Jacobs University Bremen

CO-522-B Intersession 2022

Communication Basics Lab

Lab Write-Up 4 : Digital Phase Locked Loop (PLL)

Kuranage Roche Rayan Nimnath Ranasinghe

Conducted on : 19/01/2022

Answers to the questions asked in the pre-lab.

What are the two functions of a PLL in a communications system?

The two functions of a PLL are as follows:

- Carrier Recovery: This process involves synchronising the local oscillator to the incoming signal.
- Symbol Time Recovery: This process involves the proper alignment of the sample times at the matched filter output.

What are the key components (operations) used to implement a PLL?

There are three key components used to implement a PLL:

- Multiplier: Used to get the product of the local reference from the VCO and the input reference
- Low Pass Filter: Used to remove the doubled frequency component, leaving only the baseband signal after multiplication using the multiplier
- VCO (Voltage Controlled Oscillator): The low frequency component left after the signal passes through the low pass filter produces a slow, varying voltage. This voltage is fed directly into the VCO which then influences the frequency of the oscillator as per the name depending on whether the output y(t) leads or lags with the input x(t).

When we write the second-order frequency response of the PLL, this expression relates the input and output __phase___ of the references.

What does the corner frequency of the PLL loop filter control?

The corner frequency of the PLL loop filter controls how fast the loop adapts to phase changes.

How should the loop filter corner frequency compare to the input reference frequency?

The loop filter corner frequency should be less than 0.1 times the input reference frequency or the clock being tracked.

What does it mean for a PLL to track?

For a PLL to track means to bring the control voltage of the VCO to 0 by tracking the phase of the input reference signal and the output reference signal. When it is tracked, the input and output frequencies are identical.

What does it mean for the PLL loop to be overdamped or underdamped?

For a PLL to be overdamped means that it requires a longer time to adapt to phase changes and when it is underdamped, it is faster, but it overshoots and causes oscillations.

The coefficients $\tau1$, $\tau2$ decide whether the system is overdamped or underdamped.

Why is an accumulator useful for a PLL implementation?
An accumulator uses a real number to keep track of the current phase. It decreases the number of variables to be used for definition and implementation and improves the simplicity. It can keep track of the time and phase instead of developing tables for it which is time consuming and a waste of computational power, especially since our code would do a Taylor expansion for every trigonometric calculation.

Why do we need to sample sin() in our lookup table more finely than at the normal sample rate of the system?
The sin() in our lookup table needs to be sampled more finely than at normal sample rate of the system as the PLL also requires the values which are in between the samples spaced by the sample period.

How do we keep our accumulator in the range [0,1]?
The accumulator is kept in the range [0,1] using the operation:

accum = accum- floor(accum)

What is the point of storing and restoring the state of the PLL for each block?
The storing and restoring the state of the PLL is essential for each block to store the current estimated values of phase and amplitude.
The PLL is carried out in a buffer-oriented style. Because buffer boundaries are artificial, the information from the previous buffer needs to be passed to the current buffer. Therefore the states need to be stored.

How do we handle signals with arbitrary amplitude?
The signals with arbitrary amplitude are handled by averaging the magnitude of the samples. After every block, the amplitude estimate is stored and used to scale the next block of samples to make the amplitude of the scaled signal approximately 1.

A paper design of your PLL, showing the important parameters that are needed for implementation

