1 Introduction

Basic PLL(Phase Locked loop) Operation

The PLL is a is a feedback demodulator that gives an output signal with a phase related to the phase of the input signal. With a sinusoidal input, the PLL generates a sinusoidal output with the phase closely tracked to the input. The phase is tracked using a feedback loop. It is used in synchronization tasks: carrier recovery, symbol timing recovery. The operation of PLL is shown in figure 1.

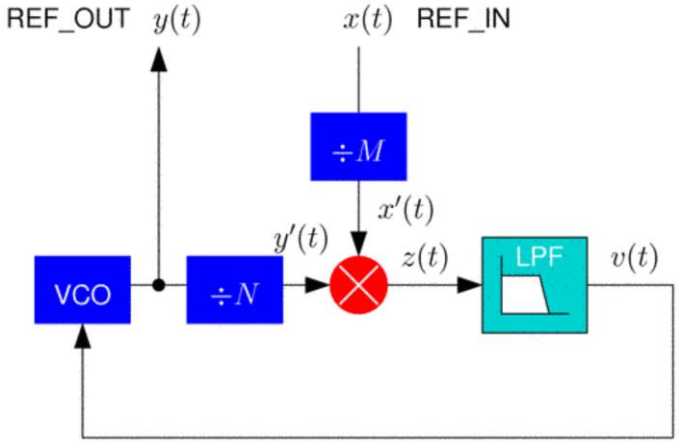


Figure 1: PLL operation chain

The signal y(t) from the Voltage controlled oscillator (VCO) is multiplied with the input signal x(t) giving a low frequency and a double frequency component. The latter is removed by the low pass filter (LPF) giving a voltage proportional to the phase difference. The VCO either speed up or slow down according to the position of the input compared to the output (leading or lagging). When the phases reach a proper alignment, the control voltage of VCO tends to 0 which shows that the phase is tracked.

Linearized Second Order PLL

With the input $x(t) = \cos(2\pi f t + \phi(t))$
And output $y(t) = \sin(2\pi f t + \hat{\phi}(t))$
After multiplication and low pass filtering, it gives a control voltage signal of :
$v(t) = \frac{1}{2}\sin(\hat{\phi} - \phi) \approx \frac{1}{2}(\hat{\phi} - \phi)$
The VCO tracks the phase with the equation:
$\hat{\phi}(t) = -k\int_{-\infty}^{t} v(\tau)d\tau$ k : loop gain
The input and output phase are related by equation:
$\frac{\hat{\Phi}(s)}{\Phi(s)} = \frac{kG(s)/s}{1+kG(s)/s}$ with G(s) the response of the loop filter.
A simple first order loop filter response can be expressed as
$G(s) = \frac{1+\tau_2 s}{1+\tau_1 s}$
With τ_1, τ_2 as coefficients that decide the response of the circuit (overdamped, critically damped, and underdamped). The loop filter parameters can be derived by
$\tau_1 = \frac{k}{\omega_0^2}$
$\tau_2 = \frac{2D}{\omega_0} - \frac{1}{k}$ with D: damping factor, ω_0: corner frequency (controls the loop's adaptation speed to phase changes).

Note that the damping factor is chosen depending on the desired response speed.

Lookup Tables

To guarantee real-time efficiency, a look-up table is used. Simply calling a trigonometric function will result in the program computing its polynomial approximation, which is not efficient in real-time DSP. In the code below, a lookup table is used. When using process (), signals are processed using the lookup table which is way faster than recomputing all the samples.

```
#include "math.h"

float cos_table[N_PERIOD];

init()
{
    int n;

    for (n=0; n<N_PERIOD; n++)
    {
        cos_table[n] = cos(2.0*pi*freq*n/sample_rate);
    }
}

process()
{
    int n;
    float input, output;

    for (n=0; n<N_PERIOD; n++)
    {
        output = input*cos_table[n];

        /* More processing here */
    }
}
```

Discretization of the PLL

To map continuous time loop filter response H(s) to discrete time, transformation $s = \frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}}$ is used. With T as the sample period and z is the discrete frequency from Z transform. The coefficients of the filter can be derived by:

$$y[n] = a_1 y[n-1] + b_0 x[n] + b_1 x[n-1]$$

$$a_1 = -\frac{T-2\tau_1}{T+2\tau_1} \qquad b_0 = \frac{T+2\tau_2}{T+2\tau_1} \qquad b_1 = \frac{T-2\tau_2}{T+2\tau_1}$$

