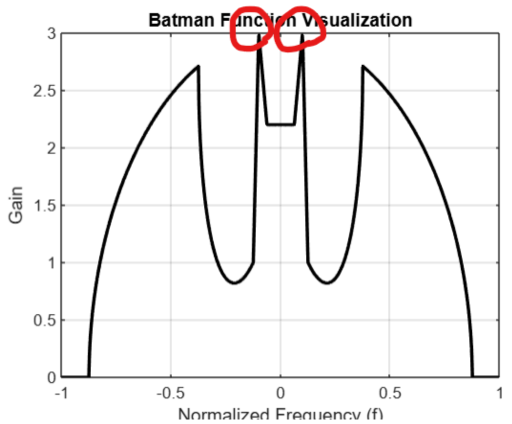
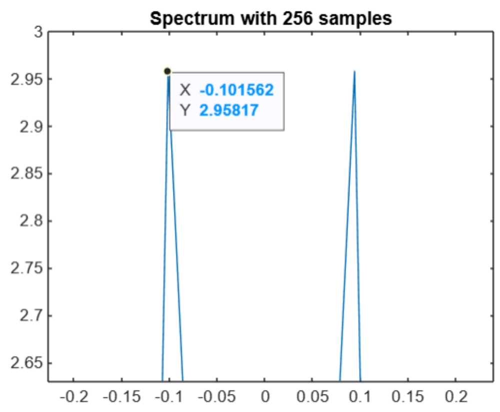
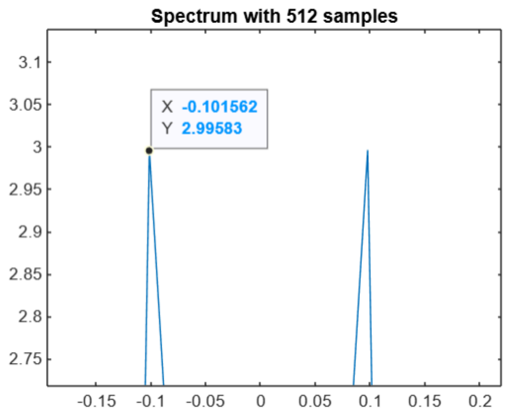
Batman Frequency Sampling Design



My opinion on the minimum number of samples is that it should be enough to make the peak value recognizable (Red circles in above image).

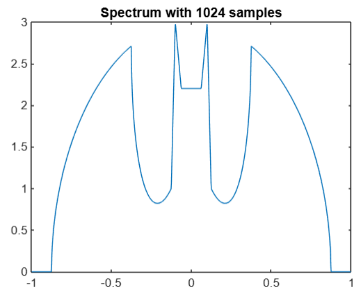


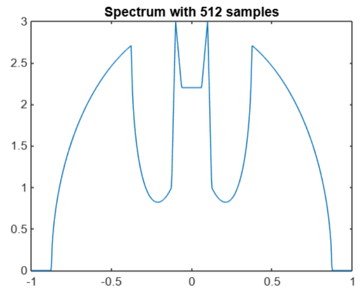


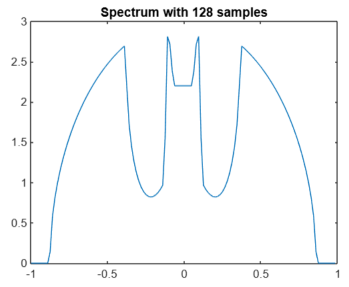
Therefore, I tried several trials with sample numbers 2 powers, and like the above image, for sampling number 512 I got 2.99 peak point, which is a reasonable error from original peak value (under 0.01 error). For 256 samples, I got 2.95, which means 0.05 error, given the peak value is 3.0, I thought this 16% (=0.05/3.0) error is critical.

- **My Code**

```Python
N = 512
f = linspace(-1, 1, N);        % normalized frequency range
x = batman(f);                % call the batman function
X = ifft(fftshift(x));
[H,w] = dtft(X, N);
plot(w/pi, abs(H))
title(['Spectrum with ', num2str(N), ' samples']);
```







And there are three plots for N = 1024 / N = 512 / N = 128. For each samples,

- When a number much higher than the minimum acceptable is used, the spectrum appears smooth and detailed. And less aliasing / distortion in the frequency response. But it has the cost of more computation and memory.
- When the minimum acceptable number of samples are used, you can identify the general shape and main lobes in the spectrum. Some spectral leakage may happen but you can efficiently execute with good-enough frequency resolution.
- When a number lower than the minimum acceptable is used, Spectrum can look jagged or aliased, and you lose important details of the signal's frequency content. That's because inverse FFT produces a poor approximation of the signal. This results in a frequency spectrum that's unreliable.