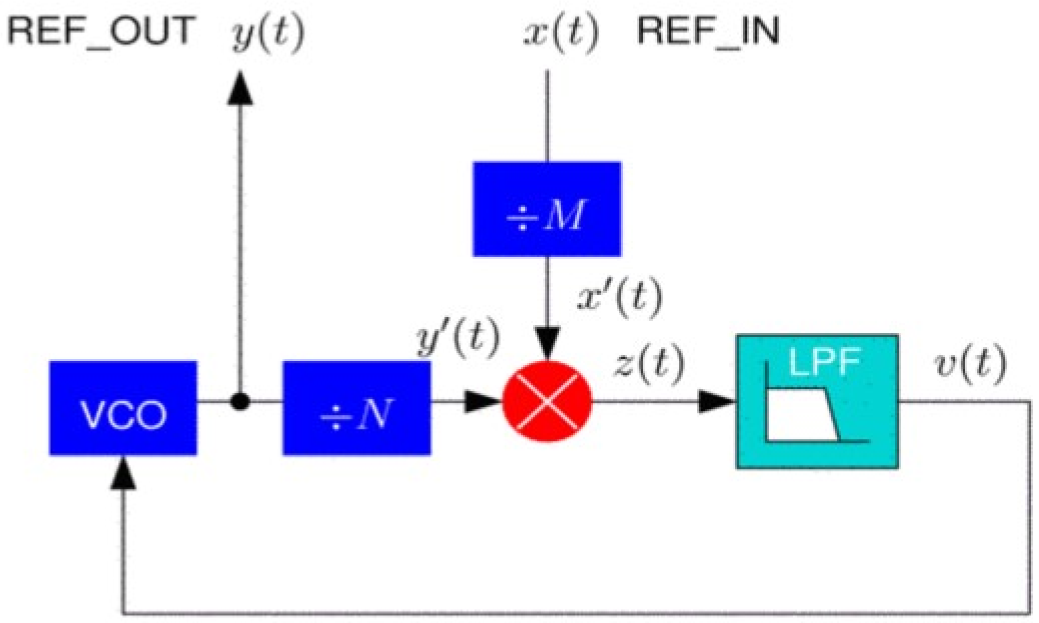


Figure 1: Basic PLL Operation Block

Here, first, when the product of the VCO's local reference y(t) and the input reference x(t) is computed, it gives us a signal with a low frequency component proportionate to the phase difference of y(t) and x(t), and a frequency component that has been doubled. This signal is passed through the Low Pass Filter (LPF) which eliminates the doubled frequency, leaving just the slowly changing voltage v(t) which is proportional to the phase difference of y(t) and x(t). The VCO receives this v(t) which responds with two types of output. When y(t) lags behind x(t), the oscillator speeds up and when y(t) leads x(t), the oscillator slows down. When ideally aligned, the control voltage of the VCO would be near zero which indicates that the loop is tracking which is the goal that's to be achieved.

A printout of your final MATLAB implementation of the PLL

The final PLL was implemented such that it could accommodate both a fixed buffer size and an arbitrary amplitude.
Our MATLAB code was divided into three parts as follows:

- A file containing the functions for storing the parameters and initialization of the block (pll_init.m)
- A file containing the functions for buffer processing of input items to generate a buffer of outputs (pll.m)
- And finally, a test file containing testing code to simulate buffer-oriented processing (test_pll.m)

The code was tested with a variety of parameters which will be addressed in the next part.

