JACOBS UNIVERSITY BREMEN

NATURAL SCIENCE LABORATORY

DIGITAL SIGNAL PROCESSING LAB

SPRING SEMESTER 2021

Digital Phase Locked Loop Implementation on DSP Board

Instructor :
Ph.D. Fangning Hu

Author :
Haseeb Ahmed

1 Introducion

1.1 Phase locked loop C code design

A phase-locked loop can be thought of as a tracking algorithm for sinusoidal signals. It compares the phases of two signals, generally, it is the input and output signal phases. It is used in two different ways :

1. As s demodulator, where it is used to follow phase or frequency modulation and
2. to track a carrier or synchronizing signal which may vary in frequency with time.

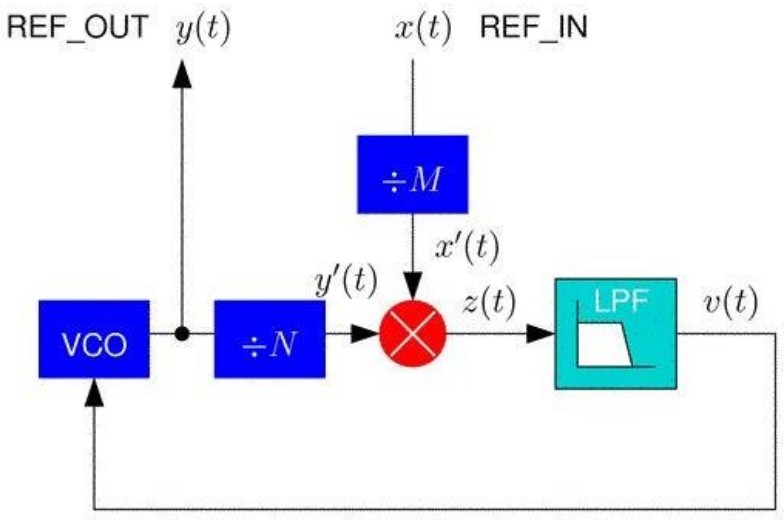

FIGURE 1 – Harmonics and sub-harmonic of PLL incoming reference

This can be easily understood by considering that the purpose of the PLL is to force the two signals coming into the phase comparator (the multiply operation) to have the same frequency [1].

In C code, we used the sine wave directly to accomplish these frequency divi- sions.

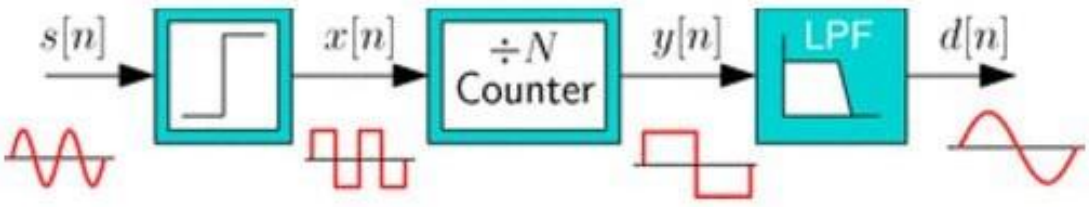

FIGURE 2 – Harmonics and sub-harmonic of PLL incoming reference

In the Figure 1, the sine wave is converted to a digital signal using a sign() ope- ration, which is then fed to a digital divide-by-N counter. Although the digital square wave y[n] is sufficient in many applications (such as symbol timing reco- very), if a sine wave is again needed, this can be obtained by low-pass filtering [1].

