Phase Lock Loop for Single-Phase Signals and Moving Average Filter Design

An Application with dSPACE-SCALEXIO

davide bagnara

August 31, 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this document the following topics are slightly covered:

- phase locked loop for single phase signals;
- moving average filter;
- discrete Fourier transform;
- Simulink C-caller;
- dSPACE SCALEXIO.

The idea of the document is to propose an laboratory application of a single-phase *PLL*, and moving average filter implemented in a fast prototyping equipment. The moving average filter will be implemented using customized C-code and Simulink C-caller.

1.1 Nomenclature

Here a list of symbols, variables, and parameters used along the document:

- *PLL*: phase locked loop;
- *VCO*: voltage controlled oscillator;
- *PI*: proportional integral controller;
- *LF*: loop filter;
- *PD*: phase detector;
- *SOGI*: second order generalized integrator;
- *QSG*: quadrature signal generator;
- *MAVG*: moving average;
- RMB: right mouse button;
- LMB: left mouse button;
- HMI: human machine interface;
- ConfigurationDesk: dSPACE application used to configure a project with the scalexio equipment;
- ControlDesk: dSPACE application used as HMI;

- $v, \ v_{signal} \quad \left[\mathrm{V}\right]$: input signals;
- $\alpha\beta$: direct and quadrature components of a vector quantity, generally with respect to a stationary reference frame;
- $\xi\eta$: additional direct and quadrature components of a vector quantity, generally with respect to a rotating reference frame;

Chapter 2

Single-Phase PLL

2.1 Basic Structure of a Phase-Locked Loop

The basic structure of the phase-locked loop (*PLL*) is shown in Figure 2.1. It consists of three fundamental blocks:

- *phase detector* (*PD*). This block generates an output signal proportional to the phase difference between the input signal, v_{signal}, and the signal generated by the internal oscillator of the *PLL*, v^{pll}. Depending on the type of *PD*, high frequency AC components appear together with the DC phase-angle difference signal.
- *loop filter* (*LF*). This block presents a low pass filtering characteristic to attenuate the high frequency AC components from the PD output. Typically, this block is constituted by a first order low pass filter or a PI controller.
- *voltage controlled oscillator* (*VCO*). This block generates at its output an AC signal whose frequency is shifted with respect to a given central frequency, ω_{ff}, as a function of the input voltage provided by the *LF*.

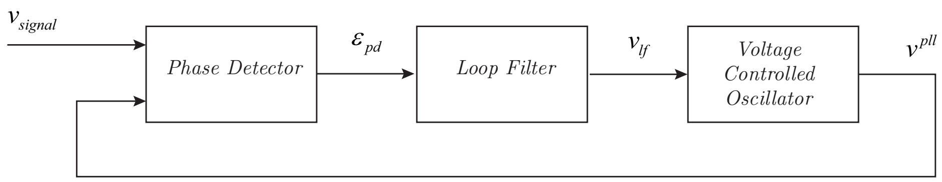


Figure 2.1: Basic structure of a PLL.

The block diagram of an elementary *PLL* is shown in Figure 2.2. In this case *PD* is implemented by means of the *Superheterodyne* technique, the *LF* is based on a *PI* controller and the *VCO* consists of a sinusoidal function supplied by a linear integrator.

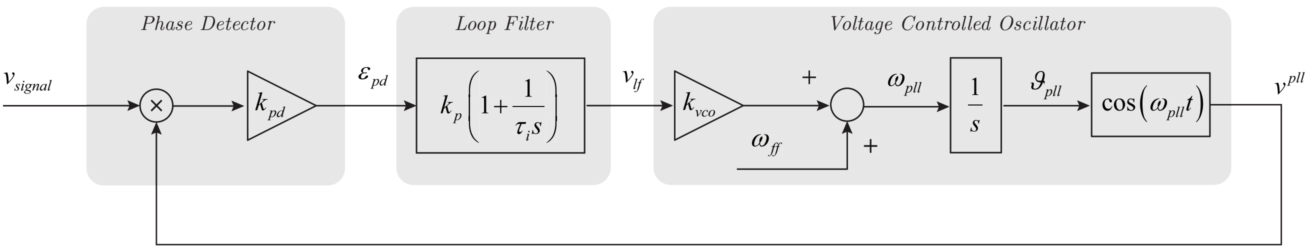


Figure 2.2: Block diagram of an elementary *PLL*.

If the input signal applied to this system is given by

$$v = V\sin(\vartheta) = V\sin(\omega t + \phi) \tag{2.1.1}$$

and the signal generated by the *VCO* is given by

$$v^{pll} = \cos(\vartheta_{pll}) = \cos(\omega_{pll}t + \phi_{pll}) \tag{2.1.2}$$

the phase error signal from the multiplier *PD* output can be written as

$$\begin{aligned}\varepsilon_{pd} &= Vk_{pd}\sin(\omega t + \phi)\cos(\omega_{pll}t + \phi_{pll}) \\ &= \frac{Vk_{pd}}{2}\Big[\sin[(\omega - \omega_{pll})t + (\phi - \phi_{pll})] + \sin[(\omega + \omega_{pll})t + (\phi + \phi_{pll})]\Big]\end{aligned} \tag{2.1.3}$$

The high frequency components $(\omega + \omega_{pll})$ of the *PD* error signal will be cancelled out by the *LF*, only the low frequency term $(\omega - \omega_{pll})$ will be processed, therefore, the *PD* error signal to be considered is

$$\varepsilon_{pd} = \frac{Vk_{pd}}{2}\sin[(\omega - \omega_{pll})t + (\phi - \phi_{pll})] \tag{2.1.4}$$

If it is assumed that the *VCO* is well tuned to the input frequency, i.e. with $\omega \approx \omega_{pll}$, the DC term of the phase error is given as follows

$$\varepsilon_{pd} = \frac{Vk_{pd}}{2}\sin(\phi - \phi_{pll}) \tag{2.1.5}$$

It can be observed in (2.1.5) that the multiplier *PD* produces nonlinear phase detection because of the sinusoidal function. However, when phase error is very small, i.e. when $\phi \approx \phi_{pll}$, the output of the multiplier *PD* can be linearized in the vicinity of such an operating point since $\sin(\phi - \phi_{pll}) \approx \sin(\vartheta - \vartheta_{pll}) \approx (\vartheta - \vartheta_{pll})$. Therefore, once the *PLL* is locked, the relevant term of the phase error signal is given by

$$\varepsilon_{pd} = \frac{Vk_{pd}}{2}(\vartheta - \vartheta pll) \tag{2.1.6}$$

According to Eq. (2.1.6), the model presented in Figure 2.2 can be linearized around the condition of $\omega \approx \omega_{pll}$ resulting as per Figure 2.3.

