SRN: PES1UG22EC321

NAME:TUSHAR PRABHU

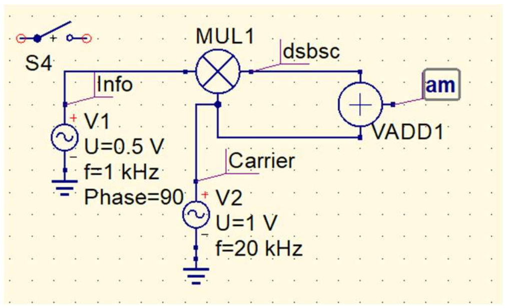
AHP Component - 1

```
import heapq
from collections import defaultdict
import math
class Node:
    def __init__(self, symbol=None, probability=None):
        self.symbol = symbol
        self.probability = probability
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.probability < other.probability

def generate_huffman_tree(probabilities):
    heap = [Node(symbol=symbol, probability=prob) for symbol, prob in probabilities.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged_node = Node(probability=left.probability + right.probability)
        merged_node.left = left
        merged_node.right = right
        heapq.heappush(heap, merged_node)

    return heap[0]

def generate_huffman_codes(root, current_code="", codes=None):
    if codes is None:
        codes = {}

    if root is not None:
        if root.symbol is not None:
            codes[root.symbol] = current_code
        generate_huffman_codes(root.left, current_code + "1", codes)
        generate_huffman_codes(root.right, current_code + "0", codes)

    return codes

def calculate_entropy(probabilities):
    entropy = -sum(p * math.log2(p) for p in probabilities.values() if p > 0)
    return entropy

def calculate_average_codeword_length(huffman_codes, probabilities):
    average_length = sum(len(huffman_codes[symbol]) * probabilities[symbol] for symbol in huffman_codes)
    return average_length

def calculate_efficiency(entropy, average_codeword_length):
    efficiency = entropy / average_codeword_length
    return efficiency
```

```
def main():
    # Replace this dictionary with your own probabilities
    a=eval(input("Enter the symbols as a list ,For Example: ['s0','s1','s2','s3','s4'] --->"))
    b=eval(input("Enter the probabilities as a list, For Example: [0.4,0.2,0.2,0.1,0.1] --->"))
    if(len(a)!=len(b)):
        print("Error")
        exit()
    else:
        probabilities = dict(zip(a, b))

    huffman_tree = generate_huffman_tree(probabilities)
    huffman_codes = generate_huffman_codes(huffman_tree)

    print("Huffman Codes:")
    for symbol, code in huffman_codes.items():
        print(f"{symbol}: {code}")
    entropy = calculate_entropy(probabilities)
    print(f"\nEntropy: {entropy:.4f} bits")

    average_codeword_length = calculate_average_codeword_length(huffman_codes, probabilities)
    print(f"Average Codeword Length: {average_codeword_length:.4f} bits")

    efficiency = calculate_efficiency(entropy, average_codeword_length)
    print(f"Efficiency: {efficiency:.4%}")

if __name__ == "__main__":
    main()
```

AHP Component – 2

A. For the Product Modulator the following parameters :



fc – Carrier Signal Frequency

fm – Message Signal Frequency

Ac – Carrier Signal Amplitude

Am – Message Signal Amplitude

ka – Amplitude Sensitivity

On changing the above parameters we observe the following the changes:

1. fc:

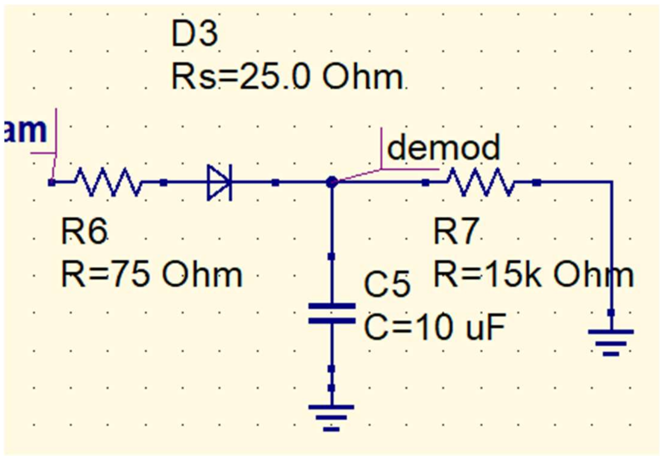
Effect on the Spectrum: As the carrier frequency the bandwidth of the modulated signal increases in the spectrum.
Effect on the Graphs: In the time domain , higher the carrier frequency faster is the rate of frequency variations according to the modulated signal.

2. fm:
Effect on Spectrum: As the fm increases the bandwidth of the modulated signal increases.
Effect on Graph: The waveform exhibits more oscillations in a given interval of time.

3. Ac:
Effect on Spectrum: As the Ac increases, this results in a more powerful or strongly modulated signal.
Effect on Graph: A larger Ac generally results in a higher amplitude of modulated signal.

4. Am:
Effect on Spectrum: As the Ac increases, this results in a more powerful or strongly modulated signal.
Effect on Graph: A larger Ac generally results in a higher amplitude of modulated signal. When the Am is too high , it can lead to overmodulation and causes distortions in the graphs.

B. Envelope Detector:



The output obtained from the envelope detector must be similar to the input signal since it is demodulating the initially modulated signal. However there certain parameters that affect the speed of the response.

1. Rs (Series Resistor):
A larger resistance value provides a smoother output similar to the initial signal, however the time required to produce the response increases. Hence a larger the resistance guarantees a more efficient output but increases the response time.

2)C (Capacitor):

A larger capacitor value provides a more smooth output similar to the initial signal, however the time required to produce the response increases. Hence a larger the capacitance guarantees a more efficient output but increases the response time.

Conclusion: Hence there is a **Trade-Off** between smoothness of the signal and the response time of the demodulated signal.

