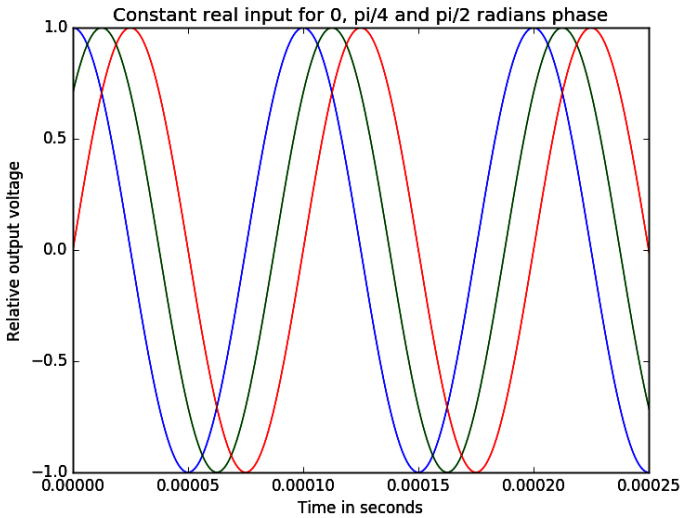
Simple Simulation of the Circuit for HF Modulation

This document describes a simulation of the basic operation of the phase shifting and upconverting circuit to make the transmitter low level signal. Filters are not included in this first version. The circuit consists first of a complex multiplier; its inputs are the baseband modulating signal in IQ, that is, complex, form and the angle by which to shift the phase, in the form of a complex number in rectangular coordinates, that is, the cos and sin of the angle to shift by. In the simulation, this multiplier is just represented by complex multiplication. The output of the multiplier is the input to the IQ (complex) input of a mixer. The lo of the mixer is a sinusoid at the center frequency of the transmitter. In the simulation, the operation of this mixer is given by the real part of the complex multiplication of the input signal with a complex sinusoid at the rf frequency. This real signal is the simulation of the signal that goes to the transmitter. (Basic description of the circuit is in *HFModulations.pdf* and *HFModulationsDiagram.pdf;* these have been distributed by email.) The purpose here is to check basic operation and show what simple baseband inputs do for us.

The simulation is written in Python; it consists of a single class with a few methods, and the listing is at the end of this document. This simulation is carried out by feeding simple input signals into the baseband input and checking the output to make sure that it is as expected. For ease of plotting test signals use an rf frequency of 10,000 Hz and a modulating frequency of 1 KHz.

The first figure shows the first few cycles of the rf output for a real constant baseband input, $1 + 0j$ for three different phase inputs. (This is a simple, useful input and output; **the two position switch shown on the block diagram referred to above should be changed to a three position switch: baseband inputs, rf input, and a constant input, allowing an easy way to switch to cw for testing or experiments.**)

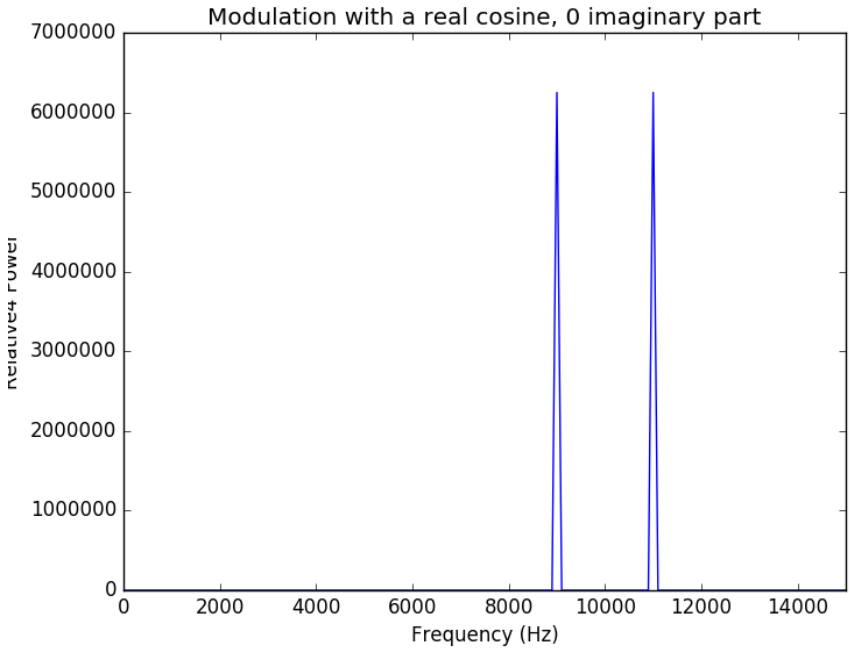


The rf output for zero phase (blue) was generated by these commands:

```
In [1]: import HFrf  # imports file HFrf

In [2]: t = HFrf.HFrf(1.0e-6, 10000, 1.e4, 1. + 0j)

In [3]: t.mtw(1000.)       # makes the test signals

In [4]: t.bb2rf(t.con)  # makes the rf signal

In [5]: plot(t.t, t.rf)
Out[5]: [<matplotlib.lines.Line2D at 0x113c6c710>]

In [6]: xlim(0.,.00025)
Out[6]: (0.0, 0.00025)
```