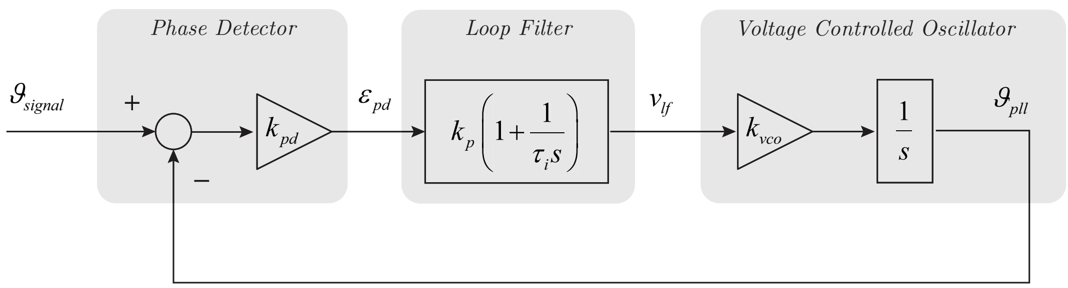


Figure 2.3: Small signal model of an elementary *PLL*.

According to the block diagram of Figure 2.3 a frequency domain analysis brings to the following transfer functions (consider $k_{pd} = k_{vco} = 1$):

$$H(s) = PD(s)LF(s)VCO(s) = \frac{k_p s + \frac{k_p}{\tau_i}}{s^2} \tag{2.1.7}$$

$$H_\vartheta(s) = \frac{\Theta_{pll}(s)}{\Theta(s)} = \frac{H(s)}{1 + H(s)} = \frac{k_p s + \frac{k_p}{\tau_i}}{s^2 + k_p s + \frac{k_p}{\tau_i}} \tag{2.1.8}$$

$$E_\vartheta = \frac{E_{pd}(s)}{\Theta(s)} = 1 - H_\vartheta(s) = \frac{s^2}{s^2 + k_p s + \frac{k_p}{\tau_i}} \tag{2.1.9}$$

The open loop transfer function of Eq. (2.1.7) shows that the *PLL* has two poles at the origin, which means that is able to track even a constant slope ramp in the input phase angle without any steady state error.

2.2 SOGI-QSG-based PLL

Figure 2.4 shows the structure of a *VCO* based on *QSG*; the structure consists of an adaptive filter.

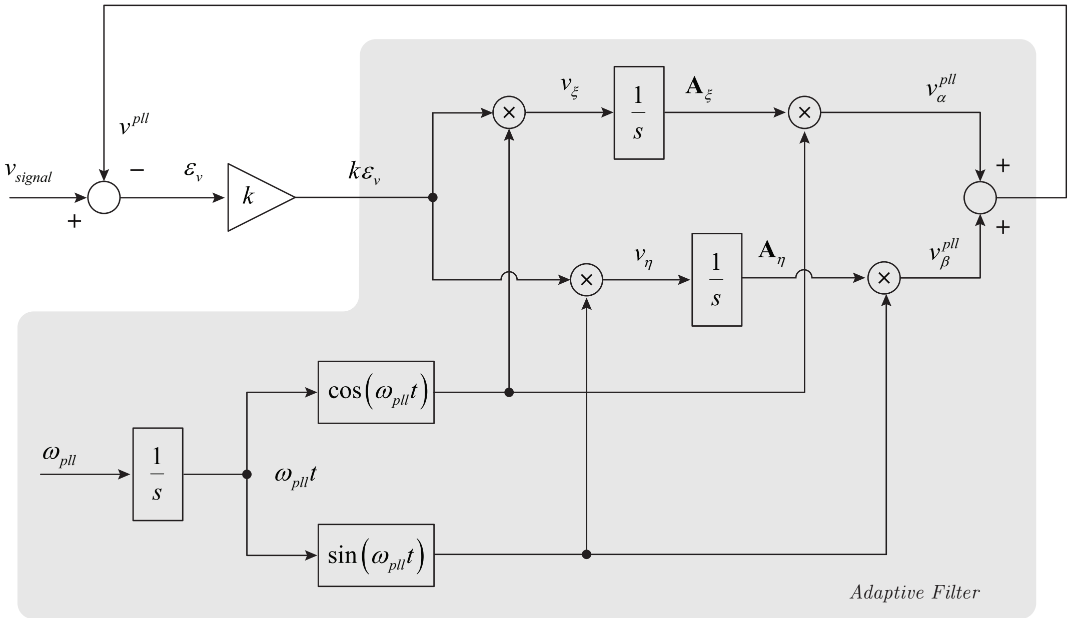


Figure 2.4: Voltage controlled oscillator based on a adaptive filter.

Defining $g = k\epsilon_v$, the v_ξ, and v_η components of Figure 2.4 can be written as follows

$$v_\xi = g\cos(\omega_{pll}) = \frac{1}{2}g\Big(e^{j\omega_{pll}t} + e^{-j\omega_{pll}t}\Big) \tag{2.2.1}$$

$$v_\eta = g\cos(\omega_{pll}) = -j\frac{1}{2}g\Big(e^{j\omega_{pll}t} - e^{-j\omega_{pll}t}\Big) \tag{2.2.2}$$

The $\mathbf{A}_\xi$, and $\mathbf{A}_\eta$ terms which correspond to the output of the integrators for v_ξ and v_η, can be expressed in the Laplace domain as follows

$$\mathbf{A}_\xi = \frac{1}{s}v_\xi(s) = \frac{1}{2s}\Big[g(s + j\omega_{pll}t) + g(s - j\omega_{pll}t)\Big] \tag{2.2.3}$$

$$\mathbf{A}_\eta = \frac{1}{s}v_\eta(s) = -j\frac{1}{2s}\Big[g(s + j\omega_{pll}t) - g(s - j\omega_{pll}t)\Big] \tag{2.2.4}$$

and the v_α^{pll}, v_β^{pll} terms result as follows

$$\begin{aligned} v_\alpha^{pll} &= \frac{1}{2}\Big[\mathbf{A}_\xi(s + j\omega_{pll}t) + \mathbf{A}_\xi(s - j\omega_{pll}t)\Big] \\ &= \frac{1}{4(s + j\omega_{pll})}\Big[g(s) + g(s + j\omega_{pll}t)\Big] + \frac{1}{4(s - j\omega_{pll})}\Big[g(s) + g(s - j\omega_{pll}t)\Big] \end{aligned} \tag{2.2.5}$$

$$\begin{aligned} v_\beta^{pll} &= -j\frac{1}{2}\Big[\mathbf{A}_\xi(s+j\omega_{pll}t) - \mathbf{A}_\xi(s-j\omega_{pll}t)\Big] \\ &= \frac{1}{4(s+j\omega_{pll})}\Big[g(s)+g(s+2j\omega_{pll}t)\Big] + \frac{1}{4(s-j\omega_{pll})}\Big[g(s)-g(s-2j\omega_{pll}t)\Big] \end{aligned} \tag{2.2.6}$$

The term $v^{pll} = v_\alpha^{pll} + v_\beta^{pll}$ results as follows

$$v^{pll} = v_\alpha^{pll} + v_\beta^{pll} = \frac{s}{s^2+\omega_{pll}^2} g(s) \tag{2.2.7}$$

Consequently, the transfer functions of the adaptive filter VCO structure of Figure 2.4 are given by

$$\frac{v^{pll}}{\varepsilon_v}(s) = \frac{ks}{s^2+\omega_{pll}^2} \tag{2.2.8}$$

$$\frac{v^{pll}}{v}(s) = \frac{ks}{s^2+ks+\omega_{pll}^2} \tag{2.2.9}$$

$$\frac{\varepsilon_v}{v}(s) = \frac{s^2+\omega_{pll}^2}{s^2+ks+\omega_{pll}^2} \tag{2.2.10}$$

The structure of Figure 2.4 can be used of quadrature signal generator (QSG) by adding a scaled integrator at the output of the adaptive filter, as in Figure 2.5 is shown.