Upsample and Downsample

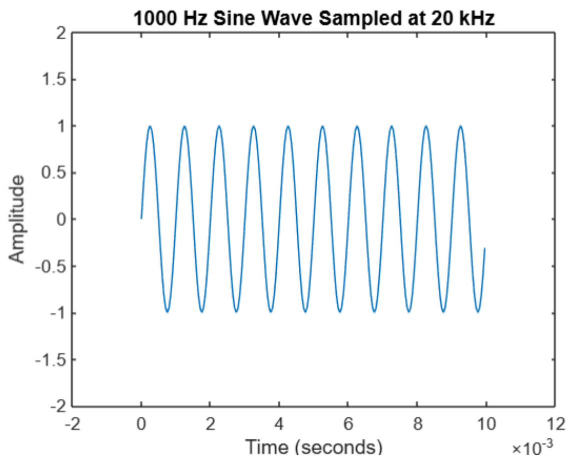
2) Generate samples of a 1000 Hz sine wave that is sampled at 20 kHz. Play the tone in MATLAB using the sound command to get an idea of what it sounds like.

- code

```
Python
fs = 20000;          % Sampling frequency in Hz
f = 1000;            % Sine wave frequency in Hz
duration = 1;        % Duration in seconds

t = 0:1/fs:duration;               % Time vector
y = sin(2 * pi * f * t);           % Generate sine wave

% Plot the waveform (first few milliseconds for clarity)
figure;
plot(t(1:200), y(1:200));          % Plot the first 200 samples (10 ms)
axis([-0.002 0.012 -2 2])
xlabel('Time (seconds)');
ylabel('Amplitude');
```

```
title('1000 Hz Sine Wave Sampled at 20 kHz');

% Play the sound
sound(y, fs);
```

- Explanation



Like the above image, I could find 10 peak points within 0.01 seconds, which means 1000 Hz sine wave, And 200 samples corresponding to 0.01ms, which means sampled at 20 kHz.
This signal sounds like a monotonous beeping sound like sound check.

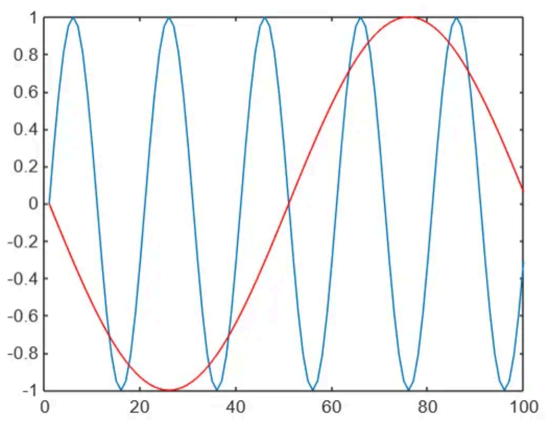
3) Upsample the sinewave to 100 kHz. Listen to the signal after you have inserted the zeros and then after you perform the interpolation, and describe the results. Plot samples of the waveform before and after upsampling, ensuring that the time axis corresponds to real time (e.g., seconds) and not sample time. Are the waveforms the same? Explain.

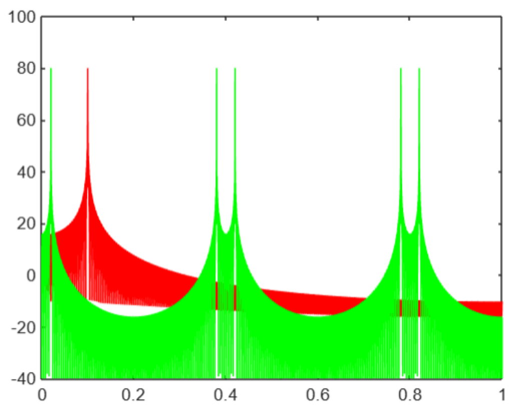
- code

```Python
clc;
clear;

fs = 20000;          % Sampling frequency in Hz => 20 kHz
f = 1000;            % Sine wave frequency in Hz => 1000 Hz
duration = 1;        % Duration in seconds

t = 0:1/fs:duration;            % Time vector
x = sin(2 * pi * f * t);        % Generate sine wave

size(x) % Should display [1 20001]
```

```
% Upsample to 100 kHz
% Upsampling factor L = 100 kHz / 20 kHz = 5
L = 5;
% Create upsampled signal by inserting L-1 zeros between each sample
x_up = zeros(1, length(x) * L);

x_up(1:L:end) = x; % Place original samples at every Lth position
fs_up = fs * L; % New sampling frequency = 100 kHz
t_up = 0:1/fs_up:duration; % New time vector

h = firpm(100, [0 0.2 0.3 1], [5 5 0 0]);
x_upsampled = conv(x_up, h);

figure(1);
plot(x(101:200))
hold
plot(x_upsampled (101:200),'r')
hold

figure(2);
[Hy,w]=dtft(x(100:end),100000);
plot(w/pi,20*log10(abs(Hy)),'r')
hold
[Ht,w]=dtft(x_up(100:end),500000);
plot(w/pi,20*log10(abs(Ht)),'g')

axis([0 1 -40 100])
```

