Project Course 2010

Finite-Wordlength Implementation of Controllers

- Computer arithmetic
  - Floating-point arithmetic
  - Fixed-point arithmetic
- Controller realizations

Based on material from

- K. J. Åström, B. Wittenmark: *Computer-Controlled Systems*, 3rd Ed., 1997.
- B. Lincoln: "Fixed-point controller implementation", Lecture in Embedded Systems, 2004.
- A. Cervin, K.-E. Årzén: "Implementation Aspects 2", Lecture in Real-Time Systems, 2008.

Floating-Point Arithmetic

Hardware-supported on modern high-end processors (FPUs)

Number representation:

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

- f: mantissa, significand, fraction
- 2: base
- e: exponent

The binary point is variable (floating) and depends on the value of the exponent

Dynamic range and resolution

Fixed number of significant digits

IEEE 754 Binary Floating-Point Standard

Used by almost all FPUs; implemented in software libraries

Single precision (Java/C `float`):

- 32-bit word divided into 1 sign bit, 8-bit biased exponent, and 23-bit mantissa (≈ 7 decimal digits)
- Range: $2^{-126} - 2^{128}$

Double precision (Java/C `double`):

- 64-bit word divided into 1 sign bit, 11-bit biased exponent, and 52-bit mantissa (≈ 15 decimal digits)
- Range: $2^{-1022} - 2^{1024}$

Supports Inf and NaN

What is the output of this program?

```
#include <stdio.h>

main() {

  float a[] = { 10000.0, 1.0, 10000.0 };
  float b[] = { 10000.0, 1.0, -10000.0 };
  float sum = 0.0;
  int i;

  for (i=0; i<3; i++)
    sum += a[i]*b[i];

  printf("sum = %f\n", sum);
}
```

Remarks:

- The result depends on the order of the operations
- Finite-wordlength operations are neither associative nor distributive

Arithmetic in Embedded Systems

Small microprocessors used in embedded systems typically do not have hardware support for floating-point arithmetic

Options:

- Software emulation of floating-point arithmetic
  - compiler/library supported
  - large code size, slow
- Fixed-point arithmetic
  - often manual implementation
  - fast and compact

Fixed-Point Arithmetic

Represent all numbers (parameters, variables) using **integers**

Use **binary scaling** to make all numbers fit into one of the integer data types, e.g.

- 8 bits (`char`, `int8_t`): $[-128, 127]$
- 16 bits (`short`, `int16_t`): $[-32768, 32767]$
- 32 bits (`long`, `int32_t`): $[-2147483648, 2147483647]$

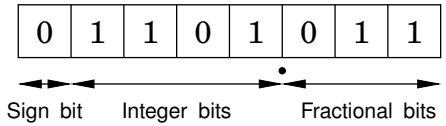
Challenges

- Must select data types to get sufficient numerical precision
- Must know (or estimate) the minimum and maximum value of every variable in order to select appropriate scaling factors
- Must keep track of the scaling factors in all arithmetic operations
- Must handle potential arithmetic overflows

Fixed-Point Representation

In fixed-point representation, a real number x is represented by an integer X with $N = m + n + 1$ bits, where

- N is the wordlength
- m is the number of integer bits (excluding the sign bit)
- n is the number of fractional bits



"Q-format": X is sometimes called a $Qm.n$ or Qn number

Conversion to and from fixed point

Conversion from real to fixed-point number:

$$X := \text{round}(x \cdot 2^n)$$

Conversion from fixed-point to real number:

$$x := X \cdot 2^{-n}$$

Example: Represent $x = 13.4$ using $Q4.3$ format

$$X = \text{round}(13.4 \cdot 2^3) = 107 \ (= 01101011_2)$$

A Note on Negative Numbers

In almost all CPUs today, negative integers are handled using **two's complement**: A "1" in the sign bit means that 2^N should be subtracted

Example ($N = 8$):

Binary representation	Interpretation
00000000	0
00000001	1
⋮	⋮
01111111	127
10000000	-128
10000001	-127
⋮	⋮
1111111	-1

Range vs Resolution for Fixed-Point Numbers

A $Qm.n$ fixed-point number can represent real numbers in the range

$$[-2^m,\ 2^m - 2^n]$$

while the resolution is

$$2^{-n}$$

Fixed range and resolution

- n too small $\Rightarrow$ poor resolution
- n too large $\Rightarrow$ risk of overflow

Fixed-Point Addition/Subtraction

Two fixed-point numbers in the same $Qm.n$ format can be added or subtracted directly