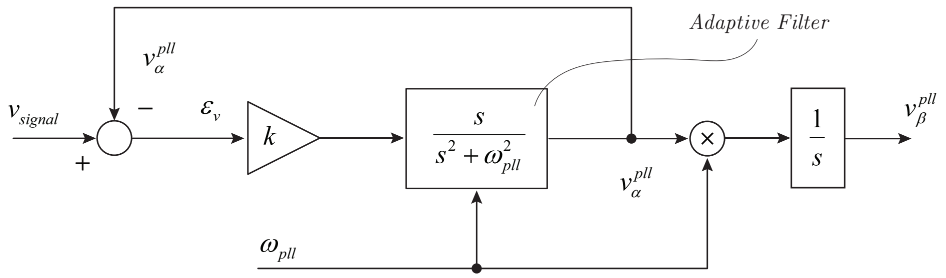


Figure 2.5: Quadrature signal generator based on a adaptive filter.

Clearly, the response of the AF block, shown in Figure 2.4 is defined by Eq. (2.2.8) in the case of applying a likewise sinusoidal signal (sine or cosine) with frequency ω_{pll} to its input.

Recalling that

$$\mathscr{L}\Big[\sin(\omega_{pll})\Big] = \frac{\omega_{pll}}{s^2+\omega_{pll}^2} \tag{2.2.11}$$

$$\mathscr{L}\Big[\cos(\omega_{pll})\Big] = \frac{s}{s^2+\omega_{pll}^2} \tag{2.2.12}$$

the time response of the system characterized by Eq. (2.2.8) in the presence of sinusoidal inputs is given by

$$\mathscr{L}^{-1}\left[\frac{\omega_{pll}}{s^2+\omega_{pll}^2}\frac{s}{s^2+\omega_{pll}^2}\right] = \frac{1}{2}t\sin(\omega_{pll}t) \tag{2.2.13}$$

and

$$\mathscr{L}^{-1}\left[\frac{s}{s^2+\omega_{pll}^2}\frac{s}{s^2+\omega_{pll}^2}\right] = \frac{1}{2}\left[\frac{\sin(\omega_{pll}t)}{\omega_{pll}} + t\cos(\omega_{pll}t)\right] \approx \frac{1}{2}t\cos(\omega_{pll}t) \tag{2.2.14}$$

An efficient implementation of the structure of Figure 2.5 is shown in Figure 2.6, where

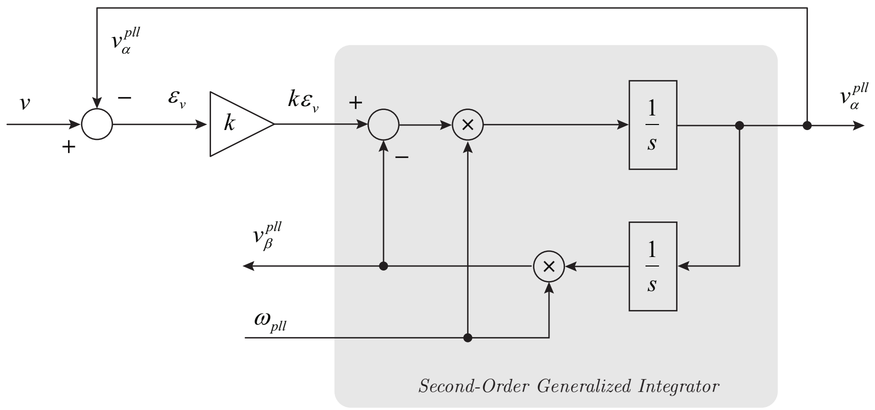


Figure 2.6: Second order adaptive filter based on an second order generalized integrator and a quadrature signal generator (*SOGI-QSG*).

$$SOGI(s) = \frac{v_\alpha^{pll}}{\varepsilon_v}(s) = \frac{k\,\omega_{pll}s}{s^2 + \omega_{pll}^2} \tag{2.2.15}$$

$$D(s) = \frac{v_\alpha^{pll}}{v}(s) = \frac{k\,\omega_{pll}s}{s^2 + k\,\omega_{pll}s + \omega_{pll}^2} \tag{2.2.16}$$

$$Q(s) = \frac{v_\beta^{pll}}{v}(s) = \frac{k\,\omega_{pll}^2}{s^2 + k\,\omega_{pll}s + \omega_{pll}^2} \tag{2.2.17}$$

Based on the structure of the *SOGI-QSG* of Figure 2.6 it is possible to implement a *SOGI-QSG*-based *PLL* as shown in Figure 2.7.

