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

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

persistent x

if isempty(x)
   x=0;
end
```

initialize x 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.

$$H(s) = 30\left[\frac{s+3}{s+9}\right] = 30 - \frac{180}{s+9}$$, $T_s = 0.1$ (10Hz)

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

persistent x

if isempty(x)
   x=0;
end

% define c0, c1, alpha, beta
c0 = 30;
c1 = -180;
alpha = 0.4066;
beta = 0.0659;

% compute u
e = yd-y;
u = c0*e+c1*x;

% update x
x = alpha*x+beta*e;

end
```

$$H(s) = K\frac{(s-z_{c1})(s-z_{c2})}{(s-p_{c1})(s-p_{c2})} = c_0 + \frac{c_1}{s-p_{c1}} + \frac{c_2}{s-p_{c2}}$$

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

persistent x1 x2

if isempty(x1)
   x1=0;
   x2=0;
end

% define constants
% ...
%

% compute u
e = yd-y;
u = c0*e+c1*x1+c2*x2;

% update x
x1 = alpha1*x1+beta1*e;
x2 = alpha2*x2+beta2*e;

end
```

An implementation with 2 poles in $H(s)$, hence 2 Diff. eq'ns which need to be solved ($x_1(t_k), x_2(t_k)$)

$$H(s) = K\frac{(s-z_{c1})(s-z_{c2})}{(s-p_{c1})(s-p_{c2})} = C_0 + \frac{C_1}{s-p_{c1}} + \frac{C_2}{s-p_{c2}}$$

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

persistent x

if isempty(x)
   x=[0;0];
end

% define constants
% ...
%

% compute u
e = yd-y;
u = c0*e+c1*x(1)+c2*x(2);

% update x
x(1) = alpha1*x(1)+beta1*e;
x(2) = alpha2*x(2)+beta2*e;

end
```

Alternate implementation using Matlab arrays

Extention to 3 or more poles in H(s) straightforward following same general pattern.

Note that, with two or more poles in $H(s)$, we could write the implementation equations in Matlab in the general form:

$$u = \underline{C} * x + D * e$$

$$\underline{x} = \underline{a} .* \underline{x} + \underline{b} * e$$

where $\underline{x}$ is vector with all the different x_i variables

$$\underline{a} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_M \end{bmatrix} \qquad \underline{b} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_M \end{bmatrix} \qquad M = \#\text{poles in } H(s)$$

$$\underline{C} = [C_1 \; C_2 \cdots C_M] \qquad D = C_0$$

Even more generally:

$$u = \underline{C} * \underline{x} + D * e$$

$$\underline{x} = A * \underline{x} + \underline{b} * e$$

"State space" representation of (discretized) controller dynamics

where

$$A = \text{diag}\{\alpha_1, \alpha_2, \cdots, \alpha_{\underline{M}}\}$$

$$= \begin{bmatrix} \alpha_1 & & & \emptyset \\ & \alpha_2 & & \\ & & \ddots & \\ \emptyset & & & \alpha_M \end{bmatrix}$$

$\underline{M} \times \underline{M}$ matrix

$(\underline{M} = \#\text{ poles in } H(s))$

It is precisely the A, B, C, D components of this generalized "state space" form that Matlab's discretization functions will give us for any $H(s)$ no matter how complicated:

$$[A,B,C,D] = \text{ssdata}(\,\text{ss}(\text{c2d}(H, T_s, \text{'tustin'}))\,)$$

Does the discretization and gives us the matrix/vectors to use in the general form of the implementation eq'ns.

Note: A may not be diagonal generally

The zero-order hold (ZOH) discretization techniques we have examined are necessary in light of the $e(t)$ the computer actually "sees"

However, it is <u>Not</u> an exact solution of the implementation equations. Those Assume $e(t)$ is a continuous function, while ZOH discretization approximates $e(t)$ as a "staircase" function.