- Explanation





Given the original 20 kHz sampling frequency, upsampling to 100 kHz means 5 times interpolation. Therefore the above plot before and after interpolation looks good. Those waveforms look similar excluding their wavelength. That's because I used a proper low pass filter.

And, Sound after zero padding was very spiky and contains high-frequency artifacts. So the sound was harsh or distorted. Sounds after interpolation were much cleaner and smoother. This is because during filtering our FIR filter made it into a clean sine wave.

4) Downsample the original sinewave to 4 kHz, with and without the anti-aliasing filter. Does the antialiasing filter make a significant difference in this case? Why or why not? Again plot the original and downsampled signal using real time on the horizontal axis, and explain any differences.

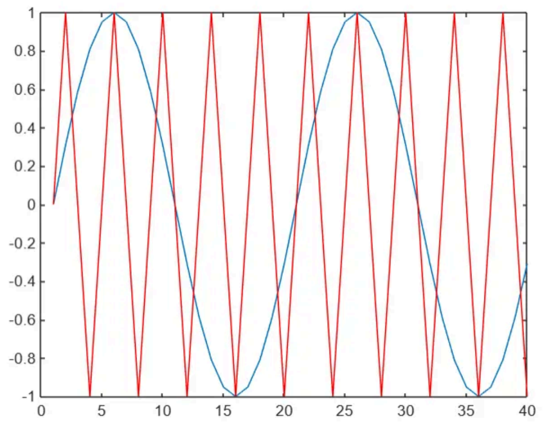
- code

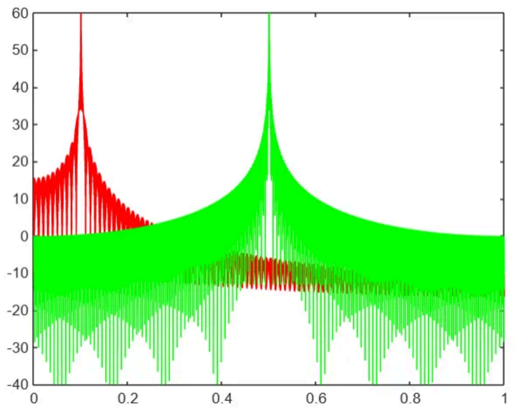
```
Python
clc;
clear;

fs = 20000;          % Sampling frequency in Hz => 20 kHz
f = 1000;            % Sine wave frequency in Hz => 1000 Hz
duration = 1;        % Duration in seconds

t = 0:1/fs:duration;            % Time vector
x = sin(2 * pi * f * t);        % Generate sine wave

size(x) % Should display [1 20001]

% Apply the LPF to the signal before downsampling
h = firpm(50, [0 0.2 0.3 1], [1 1 0 0]);    % Parks-McClellan low-pass filter
```

```
x_filtered = filter(h, 1, x);             % Filter the signal

% Downsample to 4 kHz like this x_down = x(1:5:end);
x_down = x_filtered(1:5:end);             % Downsample by a factor of 5
fs_down = fs / 5;                         % New sampling rate: 4 kHz

% % In case of non-filtered
% x_down = x(1:5:end);             % Downsample by a factor of 5
% fs_down = fs / 5;                         % New sampling rate: 4 kHz

% plot and display
t_down = 0:1/fs_down:duration;           % Time vector for downsampled signal

figure(1);
plot(x(101:140))
hold
plot(x_down(101:140),'r')
hold

figure(2);
[Hy,w]=dtft(x(100:end),40000);
plot(w/pi,20*log10(abs(Hy)),'r')
hold

[Ht,w]=dtft(x_down(100:end),10000);
plot(w/pi,20*log10(abs(Ht)),'g')
hold

axis([0 1 -40 60])
```