For the sine wave look up table we used the operation :

$$accum \leftarrow accum + f - \frac{k}{2\pi} v[n]$$

The output was generated using the following C code, specifically in lines 24 and 25.

```
1  /*=========================================================================
2   *  sin_tables.c
3   *          Contains code to setup and access a SIN and COS table for use in
4   *          real-time DSP programs.
5   *
6   *=========================================================================*/
7  #include"sin_tables.h" #include"math.h"
8
9  float sin_table[SIN_TABLE_SIZE];
10 float cos_table[SIN_TABLE_SIZE];
11
12 /*-------------------------------------------------------------------------
13  *  SIN_init()
14  *          Initializes the SIN and COS table with values of a single period.
15  *-------------------------------------------------------------------------*/
16 void SIN_init()
17 {
18
19       int i;
20
21       /* Initialize sine table */
22       for (i=0; i<SIN_TABLE_SIZE; i++)
23       {
24             sin_table[j]=sin(2.0*M_PI*( float)i/(float)SIN_TABLE_SIZE);
25             cos_table[i]=cos(2.0*M_PI*(  float)i/(float)SIN_TABLE_SIZE);
26       }
27 }
```

A second lookup table can be used to find a doubled frequency s2, if sin line 24 is locked to an incoming signal.

s2 = sin_table[(int)((float)SIN_TABLE_SIZE*accum)];

The C code was generated using the following lines of equation in the lineari- zed second-order and discretization of the PLL [1].

1. These were discussed very thoroughly in the matlab implementation of the PLL

The corresponding lines are the locations of the equations in pll.ccode.