For discrete PLL, discrete frequency with period T = 1.0 is used. Also, a phase accumulator method is used. The latter is a single real number that keeps track of the phase. At each time step, the accumulator indicates the current position in the sin lookup table. The accumulator follows the operation: $\text{accum} \leftarrow \text{accum} + f - \frac{k}{2\pi} v[n]$ where $\text{accum} + f$ is the phase increase of the sinusoidal signal due to advance of time and $-\frac{k}{2\pi} v[n]$ is the phase shift due to the control signal present on the VCO. The accumulator wraps the value that is higher than 1 or below 0 with the code:

accum = accum – floor(accum);

2 Execution/Evaluation

Check – off

Demonstrate to the TA that your PLL implementations work correctly. Your PLL should be able to handle changes in frequency of about plus or minus 10% of the nominal reference frequency.
Details of this is shown in lap write up section below.

Lab Write up

(1) Answers to the questions asked in the pre-lab.

1. What are the two functions of a PLL in a communications system?
PLL are used for two important synchronization tasks
Carrier recovery: synchronizes local oscillator to the incoming signal.
Symbol timing recovery: properly aligns the sample times at the matched filter output.
2. What are the key components (operations) used to implement a PLL?
LPF: Remove the double frequency component and leave only the baseband signal, which is a voltage component that varies according to the phase difference.
VCO: having the phase difference input, it causes the oscillator to speed up or slow down depending on the input output relative position (leading or lagging).
3. When we write the second-order frequency response of the PLL, this expression relates the input and output ***_phase_*** *of the references.*
4. What does the corner frequency of the PLL loop filter control?
Corner frequency controls how fast the loop adapts to phase changes.
5. How should the loop filter corner frequency compare to the input reference frequency?
The corner frequency should be lower (<0.1) than the frequency of the sinusoid (input reference frequency) or clock being tracked.
6. What does it mean for a PLL to track?
PLL tracks the phase of the input signal and output signal until the phase difference is small enough. When the phase of input and output signals reach proper alignment, the VCO voltage would be close to 0. When it is tracked, the input and output frequency are identical.
7. What does it mean for the PLL loop to be overdamped or underdamped?
Overdamped: requires a long time to adapt to phase changes.
Underdamped: fast response but tendency to overshoot the target leading to oscillations.
The coefficients τ_1, τ_2 decide whether the system is overdamped or underdamped.
8. Why is an accumulator useful for a PLL implementation?
Instead of having time variable t and a separate phase phi, a single real number called as accumulator is used. Accumulator keeps track of the current phase by telling the current position in the sin lookup table, where the lookup table holds just one sinusoidal cycle.
9. Why do we need to sample sin() in our lookup table more finely than at the normal sample rate of the system?
The sin() function needs to be sampled more finely as the PLL often needs samples which lie in between the samples spaced by the sample period and the phase can change continuously, in that interval in particular, to track the input.

10. How do we keep our accumulator in the range [0,1]?

The accumulator is kept in the range [0,1] by wrapping around when the value exceeds 1 or drops below 0. This is done by MATLAB with the code: accum = accum - floor(accum);. floor(accum) is the floor function of accum or the nearest integer to it. Thus the difference will always be in [0,1] (Property of the floor function).

11. What is the point of storing and restoring the state of the PLL for each block?

The state of the PLL is stored and restored after every block in order to store the current estimate of the phase and amplitude. The PLL is carried as buffer – oriented style. Because buffer boundaries are artificial, the information from the previous buffer need to be passed to the current buffer. Therefore the states need to be stored.

12. How do we handle signals with arbitrary amplitude?

The block of samples has to be scaled so that it has approximately unit amplitude. This is done 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.

