Linear Controllers

From Design to Implementation

Dr.Varodom Toochinda

http://www.controlsystemslab.com

July 2009

Linear control analysis and design has been an active subject for some time, perhaps since WW II. Among all techniques and approaches invented, many have become standards in control engineering courses, say, Bode and Nyquist plots, while others may have only aesthetic values. Now humankinds reach an era where digital systems dominate our world. Few people, if any, would want to use an analog PID controller constructed from operational amplifiers. An analog control circuit is inflexible, consumes more space, and its operating point depends on factors like temperature and component aging. A digital controller could overcome such disadvantages, assuming that it is designed and implemented properly.

Crafting a decent digital controller involves many steps with a lot of details. This document is by no means complete. We hope it could serve as a guideline for you, to delve more into this fascinating field by yourself.

Necessary steps can be summarized as follows:

1. Initial Study: get basic knowledge of the system under control (normally called a "plant")
2. Strategy Select: choose the control scheme. For PID control, you can go to step 6
3. Modeling: find a suitable math model for the plant, either from physics or system identification (preferred) , and verify that the model is good enough.
4. Analysis & Design: use CAD software tools to get a controller, evaluate stability and performance.
5. Simulation: see if the controller satisfies your needs. If not, go back to 4
6. Implementation: discretize the controller and program the target system

In this document we discuss only linear control and emphasize on implementation procedure. It is helpful, though, to provide a brief introduction to the other steps above.

Pre-design Phase

Independent from the control scheme used, knowledge of the plant is always useful. At least we must have basic understanding on important issues, such as, what are the inputs and outputs? which ones could be used by the controller? Some parameters could affect the system more than others. Factors like sensor placement could also help or hurt. Note that, depending on system complexity, this kind of information may not be trivial to find.

After the initial study, we need to decide on the type of controller. A Proportional, Integral, Derivative (PID) controller is very convenient to use, since it requires no knowledge on plant model. Users simply adjust the 3 PID gains to achieve the desired controlled response. The majority of industrial controllers nowadays are variants of the PID type. Despite the ease of implementation and operation, PID controllers have limitations. Basically they have 2^{nd} order dynamics, which may not be adequate to compensate a high-order plant. This document omits the discussion of PID control.

For a more complex plant, therefore, we may prefer to craft a linear controller for it. This usually involves a few iterations of analysis, design, and simulation, before the controller is acceptable for implementation. Before we can do that, knowledge of the plant in a form software tools could understand must be available. By this we refer to a mathematical model that accurately represents the real plant dynamics. Of course we can never achieve a perfect match. How close a math model to the real plant dictates the quality of the resulting controller, since it is designed to compensate the dynamics of the model.

There are vast numbers of modeling scheme purposed in the literature, though they can be classified to 2 main types: using laws from physics, or using system identification (often abbreviated as Sys ID). In the first type we derive an equation from the underlying physics, and then try to measure the parameters in that equation from the real system. This could work fine only for a simple plant, such as a brushed DC motor. For an brushless AC servomotor with microcontroller drive, in contrast, we cannot construct using knowledge from physics a differential equation that could capture all the dynamics precisely. That is why the Sys ID approach is preferred for more complex plants.

Several Sys ID methods have been used successfully in practice. From our experience we prefer the LS (Least-Square) and swept-sin methods. Since modeling is not the essence of this document, we will leave it for future documents.

Control Analysis, Design, and Simulation

The analysis and design procedures normally go together. In the long past they were done by hand, like plotting Bode graphs on paper. Nowadays, though many standard textbooks still discuss hand-plotting Bode, nobody wants to do it anymore. Integrated CAD software like MATLAB/Simulink (www.mathworks.com) has functions for all of these procedures, plus the simulation feature that was never heard of during B.C (Before Computer) era. Open-source gurus would opt for Scilab , which can be downloaded from www.scilab.org . Of course, knowledge of the underlying theories is still essential in verifying the results.

A simple feedback control diagram is shown in Figure 1. C and P represent the controller and plant, respectively. The input to C is the error e, the difference between command r and the measured output y_m. Note that y_m may differ from real plant output y due to the sensor noise n. Moreover, we may have some disturbance d in certain parts of the system, say, at the plant output like shown in Figure 1.

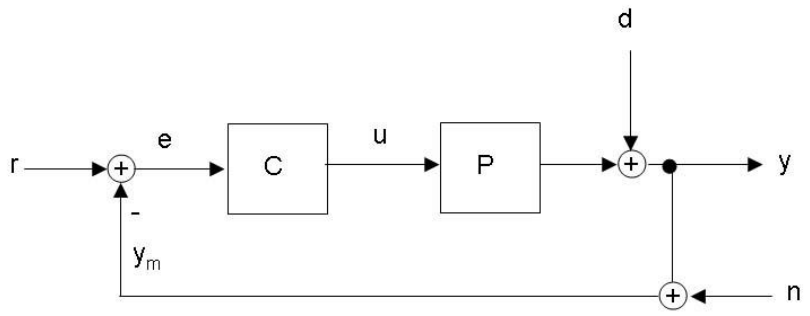


Figure 1: A feedback control system

Two main properties of feedback control that defines its use in the real world are “stability” and “performance.” Stability is the most important quality, in the sense that unstable systems are not only useless, but it could be catastrophic. One simple way to analyze stability is by plotting the poles of a closed-loop transfer function in complex plane, to see whether it lies in the stability region; i.e., left-half plane for continuous-time, or inside the unit circle for discrete-time.

We do not have stability issue in open-loop control, say, a stepper-motor. Closing the feedback loop could make a system unstable. So why bother? The second quality comes into play now. Adding feedback could improve the performance of an open-loop system. Suppose we command a stepper to rotate 1 degree. The load is too heavy for the motor so the movement falls short 0.1 degree. Without feedback we never know the output does not reach the target value, called the “setpoint.” Hence, performance of a feedback system can be evaluated in time domain from a “unit-step response” like shown in Figure 2.

