# Práctica de Laboratorio 6: Utilización y evaluación de estimadores de estado de predicción y actual.


Elías Álvarez
Carrera de Ing. Electrónica
Universidad Católica Nuestra Señora de la Asunción
Asunción, Paraguay
Email: elias.alvarez@universidadcatolica.edu.py

Tania Romero
Carrera de Ing. Electrónica
Universidad Católica Nuestra Señora de la Asunción
Asunción, Paraguay
Email: tania.romero@universidadcatolica.edu.py

Docente: Lic. Montserrat González
Facultad de Ingeniería
Universidad Católica Nuestra Señora de la Asunción
Asunción, Paraguay

Docente: PhD. Enrique Vargas
Facultad de Ingeniería
Universidad Católica Nuestra Señora de la Asunción
Asunción, Paraguay


## I. Introducción

## II. Objetivos

- Diseñar controladores digitales para una planta analógica utilizando estados estimados.
- Aplicar el diseño de realimentación de estados con estimadores de predicción y de actualización.
- Presentar y discutir los resultados experimentales en comparación con los obtenidos por simulación, dentro de un informe técnico razonado.

## III. Desarrollo

Considerando la planta analógica esquematizada en la Fig. 1, se procedió a diseñar compensadores digitales utilizando en forma conjunta la técnica de *ubicación arbitraria de polos* y el empleo de *estimadores de estado* en tiempo discreto, tal como lo exige la consigna de la Práctica 6. En esta sección se documentan las decisiones de diseño, los cálculos realizados en MATLAB (código completo en el Apéndice, ver Código **??**) y la comparación entre las dos arquitecturas de estimador: el **estimador de predicción** y el **estimador actual**.

### *III-A. Selección del tiempo de muestreo*

El primer paso fue fijar un tiempo de muestreo $T_s$ que permita capturar la dinámica más rápida de la planta. Para ello se partió del modelo continuo obtenido a partir de los valores medidos de resistencias y capacitores:

$$C_1 = 211{,}100 \times 10^{-9}\,\mathrm{F}, \qquad C_2 = 103{,}070 \times 10^{-9}\,\mathrm{F},$$
$$R_1 = 47{,}000 \times 10^{3}\,\Omega, \qquad R_2 = 81{,}090 \times 10^{3}\,\Omega,$$
$$R_3 = 6{,}800 \times 10^{3}\,\Omega, \qquad R_4 = 14{,}760 \times 10^{3}\,\Omega.$$

Con estos valores se definieron las constantes de tiempo y las ganancias

$$\tau_1 = R_2 C_1, \qquad \tau_2 = R_4 C_2, \qquad k_1 = -\frac{R_2}{R_1}, \qquad k_2 = -\frac{R_4}{R_3},$$

y se armó el modelo continuo en espacio de estados:

$$F = \begin{bmatrix} -\dfrac{1}{\tau_1} & 0 \\ \dfrac{k_2}{\tau_2} & -\dfrac{1}{\tau_2} \end{bmatrix}, \quad G = \begin{bmatrix} \dfrac{k_1}{\tau_1} \\ 0 \end{bmatrix}, \quad H = \begin{bmatrix} 0 & 1 \end{bmatrix}, \quad J = 0,$$

que coincide con la planta analógica real.

El cálculo en MATLAB se hizo con el siguiente fragmento (el código completo está en el Apéndice, Código **??**):

```
clear all; clc; close all;

% --- Planta continua (modelo medido) ---

C2 = 103.07e-9;  C1 = 211.1e-9;
R3 = 6.8e3;      R4 = 14.760e3;
R1 = 47e3;       R2 = 81.09e3;

tau1 = R2*C1;            k1 = -R2/R1;
tau2 = R4*C2;            k2 = -R4/R3;

F = [ -1/tau1,     0;
k2/tau2, -1/tau2 ];
G = [ k1/tau1; 0 ];
H = [0 1];  J = 0;

sysC = ss(F,G,H,J);
Gc   = tf(sysC);

% Ts base vía polo más rápido continuo
fn = max(abs(zpk(Gc).P{1})) / pi;
Tn = 1/fn;
Ts = Tn/4;    % 4 muestras por la diná
mica más rápida

% Discretización
sysD = c2d(sysC, Ts, 'zoh');
[A,B,C,D] = ssdata(sysD);

% Controlabilidad / observabilidad
Co = ctrb(A,B);
Ob = obsv(A,C);
rCo = rank(Co); rOb = rank(Ob);
```

Listing 1. Cálculo de $T_s$ a partir del polo continuo más rápido.

La lógica fue:

1. obtener los polos continuos de la planta;
2. tomar el polo de mayor módulo (el más rápido);
3. definir el período asociado $T_n = 1/f_n$;
4. fijar el tiempo de muestreo como

$$T_s = \frac{T_n}{4},$$

es decir, cuatro muestras sobre la parte más rápida de la dinámica.

Con ese $T_s$ se discretizó la planta y se verificó que

$$\text{rango}(\text{ctrb}(A, B)) = \text{rango}(\text{obsv}(A, C)) = 2,$$

por lo que el diseño por ubicación de polos y el diseño de observadores son viables en discreto.

### *III-B. Diseño del controlador por realimentación de estados*

Con el modelo discreto $(A, B, C, D, T_s)$ se diseñó primero el lazo de control. Se eligió la pareja compleja

$$p_{\text{ctrl}} = 0{,}8 \pm j\,0{,}2$$

porque:

- queda más lenta que un diseño agresivo, por lo que el esfuerzo de control no se dispara;
- está dentro del círculo unidad y asegura un régimen aceptable;
- está en una zona intermedia respecto de los polos discretizados de la planta.

La ganancia de realimentación se obtuvo con Ackermann:

```
% Polos de control deseados
p_ctrl = [0.8 + 1j*0.2; 0.8 - 1j*0.2];

% Ganancia de realimentación de estados
K = acker(A,B,p_ctrl);

% Prefiltro para que y_ss = r (caso SISO y D=0)
[~,~,Nbar] = refi(A,B,C,K);
```

Listing 2. Cálculo de $K$ y del prefiltro $N_{\text{bar}}$.

La ley de control que se implementó fue entonces

$$u[k] = N_{\text{bar}}\, r[k] - K\, \hat{x}[k],$$

donde $\hat{x}[k]$ es el *estado estimado*. Esto es así porque en esta planta sólo se mide la salida

$$y[k] = C\, x[k] = x_2[k],$$

y con esa única medición se debe reconstruir el vector completo de estado $x[k]$; por lo tanto, el estimador no es opcional.

### *III-C. Diseño de los dos estimadores de estado*

A partir de la única medición de la salida $y[k]$ se debe llegar a reconstruir el vector de estado completo $x[k] = [x_1\ x_2]^\top$; por eso el estimador es una parte fundamental del diseño. Se implementaron las dos variantes que se piden en el laboratorio:

1. **Estimador de predicción**:

$$\hat{x}[k+1] = A\,\hat{x}[k] + B\,u[k] + L_{\text{pred}}\,(y[k] - C\,\hat{x}[k]).$$

2. **Estimador actual** (con la predicción $z[k+1]$ que usás en tu código):

$$z[k+1] = A\,\hat{x}[k] + B\,u[k],$$
$$\hat{x}[k+1] = z[k+1] + L_{\text{act}}\,(y[k+1] - C\,z[k+1]).$$

Para comparar el efecto de la dinámica del observador se usaron dos ubicaciones en el plano-$z$:

- caso 1 (moderado): $p_{\text{obs}} = 0{,}4 \pm j\,0{,}4$;
- caso 2 (rápido): $p_{\text{obs}} = 0{,}2 \pm j\,0{,}2$.

Las ganancias se obtuvieron por dualidad:

```
% Polos del observador (cambiar 0.4+0.4j <-> 0.2+0.2j según el caso)
p_obs = [0.2+1j*0.2, 0.2-1j*0.2];

% Estimador de PREDICCIÓN: corrige con y[k]
L_pred = acker(A', C', p_obs).';

% Estimador ACTUAL: corrige con y[k+1], usa C*A
L_act  = acker(A', (C*A)', p_obs).';
```

Listing 3. Ganancias de los dos estimadores.

La condición buscada fue que los polos del observador queden más cerca del origen que los polos del lazo cerrado $A - BK$, de modo que el error de estimación desaparezca en pocas muestras y la dinámica dominante siga siendo la de los polos que se ubicaron para la planta.

### *III-D. Simulación comparativa*

Con $K$, $N_{\text{bar}}$, $L_{\text{pred}}$ y $L_{\text{act}}$ se corrieron dos simulaciones sobre el mismo modelo discreto $(A, B, C, D, T_s)$: una con `sim_predictor` y otra con `sim_current`. En ambas se usó:

- la misma referencia alternada;
- las mismas condiciones iniciales no nulas para $x[0]$ y $\hat{x}[0]$;
- y el mismo horizonte de tiempo.

```
simPred = sim_predictor(A,B,C,D, K, L_pred, Nbar, Ts, Tsim, x0, xh0, r);
simPred.title = "Estimador de prediccion";

simCurr = sim_current(A,B,C,D, K, L_act, Nbar, Ts, Tsim, x0, xh0, r);
simCurr.title = "Estimador actual";
```

Listing 4. Simulación de ambas arquitecturas.