AHP Component – 3

Python Code:

```
# PES1UG22EC321
from scipy import signal
from scipy.signal import butter, lfilter
import numpy as np
import matplotlib.pyplot as plt

N = 1e5  # Number of samples for visualization as a waveform
Ac = 1  # Carrier peak amplitude
Am = 0.5  # Message signal peak amplitude
freq_c = 10e3  # carrier frequency
freq_m = 500  # message frequency
fs = 2.5 * (freq_c + freq_m)  # sampling frequency is at least twice the highest
frequency

def butter_lowpass(cutoff, fs, order=4):
    nyq = 0.5 * fs
    normal_cutoff = cutoff / nyq  # the critical frequencies must be normalized
    b, a = butter(order, normal_cutoff, btype='lowpass', analog=False)
    return b, a

def butter_lowpass_filter(data, cutoff, fs, order=4):
    b, a = butter_lowpass(cutoff, fs, order=order)  # a and b coefficients of the
LPF modeled as FIR/IIR filter
    y = lfilter(b, a, data)  # filter output
    return y

order = 4
cutoff = freq_m

time = np.arange(N) / fs  # time instants of the samples of duration (1/fs) seconds
c = Ac * np.cos(2 * np.pi * freq_c * time)  # Carrier signal
m = Am * np.cos(2 * np.pi * freq_m * time)  # Message signal
s = m * c  # DSB-SC signal
y = s * c  # product modulator output at receiver

x = butter_lowpass_filter(y, cutoff, fs, order)  # demodulated signal

# Coherent detector
theta = 2 * np.pi * freq_c * time
coherent_detector_output = 2 * x * np.cos(theta)

# Plotting
f, Pxx_spec = signal.welch(s, fs, 'flattop', 1024, scaling='spectrum',
return_onesided=True)
plt.figure()
plt.plot(f, np.sqrt(Pxx_spec))  # magnitude spectrum of DSBSC wave
plt.xlabel('Frequency [Hz]')
plt.ylabel('Magnitude')
plt.title('Power Spectrum of DSB-SC Wave')
plt.grid()

f_modulator, Pxx_modulator = signal.welch(y, fs, 'flattop', 1024,
scaling='spectrum', return_onesided=True)
plt.figure()
plt.plot(f_modulator, np.sqrt(Pxx_modulator))  # magnitude spectrum of product
modulator output
plt.xlabel('Frequency [Hz]')
plt.ylabel('Magnitude')
plt.title('Power Spectrum of Product Modulator Output')
plt.grid()

f_lowpass, Pxx_lowpass = signal.welch(x, fs, 'flattop', 1024, scaling='spectrum',
return_onesided=True)
plt.figure()
plt.plot(f_lowpass, np.sqrt(Pxx_lowpass))  # magnitude spectrum of low-pass filter
output (demodulated signal)
plt.xlabel('Frequency [Hz]')
plt.ylabel('Magnitude')
plt.title('Power Spectrum of Demodulated Signal')
plt.grid()

plt.figure()
plt.plot(time[0:200], x[0:200])  # demodulated signal waveform
plt.xlabel('Time [seconds]')
plt.ylabel('Voltage [volts]')
plt.title('Demodulated Signal')
plt.grid()

plt.figure()
plt.plot(time[0:200], coherent_detector_output[0:200])  # Coherent detector output
waveform
plt.xlabel('Time [seconds]')
plt.ylabel('Voltage [volts]')
plt.title('Coherent Detector Output')
plt.grid()

plt.show()
```

Observations:

In the python code we have the following parameters :

1. N : Number of samples in the waveform
2. Ac : Carrier Signal Peak Amplitude
3. Am : Message Signal Peak Amplitude
4. freq_c : Carrier Signal Frequency
5. freq_m : Message Signal Frequency
6. Fs : Sampling Frequency (Nyquist Rate)

By varying the above parameters in the power spectrum graph of the modulated and demodulated graph we observe that :

1. Number of Samples (N):

- Increasing N will improve frequency resolution in the power spectrum graphs, allowing for a clearer representation of signal components.

- Decreasing N may lead to spectral leakage and less accurate frequency information.

2. Carrier Peak Amplitude (Ac):

   - Increasing Ac will increase the amplitude of the carrier signal, affecting the amplitude of the sidebands in the power spectrum of the modulated signal.

3. Message Signal Peak Amplitude (Am):

   - Increasing Am will increase the modulation depth and potentially result in a broader frequency spectrum with more pronounced sidebands.

4. Carrier Frequency (freq_c):

   - Increasing freq_c will shift the entire power spectrum to higher frequencies.

5. Message Frequency (freq_m):

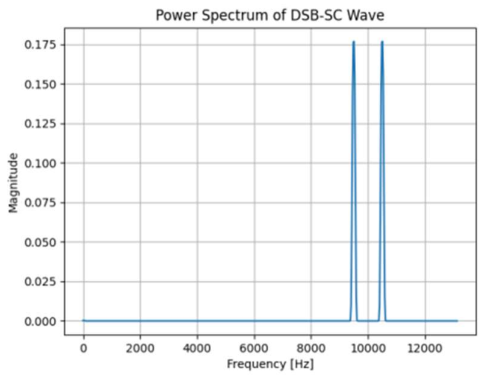
   - Increasing freq_m will shift the spectrum to higher frequencies and may increase the spacing between sidebands in the modulated signal's power spectrum.

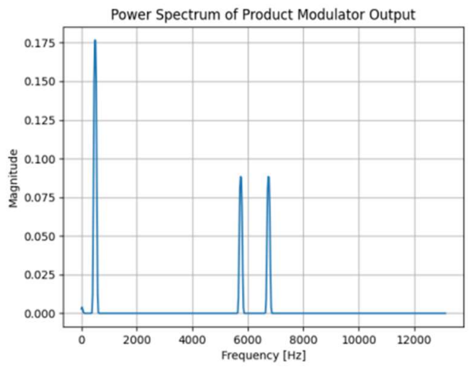
6. Sampling Frequency (fs):

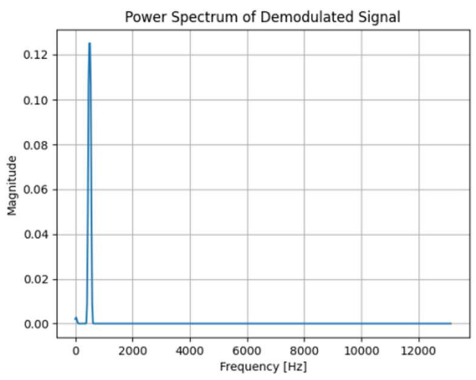
   - Increasing fs will improve frequency resolution and may provide a more accurate representation of the power spectrum.

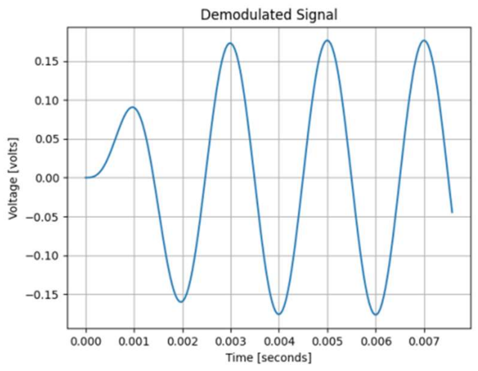
   - Decreasing fs below the Nyquist frequency could lead to aliasing, distorting the power spectrum.