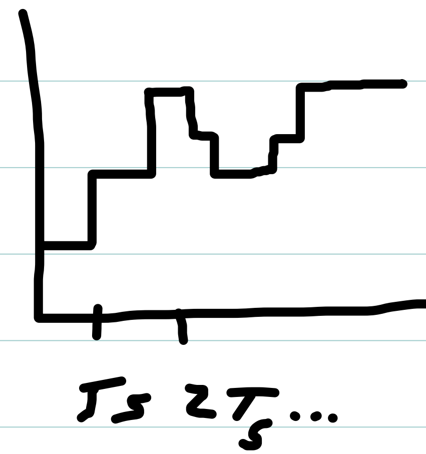
In fact, there is a <u>transfer function</u> that models the conversion from continuous $e(t)$ to staircase e_k samples:

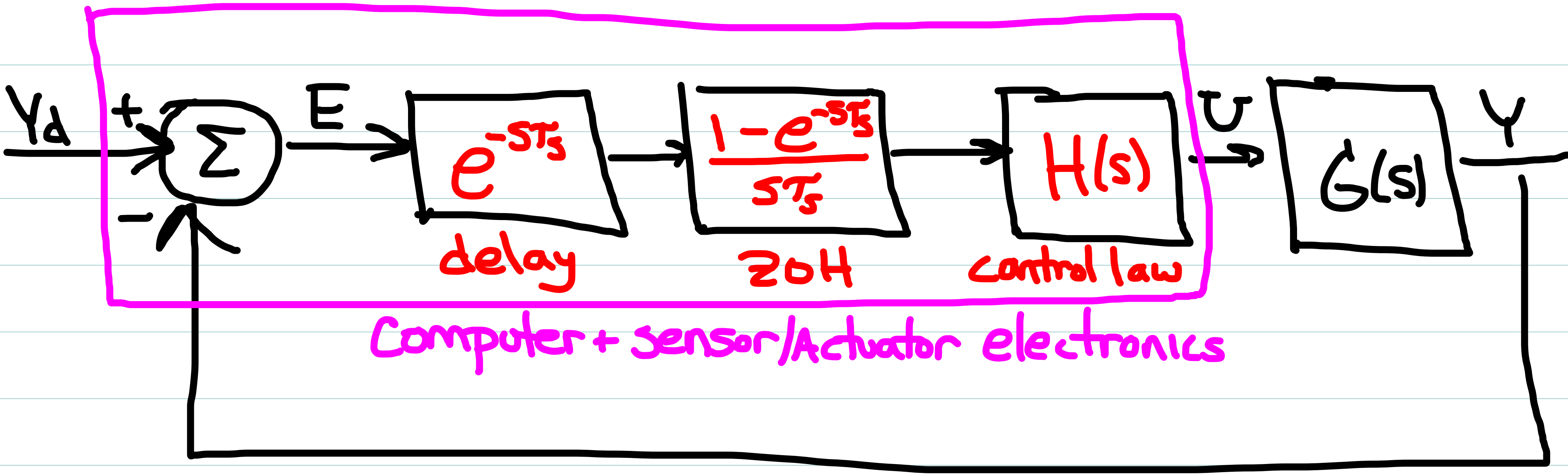
$$ZOH(s) = \frac{1 - e^{-sT_s}}{sT_s}$$

where as usual T_s is the interval between samples.

ZOH(s)

$$e(t) \longrightarrow \boxed{\frac{1-e^{-sT_s}}{sT_s}} \longrightarrow e_k$$





Note that ZOH(s), like the pure delay TF e^{-sT_s}, is transcendental, not described by a finite number of poles and zeros.

However its effect on Bode can be quantified, like the delay term.

Unlike the delay, ZOH(s) <u>does</u> affect the mag diagram, but only at high freqs ($\omega > 2/T_s$ or so).

The principle effect of the ZOH discretization, like the delay term, is to introduce negative phase.

In fact, the phase loss due to ZOH is exactly half that due to the delay, i.e. $-\omega T_s/2$