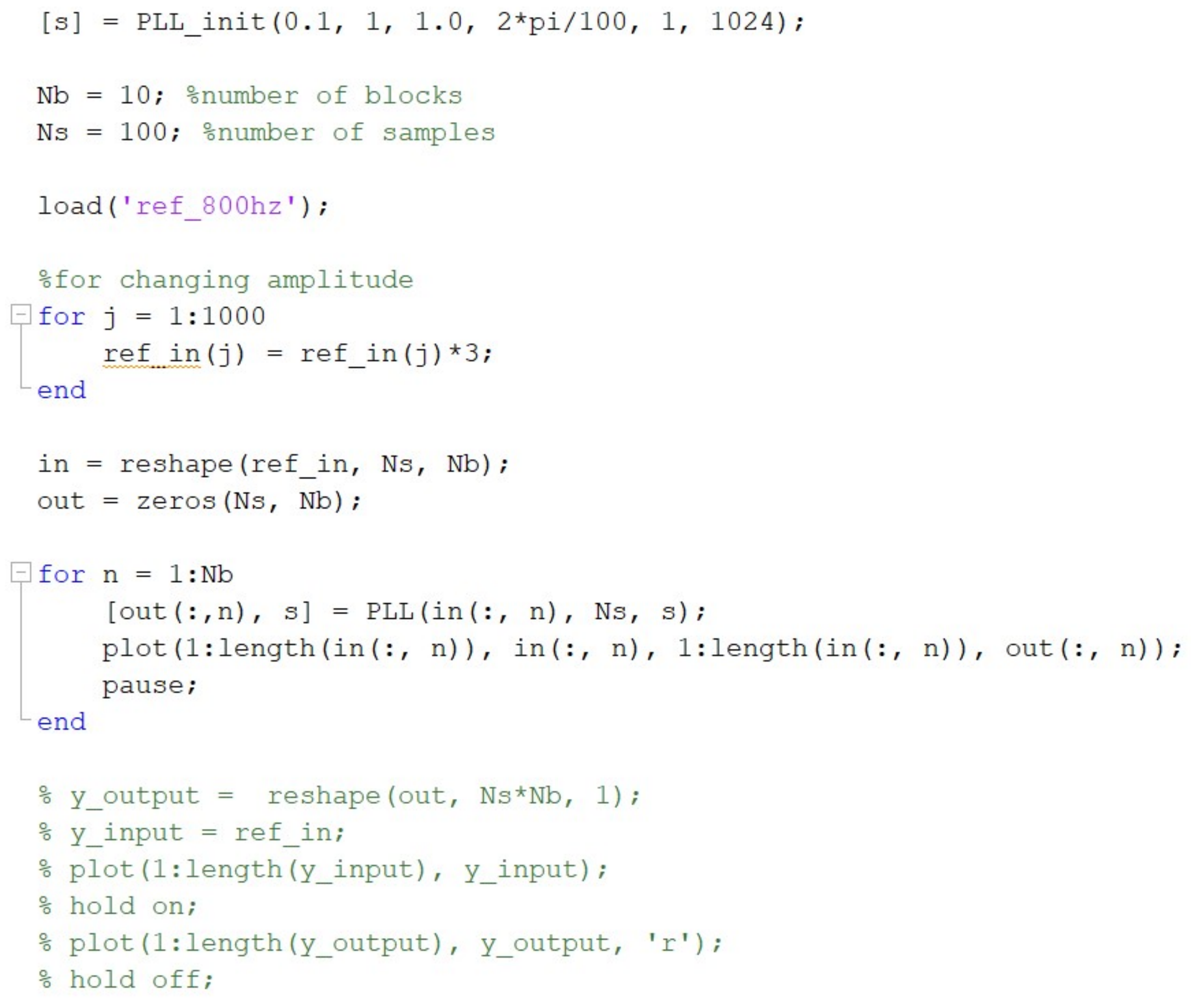
MATLAB code for ‘pll_init.m’

```
function [s] = PLL_init(f, D, k, w0, T, table_size)
%defining parameters

s.f = f;
s.D = D;
s.k = k;
s.w0 = w0;
s.T = T;

%creating a look-up table

for i=0 : table_size-1
   s.sine_table(i+1) = sin(2*pi*i/table_size);
end

%initializing filter coefficients

s.tau1 = s.k/(s.w0)^2;
s.tau2 = 2*s.D/s.w0 - 1/s.k;
s.a1 = -(s.T - 2*s.tau1) / (s.T + 2*s.tau1);
s.b0 = (s.T + 2*s.tau2) / (s.T + 2*s.tau1);
s.b1 = (s.T - 2*s.tau2) / (s.T + 2*s.tau1);

%initializing state variables

s.out_old = 0.0;
s.z_old = 0.0;
s.v_old = 0.0;
s.accum = 0.0;


end
```

MATLAB code for ‘pll.m’

```
function [out, state_out] = PLL(in, N, s)

out = zeros(size(in));

%For changing amplitude
amp = 0;

for sample_idx = 1:N
    amp = amp + abs(in(sample_idx));
end

amp_est = amp/N/(2/pi);

for i = 1:N

    scale(i) = in(i)/amp_est;

    %computing phase difference
    z = scale(i) * s.out_old;
    v = s.a1*s.v_old + s.b0*z + s.b1*s.z_old;
    s.accum = s.accum + s.f - (s.k/(2*pi))*v;
    s.accum = s.accum - floor(s.accum);

    %Use the sine table to calculate the output
    out(i) = s.sine_table(floor(1024*s.accum)+1);

    %update the state variables
    s.out_old = out(i);
    s.z_old = z;
    s.v_old = v;
end

state_out = s;
```

