```
function u = control(yd,y)
% Stub to illustrate most general form
% of discretized implementation equations

% Note: more efficient to define numerical
% components once, when we initialize x
  persistent x Ad Bd Cd Dd
  if isempty(x)
    Ad = []; % square matrix
    Bd = []; % column vector
    Cd = []; % row vector
    Dd = []; % scalar
    x = zeros(size(Ad,1),1);
    % Note: one state (xi variable) for each row of A
    %  (equivalently, for each pole in H(s))
  end

% Do the actual calculations
  e = yd - y;
  u = Cd*x + Dd*e;
  x = Ad*x + Bd*e;

end
```

Fill with #s for your H(s)

Standard template

Implementation of pole at origin (ZOH)

If $p_c = \phi$ (comp pole at origin), then clearly

$$\alpha = \exp[\phi T_s] = 1$$

in the implementation eq'n. However $\beta = \frac{(1-1)}{\phi}$ is indeterminate.

If we look more carefully at $\lim_{p_c \to \phi} \left[\frac{1 - \exp[p_c T_s]}{-p_c} \right]$

This yields the correct value $\beta = T_s$ for this case.

Thus for $\dot{x}(t) = e(t)$

we have $x(t_{k+1}) = x(t_k) + T_s\, e(t_k)$

i.e. $x_{k+1} = x_k + T_s e_k$

A closer look

$$\dot{x}(t) = e(t) \Rightarrow x_{k+1} = x_k + T_s e_k$$

equiv $\Rightarrow$ $$x(t+dt) = x(t) + dt\, e(t)$$

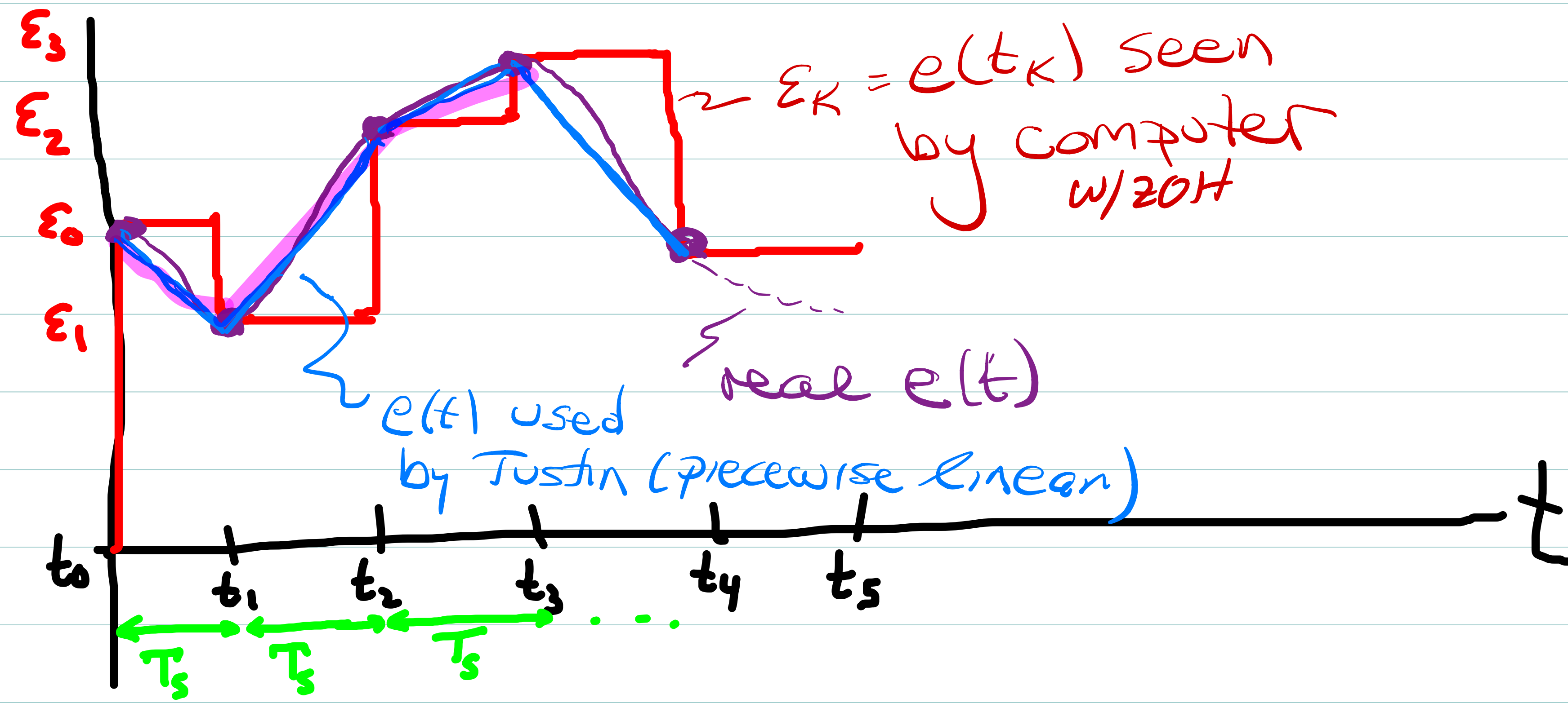
So our ZOH discretization strategy is equivalent to a simple (and not terribly accurate) Euler method for numerically integrating