The result will have the same number of fractional bits

$$z = x + y \quad \Leftrightarrow \quad Z = X + Y$$

$$z = x - y \quad \Leftrightarrow \quad Z = X - Y$$

- The result will in general require $N + 1$ bits; risk of overflow

Example: Addition with Overflow

Two numbers in $Q4.3$ format are added:

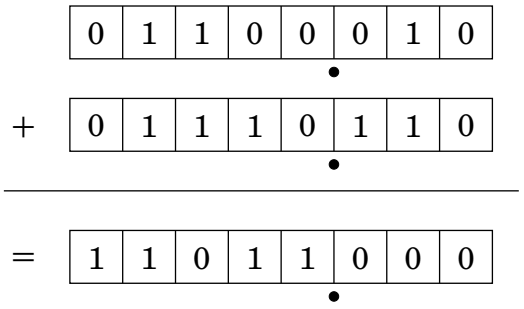
$$x = 12.25 \quad \Rightarrow \quad X = 98$$

$$y = 14.75 \quad \Rightarrow \quad Y = 118$$

$$Z = X + Y = 216$$

This number is however out of range and will be interpreted as

$$216 - 256 = -40 \quad \Rightarrow \quad z = -5.0$$



Fixed-Point Multiplication and Division

If the operands and the result are in the same Q-format, multiplication and division are done as

$$z = x \cdot y \quad \Leftrightarrow \quad Z = (X \cdot Y)/2^n$$

$$z = x/y \quad \Leftrightarrow \quad Z = (X \cdot 2^n)/Y$$

- Double wordlength is needed for the intermediate result
- Division by 2^n is implemented as a right-shift by n bits
- Multiplication by 2^n is implemented as a left-shift by n bits
- The lowest bits in the result are truncated (round-off noise)
- Risk of overflow

Example: Multiplication

Two numbers in $Q5.2$ format are multiplied:

$$x = 6.25 \quad \Rightarrow \quad X = 25$$

$$y = 4.75 \quad \Rightarrow \quad Y = 19$$

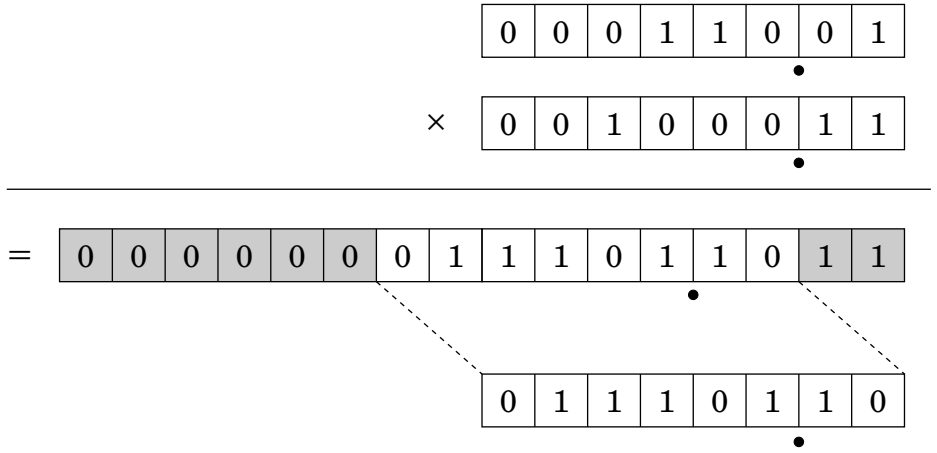
Intermediate result:

$$X \cdot Y = 475$$

Final result:

$$Z = 475/2^2 = 118 \quad \Rightarrow \quad z = 29.5$$

(exact result is 29.6875)



Multiplication of Operands with Different Q-format

In general, multiplication of two fixed-point numbers $Qm_1.n_1$ and $Qm_2.n_2$ gives an intermediate result in the format

$$Qm_1+m_2.n_1+n_2$$

which may then be right-shifted $n_1+n_2-n_3$ steps and stored in the format

$$Qm_3.n_3$$

Common case: $n_2 = n_3 = 0$ (one real operand, one integer operand, and integer result). Then

$$Z = (X \cdot Y)/2^{n_1}$$

Implementation of Multiplication in C

Assume $Q4.3$ operands and result

```
#include <inttypes.h>     /* define int8_t, etc. (Linux only)      */
#define n 3               /* number of fractional bits             */
int8_t X, Y, Z;           /* Q4.3 operands and result              */
int16_t temp;             /* Q9.6 intermediate result              */
...
temp = (int16_t)X * Y;    /* cast operands to 16 bits and multiply */
temp = temp >> n;         /* divide by 2^n                         */
Z = temp;                 /* truncate and assign result            */
```

