

Interaction between Control and Scheduling

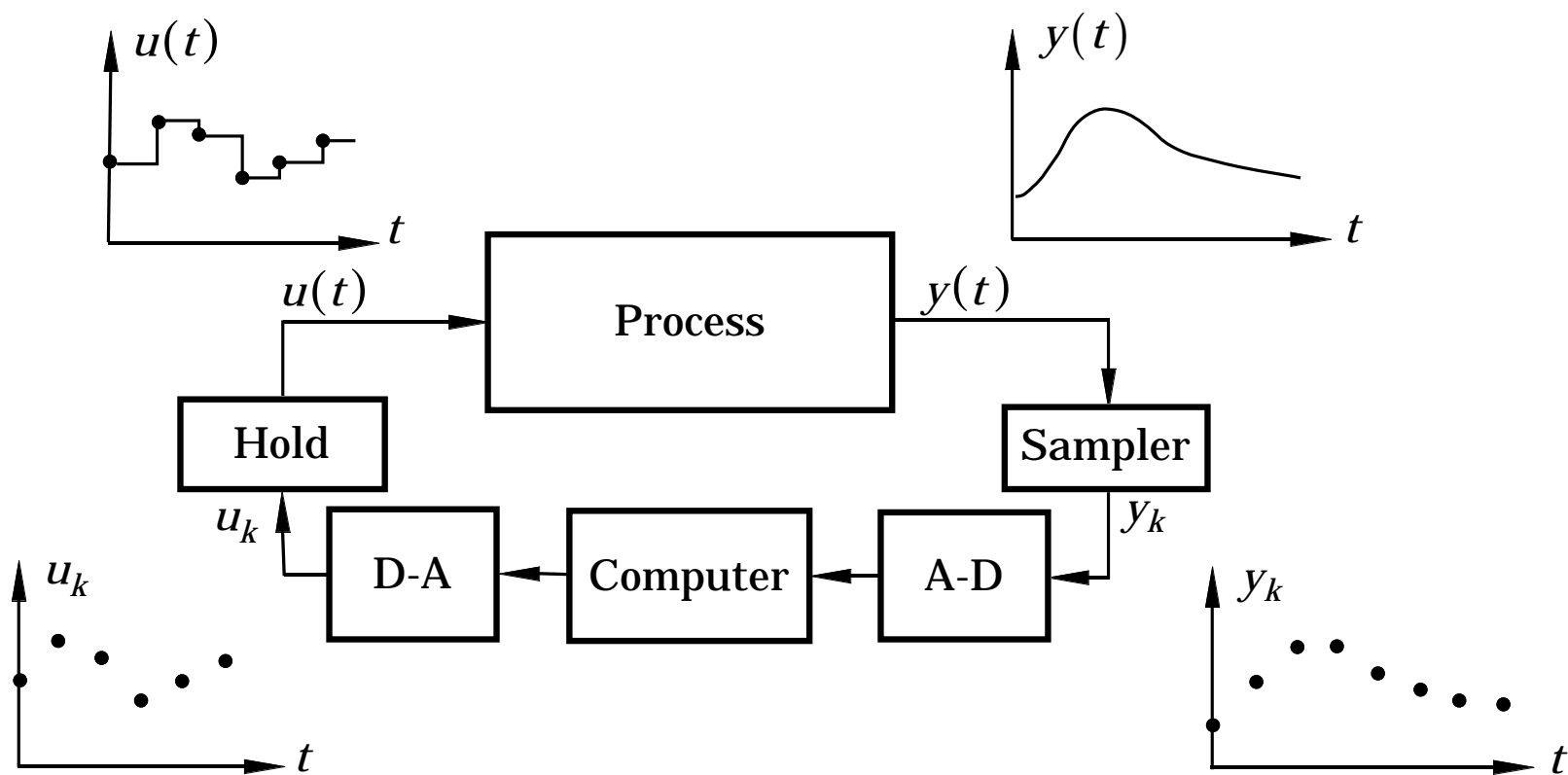
Karl-Erik Årzen, Anton Cervin

Session Outline

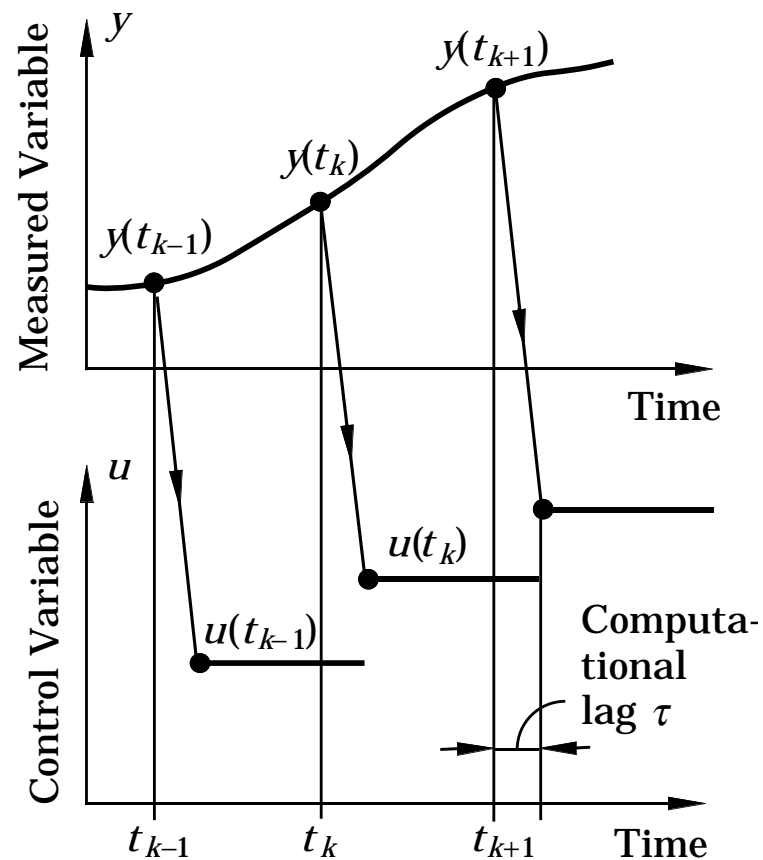
- **Control Loop Timing**
- Temporal Non-Determinism
- Switching
- The Jitter Margin
- Subtask Scheduling

Assumptions

In this session we will focus on periodically sampled control loops.

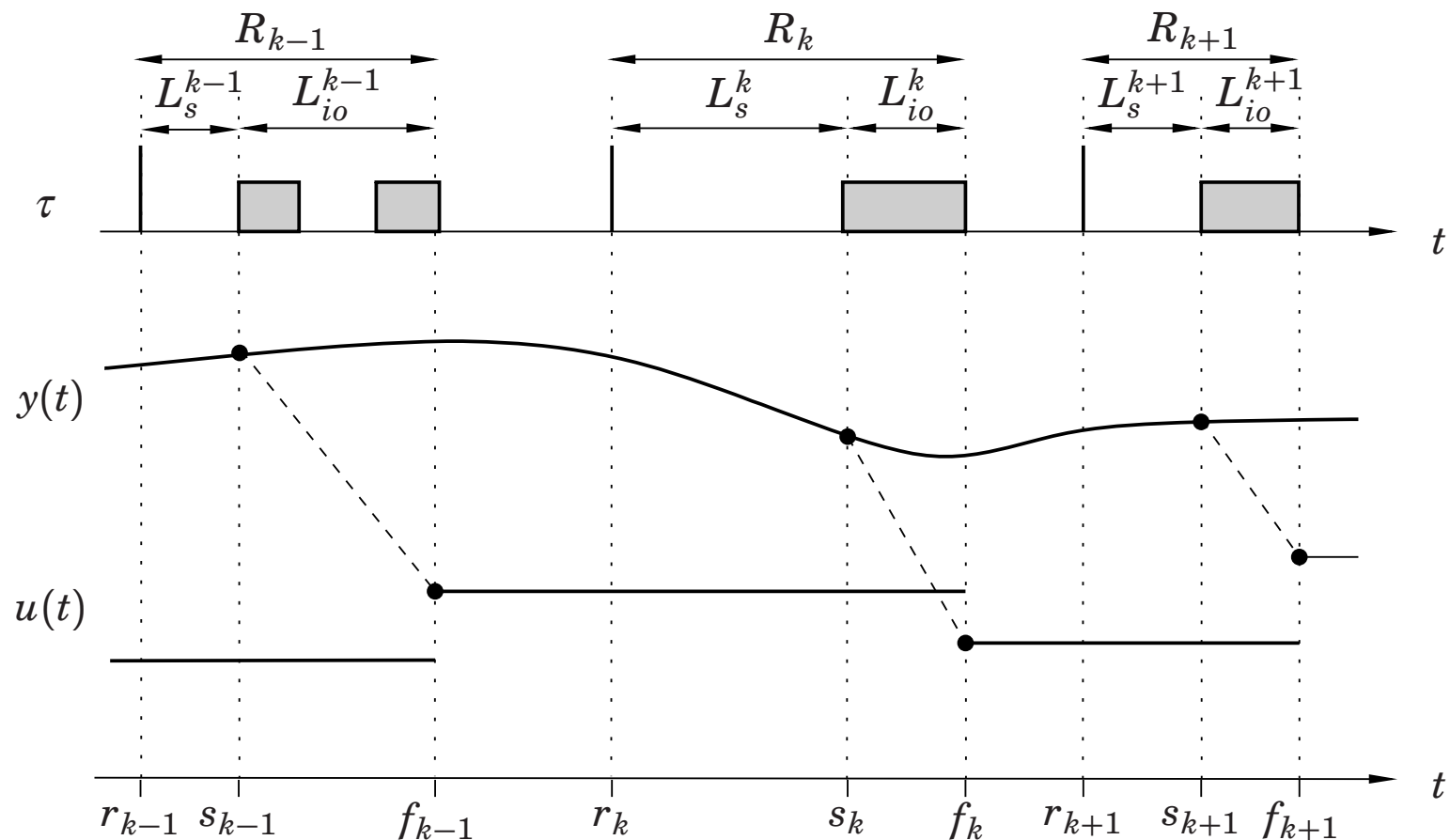


Ideal Controller Timing



- Output $y(t)$ sampled periodically at time instants $t_k = kh$
- Control $u(t)$ generated after short and constant time delay τ

Real Controller Timing



- Control task τ released periodically at time instances $r_k = kh$
- Output $y(t)$ sampled after time-varying **sampling latency** L_s
- Control $u(t)$ generated after time-varying **input-output latency** L_{io}

Jitter

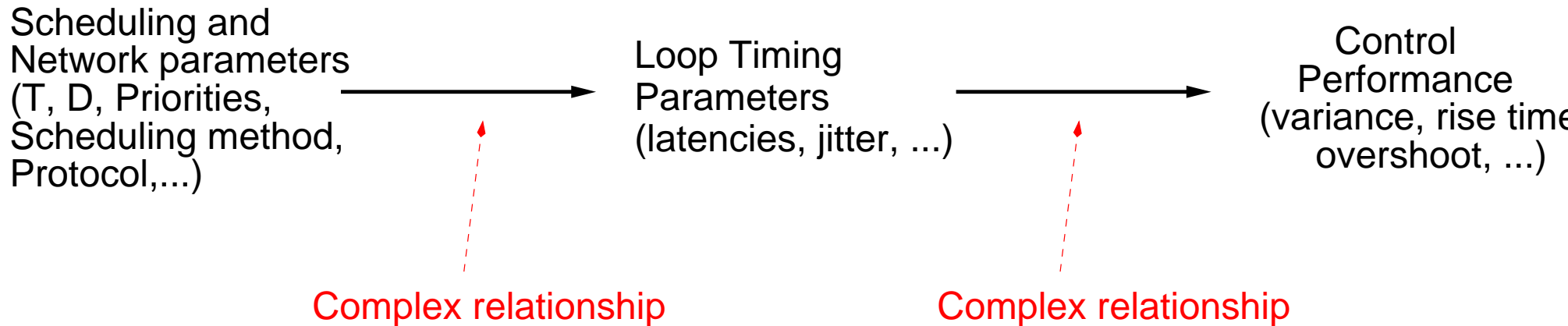
Sampling jitter:

$$J_s = \max_k L_s^k - \min_k L_s^k$$

Input-output jitter:

$$J_{io} = \max_k L_{io}^k - \min_k L_{io}^k$$

Timing Relationships



Possibilities for analysis:

- Simulation – the TrueTime tool
- “Numerical Analysis” – the Jitterbug tool
- Theoretical results – e.g., the Jitter Margin

Implementing Periodic Controller Tasks

Three main issues:

1. How do we achieve periodic execution?
2. When is the sampling performed?
3. When is the control signal sent out?

1. How Do We Achieve Periodic Execution?

Options:

1. Using a static schedule (cyclic executive)
 - High temporal determinism but inflexible
 - Does not require any sophisticated RTOS support
2. In interrupt handlers associated with timers
3. As self-scheduling threads in an RTOS/kernel using time primitives such as sleep/delay/WaitTime (relative wait) or sleepUntil/delayUntil/WaitUntil (absolute wait)
4. Using an RTOS/kernel with built-in support for periodic tasks
 - implement the tasks as simple procedures/methods that are registered with the kernel
 - not yet common in commercial RTOS

Implementing Self-Scheduling Periodic Tasks

Attempt 1:

```
LOOP
```

```
    PeriodicActivity;
```

```
    WaitTime(h);
```

```
END;
```

Does not work.

Period $> h$ and time-varying.

The execution time of PeriodicActivity is not accounted for.

Implementing Self-Scheduling Periodic Tasks

Attempt 2:

LOOP

```
Start = CurrentTime();
```

```
PeriodicActivity;
```

```
Stop = CurrentTime();
```

```
C := Stop - Start;
```

```
WaitTime(h - C);
```

END;

Does not work. An interrupt causing suspension may occur between the assignment and WaitTime.

In general, a WaitTime (Delay) primitive is not enough to implement periodic processes correctly. A WaitUntil (DelayUntil) primitive is needed.

Implementing Self-Scheduling Periodic Tasks

Attempt 3:

```
t = CurrentTime();  
LOOP  
    PeriodicActivity;  
    t = t + h;  
    WaitUntil(t);  
END;
```

Correct in case no overruns occur.