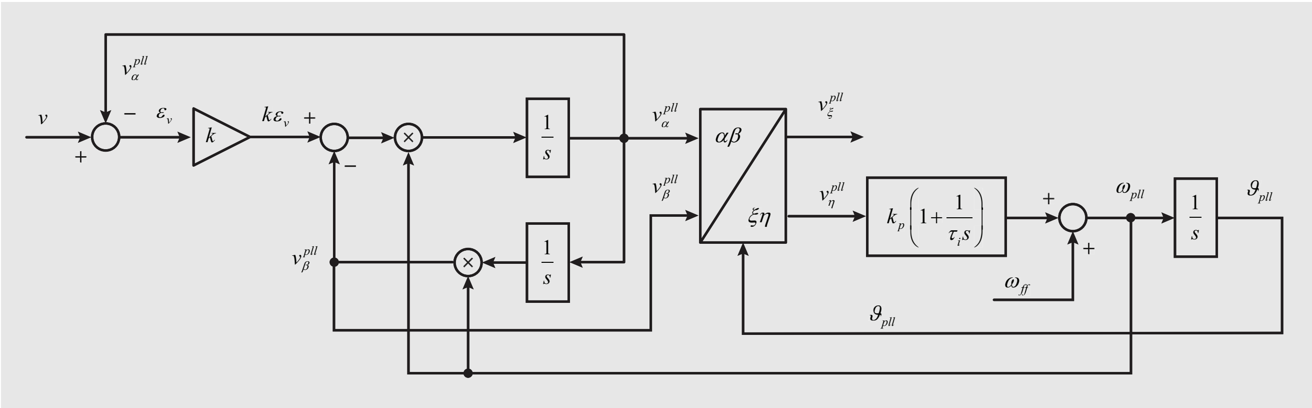
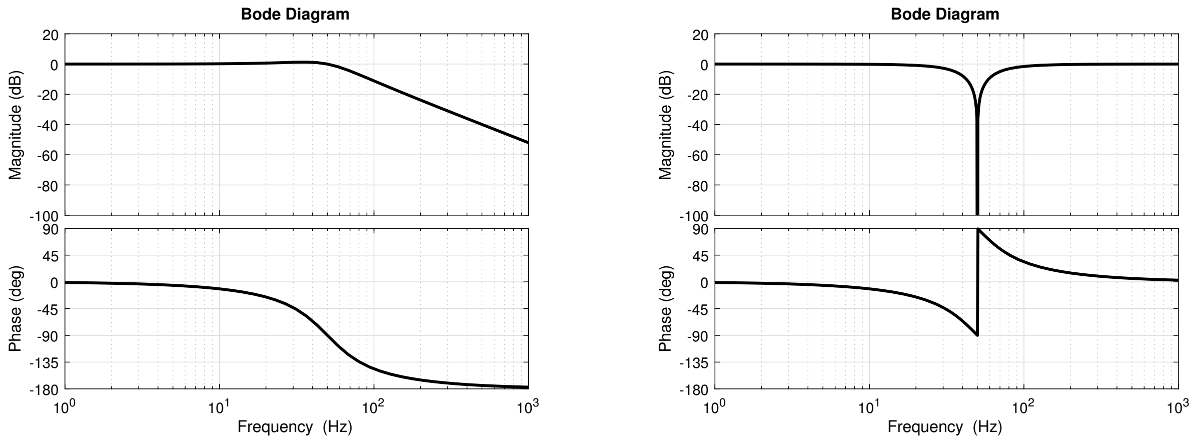


Figure 2.7: Diagram of the *SOGI*-based *PLL* (*SOGI-QSG*).

The transfer function from the input signal v to the error signal ε_v is given by

$$E(s) = \frac{\varepsilon_v}{v}(s) = \frac{s^2 + \omega_{pll}^2}{s^2 + k\,\omega_{pll}s + \omega_{pll}^2} \tag{2.2.18}$$

The transfer function of Eq. (2.2.18) responds to a second order notch filter, with zero gain at the centre frequency (ω_{pll}).



(a) Bode diagram of the $Q(s)$ transfer function in a *SOGI-QSG*.

(b) Bode diagram of the $E(s)$ transfer function in a *SOGI-QSG*.

Figure 2.8: Bode diagram of the *SOGI-QSG* response.

Another useful possible implementation of the *SOGI-QSG*-based *PLL* is based on the direct and inverse Park transforms, as shown in Figure 2.9

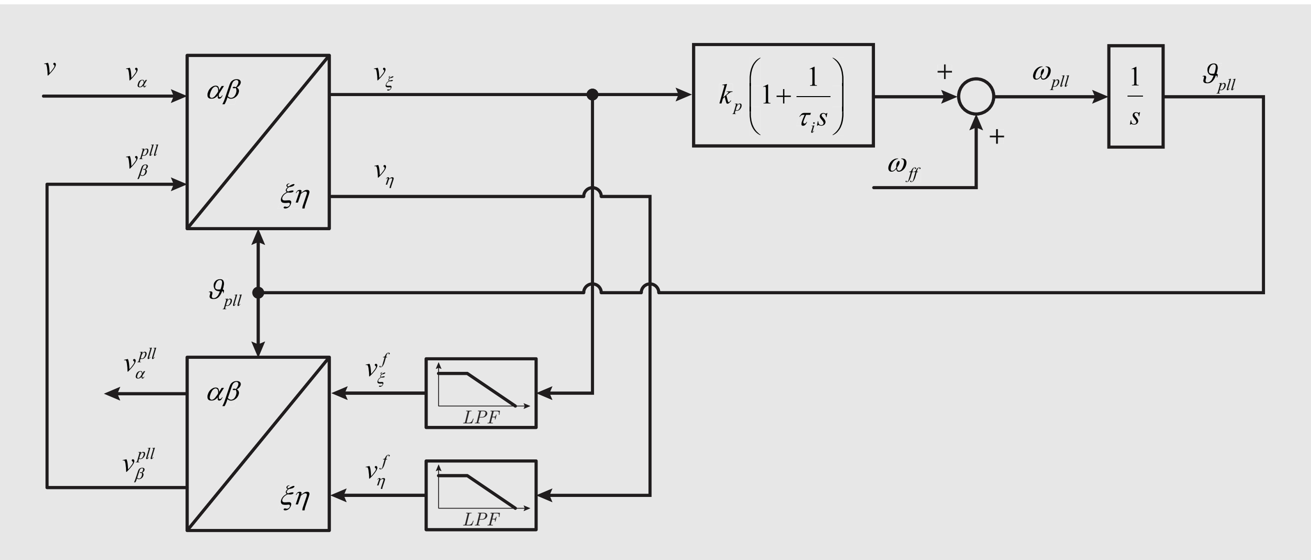


Figure 2.9: PLL based on the on direct and inverse Park transform.

An equivalent transfer function block diagram is presented in Figure 2.10. In this configuration the transformation $v_\alpha \rightarrow v_\beta^{pll}$ is represented as ω_{pll}/s in Laplace domain. An intuitive explanation of its operation principle can be given if it is assumed that the *PLL* is well tuned to the input signal frequency. Under such operation conditions, if v_α and v_β^{pll} are not in quadrature, the virtual input vector, v_α^{pll}, resulting from these signals will have neither constant amplitude nor rotation speed.

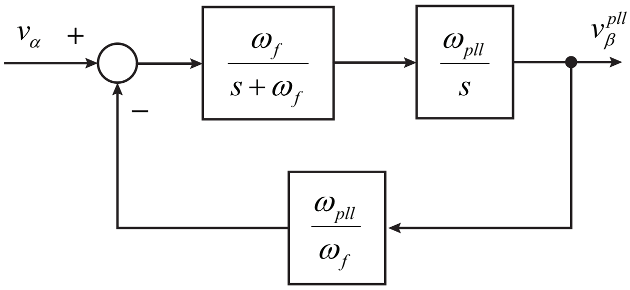


Figure 2.10: Equivalent block diagram of the PLL based on the on direct and inverse Park transform.

Therefore v_ξ and v_η waveform resulting from direct Park transformation will have harmonics components. These harmonics will be suppressed by the *LPF* blocks generating, as well, the components v_ξ^f and v_η^f. The v_α and v_β^{pll} components resulting from the inverse Park transformation of the components v_ξ^f and v_η^f will be in quadrature, though v_α and v_α^{pll} will not be in phase if the *PLL* is not perfectly synchronized. As the *PLL* locks the phase angle of the input signal v_α, the components v_α^{pll} and v_β^{pll} will be respectively in phase and in quadrature respect to the input signal v_α.