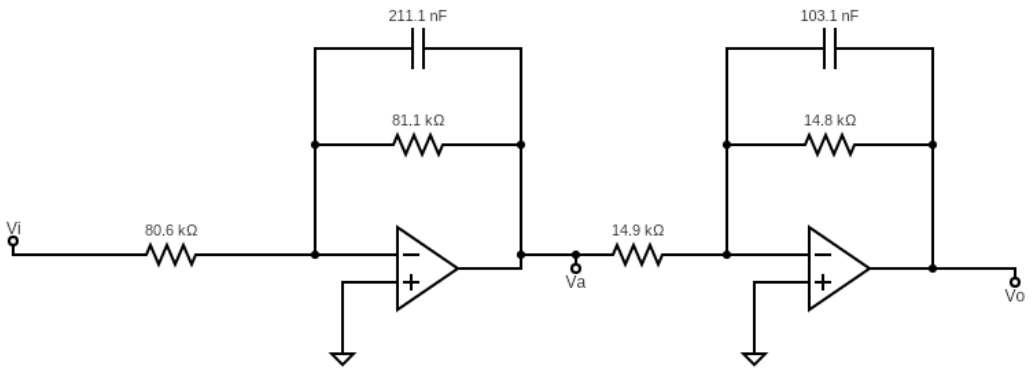
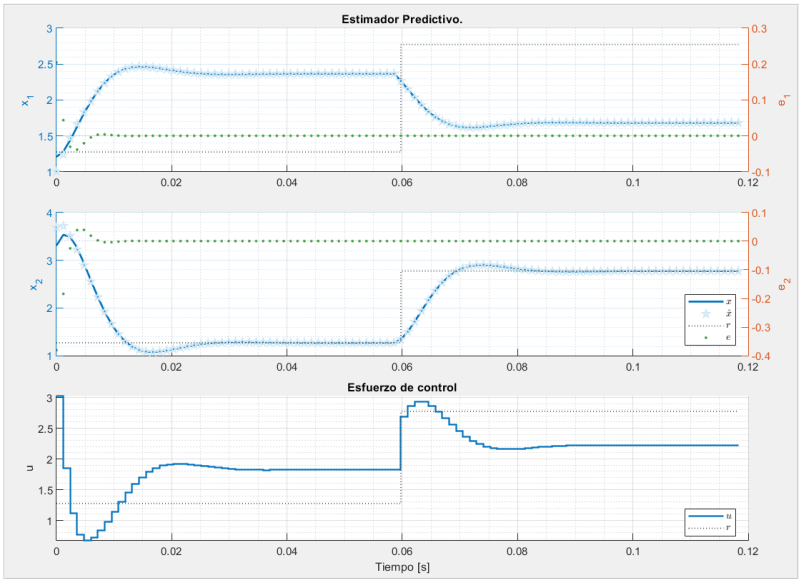


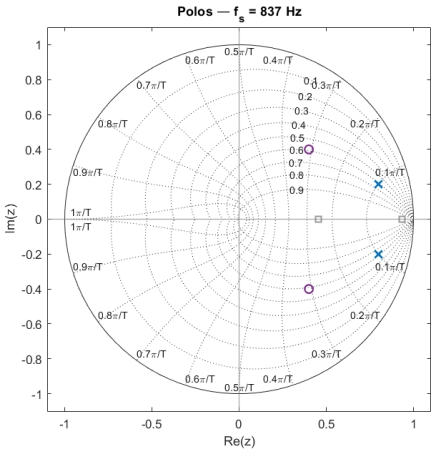

Figura 1. Planta analógica discretizada para el laboratorio.

### *III-E. Discusión de resultados*

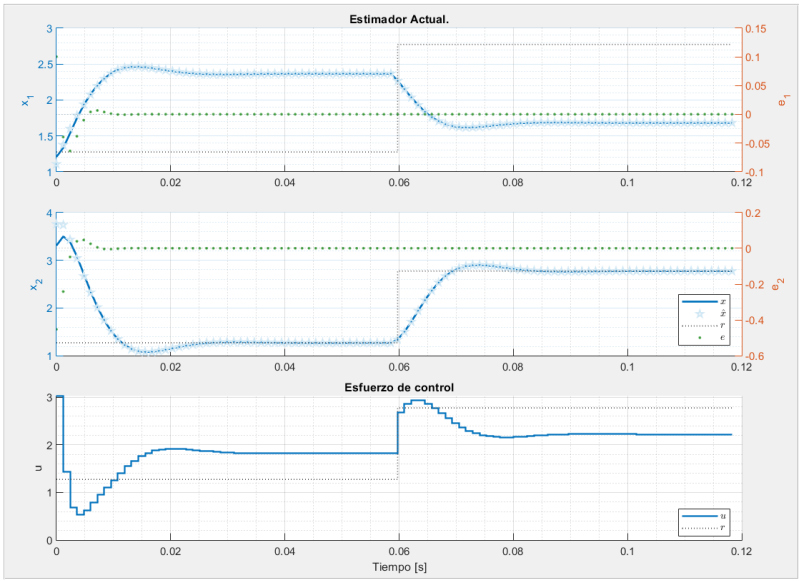
De las cuatro simulaciones se observa que:

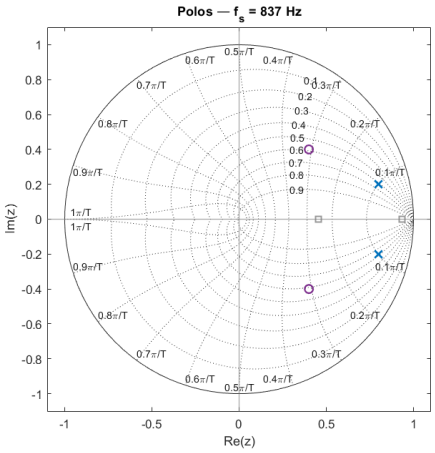
1. Para una misma ubicación de polos del observador, el estimador actual converge en menos muestras que el estimador de predicción.
2. Al acelerar los polos del observador (de $0{,}4 \pm j0{,}4$ a $0{,}2 \pm j0{,}2$), ambos estimadores se vuelven muy rápidos y la diferencia entre sus trayectorias de estado se reduce.
3. Como la ley de control es la misma en todos los casos,
$$u[k] = N_{\text{bar}} r[k] - K\hat{x}[k],$$
la salida en lazo cerrado termina siendo prácticamente igual una vez que el estimador alcanza al estado real.
4. La elección de polos del controlador en $0{,}8 \pm j0{,}2$ permitió que las diferencias que se ven en las figuras se deban al observador y no a una saturación del actuador.



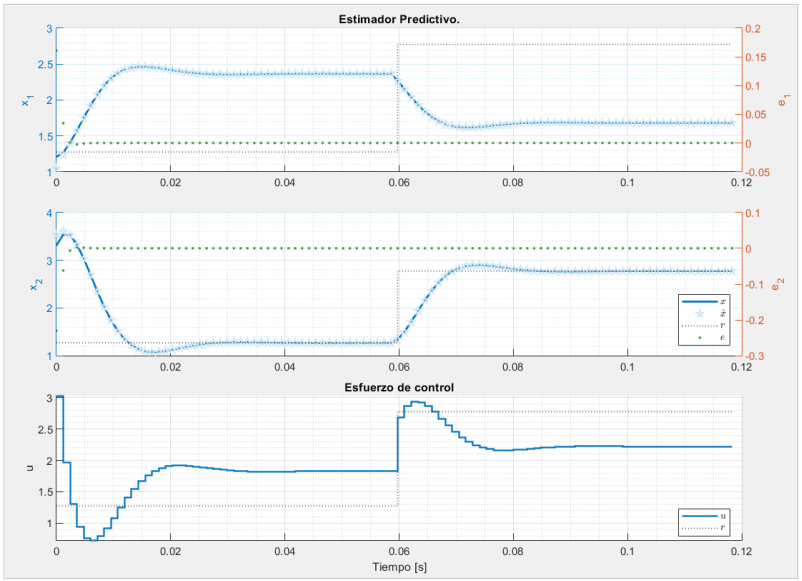

(a) Predicción, $p_{\text{obs}} = 0{,}4 \pm j0{,}4$

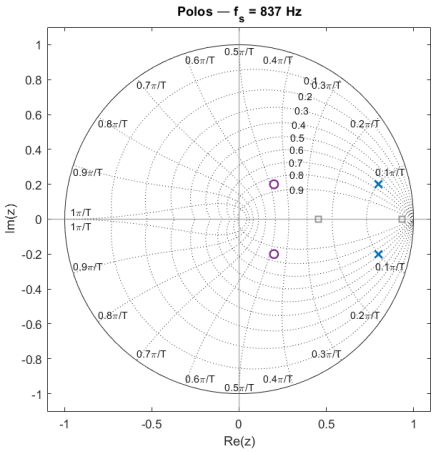


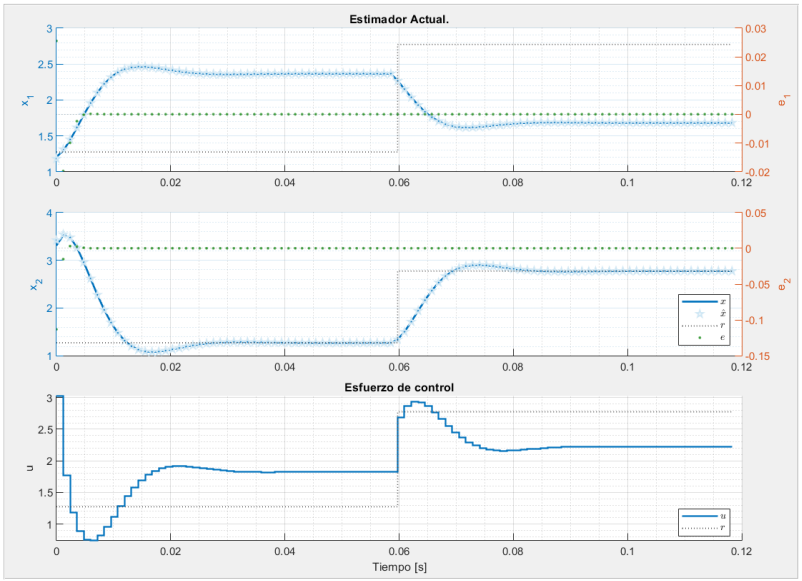

(b) Actual, $p_{\text{obs}} = 0{,}4 \pm j0{,}4$

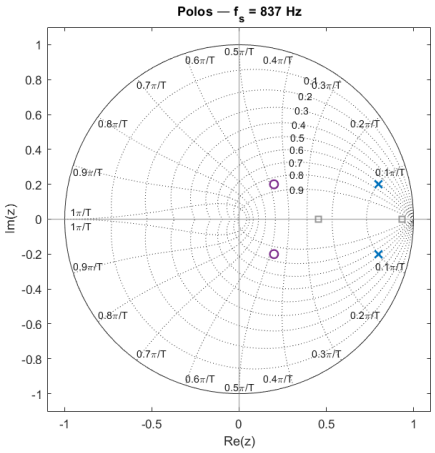
Figura 2. Comparación de respuestas para estimador de predicción y actual con polos de observador moderados.



