

Controller Implementation

Recall we have shown

$$U(s) = H(s)E(s) = \left[C_0 + \sum \frac{C_i}{s - a_i} \right] E(s)$$

where a_i are poles of $H(s)$, and C_0, C_1, C_2, \dots are PFE coeffs, with $C_0 = 0$ if $p(H) > 0$ ($H(s)$ has more poles than zeros).

$$\text{Let } X_i(s) = \left[\frac{1}{s - a_i} \right] E(s)$$

$$\text{then } u(t) = C_0 e(t) + \sum C_i x_i(t)$$

where $e(t) = y_d(t) - y(t)$, and $x_i(t)$ are sol'n's of

$$\dot{x}_i(t) = a_i x_i(t) + e(t)$$

The discrete time stepping under which the computer and sensor/actuator electronics operate mean that $u(t)$ will be computed only at integer multiples of the sample interval, T_s .

i.e. at $t_k = kT_s$, for $k = 0, 1, 2, \dots$

Let $u_k = u(t_k) = u(kT_s)$, and $e_k = e(t_k) = e(kT_s)$

From above:

$$u_k = C_0 e_k + \sum C_i x_i(t_k)$$

We need to know how to evaluate $x_i(t_k)$, i.e. sol'n of

$$\dot{x}_i(t) = a_i x_i(t) + e(t) \quad \text{evaluated at } t = t_k$$

Focus on just a single one of these eq'ns, since they are identical except for coeffs a_i :

$$\dot{x}(t) = ax(t) + \varepsilon(t) \quad \left(\text{Let } \varepsilon(t) = e(t) \text{ here, to avoid confusion with } e^{at} \right)$$

Assume $\varepsilon(t)$ is a step of size ε_0 , and $x(0) = x_0$

$$\text{Then } X(s) = \frac{x_0}{s-a} + \frac{\varepsilon_0}{s(s-a)}$$

$$\Rightarrow x(t) = e^{at} x_0 + \frac{1}{(-a)} (1 - e^{at}) \varepsilon_0$$

Note: $e(t)$ will not generally be a step, even if $y_d(t)$ is!

But, the above is a useful intermediate result, as we will see next.

Here

$$X(t) = e^{at}x_0 + \left(-\frac{1}{a}\right)(1 - e^{at})\varepsilon_0$$

$$\text{So } X(T_5) = e^{aT_5}x_0 + \left(-\frac{1}{a}\right)(1 - e^{aT_5})\varepsilon_0$$

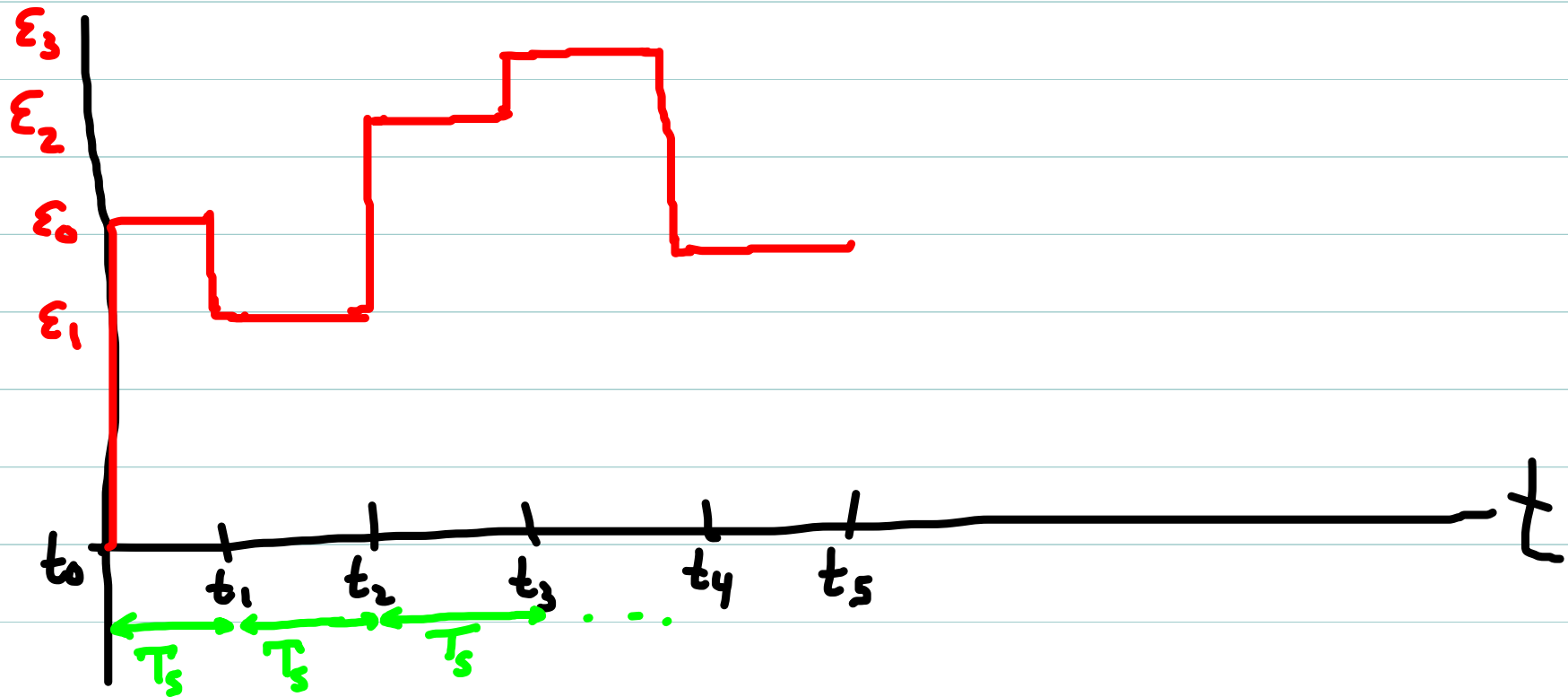
$$\text{Let } \alpha = e^{aT_5}, \quad \beta = (1 - \alpha)/(-a)$$

