# Mechatronic Control Systems:

# An Implementation Perspective

Karl–Erik Årzén & Anton Cervin
Dept of Automatic Control
Lund University



## Outline

- Mechatronic Control Problems
- Segway Example
- Discrete Control Systems
- Continuous Control Systems
  - Design Methodologies
  - Aliasing
  - Arithmetics
  - Realization
  - Computation Delay
  - Example: PID Controller
  - Implementation Paradigms



## More Information

The material presented in these two lectures is primarily based on the following courses:

- Reglerteknik AK (Basic Course in Automatic Control)
- Computer–Controlled Systems (Digital Reglering)
- Real–Time Systems
- Nonlinear Control and Servo Systems



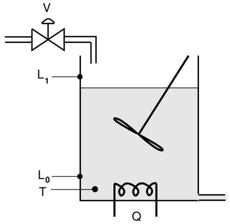
## More Information cont.

Strong Relations to other courses:

- Automation
- Embedded Systems
- Real–Time Programming
- Electronics (Elektronik) – analog computations, OP–amplifiers
- Design of Digital Circuits (Digitalteknik) – finite state machines, boolean functions
- Datorteknik – arithmetics
- Numerical Analysis
- Computational Mechatronics
- and many more ...

## A Typical Control Problem

Raw material buffer tank with heating (non–mechatronic, but still)




Goals:

- Temperature control: PI–controller
- Level control: open $V$ when level below $L_0$, keep the valve open until level above $L_1$
- Sensor fault detection: Generate alarm whenever $L_1$ is true and $L_0$ is false



## Characteristics

- Concurrent activities.
- Timing requirements – more or less hard.
- Discrete (binary) and analog signals.
- Continuous (time–driven) control and Discrete (event–driven) control
- Discrete control consists of both sequence control logic (state–machine oriented) and combinatorial logic (inter–locks) (Alarm = $L_1$ AND NOT $L_0$)

The above characteristics hold for almost all control applica–tions, whether mechatronic or not.



## Mechatronic System Characteristics

Mechatronic systems are often embedded systems.

The "computing device" is an embedded part of a mechatronic device/equipment.

Constraints on cost and size generate constraints on execution time, "execution space" (memory, word–length, chip–size), power usage, bandwidth, fault–tolerance, ... ...

"Resource–Constrained Control"





## Mini-Segway

**An Example of a Mechatronic Embedded Control System**

# Inspiration





# Sensors & Actuators

- Wheel encoders
- Accelerometer & Gyro
  - used together to measure the pendulum angle
  - accelerometer:
    - accurate static measurement of angle
    - does not give any useful information during acceleration of the Segway
  - gyro:
    - accurate dynamic measurement of angle velocity
    - integration to get angle
    - integrated value drifts over time due to measurement errors
  - complimentary filter technique
    - block high-frequency parts of accelerometer signal
    - block low-frequency parts of the gyro signal
- Two DC motors





# Processor

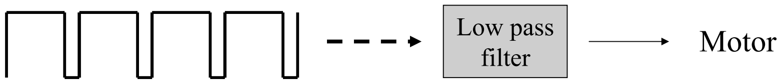
- ATMEL AVR Mega16
  - 16 kByte program memory
  - 2 kByte RAM
  - No hardware support for floating point
    - software emulation
  - AD converters
  - Digital outputs
  - RS 232 communication
  - No RTOS, no threads/tasks – only interrupts and timers
    - definutely no Java
  - Programmed in C with gcc compiler



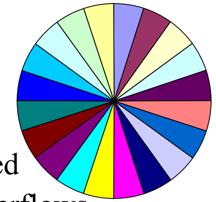


# IO interface

- Motors
  - Digital output
  - Pulse width modulation (PWM)



# IO Interface



- Wheel encoders
  - optical sensor
  - bit counter that is incremented or decremented
  - interrupt generated when an 8-bit counter overflows
    - wheel position and velocity calculated
- Gyro
  - AD converter
  - 10 bits
- Accelerometer
  - sensor generates PWM signals
  - 16 bit counter to measure time intervals





# Software Structure

- Event driven
- Interrupts:
  - Wheel encoders
  - AD converter
  - Timer interrupt for controller
  - Interrupts for sending and receiving over RS232 (UART)





# Control Design

- State feedback controller
  - states: pendulum angle, angle velocity, wheel position, wheel velocity
  - velocities approximated using differences
- Design process:
  - model in continuous time (physics + experiment)
  - sampled to a discrete-time model ($h = 16.4$ ms)
  - design using LQ and poleplacement



# Implementation Technologies

- Analog Computations
  - Pure mechanics
  - Pneumatics
  - Discrete Analog Electronics
  - Analog ASICs
- Digital Computations
  - Electro-mechanical relay systems
  - Discrete Digital Electronics
  - Digital ASICs
  - PLA (Programmable Logic Arrays)
  - FPGA (Field-Programmable Gate Arrays)
  - Digital signal processors (DSP)
  - General Computers (micro-controllers, micro-processors, ...)

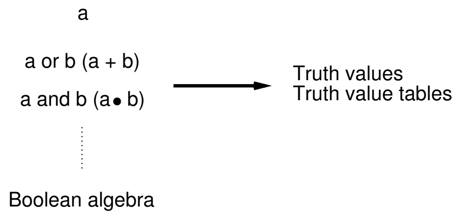
# Outline

- Mechatronic Control Problem
- Segway Example
- **Discrete Control Systems**
- Continuous Control Systems
  - Analog Implementation
  - Digital Implementation
    - ∗ Design Methodologies
    - ∗ Aliasing
    - ∗ Arithmetics
    - ∗ Realization
    - ∗ Computation Delay
    - ∗ Example: PID Controller
    - ∗ Implementation Paradigms

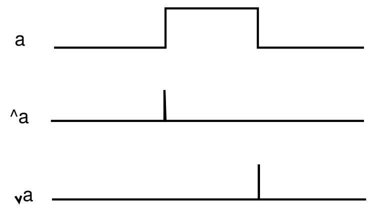


# Basic Elements

- Boolean (binary) signals – $0, 1$, $false, true$, $a, \bar{a}$
- expressions




- events

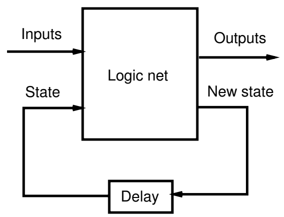





# Logic Nets

- Combinatorial nets – boolean functions
  - outputs = f(inputs)
  - interlocks, "förreglingar"
- Sequence nets
  - newstate = f(state,inputs)
  - outputs = g(state,inputs)
  - state machines (automata)

Asynchronous nets or synchronous (clocked) nets






Course on Design of Digital Circuits

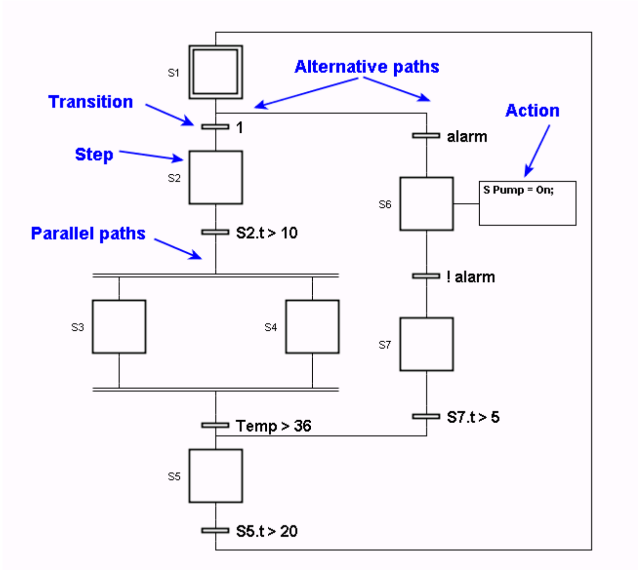
# Grafcet

Extended state machine formalism for implementation of sequence control

Industrial name: Sequential Function Charts (SFC)

Defined in France in 1977 as a formal specification and realization method for logical controllers

Standardized in IEC 848. Part of IEC 1131–3




Editors and compilers.