Will try to catch up if the actual execution time of PeriodicActivity occasionally becomes larger than the period (a too long period is followed by a shorter one to make the average correct)

2. When is the Sampling Performed?

Two options:

- At the beginning of the controller task
 - gives rise to sampling jitter
 - still quite common
- At the nominal task release instants
 - using a dedicated high-priority sampling task or in the clock interrupt handler
 - somewhat more involved scheme
 - minimizes the sampling jitter (but increases the average

3. When Is the Control Signal Sent Out?

Three Options:

- At the end of the controller task
 - creates a longer than necessary input-output latency
- As soon as it can be sent out
 - minimizes the input-output latency
 - controller task split up in two parts: CalculateOutput and UpdateState
- At the next sampling instant
 - minimizes the latency jitter
 - gives a much longer latency than necessary
 - often gives worse performance, also if the constant delay is compensated for
 - delay compensation easy

Minimizing the Input-Output Latency

General linear controller:

$$\begin{aligned}x(k+1) &= Fx(k) + Gy(k) + G_r y_{ref}(k) \\ u(k) &= Cx(k) + Dy(k) + D_r y_{ref}(k)\end{aligned}$$

Do as little as possible between AdIn and DaOut

```
PROCEDURE Regulate;
BEGIN
  AdIn(y);
  (* CalculateOutput *)
  u := u1 + D*y + Dr*yref;
  DaOut(u);
  (* UpdateStates *)
  x := F*x + G*y + Gr*yref;
  u1 := C*x;
END Regulate;
```

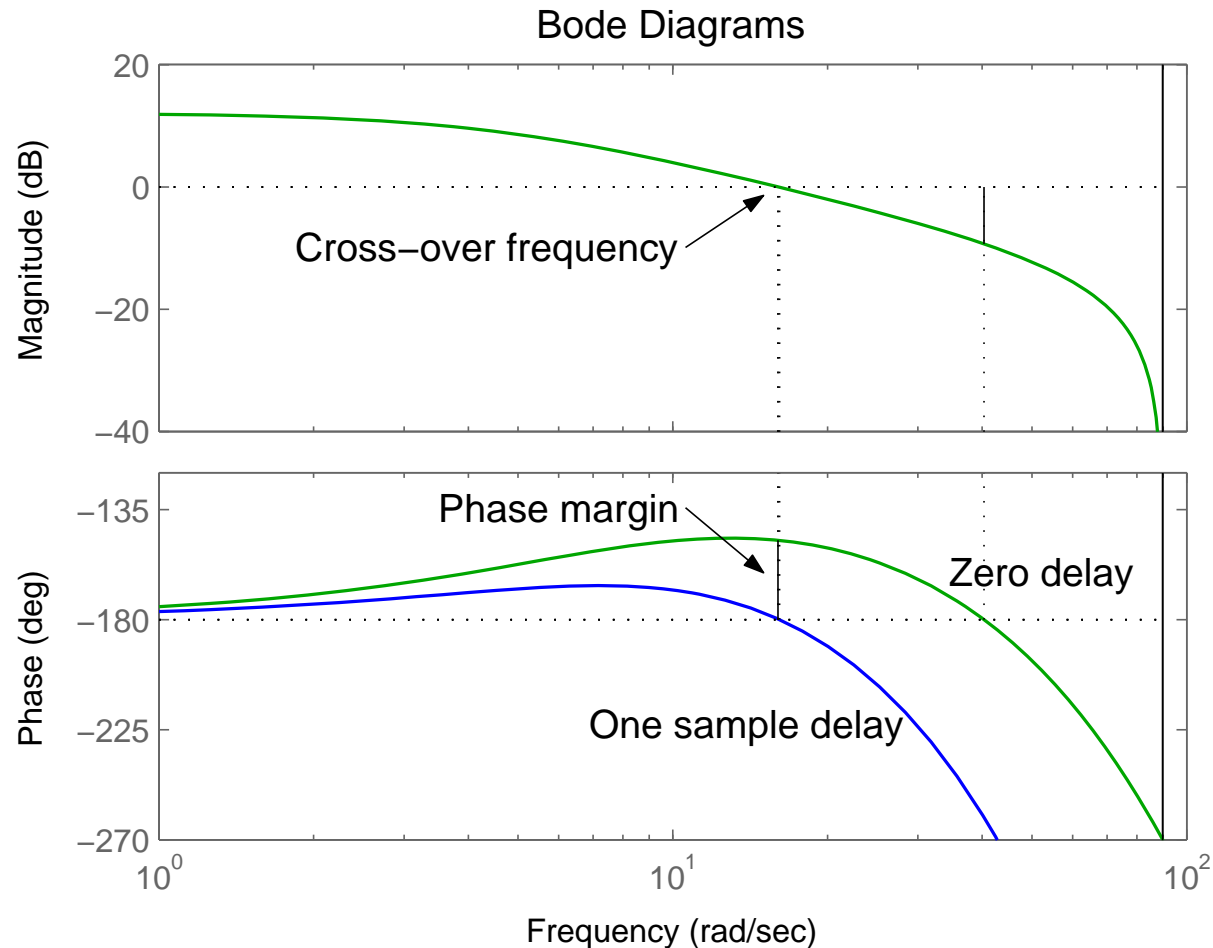
Session Outline

- Control Loop Timing
- **Temporal Non-Determinism**
- Switching
- The Jitter Margin
- Subtask Scheduling

Why is Input-Output Latency Bad?

A constant input-output latency decreases the phase margin.

Example: Loop gain with zero delay or one sample delay:



Computing the Delay Margin*

We have

- Phase margin $\phi_m = 32.4^\circ$
- Crossover frequency $\omega_c = 16.0$ rad/s

How large delay L can be tolerated before we lose stability?

The delay is modeled by $G(s) = e^{-sL}$

At crossover frequency: $\arg G(i\omega_c) = \arg e^{-i\omega_c L} = -\omega_c L$

To retain a positive phase margin, we must have

$$\omega_c L < \phi_m$$

$$16.0 L < 32.4^\circ \frac{\pi}{180^\circ}$$

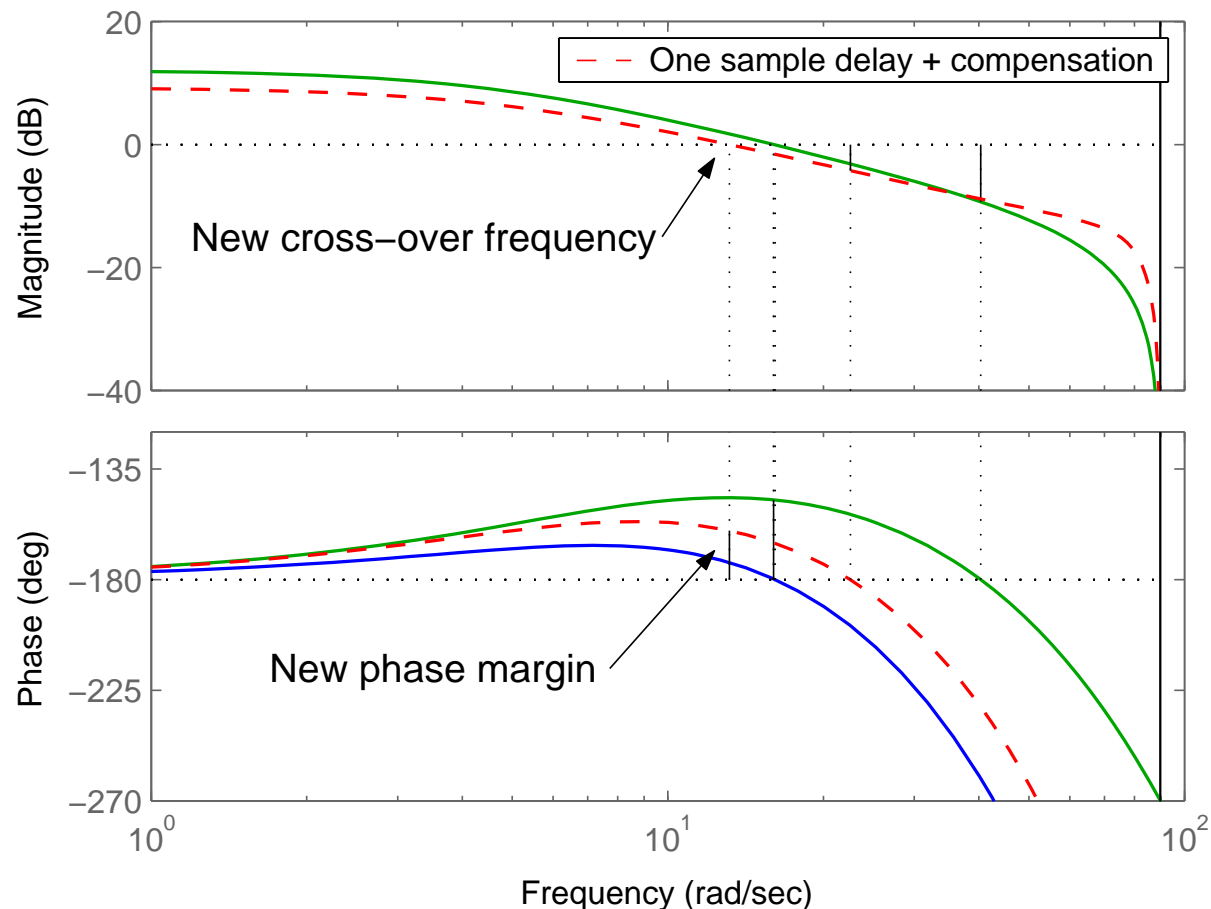
$$L < 0.035$$

* Since we have a sampled system, the analysis is only approximate

Delay Compensation

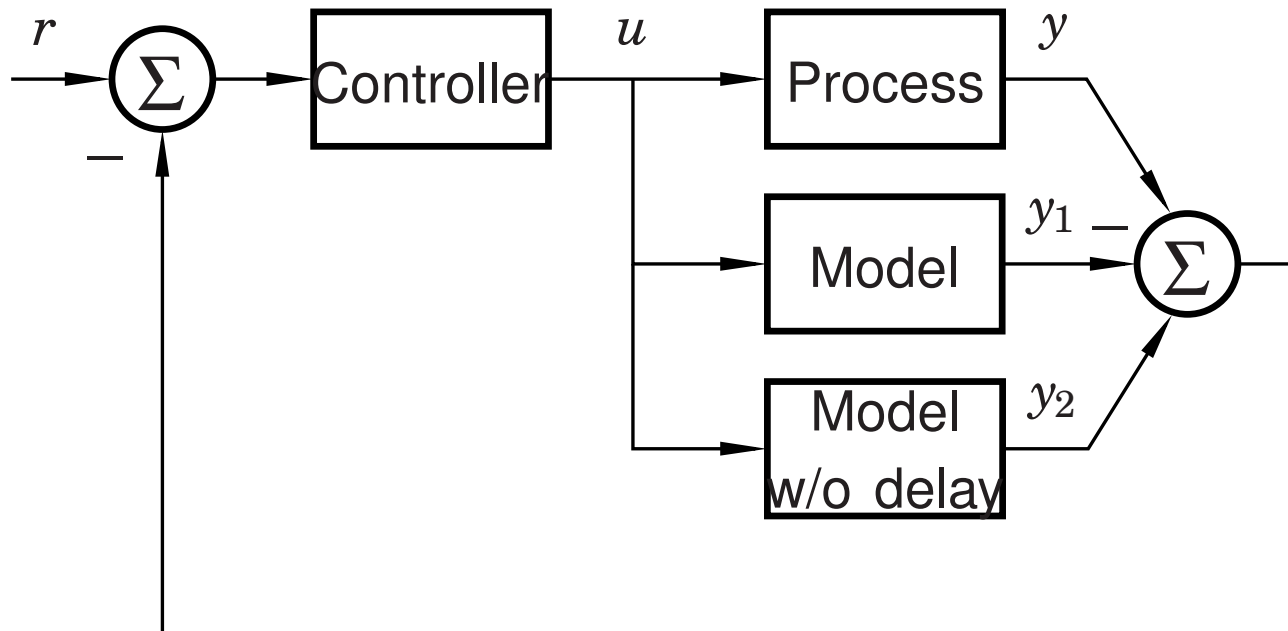
If the delay is constant and known, it is straightforward to compensate for it in the design.

Delay compensation:



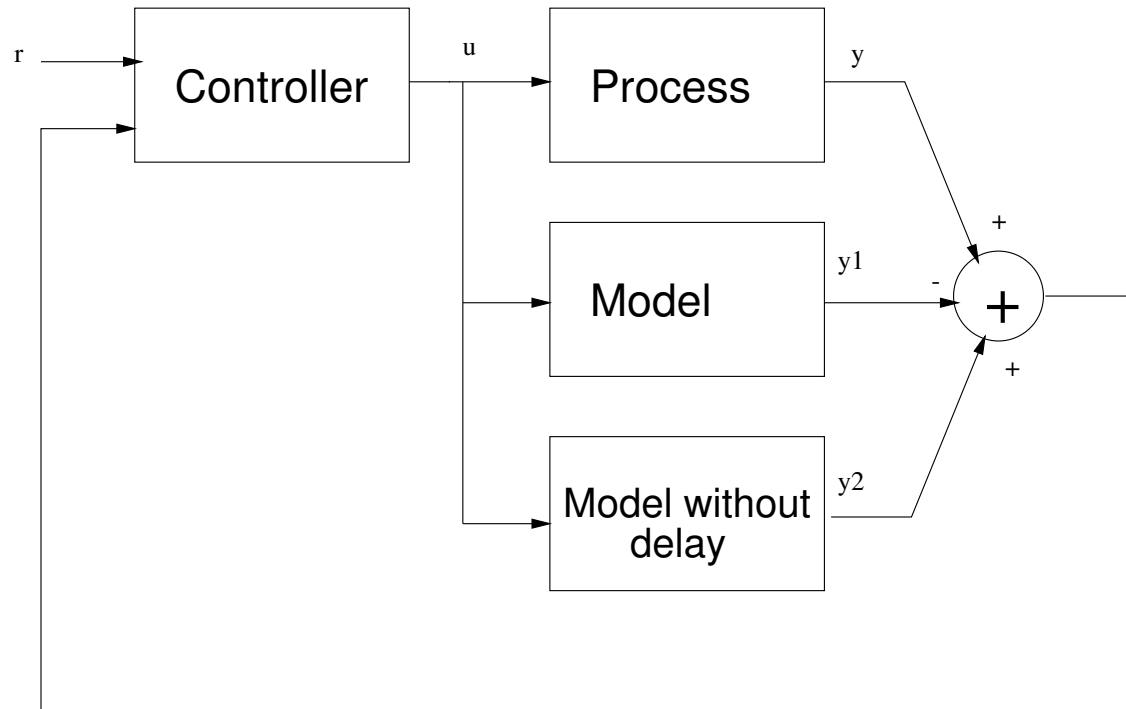
Delay compensation using Smith predictor

Idea: control against simulated model without delay:



- Requires **accurate** and **stable** model

The Smith Predictor



With a perfect model the controller does not see any delay

The control performance the same as without any delay (with the exception that the output will be delayed)

The Smith Predictor

Assume that the process is given by $P(s) = P_0(s)e^{-sL}$ and that we have a perfect model $\hat{P}(s) = P(s)$.