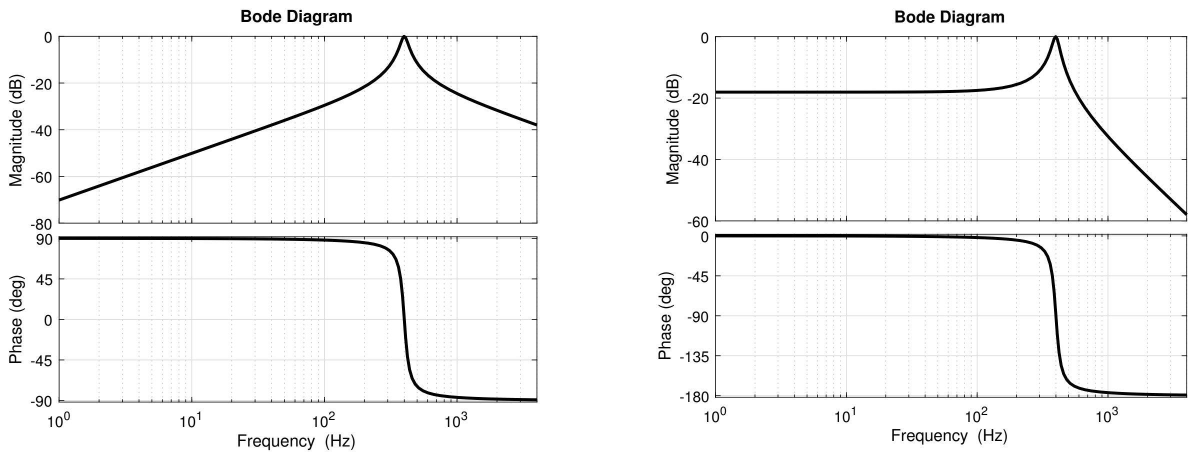
From the equivalent block diagram of Figure 2.10 the following transfer functions can be derived

$$\boxed{\frac{V_\beta^{pll}}{V_\alpha}(s) = \frac{\omega_f \omega_{pll}}{s^2 + \omega_f s + \omega_{pll}^2}} \tag{2.2.19}$$

$$\boxed{\frac{V_\alpha^{pll}}{V_\alpha}(s) = \frac{s\omega_f}{s^2 + \omega_f s + \omega_{pll}^2}} \tag{2.2.20}$$

where the relation of Eq. (2.2.21) has been applied.

$$V_\beta^{pll}(s) = \frac{\omega_{pll}}{s} V_\alpha^{pll}(s) \tag{2.2.21}$$



(a) Bode diagram of the $\frac{V_\alpha^{pll}}{V_\alpha}(s)$ transfer function, with $\omega_f = 2\pi 50\,\text{Hz}$, and $\omega_{pll} = 2\pi 400\,\text{Hz}$.

(b) Bode diagram of the $\frac{V_\beta^{pll}}{V_\alpha}(s)$ transfer function, with $\omega_f = 2\pi 50\,\text{Hz}$, and $\omega_{pll} = 2\pi 400\,\text{Hz}$.

Figure 2.11: Bode diagram of the *QSG-PLL* based on direct and inverse Parck transform.

Chapter 3

Moving Average Filter

3.1 Basic Idea of the Moving Average Filter

The moving average (MAVG) filter is used, in this contest, to calculate the average value of a periodic waveform. The basic consist of to sample the whole period of the waveform and calculate the mean value according to

$$x_m(k) = \frac{1}{N}\sum_{i=0}^{N-1} x(k-i) \tag{3.1.1}$$

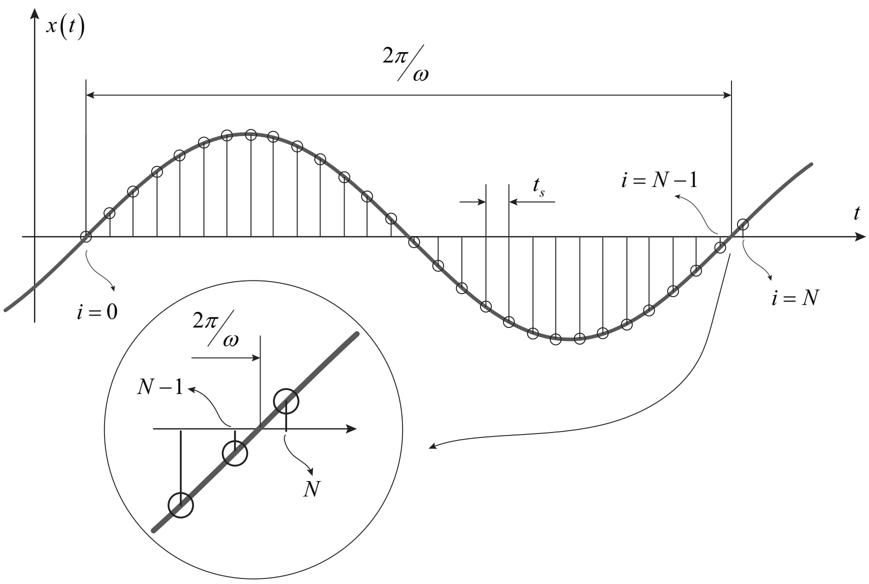


Figure 3.1: Principle of the moving average filter.

Representing Eq. (3.1.1) with the $\mathscr{Z}$-transform, it results

$$X_m(z) = \frac{1}{N}\mathscr{Z}\left[\sum_{i=0}^{N-1} x(k-i)\right] = \frac{1}{N}X(z)\sum_{i=0}^{N-1} z^{-i} = \frac{1}{N}X(z)\frac{1-z^{-N}}{1-z^{-1}} \tag{3.1.2}$$

The term $X_s(z) = X(z)\frac{1-z^{-N}}{1-z^{-1}}$ represents the summation of the $x(k)$ samples along a window period of N-samples. the mean value of the $x(k)$ signal results, in the $\mathscr{Z}$-transform domain,

$$X_m(z) = \frac{1}{N}X_s(z) \tag{3.1.3}$$

the representation of the term $X_s(z) = X(z)\frac{1-z^{-N}}{1-z^{-1}}$ into the discrete time domain results as follows

$$x_s(k) = x_s(k-1) + x(k) - x(k-N) \tag{3.1.4}$$

and the mean value of the $x(k)$ signal results, along a window period of N-samples,

$$x_m(k) = \frac{1}{N} x_s(k) \tag{3.1.5}$$

From an implementation point of view Eq. (3.1.4) shall be divided into two equations:

$$x_s(k) = x_s(k-1) + x(k) \qquad \text{until the buffer of } N\text{-elements is not yet loaded} \tag{3.1.6}$$

$$x_s(k) = x_s(k-1) + x(k) - x(k-N) \qquad \text{when the buffer of } N\text{-elements is already loaded} \tag{3.1.7}$$