# Statecharts

D. Harel, 1987

Statecharts =

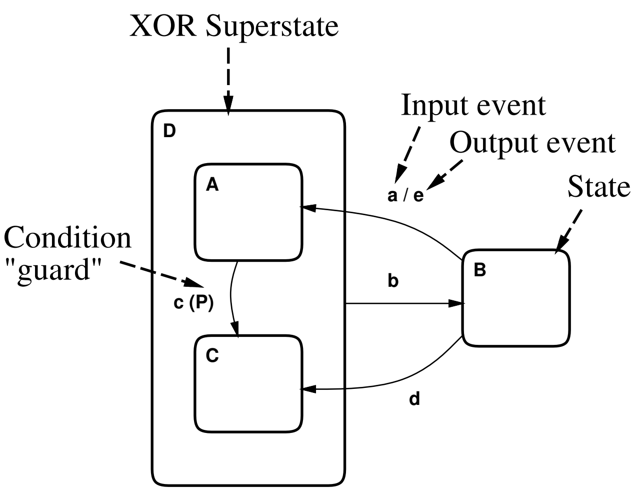
- state–transition graphs
- hierarchy
- concurrency
- history

The state–machine formalism used within UML (Unified Model–ing Language).

Several different tools available.



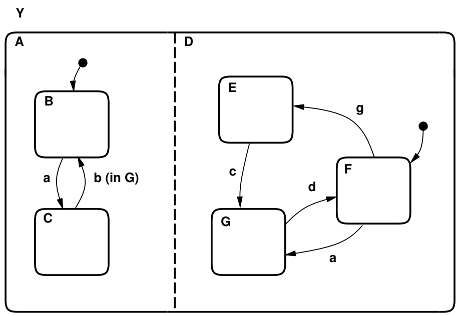
# Statechart Syntax






# Statecharts Concurrency

AND Superstates:




Y is the *orthogonal product* of A and D

When in state (B,F) and event $\mathbf{a}$ occurs, the system transfers *simultaneously* to (C,G).

# Implementation alternatives

Several possibilities:

- discrete digital electronics
- digital ASICs
- PLA (AND–gates and OR–gates, minimal conjunctive form)
- FPGA (more general gates)
- Processors
  - single–bit CPUs (simple PLCs (Programmable Logic Controllers))
  - micro controllers/processors

What to choose depends on cost, size, requirements on flexibility, ...

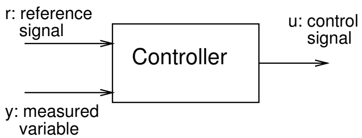


# Outline

- Mechatronic Control Problem
- Segway Example
- Discrete Control Systems
- **Continuous Control Systems**
  - Analog Implementation
  - Digital Implementation
    - * Design Methodologies
    - * Aliasing
    - * Arithmetics
    - * Realization
    - * Computation Delay
    - * Example: PID Controller
    - * Implementation Paradigms



# Continuus Control Systems




General Controller Form:

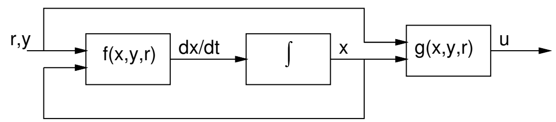
$$\begin{aligned}\frac{dx}{dt} &= f(x, y, r)\\ u &= g(x, y, r)\end{aligned}$$

Linear case:

$$\begin{aligned}f(x, y, r) &= Fx + Gy + Hr\\ g(x, y, r) &= Cx + Dy + Er\end{aligned}$$



# Implementation




Integration + function generation

Linear case:

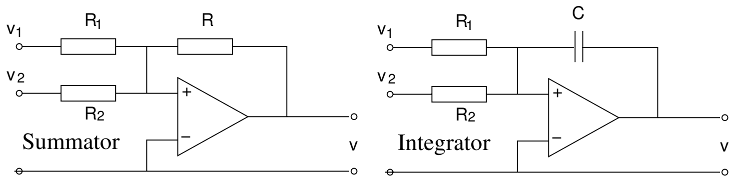
- summation + accumulation
- multiplication with coefficient
- scalar product

Non–linear elements:

- selector logic (min/max, comparison)
- general non–linear elements for reference signal generation, gain–schedules, adaptation (can often be implemented as lookup–tables)
- ...

## Implementation with Analog Electronics

Using operational amplifiers and passive elements (resistors, capacitors) it is straightforward to implement summation and integration




Summator:

$$v = -(\frac{R}{R_1}v_1 + \frac{R}{R_2}v_2)$$

Integrator:

$$v = -(\frac{1}{R_1C}\int_0^t v_1(\tau)d\tau + \frac{1}{R_2C}\int_0^t v_2(\tau)d\tau)$$



## Scaling

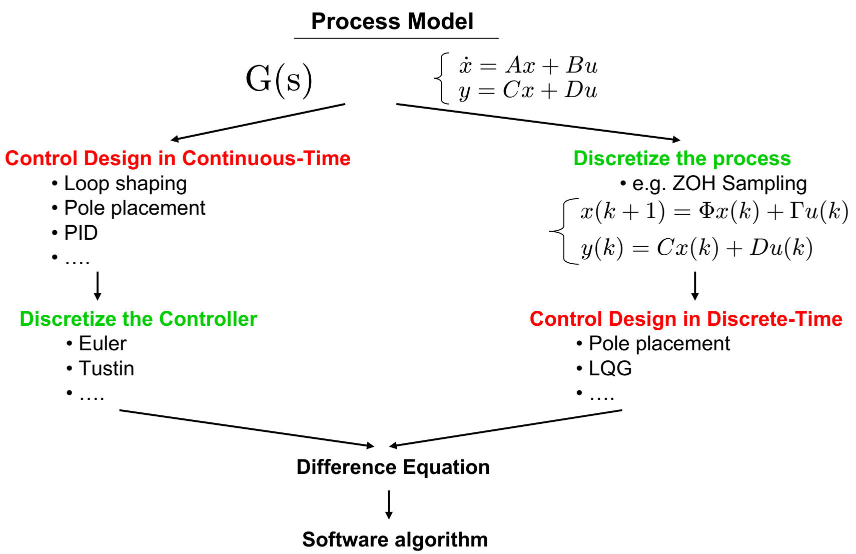
Physical variables (positions, forces, temperatures, ...) are represented as electrical signals (voltages) that have some specified limit (e.g. $\pm 10V$)

It is important to scale the variables appropriately to avoid overloads and saturations.

Within the permissible operating range it is desirable to have each variable assume as large absolute values as possible to minimize errors due to offset voltages, noise etc.



## Controller Synthesis

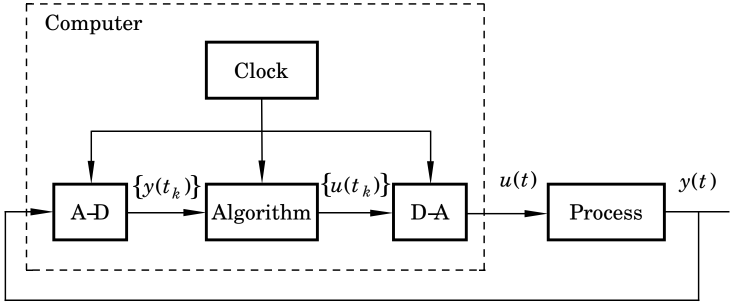



## Discrete-time Implementation

Digital controllers can be designed in two different ways:

- Discrete time design
  - sampled (digital) control theory
  - shift operators (z-transforms)
  - $u(k) = k_1 y(k) + k_2 u(k-1)$
  - $h$ a design parameter
- Continuous time design + discretization
  - Laplace transform
  - $U(s) = G_c(s)E(s)$
  - approximate the continuous design
  - fast, fixed sampling

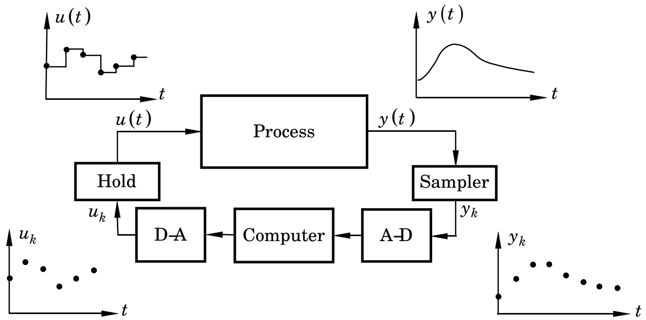
# Sampled Control Theory




The basic idea: Look at the sampling instances only!