This gives the transfer function

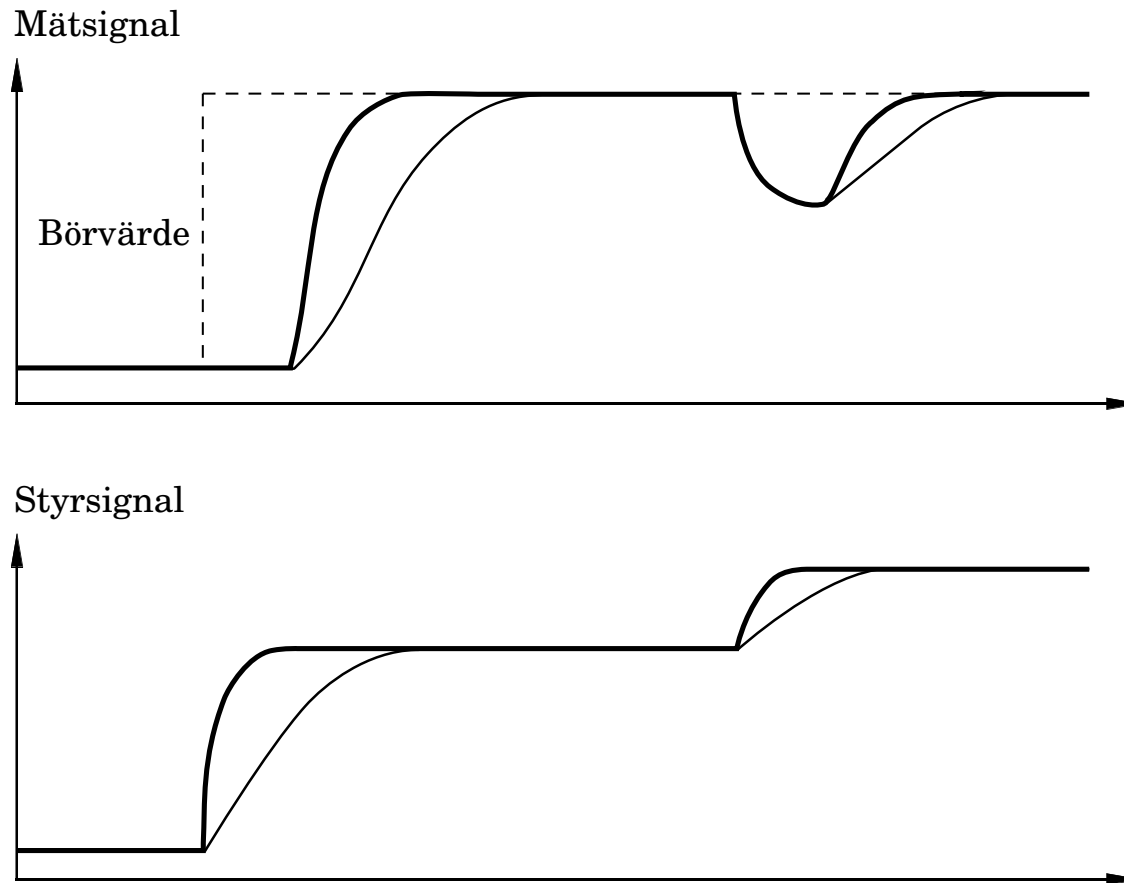
$$Y(s) = \frac{P_0 C}{1 + P_0 C} e^{-sL} R(s)$$

The same as if without any delay + a pure delay

Ideally the controller can be designed for without delay

In practice due to model errors and disturbances the delay must be taken into account in the control design (a more conservative design)

PI versus Smith



However, a delay compensating controller can never undo the delay

Delays in Discrete Time

Include the delay in the discrete time model:

$$\frac{dx(t)}{dt} = Ax(t) + Bu(t - \tau), \quad \tau < h$$

$$x(kh + h) - \Phi x(kh)$$

$$= \int_{kh}^{kh+h} e^{A(kh+h-s)} B u(s - \tau) ds$$

$$= \int_{kh}^{kh+\tau} e^{A(kh+h-s)} B ds u(kh - h) + \int_{kh+\tau}^{kh+h} e^{A(kh+h-s)} B ds u(kh)$$

$$= \Gamma_1 u(kh - h) + \Gamma_0 u(kh)$$

LTI system!

A state-space model (with extra state $z(kh) = u(kh - h)$)

$$\begin{pmatrix} x(kh + h) \\ z(kh + h) \end{pmatrix} = \begin{pmatrix} \Phi & \Gamma_1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} x(kh) \\ z(kh) \end{pmatrix} + \begin{pmatrix} \Gamma_0 \\ I \end{pmatrix} u(kh)$$

Can easily be extended to $\tau > h$

Design:

- apply arbitrary discrete time design using the augmented model
- e.g., LQG-design

LQG with Deadtime Compensation

Designs a discrete-time LQG controller with direct term for a continuous-time system assuming a constant sampling interval h and a constant time delay τ .

Controller:

$$u(k) = -L\hat{x}_e(k|k)$$

$$\hat{x}_e(k|k) = \hat{x}_e(k|k-1) + K_f(y(k) - C_e\hat{x}_e(k|k-1))$$

$$\hat{x}_e(k+1|k) = \Phi_e\hat{x}_e(k|k-1) + \Gamma_e u(k) + K(y(k) - C_e\hat{x}_e(k|k-1))$$

Used in most of our examples.

Jitterbug command: `lqgdesign`

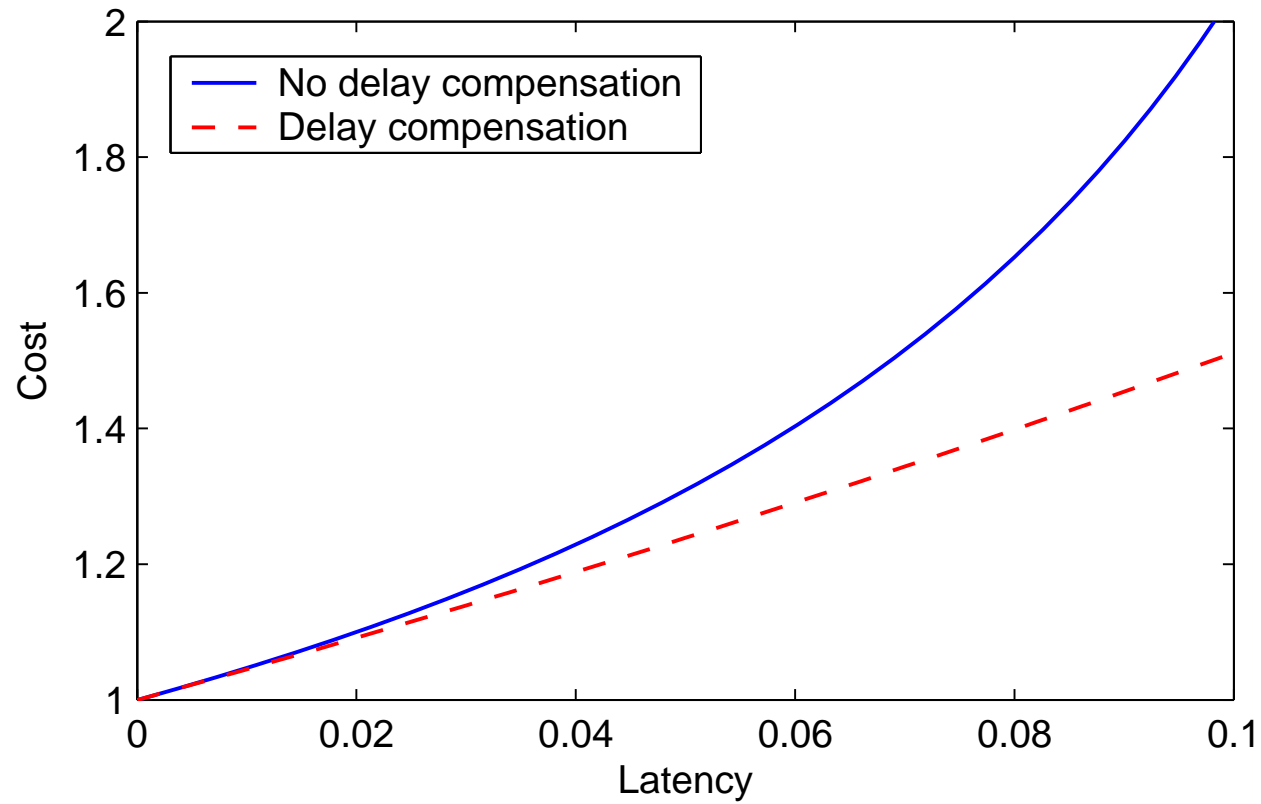
Why is Jitter Bad?

- The controllers were designed assuming a constant h
- The jitter can be interpreted as a process disturbance
- Very hard to analyze in the general case
 - counter-intuitive anomalies can be found
- The Jitterbug toolbox can be used to evaluate the effect of jitter for a given case
- Many jitter compensation schemes have been developed

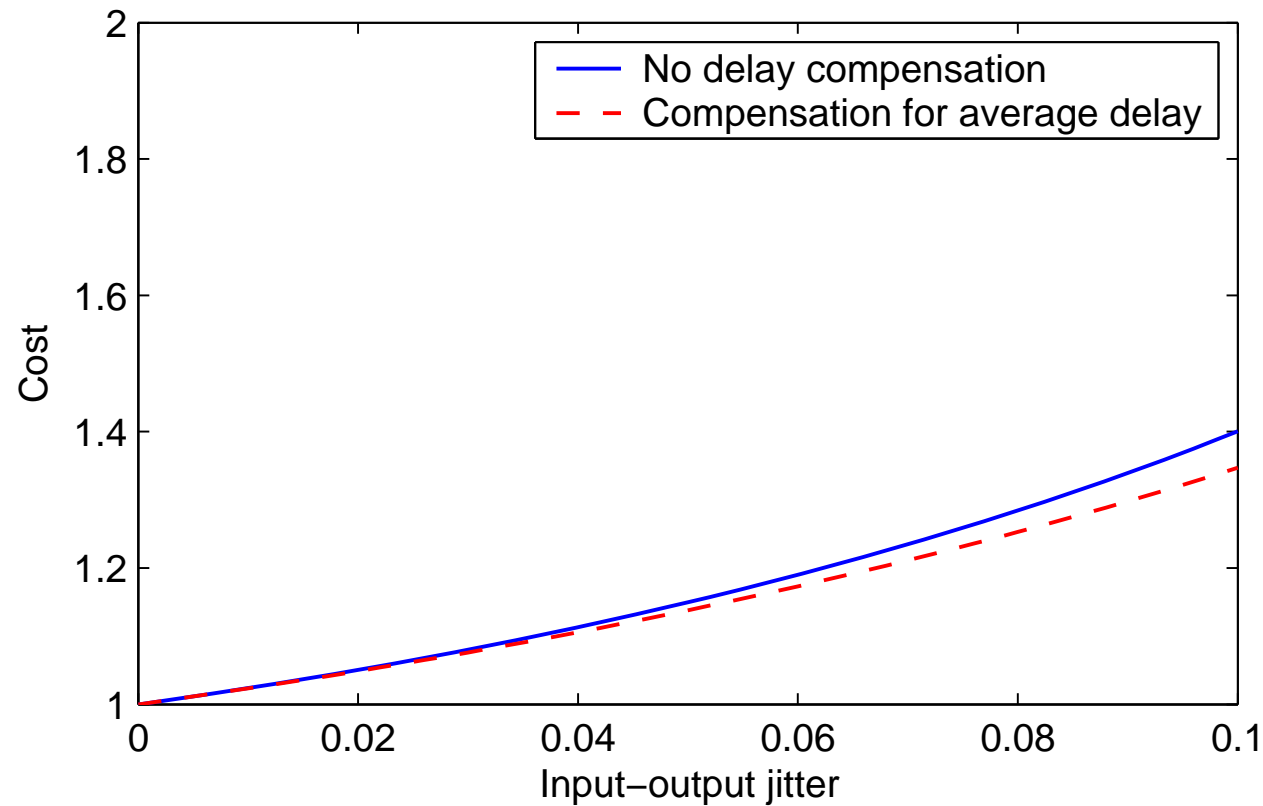
Example: DC Servo with IO Latency and Jitter

- Process: $P(s) = \frac{1}{s(s+1)}$
- LQG controller with or without delay compensation
- Process noise $R_{1c} = 1$, measurement noise $R_2 = 0.01$
- Cost function: $J = \mathbf{E}\{y^2(t) + 0.001u^2(t)\}$
- Periodic sampling with $h = 0.1$
- Constant or random (uniform distribution) IO latency

Constant Input-Output Latency



Input-Output Jitter



Note: having uniform jitter J is only slightly worse than having a constant latency of $L = J/2$.

Compensation for Sampling Jitter

Rule of thumb: Jitter that is less than 10% of the nominal sampling period need not to be compensated for

Two approaches:

- Gain scheduling
- Robust design methods

Gain Scheduling

Assume that the sampling period can be measured

Store several sets of pre-calculated controller parameters in a table with the sampling period as input parameter.

Switch controller parameters when the sampling period changes

Assumes that the sampling period varies slowly, i.e., not so realistic for jitter

May cause switching transients

Gain Scheduling

What if the sampling period varies fast?

Parameterize the controller parameters in terms of the sampling period

For example:

$$\frac{dx(t)}{dt} \approx \frac{x(t_{k+1}) - x(t_k)}{h_k}$$

Works often well for low order controllers, e.g., PID.

Ad hoc method with no formal guarantees

Robust Design Methods

Design the controller to be robust against timing variations

Several robust design methods are available

- H_∞
- Quantitative Feedback Theory (QFT)
- μ -design
- ...