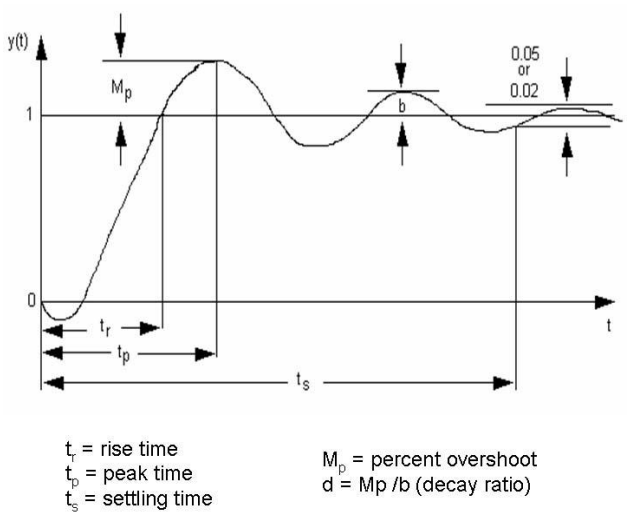


Figure 2: a unit-step response

Note that, even for a PID controller, the step response is often used to adjust the gains that give the desired output, which depends on a particular application. For example, some systems may tolerate certain overshoot but others may not.

Of course, the step response of Figure 2 shows only one type of performance, called "tracking." This represents the ability of plant output in tracking the command signal. There are other types of performance. In Figure 1 we see that the disturbance d and measurement noise n are unwanted signals. We do not want them to affect the plant output. The ability of controller to get rid of d and n are called disturbance and noise attenuation performance. There may be other types of performance specifications, but these are the most basic ones.

The closed-loop performance could be evaluated in the frequency domain using tools like Bode plot. There are relationships between the time and frequency domain responses. Details could be found in most control textbooks.

Now we focus on linear control design. It is often classified to classical and modern, and sometimes we heard the word "post-modern," since they started using the term modern around 1960. The approaches branching from these categories may be quite hard to keep track. We have, among others, the QFT, LQR, LQG/LTR, H_2, H_∞ etc., with their advantages and disadvantages. Later developments try to combat uncertainties and the term "robust control" emerges.

Good news. For many industrial control systems, simple loop-shaping design scheme suffices. Even PID controllers are still used worldwide with good results. Remember that, a controller is as good as a math model of the plant used in the design. So if your model sucks, the most sophisticated robust control scheme could not help with anything.

A useful control design tool in MATLAB Control Toolbox, known as sisotool( ), is worth mentioning. This function has graphical interface and is easy to use. Figure 3 shows the GUI window when you invoke sisotool from MATLAB command line. The window can be customized to your needs. For example, if the root locus display is not helpful, it could be turned off. Loopshaping method can be conveniently performed by sisotool. We can add poles and zeros of our controller to the Bode plot, or drag them along the curve, and see their effects immediately. This leads to a "trial-and-error" design method that is not possible in the past. After we're satisfied with the result, the controller can be exported to the workspace.

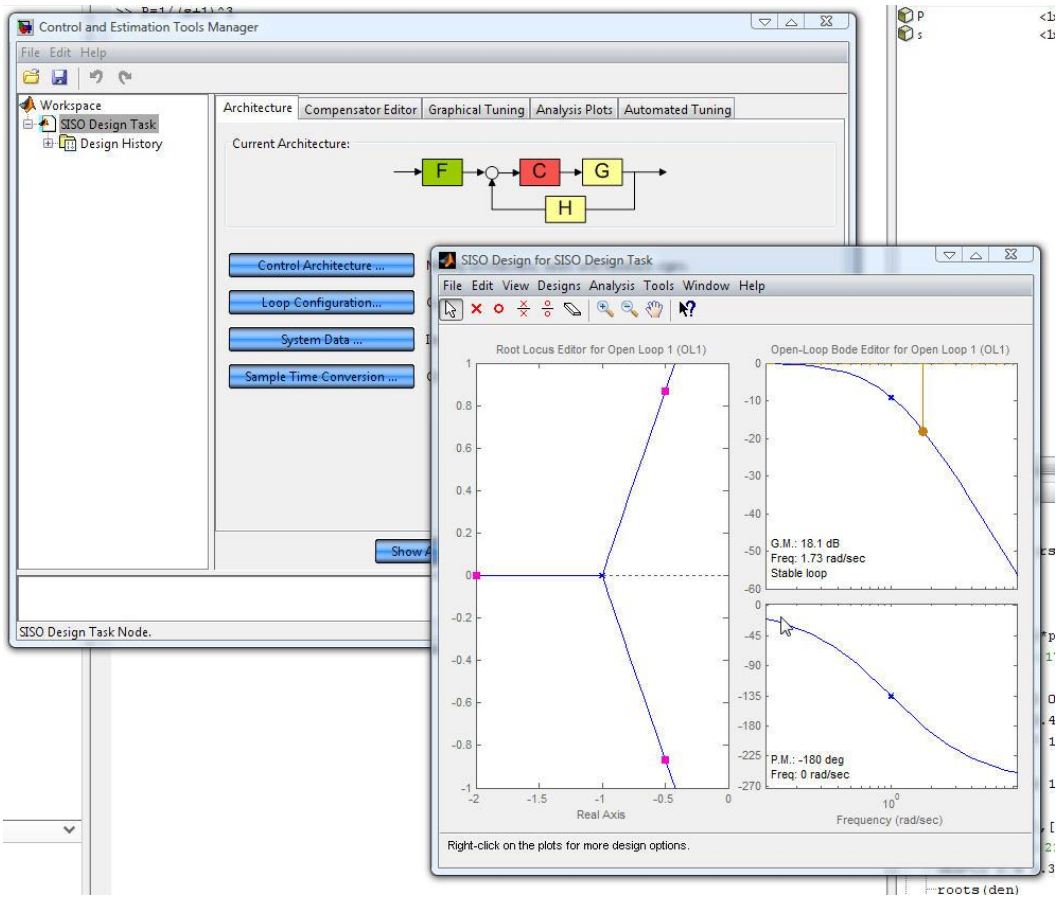


Figure 3: GUI window of sisotool( )

Even though in the end a controller is implemented digitally, the analysis and design of controller is usually done in continuous-time domain. Direct design in discrete-time domain is not prevalent at the time of this writing. So our controller at this stage is either in the form of continuous-time state-space equations, or a transfer function (rational function of complex variable s). We can perform closed-loop simulation using this controller together with the math model of the plant. Disturbance and noise can be injected to certain parts of the feedback

diagram to resemble real work environment. Figure 4 shows a SIMULNK setup of CNC axis under PID control.