# Sampled Control Theory

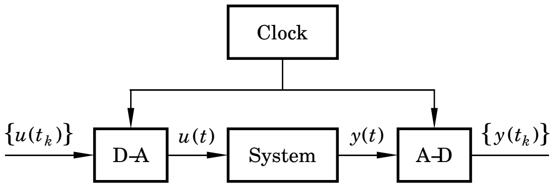



- System theory analogous to continuous time linear sys–tems
- Better performance can be achieved
- Problems with inter–sample behavior



# Sampling of Systems

Look at the system from the point of view of the computer




Zero–order–hold sampling of a system

- Let the inputs be piecewise constant
- Look at the sampling points only
- Use linearity and calculate step responses when solving the system equation



# Sampling a continuous–time system

System description

$$\begin{aligned} \frac{dx}{dt} &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned}$$

Solve the system equation

$$\begin{aligned} x(t) &= e^{A(t-t_k)}x(t_k) + \int_{t_k}^{t} e^{A(t-s')}Bu(s')\,ds' \\ &= e^{A(t-t_k)}x(t_k) + \int_{t_k}^{t} e^{A(t-s')}\,ds'\,Bu(t_k) \quad (u \text{ const.}) \\ &= e^{A(t-t_k)}x(t_k) + \int_{0}^{t-t_k} e^{As}\,ds\,Bu(t_k) \quad \text{(variable change)} \\ &= \Phi(t,t_k)x(t_k) + \Gamma(t,t_k)u(t_k) \end{aligned}$$

## The General Case

$$
\begin{aligned}
x(t_{k+1}) &= \Phi(t_{k+1}, t_k)x(t_k) + \Gamma(t_{k+1}, t_k)u(t_k) \\
y(t_k) &= Cx(t_k) + Du(t_k)
\end{aligned}
$$

where

$$
\begin{aligned}
\Phi(t_{k+1}, t_k) &= e^{A(t_{k+1}-t_k)} \\
\Gamma(t_{k+1}, t_k) &= \int_0^{t_{k+1}-t_k} e^{As} ds \; B
\end{aligned}
$$



## Periodic sampling

Assume periodic sampling, i.e. $t_k = k \cdot h$, then

$$
\begin{aligned}
x(kh + h) &= \Phi x(kh) + \Gamma u(kh) \\
y(kh) &= Cx(kh) + Du(kh)
\end{aligned}
$$

where

$$
\begin{aligned}
\Phi &= e^{Ah} \\
\Gamma &= \int_0^h e^{As} \, ds \, B
\end{aligned}
$$

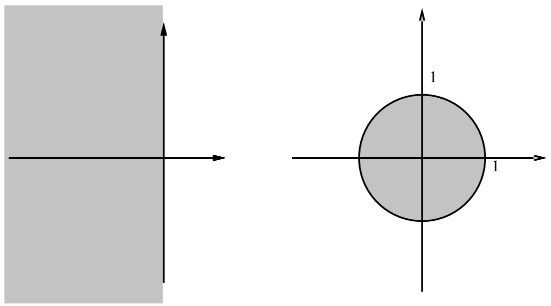
NOTE: Time–invariant linear system!



## Stability region

In continuous time the stability region is the complex left half plane, i.e„ the system is stable if all the poles are in the left half plane.

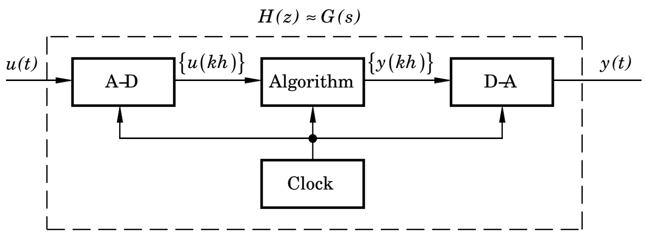
In discrete time the stability region is the unit circle.






## Discretization of Continuous Time Design

Basic ideas: Reuse the design




$G(s)$ is designed based on analog techniques

Want to get:

- A/D + Algorithm + D/A $\approx G(s)$

Methods:

- Approximate $s$, i.e., $G(s) \Rightarrow H(z)$
- Other methods

## Approximation Methods

Forward Difference (Euler forward method)

$$\frac{dx(t)}{dt} \approx \frac{x(t+h) - x(t)}{h}$$

$$s = \frac{z-1}{h}$$

Backward Difference (Euler backward method)

$$\frac{dx(t)}{dt} \approx \frac{x(t) - x(t-h)}{h}$$

$$s = \frac{z-1}{zh}$$



## Approximation Methods, cont

Tustin (trapeziodal, bilinear):

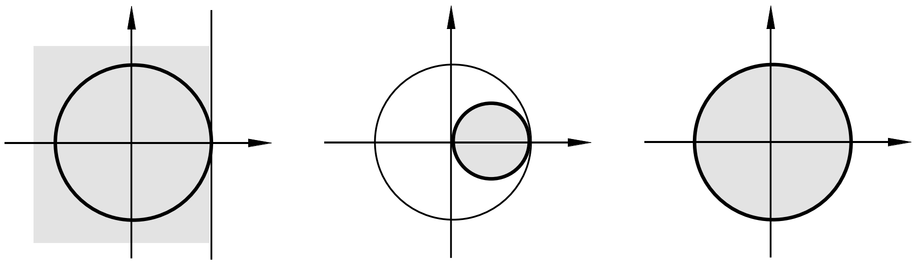
$$\frac{\dot{x}(t+h) + \dot{x}(t)}{2} \approx \frac{x(t+h) - x(t)}{h}$$

$$s = \frac{2}{h} \frac{z-1}{z+1}$$



## Stability of Approximations

How is the continuous–time stability region (left half plane) mapped?



Forward differences Backward differences Tustin



## Basic Operations

Integration ⇒ Summation + accumulation

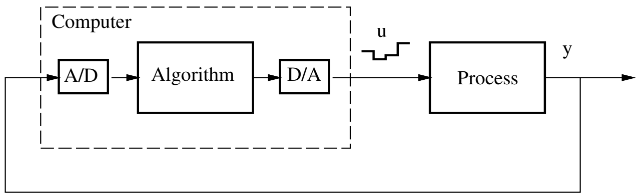
Derivation ⇒ Difference approximation (in the simplest case)

Selector logic and nonlinearities straightforward

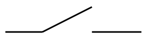
Scalar products main operation in controllers and filters

DSPs optimized for this.

# Issues: Sampling and Aliasing




AD–converter acts as sampler

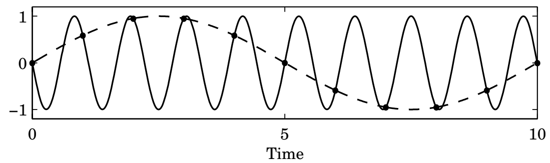



DA–converter acts as a hold device

Normally, zero–order–hold is used $\Rightarrow$ piecewise constant control signals



# Aliasing




$\omega_N = \omega_s/2 =$ Nyquist frequency, ($\omega_s =$ sampling freq.)

Frequencies above the Nyquist frequency are folded and appear as low–frequency signals.

The fundamental alias frequency for a frequency $f_1 > f_N$ is given by

$$f = |(f_1 + f_N) \mod (f_s) - f_N|$$

Above: $f_1 = 0.9$, $f_s = 1$, $f_N = 0.5$, $f = 0.1$



# Prefilters

Anti–aliasing filter

Analog low–pass filter that eliminates all frequencies above the Nyquist frequency

- Analog filter
  - 2–6th order Bessel or Butterworth
  - Difficulties with changing $h$ (sampling interval)
- Digital Filter
  - Fixed, fast sampling with fixed analog filter
  - Control algorithm at a slower rate together with digital LP–filter
  - Easy to change sampling interval

The filter may have to be included in the design.