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

Input signal: x(t)
Reference signal, output: y(t)
Product of input and output: z(t)
Low pass filter output- slowly varying voltage: v(t)

In the matlab code in (3), the parameters were implemented as:

Input signal: x(n)	initial input: xm1
Reference: y(n)	initial reference: y1
Product: z(n)	initial product: ym1
Voltage: v(n)	initial voltage: vm1

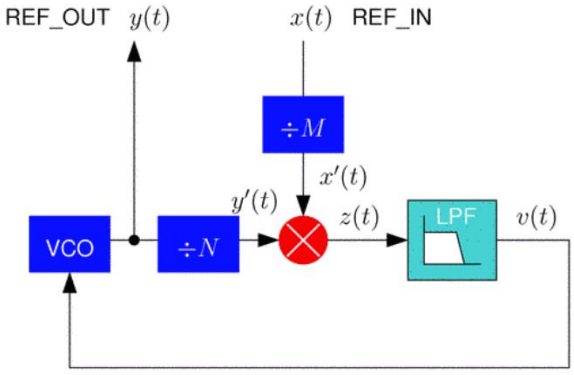


Figure 1. PLL Diagram

(3) A printout of your final MATLAB implementation of the PLL

The code with arbitrary amplitude is implemented inside the code but as a comment.

PLL initialization function code:

```
function [state, y] = pllinit(f, D, k, w0, T)
%% Creates and initializes a new phase locked loop.
% Inputs:
%         f - Nominal ref. frequency
%         D - Damping factor
%         k - Loop gain
%         w0 - Loop corner frequency
%         T - Sample period
%bfs = buffer size
% Outputs:
%          state    -   Current/Initial state
%% Add/Save parameters
    state.f= f;
    state.D= D;
    state.k= k;
    state.w0= w0;
    state.T=T;
%% compute coefficients
    tau1 = k/(w0 * w0) ;
    tau2 = 2*D/w0 - 1/k ;
    state.a1=-(T-2*tau1)/(T+2*tau1);
    state.b0=(T+2*tau2)/(T+2*tau1);
    state.b1=(T-2*tau2)/(T+2*tau1);
%% lookup table
    state.sin_table = sin(2*pi*linspace(0,1023/1024, 1024));
%% Create initialized state variables
    state.ym1 = 0;
    state.xm1 = 0;
    state.zm1 = 0;
    state.vm1 = 0;
    state.acc = 0;
    % for amplitude modulation
    state.amp_est = 1;
end
```

PLL function code:

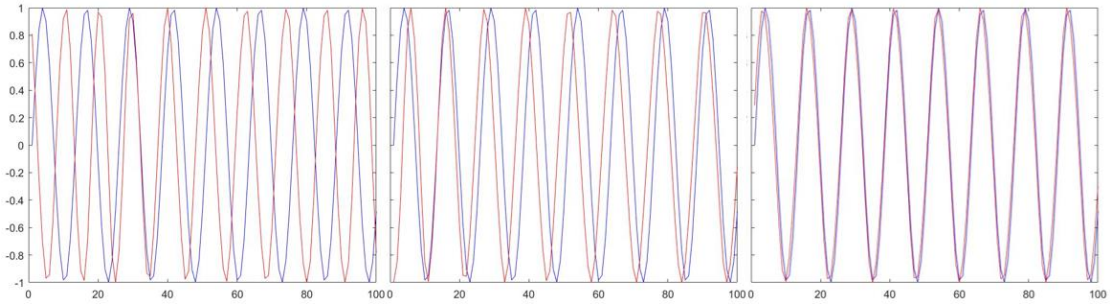
```
function[state_out,y] = pll(state, a)
%% Executes the phase locked loop .
% Inputs:
%   state       Current/Input state
%   a           input function
% Outputs:
%   state_out   Output state variables to be stored for next iteration
%% Get state
s = state;
z = zeros (size(a));
v = zeros (size(a));
y = zeros (size(a));
x = a;
amp = 0;

%% obtain the output of LPF
for n= 1:length(a)
    amp = amp + abs(x(n));
    z(n) = s.ym1*x(n)/s.amp_est;
    v(n) = s.vm1*s.a1 + z(n)*s.b0 + s.b1*s.zm1 ;
    s.zm1 = z(n) ;
    s.vm1 = v(n) ;
    % accumulator
    s.acc = s.acc + s.f - (s.k/(2*pi))*v(n) ;
    s.acc = s.acc - floor(s.acc);
    %sine table
    y(n) = s.sin_table(floor(1024*s.acc)+1);

    %
    s.ym1 = y(n);
    s.zm1 = z(n);
    s.xm1 = x(n);
end
    s.amp_est = amp/length(x)/(2/pi);
%% Return updated state
state_out = s;
```

PLL testing code:

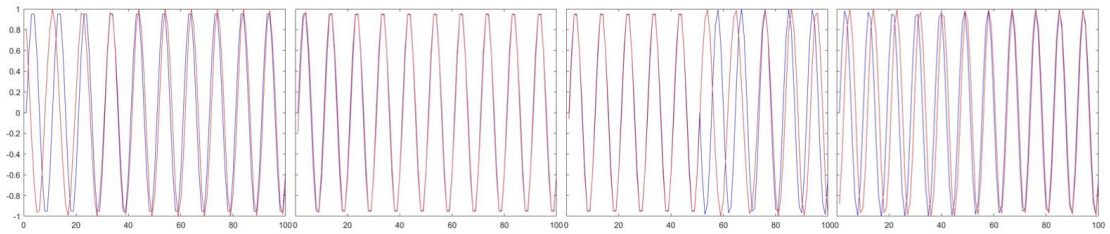
```
% Test the PLL.
% Global parameters
Nb = 10;    % Number of buffers
Ns = 100;   % Samples in each buffer
f = 0.1; k = 1; D = 1; w0 = 2*pi/100; T = 1;

%initialize PLL
pllstate = pllinit(f, D, k, w0, T);
load('ref_stepf');

%Generate random samples
x = ref_in;
%% for amplitude modualtion
for j = 1:1000
    x(j) = x(j)*3;
end
%% reshape buffers
xb = reshape(x, Ns, Nb);
%Output samples
y = zeros(Ns, Nb);

%Process each buffer
for k = 1:Nb
    [state_out y(:,k)] = pll(pllstate, xb(:,k));
end

%Convert individual buffers back into a continouos signal
y_out = reshape(y, Ns*Nb, 1);

plot(n,x, 'b', n, y_out, 'r');
```

(4) A plot showing the simulated performance of the final PLL, demonstrating that the PLL can adapt to abrupt changes in frequency/phase

Two reference data were used. Signal with constant 800Hz sinusoid (ref_800hz) and signal with four different frequencies (ref_stepf). The plot obtained by the two reference signals are shown below.



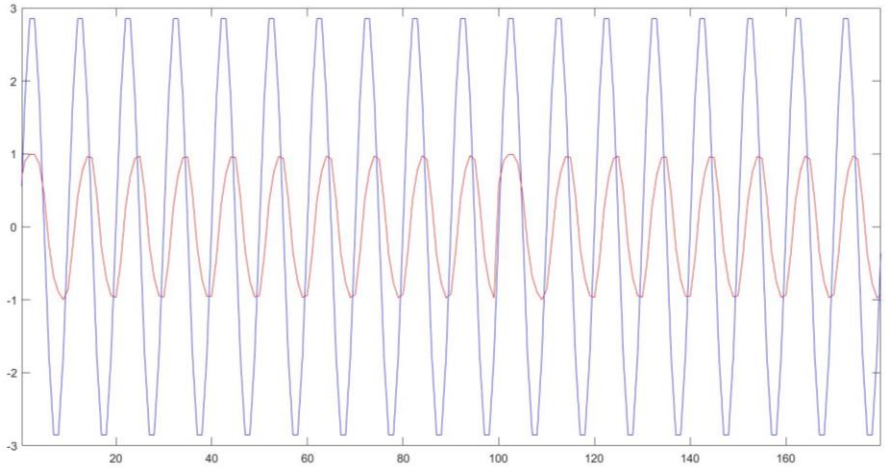
ref_800hz



ref_stepf

It can be seen that the code is tracking the phase.
Also, it tracks with the frequency change as shown in ref_stepf graph.

Graph with change of amplitude



Tracking is visible in the plot with the amplitude change. The amplitude doesn't affect the PLL.

3 **Conclusion**

In this lab, PLL was studied and implemented using Matlab. PLL is one of the most common synchronization methods used, in our case, for carrier recovery and symbol timing recovery. When implementing the PLL, the importance of the look-up table was perceived. The latter enables to guarantee real-time efficiency compared to simply calling trigonometric functions inside the program. As a last step, PLL discretization was realized using the phase accumulator method, which keeps track of the current phase at each time step. A condition on the accumulator is to be in [0,1], which can be realized in Matlab through subtracting the floor function of the accumulator from the accumulator value. The PLL was coded using matlab. As shown in lab write up, the graph showed that the PLL code tracked the phase. There were some difficulties in when checking the PLL function with the change of amplitude, but it was solved with a minor code fix.

4 **References**

[1] http://dsp-fhu.user.jacobs-university.de/?page_id=229