The procedure in using open-source software Scilab/Scicos is basically the same. Only the interface and command syntax are different.

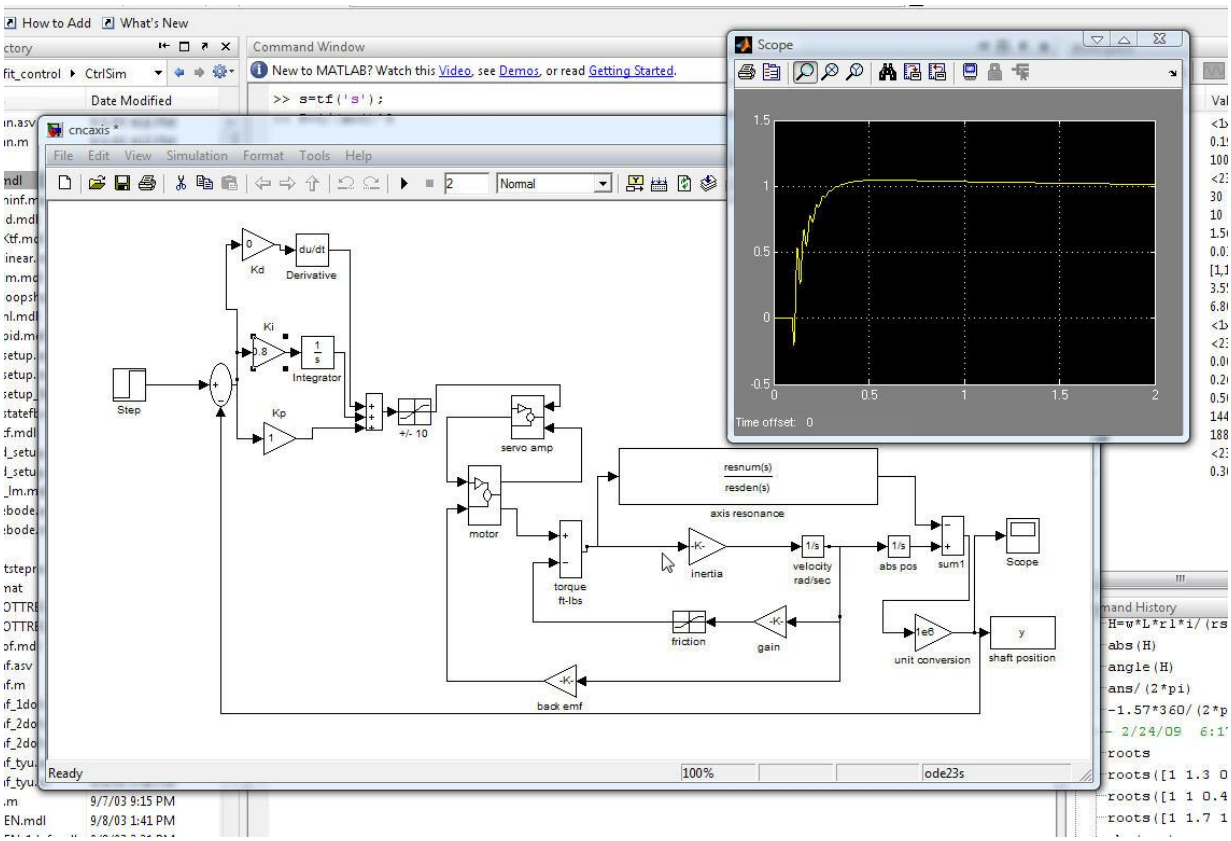


Figure 4: a simulation setup using SIMULINK

Controller Implementation

As stated earlier, most controllers nowadays are implemented digitally. So the feedback diagram in Figure 1 needs to be changed to the one in Figure 5. Now we have two domains separated by the vertical dotted line; i.e., the controller side is discrete, and the plant side is continuous. To connect them together, we need additional components to convert the signals from one form to another. The block labeled DAC (Digital-to-Analog Converter) and ADC (Analog-to-Digital Converter) serve this purpose. Most modern microcontrollers have built-in ADC units, but those with internal DAC's are rare. It is not difficult to find an external DAC chip with easy interface.