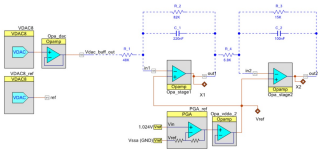

(a) Predicción, $p_{\text{obs}} = 0{,}2 \pm j0{,}2$



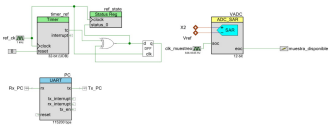




(b) Actual, $p_{\text{obs}} = 0{,}2 \pm j0{,}2$

Figura 3. Comparación de respuestas para estimador de predicción y actual con polos de observador rápidos.



(a) PSoC: planta y DAC (pág. 1).



(b) PSoC: ref, UART, ADC e ISR (pág. 2).

Figura 4. Implementación en PSoC del controlador con estimador de predicción.

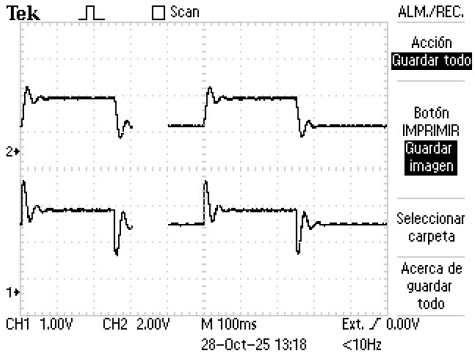



Figura 5. Respuesta experimental del esquema con estimador de predicción con errores.

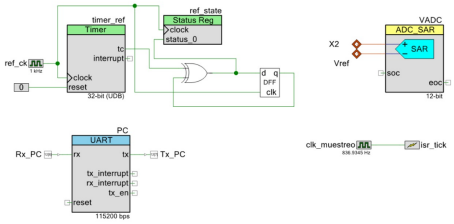



Figura 6. PSoC: esquema modificado para el estimador actual (pág. 2).

## IV. Implementación

En esta sección se documenta la implementación en el PSoC de los dos esquemas diseñados en MATLAB: el controlador con *estimador de predicción* y el controlador con *estimador actual*, ambos usando los polos del observador en

$$p_{\text{obs}} = 0{,}2 \pm j\,0{,}2.$$

Los códigos fuente completos están en el Apéndice:

- controlador con estimador de predicción: Código **??**;
- controlador con estimador actual: Código **??**;
- versión reducida para depurar tiempos: Código **??**.

### IV-A. Implementación del estimador de predicción

La primera implementación que se probó en hardware fue la del estimador que corrige en el mismo instante $k$. El proyecto de PSoC quedó armado así:

- **Página 1**: planta + DAC para inyectar el esfuerzo, como en los labs anteriores.
- **Página 2**: toda la "circuitería de servicio": generación/carga de la referencia por UART (amplitud y período del tren de pulsos), ADC para medir la salida y el pin de la ISR conectado al *End of Conversion* del ADC.

El código correspondiente es el Código **??**. La lógica es la misma que en la simulación: 1) leer $y[k]$ del ADC, 2) calcular $u[k] = N_{\text{bar}} r[k] - K\hat{x}[k]$, 3) actualizar $\hat{x}[k{+}1] = A\hat{x}[k] + Bu[k] + L_{\text{pred}}(y[k] - C\hat{x}[k])$, 4) escribir $u[k]$ en el DAC.

En el prototipado funcionó, pero la forma de la respuesta no coincidía con la de la simulación: el esfuerzo tenia una forma diferente y la salida tenía sobreimpulso y un pequeño rizado. Eso se ve en la Fig. 5.

La causa fue el **retardo real** del lazo de interrupción: el cálculo del estimador + el cálculo del control estaban demasiado cerca del siguiente muestreo, así que en vez de aplicar el control en $T_s$ se estaba aplicando "un poquito después", y eso la simulación no lo contemplaba.

### IV-B. Implementación del estimador actual

Como el predictor no quedó igual al modelo, se intentó el esquema con *estimador actual*. En este caso solo cambió la segunda página del PSoC (la de la lógica), porque la planta y el DAC quedaron igual.

El código usado es **??**. A diferencia del predictor, la secuencia en la ISR es: 1) formar la predicción $z[k{+}1] = A\hat{x}[k]{+}Bu[k]$, 2) aplicar el esfuerzo al DAC, 3) medir la nueva salida, 4) corregir $\hat{x}[k{+}1] = z[k{+}1]{+}L_{\text{act}}(y[k{+}1]{-}Cz[k{+}1])$.

Esta versión, al principio, **no funcionó siquiera**: La salida oscilaba entre los valores extremos del DAC. Eso nos llevó a revisar tiempos de ejecución con el depurador.

### IV-C. Retardo medido y reloj

Las pausas en la ISR y los mensajes por serie mostraron que el tiempo desde el *End of Conversion* hasta que el DAC actualizaba era comparable con $T_s$. En esas condiciones, la suposición del diseño ("el control se aplica en el mismo instante de muestreo") deja de ser válida.

Para reducir ese retardo se decidió **subir la frecuencia del *master clock*** del PSoC. Con más clock, el mismo código C ocupó menos porcentaje del período y las dos versiones (predictor y actual) empezaron a parecerse más al caso ideal.

Se ve que, con el clock más rápido, el sobreimpulso baja y el esfuerzo de control se acerca más al de la simulación. O sea: el problema principal no era "qué estimador usar" sino **cuánto tarda el micro en hacer las cuentas**.

Cuadro I
COMPARACIÓN EXPERIMENTAL Y SIMULADA DEL DESEMPEÑO.

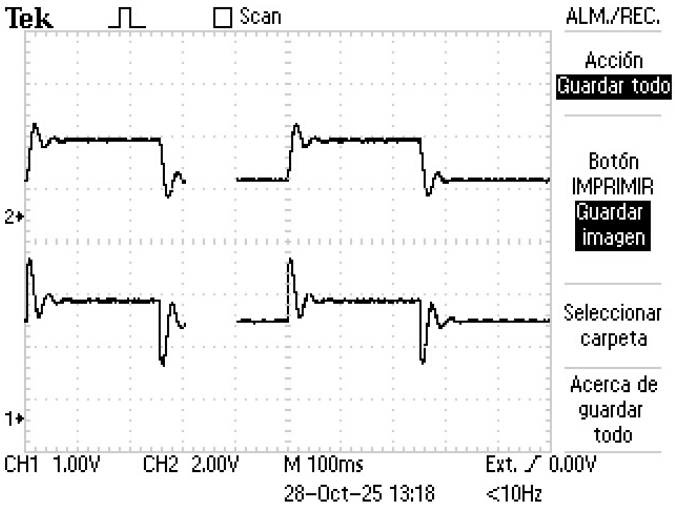
| **Caso** | $t_s$ [ms] | OS [ %] |
|---|---|---|
| Predictor (Sim) | 7,000 | 4,600 |
| Predictor (clock bajo) | 3,580 | <10,000 |
| Predictor (clock alto) | 6,520 | 18,600 |
| Actual (Sim) | 7,000 | 4,600 |
| Actual (clock alto) | 5,940 | ≈10,000 |

## V. RESULTADOS

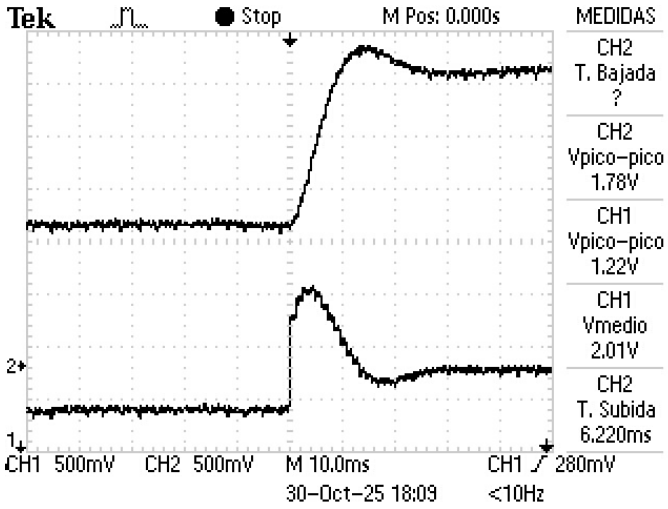
En las figuras anteriores se aprecia que la implementación con un tiempo de muestreo insuficiente difiere notablemente de la respuesta esperada, mientras que tanto el observador *predictivo* como el *actual* logran un desempeño similar al de la simulación, aunque con un sobreimpulso mayor al teórico. Las discrepancias se explican principalmente por las tolerancias de los componentes pasivos, pequeñas desviaciones en las ganancias reales del sistema y efectos de cuantización de los conversores DAC/ADC.

La tabla I resume los valores medidos y simulados. Se observa que con el *clock* bajo la respuesta dista mucho de la esperada, evidenciando la importancia del período de que se cumplan los supuestos en el diseño. En cambio, con el *clock* alto, ambas variantes del observador mantienen una dinámica coherente con lo diseñado, aunque con un sobreimpulso mayor al simulado. El incremento del OS puede atribuirse a las tolerancias de resistencias y capacitores, a la dispersión de los polos reales respecto a los teóricos y a las no idealidades del hardware.

En conjunto, los resultados confirman que ambos esquemas de estimador son implementables y que **la elección adecuada del $T_s$ y del reloj interno del PSoC es tan determinante como la ubicación de polos** para lograr un control estable y preciso.

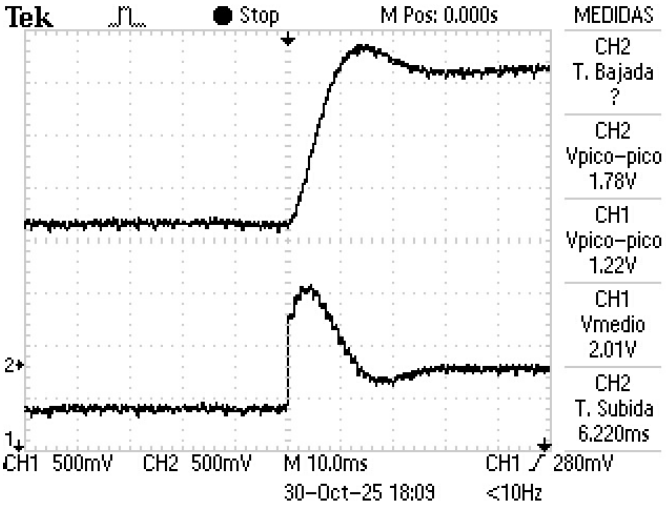



(a) Estimador de predicción (clock original).