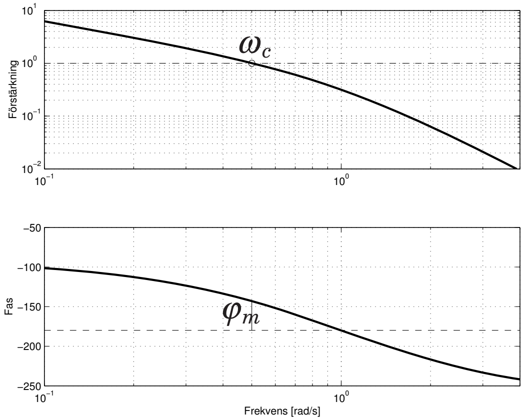
# Choice of sampling interval

Nyquist's sampling theorem:

> "We must sample at least twice as fast as the highest frequency we are interested in"

- What frequencies are we interested in?

Typical loop transfer function $L(i\omega) = P(i\omega)C(i\omega)$:




- $\omega_c$ = cross-over frequency, $\varphi_m$ = phase margin
- We should have $\omega_s \gg 2\omega_c$



## Sampling interval rule of thumb

A sample-and-hold (S&H) circuit can be approximated by a delay of $h/2$.

$$G_{S\&H}(s) \approx e^{-sh/2}$$

This will decrease the phase margin by

$$\arg G_{S\&H}(i\omega_c) = \arg e^{-i\omega_c h/2} = -\omega_c h/2$$

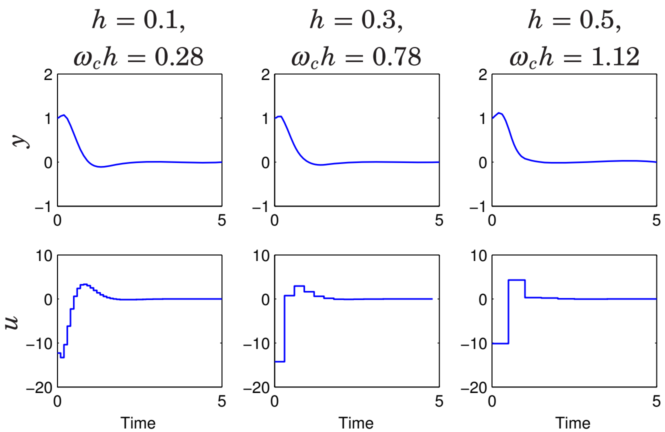
Assume we can accept a phase loss between $5^\circ$ and $15^\circ$. Then

$$\boxed{0.15 < \omega_c h < 0.5}$$

This corresponds to a Nyquist frequency about 6 to 20 times larger than the crossover frequency



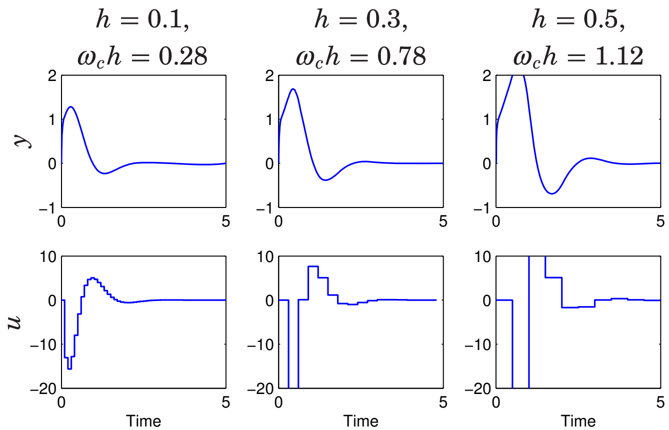
## Example: control of inverted pendulum




- Large $\omega_c h$ may seem OK, but beware!
  - Digital design assuming perfect model
  - Controller perfectly synchronized with initial disturbance



## Pendulum with non-synchronized disturbance



## Accounting for the anti–aliasing filter

Assume we also have a second–order Butterworth anti–aliasing filter with a gain of 0.1 at the Nyquist frequency. The filter gives an additional phase margin loss of $\approx 1.4\omega_c h$.

Again assume we can accept a phase loss of $5^\circ$ to $15^\circ$. Then

$$\boxed{0.05 < \omega_c h < 0.14}$$

This corresponds to a Nyquist frequency about 23 to 70 times larger than the crossover frequency



## Outline

- Mechatronic Control Problem
- Segway Example
- Discrete Control Systems
- Continuous Control Systems
  - Analog Implementation
  - Digital Implementation
    - Design Methodologies
    - Aliasing
    - **Arithmetics**
    - Realization
    - Computation Delay
    - Example: PID Controller
    - Implementation Paradigms



## Issues: Computer Arithmetics

Control analysis and design assumes floating point arithmetics (i.e. high range and resolution)

Hardware–supported on modern high–end processors (e.g., floating point ALUs (Arithmetic–Logic Units))

Representation:

$$\pm f \times 2^{\pm e}$$

- $f$: mantissa, significant, fraction
- $2$: radix or base
- $e$: exponent



## IEEE 754 Standard

Used by almost all floating–point processors (except certain DSPs)

Single precision (Java/C float):

- 32–bit word divided into 1 sign bit, 8–bit exponent, and 23–bit mantissa
- Range: $2^{-126} - 2^{128}$

Double precision format (Java/C double):

- 64–bit word divided into 1 sign bit, 11–bit exponent, and 52–bit mantissa.
- Range: $2^{-1022} - 2^{1024}$

Supports infinity and NaN

# Floating-point emulation

Emulate floating-point arithmetics in software

Approaches:

- compiler supported
- manually
  - e.g., floating point variables represented as C structs
  - floating point operations in the form of a library

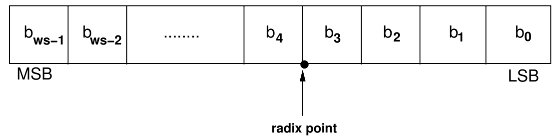
Problems:

- Code size becomes too large
- Slows down execution speed
- Non-trivial



# Fixed Point Arithmetics

Use the binary word directly for representing numbers




- MSB – Most significant bit
- LSB – Least significant bit
- ws – word-size

Unsigned versus signed



# Fixed Point Arithmetics

Integer arithmetics:

- radix point to right of LSB
- 16 bits signed integer gives range $-32768 \leq \hat{x} \leq 32767$ $((-2^{15}) - (2^{15} - 1))$

Fractional arithmetics:

- radix point to right of MSB (signed)
- $0.10011001$

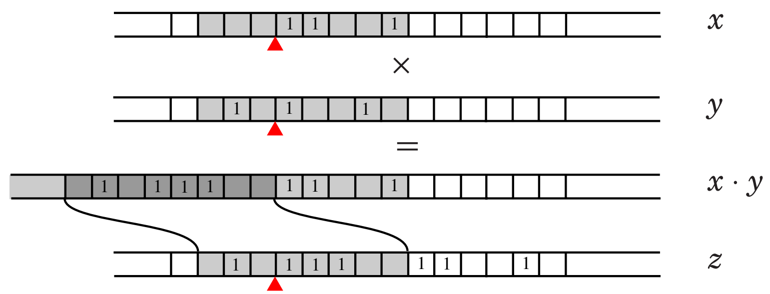
Generalized fixed point arithmetics:

- application-defined radix point
- $1101.0110$
- Scaling: $x = \hat{x}/2^4$ – shifting the radix point



# Fixed Point Calculations

Fixed point multiplication involves quantization




Fixed-point addition is error-free

Quantization (truncation or rounding)

- modeled as "noise"

Overflow (wrap-around or saturation)

## Example: Scalar products

Many controllers and filters involve calculations of scalar products, e.g.,
$u = -Lx = -[l_1 l_2 l_3][x_1 x_2 x_3]^T = -l_1 x_1 - l_2 x_2 - l_3 x_3$

Consider the vectors

$$\begin{aligned} a &= (100 \quad 1 \quad 100) \\ b &= (100 \quad 1 \quad -100) \end{aligned}$$

The true scalar product is $1$

When computed in fixed point representation using a precision corresponding to three decimal places, the result will be $0$ ($100 \times 100 + 1 \times 1$ is rounded to $10000$)

The result depends on the order or the operations.