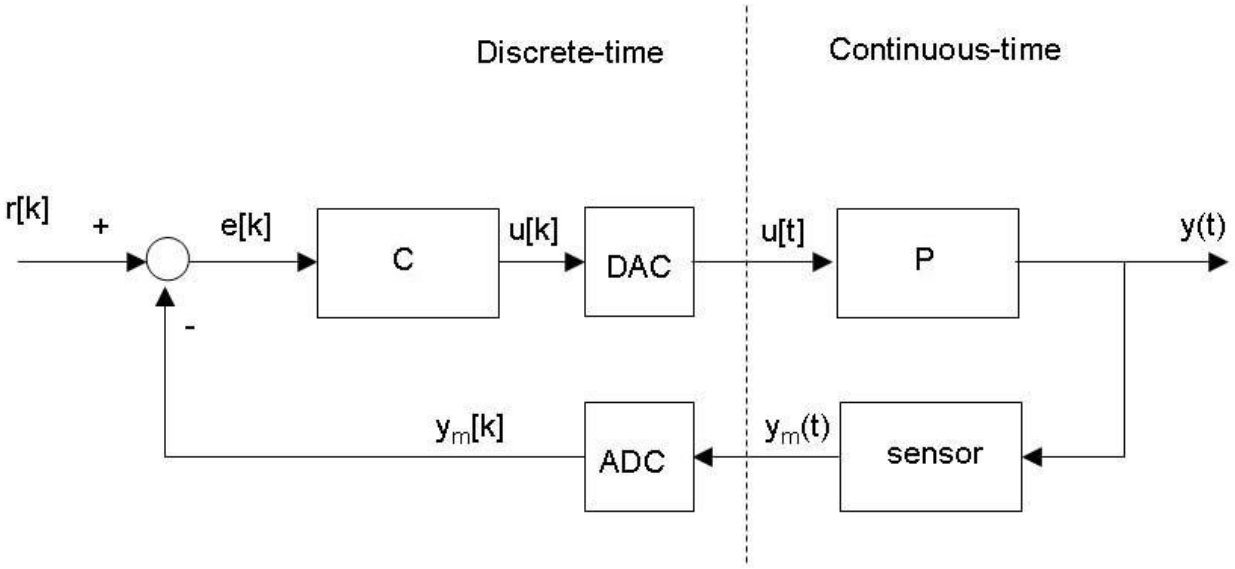


Figure 5: A digital control system

Note that Figure 5 shows a basic configuration. Some systems may differ slightly. For instance, motion control applications usually get feedback from motor encoders, which are digital in nature. In such case the ADC is not required. For some motor amplifier that receives a PWM (Pulse-Width-Modulation) input, the PWM unit then replaces the DAC. Nevertheless, the essence remains. The controller is digital. It is described either by difference equations, or by a Z- transfer function. Before we get into that, an important issue must be addressed.

Sampling an Analog Signal

While a real world signal is continuous, a computer works with values stored in a bank of memory. At one moment in time, a value to be processed is addressed by a data pointer. So, as shown in Figure 6, the continuous signal x(t) has to be sampled into a sequence x[k], where integer k is the index, or the relative address of data to be selected by the pointer. The signal x(t) is sampled at fixed time interval Δ, called the sampling period. (In some areas of engineering such as communication systems, multi-rate sampling may exist, but for digital control applications there is no use for such complicated schemes.)

A question follows naturally. How often should we sample the signal x(t)? Common sense tells us the more frequent, the better, since it is obvious from Figure 6 if the samples x[k] are close together, they should represent x(t) very well. That is valid. But, well, we tend to forget one thing. The more frequent the sampling, the more data we have to keep. In a modern desktop PC

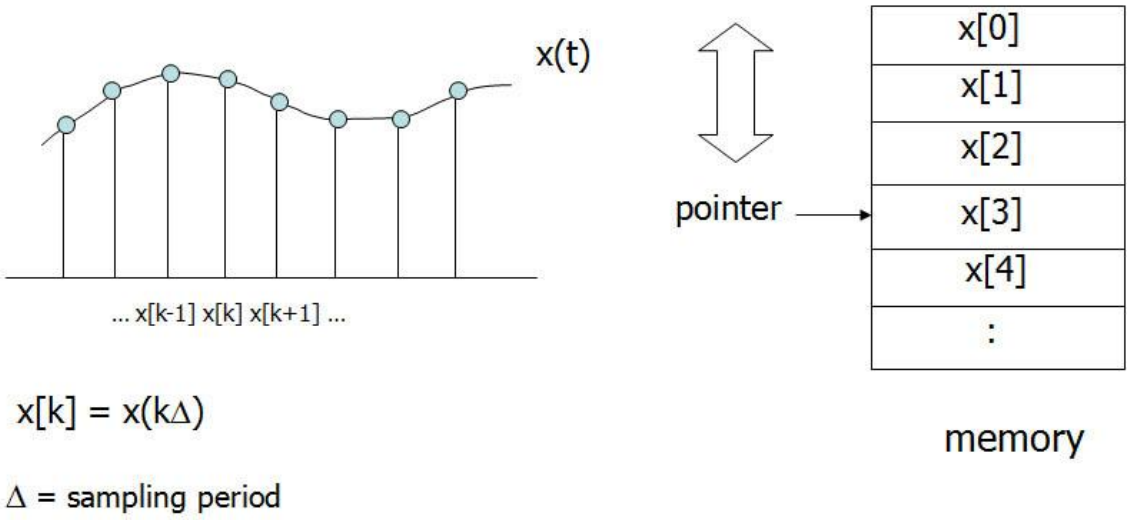


Figure 6: Sampling a continuous signal

this does not sound like a problem. But for embedded control applications, resources are expensive. The memory allocated for data process may be limited.

Let us investigate what happens if the sampling rate is too low. Figure 7 shows such a scenario. The original signal we want to sample is x(t). Undersampling causes the reconstructed signal x'(t) much different from the original. This problem is called aliasing. Another good example of aliasing is when we watch a movie, sometimes we notice a moving car with its wheels turn in the opposite direction. The film is an image sampling system. That happens when the frame rate is too slow relative to the angular velocity of the wheel.

So, how could we select a proper sampling frequency? Intuitively, we can guess from Figure 7 that if we sample at least twice the frequency of the red sine wave, things should work fine. That is in fact an established result, known as the Nyquist-Shannon Sampling Theorem. One could find the details elsewhere. Here we state only the essence, in plain English: the sampling rate must be at least twice the system bandwidth.

This sampling theorem only gives us a lower bound. Practically we would want a higher sampling rate, say, 10 times the system bandwidth. Of course, tradeoffs between sampling rate and amount of data memory need to be considered. Also, if a controller is working in realtime, it has to fetch a data point, do some processing, and output something. We have to make sure the whole algorithm could finish within the sampling time interval.

Figure 8 shows a basic structure for a real-time control algorithm. Notice that it has to be implemented as a timer interrupt service routine to achieve a fixed, precise sampling period.