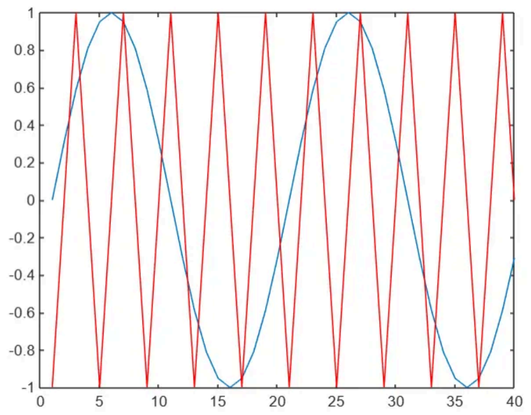
- Explanation

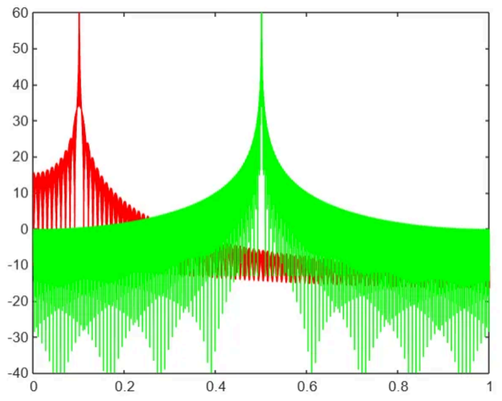
a. Before applying filter





b. After filtering





Downsampling the original sine wave to 4 kHz means 5 times downsampling. Original signal had a 0.1π peak frequency. Therefore even if we extend this signal 5 times horizontally, it still falls into our interest region, which is 0 ~ π. Even without a filter, It works fine here due to signal characteristics, But the slight difference is, the filter removes higher frequencies if they exist.

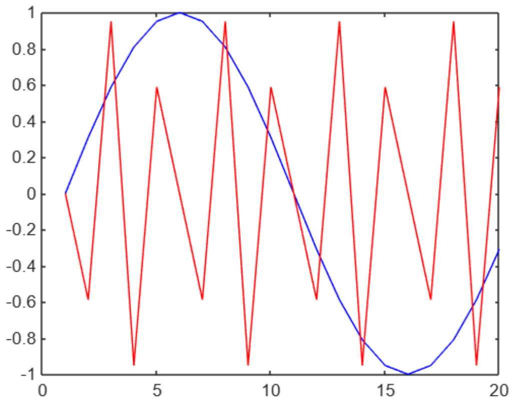
5) Predict what would happen if you kept only one of every 12 samples of the original sine wave, without using any low-pass filter. Be as precise as possible. Then listen to the result and see if you were correct.

- Code

```
Python
clc;
clear;

% fs = 20000;         % Sampling frequency in Hz => 20 kHz
% But, 20000/12=1666.67 is outside the range for sound function [3000, 768000]
% So I used 40000 just for sound check
fs = 40000;
```

```
f = 1000;            % Sine wave frequency in Hz => 1000 Hz
duration = 1;        % Duration in seconds

t = 0:1/fs:duration;            % Time vector
x = sin(2 * pi * f * t);        % Generate sine wave

size(x) % Should display [1 20001]

% Downsample kept only one of every 12 samples of the original sine wave
x_down = x(1:12:end);             % Downsampling by factor of 12
fs_down = fs / 12;                % New sampling frequency: ~1666.67 Hz
t_down = 0:1/fs_down:duration;    % Time vector for downsampled signal

% Play original signal with sound function
sound(x, fs);

% Play downsampled signal with sound function
sound(x_down, fs_down);

% Plot Original signal with downsampled signal
figure;
plot(x(1:20), 'b'); % Plot first 500 samples of original
hold on;
plot(x_down(1:20), 'r');
```

- Explanation



The original signal sampled at 20 kHz plays a clean 1000 Hz tone. But the downsampled version, at ~1.67 kHz, sounds distorted like robotic sound. This is because the new Nyquist rate is ~833 Hz, which is below the original 1000 Hz tone. Therefore, aliasing occurs and the tone is folded to a lower frequency.