$\tau_1 = \frac{k}{2\omega_0}$ ..... line 54

$\tau_2 = \frac{2D}{\omega_0} - \frac{1}{k}$ ..... line 55

$a_1 = -\frac{T - 2\tau_1}{T + 2\tau_1}$ ..... line 56

$b_0 = \frac{T + 2\tau_2}{T + 2\tau_1}$ ..... line 57

$b_0 = \frac{T - 2\tau_1}{T + 2\tau_1}$ ..... line 58

2 Execution and Evaluation

2.1 Problem 1

- A printout of your final C implementation of the PLL. You can just print out the pll_init() and pll() functions (you do not have to print all of the surrounding dsp_ap.c code !).

```
/*=========================================================================
 *  pll.h
 *          Global definitions for PLL code.
 *=========================================================================*/
#ifndef _pll_h_ #define
_pll_h_

/*-- Defines ------------------------------------------------------------------------------------------------*/

#define PLL_SEG_ID                  0/* IDRAM - fast. PLL doesn                    't need
 ↔    much */
#define PLL_BUFFER_ALIGN            4/* Just align on word boundary. */

/*-- Structures ---------------------------------------------------------------------------------------------*/
typedefstruct
{
      /* Filter parameters */
      float a1, b0, b1; float f0,
      damp_fact; float K;
      float mult;

      /* State variables */
      float      x_nm1;
      float      y_nm1;
      float      z_nm1;
      float      v_nm1;
      float      accum;
      float     accum2;
      float Ap;

} pll_state_def;
/*-- Function Prototypes-------------------------------------------------------------------------------------*/

pll_state_def*pll_init( float f0, float T, float K, float damp_fact, float
 ↔    mult);

void pll(pll_state_def*s, constfloat x_in[], float y_out[]);

#endif/* _pll_h_ */
```

```
/*===========================================================================
 * pll.c
 *          Code to implement a simple frequency locked loop on the DSP.
 *===========================================================================*/

#include<std.h>
#include<sys.h>
#include<dev.h>
#include<sio.h>
#include<math.h>

#include"dsp_ap.h"
#include"pll.h"

#include"sin_tables.h"

/* Mathematical constants */
#define M_PI        3.14159265358979

/*---------------------------------------------------------------------------
 * pll_init
 *          Initializes state of the PLL.
 * Inputs:
 *          f0          Nominal center frequency of PLL (discrete)
 *          T           Time constant of loop filter
 *          k           Gain factor
 *          damp        Damping factor (1.0=critically damped)
 *          mult        Frequency multiplier on output
 * Notes:
 *          The multiplier operation is not implemented. You will have to
 *          consider how to do this!
 *---------------------------------------------------------------------------*/
pll_state_def*pll_init( float f0, float T, float K, float damp, float mult)
{
      pll_state_def*s;
      float wn, tau1, tau2;

      /* Allocate a new pll_state_def structure. Holds state and parameters. */
      if ((s=(pll_state_def*)MEM_calloc(PLL_SEG_ID,                sizeof(pll_state_def),
       ↔    PLL_BUFFER_ALIGN))==NULL)
      {
            SYS_error("Unable to create state structure for PLL.", SYS_EUSER,0);

            return(0);
      }

      /* Copy input parameters */
      s->f0=f0;
      s->damp_fact=damp;
      s->K=K;
      s->mult=mult;
      float Tp=1.0;
      /* Compute the filter coefficients */
      /* Add your code here !!! */

      wn=(2*M_PI)/(T);
```

```
54          tau1=(s->K)/(wn*wn);
55          tau2=2*(s->damp_fact)/(wn)-1/(s->K); s->a1=-
56          (Tp-2*(tau1))/(Tp+2*(tau1));                    s-
            >b0=(Tp+2*(tau2))/(Tp+2*(tau1));
57          s->b1=(Tp-2*(tau2))/(Tp+2*(tau1));
58
59          /* Set state variables (initially all 0) */ s->z_nm1=0;
            s->v_nm1=0;
60          s->x_nm1=0;
61          s->y_nm1=0;
            s->accum=0;
62          s->accum2=0;
63          /* Set initial block amplitude (cannot be 0!) */ s->Ap=1.0;
64
65          return(s);
66
67
68
69
70
71
72  }
73
74  /*-------------------------------------------------------------------------
75    * pll
76    *          PLL process function.
77    *-------------------------------------------------------------------------*/
78  void pll(pll_state_def*s, constfloat x_in[], float y_out[])
79  {
80          int n;
81          float A;
82
83          float x_n;
84          float z_n;
85          float v_n;
86          float y_n;
87          float y_n2;
88
89          /* Add other temporary variables as needed. */
90          /* Do not put any arrays as local variables! */
91
92
93          /*
94            * If signal level is below some threshold, make Ap large, which has the
95            * effect of just doing holdover mode.
96            */
97          if (s->Ap<1.0E-3)
98          {
99                  s->Ap=100.0;
100         }
101
102         A=0.0;/* Variable for computing amplitude */
103         for (n=0; n<BUFFER_SAMPLES; n++)
104         {
105                 /* Input sample (input reference) */
106                 /* Take the sign of the input signal. */
107                 x_n=x_in[n];
108
```

```
109             /* Estimate amplitude from summed |x| */
110             A=A+fabs(x_n);
111
112             /* Add your code here to do PLL operation !!! */
113             /* Code should generate y_n from x_n. */
114
115             z_n=x_n*(s->y_nm1)/s->Ap;
116             v_n=s->a1*s->v_nm1+s->b0*z_n+s->b1*s->z_nm1;
117
118             s->accum=s->accum+s->f0-((s->K)/(2*M_PI))*v_n;
119             s->accum=s->accum-floor(s->accum);
120             y_n=sin_table[(   int)((float)SIN_TABLE_SIZE*s->accum)];
121
122             s->accum2=s->accum2+s->mult*(s->f0-((s->K)/(2*M_PI))*v_n);
123             s->accum2=s->accum2-floor(s->accum2);
124             y_n2=sin_table[(   int)((float)SIN_TABLE_SIZE*s->accum2)];
125
126             /* Put output sample */
127             y_out[n]=y_n2;
128
129             /* Shift current variables to previous values. */
130             s->z_nm1=z_n;
131             s->v_nm1=v_n;
132             s->x_nm1=x_n;
133             s->y_nm1=y_n;
134         }
135
136         /* Get amplitude estimate for next block (compute from A) */
137         s->Ap=(A/n)*(M_PI/2);
138         /*I initially used this:
139          *        s->Ap = A/(BUFFER_SAMPLES/(2/(M_PI)));
140          *        It did not work
141          */
142     }
```

2.2 Problem 2

- A description of how you tested your PLL. Indicate at what frequencies your PLL is no longer able to track.

The input to test the PLL was set to be inline. Then, using the Soundcard pro- gram, the relation between the phase of the input to output was displayed. The PLL tracked the input signal for frequency of about 800Hz. Input was sent to the left channel and output to the right channel. The input and output were sent to the scope screen and consequently, it was observed that the PLL was tracking based on the three criteria to the reference frequency (unit, half and double) frequencies. I also changed the position of the switch at unit, half and double the frequency and saw that for all the three conditions, the PLL was tracking. Generally, the PLL is supposed to handle changes in frequency of at least 10% of the nominal refe- rence frequency of 800Hz. For frequencies below 400Hz and beyond 1500Hz, the

$\pm$

pll stopped tracking.