Implementation of Multiplication in C with Rounding and Saturation

```
#include <inttypes.h>     /* defines int8_t, etc. (Linux only)     */
#define n 3               /* number of fractional bits             */
int8_t X, Y, Z;           /* Q4.3 operands and result              */
int16_t temp;             /* Q9.6 intermediate result              */
...
temp = (int16_t)X * Y;    /* cast operands to 16 bits and multiply */
temp = temp + (1 << n-1); /* add 1/2 to give correct rounding      */
temp = temp >> n;         /* divide by 2^n                         */
if (temp > INT8_MAX)      /* saturate the result before assignment */
   Z = INT8_MAX;
else if (temp < INT8_MIN)
   Z = INT8_MIN;
else
   Z = temp;
```

Implementation of Division in C with Rounding

```
#include <inttypes.h>     /* define int8_t, etc. (Linux only)      */
#define n 3               /* number of fractional bits             */
int8_t X, Y, Z;           /* Q4.3 operands and result              */
int16_t temp;             /* Q9.6 intermediate result              */
...
temp = (int16_t)X << n;   /* cast operand to 16 bits and shift     */
temp = temp + (Y >> 1);   /* Add Y/2 to give correct rounding      */
temp = temp / Y;          /* Perform the division (expensive!)     */
Z = temp;                 /* Truncate and assign result            */
```

Realizations

A linear controller

$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_n z^{-n}}{1 + a_1 z^{-1} + \ldots + a_n z^{-n}}$$

can be realized in a number of different ways with equivalent input-output behavior, e.g.

- Direct form
- Companion (canonical) form
- Series (cascade) or parallel form

Direct Form

The input-output form can be directly implemented as

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

- Nonminimal (all old inputs and outputs are used as states)
- Very sensitive to roundoff in coefficients
- Avoid!

Companion Forms

E.g. controllable or observable canonical form

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

$$u(k) = \begin{pmatrix} b_1 & b_2 & \cdots & b_n \end{pmatrix} x(k)$$

- Same problem as for the Direct form
- Very sensitive to roundoff in coefficients
- Avoid!

Pole Sensitivity

How sensitive are the poles to errors in the coefficients?

Assume characteristic polynomial with distinct roots. Then

$$A(z) = 1 - \sum_{k=1}^{n} a_k z^{-k} = \prod_{j=1}^{n} (1 - p_j z^{-1})$$

Pole sensitivity:

$$\frac{\partial p_i}{\partial a_k}$$

The chain rule gives

$$\frac{\partial A(z)}{\partial p_i} \frac{\partial p_i}{\partial a_k} = \frac{\partial A(z)}{\partial a_k}$$

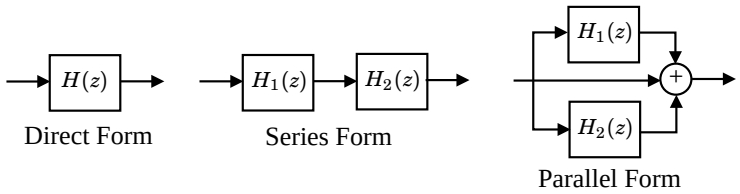
Evaluated in $z = p_i$ we get

$$\frac{\partial p_i}{\partial a_k} = \frac{p_i^{n-k}}{\prod_{j=1, j\neq i}^{n} (p_i - p_j)}$$

- Having poles close to each other is bad
- For stable filter, a_n is the most sensitive parameter

Better: Series and Parallel Forms

Divide the transfer function of the controller into a number of first- or second-order subsystems:



- Try to balance the gain such that each subsystem has about the same amplification

