af

Fs = 44100        # Sampling rate for s(t)
Fb = 100          # Bit rate for dn sequence
txt = 'Test'      # Input text string

bits = 8          # Number of bits
dn =  # >> Convert txt to bitstream dn here <<
N = len(dn)       # Total number of bits

Tb = 1/float(Fb)               # Time per bit
ixL = np.round(-0.5*Fs*Tb)     # Left index for time axis
ixR = np.round((N-0.5)*Fs*Tb)  # Right index for time axis
tt = np.arange(ixL,ixR)/float(Fs)  # Time axis for s(t)
st =  # >> Generate flat-top PAM signal s(t) here <<
```

Hints: The following Python module can be used to convert a text string in `txt` to ASCII using `bits` bits per character and then to a serial bitstream in `dn` (and vice versa).

```
# File: ascfun.py
# Functions for conversion between ASCII and bits
from numpy import np

def asc2bin(txt, bits=8):
    """
    ASCII message to serial binary conversion
    >>>>> dn = asc2bin(txt, bits) <<<<<
    where  txt        ASCII message (text string)
           abs(bits)  bits per character, default: 8
           bits > 0   LSB first parallel to serial conv
           bits < 0   MSB first parallel to serial conv
           dn         binary output DT sequence
    """

    txtnum = np.array([ord(c) for c in txt], np.int16)
    if bits > 0:
        # powers of 2: 2**0, 2**-1, 2**-2, ..., 2**-(bits-1)
        p2 = np.array(np.power(2.0, -np.arange(bits)), np.float32)
    else:
        # powers of 2: 2**(bits+1), ..., 2**-2, 2**-1, 2**0
        p2 = np.array(np.power(2.0, np.arange(bits+1,1)), np.float32)
    # 2-dim array of bits, one row per character in txt
    B = np.array(np.mod(np.floor(np.outer(txtnum,p2)),2),np.int8)
    # parallel to serial conversion
    return np.reshape(B, -1)

def bin2asc(dn, bits=8, flg=1):
    """
    Serial binary to ASCII text conversion
    >>>>> txt = bin2asc(dn, bits, flg) <<<<<
    where  dn         binary input sequence
           abs(bits)  bits per char, default=8
           bits > 0   LSB first parallel to serial
           bits < 0   MSB first parallel to serial
           flg != 0   limit range to [0...127]
           txt        output text string
    """

    # >>Function to be completed in part (b)<<
```

To generate `st` from `dn` you might consider using the differentiation/integration technique from E1(c) by differentiating `dn`, placing the resulting impulses spaced apart by `Tb` at $t = -T_b/2, T_b/2, 3T_b/2, 5T_b/2, \ldots$ in `st` and then integrating over `st` to obtain a rectangular flat-top PAM signal. The graph below shows `st` for the text string `Test`.



When you have completed the code in `ftpam01` and you have verified that it works correctly, generate a known test signal, e.g., with `txt='MyTest'` and save the resulting signal `st` in a `wav` file using the commands (see the wavfun module in the Appendix)

```
import wavfun as wf
wf.wavwrite('MyTest.wav',Fs,st/float(max(abs(st))))  # Write wav-file
```

so that you can use it in the second part of the experiment. Note that the amplitude A of signals in `wav`-files is restricted to $A \leq 1$ and for this reason `st/float(max(abs(st)))` is written with the `wavwrite` command rather than just `st`.

(b) The goal of this part of the experiment is to take a received flat-top PAM signal `rt` of the kind generated in (a) and to convert it back to a text string. It is assumed that the received signal `rt` is available in the form of a `wav` file. Here is the starting point for a flat-top PAM receiver script file called `ftpam_rcvr01`:

```
# File: ftpam_rcvr01.py
# Script file that accepts a binary unipolar flat-top PAM
# signal r(t) with bitrate Fb and sampling rate Fs as
# input and decodes it into a received text string.
# The PAM signal r(t) is received from a wav-file with
# sampling rate Fs. First r(t) is sampled at the right
# DT sequence sampling times, spaced Tb = 1/Fb apart. The
# result is then quantized to binary (0 or 1) to form the
# estimated received sequence dnhat which is subsequently
# converted to 8-bit ASCII text.
import numpy as np
import ascfun as af
import wavfun as wf

rt, Fs = wf.wavread('MyTest.wav')
Fb = 100                      # Data bit rate
Tb = 1/float(Fb)              # Time per bit
bits = 8                      # Number of bits/char
N = np.floor(len(rt)/float(Fs)/Tb)  # Number of received bits
dnhat =  # >>Sample and quantize the received PAM signal here<<