Better idea:

$$x(t+dt) = x(t) + \frac{dt}{2}\left[e(t) + e(t+dt)\right]$$

i.e. a trapezoidal numerical approximation

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 $t_K \le t < t_{K+1}$.

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

Equivalent discrete equations (trap. integrate)

$$x(t+dt) = x(t) + \frac{dt}{2}\left[e(t) + e(t+dt)\right]$$

$$\Rightarrow \quad x_{k+1} = x_k + \frac{T_s}{2}\left[e_k + e_{k+1}\right]$$

Which seems to require knowledge of future (e_{k+1})

But:

Let $\quad z_k = x_k - \frac{T_s}{2} e_k$

Then $\quad z_{k+1} = x_{k+1} - \frac{T_s}{2} e_{k+1}$

$$= x_k + \frac{T_s}{2} e_k + \cancel{\frac{T_s}{2} e_{k+1}} - \cancel{\frac{T_s}{2} e_{k+1}}$$

$$\Rightarrow z_{k+1} = z_k + T_s e_k$$

Trapezoidal ("Tustin") Discretization

So $\dot{x}(t) = e(t)$ can more accurately be discretized with the pair of equations

$$\boxed{\begin{aligned} z_{k+1} &= z_k + T_s e_k \\ x_k &= z_k + \frac{T_s}{2} e_k \end{aligned}}$$

Extension to general 1st order DEs is known as

"Tustin's method"

Can be ~~Generally~~ more accurate than simple ZOH.

$\Rightarrow$ most commonly used in practice

Straightforward to calculate, but algebraically tedious

Matlab's "c2d" function is very helpful to get the $[A_d, B_d, C_d, D_d]$ for either ZOH (default) or Tustin discretization

$$[A_d, B_d, C_d, D_d] = \text{ssdata}(\text{c2d}(H, T_s, \text{option}))$$

Omit "option" for ZOH, or use 'tustin' to specify that method.