2.3 Problem 3

- A screen shot (like from the Scope program) showing that the PLL is tracking your input signal.

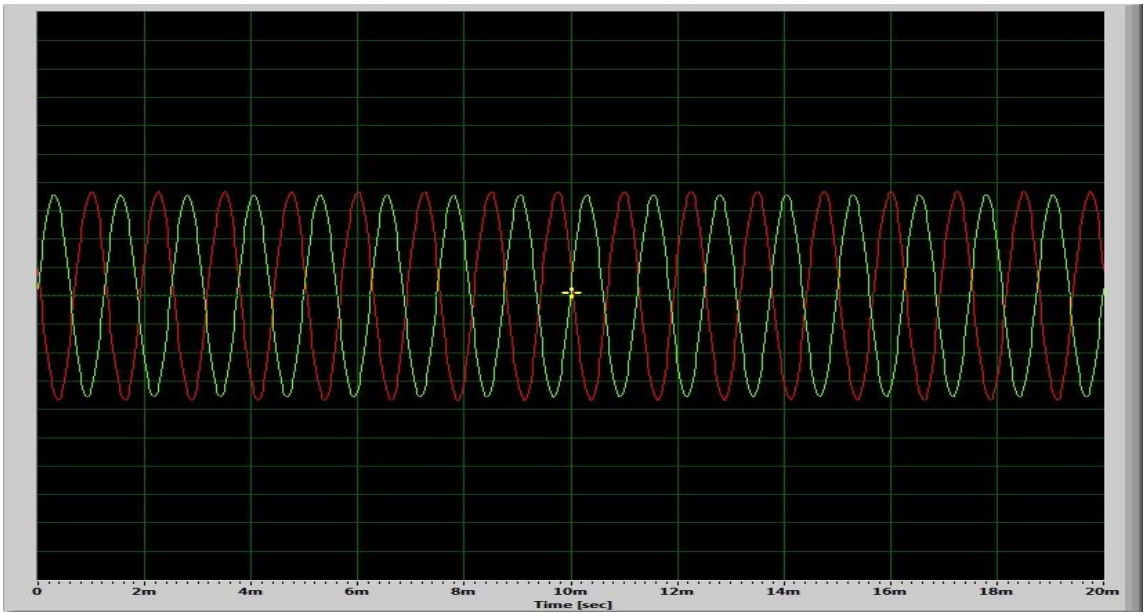

FIGURE 3 – PLL tracking at unit frequency - switch state 0 with

the following frequency spectrum :