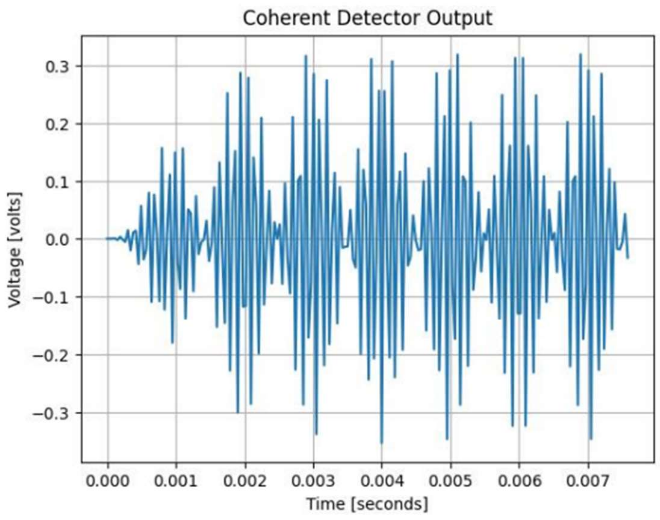
<u>Graphs:</u>











AHP Component – 4

PART-A

Python Code:

a)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
# PES1UG22EC321
def continuous_time_signal(t):
    # Example continuous-time sinc signal
#PES1UG22EC302 Sushant R Naik
    return np.sinc(0.25 * (t - 5))**2

def ideal_sampling(signal, fs, Ts):
    # Ideal sampling of continuous-time signal
    sampled_indices = np.arange(0, len(signal), int(fs * Ts))
    sampled_signal = signal[sampled_indices]
    sampled_time = np.arange(0, len(signal), int(fs * Ts)) / fs
    return sampled_time, sampled_signal
```

```
def ideal_reconstruction(sampled_time, sampled_signal, fs):
    # Ideal reconstruction using a low-pass filter
    reconstructed_signal = signal.lfilter([1], [1, 0], sampled_signal)
    reconstructed_time = np.linspace(0, sampled_time[-1],
len(reconstructed_signal))
    return reconstructed_time, reconstructed_signal

# Parameters
fs_original = 1000  # Original continuous-time signal frequency
Ts_original = 1/fs_original  # Original continuous-time signal sampling interval

# Generate continuous-time sinc signal
t_continuous = np.linspace(0, 10, 1000)
signal_continuous = continuous_time_signal(t_continuous)

# Ideal sampling
fs_sampled = 100 * fs_original  # Choose sampling frequency (e.g., 100 times
higher)
Ts_sampled = 1/fs_sampled
sampled_time, sampled_signal = ideal_sampling(signal_continuous, fs_sampled,
Ts_sampled)

# Ideal reconstruction
reconstructed_time, reconstructed_signal = ideal_reconstruction(sampled_time,
sampled_signal, fs_sampled)

# Plotting
plt.figure(figsize=(12, 8))

plt.subplot(3, 1, 1)
plt.plot(t_continuous, signal_continuous, label='Continuous-Time Signal')
plt.title('Continuous-Time Signal')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.legend()

plt.subplot(3, 1, 2)
plt.stem(sampled_time, sampled_signal, markerfmt='ro', basefmt='r', label='Sampled
Signal')
plt.title('Sampled Signal')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(reconstructed_time, reconstructed_signal, label='Reconstructed Signal')
plt.title('Reconstructed Signal')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.legend()

plt.tight_layout()
plt.show()
```

b)

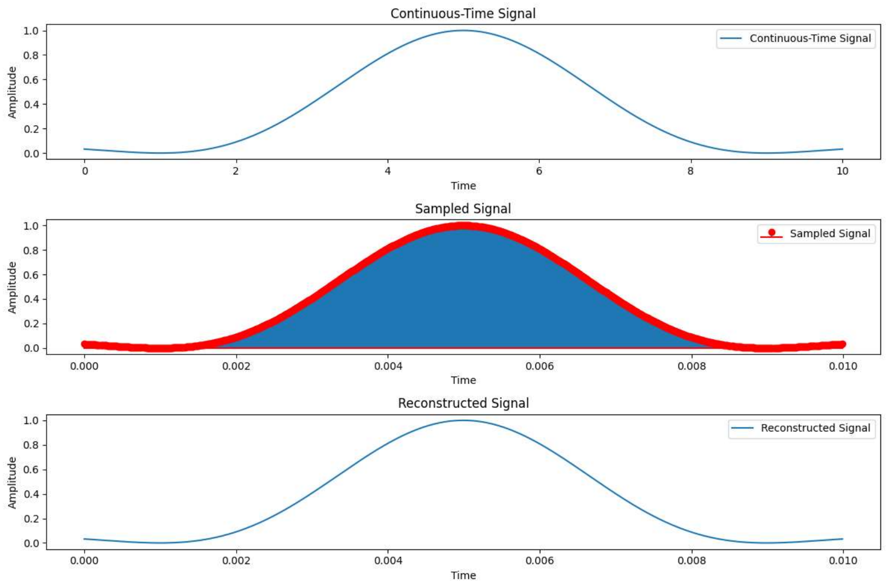
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
# PES1UG22EC321
def continuous_time_signal(t):
```

```
#PES1UG22EC302 Sushant R Naik
    return 2 * np.cos(6 * np.pi * t + 0.1) + 2 * np.cos(10 * np.pi * t + 0.2)

def ideal_sampling(signal, fs, Ts):
    # Ideal sampling of continuous-time signal
    sampled_indices = np.arange(0, len(signal), int(fs * Ts))
    sampled_signal = signal[sampled_indices]
    sampled_time = np.arange(0, len(signal), int(fs * Ts)) / fs
    return sampled_time, sampled_signal

def ideal_reconstruction(sampled_time, sampled_signal, fs):
    # Ideal reconstruction using a low-pass filter
    reconstructed_signal = signal.lfilter([1], [1, 0], sampled_signal)
    reconstructed_time = np.linspace(0, sampled_time[-1],
len(reconstructed_signal))
    return reconstructed_time, reconstructed_signal

# Parameters
fs_original = 1000  # Original continuous-time signal frequency
Ts_original = 1/fs_original  # Original continuous-time signal sampling interval

# Generate continuous-time sinc signal
t_continuous = np.linspace(0, 1, 1000)
signal_continuous = continuous_time_signal(t_continuous)

# Ideal sampling
fs_sampled = 100 * fs_original  # Choose sampling frequency (e.g., 100 times
higher)
Ts_sampled = 1/fs_sampled
sampled_time, sampled_signal = ideal_sampling(signal_continuous, fs_sampled,
Ts_sampled)

# Ideal reconstruction
reconstructed_time, reconstructed_signal = ideal_reconstruction(sampled_time,
sampled_signal, fs_sampled)

# Plotting
plt.figure(figsize=(12, 8))

plt.subplot(3, 1, 1)
plt.plot(t_continuous, signal_continuous, label='Continuous-Time Signal')
plt.title('Continuous-Time Signal')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.legend()

plt.subplot(3, 1, 2)
plt.stem(sampled_time, sampled_signal, markerfmt='ro', basefmt='r', label='Sampled
Signal')
plt.title('Sampled Signal')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(reconstructed_time, reconstructed_signal, label='Reconstructed Signal')
plt.title('Reconstructed Signal')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.legend()

plt.tight_layout()
plt.show()
```