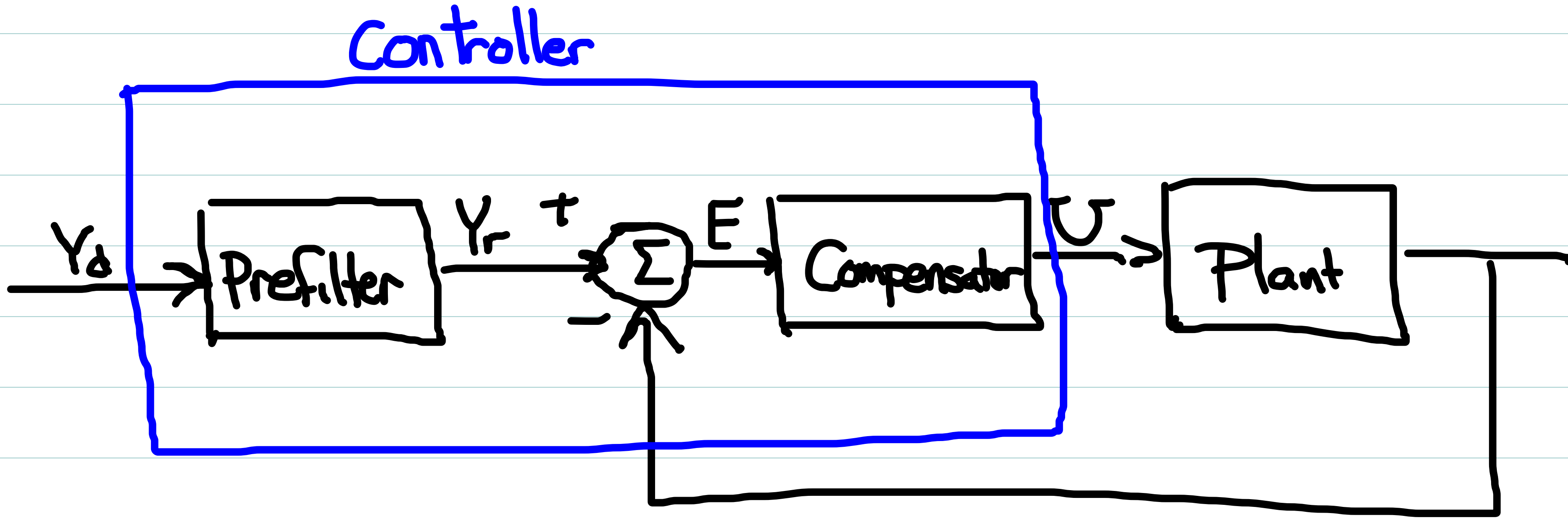
As a consequence:

$$\underbrace{\gamma_{eff}}_{\text{effective phase margin}} = \underbrace{\gamma_0}_{\text{nominal phase margin}} - \underbrace{\omega_\gamma T_s}_{\text{phase loss from delay}} - \underbrace{\tfrac{1}{2}\omega_\gamma T_s}_{\text{phase loss from ZOH discretization}}$$

So: $$\gamma_{eff} = \gamma_0 - \underbrace{1.5}_{\text{50\% more phase loss}}\,\omega_\gamma T_s$$