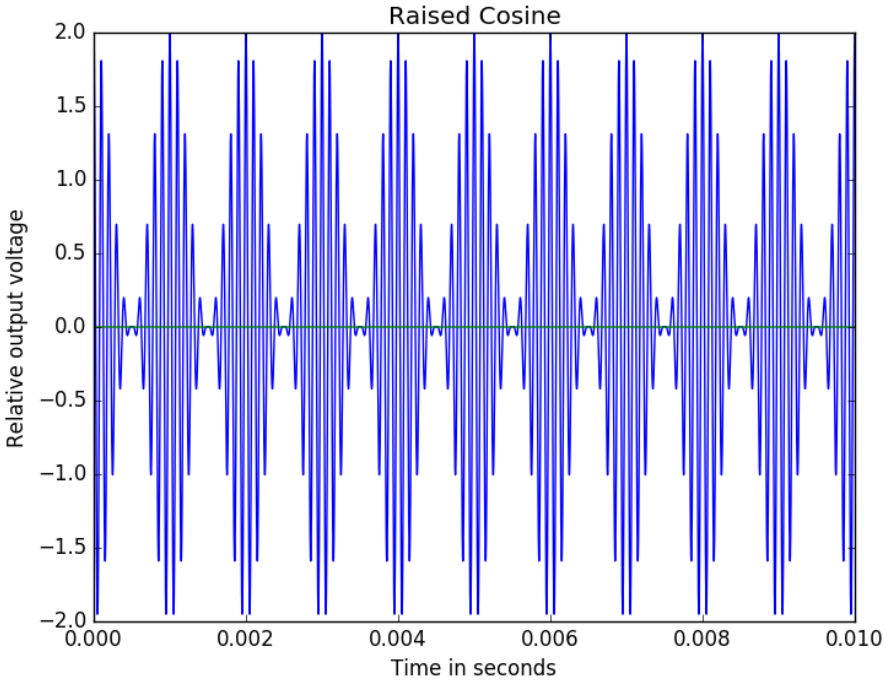
(Details of plot labeling, etc. not shown)
The other lines are made by changing `t.phShift` and rerunning `t.bb2rf`.

The next simplest baseband input is a real cosine ($\cos 2\pi f_m t + 0j$). This uses the test waveform `t.rs` and results in an output that has a symmetrical spectrum, as shown here:

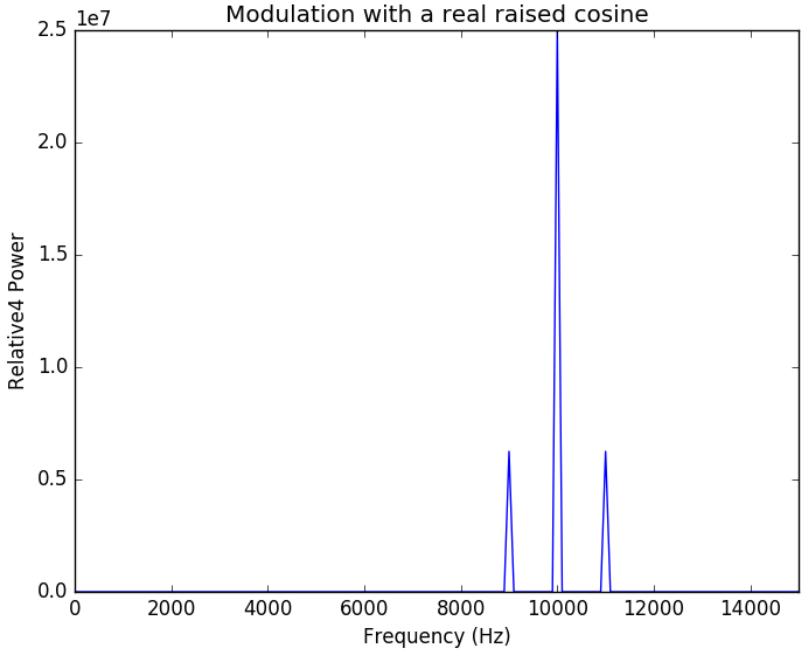


Thus we can generate two frequencies in the rf with a single baseband cosine if we want a symmetrical bandpass.