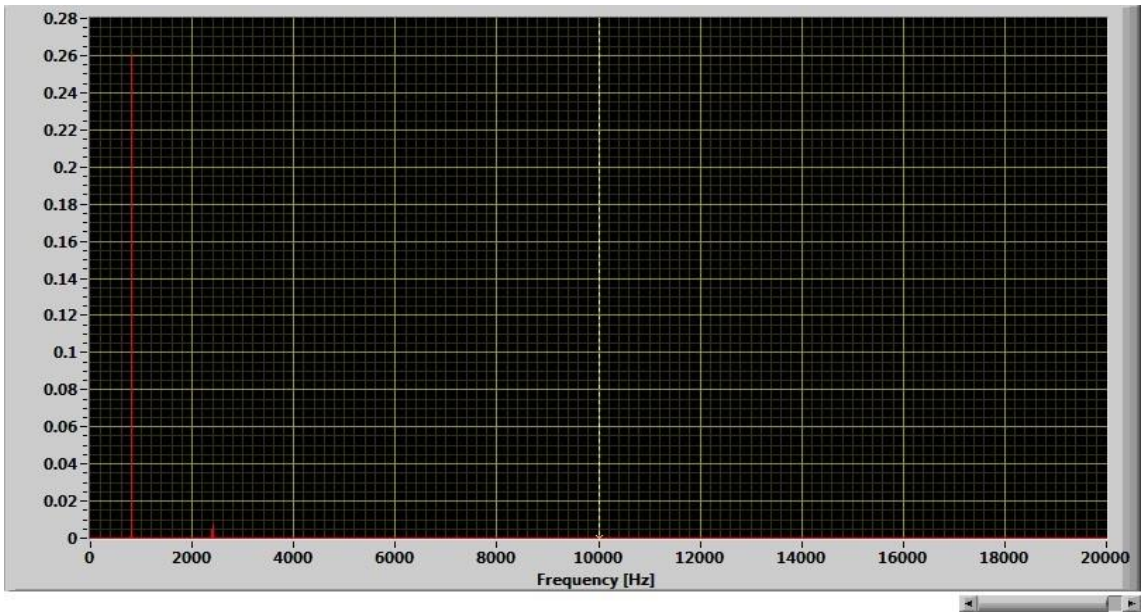

FIGURE 4 – PLL tracking at unit frequency - frequency spectrum

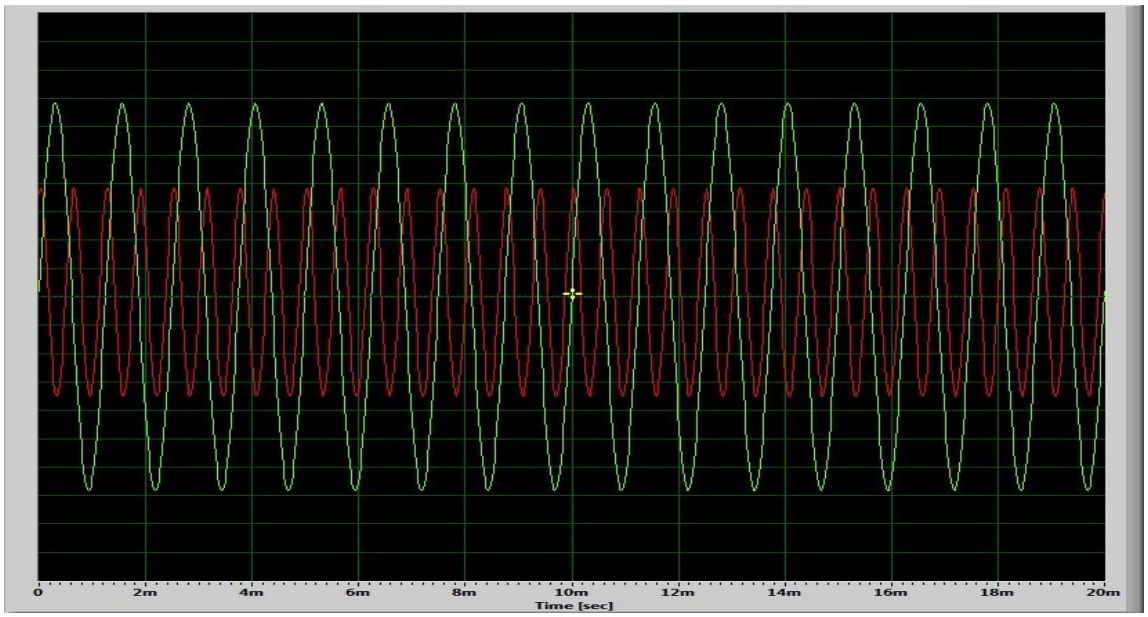

FIGURE 5 – PLL tracking at double frequency - switch state 1 with

the following frequency spectrum :