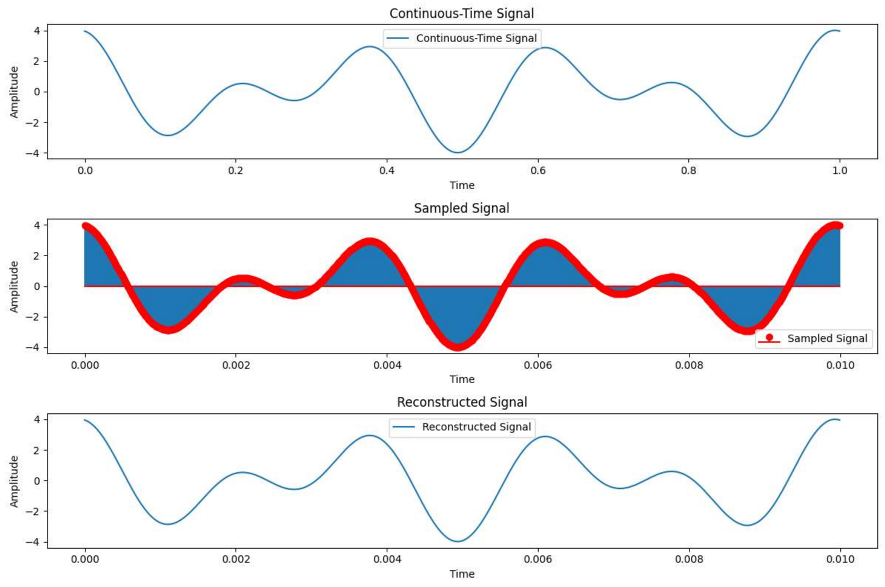
Graphs For Ideal Sampling and Reconstruction of the Input Continuous Time Signal

# PES1UG22EC321

A. sinc(0.25*(t - 5))2**



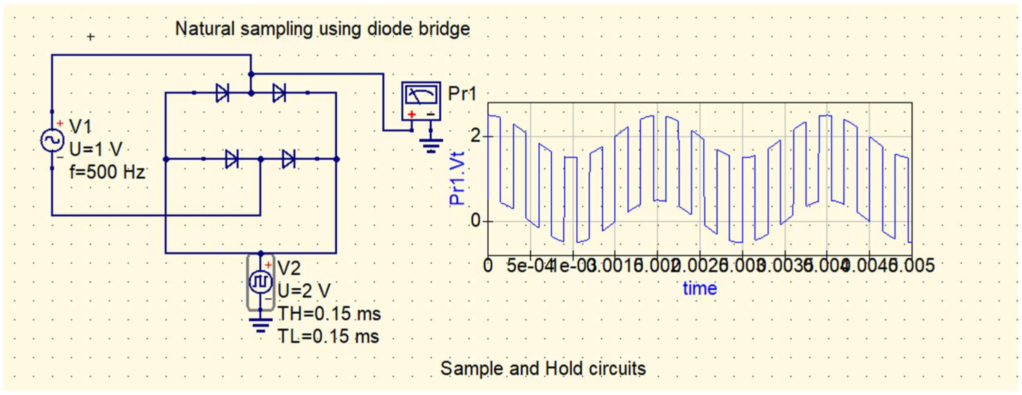
B. cos(6 * pi * t + 0.1) + 2 * cos(10 * pi * t + 0.2)



PART-B

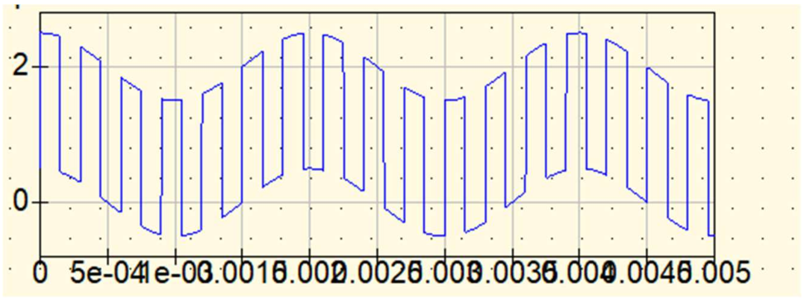
1) ***# PES1UG22EC321***

(a)

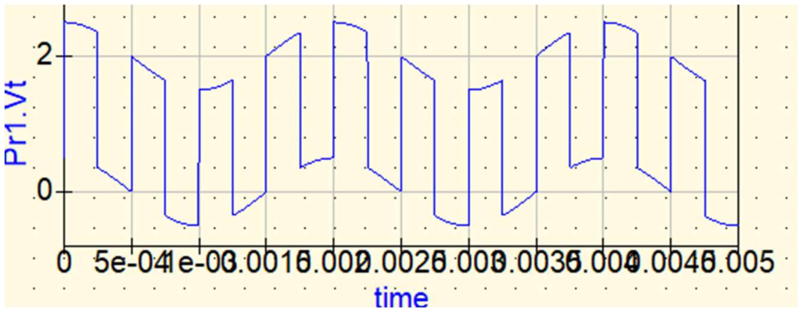


On changing the values of duration of the high pulse and low pulse we obtain the following graphs:

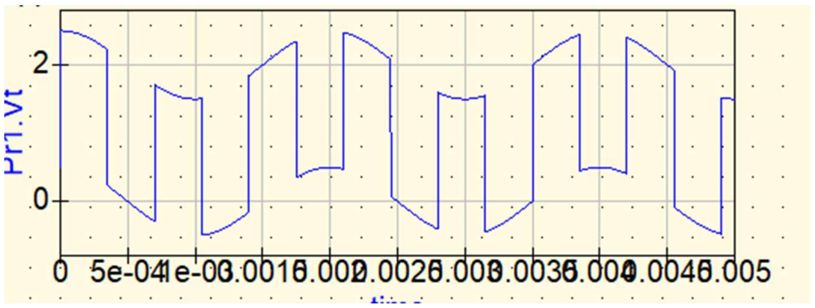
- TH = TL = 0.15ms



- TH = TL = 0.25ms



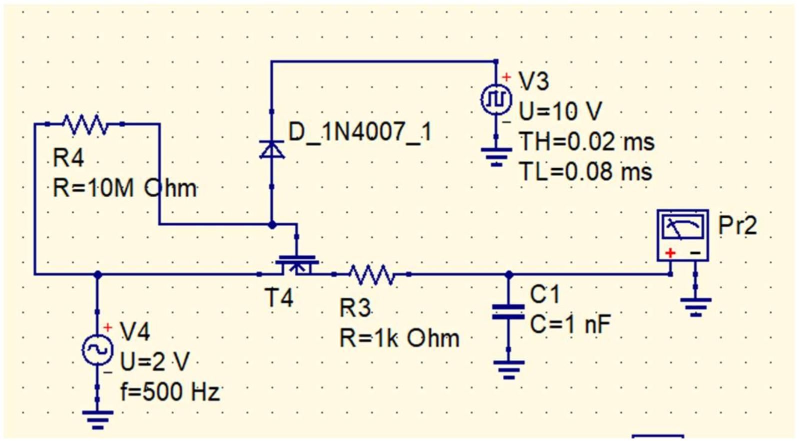
- TH = TL = 0.35ms



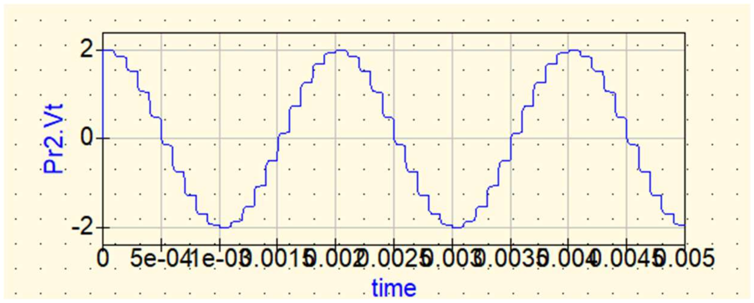
Hence it is observed that if the TH and TL values are increased the Frequency of the Output Voltages gets significantly decreased .