MATLAB code for ‘test_pll.m’

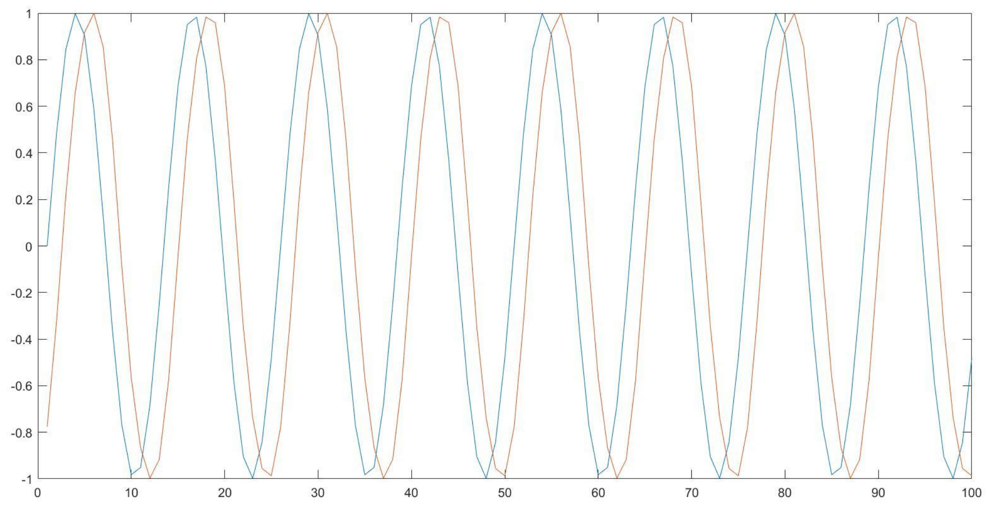


```matlab
[s] = PLL_init(0.1, 1, 1.0, 2*pi/100, 1, 1024);

Nb = 10; %number of blocks
Ns = 100; %number of samples

load('ref_800hz');

%for changing amplitude
for j = 1:1000
    ref_in(j) = ref_in(j)*3;
end

in = reshape(ref_in, Ns, Nb);
out = zeros(Ns, Nb);

for n = 1:Nb
    [out(:,n), s] = PLL(in(:, n), Ns, s);
    plot(1:length(in(:, n)), in(:, n), 1:length(in(:, n)), out(:, n));
    pause;
end

% y_output =  reshape(out, Ns*Nb, 1);
% y_input = ref_in;
% plot(1:length(y_input), y_input);
% hold on;
% plot(1:length(y_output), y_output, 'r');
% hold off;
```

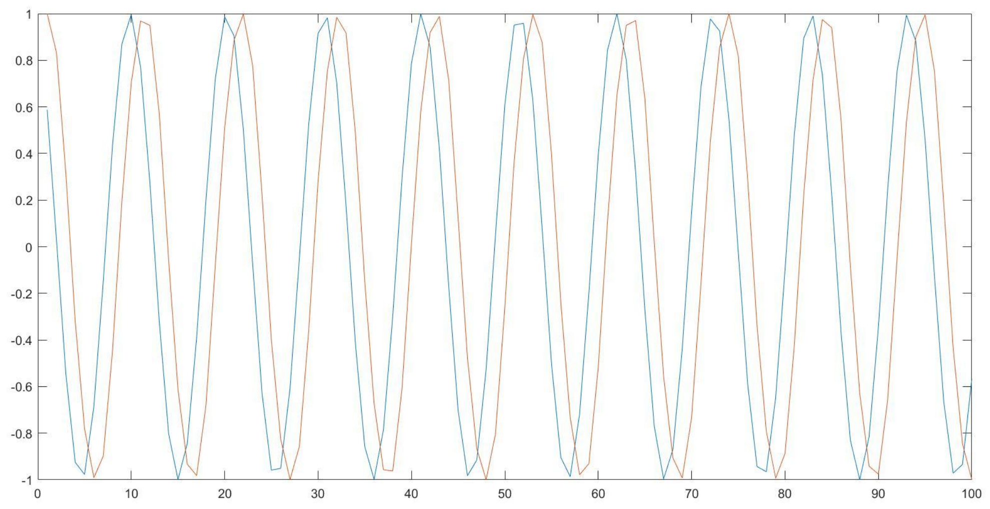
<u>A plot showing the simulated performance of the final PLL, demonstrating that the PLL can adapt to abrupt changes in frequency/phase</u>