txthat = # >>Convert bitstream dnhat to received text string<<
print(txthat)                 # Print result
```

Note: Because $s(t)$ is obtained from d_n with bit rate $F_b = 1/T_b$ by setting

$$s(t) = d_n \,, \qquad (n - 1/2)\, T_b \le t < (n + 1/2)\, T_b \,,$$

the first sample of the received CT signal in the `wav` file is at time $t = -T_b/2$ and not at time $t = 0$. Also, the signal in a 16-bit `wav` file may be scaled because the amplitude A has to satisfy $A \le 1$.

Test your receiver with the `MyTest.wav` file that you generated in (a). When you have verified that your receiver works properly, use it to receive the signal in `ftpam_sig01.wav` and convert it back to a text string. The bit rate that was used for this signal is $F_b = 100$.

(c) Use your `ftpam_rcvr01` receiver to receive the signals in `ftpam_sig02.wav` and in `ftpam_sig03.wav` and convert them back to text strings. The bit rate of the second signal in `ftpam_sig02.wav` is unknown and you have to determine it from the signal itself. How difficult is it to find the bit rate and how crucial is it to have exactly the right value? The signal in `ftpam_sig03.wav` uses a bit rate of $F_b = 100$ and noise has been added to it. To record it as a `wav` file it had to be scaled to have a maximum amplitude of 1 (including noise), so it is crucial that you use the right threshold at the receiver.

E3. Pulse Code Modulation. (Experiment for ECEN 5002, optional for ECEN 4652) **(a)** Complete the following Python module with the `mt2pcm` and `pcm2mt` functions as described below:

```
# File: pcmfun.py
# Functions for conversion between m(t) and PCM representation
from numpy import np

def mt2pcm(mt, bits=8):
    """
    Message signal m(t) to binary PCM conversion
    >>>>> dn = mt2pcm(mt, bits) <<<<<
    where  mt     normalized (A=1) "analog" message signal
           bits   number of bits used per sample
           dn     binary output sequence in sign-magnitude
                  form, MSB (sign) first
    """
    # >>Your code goes here<<

def pcm2mt(dn, bits=8):
    """
    Binary PCM to message signal m(t) conversion
    >>>>> mt = pcm2mt(dn, bits) <<<<<
    where  dn     binary output sequence in sign-magnitude
                  form, MSB (sign) first
           bits   number of bits used per sample
           mt     normalized (A=1) "analog" message signal
    """
    # >>Your code goes here <<
```

The function `mt2pcm` is used to convert the (unquantized) samples of a message signal `mt` to a PCM bit string `dn` with `bits` bit quantization per sample. The function `pcm2mt` is used to convert the bit string `dn` back to "analog" samples. Test the two functions by using a sine signal for $m(t)$ in `mt2pcm`, then feeding the resulting `dn` into `pcm2mt` and finally plotting this result together with the original $m(t)$ in the same graph. Use a frequency of ≈ 100 Hz and a sampling rate in the range $4000 \ldots 8000$ Hz for the sine signal. Try 3-bit and 8-bit quantization.

(b) The signals in the files `pcm_sig01.wav` and `pcm_sig02.wav` are PCM signals with a bitrate of 64 kbit/sec, using 8-bit quantization, sign-magnitude format and MSB-first transmission. Flat-top PAM was used to convert the bit squences `dn` to a CT signal with amplitude zero for `dn=0` and positive amplitude for `dn=1`. The first signal is noiseless and the second signal has noise added to it. Note that the noisy signal had to be scaled to have an amplitude less than 1. Demodulate the two signals and convert them back to "analog" message signals. Write the m(t) signals to a wav-file, listen to them in a sound player and describe their content. Use the PCM test signals in `pcm_test01.wav` and `pcm_test02.wav` to test your PCM receiver. The first test signal is a 3-bit (sign-magnitude, MSB first) PCM signal with bit rate $F_b = 24000$ and the second test signal is a 8-bit (sign-magnitude, MSB first) PCM signal. The message signal $m(t)$ is in both cases a sine with frequency $f = 233.3$ Hz.

3 Appendix

Python module `wavfun` for reading and writing N-channel 16-bit PCM wav-files. Note that the amplitude A of signals recorded in wav-files is limited to amplitudes $A \leq 1$.

```
# File: wavfun.py
# Functions for reading and writing 16-bit PCM wav-files in Python.
import numpy as np
import scipy.io.wavfile as sio_wav

def wavread(fname):
    """
    Read N-channel 16-bit PCM wav-file
    >>>>> Fs, rt = wavread(fname) <<<<<
    where  fname   file name of wav-file
           Fs      sample rate of wav-file
           rt      data read from wav-file, N channels,
                   Nsamples data samples per channel,
                   (Nsamples x N) numpy array of type np.float32,
                   data samples normalized to range -1 ... +1
    """
    Fs, rt = sio_wav.read(fname)
    if rt.dtype is np.dtype(np.int16):
        rt = rt/float(2**15-1)  # convert to -1...+1 range
        rt = np.array(rt, np.float32)
    else:
        print('not a 16-bit PCM wav-file')
        rt = []
    return Fs, rt

def wavwrite(fname, Fs, xt):
    """
    Write N-channel 16-bit PCM wav-file
    >>>>> wavwrite(fname, Fs, xt) <<<<<
    where  fname   file name of wav-file
           Fs      sample rate of wav-file
           xt      data to be written to wav-file
                   (Nsamples x N) numpy array of floats
                   normalized to range -1...+1
                   N channels, Nsamples data samples per channel
    """
    # convert to np.int16 data type
    xt = np.array((2**15-1)*xt, np.int16)
    sio_wav.write(fname, Fs, xt)
```