(b)

- **Circuit Diagram:**

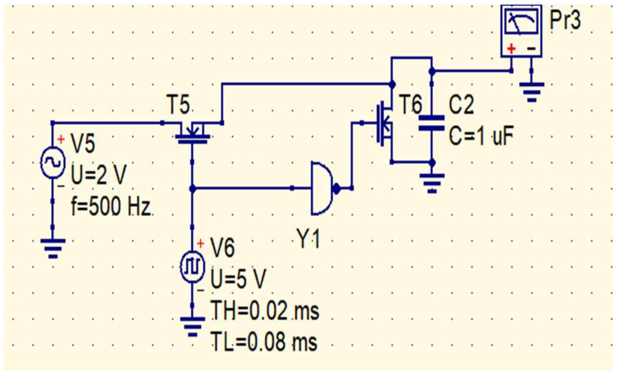


- **Graphs:**

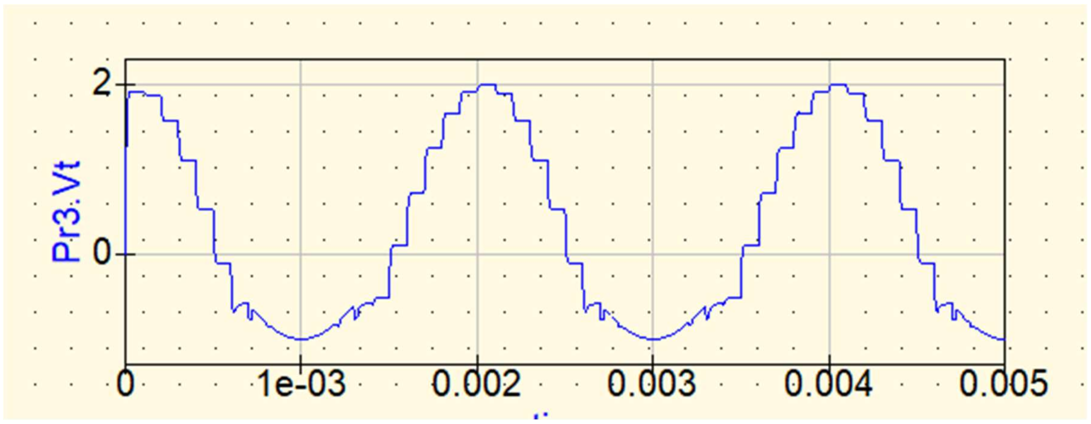


(c)