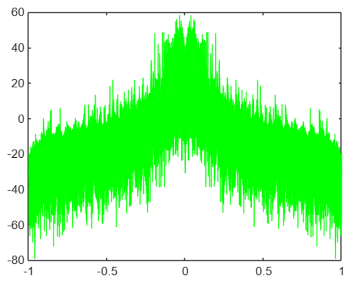
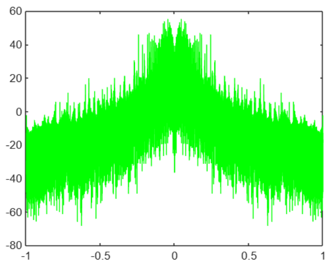
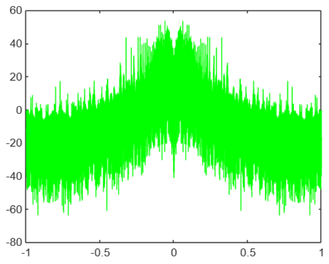
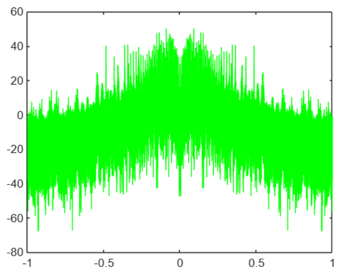
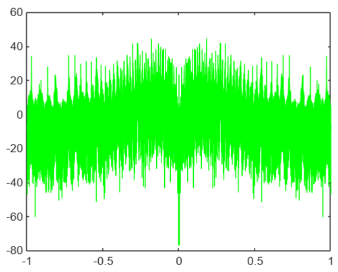
In my code, if I use 20 kHz and 12 times downsampling, sample frequency (20,000/12 = 1666.67) is outside the range for sound function. So I used 40 kHz just for sound check

6) Load the file tchaikovsky.mat from the MATLAB Files module on Canvas, which contains a portion of Tchaikovsky's famous "Dance of the Sugar Plum Fairy" sampled at 44.1 kHz. Downsample the file so the new sampling rates are 5/6 , 2/3 , 1/2 , 1/3 , 1/6 of the original rate respectively. Describe how the quality of the music clip changes. For the cases of 1/2 , 1/3 and 1/6, downsample without using the low-pass filter. How important is the anti-aliasing filter? [You can write a function that can change the sampling rate by any factor and then test it by just changing the variables of the function]

- code

```
Python
close all;
clc;
clear;

% This file contains portion of Tchaikovsky's famous
% "Dance of the Sugar Plum Fairy" sampled at 44.1 kHz.
load tchaikovsky.mat
originalFs = 44100;

% Downsample the file so the new sampling rates are
% 5/6 , 2/3 , 1/2 , 1/3 , 1/6 of the original rate respectively.
% For the cases of 1/2 , 1/3 and 1/6,
% downsample without using the low-pass filter.

ratios = [5/6, 2/3, 1/2, 1/3, 1/6];
labels = ["5/6", "2/3", "1/2", "1/3", "1/6"];
sampled_music = cell(1, length(ratios));  % Use cell array to store
different-length signals

for i = 1:length(ratios)
    ratio = ratios(i);

    newFs = originalFs * ratio;
    fprintf('\n--- Playing %s of original (%.0f Hz) ---\n', labels(i), newFs);

    % If ratio > 1/2, apply anti-aliasing filter
    if ratio > 1/2
        sampled_music{i} = resample(sugar_plum, round(ratio*1000), 1000);  %
Uses anti-aliasing
    else
        step = round(1/ratio);
        sampled_music{i} = sugar_plum(1:step:end);  % No filtering
    end

    % Plot a short segment of each version
    figure(i);
    plot(sampled_music{i}(60000*ratios(i):120000*ratios(i)));
    title(['Downsampled at ', labels(i), ' of Original Rate']);

    sound(y_resampled, round(newFs));
    pause(length(y_resampled)/newFs + 1); % Wait for playback to finish
end
```

- Explanation

From top left to bottom right, those images show log scale dtft result from ⅚ , ⅔ , ½ , ⅓ , ⅙ downsampling.

As the downsampling ratio increases, which means more aggressive downsampling, the sound becomes more distorted. For low downsampling (5/6, 2/3), the music still sounds recognizable, although slightly muffled. However, for stronger downsampling (1/2, 1/3, 1/6) without filtering, aliasing artifacts become very noticeable producing a harsh, unnatural sound. Those frequencies can alias into the lower frequency band, producing noise and distortion.
By applying a low-pass filter before downsampling, frequency components that would alias filtered, this leads to muffled but natural sounds.