and $$T_{max} = \gamma_0 / 1.5\,\omega_\gamma$$

"Prefilter" Designs

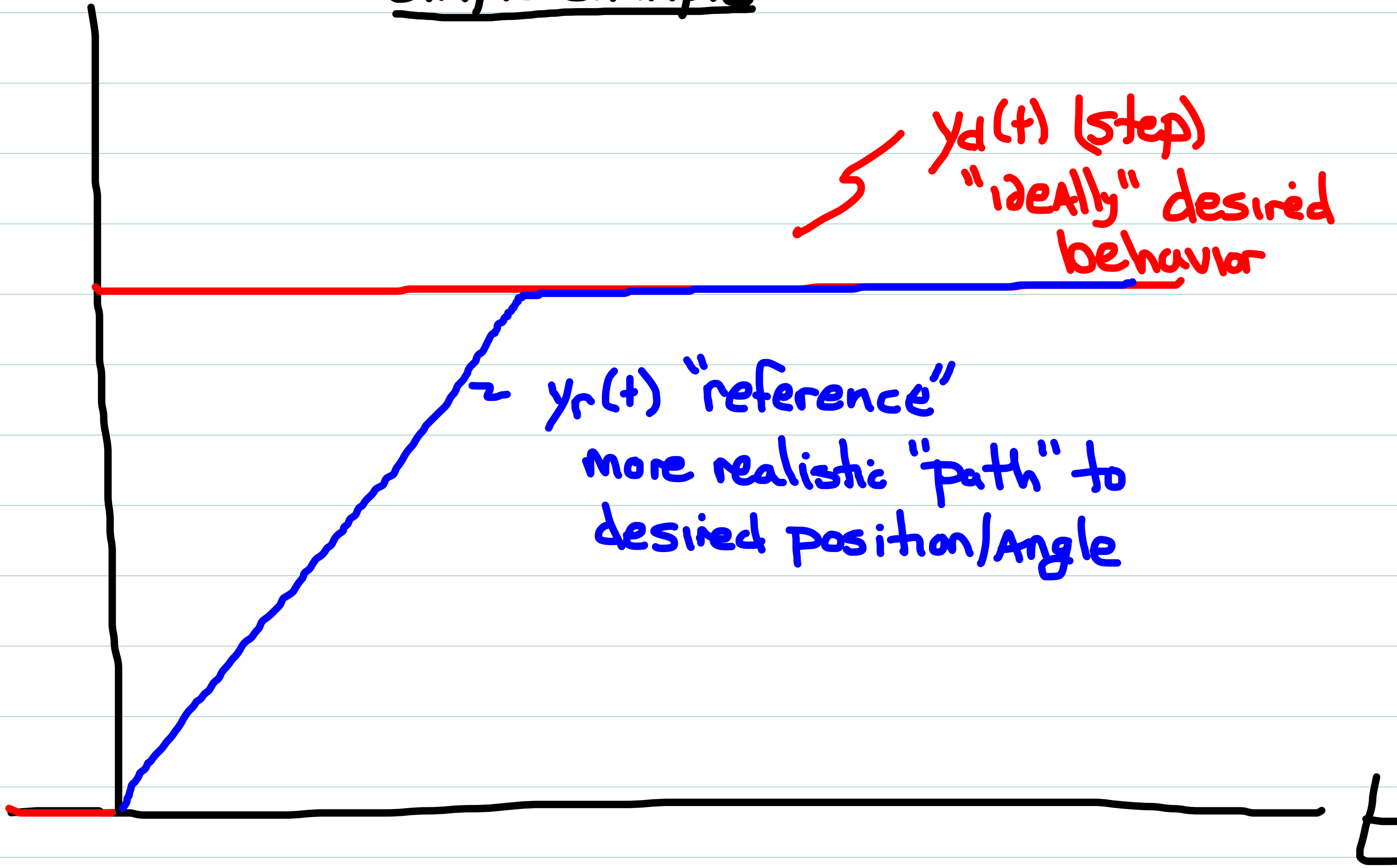


⇒ Prefilter is an extra degree of freedom in controller design

⇒ "Smooths" or "shapes" $y_d(t)$ into a "more reasonable" trajectory $y_r(t)$ which is easier for feedback loop to track

⇒ Can minimize some undesireable features of transient response, especially overshoot.

Simple Example



Reference trajectory goes to same value as $\theta_d(t)$, but in a smoother, less sudden, fashion

A useful framework for studying prefilter is to assume its action can be modeled by a transfer function $F(s)$:

$$Y_r(s) = F(s)\,Y_d(s)$$

for example, if $F(s) = \frac{1}{\tau s+1}$, $\tau > 0$ then

When using a prefilter we have:

$$Y(s) = T(s)Y_r(s) = \boxed{T(s)F(s)Y_d(s)}$$

where $T(s) = \dfrac{G(s)H(s)}{1+G(s)H(s)}$ as usual.

Recall that $H(s)$ typically has LHP zeros

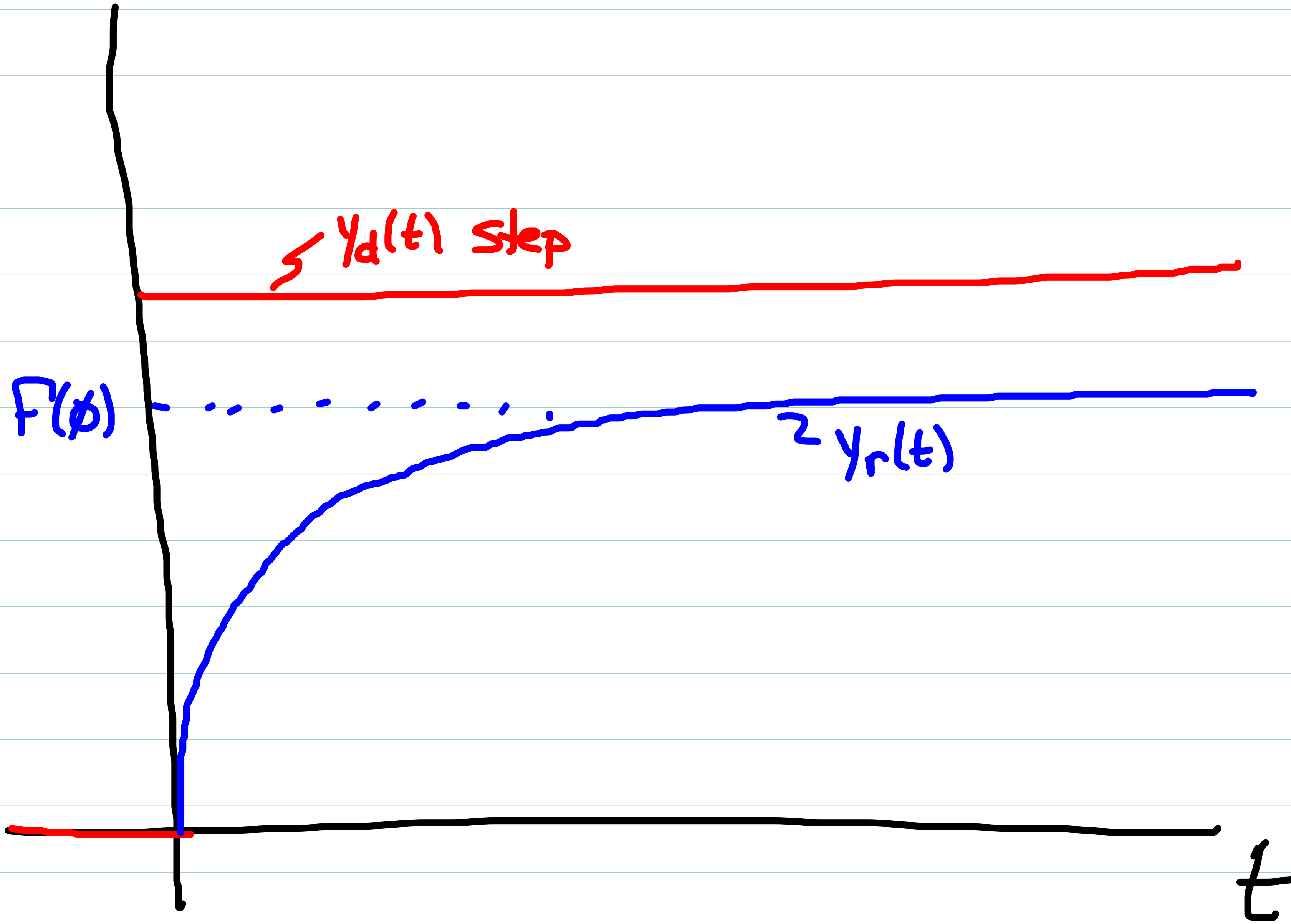
$\Rightarrow$ These zeros are also zeros of $T(s)$