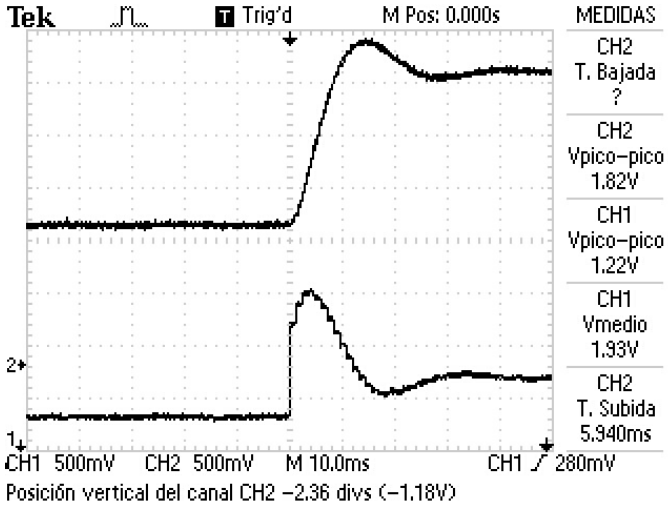



(b) Estimador predicción (clock aumentado).

Figura 8. Respuestas experimentales cambiando el clock.

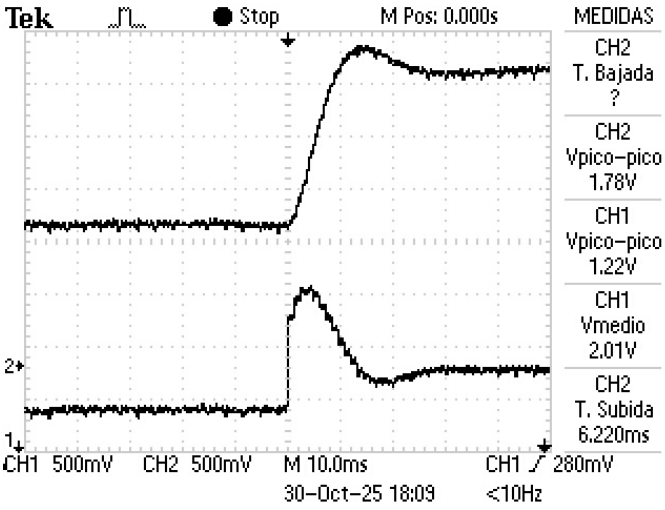



(a) Estimador de predicción (clock aumentado).

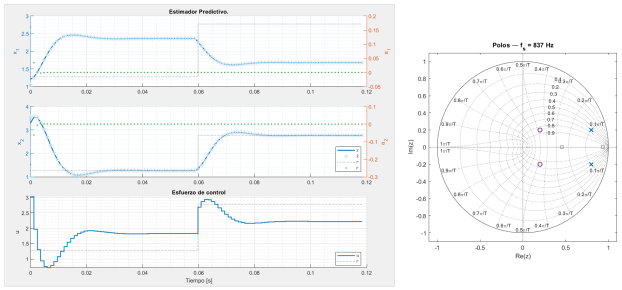



(b) Estimador actual (clock aumentado).

Figura 9. Respuestas experimentales con el clock aumentado.

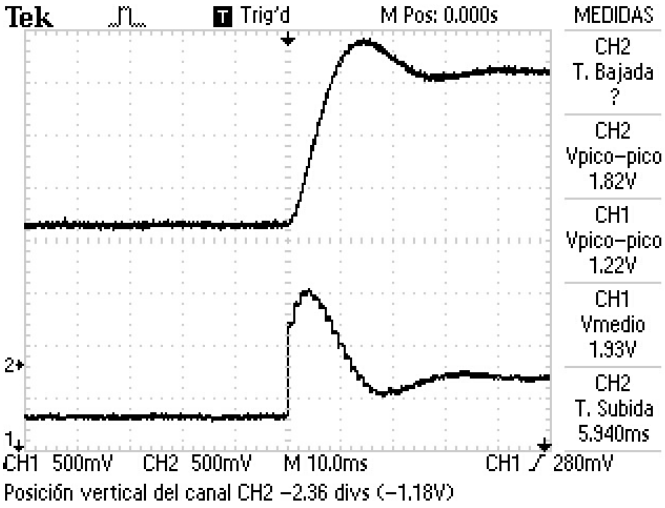



(a) Estimador de predicción (clock aumentado).

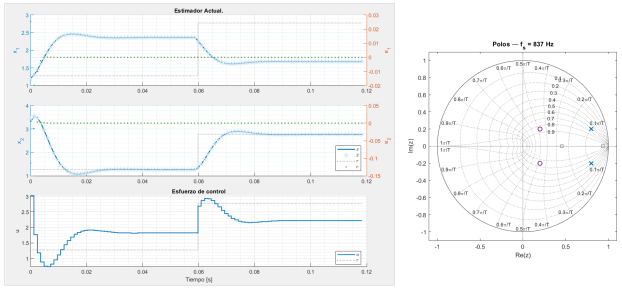


(b) Estimador de predicción (simulación).

Figura 10. Respuesta experimental con el clock aumentado vs simulación.




(a) Estimador de actual (clock aumentado).



(b) Estimador de actual (simulación).

Figura 11. Respuesta experimental con el clock aumentado vs simulación.

## APÉNDICE

En este anexo se incluyen los archivos en C utilizados en el PSoC para implementar los dos esquemas de control (estimador de predicción y estimador actual), así como las librerías de apoyo y los scripts de MATLAB.

### A. *Código en C — Estimador predictivo*

```c
#include "project.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#include "config.h"
#include "mat.h"        /* tu librería */
#include "io.h"
#include "serial.h"
#include "control.h"

/* ===== Estado global ===== */
volatile char   flag = '0';
volatile double ref_max = (float)(A_AMPL/2);
volatile double ref_min = (float)(-A_AMPL/2);
volatile double ref     = 0.0;          /* referencia en AC */
volatile double y_last_volts = 0.0;     /* última muestra del ADC (tu hardware ya entrega AC)
     */

volatile uint8 mode = MODE_CLOSED;
static control_state_t ctrl_st;

/* ===== PEGAR EN main.c ===== */
static const control_model_t M = {
    .Ad = {.filas=2,.columnas=2,.d={{0.932581, 0.000000},
                                    {-1.145618, 0.455938}}},
    .Bd = {.filas=2,.columnas=1,.d={{-0.117824},
                                    {0.080093}}},
    .Cd = {.filas=1,.columnas=2,.d={{0.000000, 1.000000}}},
    .K  = {.filas=1,.columnas=2,.d={{2.907022, 1.636048}}},
    .K0 = {.filas=1,.columnas=1,.d={{0.569878}}},
    .L  = {.filas=2,.columnas=1,.d={{-0.503374},
                                    {0.988519}}}
};

/* poles(A):   0.455938  0.932581 */
/* poles(A-BK):   0.800000  0.800000 */
/* poles(A-LC):   0.200000  0.200000 */
/* Ts = 1.19483659e-03 s */
/* Podés cambiar a CL_CURRENT por UART ('law:curr') si luego querés */
static control_law_t g_law = CL_PREDICTOR;

/* ===== ISR de muestreo ===== */
CY_ISR(adquirirMuestra)
{
    /* NO limpiar pending aquí; el EOC siguiente genera un nuevo pulso */
    /* NO deshabilitar la interrupción aquí; la dejamos armada permanentemente */

    /* Conmutar referencia (en AC) por pin/switch */
    ref = (ref_state_Read()) ? ref_max : ref_min;

    /* Mostrar la referencia en el DAC auxiliar (para el osciloscopio) */
    io_ref_write((float)(ref + VDDA2));

    /* Leer ADC -> en tu hardware ya está en AC (centrado) */
    y_last_volts = io_adc_read_volts();

    flag = '1';
}
```

```
60
61  /* ===== Utilidades ===== */
62  static long my_lroundf(float x) { return (x >= 0.0f) ? (long)(x + 0.5f) : (long)(x - 0.5f); }
63  /* ===== helpers de parsing y print ===== */
64  static float parse_float_local(const char *s) {
65      char buf[32]; size_t i=0;
66      while (*s && i<sizeof(buf)-1) {
67          char c = *s++;
68          if (c==',') c='.';
69          if (!((c>='0'&&c<='9') || c=='+' || c=='-' || c=='.' || c=='e' || c=='E')) break;
70          buf[i++] = c;
71      }
72      buf[i]='\0';
73      return (float)atof(buf);
74  }
75
76  static long parse_uint_local(const char *s) {
77      while (*s==' '||*s=='\t') ++s;
78      long v=0; while (*s>='0'&&*s<='9') { v = v*10 + (*s-'0'); ++s; }
79      return v;
80  }
81
82  /* Set de amplitud 'a' => ref_max = +a/2, ref_min = -a/2, imprime en mV */
83  static void set_amp(float a) {
84      if (a < 0.0f || a > 1.5f) { serial_puts("ERR: a out of range (0..1.5)\r\n"); return; }
85      uint8 intr = CyEnterCriticalSection();
86      ref_max = +a*0.5f; ref_min = -a*0.5f;
87      ref = (ref_state_Read()) ? ref_max : ref_min;
88      CyExitCriticalSection(intr);
89
90      /* imprimir sin %f */
91      int amp_mV = (int)(a*1000.0f + 0.5f);
92      serial_puts("OK: a="); char msg[32]; snprintf(msg,sizeof(msg),"%d mVpp\r\n", amp_mV);
        serial_puts(msg);
93  }
94
95  /* STATUS reducido: m, a, p (sin %f) */
96  static void cmd_status(void) {
97      float a = (float)(ref_max - ref_min);
98      unsigned per = (unsigned)timer_ref_ReadPeriod();
99      int a_mV = (int)(a*1000.0f + 0.5f);
100     char msg[96];
101     snprintf(msg,sizeof(msg),
102         "STATUS:\r\n  m=%s\r\n  a=%d mVpp\r\n  p=%u\r\n",
103         (mode==MODE_OPEN)?"open":"closed", a_mV, per);
104     serial_puts(msg);
105 }
106 static void on_serial_line(const char *line_in)
107 {
108     char line[RX_BUF_SZ];
109     strncpy(line, line_in, sizeof(line));
110     line[sizeof(line)-1]='\0';
111
112     /* trim/basura */
113     size_t n=strlen(line);
114     while (n && (line[n-1]=='\r'||line[n-1]=='\n'||isspace((unsigned char)line[n-1]))) line[--
        n]='\0';
115     char *p=line; while(*p && isspace((unsigned char)*p)) ++p;
116     while(*p && !isalnum((unsigned char)*p)) ++p;
117     for(char *q=p; *q; ++q) *q=(char)tolower((unsigned char)*q);
118
119     if (!strcmp(p,"s") || !strcmp(p,"status")) { cmd_status(); return; }
120     if (!strcmp(p,"help") || !strcmp(p,"?")) {
121         serial_puts("a:<0..1.5>, p:<uint16>, m:<open|closed|1|0>, s\r\n"); return;
122     }
123
124     char *sep = strchr(p, ':');
```

```
    if (!sep) { sep=strchr(p,' '); if(sep) *sep=':'; }
    if (!sep) { serial_puts("ERR: expected a:<val> | p:<val> | m:<val> | s\r\n"); return; }
    *sep='\0'; const char *key=p, *val=sep+1; while(*val && isspace((unsigned char)*val)) ++val;

    if (!strcmp(key,"a")) { set_amp(parse_float_local(val)); return; }

    if (!strcmp(key,"p")) {
        long per = parse_uint_local(val);
        if (per<0 || per>65535) { serial_puts("ERR: p out of range (0..65535)\r\n"); return; }

        timer_ref_Stop(); timer_ref_WritePeriod((uint16)per); timer_ref_WriteCounter((uint16)per); timer_ref_Start();
        serial_puts("OK: p set\r\n"); return;
    }

    if (!strcmp(key,"m")) {
        if (!strcmp(val,"open")||!strcmp(val,"1")) { mode=MODE_OPEN; serial_puts("OK: m=open\r\n"); }
        else if (!strcmp(val,"closed")||!strcmp(val,"0")) { mode=MODE_CLOSED; serial_puts("OK: m=closed\r\n"); }
        else { serial_puts("ERR: m must be open|closed|1|0\r\n"); }
        return;
    }

    serial_puts("ERR: unknown key (use a, p, m, s)\r\n");
}



/* ===== main ===== */
int main(void)
{
    CyGlobalIntEnable;

    io_init();
    control_init(&ctrl_st);
    serial_init(on_serial_line);

    /* Timer/ISR de muestreo */
    muestra_disponible_StartEx(adquirirMuestra);
    muestra_disponible_ClearPending();   /* arranque limpio */
    muestra_disponible_Enable();
    timer_ref_Start();

    for(;;) {
        /* UART no bloqueante */
        serial_poll();

        /* procesar 1 muestra */
        if (flag=='1') {
            flag='0';

            float u_phys = control_step(&ctrl_st, &M,
                                        (float)ref,             /* AC */
                                        (float)y_last_volts,    /* AC en tu hardware */
                                        mode,
                                        g_law);

            io_dac_write(u_phys);

            /* ya NO re-armamos la IRQ: queda habilitada siempre */
        }
    }
}
```