To avoid this it is common to use higher resolution in the accumulator and round to a smaller resolution afterwards.



## Fixed–Point Arithmetics Problems

- Quantization
  Fixed–point values are rounded or truncated.
  - Coefficient Quantization: Poles and zeros end up some–where else
  - Signal (state) Quantization:
    - Noise is added in each operation
    - Quantization may cause signal bias
    - Quantization may cause limit cycles. Either in the output only (LSB) or in the entire system through feedback.
- Overflow
  Adding/Multiplying two sufficiently large numbers can produce a result that does not fit into the representation.
  - Scaling important both of variables and of coefficients.
  - Overflow characteristics. Saturation or wrap–around? Hardware supported overflow detection or not.



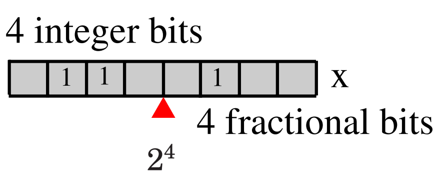
## Example: Coefficient Quantization

An example controller

$$C(z) = \frac{z^4 - 2.13z^3 + 2.351z^2 - 1.493z + 0.5776}{z^4 - 3.2z^3 + 3.997z^2 - 2.301z + 0.5184}$$

8–bit fixed point coefficients with $x = \hat{x}/2^4$, so

$$x \in \left[-8.0 \ldots 7.9375\right]$$






## Example: Coefficient Quantization

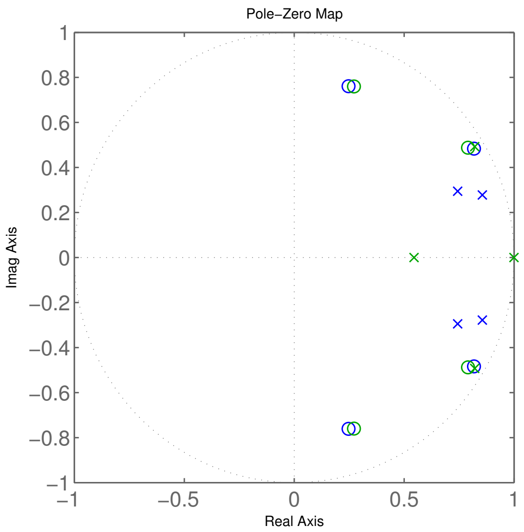
- Original:

$$C(z) = \frac{z^4 - 2.13z^3 + 2.351z^2 - 1.493z + 0.576}{z^4 - 3.2z^3 + 3.997z^2 - 2.301z + 0.5184}$$

- Quantized:

$$C(z) = \frac{z^4 - 2.125z^3 + 2.375z^2 - 1.5z + 0.5625}{z^4 - 3.188z^3 + 4z^2 - 2.312z + 0.5}$$

# Example






# Issues: Realization of Digital Controllers

A digital controller

$$u(k) = H(q^{-1})y(k) = \frac{b_0 + b_1 q^{-1} + \cdots + b_m q^{-m}}{1 + a_1 q^{-1} + a_2 q^{-2} + \cdots + a_n q^{-n}} y(k)$$

can be realized in a number of different ways with equivalent input–output behavior (different choice of state variables)

Issues:

- number of storage elements (memory)
- number of non–zero non–one coefficients
- coefficient range
- sensitivity towards coefficient quantization
- sensitivity towards state quantization
  - order of computations matters



# Direct and Companion Forms

$$u(k) = \sum_{i=0}^{m} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i)$$

Not minimal ($n + m$ states)

Companion forms (e.g., observable canonical form or controllable canonical form):

$$x(k+1) = \begin{pmatrix} -a_1 & 1 & 0 & \cdots & 0 \\ -a_2 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -a_{n-1} & 0 & 0 & \cdots & 1 \\ -a_n & 0 & 0 & \cdots & 0 \end{pmatrix} x(k) + \begin{pmatrix} b_1 \\ \vdots \\ b_{m-1} \\ b_m \\ 0 \end{pmatrix} y(k)$$

$$u(k) = \begin{pmatrix} 1 & 0 & \cdots & 0 \end{pmatrix} x(k)$$

Minimal

Coefficients in the characteristic polynomial are the coefficients in the realization. Sensitive to computational errors if the systems are of high order and if the poles or zeros are close to each other.



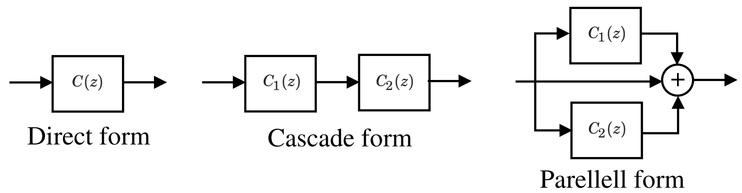
# Example

A linear system can be rewritten in many ways:

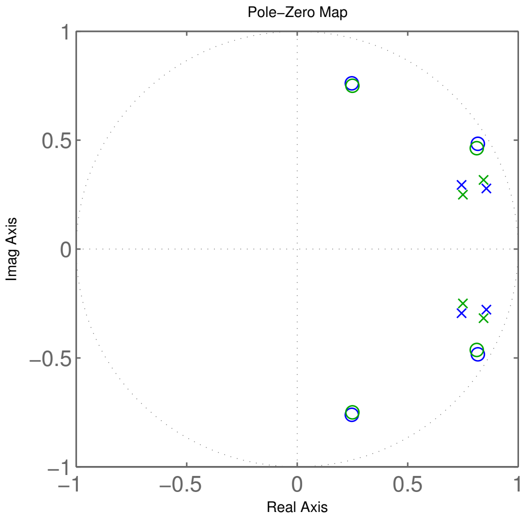
$$C(z) = \frac{z^4 - 2.13z^3 + 2.351z^2 - 1.493z + 0.5776}{z^4 - 3.2z^3 + 3.997z^2 - 2.301z + 0.5184}$$

$$= \left(\frac{z^2 - 1.635z + 0.9025}{z^2 - 1.712z + 0.81}\right)\left(\frac{z^2 - 0.4944z + 0.64}{z^2 - 1.488z + 0.64}\right)$$

$$= 1 + \frac{-5.396z + 6.302}{z^2 - 1.712z + 0.81} + \frac{6.466z - 4.907}{z^2 - 1.488z + 0.64}$$



# Cascade form

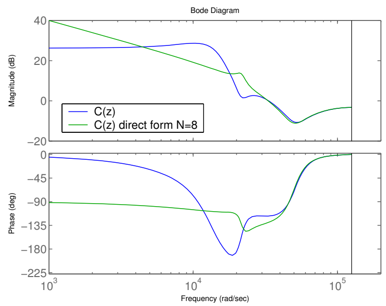




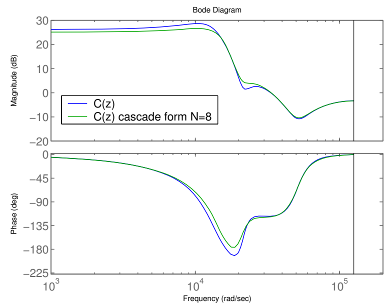

# Well-conditioned realizations

Parallel (diagonal/Jordan) and cascade (series) forms have normally the best numerical properties.

If poles (zeroes) are far apart, direct form is usable.




Direct form



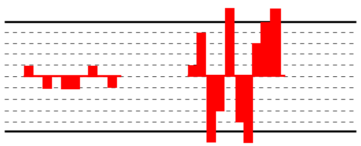

Cascade form

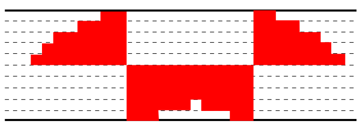


# State saturation

For fixed point arithmetics, there is a balance:

- Too high gain in some part of system will cause state to overflow.
- Too low gain in some part of system will cause a lot of quantization errors.