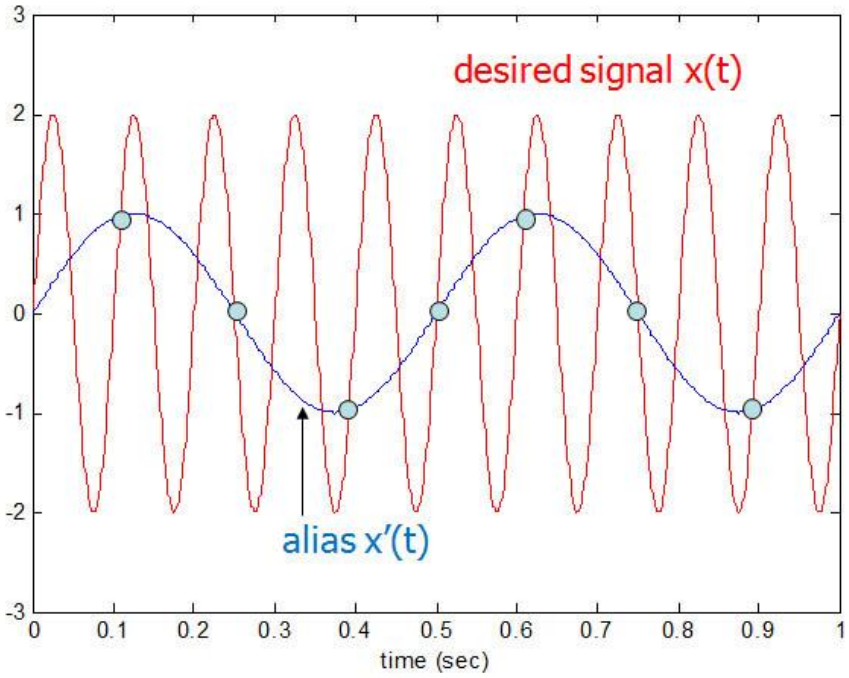


Figure 7 Aliasing problem

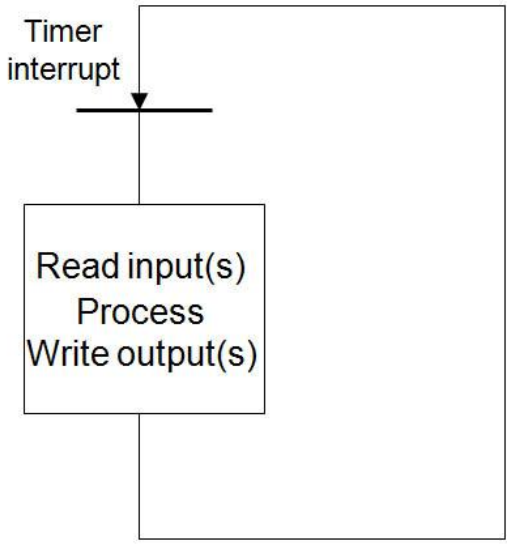


Figure 8: A real-time control algorithm

Discrete System Representation

A continuous-time dynamical system can be represented in the time domain by a differential equation. We can use Laplace transform to find a representation in the frequency domain, called a transfer function. Similar mechanisms exist for discrete-time. A discrete-time dynamical system can be represented in time domain by a difference equation. The math tool to convert it to a transfer function is called Z-transform.

Figure 9 shows 3 basic elements of a discrete system: summer, multiplier, and delay. The first two operators behave the same as in continuous time. The third one is unique to discrete world, but nothing is complicated about it. Output from the D block is simply the input delayed by one sample. In Figure 9, let's say the input to D is e[2], then the output of D equals e[1]. When we convert the system using Z-tranform, what comes out is a rational function of a complex variable z. Without going into the theory, the point to remember is the unit delay D transforms to z^{-1} in the Z-domain.

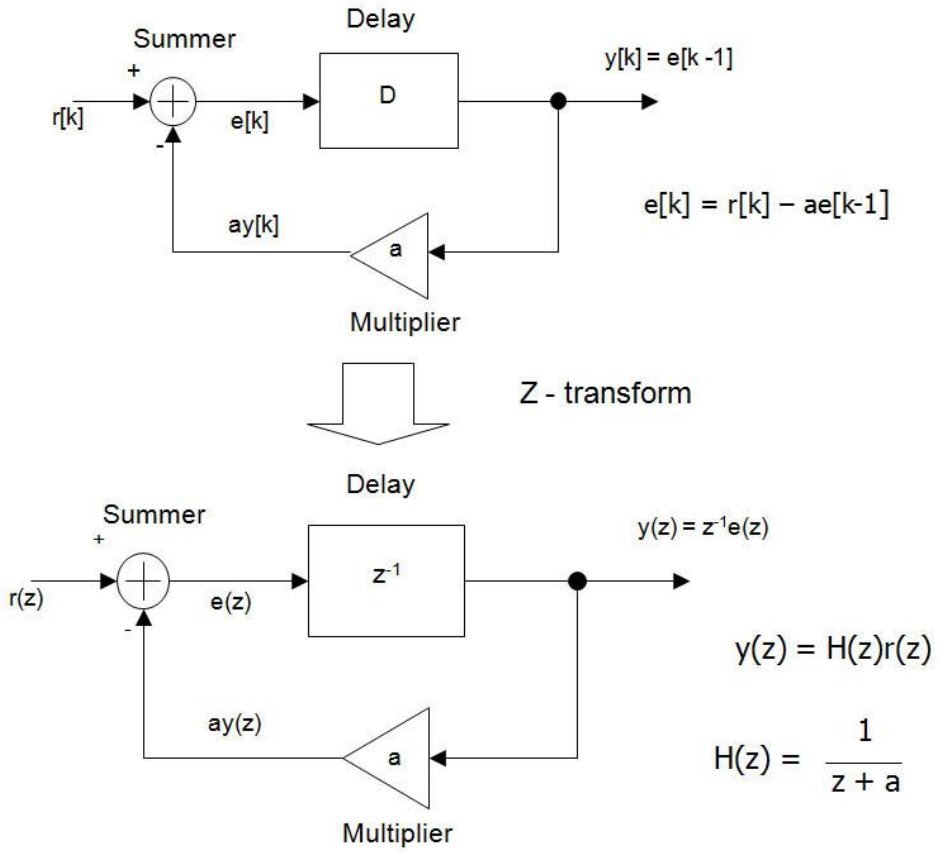


Figure 9: Discrete system representation in time and frequency domain

Remember that the controller achieved in our design is normally in the form of continuous-time transfer function. There exist mathematical relationships to convert it to a discrete-time transfer function. One may want to try a simple one on a paper, for the fun of it. Practically, we 're likely to use the c2d( ) function in MATLAB Control Toolbox. (Type help c2d in the command window to get more information of the function, as well as its arguments.)