Session Outline

- Control Loop Timing
- Temporal Non-Determinism
- **Switching**
- The Jitter Margin
- Subtask Scheduling

Switching Controller Task Parameters

Jitter in sampling and latency

- stochastic changes in controller task parameters (period and execution time)
- caused by the implementation platform

Sometimes it can be useful to change the controller task parameters intentionally

- deterministic changes in order to adapt to changing work loads
- generated by a controller when it changes modes (e.g. changes its execution time demands)
- generated by a scheduler when the resources change

May cause both scheduling and control problems

Mode Changes and Scheduling

A task set that is schedulable under fixed priority scheduling before the mode change occurs and after the mode change has occurred may not necessarily be schedulable during the mode change (in transition phase)

Special mode change protocols are needed

Easier under EDF (Earliest Deadline First) scheduling than under fixed priority scheduling

Switching-Induced Instabilities

Deterministic changes of task parameters may lead to instability

Example:

Process:

$$\dot{x} = Ax + Bu$$

$$y = Cx$$

where

$$A = \begin{bmatrix} 0 & 1 \\ -10000 & -0.1 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad C = [1 \quad 0]$$

The system is stable with poles in $p_{1,2} = -0.05 \pm 100i$.

Sampled with $h_1 = 0.002\text{s}$ and $h_2 = 0.094\text{s}$

$$\mathbf{x}_{k+1} = \Phi_i \mathbf{x}_k + \Gamma_i \mathbf{u}_k$$

$$\mathbf{y}_k = \mathbf{C}_i \mathbf{x}_k$$

$$i \in \{1, 2\}$$

where $\Phi_i = e^{Ah_i}$, $\Gamma_i = \int_0^{h_i} e^{As} B ds$

Both discrete-time systems are stable

State feedback controllers: $u = -K_i x$

LQ-design:

$$J = \int_0^{\infty} (x(t)^T Q_c x(t) + u(t)^T R u(t)) dt$$

with

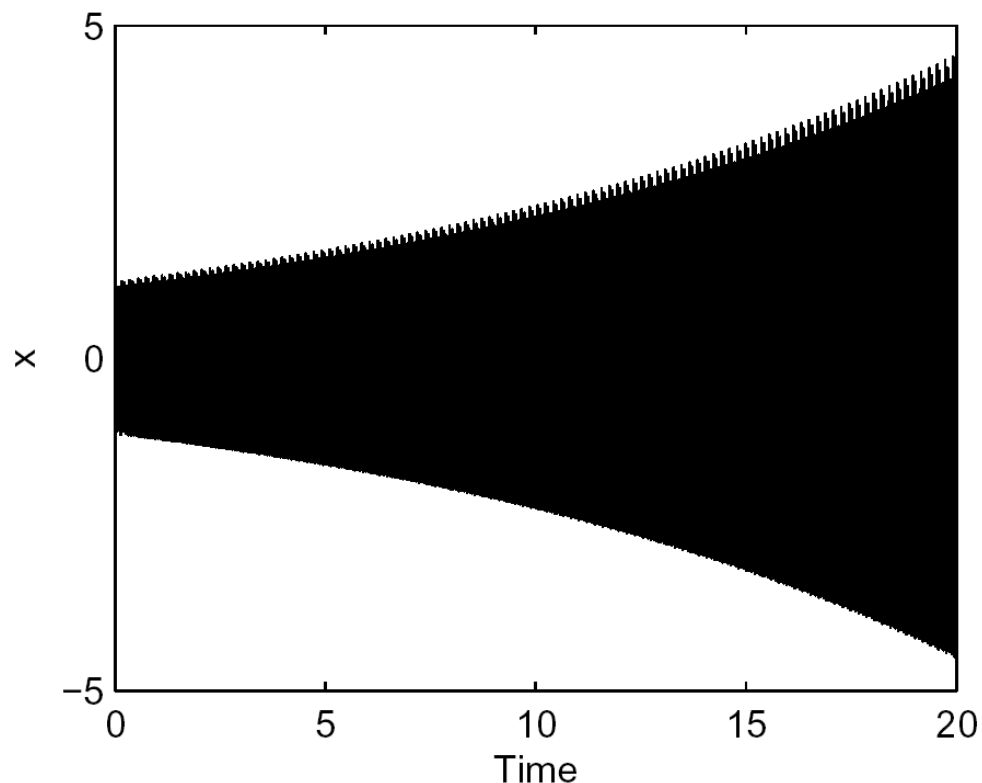
$$Q_c = \begin{bmatrix} 20000 & 0 \\ 0 & 20000 \end{bmatrix} \quad R = 50$$

Both closed-loop systems, $\Phi_i - \Gamma_i K_i$, are stable

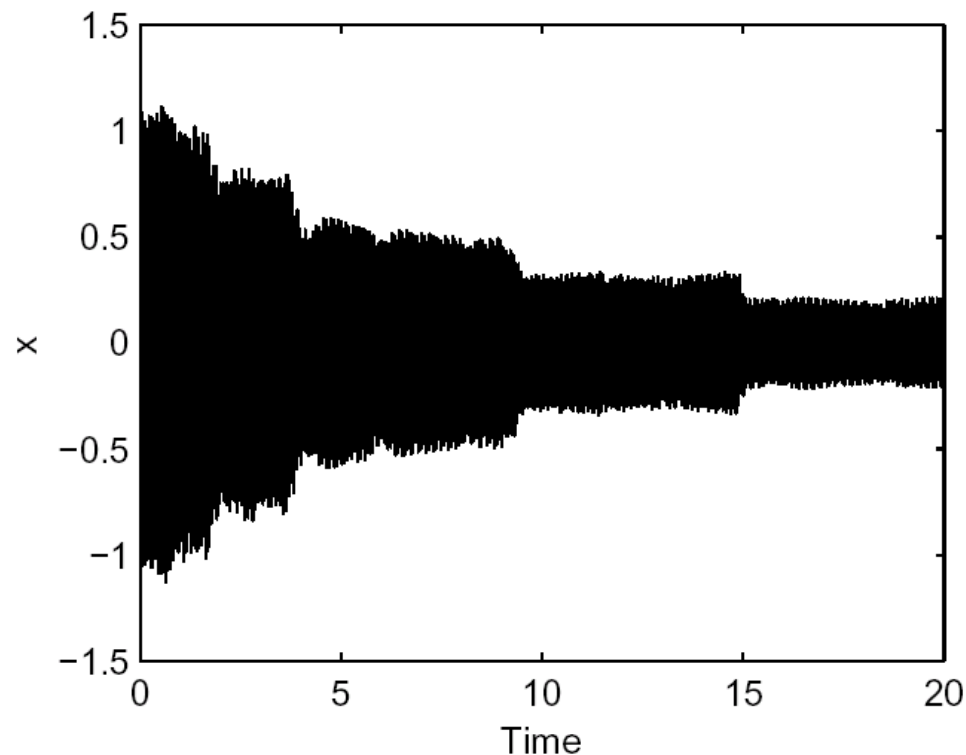
- eigenvalues inside unit circle

However, the switching sequence $h_1, h_2, h_2, h_1, h_2, h_2, \dots$ gives an unstable system

- eigenvalues of $(\Phi_2 - \Gamma_2 K_2)^2 (\Phi_1 - \Gamma_1 K_1)$ outside the unit circle



If we instead switch between the two controller stochastically using the relative frequency 67% for h_2 and 33% for h_1 the resulting system is stable (in the mean-square sense).



The phenomenon can in principle occur also in other cases:

- change of sampling interval for the same controller
- change of input output latency

However, it is rare and so far we have not seen any “realistic” examples where it has occurred.

Switching & Controller State

Switching sampling intervals may also cause problems for controllers on input-output form

$$u(k) = a_1y(k) + a_2y(k - 1) + a_3y(k - 2) + b_1u(k - 1) + b_2u(k - 2)$$

Remedy:

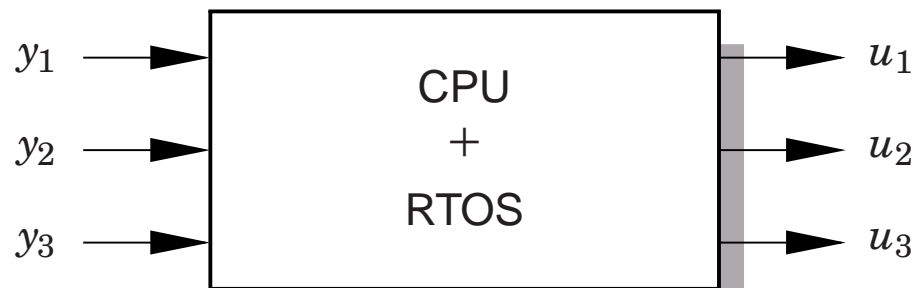
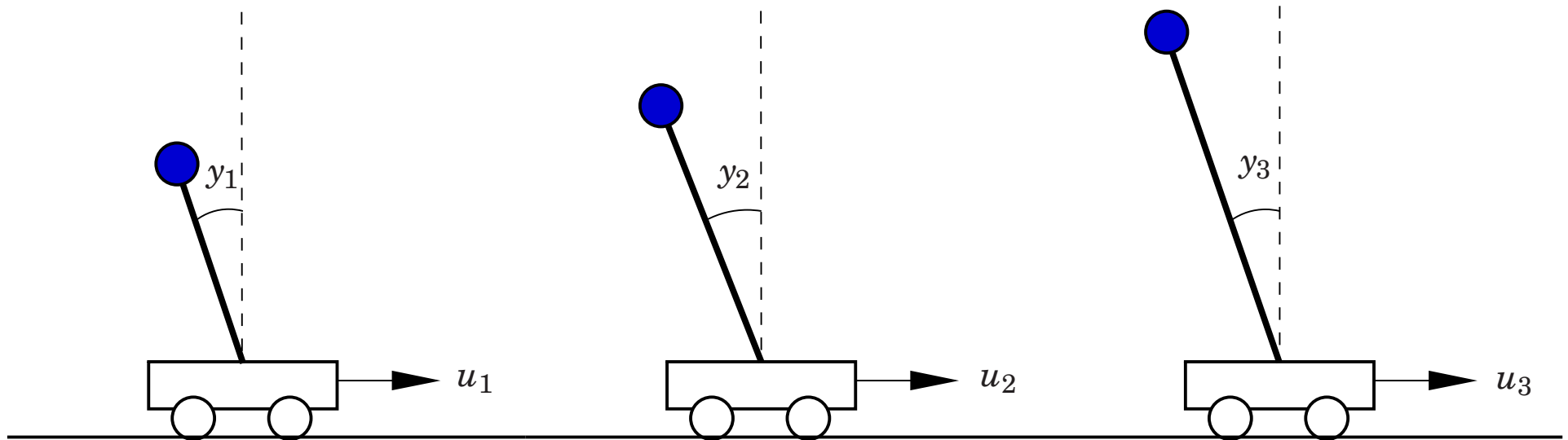
- only allow switches in stationarity
- use an observer (Kalman filter) to estimate the signal values at the new points in time

Session Outline

- Control Loop Timing
- Temporal Non-Determinism
- Switching
- **The Jitter Margin**
- Subtask Scheduling

Inverted Pendulum Example

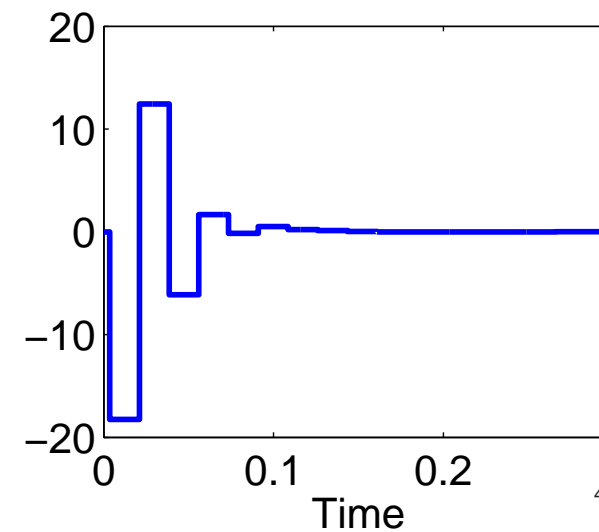
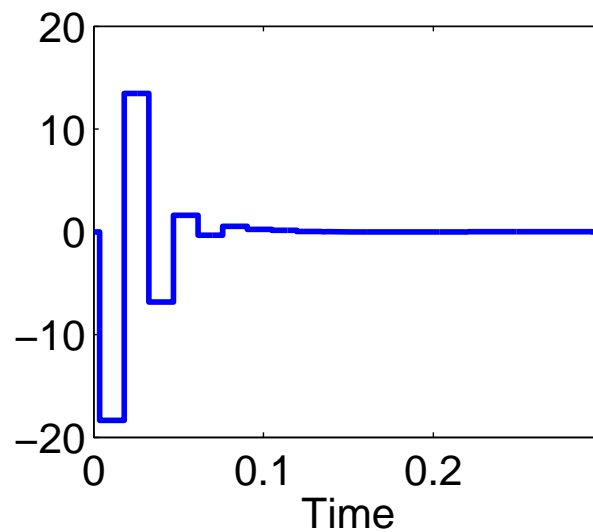
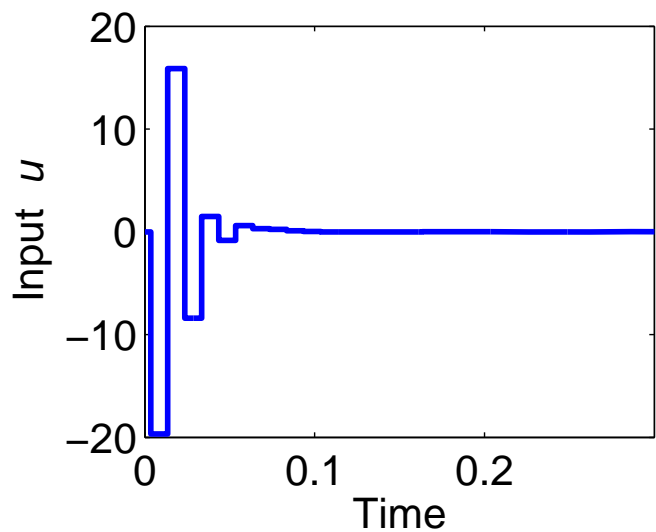
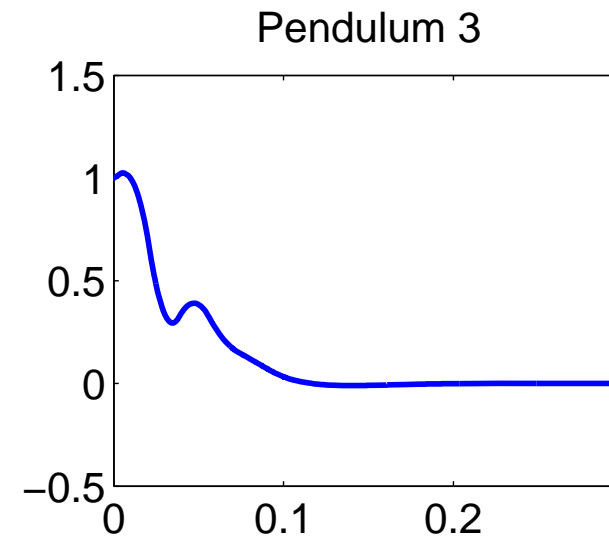
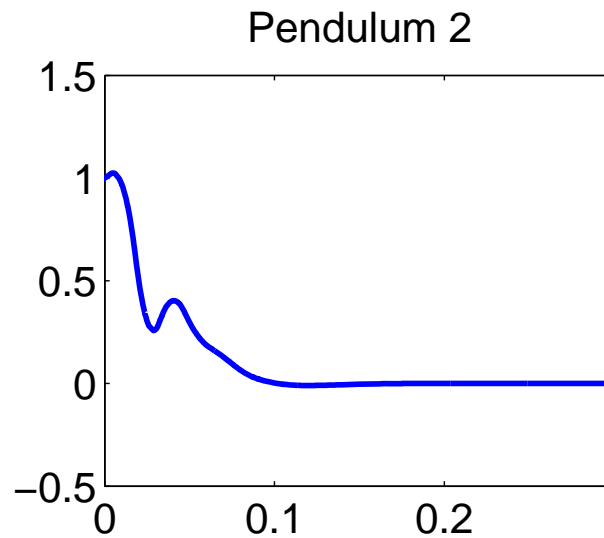
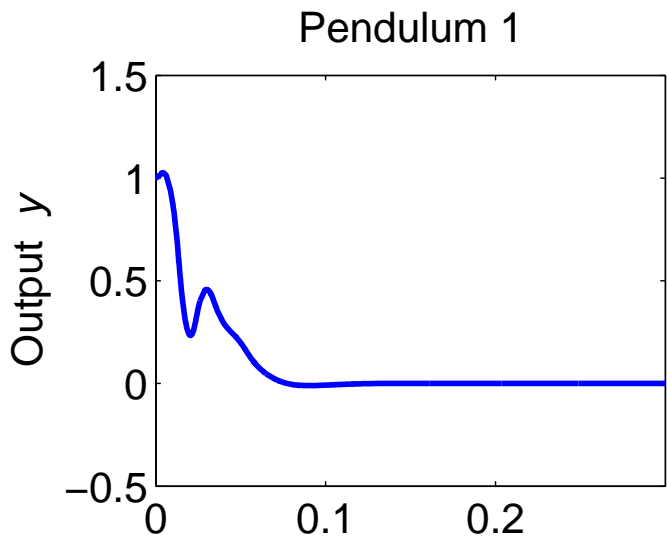
Control of three inverted pendulums using one CPU:



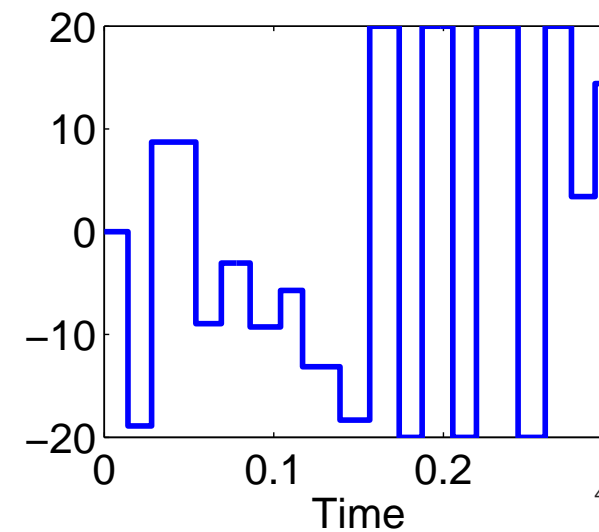
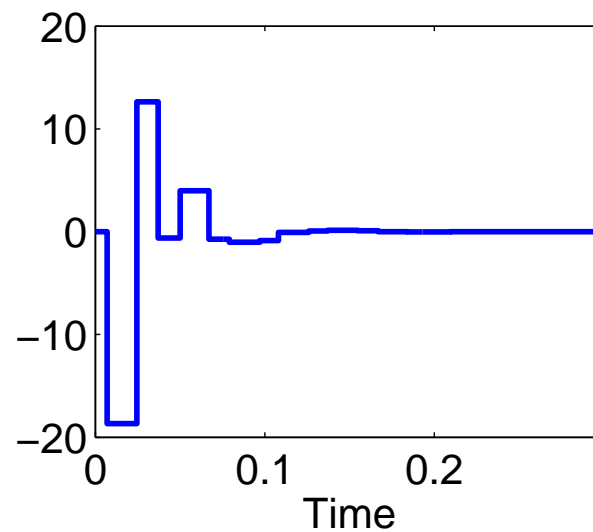
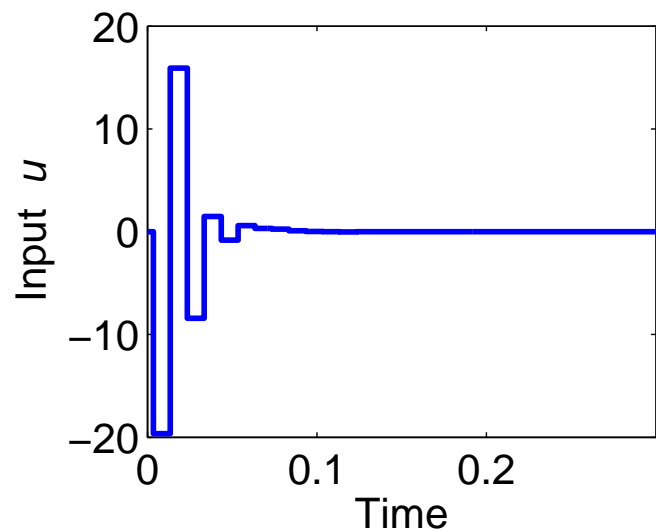
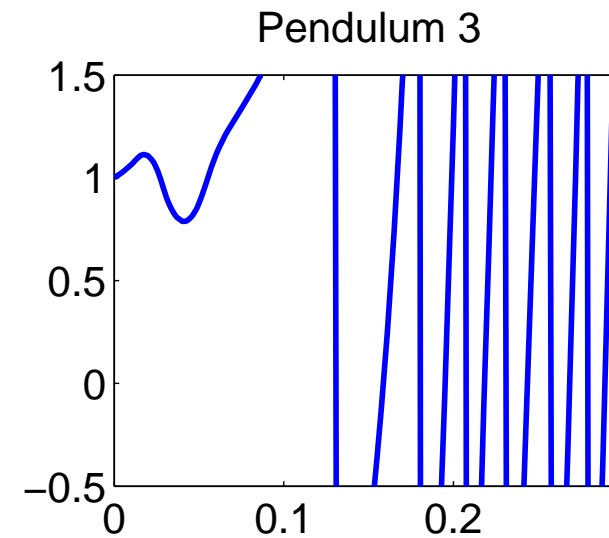
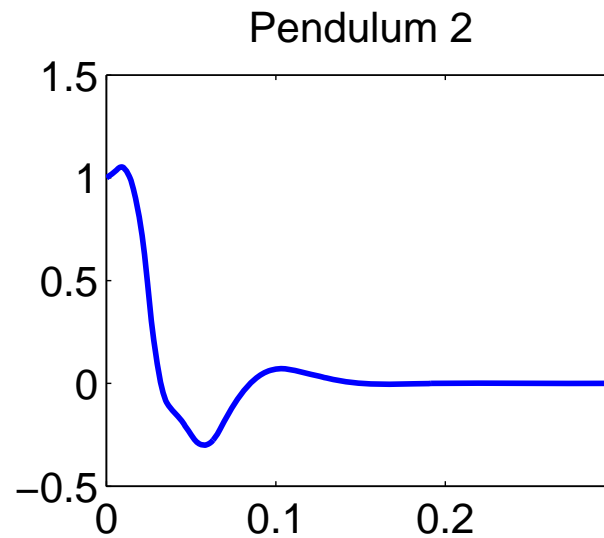
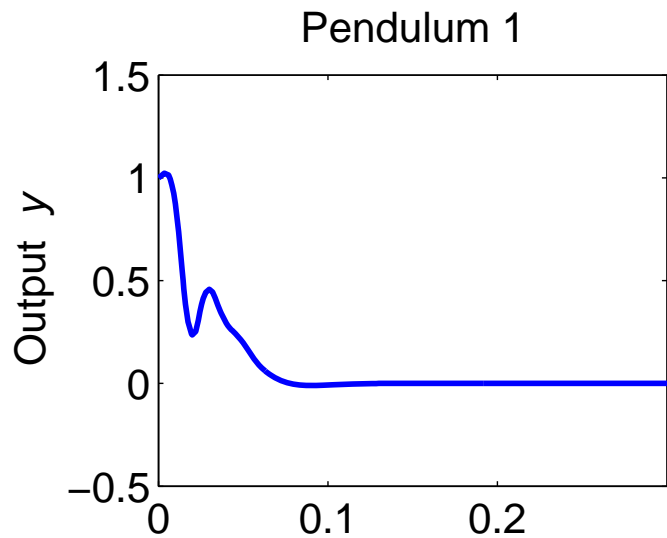
Design

- Discrete-time LQG controllers
- Sampling intervals: $(T_1, T_2, T_3) = (10, 14.5, 17.5)$ ms
- Assumed execution time: $C_i = 3.5$ ms
- Controllers designed assuming delay of 3.5 ms
 - Jitterbug command: `lqgdesign`
- Schedulable under both RM and EDF (with $D_i = T_i$)

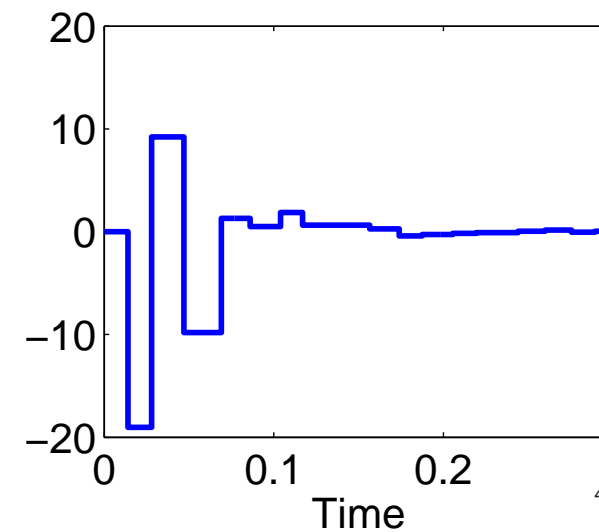
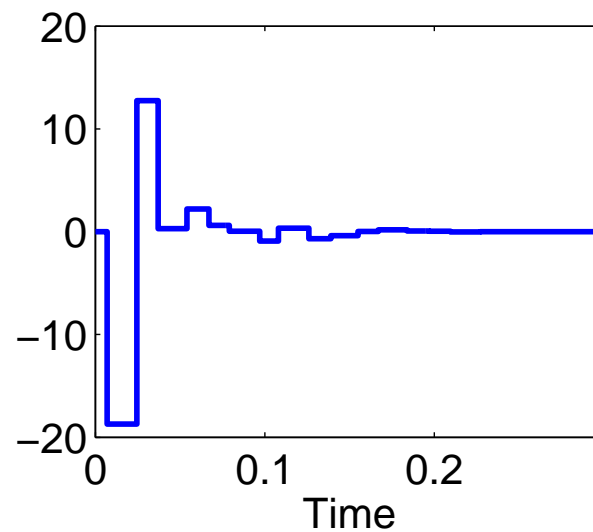
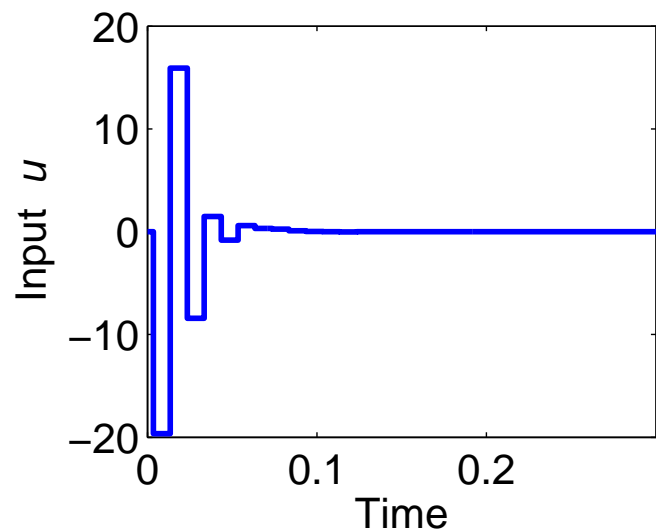
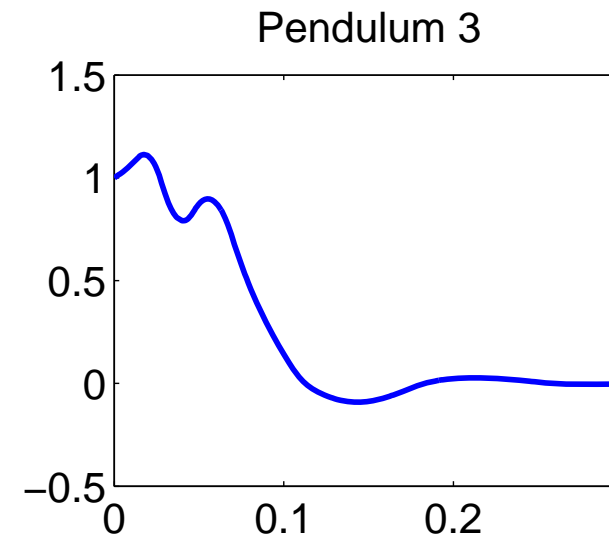
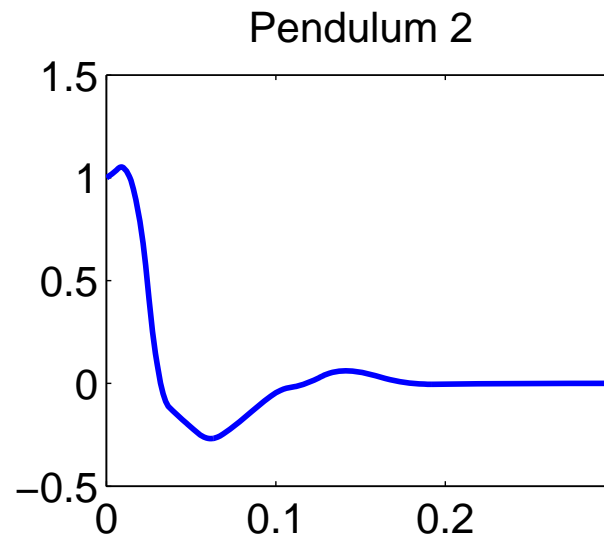
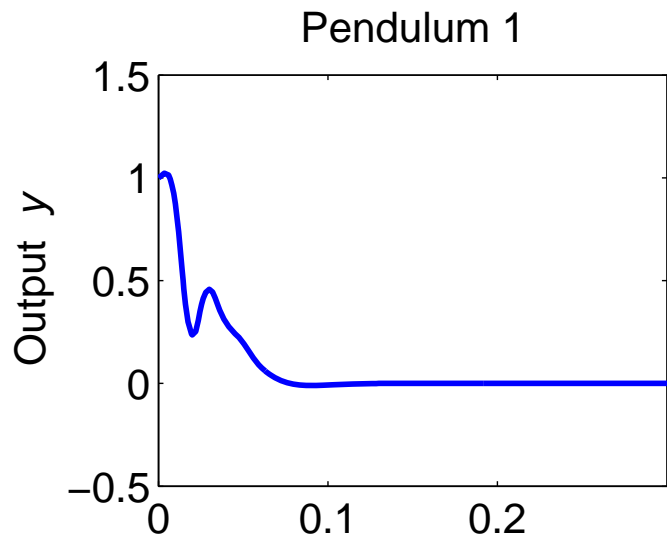
Simulation 1 – No Scheduling Interference