Define ω the pulsation of input signal $x(t)$ and t_s the sampling time of the *MAVG* filter. An important aspect concerning the performance of the *MAVG* filter lays when an integer number of sampling time t_s doesn't cover perfectly the period $(2\pi/\omega)$ of the input signal. In this case exist a technique which is able to overcome the error occurred when the buffer doesn't cover perfectly the period of the input signal. The algorithm consists of to implement two *MAVG* filter with different buffer windows, respectively with*

$$N_{\text{inf}} = \text{int}\left(\frac{2\pi}{\omega t_s}\right) \tag{3.1.8}$$

$$N_{\text{sup}} = \text{int}\left(\frac{2\pi}{\omega t_s}\right) + 1 \tag{3.1.9}$$

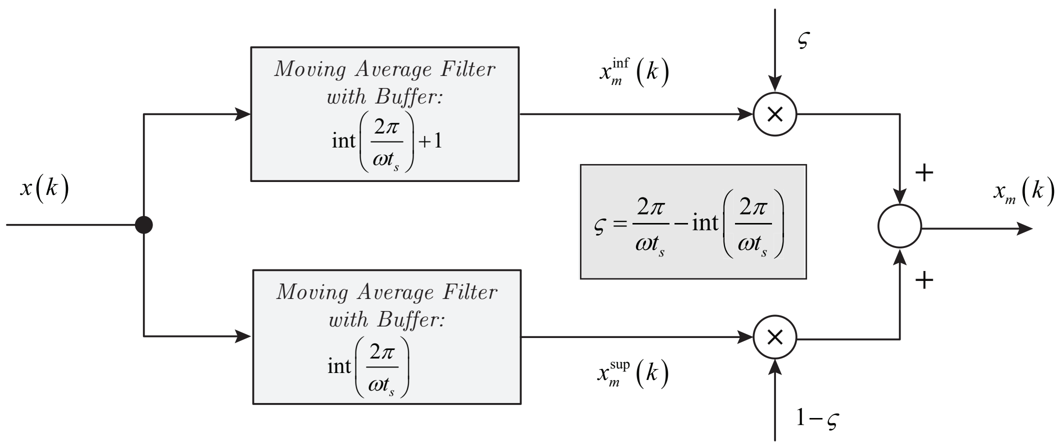


Figure 3.2: Algorithm description for not integer t_s overlapping of the signal period.

the two average quantities will be summed with the relative weight, as follows (see also Figure 3.2)

$$x_m(k) = \zeta x_m^{\text{inf}} + \left(1-\zeta\right) x_m^{\text{sup}} \tag{3.1.10}$$

where

$$\zeta = \frac{2\pi}{\omega t_s} - \text{int}\left(\frac{2\pi}{\omega t_s}\right) \tag{3.1.11}$$

*Consider that the function int() ≡ floor()

3.2 Buffer Management of the Moving Average Filter

An important aspect for the implementation of an efficient *MAVG* filter is the management of the buffer used to calculate the average value of the input signal. According to Figure 3.3 the buffer used for the *MAVG* can be considered a FIFO. To access to the buffer data it is necessary to consider two case:

– the current buffer index i is greater than the offset window (N);

$$i_N = i - d \tag{3.2.1}$$

– the current buffer index i is less than the offset window (N)

$$i_N = i - d + N_{\max} \tag{3.2.2}$$

where $N_{\max}$ is the size of the buffer.

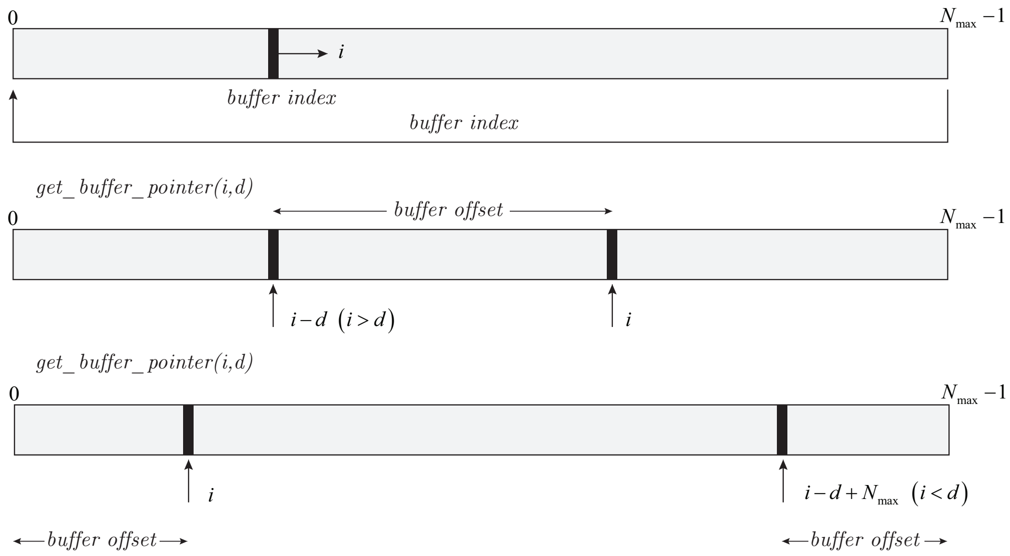


Figure 3.3: Buffer management of the moving average filter.

The buffer size shall be selected according to the maximum period of the input signal

$$N_{\max} = \frac{2\pi}{\omega_{\min} t_s} \tag{3.2.3}$$

3.3 C-code Implementation

The moving average filter has been implemented in the *Simulink* model as *Custom Code* by the *C-Caller*†.

The code structure is composed by four files:

– `mavgflt_simulink.h`: the file structure data for the structure for the interface between the *Simulink* model and the inner custom code.

– `mavgflt.h`: the file include structure data and header functions for the *MAVG* filter

†C-Caller is used to integrate user C-code in Simulink

- `mavgflt_simulink.c`: the file include the source code for the interface between the *Simulink* model and the inner custom code.
- `mavgflt.c`: the file include source code functions for the *MAVG* filter

mavgflt_simulink.h

```
#ifndef   _MAVGFLT_SIMULINK_
#define   _MAVGFLT_SIMULINK_
#define    NMAVGFLT_INSTANCES  16

typedef struct mavgflt_output_s {
 float ts;                   /*sampling time*/
 float sshort;               /*sum short (N-1 samples)*/
 float slong;                /*sum long (N samples)*/
 float pinput;               /*previous input*/
 unsigned int cbpointer;     /*current buffer pointer*/
 unsigned int cbsample;      /*current buffer sample*/
 float mavg_output;          /*moving average filter output*/
} mavgflt_output_t;

#define MAVGFLT_OUTPUT mavgflt_output_t

MAVGFLT_OUTPUT mavgflt_process_simulink(const float input, const float period, const float ts,
unsigned char reset, unsigned char instance);
#endif
```