- **Circuit Diagram:**



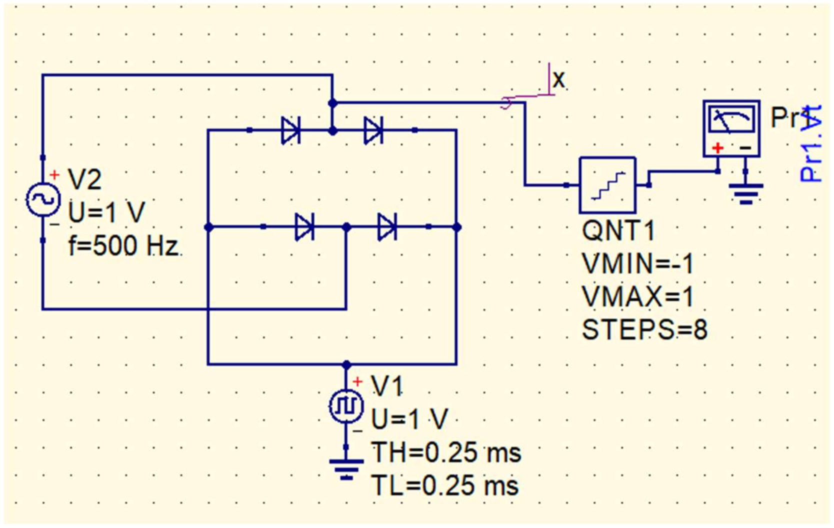
- **Graph:**

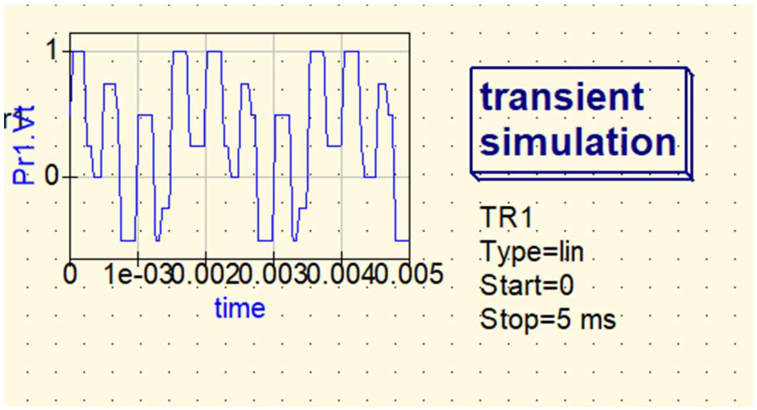


<u>AHP Component – 5</u>

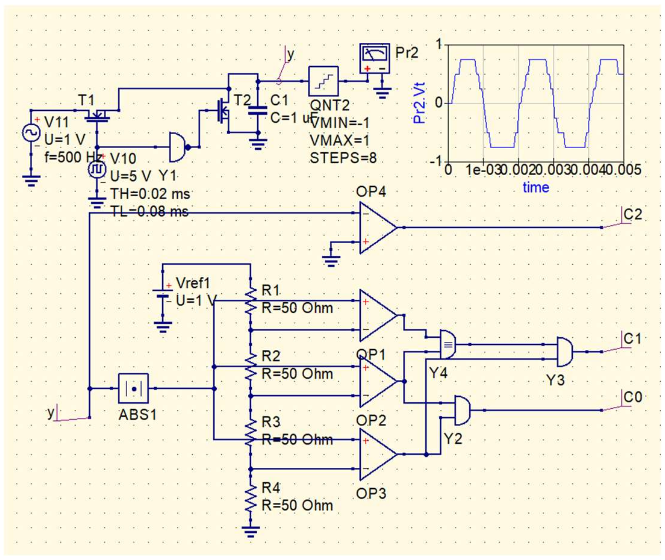
# PES1UG22EC321

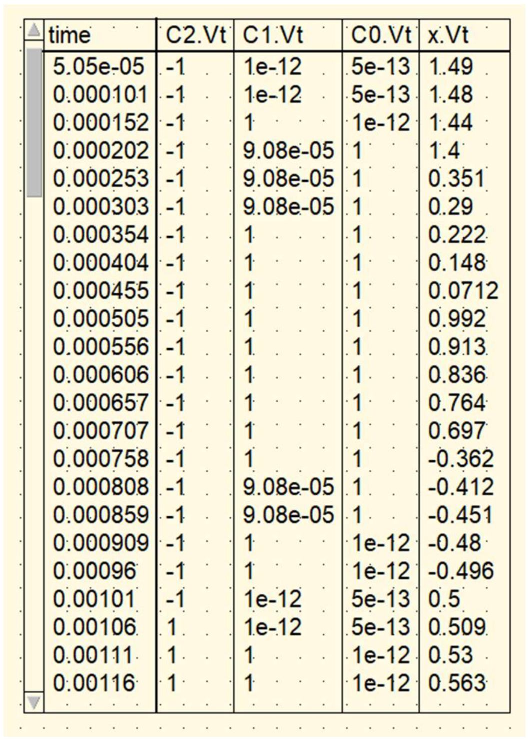
1)





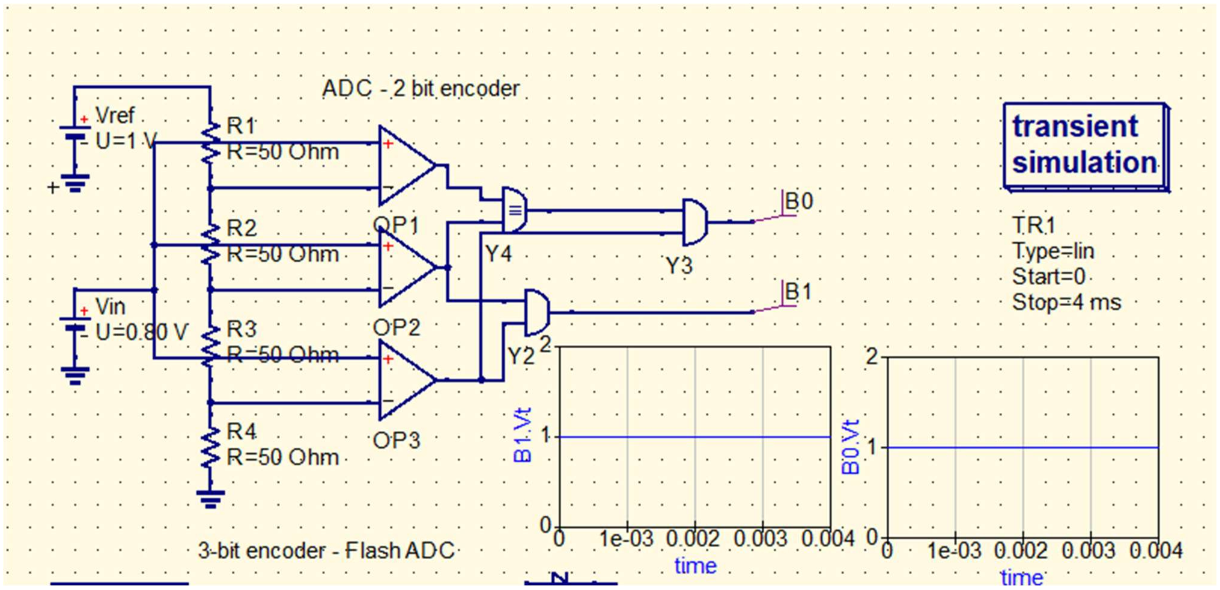
2)



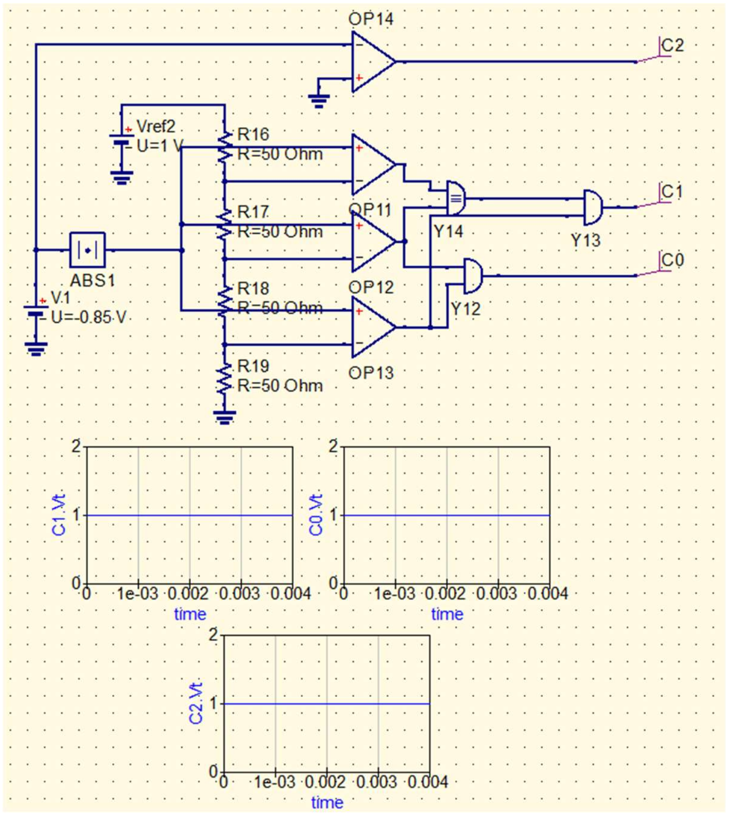


time	C2.Vt	C1.Vt	C0.Vt	x.Vt
5.05e-05	-1	1e-12	5e-13	1.49
0.000101	-1	1e-12	5e-13	1.48
0.000152	-1	1	1e-12	1.44
0.000202	-1	9.08e-05	1	1.4
0.000253	-1	9.08e-05	1	0.351
0.000303	-1	9.08e-05	1	0.29
0.000354	-1	1	1	0.222
0.000404	-1	1	1	0.148
0.000455	-1	1	1	0.0712
0.000505	-1	1	1	0.992
0.000556	-1	1	1	0.913
0.000606	-1	1	1	0.836
0.000657	-1	1	1	0.764
0.000707	-1	1	1	0.697
0.000758	-1	1	1	-0.362
0.000808	-1	9.08e-05	1	-0.412
0.000859	-1	9.08e-05	1	-0.451
0.000909	-1	1	1e-12	-0.48
0.00096	-1	1	1e-12	-0.496
0.00101	-1	1e-12	5e-13	0.5
0.00106	1	1e-12	5e-13	0.509
0.00111	1	1	1e-12	0.53
0.00116	1	1	1e-12	0.563