Listing 5. `main.c`: bucle principal, adquisición y lógica de control con estimador predictivo.

## B. *Código en C — Estimador actual*

```c
#include "project.h"
#include "config.h"   // SAT_MIN, SAT_MAX, VDDA2, A_AMPL

/* ========= Helpers ========= */
static inline uint8_t volt_to_dac(float v)
{
    if (v < SAT_MIN) v = SAT_MIN;
    if (v > SAT_MAX) v = SAT_MAX;
    float n = v / SAT_MAX; if (n < 0) n = 0; if (n > 1) n = 1;
    return (uint8_t)(255.0f * n);
}
static inline float sat_u_phys(float u_phys)
{
    if (u_phys > SAT_MAX) u_phys = SAT_MAX;
    if (u_phys < SAT_MIN) u_phys = SAT_MIN;
    return u_phys;
}

/* ========= Parámetros del modelo (DISCRETO, en AC) =========
   Copiados de tu export bueno (coherentes con la simulación):
   A = [ 0.932581   0
         -1.145618  0.455938 ]
   B = [ -0.117824
          0.080093 ]
   C = [ 0  1 ]
   K = [ 2.907022  1.636048 ]
   K0 = 0.569878
   Ke (L_current) = [ -0.503374
                      0.988519 ]
*/
static const float a11 = 0.932581f, a12 = 0.000000f;
static const float a21 = -1.145618f, a22 = 0.455938f;
static const float b1  = -0.117824f, b2  = 0.080093f;

static const float k1  = 2.907022f,  k2  = 1.636048f;
static const float k0  = 0.569878f;

static const float l1  = -0.503374f, l2  = 0.988519f;

/* ========= Estado global ========= */
static volatile uint8_t tick_flag = 0;
static volatile float   ref_ac    = 0.0f;   // referencia en AC (+-A/2)
static float xhat1 = 0.0f, xhat2 = 0.0f;    // estimador 2x1

/* Para probar sin UART: amplitud fija A_AMPL */
static const float ref_max =  0.5f * A_AMPL;
static const float ref_min = -0.5f * A_AMPL;

/* ========= ISR de tick (NO des/clear dentro de la ISR) ========= */
CY_ISR(adquirirMuestra)
{
    // referencia por pin (AC)
    ref_ac = ref_state_Read() ? ref_max : ref_min;
    // DAC de referencia (para el osciloscopio)
    VDAC8_ref_SetValue(volt_to_dac(ref_ac + VDDA2));

    // marcar que hay que ejecutar un paso de control
    tick_flag = 1u;

    // NO llamar Disable/Enable/ClearPending aquí.
}
```

```c

/* ========= main ========= */
int main(void)
{
    CyGlobalIntEnable;

    /* Analógico / DAC / ADC */
    Opa_dac_Start();
    Opa_stage1_Start();
    Opa_vdda_2_Start();
    Opa_stage2_Start();

    VDAC8_Start();
    VDAC8_ref_Start();

    VADC_Start();  // Configurar el ADC en "Firmware Trigger" (no free run)

    /* Timer + ISR */
    isr_tick_StartEx(adquirirMuestra);
    timer_ref_Start();     // fija tu Ts

    for (;;)
    {
        if (tick_flag)
        {
            tick_flag = 0u;


            float u_ac = k0 * ref_ac - (k1 * xhat1 + k2 * xhat2);

            /* ===== 2) Aplicar esfuerzo (físico) y saturar ===== */
            float u_phys = sat_u_phys(u_ac + VDDA2);
            VDAC8_SetValue(volt_to_dac(u_phys));

            /* u aplicado en AC (post-sat) */
            float u_ac_apl = u_phys - VDDA2;

            /* ===== 3) Predicción a k+1: z = A*xhat + B*u_aplicado ===== */
            float z1 = a11 * xhat1 + a12 * xhat2 + b1 * u_ac_apl;
            float z2 = a21 * xhat1 + a22 * xhat2 + b2 * u_ac_apl;

            /* ===== 4) Medición y[k+1] (AC) ===== */
            VADC_StartConvert();
            while (VADC_IsEndConversion(VADC_RETURN_STATUS) == 0) { /* busy wait corto */ }
            float y_ac = VADC_CountsTo_Volts(VADC_GetResult16());   // tu hardware ya entrega
     AC


            float innov = y_ac - z2;
            xhat1 = z1 + l1 * innov;
            xhat2 = z2 + l2 * innov;
        }
    }
}
```

Listing 6. `main.c`: bucle principal, adquisición y lógica de control con estimador actual.

### C. *Código en C — Versión simplificada (debug)*

```c
#include "project.h"
#include "config.h"

/* ---------- Helpers ---------- */
static inline uint8_t volt_to_dac(float v)
{
    if (v < SAT_MIN) v = SAT_MIN;
```

```
    if (v > SAT_MAX) v = SAT_MAX;
    float n = v / SAT_MAX; if (n < 0) n = 0; if (n > 1) n = 1;
    return (uint8_t)(255.0f * n);
}
static inline float sat_u_phys(float u)
{
    if (u > SAT_MAX) u = SAT_MAX;
    if (u < SAT_MIN) u = SAT_MIN;
    return u;
}

/* ---------- Modelo discreto ---------- */
float a11 = 0.932581f, a12 = 0.000000f;
float a21 = -1.145618f, a22 = 0.455938f;
float b1  = -0.117824f, b2  = 0.080093f;
float c1  = 0.0f,       c2  = 1.0f;

float k1  = 2.907022f,  k2  = 1.636048f;
float k0  = 0.569878f;

float l1  = -0.503374f, l2  = 0.988519f;

/* ---------- Estado ---------- */
static volatile uint8_t tick_flag = 0;
static volatile float ref_ac = 0.0f;
static float xhat1 = 0.0f, xhat2 = 0.0f;

static const float ref_max =  0.5f * A_AMPL;
static const float ref_min = -0.5f * A_AMPL;

/* ---------- ISR de muestreo ---------- */
CY_ISR(adquirirMuestra)
{
    /* referencia AC según el switch */
    ref_ac = ref_state_Read() ? ref_max : ref_min;
    /* DAC auxiliar para scope */
    VDAC8_ref_SetValue(volt_to_dac(ref_ac + VDDA2));
    tick_flag = 1u;
}

/* ---------- MAIN ---------- */
int main(void)
{
    CyGlobalIntEnable;

    /* Front-end analógico */
    Opa_dac_Start();
    Opa_stage1_Start();
    Opa_vdda_2_Start();
    Opa_stage2_Start();

    VDAC8_Start();
    VDAC8_ref_Start();

    VADC_Start();  // Firmware trigger (NO free-run)

    muestra_disponible_StartEx(adquirirMuestra);
    timer_ref_Start(); // genera Ts

    for (;;)
    {
        if (tick_flag)
        {
            tick_flag = 0u;


            float u_ac = k0 * ref_ac - (k1 * xhat1 + k2 * xhat2);
```

```
            /* 2) Leer y[k] (AC) */
            VADC_StartConvert();
            while (VADC_IsEndConversion(VADC_RETURN_STATUS) == 0) {}
            float y_ac = VADC_CountsTo_Volts(VADC_GetResult16());


            float y_hat = c1 * xhat1 + c2 * xhat2;
            float err   = y_ac - y_hat;
            float x1n   = a11 * xhat1 + a12 * xhat2 + b1 * u_ac + l1 * err;
            float x2n   = a21 * xhat1 + a22 * xhat2 + b2 * u_ac + l2 * err;
            xhat1 = x1n;
            xhat2 = x2n;

            /* 4) DAC físico (con saturación y offset) */
            float u_phys = sat_u_phys(u_ac + VDDA2);
            VDAC8_SetValue(volt_to_dac(u_phys));
        }
    }
}
```

Listing 7. `main.c`: versión simplificada del estimador de predicción.