mavgflt.h

```
#ifndef _MAVGFLT_
#define _MAVGFLT_
#define MAVGFLT_SIZE        152
#define MAVGFLT_SIZE_MIN    78
#define MAVGFLT_SIZE_MAX    420

typedef struct mavgflt_s {
 float ts;                          /*Filter time base.*/
 float sshort;                      /*Integration with floor period length.*/
 float slong;                       /*Integration with ceil period length.*/
 float pinput;                      /*step back input value*/
 float buffer[MAVGFLT_SIZE_MAX];    /*buffer*/
 unsigned int cbpointer;            /*buffer array pointer for the newest element to be written*/
 unsigned int cbsample;             /*current number of samples for the current period*/
} mavgflt_t;

#define MAVGFLT mavgflt_t

void mavgflt_init(volatile MAVGFLT *f);
void mavgflt_ts(volatile MAVGFLT *f, volatile float ts);
void mavgflt_reset(volatile MAVGFLT *f);
float mavgflt_process(volatile MAVGFLT *f, float input, const float period);
#endif
```

mavgflt_simulink.c

```
#include <include/mavgflt.h>
#include <include/mavgflt_simulink.h>
static void init_allmavgflt_instances(MAVGFLT *const filter_list, const unsigned int filter_num)
{
    unsigned int i;
```

```
    for (i = 0; i < filter_num; ++i) {
        MAVGFLT *const filter = &filter_list[i];
        mavgflt_init(filter);
           i++;
    }
}
MAVGFLT_OUTPUT mavgflt_process_simulink(const float input, const float period, const float ts,
unsigned char reset, unsigned char instance)
{
    static MAVGFLT filter_instances[NMAVGFLT_INSTANCES] = {0};
    static unsigned int filter_initialized = 0;
    if (!filter_initialized){
        init_allmavgflt_instances(filter_instances, NMAVGFLT_INSTANCES);
           filter_initialized = 1;
 }
 if (instance < NMAVGFLT_INSTANCES) {
     const MAVGFLT* filter_instance = &filter_instances[instance];
     mavgflt_ts(filter_instance, ts);
     if(reset)
     {
         mavgflt_reset(filter_instance);
     }
     const float output_value = mavgflt_process(filter_instance, input, period);
     const MAVGFLT_OUTPUT filter_output = {
         .ts = filter_instance->ts,
         .sshort = filter_instance->sshort,
         .slong = filter_instance->slong,
         .pinput = filter_instance->pinput,
         .cbpointer = filter_instance->cbpointer,
         .cbsample = filter_instance->cbsample,
         .mavg_output = output_value
     };
     return filter_output;
 }
 else{
     const MAVGFLT_OUTPUT empty_output = {0};
        return empty_output;
    }
}
```

mavgflt.c

```
#include <include/mavgflt.h>
void mavgflt_init(volatile MAVGFLT *f)
{
    f->ts = 0.0;
    mavgflt_reset(f);
}
void mavgflt_ts(volatile MAVGFLT *f, volatile float ts)
{
    f->ts = ts;
}
void mavgflt_reset(volatile MAVGFLT *f)
{
    int i;
    f->pinput = 0.0;
    f->sshort = 0.0;
    f->slong = 0.0;
    f->cbpointer = 0;
    f->cbsample = 0;
    for (i = 0; i < MAVGFLT_SIZE_MAX; i++) {
```

```
        f->buffer[i] = 0.0;
    }
}
static unsigned int get_buffer_pointer(const unsigned int cbpointer, const unsigned int cnperiod)
{
    if (cnperiod <= cbpointer) {
        return cbpointer - cnperiod;
    }
    else {
        return cbpointer - cnperiod + MAVGFLT_SIZE_MAX;
    }
}
static float get_buffer_value(const MAVGFLT * const f, const unsigned int cnperiod)
{
    const unsigned int current_nperiod = get_buffer_pointer(f->cbpointer, cnperiod);
    return f->buffer[current_nperiod];
}
float mavgflt_process(volatile MAVGFLT *f, float input, const float period)
{
    if (period > 0.0) {
        float samples_requested_f = period / f->ts;
        if (samples_requested_f > MAVGFLT_SIZE_MAX - 1)
        samples_requested_f = MAVGFLT_SIZE_MAX - 1;
        const unsigned int cnbuffer = samples_requested_f + 1;
        const float sampling_fraction = samples_requested_f + 1 - cnbuffer;
        if (cnbuffer > f->cbsample) {
            f->sshort += f->pinput;
            f->slong += input;
            f->cbsample++;
        }
        else if (cnbuffer < f->cbsample) {
            const unsigned int decremented_cbsample = f->cbsample - 1;
            const float two_oldest_samples = get_buffer_value((MAVGFLT*) f, f->cbsample)
            + get_buffer_value((MAVGFLT*) f, decremented_cbsample);
            f->sshort += f->pinput - two_oldest_samples;
            f->slong += input - two_oldest_samples;
            f->cbsample = decremented_cbsample;
        }
        else {
            const float oldest_period_sample = get_buffer_value((MAVGFLT*) f, f->cbsample);
            f->sshort += f->pinput - oldest_period_sample;
            f->slong += input - oldest_period_sample;
        }
        f->buffer[f->cbpointer] = input;
        f->cbpointer++;
        f->pinput = input;
        if (f->cbpointer == MAVGFLT_SIZE_MAX) {
            f->cbpointer = 0;
        }
        const float cbsample_short = f->cbsample - 1;
        const float filter_output_short = cbsample_short ? f->sshort / cbsample_short : 0.0;
        const float filter_output_long = f->slong / f->cbsample;
        return filter_output_short * (1 - sampling_fraction) +
        filter_output_long * sampling_fraction;
    }
    else {
        return 0.0;
    }
}
```

Chapter 4

Implementation on dSPACE SCALEXIO

4.1 Introduction

The theory of the *PLL* and of the *MAVG* filter described along the previous chapters will be experimentally investigated using the fast prototyping equipment *SCALEXIO*. The structure of the experiment is depicted in Figure 4.1.

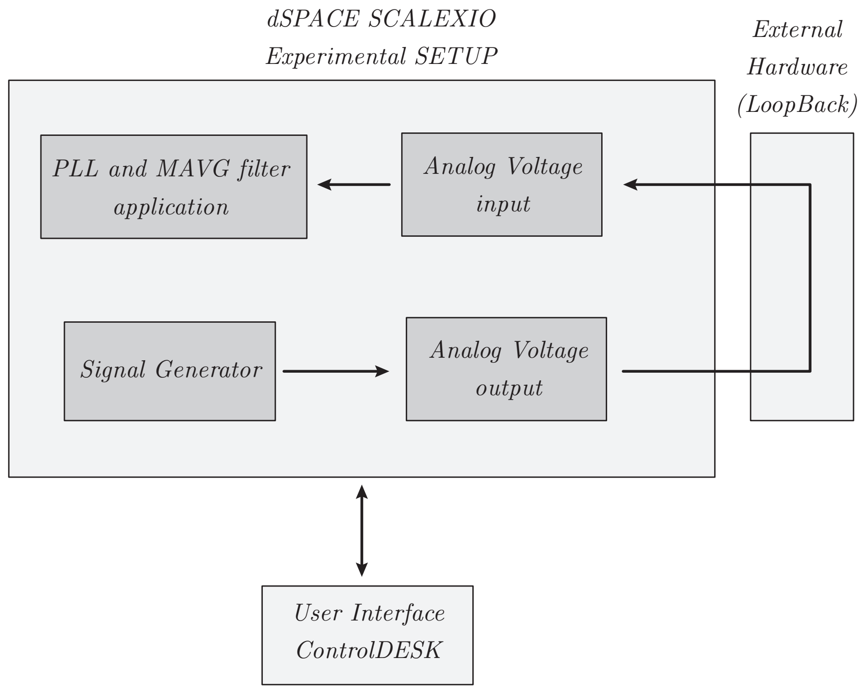


Figure 4.1: Description of the experiment.