$$\text{Then } X(T_5) = \alpha X_0 + \beta \varepsilon_0$$

$$\text{Let } X(T_5) = x_1 \Rightarrow x_1 = \alpha X_0 + \beta \varepsilon_0 = X(t_1)$$

Now, how does this help generally?

Sampling of output at discrete times $t_k = kT_s$ means that error $e(t)$ will have a staircase graph



i.e. $e(t)$ will be constant with level ϵ_k on the interval $t_k \leq t < t_{k+1}$.

Note that at t_0 , $e(t)$ does look like a step.

So, it is true for first sample interval that

$$X(t_1) = x_1 = \alpha X_0 + \beta \varepsilon_0 \quad (\text{as above})$$

But what about subsequent time steps??

Exploit time invariance: when solving constant coeff DE's, the time called zero is arbitrary. All that matters is the initial cond'n and the time elapsed since initial time.

So, to get sol'n for next sample time t_2 , we can "reset" the zero time to t_1 , and use $X(t_1)$ as $\underset{\text{new}}{\text{initial cond'n}}$.

Then, from new zero time $t = t_1$, error $e(t)$ looks like a step of height ε_1 , so: ~ (initially)

$$X(t_2) = x_2 = \alpha X_1 + \beta \varepsilon_1 \quad \text{by same logic as above}$$

We can repeat this trick for all subsequent t_k :

$$X_{k+1} = \alpha X_k + \beta \epsilon_k$$

Where $x_k = x(t_k) = x(kT_s)$

$$\alpha = e^{aT_s}, \quad \beta = (1 - \alpha) / (-a)$$

$\epsilon_k = e(t_k)$ (error at k^{th} sample time)

We have thus shown that:

$$X(t_{k+1}) = \alpha X(t_k) + \beta e(t_k)$$

is an iterative algorithm for generating the exact sol'n for $x(t)$ at each of the sample times t_k , given the staircase structure of $e(t)$.

So generally:

$$u(t_k) = u_k = c_0 e(t_k) + \sum c_i x_i(t_k)$$

Where each $x_i(t_k)$ is computed iteratively using

$$x_i(t_{k+1}) = \alpha_i x_i(t_k) + \beta_i e(t_k)$$

$$\text{where } \alpha_i = \exp[a_i T_s], \quad \beta_i = \left[\frac{1 - \alpha_i}{(-a_i)} \right]$$

and T_s is the sample interval.