## D. *Librerías del proyecto*

```
#ifndef CONFIG_H
#define CONFIG_H

/* === Límites y referencias físicas === */
#define SAT_MIN        (0.0f)
#define SAT_MAX        (4.048f)
#define MIDDLE_VOLTAGE ((SAT_MAX - SAT_MIN)/2.0f)
#define VDDA2          SAT_MAX/2

/* UART */
#define RX_BUF_SZ 64

/* Amplitud para referencia cuadrada (renombrado: NO usar 'A') */
#define A_AMPL 0.5f

/* Modo de operación */
enum { MODE_CLOSED = 0, MODE_OPEN = 1 };

#endif
```

Listing 8. `config.h`: parámetros globales del sistema.

```
#include "control.h"
#include "config.h"   /* SAT_MIN, SAT_MAX, VDDA2 */
#include <string.h>

/* Saturación de esfuerzo físico */
static inline float sat_u_phys(float u)
{
    if (u > SAT_MAX) u = SAT_MAX;
    if (u < SAT_MIN) u = SAT_MIN;
    return u;
}

void control_init(control_state_t *st)
{
    /* xhat = 0 (2x1) */
    st->xhat.filas = 2;
    st->xhat.columnas = 1;
    st->xhat.d[0][0] = 0.0;
    st->xhat.d[1][0] = 0.0;
}
```

```c

/* u = K0*r - K*xhat (AC); observador según 'law' */
float control_step(control_state_t *st,
                   const control_model_t *m,
                   float ref_ac,
                   float y_volts,
                   uint8_t mode,
                   control_law_t law)
{
    /* Lazo abierto: el DAC debe seguir la referencia en físico */
    if (mode == 1 /* MODE_OPEN */) {
        float u_phys_ol = ref_ac + VDDA2;
        return sat_u_phys(u_phys_ol);
    }


    double y_ac = (double)y_volts;  /* ya AC según tu circuito */

    /* u_ac = K0*r - K*xhat */
    Mat R   = (Mat){ .filas=1,.columnas=1,.d={{ ref_ac }} };
    Mat KX  = (Mat){ .filas=1,.columnas=1,.d={{ 0.0 }} };
    Mat K0R = (Mat){ .filas=1,.columnas=1,.d={{ 0.0 }} };

    mat_mul(&m->K,  &st->xhat, &KX);
    mat_mul(&m->K0, &R,        &K0R);

    double u_ac = K0R.d[0][0] - KX.d[0][0];

    /* Llevar a físico y saturar SOLO al salir */
    float u_phys = sat_u_phys((float)(u_ac + (double)VDDA2));

    /* u realmente aplicado en AC (post-sat) */
    double u_ac_apl = (double)u_phys - (double)VDDA2;

    /* Observador */
    Mat Cx  = (Mat){ .filas=1,.columnas=1,.d={{ 0.0 }} };
    mat_mul(&m->Cd, &st->xhat, &Cx);
    double e_y = y_ac - Cx.d[0][0];

    Mat Ax = (Mat){ .filas=2,.columnas=1,.d={{ 0.0 },{ 0.0 }} };
    Mat Bu = (Mat){ .filas=2,.columnas=1,.d={{ 0.0 },{ 0.0 }} };
    Mat Le = (Mat){ .filas=2,.columnas=1,.d={{ 0.0 },{ 0.0 }} };

    Mat U1 = (Mat){ .filas=1,.columnas=1,.d={{ 0.0 }} };
    Mat E1 = (Mat){ .filas=1,.columnas=1,.d={{ e_y }} };

    switch (law) {

        case CL_PREDICTOR:

            U1.d[0][0] = u_ac_apl;
            mat_mul(&m->Ad, &st->xhat, &Ax);
            mat_mul(&m->Bd, &U1,       &Bu);
            mat_mul(&m->L,  &E1,       &Le);

            st->xhat.d[0][0] = Ax.d[0][0] + Bu.d[0][0] + Le.d[0][0];
            st->xhat.d[1][0] = Ax.d[1][0] + Bu.d[1][0] + Le.d[1][0];
            break;

        case CL_CURRENT:

        default:

            U1.d[0][0] = u_ac_apl;
            mat_mul(&m->Ad, &st->xhat, &Ax);
            mat_mul(&m->Bd, &U1,       &Bu);

```

```
            Ax.d[0][0] += Bu.d[0][0];
            Ax.d[1][0] += Bu.d[1][0];


            Mat Cx2 = (Mat){ .filas=1,.columnas=1,.d={{ 0.0 }} };
            mat_mul(&m->Cd, &Ax, &Cx2);
            {
                double e2 = y_ac - Cx2.d[0][0];
                Mat E2 = (Mat){ .filas=1,.columnas=1,.d={{ e2 }} };
                mat_mul(&m->L, &E2, &Le);
            }

            st->xhat.d[0][0] = Ax.d[0][0] + Le.d[0][0];
            st->xhat.d[1][0] = Ax.d[1][0] + Le.d[1][0];
            break;
    }

    return u_phys;
}
```

Listing 9. `control.c`: cálculo de la ley de control.

```
#ifndef CONTROL_H
#define CONTROL_H

#include <stdint.h>
#include "mat.h"


typedef enum {
    CL_PREDICTOR = 0,
    CL_CURRENT   = 1
} control_law_t;


typedef struct {
    Mat Ad;   /* 2x2 */
    Mat Bd;   /* 2x1 */
    Mat Cd;   /* 1x2 */
    Mat K;    /* 1x2 */
    Mat K0;   /* 1x1  (prefiltro / Nbar / K0) */
    Mat L;    /* 2x1  (ganancias del observador) */
} control_model_t;

/* Estado interno del estimador */
typedef struct {
    Mat xhat; /* 2x1 */
} control_state_t;

/* Inicializa el estado del estimador en cero */
void control_init(control_state_t *st);


float control_step(control_state_t *st,
                   const control_model_t *m,
                   float ref_ac,
                   float y_volts,
                   uint8_t mode,
                   control_law_t law);

#endif /* CONTROL_H */
```

Listing 10. `control.h`: prototipos y estructuras del módulo de control.

```
#include "project.h"
#include "config.h"
```

```c
#include "io.h"

uint8_t io_volt_to_dac(float v)
{
    if (v < SAT_MIN) v = SAT_MIN;
    if (v > SAT_MAX) v = SAT_MAX;
    float norm = v / SAT_MAX;      /* 0..1 */
    if (norm < 0.0f) norm = 0.0f;
    if (norm > 1.0f) norm = 1.0f;
    return (uint8_t)(norm * 255.0f);
}

void io_init(void)
{
    Opa_dac_Start();
    Opa_stage1_Start();
    Opa_vdda_2_Start();
    Opa_stage2_Start();
    PGA_ref_Start();

    VDAC8_Start();
    VDAC8_ref_Start();

    VADC_Start();
}

float io_adc_read_volts(void)
{
    /* Bloqueante corto (ya está convertido por el timer/ISR) */
    return (float)VADC_CountsTo_Volts(VADC_GetResult16());
}

void io_dac_write(float volts)
{
    VDAC8_SetValue(io_volt_to_dac(volts));
}

void io_ref_write(float volts)
{
    VDAC8_ref_SetValue(io_volt_to_dac(volts));
}
```

Listing 11. `io.c`: funciones de entrada/salida (ADC, DAC, UART).

```c
#ifndef IO_H
#define IO_H
#include <stdint.h>

/* Init de periféricos HW */
void io_init(void);

/* ADC -> voltios */
float io_adc_read_volts(void);

/* DAC principal: esfuerzo */
void io_dac_write(float volts);

/* DAC auxiliar: referencia visual al scope */
void io_ref_write(float volts);

/* Map 0..SAT_MAX -> 0..255 con clamps */
uint8_t io_volt_to_dac(float v);

#endif
```

Listing 12. `io.h`: prototipos de E/S.

```c
#include "mat.h"


void mat_zero_volatile(volatile Mat *M, int r, int c) {
    M->filas = r; M->columnas = c;
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            M->d[i][j] = 0.0;
}


// ===== Implementaciones =====
void mat_zero(Mat *M, int r, int c) {
    M->filas = r; M->columnas = c;
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            M->d[i][j] = 0.0;
}

void mat_copy(Mat *dst, const Mat *src) {
    dst->filas = src->filas; dst->columnas = src->columnas;
    for (int i = 0; i < src->filas; i++)
        for (int j = 0; j < src->columnas; j++)
            dst->d[i][j] = src->d[i][j];
}


void mat_copy_volatile(Mat *dst, volatile Mat *src) {
    dst->filas = src->filas; dst->columnas = src->columnas;
    for (int i = 0; i < src->filas; i++)
        for (int j = 0; j < src->columnas; j++)
            dst->d[i][j] = src->d[i][j];
}

void mat_add(const Mat *A, const Mat *B, Mat *C) {
    C->filas = A->filas; C->columnas = A->columnas;
    for (int i = 0; i < A->filas; i++)
        for (int j = 0; j < A->columnas; j++)
            C->d[i][j] = A->d[i][j] + B->d[i][j];
}

void mat_sut(const Mat *A, const Mat *B, Mat *C) {
    C->filas = A->filas; C->columnas = A->columnas;
    for (int i = 0; i < A->filas; i++)
        for (int j = 0; j < A->columnas; j++)
            C->d[i][j] = A->d[i][j] - B->d[i][j];
}

void mat_mul(const Mat *A, const Mat *B, Mat *C) {
    C->filas = A->filas; C->columnas = B->columnas;
    for (int i = 0; i < A->filas; i++) {
        for (int j = 0; j < B->columnas; j++) {
            double acc = 0.0;
            for (int k = 0; k < A->columnas; k++)
                acc += A->d[i][k] * B->d[k][j];
            C->d[i][j] = acc;
        }
    }
}

void ss_step(const Mat *A, const Mat *B, const Mat *C, const Mat *D,
             const Mat *xk, const Mat *uk, Mat *xk1, Mat *yk)
{
    Mat Ax, Bu, Cx, Du;
    mat_mul(A, xk, &Ax);
    mat_mul(B, uk, &Bu);
```

```
    mat_add(&Ax, &Bu, xk1);

    mat_mul(C, xk, &Cx);
    mat_mul(D, uk, &Du);
    mat_add(&Cx, &Du, yk);
}
```