Example: $$H(s) = \frac{.25(s+3)^3}{s(s/15+1)^2}$$

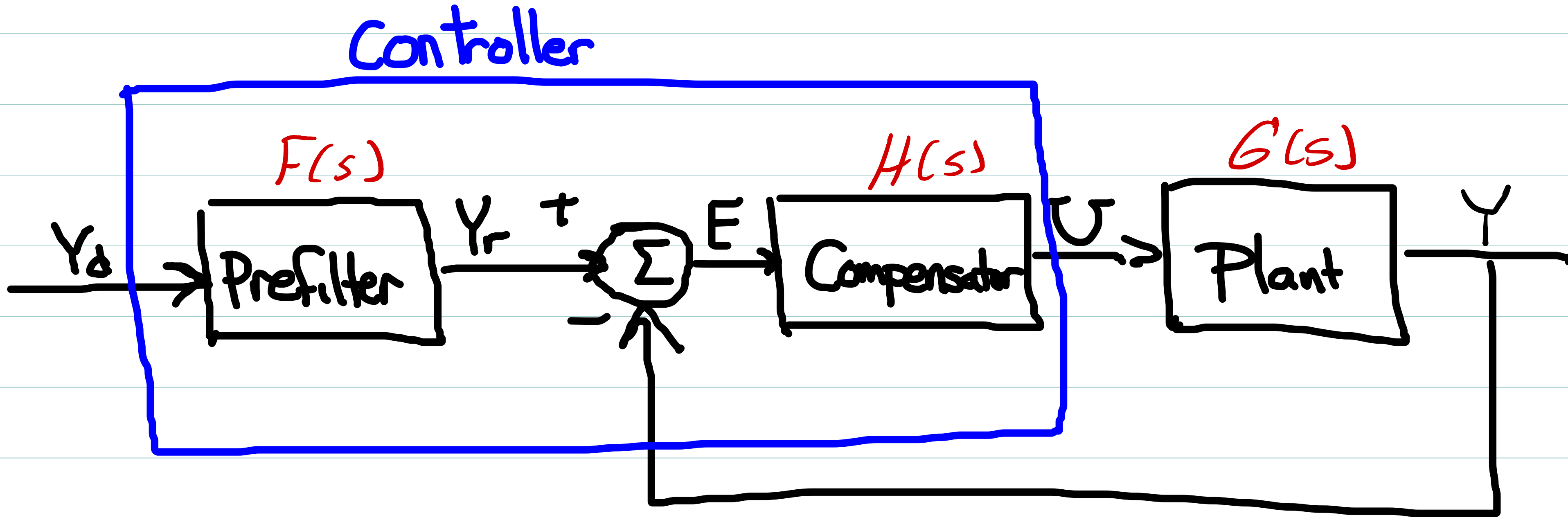
```
[Ab, Bb, Cd, Dd] = ssdata(c2d(H, .05, 'tustin'));
```

<see m-file>

```
function u = control(yd,y)
% Specific illustration of implementation
% equation using the results of discretizing
% the example H(s) in makesscomp.m

  persistent x Ad Bd Cd Dd
  if isempty(x)
    Ad = [1.9091   -1.1157    0.4132;
          1.0000         0         0;
          0          0.5000        0];
    Bd = [8; 0; 0];
    Cd = [-3.1061    5.1075   -3.9777];
    Dd = 36.9609;
    x = zeros(size(Ad,1),1);
  end
  % Note: the default display of numerical results in Matlab↙
is
  % 4 decimal digits -- less than provided by a C/C++ "float"↙
type.
  % This may not provide sufficient accuracy in practice.
  % Recall that Matlab actually does all of its calculations
  % in double precision (15 decimal digits), and you can see↙
all
  % of them (to copy into control.m) using "format long".

  e = yd - y;
  u = Cd*x + Dd*e;
  x = Ad*x + Bd*e;

end
```