Your digital system should have gain $\gamma \approx 1$.

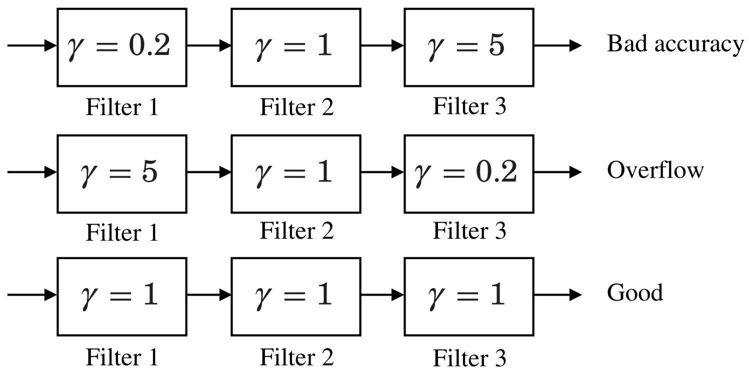
What is $\gamma$?

*The gain of the system for the kind of input signal we expect*



# State saturation

Spread the gain:



# State saturation

How to pair and order poles and zeros?

Jackson's rules (1970):

- Pair the pole closest to the unit circle with its closest zero. Repeat until all poles and zeros are taken.
- Order the filters in increasing or decreasing order based on the poles closeness to the unit circle.

This will push down high internal resonance peaks.



# Summing up

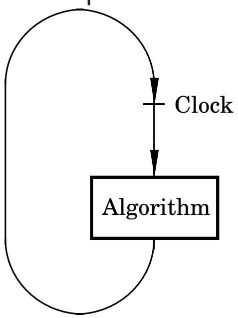
Problems and solutions:

- Coefficient quantization:
  - Avoid direct forms and companion forms
  - Always split systems into first– and second–order systems (cascade, parallel form)
- State quantization:
  - Can be modeled as noise sources after multiplicators
  - Use double–size accumulator
- State saturation:
  - Have equal gains ($\gamma \approx 1$) for all systems
  - Use Jackson's rules for pole–zero sorting



# Issues: Computational Delay

Most controller are based on periodic sampling.




Basic problem: $u(k) = f(y(k), .)$

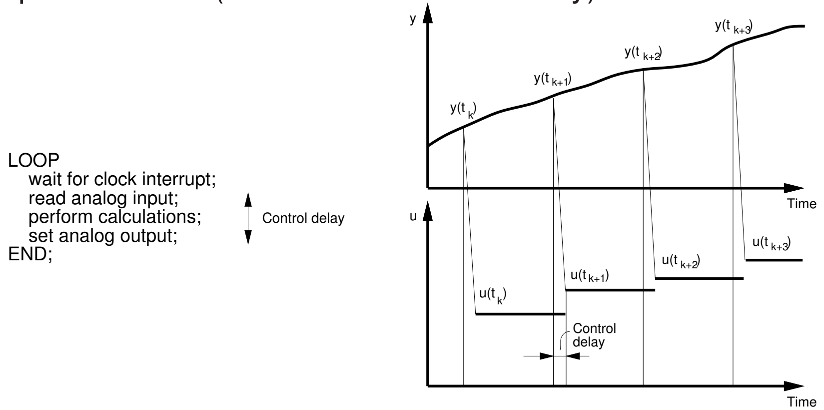
Computation time not accounted for.



# Computational Delay

Problem: $u(k)$ cannot be generated instantaneously at time $k$ when $y(k)$ is sampled

Delay (computational delay or input–ouput latency) due to computation time (and communication delay)

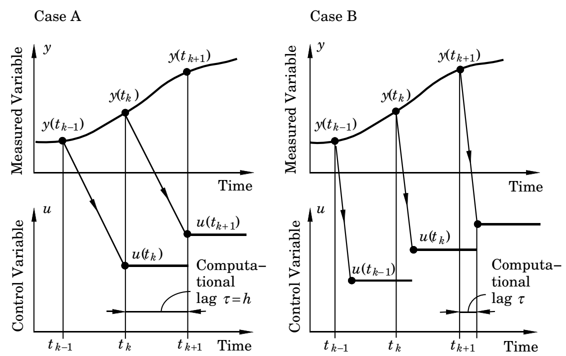


# Four Approaches

1. Design the controller to be robust against variations in the computational delay
   - complicated
2. Ignore the computational delay
   - often justified, since it is small compared to $h$
   - write the code so that the delay is minimized, i.e., minimize the operations performed between AD and DA
   - divide the code into two parts: CalculateOutput and Updat–eStates
3. Compensate for the computational delay
   - include the computational delay in model and the design
   - sampling of systems with time delays
   - write the code so that the delay is constant



4. Include an delay of one sample in the controller
   - do not send out the control signal until the start of next sample
   - computational delay $= h$
   - easier way to compensate (multiple of the sampling interval)






# Minimize Control Delays

General Controller representation:

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

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;
```



# Sampling interval and delay rule of thumb

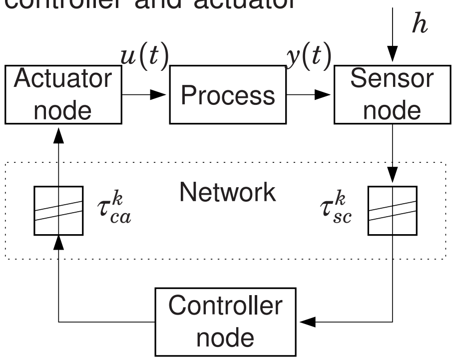
Assume that the delay is $\tau$. This gives an additional phase margin loss of $-\omega_c\tau$. Extending our first rule of thumb we get

$$\boxed{0.15 < \omega_c(h + 2\tau) < 0.5}$$

- If the delay is too large, we **must** decrease the speed of the controlled system (i.e. the cross–over frequency $\omega_c$)
  - The delay imposes a fundamental performance limita–tion

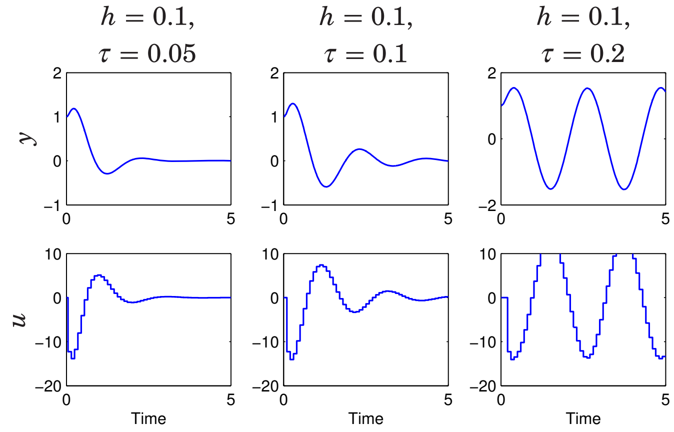
# Other sources of time delays

- Deadtime in the process
  - deadtime after the actuator
  - deadtime before the sensor
- Communication delays
  - between sensor and controller
  - between controller and actuator






# Pendulum controller with time delay




- No delay compensation



# Delay margin

Suppose the loop transfer function without delay has

- cross-over frequency $\omega_c$
- phase margin $\varphi_m$

Phase margin loss due to delay:

$$\arg e^{-i\omega_c \tau} = -\omega_c \tau$$

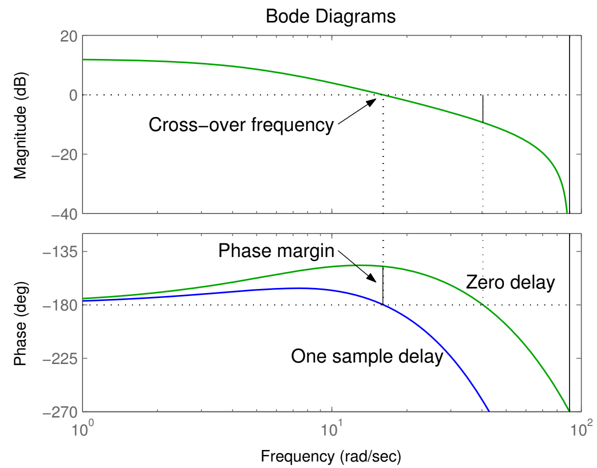
Closed-loop system stable if

$$\omega_c \tau < \varphi_m \quad \Leftrightarrow \quad \tau < \frac{\varphi_m}{\omega_c}$$

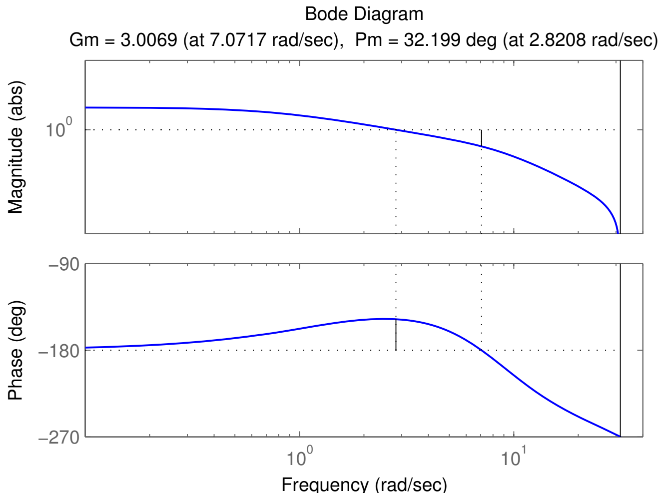
$\tau_m = \dfrac{\varphi_m}{\omega_c}$ is called the **delay margin**



# Why is delay bad?



# Example: delay margin for pendulum controller

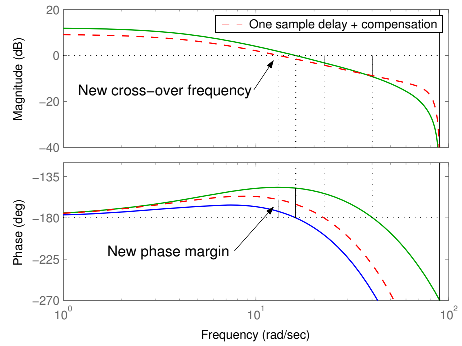



$\varphi_m = 32^\circ$, $\omega_c = 2.8$ rad/s $\Rightarrow$ $\tau_m = \frac{32\pi}{180 \cdot 2.8} = 0.2$



# Delay Compensation

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



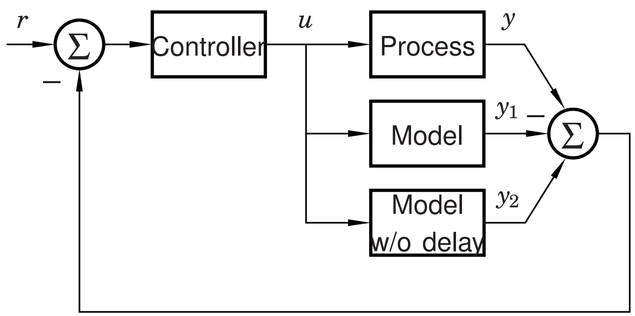

Continuous–time: Smith predictor or lead compensation

Discrete–time: Include the delay in the process model



# Delay compensation using Smith predictor

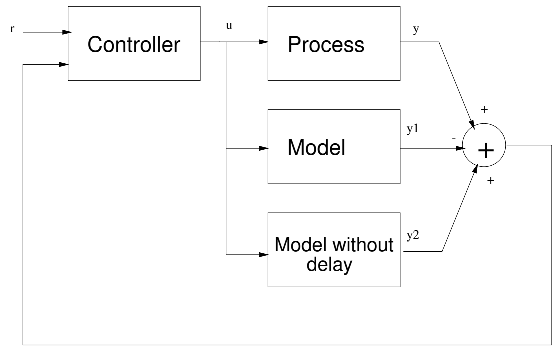
Idea: control against simulated model without delay:




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



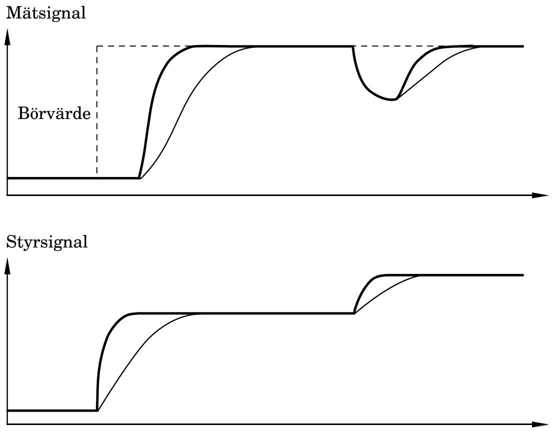
# The Smith Predictor




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)

# PI versus Smith




However, a delay compensating controller can never undo the delay



# 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)



# Why is jitter bad?

The effects of sampling jitter and input–output latency jitter are quite hard to analyze.

If one can measure the actual jitter every sample, it is possible to design controllers that, at least partly, can compensate for the jitter.

For sampling jitter, this corresponds to re–sample the controller in every sample.



# Reasons for delays and jitter

- Computation time (possibly varying)
- Preemption (blocking) by other activities that are more important (have higher priority)
- Blocking due to access of shared resources
- Temporally non–deterministic implementation platform (hardware, OS)
- Communication delays

# An Example: PID Control

Textbook Algorithm:

$$u(t) = K(e(t) + \frac{1}{T_I}\int^t e(\tau)d\tau + T_D\frac{de(t)}{dt})$$

$$U(s) = K(E(s) + \frac{1}{sT_I}E(s) + T_D s E(s))$$

$$= P + I + D$$



# A better algorithm

$$U(s) = K(\beta y_r - y + \frac{1}{sT_I}E(s) - \frac{T_D s}{1 + sT_D/N}Y(s))$$

Modifications:

- Setpoint weighting ($\beta$) in the proportional term improves set–point response
- Limitation of the derivative gain (low–pass filter) to avoid derivation of measurement noise
- Derivative action only on $y$ to avoid bumps for step changes in the reference signal



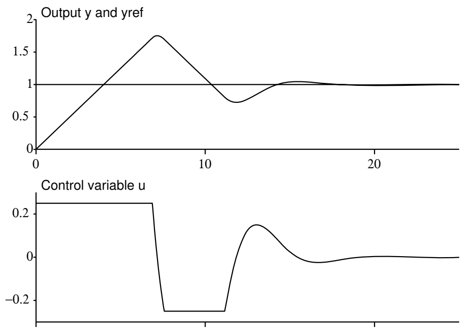
# Control Signal Limitations

All actuators saturate.

Problems for controllers with integration.

When the control signal saturates the integral part will continue to grow – integrator (reset) windup.

When the control signal saturates the integral part will integrate up to a very large value. This may cause large overshoots.

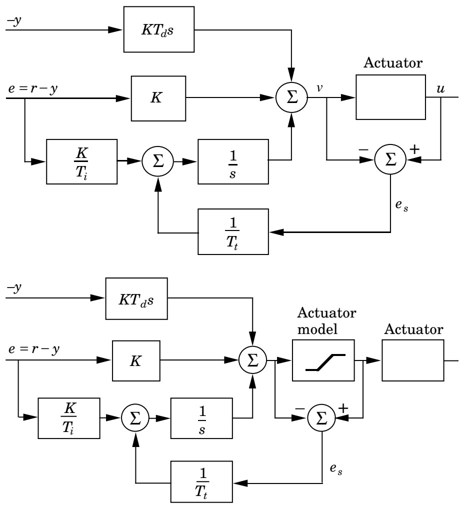





# Tracking

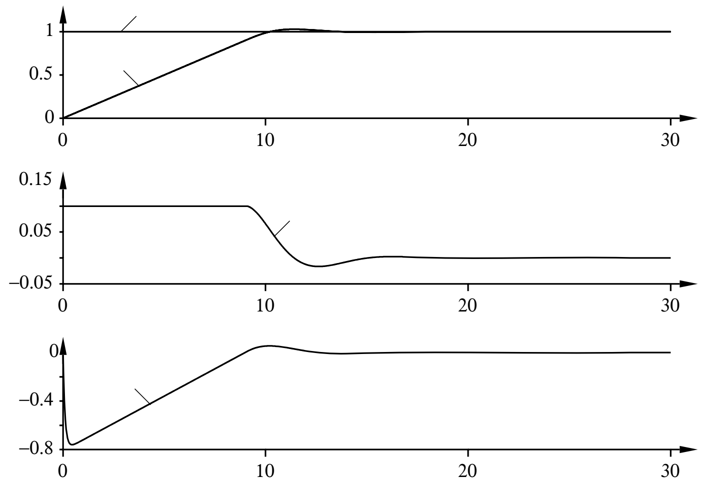
- when the control signal saturates, the integral is recom– puted so that its new value gives a control signal at the saturation limit
- to avoid resetting the integral due to, e.g., measurement noise, the re–computation is done dynamically, i.e., through a LP–filter with a time constant $T_r$.

# Tracking






# Tracking






# Discretization

**P-part:**

$$u_P(k) = K(\beta y_{sp}(k) - y(k))$$



# Discretization

**I-part:**

$$I(t) = \frac{K}{T_I}\int_0^t e(\tau)d\tau$$

$$\frac{dI}{dt} = \frac{K}{T_I}e$$

- Forward difference

$$\frac{I(t_{k+1}) - I(t_k)}{h} = \frac{K}{T_I}e(t_k)$$