Listing 13. `mat.c`: operaciones matriciales.

```
#ifndef MAT_H
#define MAT_H

#define MAX 10

// ===== Estructura de Matriz =====
typedef struct {
    int filas;
    int columnas;
    double d[MAX][MAX];
} Mat;

// ===== Prototipos =====
void mat_zero_volatile(volatile Mat *M, int r, int c);
void mat_zero(Mat *M, int r, int c);
void mat_copy(Mat *dst, const Mat *src);
void mat_copy_volatile(Mat *dst, volatile Mat *src);
void mat_add(const Mat *A, const Mat *B, Mat *C);
void mat_sut(const Mat *A, const Mat *B, Mat *C);
void mat_mul(const Mat *A, const Mat *B, Mat *C);
void ss_step(const Mat *A, const Mat *B, const Mat *C, const Mat *D,
             const Mat *xk, const Mat *uk, Mat *xk1, Mat *yk);

#endif // MAT_H
```

Listing 14. `mat.h`: definición de `Mat` y prototipos.

```
#include "project.h"
#include "config.h"
#include "serial.h"
#include <string.h>
#include <ctype.h>
#include <stdio.h>

static serial_line_cb_t g_on_line = 0;
static char rx_buf[RX_BUF_SZ];
static uint8_t rx_len = 0;

void serial_init(serial_line_cb_t on_line)
{
    g_on_line = on_line;
    PC_Start();
    PC_PutString(
        "\r\nReady. Commands:\r\n"
        "  a:<float 0..1.5>     amplitud AC (admite 0,6 o 0.6)\r\n"
        "  p:<uint16>           periodo timer\r\n"
        "  m:<open|closed|1|0>  modo\r\n"
        "  s                    status\r\n"
    );
}



void serial_puts(const char *s) { PC_PutString(s); }

/* sane \r\n handling + overflow protection */
void serial_poll(void)
{
```

```
    int ch;
    while ((ch = PC_GetChar()) != 0) {
        if (ch == '\r' || ch == '\n') {
            if (rx_len > 0) {
                rx_buf[rx_len] = '\0';
                if (g_on_line) g_on_line(rx_buf);
                rx_len = 0;
            }
            /* colapsar múltiples \r\n consecutivos */
        } else {
            if (rx_len < (RX_BUF_SZ - 1)) {
                rx_buf[rx_len++] = (char)ch;
            } else {
                rx_len = 0;
                PC_PutString("ERR: line too long\r\n");
            }
        }
    }
}
```

Listing 15. `serial.c`: funciones de comunicación serie.

```
#ifndef SERIAL_H
#define SERIAL_H
#include <stdint.h>
#include <stddef.h>

typedef void (*serial_line_cb_t)(const char *line);

/* Inicializa UART PC y registra callback por línea */
void serial_init(serial_line_cb_t on_line);

/* Poll no-bloqueante: drena RX, arma líneas y llama callback */
void serial_poll(void);

/* Envío utilitario */
void serial_puts(const char *s);

#endif
```

Listing 16. `serial.h`: prototipos del módulo serie.

## E. Código en *MATLAB*

```
clear all; clc; close all;
addpath('..\');
%% --- Planta continua (tu modelo) ---
C2 = 103.07e-9;  C1 = 211.1e-9;
R3 = 6.8e3;    R4 = 14.760e3;
R1 = 47e3;     R2 = 81.09e3;

tau1 = R2*C1;            k1 = -R2/R1;
tau2 = R4*C2;            k2 = -R4/R3;

F = [ -1/tau1,      0;
       k2/tau2, -1/tau2 ];
G = [ k1/tau1; 0 ];
H = [0 1];
J = 0;

sysC = ss(F,G,H,J);
Gc   = tf(sysC);

% Ts base vía polo más rápido continuo (heurística)
fn = max(abs(zpk(Gc).P{1}))/pi;
Tn = 1/fn;
```

```
Ts = Tn/4;              % un único Ts; ajustá si querés

sysD = c2d(sysC, Ts, 'zoh');
[A,B,C,D] = ssdata(sysD);

%% --- Chequeo de controlabilidad/observabilidad, como pediste ---
Co = ctrb(A,B);
Ob = obsv(A,C);
rCo = rank(Co);
rOb = rank(Ob);
n   = size(A,1);

fprintf('rank(ctrb) = %d, n = %d\n', rCo, n);
fprintf('rank(obsv) = %d, n = %d\n', rOb, n);

%% --- Polos objetivo en z (únicos y fijos) ---
p_ctrl = [0.8 + 1j*0.2; 0.8 - 1j*0.2];   % TU pedido

% Ganancia de realimentación K por Ackermann
K = acker(A,B,p_ctrl);

% Prefiltro Nbar (para y_ss = r con D=0, SISO)

[~,~,Nbar] = refi(A,B,C,K);

%% --- "N veces más rápido" para el observador en discreto ---
p_obs  = 0.2+1j*0.2;         % eleva cada polo


p_obs = [p_obs(1), conj(p_obs(1))];

% Ganancia del observador por dualidad (esta es la forma correcta):
L_pred = acker(A', C', p_obs).' ;   % para estimador de PREDICCIÓN
L_pred_a = acker(A',(C*A)',p_obs).'; % para estimador ACTUAL



%% ================== SIMULACIÓN Y PLOTS (loop + función) ==================
% Parámetros de simulación
N = 50;
M = N-10;
Amp = 1.5;
Tsim = 0:Ts:(2*N-1)*Ts;
r    = Amp/2*[-ones(1,N),ones(1,N)];
x0   = Amp/2*[-1; 1];
xh0  = Amp/2*[-1.5; 1.5];

% Sistemas para polos
sys_open = ss(A,B,C,D,Ts);
Acl      = A - B*K;
sys_cl   = ss(Acl, B, C, D, Ts);

% --- después de simular ambos:
simPred = sim_predictor(A,B,C,D,K,L_pred,   Nbar,Ts,Tsim,x0,xh0,r);
simPred.title = "Estimador Predictivo.";
simCurr = sim_current  (A,B,C,D,K,L_pred_a, Nbar,Ts,Tsim,x0,xh0,r);
simCurr.title = "Estimador Actual.";
P_open     = eig(A);
P_cl       = eig(A - B*K);
r =r +2.024;
sim = {simPred, simCurr};
for i = 1:numel(sim)
    fancy_plot_states_control( ...
        Tsim, ...
        sim{i}.X, sim{i}.Xhat, sim{i}.U, r, ...
        P_open, P_cl, sim{i}.P, sim{i}.title);
```

```
end


fancy_plot_states_control_combo( ...
    Tsim, ...
    simPred.X, simCurr.X, ...
    simPred.Xhat, simCurr.Xhat, ...
    simPred.U, simCurr.U, r, ...
    P_open, P_cl, simPred.P, simCurr.P, ...
    'Predicción vs Actual');

%% ======================== FUNCIONES AUXILIARES ============================
function out = sim_predictor(A,B,C,D,K,L,Nbar,Ts,T, x0,xh0,r)
    nx = size(A,1);  N = numel(T);
    x  = x0(:);      xh = xh0(:);
    X  = zeros(nx,N); XH = zeros(nx,N); U = zeros(1,N);

    for k = 1:N
        rk = r(k);

        % 1) Control con estado estimado a k: u[k]
        u = Nbar*rk - K*xh;

        % 2) Medición a k (ANTES de propagar x): y[k]
        y = C*x + D*u;         % si D=0, no cambia nada

        % 3) Observador de PREDICCIÓN: usa y[k]
        xh = A*xh + B*u + L*(y - C*xh);

        % 4) Planta real a k+1
        x  = A*x + B*u;

        % 5) Log
        X(:,k)  = x;      % estado real en k+1 (está bien si eres consistente)
        XH(:,k) = xh;     % estimado en k+1
        U(k)    = u;
    end

    out.t = T;
    out.X = X+2.024; out.Xhat = XH+2.024; out.U = U+2.024;
    out.P = eig(A - L*C);    % dinámicas del error
end



function out = sim_current(A,B,C,D,K,Ke,Nbar,Ts,T, x0,xh0,r)
% Estimador ACTUAL (current): corrige con y[k+1] sobre la predicción z[k+1]
    nx = size(A,1);
    N  = numel(T);

    x  = x0(:);       % estado real
    xh = xh0(:);      % estado estimado

    X  = zeros(nx,N);
    XH = zeros(nx,N);
    U  = zeros(1, N);

    for k = 1:N
        rk = r(k);

        % 1) control con ESTADO ACTUAL (corregido a k): u[k]
        u   = Nbar*rk - K*xh;

        % 2) planta a k+1
        x   = A*x + B*u;