Simulation 2 – Rate-Monotonic Scheduling



Simulation 3 – Earliest-Deadline-First Scheduling



Questions

- How much jitter is there under various scheduling policies?
 - Simulation
 - Jitter analysis
- How much jitter do the control loops tolerate?
 - Simulation
 - The jitter margin

Jitter analysis + the jitter margin give *hard stability results*

Jitter Analysis – Rate-Monotonic Scheduling

- R_i – worst-case response time of task i

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- R_i^b – *best-case* response time of task i

$$R_i^b = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^b}{T_j} - 1 \right\rceil C_j$$

- J_i – worst-case input-output jitter of task i :

$$J_i = R_i - R_i^b$$

(Analysis for earliest-deadline-first scheduling also exists)

The Pendulum Example – RM Scheduling

Task	T	C	R	R^b	J
1	10	3.5	3.5	3.5	0
2	14.5	3.5	7.0	3.5	3.5
3	17.5	3.5	14.0	3.5	10.5

The Delay Margin

- L_m – delay margin, the longest delay a loop can tolerate without becoming unstable
- Simple to compute
 - Continuous-time system: $L_m = \varphi_m / \omega_c$
 - * φ_m – phase margin [rad]
 - * ω_c – cross-over frequency [rad/s]
 - Sampled-data system: need to compute a root locus with respect to the delay

Delay Margins in the Pendulum Example

The maximum delay is equal to the response time R

Compute the delay margin L_m for each controller:

Task	T	C	R	L_m
1	10	3.5	3.5	9.8
2	14.5	3.5	7.0	12.5
3	17.5	3.5	14.0	14.6

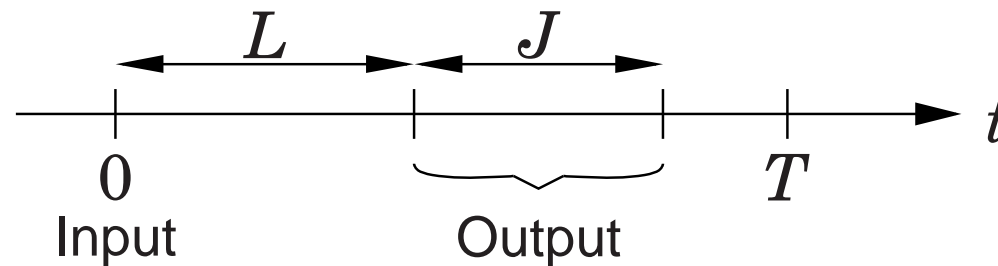
$\forall i : R_i < L_{mi}$. Still, system 3 was seen to be unstable!

The delay margin is only valid for constant delays!

The Jitter Margin

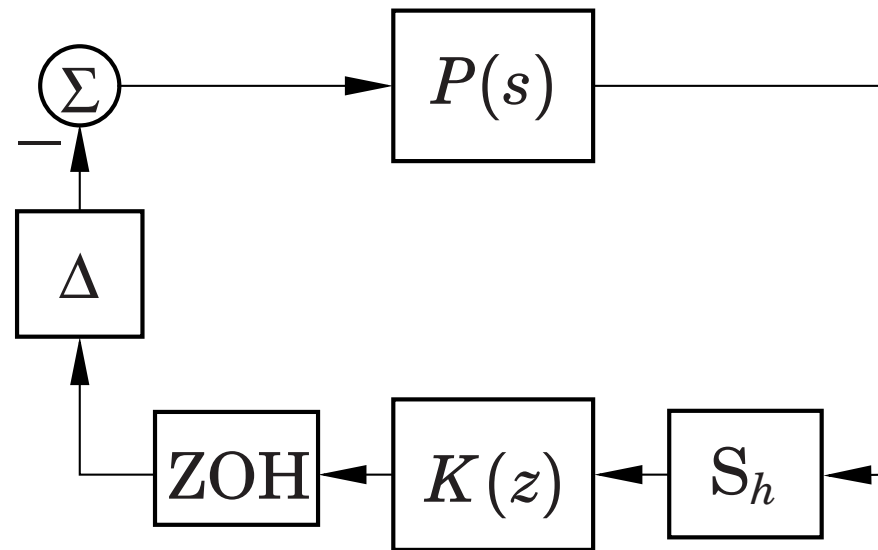
Assumptions:

- Periodic sampling (high-prio/interrupt-driven)
- Arbitrarily time-varying input-output delay $\Delta \in [L, L + J]$
 - L – constant part
 - J – jitter



Jitter margin $J_m(L)$ – the largest J for which stability can be guaranteed given a value of L

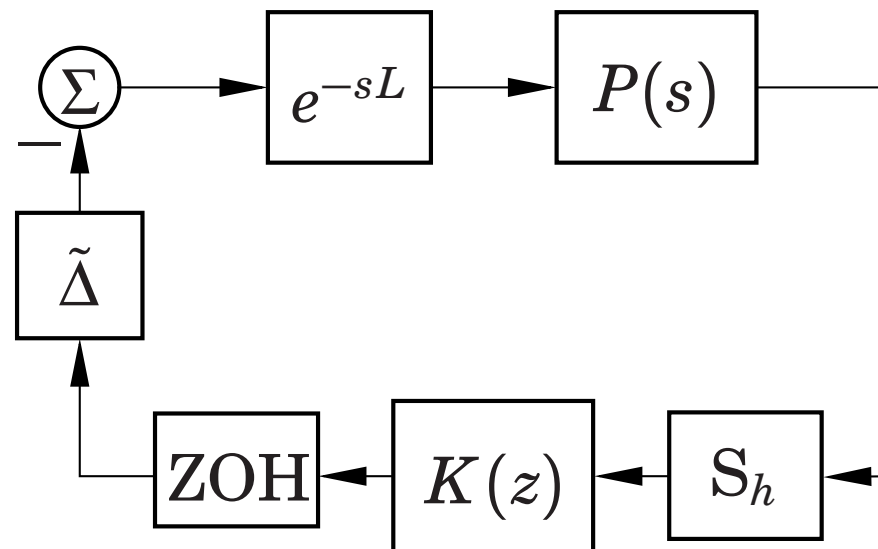
Checking Stability



- Continuous-time plant $P(s)$
- Discrete-time controller $K(z)$
- Arbitrarily time-varying delay $\Delta \in [L, L + J]$
- Closed-loop system assumed stable for $\Delta = L$

Checking Stability

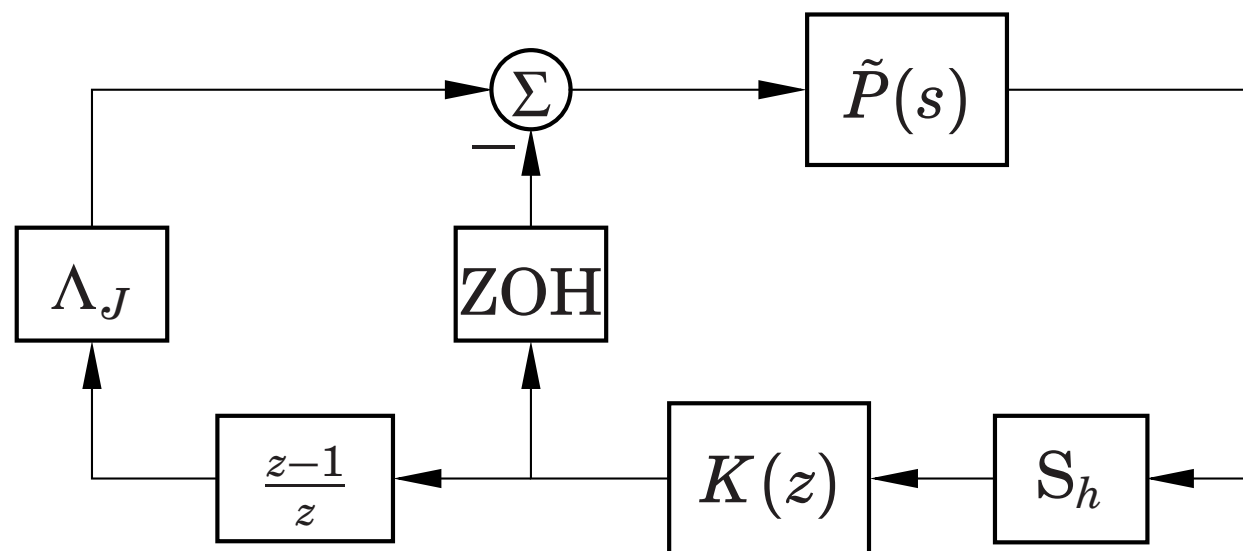
Include the constant delay L in the plant:



- New time-varying delay $\tilde{\Delta} \in [0, J]$
- New plant $\tilde{P}(s) = P(s)e^{-sL}$

Checking stability

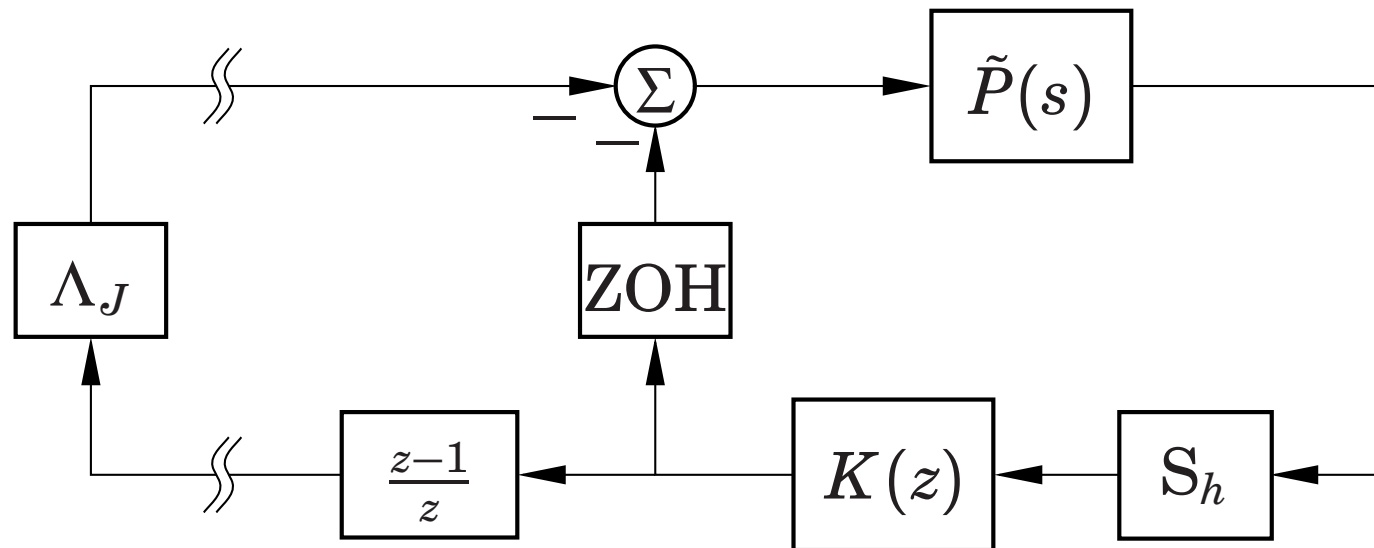
Rewrite the control output as one direct path and one error path:



- Difference operator $\frac{z-1}{z}$
- Time-varying gate function Λ_J (open at most J seconds every sample)

Checking Stability

Apply the small gain theorem:



- L_2 -gain of gate function: $\|\Lambda_J\| = \sqrt{J}$
- L_2 -gain of the rest:

$$\|H\| = \max_{\omega} \left\{ \left| \frac{P_{\text{alias}}(\omega)K(e^{i\omega})}{1 + P_{\text{ZOH}}(e^{i\omega})K(e^{i\omega})} \right| |e^{i\omega} - 1| \right\}$$

Checking Stability

The closed-loop system is stable if $\|\Lambda_J\| \|H\| < 1 \Leftrightarrow$

$$\left| \frac{P_{\text{alias}}(\omega) K(e^{i\omega})}{1 + P_{\text{ZOH}}(e^{i\omega}) K(e^{i\omega})} \right| < \frac{1}{\sqrt{J} |e^{i\omega} - 1|}, \quad \forall \omega \in [0, \pi]$$