Conceptually the experiment consists of to generate a signal from the scalexio box and to loop it back into the scalexio itself where the *PLL* application is applied. The loop back will be implemented by an analog output and an analog input available on DS6101 board of the scalexio.

Via Simulink an application of the *PLL* and of the *MAVG* filter is designed as well as a signal generator. The output of the signal generator is connected to an analog voltage output of the *SCALEXIO*. The analog voltage output is looped back to an analog voltage input of the *SCALEXIO* and it is used as signal input to the *PLL* and *MAVG* filter application. The analog voltage input and output are taken from the DS6101 board available on the *SCALEXIO* according Figure 4.2. The connection between

Simulink model and the scalexio I/O will activated via *ConfigurationDesk* during the phase of creation of the project.

4.2 Configuring the experiment

In this section the configuration of the project/experiment on *ConfigurationDesk* is shown. Figure 4.2 shows the hardware and model interface structure which must be implemented in the *ConfigurationDESK* application.

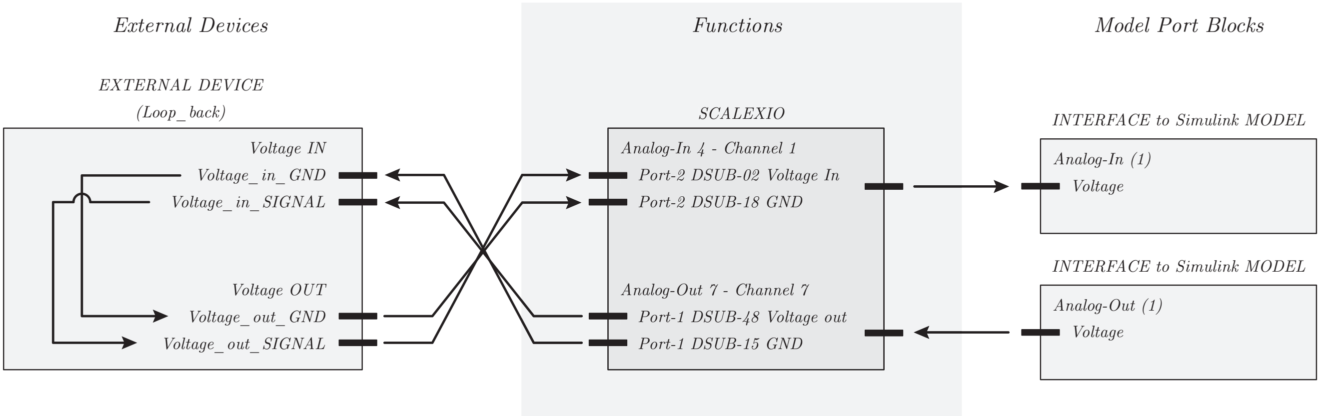


Figure 4.2: Hardware configuration of the experiment.

Supposing to be in the *ConfigurationDESK* application and to have already created and loaded the application:

– from *signal chain* drag and drop the *Analog In 4 - Channel 1* from *Hardware Resource* to *Global/Signal Chain* window;

– again, drag and drop the *Analog Out 7 - Channel 1* from *Hardware Resource* to *Global/Signal Chain* window;

– select *Voltage In (1)* and RMB and select *Propagate to ConfigurationDesk Model Interface*;

– select *Voltage In (1)* and RMB and select *Propagate to ConfigurationDesk Model Interface*;

Now in the *Model Port Blocks* of the *Signal Chain* window the are to additional blocks which are the interface between the scalexio device and the simulink model.

– from *signal chain* RMB on *Voltage In (1)* and select *Propagate to Simulink Model*;

– from *signal chain* RMB on *Voltage Out (1)* and select *Propagate to Simulink Model*;

Now two additional interface *dSPACE* blocks shall be available in the simulink model, *Voltage - Data Inport Block* and *Voltage - Data Outport Block* respectively.

The creation of the *external device* can be done in two different ways:

– from *External Devices* available on the left window of the *Signal Chain* tab RMB and *Import Devices* (merge/replace). The external device can be imported as file **.dtfx* file;

– the external device can be created directly from *External Devices* tab available on the left window of the *Signal Chain* tab, see the Section 4.3.

After the connection of the blocks, which must be done according to the purpose of the project, the *Signal Chain* window will appear similarly to Figure 4.2.

4.3 Creating the external hardware

The external device interface can be created from the *External Devices* tab of the left window of the *Signal Chain* tab of the *ConfigurationDesk*, as follows:

- from *External Devices* tab, available on the left window of the *Signal Chain* tab RMB and select *New/Device*. Suppose to rename it: *Loop_back*;
- RMB on *Loop_back* new device and select *New/Port* and rename it, e.g.: *Voltage_in_GND*;
- select the *Voltage_in_GND* port and update information on *Electrical Interface* available of the right window of the *Signal Chain* tab;
  - from *Electrical Interface* select *Device Port Settings/Port Type* and select *In*;
  - from *Electrical Interface* select *Device Port Settings/Port Attributes* and select *Voltage*;
- RMB on *Loop_back* new device and select *New/Port* and rename it, e.g.: *Voltage_in_SIGNAL*;
- select the *Voltage_in_SIGNAL* port
  - from *Electrical Interface* select *Device Port Settings/Port Type* and select *In*;
  - from *Electrical Interface* select *Device Port Settings/Port Attributes* and select *Voltage*;
- RMB on *Loop_back* new device and select *New/Port* and rename it, e.g.: *Voltage_out_GND*;
- select the *Voltage_out_GND* port
  - from *Electrical Interface* select *Device Port Settings/Port Type* and select *Out*;
  - from *Electrical Interface* select *Device Port Settings/Port Attributes* and select *Voltage*;
- RMB on *Loop_back* new device and select *New/Port* and rename it, e.g.: *Voltage_out_SIGNAL*;
- select the *Voltage_out_SIGNAL* port
  - from *Electrical Interface* select *Device Port Settings/Port Type* and select *Out*;
  - from *Electrical Interface* select *Device Port Settings/Port Attributes* and select *Voltage*;

Chapter 5

Appendices

5.1 The Fourier Series

Bibliography

[1] R. Teodorescu, M. Liserre, P. Rodrıguez, *Grid converters for photovoltaic and wind power systems.* Wiley, 2011.