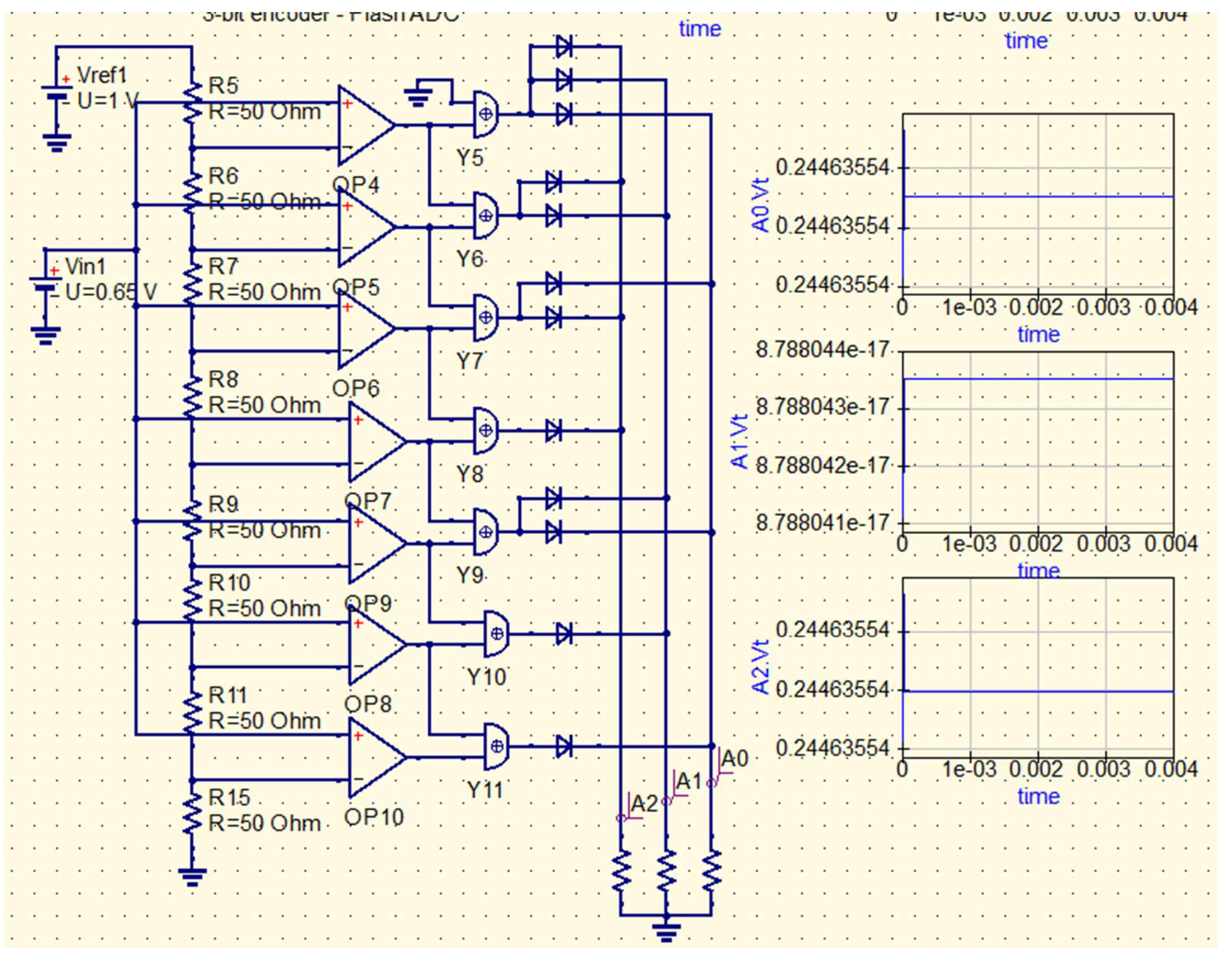
3)


ADC - 2 bit encoder
Vref
U=1 V
R1
R=50 Ohm
R2
R=50 Ohm
R3
R=50 Ohm
R4
R=50 Ohm
Vin
U=0.80 V
OP1
OP2
OP3
Y4
Y3
Y2
B0
B1
transient simulation
TR1
Type=lin
Start=0
Stop=4 ms
B1.Vt
B0.Vt
time
3-bit encoder - Flash ADC

4)


OP14
C2
Vref2
U=1 V
R16
R=50 Ohm
R17
R=50 Ohm
R18
R=50 Ohm
R19
R=50 Ohm
OP11
OP12
OP13
Y14
Y13
Y12
C1
C0
ABS1
V1
U=-0.85 V
C1.Vt
C0.Vt
C2.Vt
time

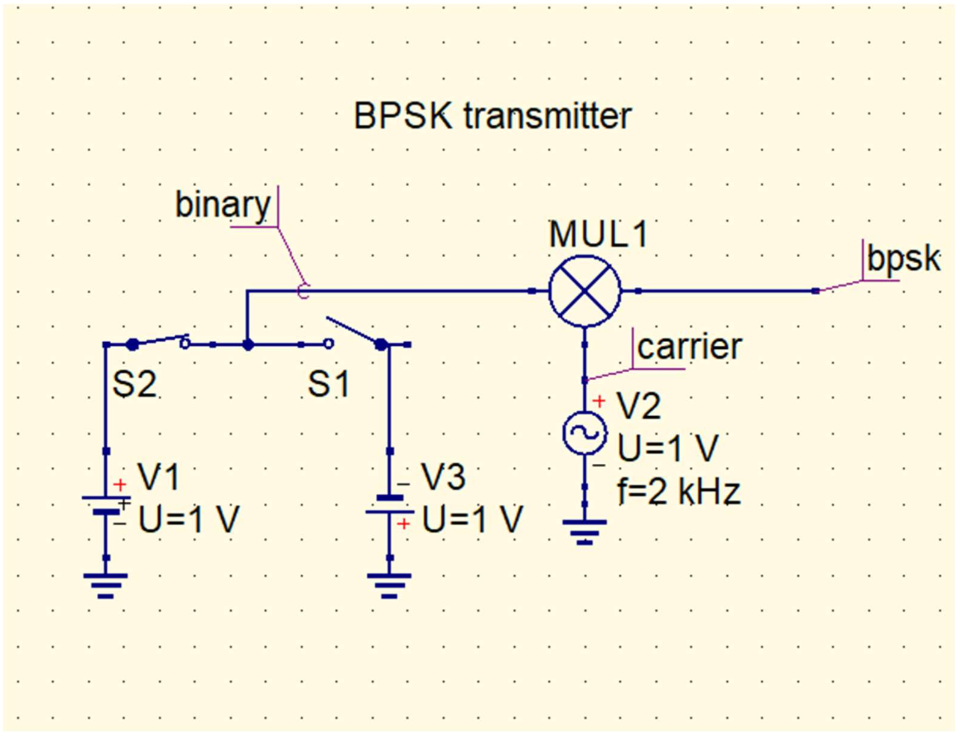
5)