If we add dc to raise the cosine so that its minimum value is zero rather than -1 $(1 + \cos 2\pi f_m t + 0j)$, then we get this waveform in the time domain:

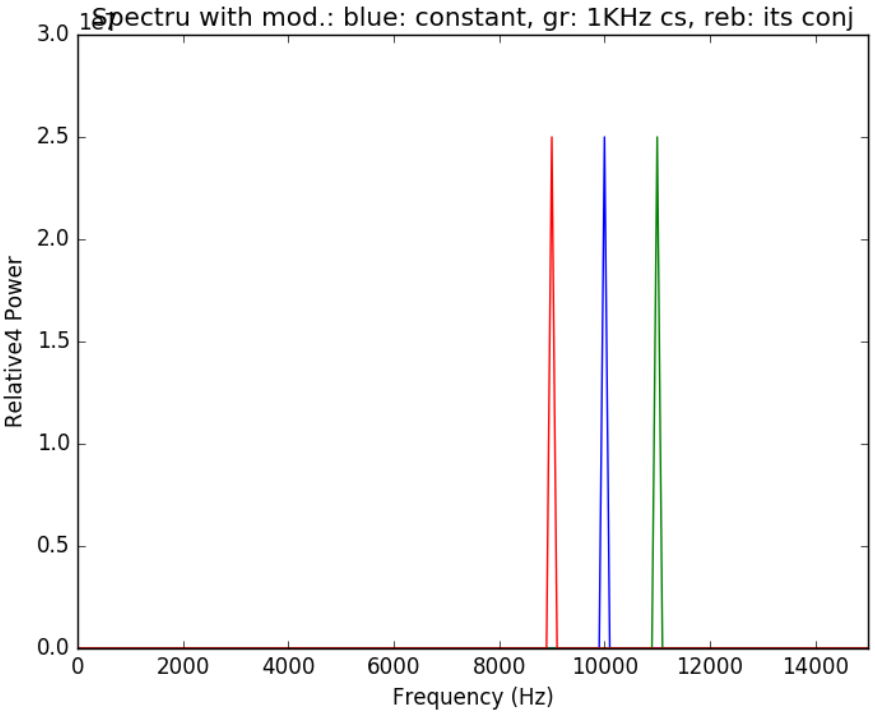


The spectrum is:



This is amplitude modulation.

Finally if we use a complex sinusoid ($\cos 2\pi f_m t + j \sin 2\pi f_m t$), we can frequency shift the rf. The center line in this plot results from `t.con`, the right hand line from `t.cs`, and the left hand line from `conj(t.cs)`.



If we want to generate two frequencies in such a way that the spectrum is not symmetric (for example the Carlson/Djuth two frequency experiment), it would be easier to generate them at rf and then bring them to baseband with the baseband mixer. We could generate two complex sinusoids at baseband, but it seems easier to generate two real sinusoids at rf.

Generating amplitude modulation with the method described here has another potential advantage. There is an issue regarding dc offsets when using analog multipliers as Dana has described. This is probably not a problem for the cos and sin phase inputs since since the sum of their squares is always one. However, the baseband input amplitude can be very small, for example during part of the amplitude modulation cycle. Thus an offset associated with this input path is a problem. We can adjust the level of the dc signal used to “raise” the cosine, at least somewhat improving the low level performance in real time.

The program listing is on the next page.

```
import numpy as np

# Class for simulating the HF baseband to rf with phase shift
# ts is the time between samples (seconds)
# nots is the number of samples in the baseband signal and rf
# frf is the frequency of the phShift  (Hz)
# phShift is the phase shift (a complex number with unity magnitude,
# cos, sin of the angle).
# self.phShift can be changed.
# The __init__ routine makes a time array (for convenient plotting).
# It also uses t to make a complex local oscillator for shifting to rf.
# The usual mixer using two real mixers is the real part of a complex
# multiplication.
#  bb2rf uses all complex operations and returns the real part.
class HFrf:
    def __init__(self, ts, npts, frf, phShift):
        self.ts = ts
        self.npts = npts
        self.frf = frf
        self.phShift = phShift
        self.t = np.linspace(0., npts*ts, num = npts, endpoint = False)
# Make the complex exponential for shifting the frequency
        self.lo = np.exp(1j*2.*np.pi*frf*self.t)

# Makes the rf from the complex baseband signal.  It uses all complex
# arithmetic and returns the real part.
# A positive phase shift moves the waveform later in time
# (Is this the correct convention to use?)
    def bb2rf(self, bb):
        self.rf = (bb*np.conj(self.phShift)*self.lo).real

# "power Spectrum" of self.rf with frequency scale
    def pspec(self):
        self.ps = np.absolute(np.fft.rfft(self.rf))**2
        self.f  = np.linspace(0., .5/self.ts, num = self.npts/2+1)

#  Makes various test baseband signals using the frequency input parameter
#  as needed.  Arrays are of length self.npts
#   1. con: 1. + 0j in all npts
#   2. cs:  a complex sinusoid at the given frequency
#   3. rs:  a complex array with a real cosine as the real part and
#           zeros for the imaginary part
#   4. rso: like rs, but offset, that is, one added to the real part
#   5. rsi: raised sin in the imaginary, 0 in real
    def mtw(self, ft):
        self.fr  = ft
        self.con = np.ones(self.npts) + 1j*np.zeros(self.npts)
        self.cs  = np.exp(1j*2.*np.pi*ft*self.t)
        self.rs  = self.cs.real + 1j*np.zeros(self.npts)
        self.rso = self.rs + 1.
        self.rsi = np.zeros(self.npts) + 1j*(1 + self.cs.imag)
```