for Tustin disc. of $H(s) = \frac{.25(s+3)^3}{s(s/15+1)^2}$

"Prefilter" Designs

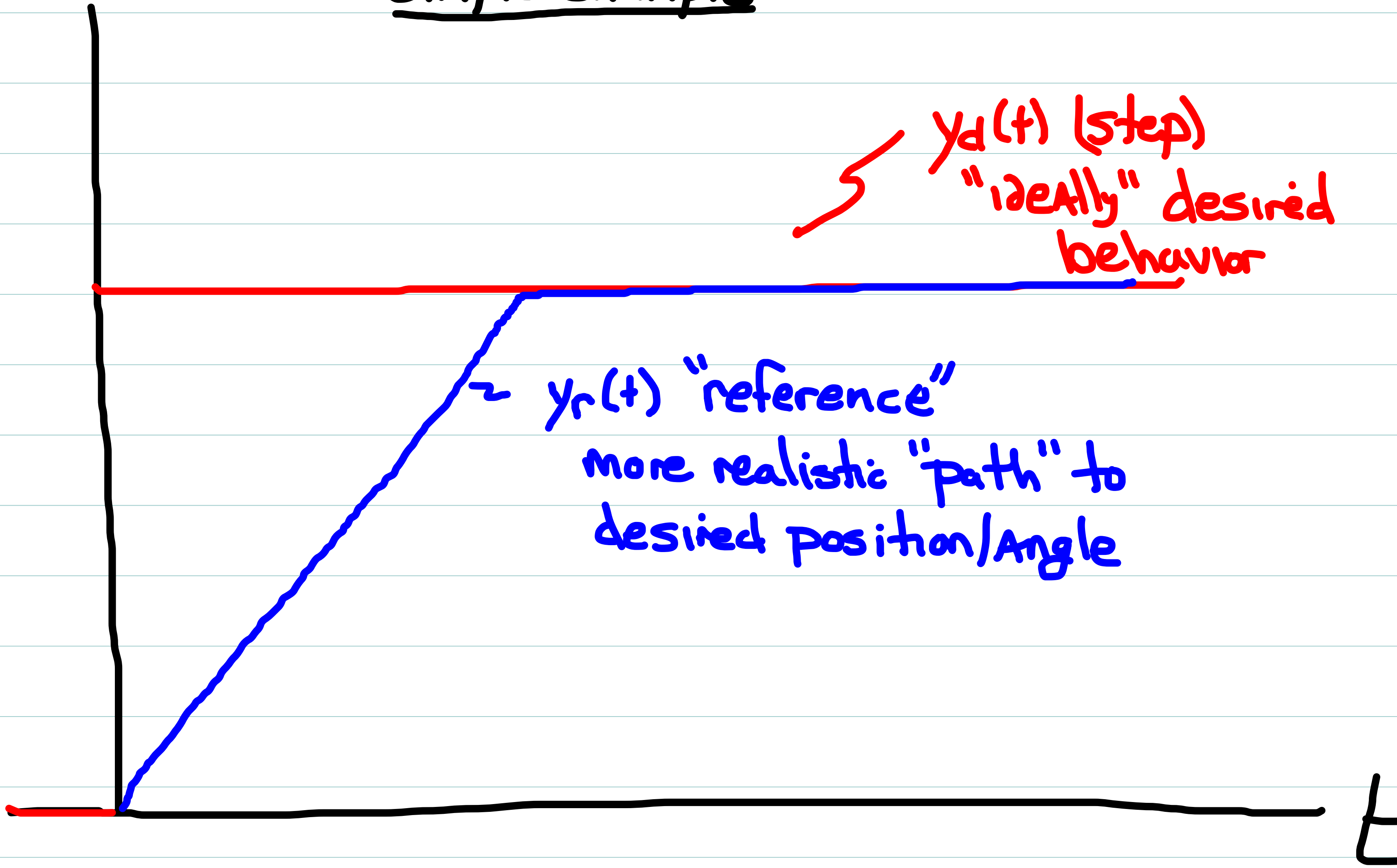


⇒ Prefilter is an extra degree of freedom in controller design

⇒ "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 $F(s) = \dfrac{1}{\tau s+1}$, $\tau > 0$ 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