Example 1: Suppose from the design, we get this continuous-time controller

$$C(s) = \frac{7000(s+0.1)(s+10)}{(s+2)^2(s+500)}$$

Using c2d( ) with sampling period 1 millisecond, and the Tustin method

```
>> Cnum=7000*conv([1 0.1],[1 10]);

>> Cden=conv(conv([1 2],[1 2]),[1 500]);

>> Csys=tf(Cnum,Cden);

>> Csys_d = c2d(Csys,0.001,'tustin')

 Transfer function:

2.809 z^3 - 2.78 z^2 - 2.809 z + 2.78

-------------------------------------

 z^3 - 2.596 z^2 + 2.194 z - 0.5976

 Sampling time: 0.001

>>
```

Note how the result depends on the sampling interval. The value put in c2d( ) must match the actual interval programmed into the timer of target system.

The transfer function in Example 1 is simple enough for direct implementation. We can easily translate the Z-transfer function to a C algorithm, or whatever computer language you use to program your microcontroller. We will postpone the procedure to a more general case in the next section, when the controller is more complex.

Implementing a high-order controller

A controller designed for a sophisticated system is often a high-order transfer function, since it must compensate for many plant dynamics. Synthesis tools like H_∞ or μ are well-known to give an excessively high-order controller, which may require some model reduction scheme to remove unnecessary dynamics. In the case the final controller is still high-order, we must take into consideration some problems it may impose.

Small embedded computers normally have limited computing performance. It may have only fixed-point math hardware. The precision is in no way compatible with a system with advanced math coprocessor, or a floating-point Digital Signal Processor (DSP). For such embedded processor, small numerical errors could significantly degrade its performance, or, worse, destabilize the system.

In a transfer function, the coefficients of higher-order terms are more sensitive to numerical errors. That is the reason we do not want to implement a high order controller as is. Instead, we factor it to a combination of Second-Order Section (SOS) in series or parallel. Below we discuss only a cascaded SOS scheme, as depicted by Figure 10.

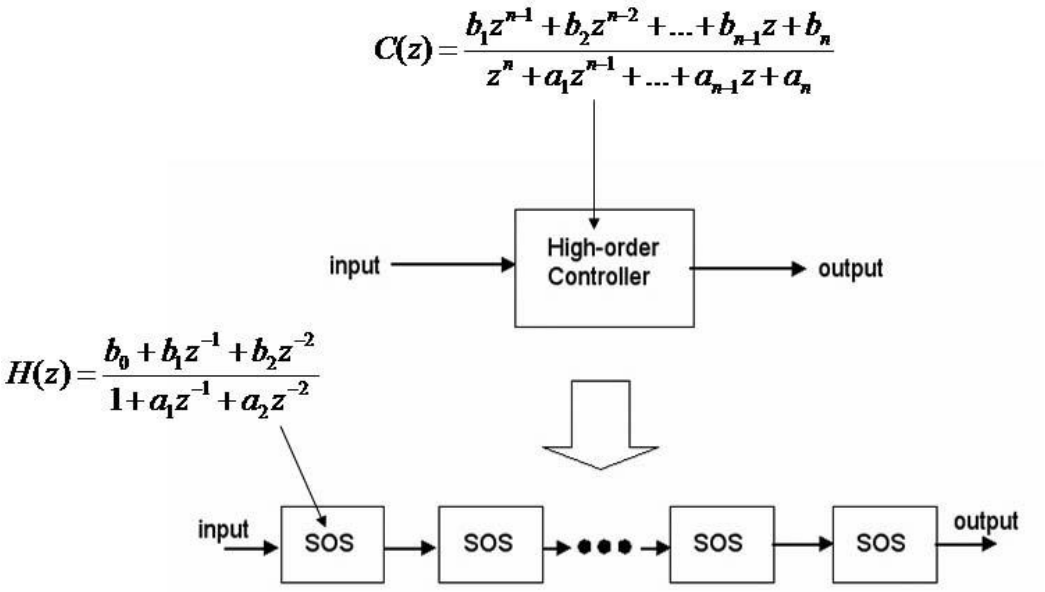


Figure 10: A high-order controller is transformed to series of SOS

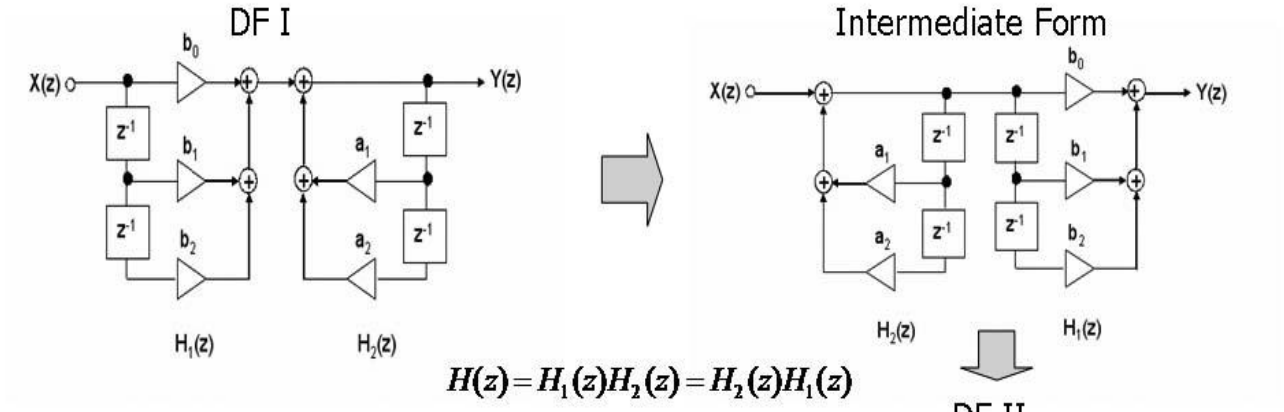
Forms of SOS

After the factorization process above, each SOS can be described as a Z-transfer function