For the case of fixed buffer sizes,

- For the file ‘ref_800hz.mat’ with an 800Hz sinusoidal signal, the following plot was produced:



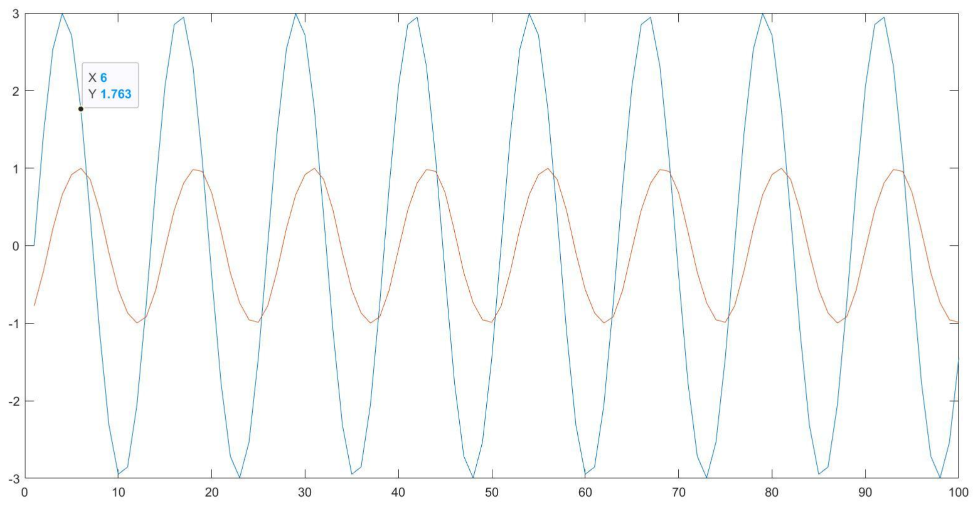
- For the file ‘ref_stepf.mat’ with four different frequencies, the following plot was produced:

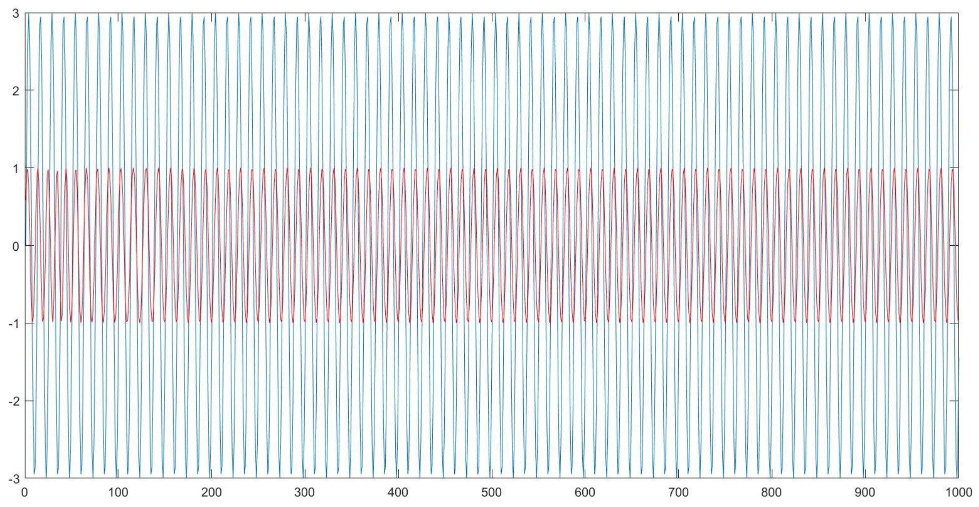


Please note that if the pause command is used on the test file, we can see how the phase starts to be tracked, i.e. the initial shapes are slightly different but after some iterations, they match the exact phase.