Here,

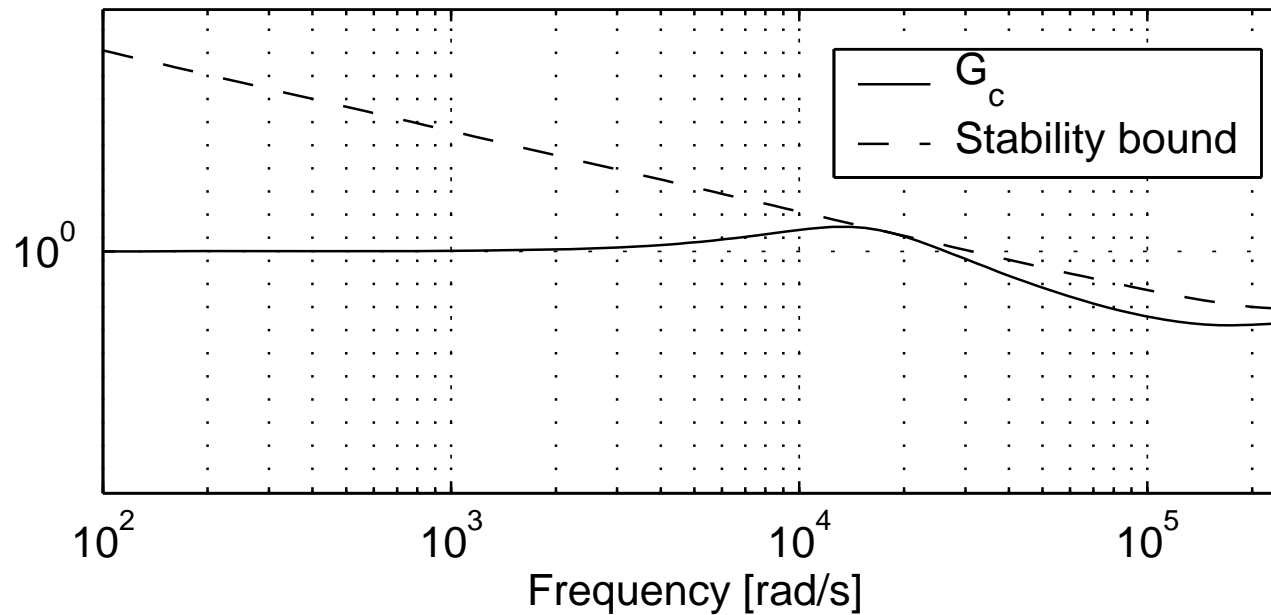
- $P_{\text{alias}}(\omega) = \sqrt{\sum_{k=-\infty}^{\infty} \left| \tilde{P}\left(i\left(\omega + 2\pi k\right)\frac{1}{h}\right) \right|^2}$
- $P_{\text{ZOH}}(z)$ is the ZOH-discretization of $\tilde{P}(s)$

(For small h , $P_{\text{alias}}(\omega) \approx P_{\text{ZOH}}(e^{i\omega})$)

Checking Stability

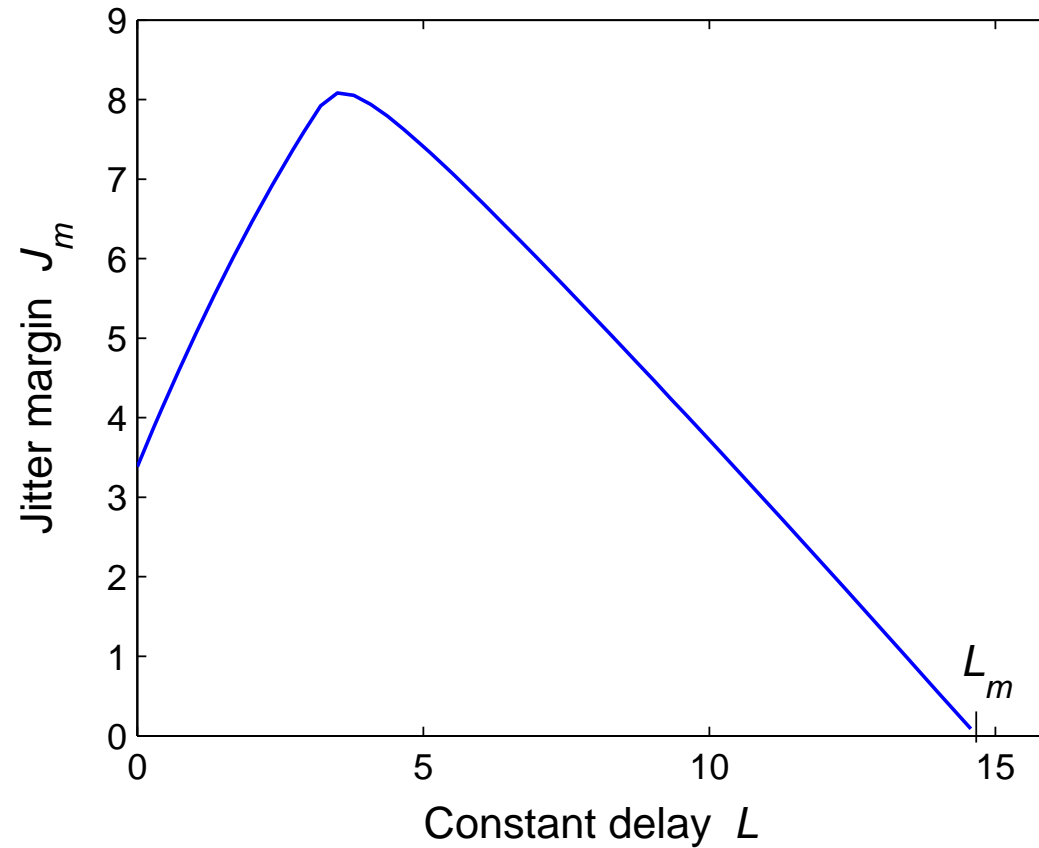
Graphical test:

Continuous-time plant, discrete-time controller



Jitter Margin – Example

$J_m(L)$ for pendulum controller 3:



- $L_m = 14.6$
- $J_m(3.5) = 8.1$

Deadline Assignment

Stability of the closed-loop systems can be guaranteed by assigning relative deadlines

$$D_i = J_m(L_i) + L_i$$

and verifying that the resulting task set is schedulable.

(In our example, assigning such deadlines gives an unschedulable system under fixed-priority scheduling)

The Pendulum Example – RM

- Compute the jitter margin $J_m(L)$ for each task
- $J < J_m(L) \Rightarrow$ Stable

Task	R	$L = R^b$	J	$J_m(L)$	Stable
1	3.5	3.5	0	4.4	Yes
2	7.0	3.5	3.5	6.4	Yes
3	14.0	3.5	10.5	8.1	No?

The Pendulum Example – EDF

Task	R	$L = R^b$	J	$J_m(L)$	Stable
1	3.5	3.5	0	4.4	Yes
2	7.5	3.5	4.0	6.4	Yes
3	10.5	3.5	7.0	8.1	Yes

(In general, EDF distributes the jitter more evenly than RM.)

Session Outline

- Control Loop Timing
- Temporal Non-Determinism
- Switching
- The Jitter Margin
- **Subtask Scheduling**

Subtask scheduling

A control algorithm normally consists of two parts:

```
while (1) {  
    read_input();  
    calculate_output();  
    write_output();  
    update_state();  
    ...  
}
```

Idea: schedule the two parts as separate tasks

- reduce delay
- reduce jitter

Task models

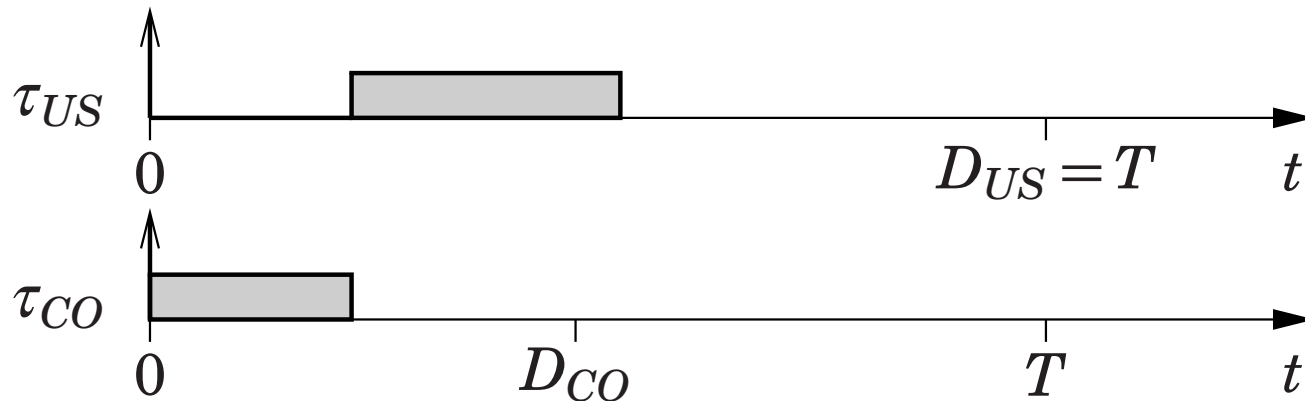
Each control task τ is divided into two subtasks:

- τ_{CO} – Calculate output
- τ_{US} – Update state
- Input and output operations are ignored in the analysis

Two possible scheduling algorithms:

- deadline-monotonic scheduling
- EDF scheduling

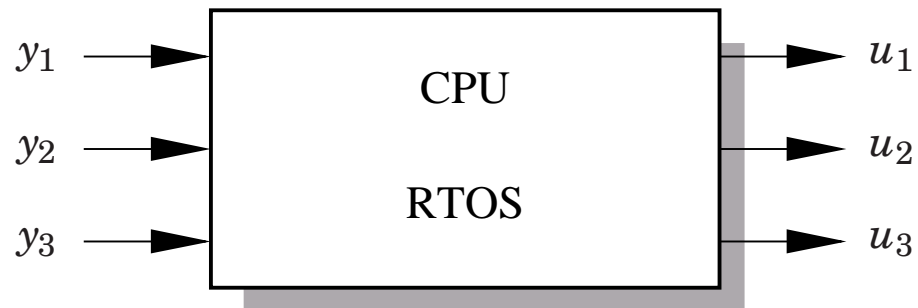
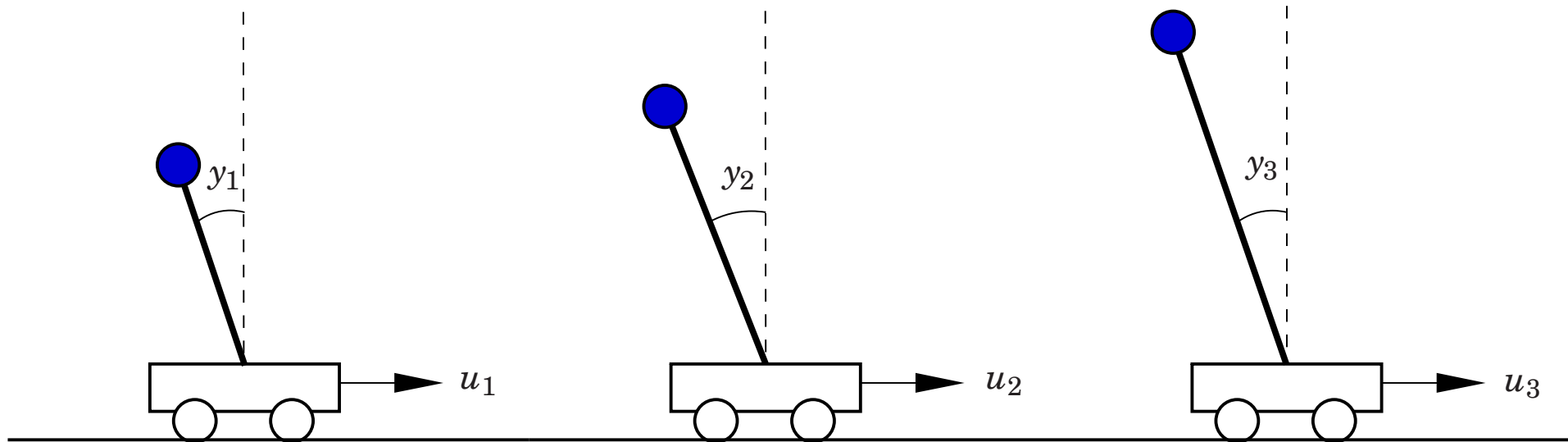
Deadline assignment (DM scheduling)



- $D_{CO} < D_{US} = T$
- We want to minimize D_{CO} for each task. Algorithm:
 1. Start by assigning $D_{CO} := T - C_{US}$ for all tasks
 2. Assign deadline-monotonic priorities to all subtasks
 3. Calculate the response time R of each subtask
 4. Assign $D_{CO} := R_{CO}$ for all tasks
 5. Repeat from 2 until no further improvement.

Example

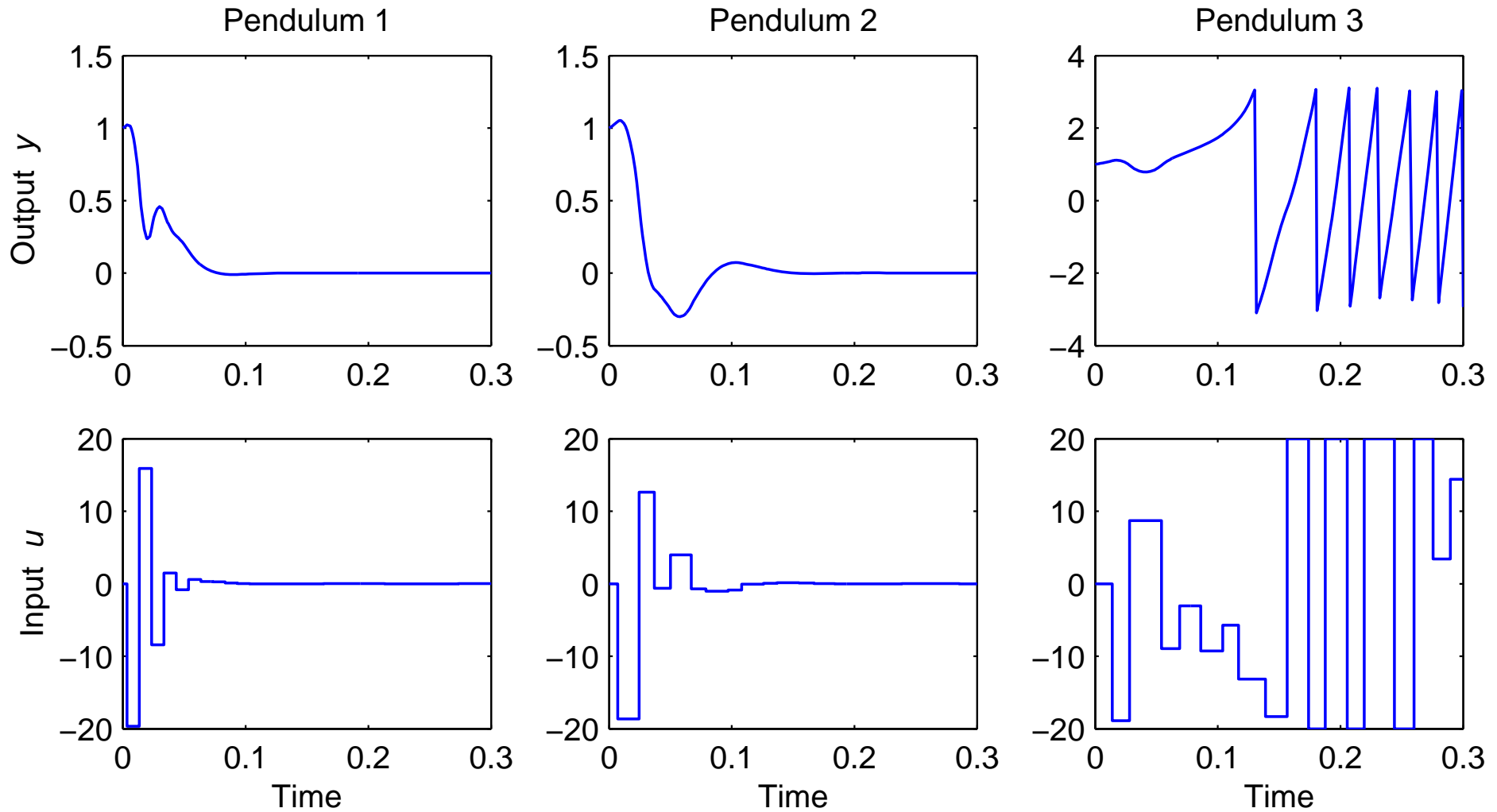
Control of three inverted pendulums using one CPU:



Example

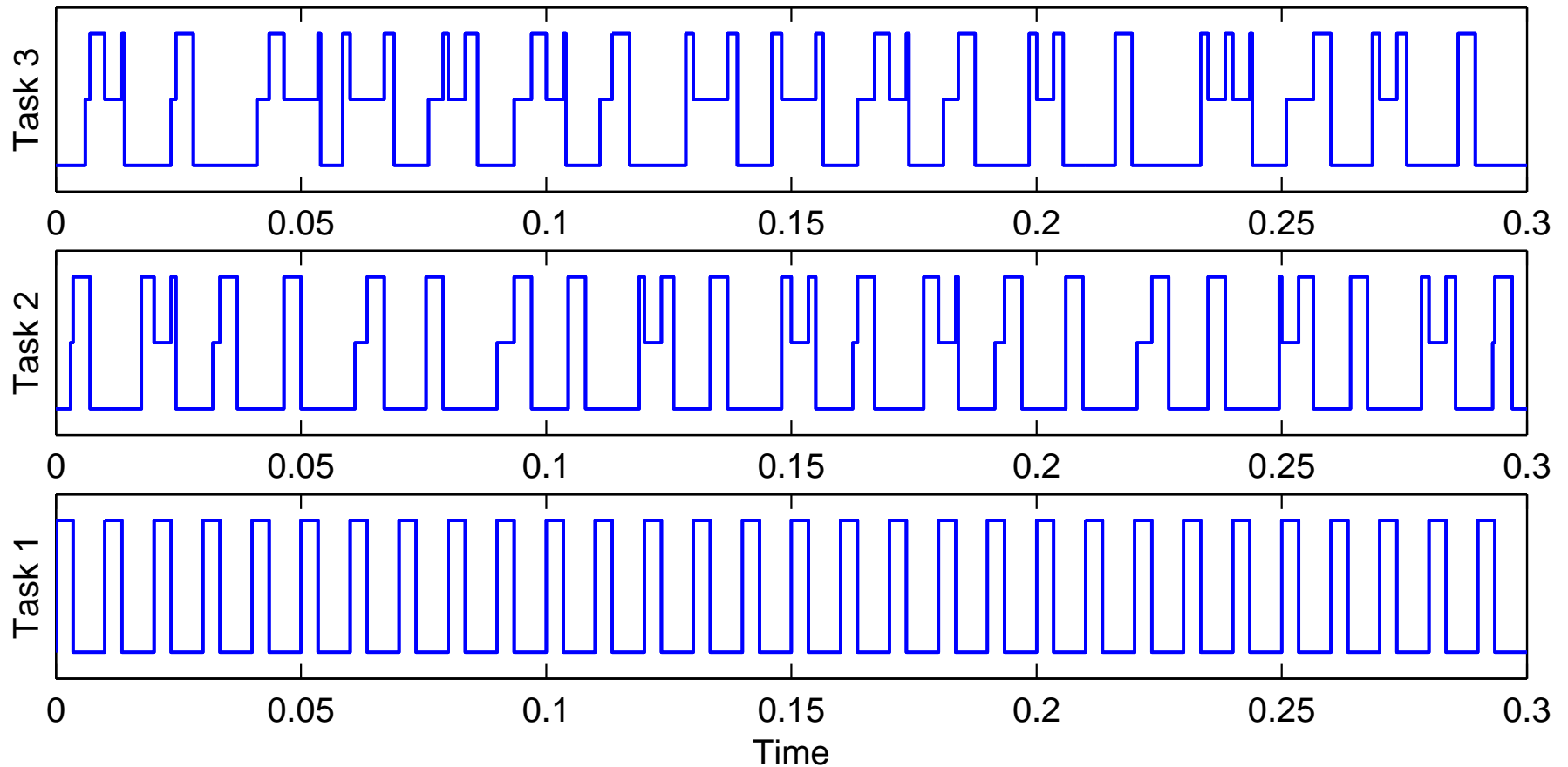
- Discrete-time LQG controllers
- Execution time: $C_i = 3.5$ ms
- Sampling intervals: $(h_1, h_2, h_3) = (10, 14.5, 17.5)$ ms
- Control delay of 3.5 ms assumed in the design

Simulation under RM scheduling



Simulation under RM scheduling

Schedule (high=running, medium=ready, low=sleeping)



- Large delay and jitter for controller 3

Subtask scheduling analysis

Each pendulum controller is divided into two subtasks:

- Calculate output: $C_{CO} = 1.5$ ms
- Update state: $C_{US} = 2.0$ ms

First iteration of algorithm:

	T	D	C	R
τ_{CO1}	10.0	8.0	1.5	1.5
τ_{US1}	10.0	10.0	2.0	3.5
τ_{CO2}	14.5	12.5	1.5	5.0
τ_{US2}	14.5	14.5	2.0	7.0
τ_{CO3}	17.5	15.5	1.5	8.5
τ_{US3}	17.5	17.5	2.0	14.0

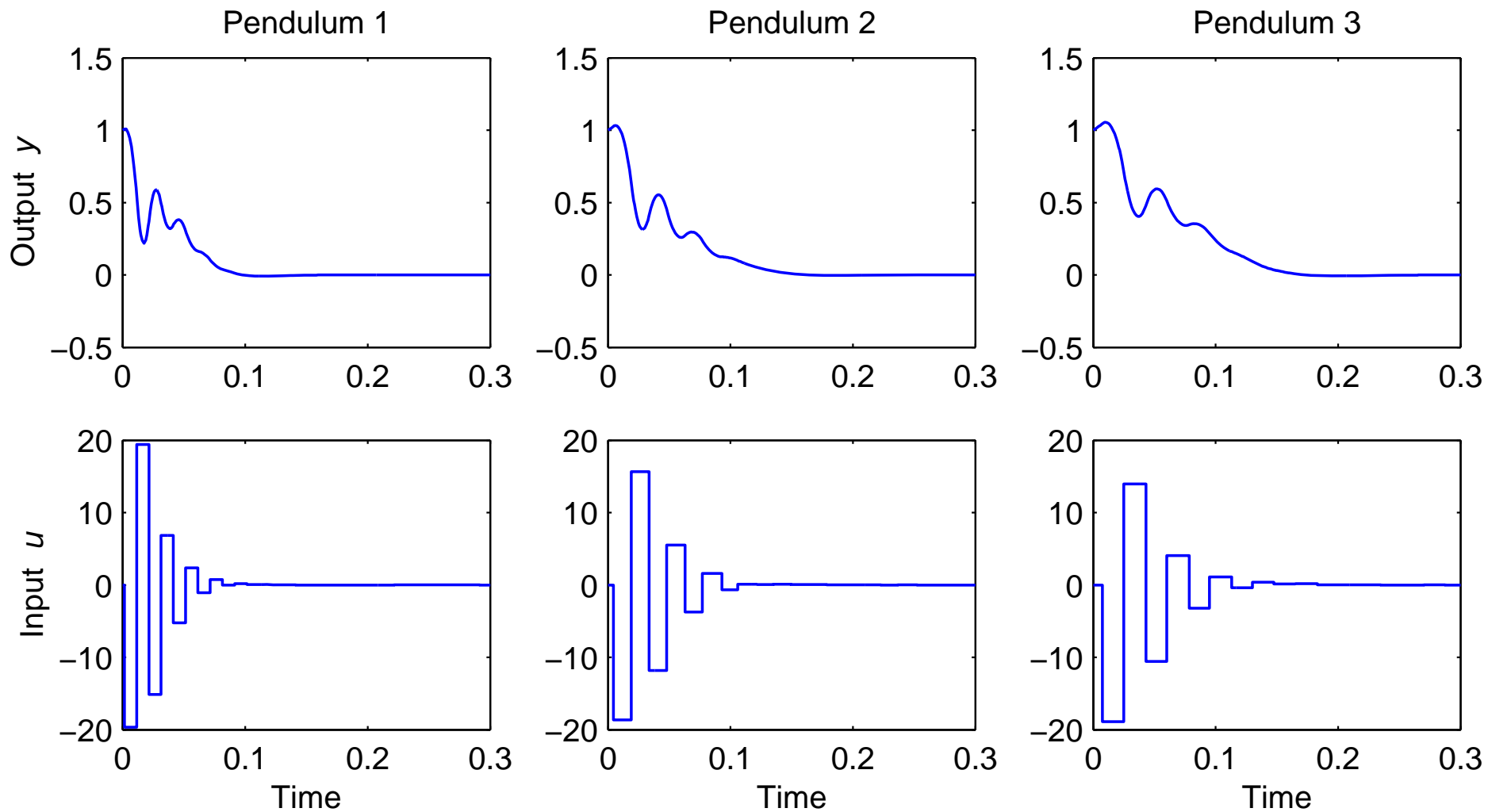
Subtask scheduling analysis

Third iteration (converged):

	T	D	C	R
τ_{CO_1}	10.0	1.5	1.5	1.5
τ_{US_1}	10.0	10.0	2.0	6.5
τ_{CO_2}	14.5	3.0	1.5	3.0
τ_{US_2}	14.5	14.5	2.0	8.5
τ_{CO_3}	17.5	4.5	1.5	4.5
τ_{US_3}	17.5	17.5	2.0	14.0

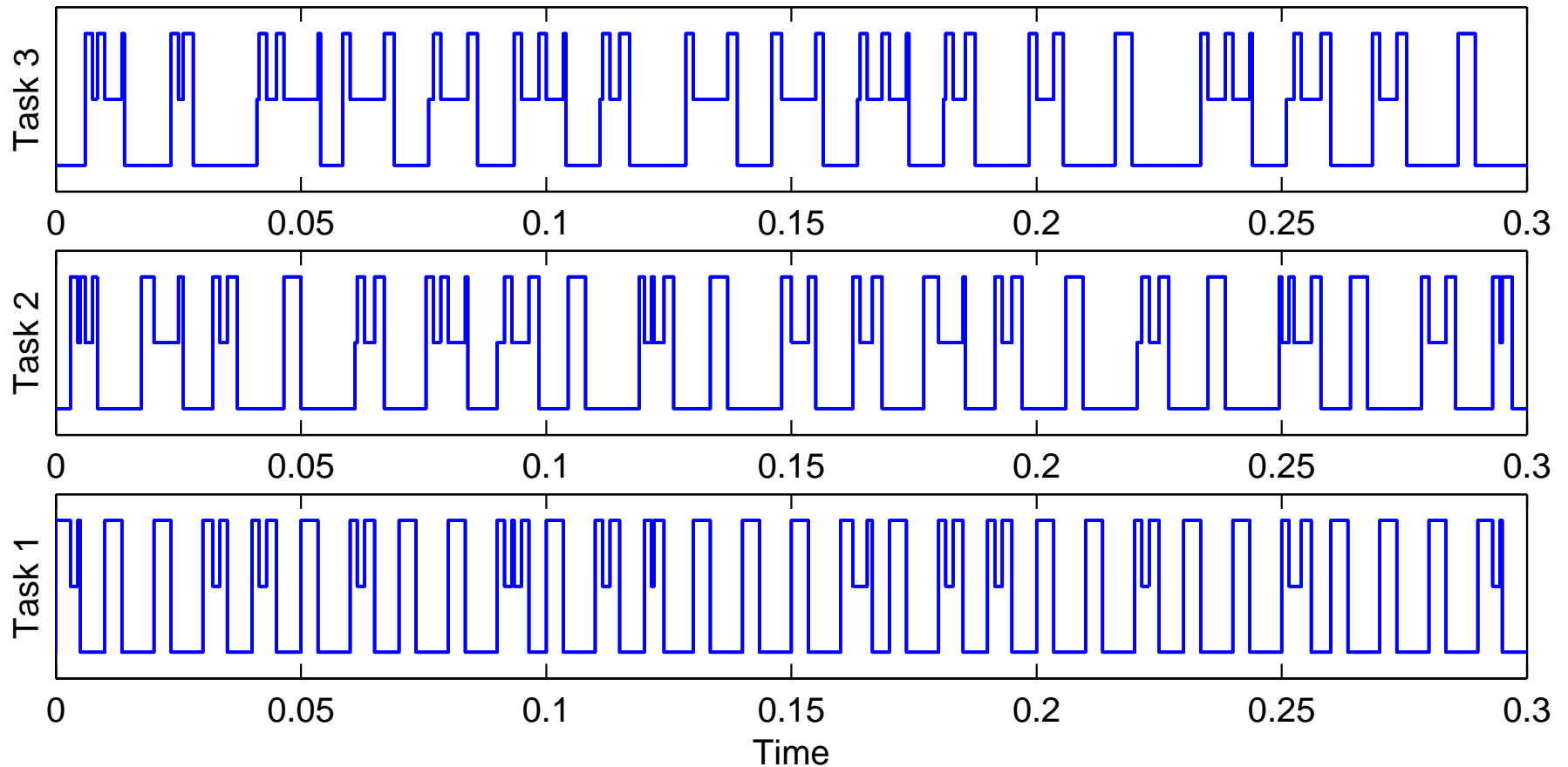
New worst-case input-output latencies: 1.5, 3.0, 4.5 ms.

Simulation under subtask scheduling



Simulation under subtask scheduling

Schedule (high=running, medium=ready, low=sleeping)



- More context switches