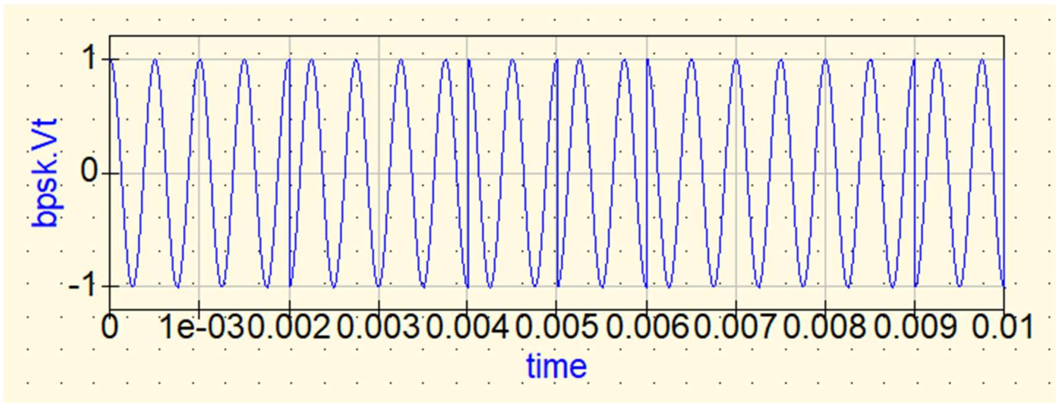
AHP – 10

# PES1UG22EC321

1) BPSK Transmitter:



Simulation Graph:



<u>Inferences :</u>

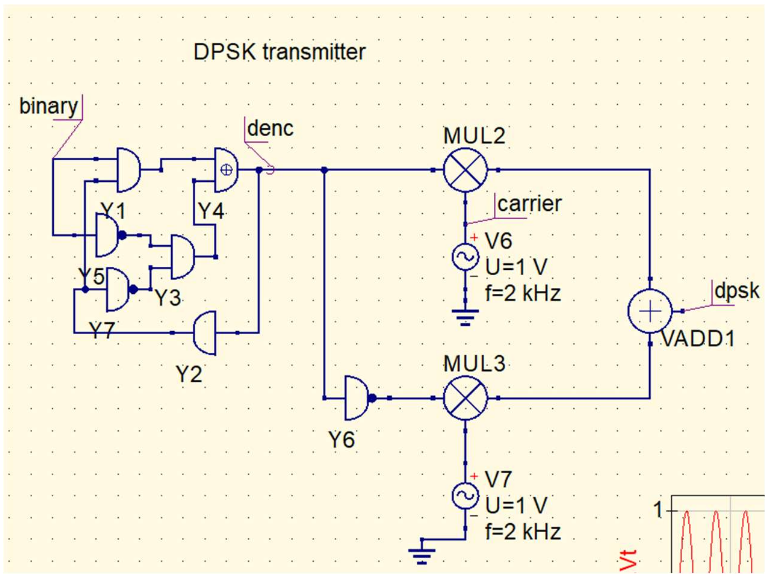
a) Carrier Frequency:

<u>Higher Frequency</u>: Increasing the carrier frequency shortens the wavelength, which can lead to better spectral efficiency and potentially higher data rates. However, it may also increase susceptibility to noise and attenuation.

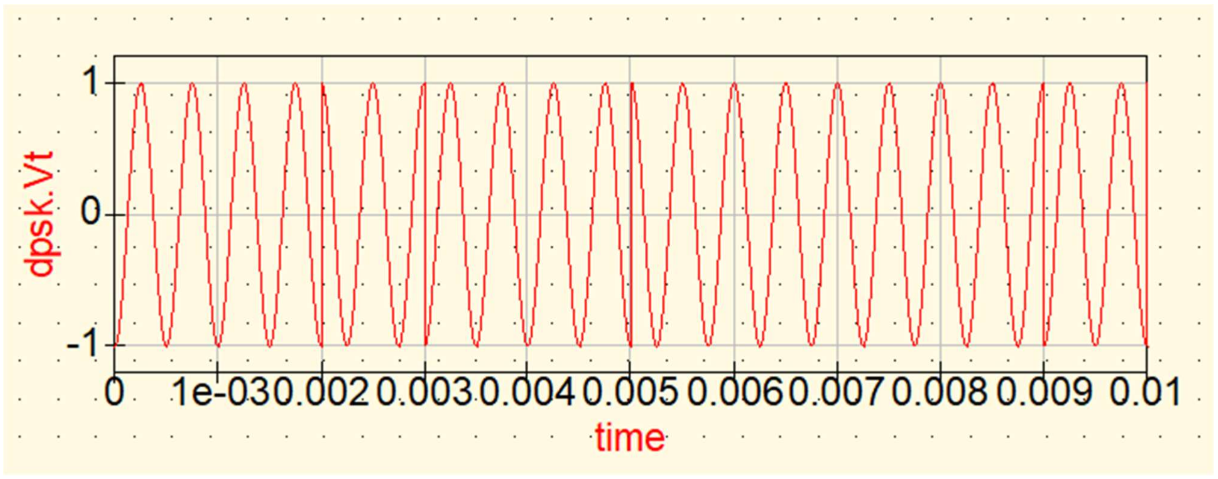
<u>Lower Frequency</u>: Lowering the carrier frequency extends the wavelength, which may improve signal propagation over longer distances but can reduce spectral efficiency and data rates.

b) Carrier Voltage: Increasing the carrier voltage can increase the signal strength and improve its robustness against noise. However, excessively high voltages may introduce nonlinear distortions.

2) DPSK Transmitter:



Simulation Graph:



Inferences:

- Voltage Levels:

<u>DC Source Voltage</u>: Increasing the voltage of the DC sources representing binary 1 and binary 0 can affect the amplitude of the modulated signals. Higher voltages may lead to stronger modulated signals, improving their robustness against noise. However, excessive voltage levels can introduce nonlinear distortions and increase power consumption.
<u>Modulator Input Voltage</u>: Adjusting the input voltage of the product modulators can directly impact the modulation depth and thus the phase shift of the DPSK signal. Higher input voltages may result in larger phase shifts, altering the signal constellation and potentially affecting demodulation accuracy.

- Carrier Frequency:

<u>Higher Frequency</u>: Increasing the carrier frequency shortens the wavelength of the carrier signal. This can lead to better spectral efficiency and potentially higher data rates. However, higher frequencies may also increase susceptibility to noise and interference, especially in high-frequency bands.
<u>Lower Frequency</u>: Lowering the carrier frequency lengthens the wavelength, which may improve signal propagation over longer distances. However, lower frequencies typically result in lower data rates and may require larger bandwidths for transmission.