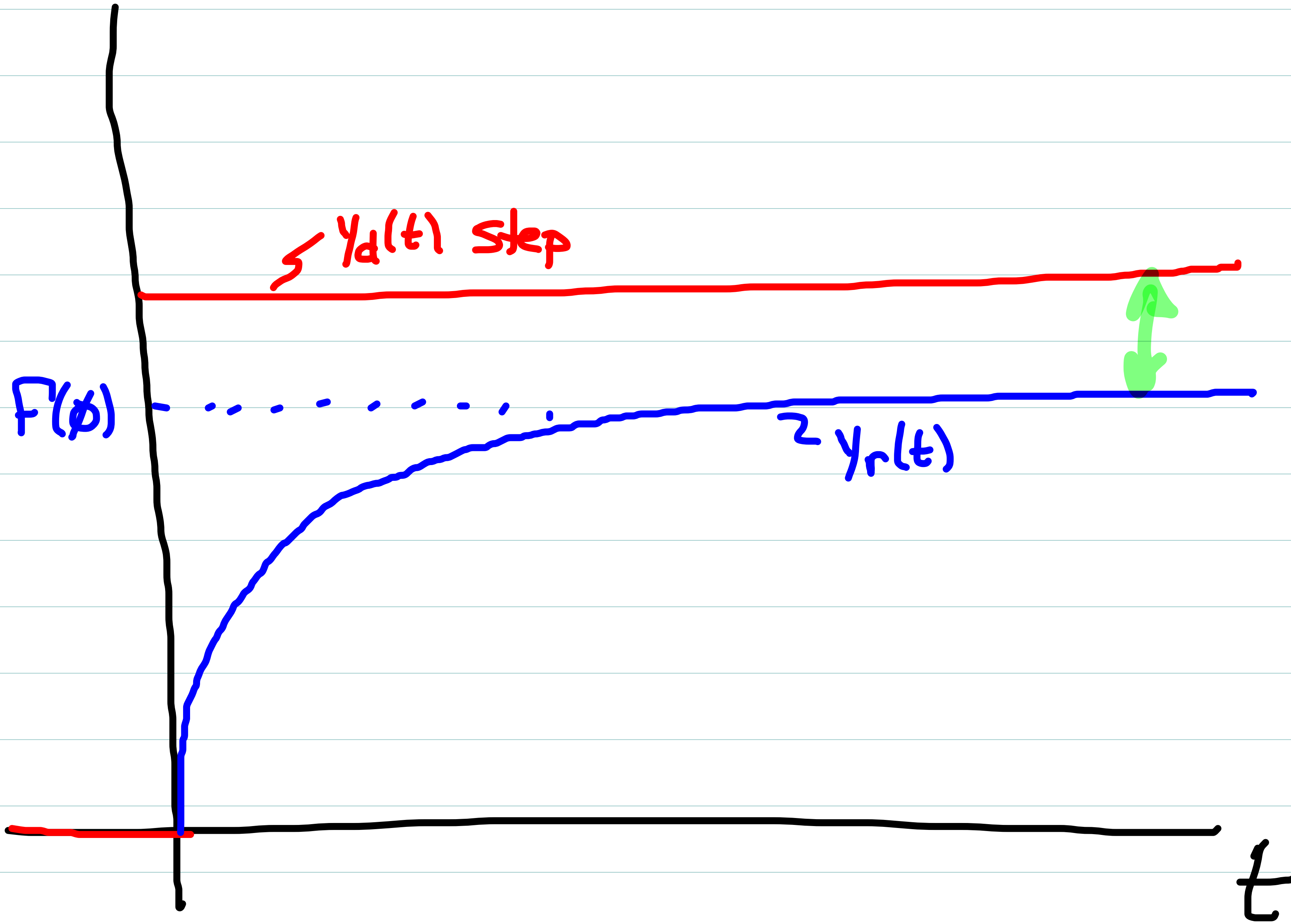
$\Rightarrow$ These zeros are also zeros of $T(s)$

$\Rightarrow$ 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:

$Y(s) = [T(s)F(s)]Y_d(s)$ Use to predict transients

$E(s) = [1 - T(s)F(s)]Y_d(s)$ Use to predict bandwidth

$U(s) = [R(s)F(s)]Y_d(s)$ Use to predict control usage

Generally a prefilter designed as above will:

⇒ greatly improve overshoot

⇒ slightly worsen tracking bandwidth

⇒ moderately reduce peak control efforts.

Generally advantageous (but increases complexity of implementation)

However, 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) and noise rejection

$\boxed{\text{Prefilter does Not affect "internal" properties of feedback loop.}}$

⇒ $F(s)$ designed after designing a good compensator $H(s)$. All the usual design rules for $H(s)$ are unaffected 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
or Tustin

⇒ 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.

Code modification w/prefilter:

$yr = Cr*xr + Dr*yd$ add

$e = yr - y;$ change

$u = C_d*x + D_d*e$

$x = A_d*x + B_d*e$

same

$xr = Ar*xr + Br*yd$ add

$\Rightarrow$

$[Ar, Br, Cr, Dr]$ obtained from $F(s)$ exactly like $[A_d, B_d, C_d, D_d]$ obtained from $H(s)$. using c2d w/same sample rate