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \qquad (1)$$

A straightforward block diagram representation is shown on the left of Figure 11, called Direct Form One (DF I). This form has direct relationship with H(z) in (1), but it has a drawback. We need 4 delay blocks. A delay is simply a memory location. Less delays mean less memory. Figure 11 shows how we can rearrange DF I to Direct Form Two (DF II), which consumes only 2 delays. Therefore, if we have several SOS's, the data memory space can be reduced in half.

Now we illustrate all implementation procedures in the next example.



$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

$$H(z) = H_1(z) H_2(z)$$

$$H_1(z) = b_0 + b_1 z^{-1} + b_2 z^{-2}$$

$$H_2(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

Figure 11: Rearrange Direct Form I into Direct Form II

Example 2: Linear control design is performed on an optical sight system, similar to the one shown in Figure 12, using H_∞ synthesis method from MATLAB/Robust Control Toolbox. This platform consists of two rotational axes, called azimuth and elevation. Suppose the controller is synthesized for each axis separately. After order reduction, we finally have this controller for the elevation axis

```
>> elcont

 Transfer function:

-1.535e004 s^4 + 7.565e008 s^3 + 9.712e010 s^2 + 6.889e011 s + 2.358e012

-------------------------------------------------------------------------

s^5 + 1947 s^4 + 1.727e006 s^3 + 9.363e008 s^2 + 1.872e008 s + 9.363e006

>>
```

Let us assume we want to implement this controller on a microcontroller with sampling period 500 microseconds. We start as in Example 1 by using c2d( ) to get a discrete-time transfer function.

Figure 12: An Optical Sight Platform (http://www.eo.kollmorgen.com)

```
>> elcont_d=c2d(elcont,0.0005,'tustin')

 Transfer function:

27.94 z^5 - 21.28 z^4 - 65.41 z^3 + 52.1 z^2 + 37.47 z - 30.82

-----------------------------------------------------------

  z^5 - 4.072 z^4 + 6.667 z^3 - 5.493 z^2 + 2.276 z - 0.377


Sampling time: 0.0005

>>
```

The controller order is 5. We want to implement it as 3 SOS's in series. (One SOS is actually first-order, of course.) A MATLAB function, zpkdata( ), could be used to find the poles, zeros, and DC gain of the controller.

```
>> format long e; [Z,P,K]=zpkdata(elcont_d,'v')

Z =

   -1.176176684771430e+000

   -1.000000000000001e+000

    9.981728777558758e-001 +1.756321789181501e-003i

    9.981728777558758e-001 -1.756321789181501e-003i

    9.413861756806828e-001

P =

    7.599502012146209e-001 +3.238128058356590e-001i

    7.599502012146209e-001 -3.238128058356590e-001i

    9.999499999834960e-001 +9.376674489747332e-007i

    9.999499999834960e-001 -9.376674489747332e-007i

    5.525339834535292e-001
```

```
K =

   2.794070182178726e+001

>>
```

From the above data, we can factor the controller

$$C(z) = \frac{27.94(z-0.9414)(z^2+2.1762z+1.1762)(z^2-1.9963z+0.9964)}{(z-0.5525)(z^2-1.5199z+0.6824)(z^2-1.9999z+0.9999)} \quad (2)$$

Listing 1 shows how to implement (2) in DF II, using C

```
#define NUM_SECTIONS      3          // number of sections
double coeff[NUM_SECTIONS*5] =  { // sections are numbered from right to left in
                                  //  (3.20)
        1.9999, -0.999, 1, -1.9963, 0.9964,          // Section 1:  -a1, -a2, b0, b1, b2
        1.5199, -0.6824, 1, 2.1762, 1.1762,          // Section 2
        0.5525, 0,  1, -0.9414, 0 };            // Section 3  (1st order section)
double control_gain = 27.94;    // gain of C(z) in (3.20)
double state[3] = {0,0,0};                  // an array to keep system states
void control(void)    // this algorithm should be implemented as a timer routine with
                      // 500 microseconds  period
{
        double *ptr_coeff, *ptr_state;          // pointers to coefficient and state
                                                // arrays
        double input, output;               // input and output of the controller
        int section_count;                      // count sections
        ptr_coeff = coeff;
```

```
        ptr_state = state;
        input = read_angle( );   // get elevation angle from sensor
        // loop for each secion
        for (section_count = 0; section_count < NUM_SECTIONS; section_count++)  {
                *(ptr_state + 2) = *(ptr_state + 1);              // update states
                *(ptr_state + 1) = *ptr_state;
                *ptr_state = input + *(ptr_coeff)*(*(ptr_state+1))
                                + *(ptr_coeff+1)*(*(ptr_state+2));
                input = *(ptr_coeff+2)*(*(ptr_state++))
                        + *(ptr_coeff+3)*(*(ptr_state++))
                        + *(ptr_coeff+4)*(*(ptr_state++));
                ptr_coeff = ptr_coeff + 5;
        }
        output =  control_gain* input;
        out_DAC(output);       // send output to servo drive
}
```

Listing 1: C implementation of the controller in Example 2

For a more complicated system, doing the SOS factorizations may be tedious. In the appendix, we provide a MATLAB function that performs this task automatically. The function returns a coefficient array that can be downloaded to a target system. (The coefficient format conforms to the one used by DSPLIB of Microchip. You can modify the script as you wish.)