Real-time implementation

$$u(t_k) = u_k = c_0 e(t_k) + \sum C_i x_i(t_k) \quad , \quad t_k \text{ is } k^{\text{th}} \text{ update time}$$

Where each $x_i(t_k)$ is computed iteratively using

$$x_i(t_{k+1}) = \alpha_i x_i(t_k) + \beta_i e(t_k)$$

$$\text{and } \Rightarrow \alpha_i = \exp[a_i T_s] \quad , \quad \beta_i = \left[\frac{1 - \alpha_i}{(-a_i)} \right]$$

$\Rightarrow T_s$ is the sample interval (inverse of sample rate)

$\Rightarrow a_i$ are poles of $H(s)$

$\Rightarrow c_0, c_1, c_2 \dots$ are PFE coeffs of $H(s)$

$$\Rightarrow e(t_k) = y_d(t_k) - y(t_k)$$

Matlab code

Simple case: $H(s) = K \Rightarrow u(t) = Ke(t)$

```
function u=control(yd,y)
```

```
% define K (number!)
```

```
K=...
```

```
% compute u
```

```
e = yd-y;
```

```
u = K*e;
```

```
end
```

$H(s)$ with 1 pole

$$H(s) = K \frac{(s - z_c)}{(s - p_c)} = c_0 + \frac{c_1}{(s - p_c)}$$

function u=control(yd,y)

% define c0, c1, alpha, beta (as numbers!)

c0=...

c1=...

alpha=...

beta=...

$\alpha = \exp[p_c * T]$ here, and

$\beta = (1 - \alpha) / (-p_c)$

% compute u

e = yd-y;

u = c0*e+c1*x;

% update x

x = alpha*x+beta*e;

end

$H(s)$ with 1 pole

$$H(s) = K \frac{(s - z_c)}{(s - p_c)} = c_0 + \frac{c_1}{(s - p_c)}$$

function u=control(yd,y)

```
% define c0, c1, alpha, beta
```

```
c0=...
```

```
c1=...
```

```
alpha=...
```

```
beta=...
```

```
% compute u
```

```
e = yd-y;
```

```
u = c0*e+c1*x;
```

```
% update x
```

```
x = alpha*x+beta*e;
```

```
end
```

Unfortunately, won't work as written!

We need the function to "remember" the values of x between calls.

Remember: Matlab functions (like C/C++ functions) have their own, private workspace (storage) for their variables, which is separate from the main workspace (main function).

Local variables in functions are cleared after the function runs.

Can prevent this clearing by declaring the variable to be "persistent" in Matlab ("Static" in C/C++).

$$H(s) = K \frac{(s - z_c)}{(s - p_c)} = c_0 + \frac{c_1}{(s - p_c)}$$

function u=control(yd,y)

persistent x

% define c0, c1, alpha, beta

c0=...

c1=...

alpha=...

beta=...

% compute u

e = yd-y;

u = c0*e+c1*x;

% update x

x = alpha*x+beta*e;

end

$$H(s) = K \frac{(s - z_c)}{(s - p_c)} = c_0 + \frac{c_1}{(s - p_c)}$$

```
function u=control(yd,y)
```

```
    persistent x
```

```
    % define c0, c1, alpha, beta
```

```
    c0=...
```

```
    c1=...
```

```
    alpha=...
```

```
    beta=...
```

```
    % compute u
```

```
    e = yd-y;
```

```
    u = c0*e+c1*x;
```

```
    % update x
```

```
    x = alpha*x+beta*e;
```

```
end
```

Still won't work!

x needs to be initialized, but only the 1st time the function is called.

Matlab initializes a persistent variable as an empty array the first time the function is run

We can test for this, and set initial value of x to our pleasing: "isempty" function for test

Note: simplest to initialize x to zero, unless there is a compelling reason not to (very rare)

$$H(s) = K \frac{(s - z_c)}{(s - p_c)} = c_0 + \frac{c_1}{(s - p_c)}$$

function u=control(yd,y)

persistent x

```
if isempty(x) initialize x
    x=0;
end first time
```

% define c0, c1, alpha, beta

c0=...

c1=...

alpha=...

beta=...

Works!

```
% compute u
```

```
e = yd-y;
```

```
u = c0*e+c1*x;
```

```
% update x
```

```
x = alpha*x+beta*e;
```

```
end
```

All our mathematical analysis ultimately boils down to 4 "magic numbers" that we plug into this standard template.