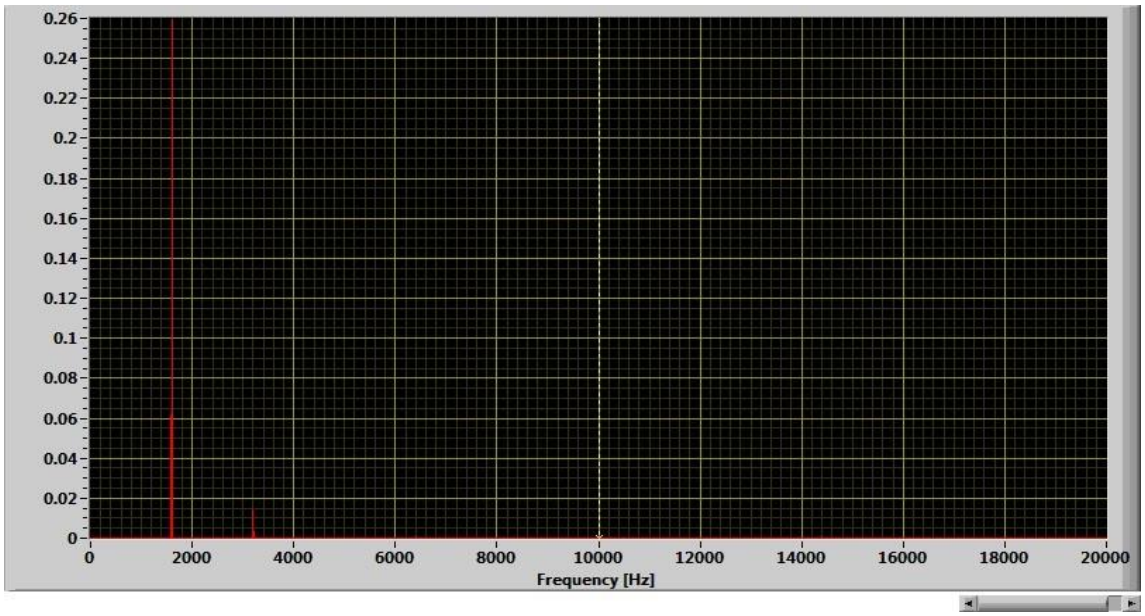

FIGURE 6 – PLL tracking at unit frequency - frequency spectrum

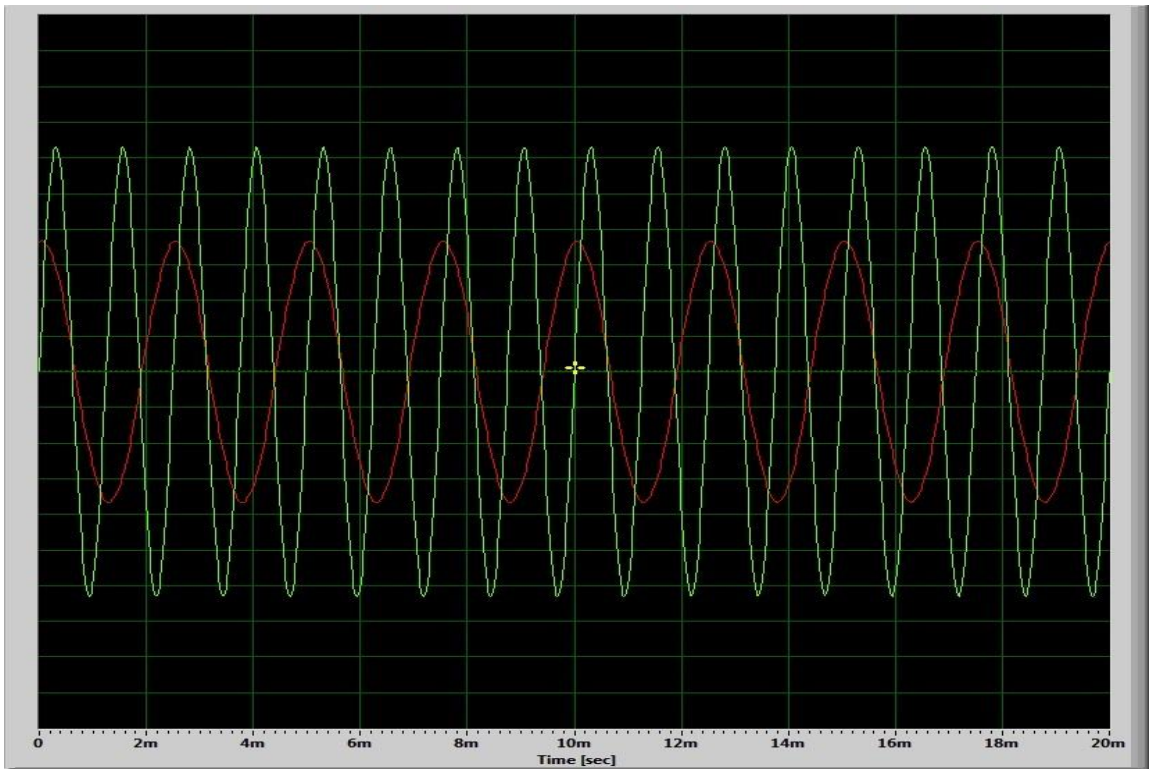

FIGURE 7 – PLL tracking at half frequency - switch state 2 with

the following frequency spectrum :

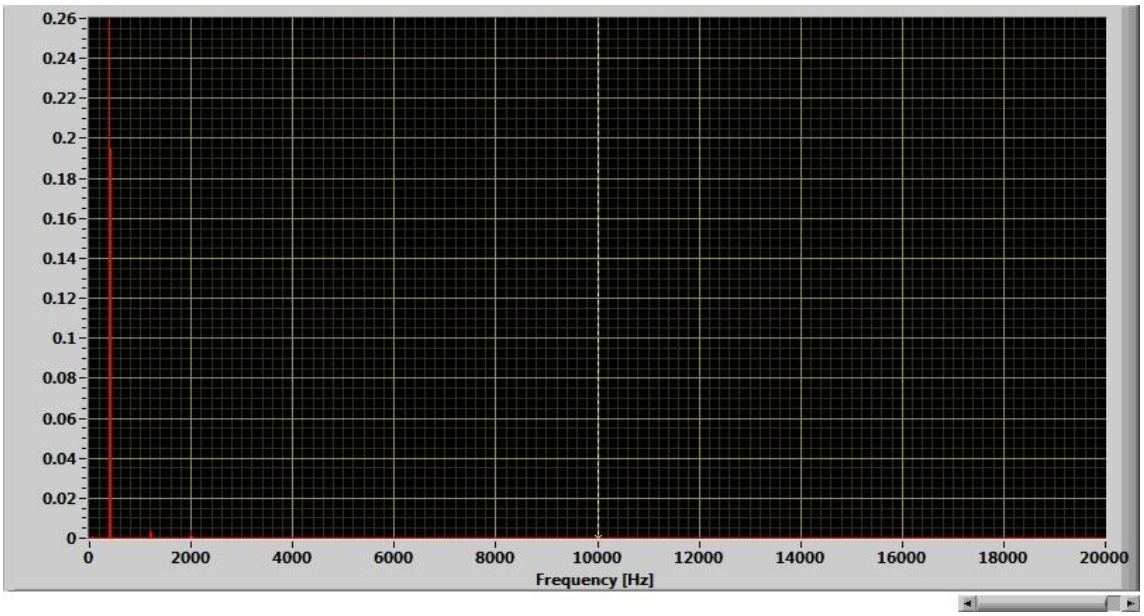

FIGURE 8 – PLL tracking at unit frequency - frequency spectrum

2.4 Problem 4

- An explanation of any problems you ran into in the design and coding and how you solved them

Writing the code for the PLL implementation was a bit tedious even though we had the Matlab implementation. Setting up the input from line in (computer) was something I missed that messed the output. It was not stable in tracking. With the help of the instructor and the TA's, I managed to get the right output.

3 Conclusion

PLL is a feedback system that tracks the phase of sinusoid after locking. It is a negative feedback system which may become unstable. Lookup tables are effi- cient in implementing PLL in matlab and C when calling sinusoids. Accumulators were used to keep track of the phase. In each time step, it tells the corresponding position in the sin() lookup table. The state of the PLL is stored and restored after every block. The PLL was able to handle changes in frequency of at least 10% of the nominal reference frequency of $800Hz$. $\pm$

4 References

[1] http://dsp-fhu.user.jacobs-university.de/?page_id=231

[2] Teodore S. Rappaport, Wireless Communications : Principles and Practice (2nd Edition)