Summary

This document is a survey of linear feedback control systems targeted on an embedded platform. Implementation issues are presented in more detail. The approaches have been used successfully in several research projects in the U.S. and Thailand. However important, we have not covered digital PID implementation. It will be discussed in a separate document.

Appendix

Listing A.1 is a MATLAB function that can be used to construct SOS coefficient arrays automatically. To verify that it works correctly, we consider this discrete transfer function

$$K_d(s) = \frac{-6455z^6 - 1.36e4z^5 - 7374z^4 + 2261z^3 - 4976s^2 + 2536z + 449}{z^6 - 2.22z^5 + 1.73z^4 + z^3 + 0.87z^2 - 0.45z + 0.072} \qquad \text{(A.1)}$$

Performing the factorization by hand, we get

$$K_d(s) = \frac{-6455(z^2 - 0.86z - 0.14)(z^2 + 0.67z + 0.52)(z^2 - 1.91z + 0.98)}{(z^2 - 1.27z + 0.27)(z^2 + 0.53z + 0.48)(z^2 - 1.48z + 0.55)} \qquad \text{(A.2)}$$

Below are outputs captured from MATLAB command window. We can verify that the results conform to (A.2)

```
>>knum = [-5000 -2.46e7 -8.25e10 -1.4e14 -8.82e15 -9.28e18 -9.38e14];
>> kden = [1 3902 1.3e7 1.65e10 6.78e12 9.44e14 2.36e13];

>> Kc = tf(knum,kden);
>> Ts = 0.001;
>> [K,sosnum, sosden, coeff] = ctf2sos(Kc,Ts);
>> coeff'

ans =

  -2.690848255553373e-001
   1.269066549790741e+000
  -1.367651966952970e-001
  -8.632346884032134e-001
   1.000000000000000e+000
  -4.846493241443658e-001
  -5.319226427315764e-001
   5.207230666683245e-001
   6.733055517740321e-001
   1.000000000000000e+000
  -5.548634086032285e-001
   1.481400501053004e+000
   9.767033937092714e-001
  -1.910575160018935e+000
   1.000000000000000e+000
```

```
>> K

K =

  -6.454957138198393e+003
```

```
function  [K,sosnum, sosden, coeff] = ctf2sos(Kc,Ts)
% Dr.Varodom Toochinda        www.dewinz.com
% This function takes a continuous-time transfer function, discretizes it
% and converts to a series of SOS. It can be used with discrete time t.f. by
% specifying negative value for Ts
sosnum=[]; sosden=[]; coeff = [];
if Ts > 0,
     Kd = c2d(Kc,Ts,'tustin');
else Kd = Kc;
format long e;
[Z,P,K] = zpkdata(Kd,'v');

% Numerator part
Zsize = size(Z,1);
if ~Zsize, % no zero
   sosnum = [1];
elseif Zsize == 1,         % only one zero
   sosnum = [0 1 -Z(1)];
else                       % at least 2 zeros
   % seperate real and complex zeros
   Zr = []; Zim = [];
   for i = 1:Zsize,
      if isreal(Z(i)),
         Zr = [Zr; Z(i)];
      else
         Zim = [Zim; Z(i)];
      end
   end
   % take care of real zeros first.
   Zrsize = size(Zr,1);
   if Zrsize,                      % at least two zeros
      if rem(Zrsize,2),            % odd number.
         sosnum = [0 1 -Zr(1)];    % first SOS is first order
         if Zrsize == 1,
            Zr = [];
         else Zr = Zr(2:Zrsize);
         end
```

```
      end
      Zrtsize = size(Zr,1);        % size of Zr after truncation
      if Zrtsize,                  % If not empty, the rest are second-order tf's
         for i = 1:2:Zrtsize,
            sosnum = [sosnum; conv([1 -Zr(i)],[1 -Zr(i+1)])];
         end
      end
   end
   % now complex zeros
   Zimsize = size(Zim, 1);
   if Zimsize, % not empty
      for i = 1:2:size(Zim,1),
         sosnum = [sosnum; conv([1 -Zim(i)],[1 -Zim(i+1)])];
      end
   end
end


% Denominator part
Psize = size(P,1);
if ~Psize, % no pole
   sosden = [1];
elseif Psize == 1,              %only one pole
   sosden = [0 1 -P(1)];
else                            % at least 2 poles
   % seperate real and complex poles
   Pr = []; Pim = [];
   for i = 1:Psize,
      if isreal(P(i)),
         Pr = [Pr; P(i)];
      else
         Pim = [Pim; P(i)];
      end
   end
   % take care of real poles first.
   Prsize = size(Pr,1);
   if Prsize,                          % at least two real poles
      if rem(Prsize,2),                % odd number.
         sosden = [0 1 -Pr(1)];        % first SOS is first order
         if Prsize == 1,
            Pr = [];
         else Pr = Pr(2:Prsize);
         end
      end
      Prtsize = size(Pr,1);        % size of Pr after truncation
```

```
      if (Prtsize),                % If not empty, the rest are second-order tf's
         for i = 1:2:Prtsize,
            sosden = [sosden; conv([1 -Pr(i)],[1 -Pr(i+1)])];
         end
      end
   end
   % now complex poles
   Pimsize = size(Pim, 1);
   if Pimsize, % not empty
      for i = 1:2:size(Pim,1),
         sosden = [sosden; conv([1 -Pim(i)],[1 -Pim(i+1)])];
      end
   end
end

 % Generate coefficients for controller in a format specfied by
% C30 DSP library i.e; {-a2[s], -a1[s], b2[s], b1[s], b0[s]}, 0<=s<S
numsize = size(sosnum,1);
densize = size(sosden,1);
maxsize = max(numsize,densize);
for s=1:maxsize,
   if s <= densize,
      coeff = [coeff -sosden(s,3) -sosden(s,2)];
   else coeff = [coeff 0 0];
   end
   if s <= numsize,
      coeff = [coeff sosnum(s,3) sosnum(s,2) sosnum(s,1)];
   else coeff = [coeff 0 0 1];
   end
end
```

Listing A-1: MATLAB function to generate coefficient array from Z-transfer function