```

```
        % 3) predicción del estimador a k+1
        z   = A*xh + B*u;
        y = C*x + D*u;              % (en tu sistema D=0)

        % corrección actual (usa y[k+1])
        xh = z + Ke*(y - C*z);

        % log
        X(:,k)  = x;
        XH(:,k) = xh;
        U(k)    = u;
    end

    out.t = T; out.X = X+2.024; out.Xhat = XH+2.024; out.U = U+2.024; out.P = eig(A - Ke*C*A);


end

function fancy_plot_states_control( ...
    t, X, Xhat, U, r, ...
    Popen, Pcl, Pobs, titulo)

    n  = size(X,1);
    Ts = t(2) - t(1);

    % Colores
    c_x     = [0 0.4470 0.7410];           % azul
    c_xhat  = [0.82 0.91 0.96];            % celeste claro
    c_ref   = [0.3 0.3 0.3];               % gris
    c_err   = [0.2 0.6 0.2];               % verde para error

    fig   = figure('Name', titulo, 'Color','w');
    left  = uipanel(fig,'Position',[0.05 0.08 0.58 0.87]);
    right = axes   (fig,'Position',[0.68 0.10 0.28 0.82]);  %#ok<LAXES>

    tl = tiledlayout(left, n+1, 1, 'TileSpacing','compact','Padding','compact');

    % ====== Estados (x, xhat) + referencia y error e = x - xhat (eje derecho) ======
    for k = 1:n
        ax = nexttile(tl); hold(ax,'on');

        % Izquierda: señales principales
        yyaxis(ax,'left');
        hX    = plot(ax, t, X(k,:),     '-', 'LineWidth', 1.6, 'Color', c_x,    'DisplayName', '$x$');
        hXhat = stairs(ax, t, Xhat(k,:),  "pentagram", 'LineWidth', 1.2, 'Color', c_xhat, 'DisplayName', '$\hat{x}$');
        hR    = stairs(ax, t, r,          ':', 'LineWidth', 1.0, 'Color', c_ref,  'DisplayName', '$r$');
        ylabel(ax, sprintf('x_%d',k));
        grid(ax,'on'); grid(ax,'minor');

        % Derecha: error
        yyaxis(ax,'right');
        e = X(k,:) - Xhat(k,:);
        hE = stairs(ax, t, e, '.', 'LineWidth', 1.2, 'Color', c_err, 'DisplayName', '$e$');
        yline(ax, 0, ':', 'Color',[0.5 0.5 0.5], 'HandleVisibility','off');
        ylabel(ax, sprintf('e_%d',k));

        if k==1
            title(ax, titulo);
        end
        if k==n
            % Creamos leyenda con handles explícitos y la bloqueamos
            yyaxis(ax,'left');
            lg = legend(ax, [hX, hXhat, hR, hE], {'$x$','$\hat{x}$','$r$','$e$'}, 'Location', 'best');
```

```
            set(lg,'Interpreter','latex','AutoUpdate','off');
        end
    end

    % ====== Control U + r ======
    axu = nexttile(tl); hold(axu,'on');
    hU = stairs(axu, t, U, '-', 'LineWidth', 1.4, 'Color', c_x,   'DisplayName', '$u$');
    hR = stairs(axu, t, r, ':', 'LineWidth', 1.0, 'Color', c_ref, 'DisplayName', '$r$');
    grid(axu,'on'); grid(axu,'minor');
    ylabel(axu,'u'); xlabel(axu,'Tiempo [s]');
    title(axu,'Esfuerzo de control');
    lg2 = legend(axu, [hU, hR], {'$u$','$r$'}, 'Location','best');
    set(lg2,'Interpreter','latex','AutoUpdate','off');

    % ====== Polos (zgrid, sin legend) ======
    axes(right); cla(right); hold(right,'on');
    zgrid; axis(right,'equal'); axis(right,[-1.1 1.1 -1.1 1.1]);
    xlabel(right,'Re(z)'); ylabel(right,'Im(z)');
    title(right, sprintf('Polos -- f_s = %.0f Hz', 1/Ts));

    % Ejes en Re=0 e Im=0 (NO entran al legend)
    xline(right, 0, '-', 'Color',[0.5 0.5 0.5],'LineWidth',0.8,'HandleVisibility','off');
    yline(right, 0, '-', 'Color',[0.5 0.5 0.5],'LineWidth',0.8,'HandleVisibility','off');

    plot(right, real(Popen), imag(Popen), 's', 'MarkerSize',7, 'LineWidth',1.2, 'Color',[0.6
    0.6 0.6]);
    plot(right, real(Pcl),   imag(Pcl),   'x', 'MarkerSize',9, 'LineWidth',1.6, 'Color',[0
    0.4470 0.7410]);
    plot(right, real(Pobs),  imag(Pobs),  'o', 'MarkerSize',7, 'LineWidth',1.4, 'Color'
    ,[0.4940 0.1840 0.5560]);
end


function fancy_plot_states_control_combo( ...
    t, X_pred, X_act, Xhat_pred, Xhat_act, U_pred, U_act, r, ...
    Popen, Pcl, Pobs_pred, Pobs_act, titulo)

    n  = size(X_pred,1);
    Ts = t(2) - t(1);

    % Colores consistentes
    c_pred     = [0 0.4470 0.7410];       % azul
    c_act      = [0.70 0.00 0.00];        % rojo oscuro
    c_xhatpred = [0.8500 0.3250 0.0980]; % naranja
    c_xhatact  = [0.4940 0.1840 0.5560]; % púrpura
    c_ref      = [0.3 0.3 0.3];
    c_open     = [0.6 0.6 0.6];
    c_errpred  = [0.2 0.6 0.2];          % verde error pred
    c_erract   = [0.0 0.6 0.6];          % cian  error act

    fig   = figure('Name', titulo, 'Color','w');
    left  = uipanel(fig,'Position',[0.05 0.08 0.58 0.87]);
    right = axes   (fig,'Position',[0.68 0.10 0.28 0.82]); %#ok<LAXES>
    tl = tiledlayout(left, n+1, 1, 'TileSpacing','compact','Padding','compact');

    % ====== Estados + errores (ejes duales) ======
    for k = 1:n
        ax = nexttile(tl); hold(ax,'on');

        % Izquierda: señales principales
        yyaxis(ax,'left');
        hXp   = plot(ax, t, X_pred(k,:),   '-', 'LineWidth', 1.6, 'Color', c_pred,     '
    DisplayName', '$x_{\mathrm{pred}}$');
        hXa   = plot(ax, t, X_act(k,:),    '-', 'LineWidth', 1.2, 'Color', c_act,      '
    DisplayName', '$x_{\mathrm{act}}$');
        hXhp  = stairs(ax, t, Xhat_pred(k,:),"pentagram", 'LineWidth', 1.2, 'Color',
    c_xhatpred, 'DisplayName', '$\hat{x}_{\mathrm{pred}}$');
```

```
        hXha  = stairs(ax, t, Xhat_act(k,:), "pentagram", 'LineWidth', 1.2, 'Color', c_xhatact,  'DisplayName', '$\hat{x}_{\mathrm{act}}$');
        hR    = stairs(ax, t, r,             ':', 'LineWidth', 1.0, 'Color', c_ref,      'DisplayName', '$r$');
        ylabel(ax, sprintf('x_%d',k));
        grid(ax,'on'); grid(ax,'minor');

        % Derecha: errores
        yyaxis(ax,'right');
        e_pred = X_pred(k,:) - Xhat_pred(k,:);
        e_act  = X_act(k,:)  - Xhat_act(k,:);
        hEp = stairs(ax, t, e_pred, '--', 'LineWidth', 1.0, 'Color', c_errpred, 'DisplayName', '$e_{\mathrm{pred}}$');
        hEa = stairs(ax, t, e_act,  '--', 'LineWidth', 1.0, 'Color', c_erract,  'DisplayName', '$e_{\mathrm{act}}$');
        yline(ax, 0, ':', 'Color',[0.5 0.5 0.5], 'HandleVisibility','off');
        ylabel(ax, sprintf('e_%d',k));

        if k==1
            title(ax, sprintf('%s -- Estados y estimaciones', titulo));
        end
        if k==n
            % Leyenda con handles explícitos y bloqueada
            yyaxis(ax,'left');
            lg = legend(ax, [hXp, hXa, hXhp, hXha, hR, hEp, hEa], ...
                {'$x_{\mathrm{pred}}$','$x_{\mathrm{act}}$', ...
                 '$\hat{x}_{\mathrm{pred}}$','$\hat{x}_{\mathrm{act}}$','$r$', ...
                 '$e_{\mathrm{pred}}$','$e_{\mathrm{act}}$'}, ...
                'Location','best');
            set(lg,'Interpreter','latex','AutoUpdate','off');
        end
    end

    % ====== Control ======
    axu = nexttile(tl); hold(axu,'on');
    hUp = stairs(axu, t, U_pred, '-', 'LineWidth', 1.4, 'Color', c_pred, 'DisplayName', '$u_{\mathrm{pred}}$');
    hUa = stairs(axu, t, U_act,  '-', 'LineWidth', 1.2, 'Color', c_act,  'DisplayName', '$u_{\mathrm{act}}$');
    hR  = stairs(axu, t, r,      ':', 'LineWidth', 1.0, 'Color', c_ref,  'DisplayName', '$r$');

    grid(axu,'on'); grid(axu,'minor'); ylabel(axu,'u'); xlabel(axu,'Tiempo [s]');
    title(axu,'Esfuerzo de control');
    lg2 = legend(axu, [hUp, hUa, hR], {'$u_{\mathrm{pred}}$','$u_{\mathrm{act}}$','$r$'}, 'Location','best');
    set(lg2,'Interpreter','latex','AutoUpdate','off');

    % ====== Polos (sin legend) ======
    axes(right); cla(right); hold(right,'on');
    zgrid; axis(right,'equal'); axis(right,[-1.1 1.1 -1.1 1.1]); xlabel(right,'Re(z)');
    ylabel(right,'Im(z)');
    title(right, sprintf('Polos -- f_s = %.0f Hz', 1/Ts));

    % Ejes en Re=0 e Im=0 (no ensucian leyenda)
    xline(right, 0, '-', 'Color',[0.5 0.5 0.5],'LineWidth',0.8,'HandleVisibility','off');
    yline(right, 0, '-', 'Color',[0.5 0.5 0.5],'LineWidth',0.8,'HandleVisibility','off');

    plot(right, real(Popen),     imag(Popen),     's', 'MarkerSize',7, 'LineWidth',1.2, 'Color', c_open);
    plot(right, real(Pcl),       imag(Pcl),       'x', 'MarkerSize',9, 'LineWidth',1.6, 'Color', c_pred);
    plot(right, real(Pobs_pred), imag(Pobs_pred), 'o', 'MarkerSize',7, 'LineWidth',1.4, 'Color', c_xhatpred);
    plot(right, real(Pobs_act),  imag(Pobs_act),  '^', 'MarkerSize',7, 'LineWidth',1.4, 'Color', c_xhatact);
end
```

Listing 17. `HojaDeCalculos.m`: cálculo de parámetros y generación de figuras.