```
I(k+1) := I(k) + (K*h/Ti)*e(k)
```

The I-part can be precalculated in UpdateStates

- Backward difference

The I-part cannot be precalculated, i(k) = f(e(k))

- Others

# Discretization

**D-part** (assume $\gamma = 0$):

$$D = K\frac{sT_D}{1 + sT_D/N}(-Y(s))$$

$$\frac{T_D}{N}\frac{dD}{dt} + D = -KT_D\frac{dy}{dt}$$

- Forward difference (unstable for small $T_D$)
- Backward difference

$$\frac{T_D}{N}\frac{D(t_k) - D(t_{k-1})}{h} + D(t_k)$$

$$= -KT_D\frac{y(t_k) - y(t_{k-1})}{h}$$

$$D(t_k) = \frac{T_D}{T_D + Nh}D(t_{k-1})$$

$$-\frac{KT_DN}{T_D + Nh}(y(t_k) - y(t_{k-1}))$$



# Discretization

**Tracking:**

```
v := P + I + D;
u := sat(v,umax,umin);
I := I + (K*h/Ti)*e + (h/Tr)*(u - v);
```



# PID code

PID–controller with anti–reset windup

```
y = yIn.get(); // A-D conversion
e = yref - y;
D = ad * D - bd * (y - yold);
v = K*(beta*yref - y) + I + D;
u = sat(v,umax,umin)}
uOut.put(u);   // D-A conversion
I = I + (K*h/Ti)*e + (h/Tr)*(u - v);
yold = y
```

$ad$ and $bd$ are precalculated parameters given by the backward difference approximation of the D–term.

Execution time for CalculateOutput can be minimized even further.



# Issues: Implementation Paradigms

Concurrent (parallel) activities.

A control system normally contains several more or less independent periodic or aperiodic activities/tasks (e.g., controllers)

It is often natural to handle the different tasks independently during design.

**Temperature Loop**

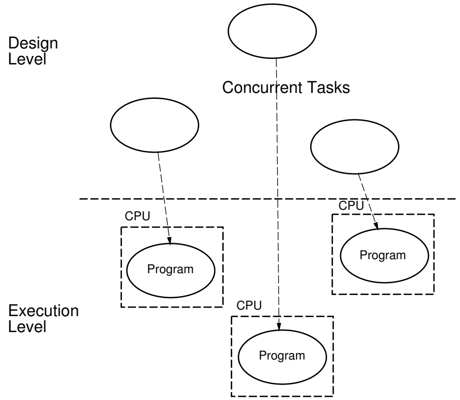
```
while (true) {
  Measure temperature;
  Calculate temperature error;
  Calculate the heater signal with PI–control;
  Output the heater signal;
  Wait for h seconds;
}
```

**Level Loop**

```
while (true) {
  Wait until level below L0;
  Open inlet valve;
  Wait unitil level above L1;
  Close inlet valve;
}
```

# Paradigms

Parallel programming:

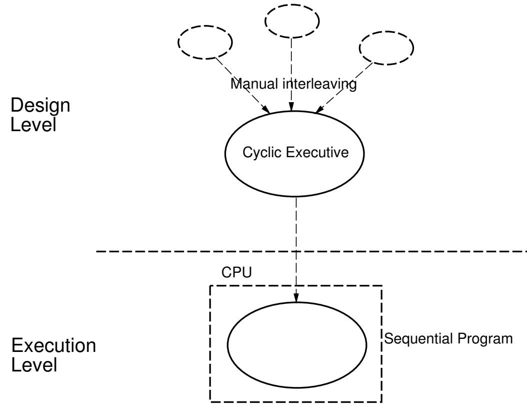



Multiprocessors, VLSI (ASIC), FPGA



# Paradigms

Sequential programming:




Small micro–controllers.



## Interleaved temperature and level loops

```
while (true) {
  while (level above L0) {
    Measure temperature;
    Calculate temperature error;
    Calculate the heater signal with PI–control;
    Output the heater signal;
    Wait for h seconds;
  }
  Open inlet valve;
  while (level below L1) {
    Measure temperature;
    Calculate temperature error;
    Calculate the heater signal with PI–control;
    Output the heater signal;
    Wait for h seconds;
  }
  Close inlet valve;
}
```

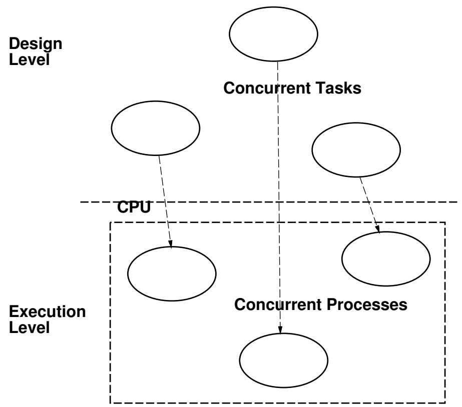
Complex and non user–friendly code if programmed manually.

Automatic code generation from synchronous programming languages



# Paradigms

Concurrent programming:




The CPU is shared between the process (switches)

Real–Time Operating Systems or Real–time Programming Language with run–time system:

- switches between processes/threads
  - Real–Time Kernel
- timing primitives
- process communication

Lectures by Klas Nilsson.



## 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?

Several options:

1. Using a static schedule (cyclic executive)?
   - High temporal determinism but inflexible
   - Does not require any sophisticated RTOS support
2. In interrupt handlers (interrupt service routines) associated with timers (typically in small microcontrollers)
3. As self–scheduling threads in a 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 4:**

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

Does not work. An interrupt may occur between the WaitUntil and CurrentTime.



# Implementing Self–Scheduling Periodic Tasks

**Attempt 4:**

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

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)

Reasonable for alarm clocks, but perhaps not for controllers.



# Implementing Self–Scheduling Periodic Tasks

**Attempt 5:** Reset the base time in case of overruns. Accept a too long sample and try to do it right afterwards.

Assumes the existence of a new WaitTime primitive that calls `CurrentTime` only if an overrun has occurred.

```
t = CurrentTime();
LOOP
  PeriodicActivity;
  t = t + h;
  NewWaitUntil(t);  // Updates t in case of overrun
END;
```

## 2. When is the sampling performed?

Two options:

- At the beginning of the controller task
  - gives rise to sampling jitter and, hence, sampling interval 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



## 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 longer latency than necessary
  - often gives worse performance, also if the constant delay is compensated for
  - delay compensation easy