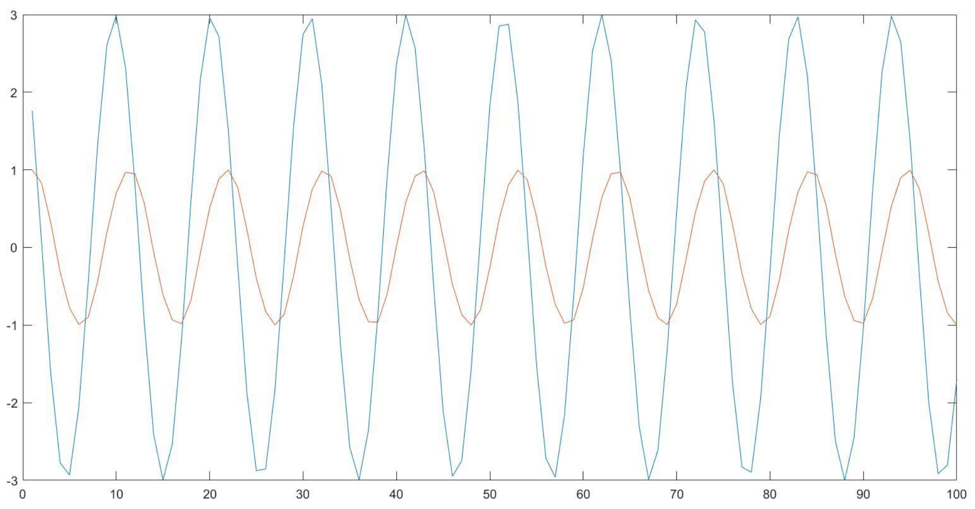
For the case of an arbitrary amplitude,

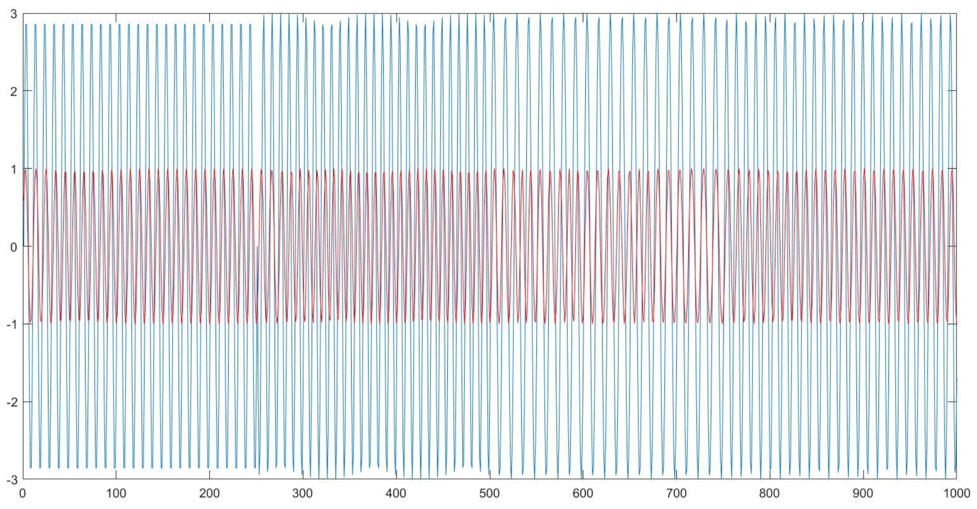
- For the file ‘ref_800hz.mat’ with an 800Hz sinusoidal signal, the following plot was produced:





- For the file ‘ref_stepf.mat’ with four different frequencies, the following plot was produced:





`

Please note again that if the pause command is used on the test file, we can see how the phase starts to be tracked, i.e. the initial shapes are slightly different but after some iterations, they match the exact phase.

As can be clearly seen, the PLL adapts itself well to sudden changes in frequency and continues tracking regardless.