$\Rightarrow$ They can substantially increase the overshoot

Use new degree of freedom $F(s)$ to <u>cancel</u> some or all zeros in $T(s)$, especially zeros used in compensator

$\Rightarrow$ $F(s)$ could have <u>poles</u> where $H(s)$ has (LHP) <u>zeros</u>!

Add'l constraint: need $F(0) = 1$ (Bode gain of 1)



If $F(0) \neq 1$, $y_r(t)$ will not converge to actual desired behavior

When using a prefilter:

$Y(s) = \boxed{T(s)F(s)} Y_d(s)$ Use to predict transients

$E(s) = \boxed{1 - T(s)F(s)} Y_d(s)$ Use to predict bandwidth

$U(s) = \boxed{R(s)F(s)} Y_d(s)$ Use to predict control usage

Generally a prefilter designed as above will:

$\Rightarrow$ greatly improve overshoot

$\Rightarrow$ slightly worsen tracking bandwidth

$\Rightarrow$ moderately reduce peak control efforts.

Generally advantageous (but increases complexity of implementation)

<u>However</u>, when using a prefilter:

⇒ still use $L(s)$ to design for stability (Nyquist / phase margin)

⇒ still use $S_i(s)$ to predict disturbance rejection

⇒ still use $T_o(s)$ to predict robustness (Δ - test)

Prefilter does <u>Not</u> affect "internal" properties of feedback loop.

⇒ $F(s)$ designed <u>after</u> designing a good compensator $H(s)$. All the usual design rules for $H(s)$ are <u>unaffected</u> by use of a prefilter.

⇒ Prefilter just adds a way to further "clean up" response of system to sharp changes in $y_d(t)$

⇒ Diff'l eq'ns corresponding to $F(s)$ can be implemented on computer in exactly same manner as for $H(s)$.

⇒ Do a PFE on $F(s)$, and use the resulting equations to generate $y_r(t)$ from $y_d(t)$

$$Y_r(s) = F(s)\,Y_d(s)$$

$$= \left[\frac{c_1}{s-f_1} + \frac{c_2}{s-f_2} + \cdots\right] Y_d(s)$$

⇒ Generate equivalent $x_k(t)$ diff eq'n driven by $y_d(t)$, and do a ZOH discretization just like for $H(s)$ equations

⇒ Then replace $y_d(t)$ with $y_r(t)$ in controller implementation i.e. use $e(t) = y_r(t) - y(t)$ in calculations for $u(t)$.

⇒ If plant has nonzero IC, good idea to initialize prefilter with $y_r(\emptyset) = y(\emptyset)$ in implementation.