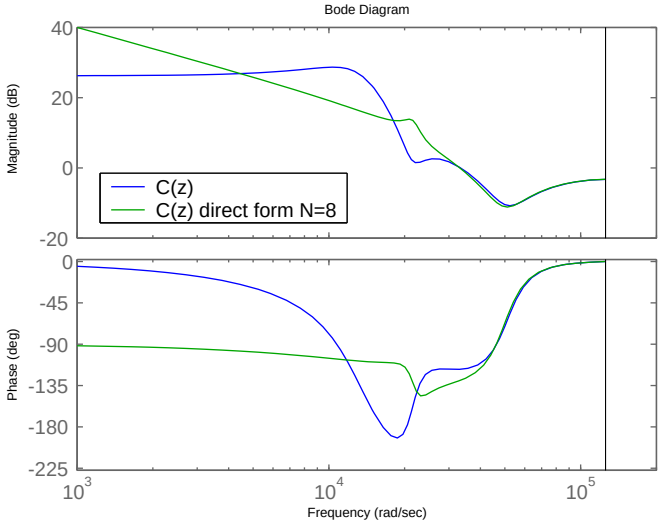
Example: Series and Parallel Forms

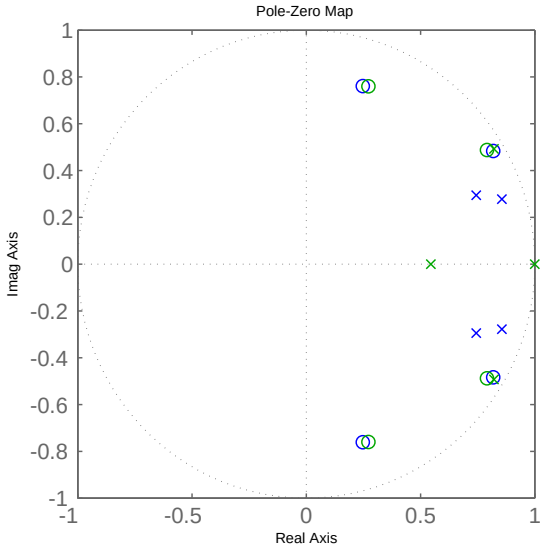
$$C(z) = \frac{z^4 - 2.13z^3 + 2.351z^2 - 1.493z + 0.5776}{z^4 - 3.2z^3 + 3.997z^2 - 2.301z + 0.5184} \quad \text{(Direct)}$$

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

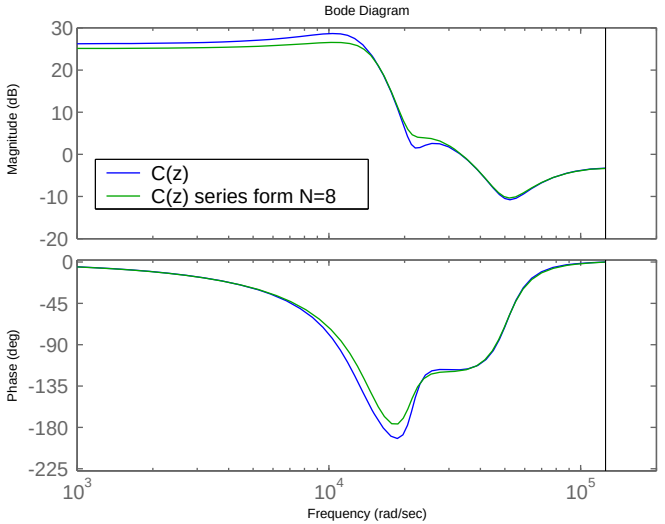
$$= 1 + \frac{-5.396z + 6.302}{z^2 - 1.712z + 0.81} + \frac{6.466z - 4.907}{z^2 - 1.488z + 0.64} \quad \text{(Parallel)}$$

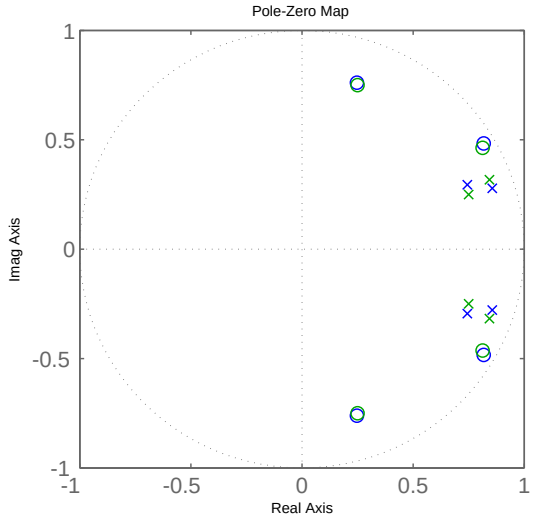
Direct form with quantized coefficients ($N = 8$, $n = 4$):





Series form with quantized coefficients ($N = 8$, $n = 4$):





Jackson's Rules for Series Realizations

How to pair and order the 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.

Well-Conditioned Parallel Realizations

Assume n_r distinct real poles and n_c distinct complex-pole pairs

Modal (diagonal/parallel/coupled) form:

$$z_i(k+1) = \lambda_i z_i(k) + \beta_i y(k) \qquad i = 1, \ldots, n_r$$

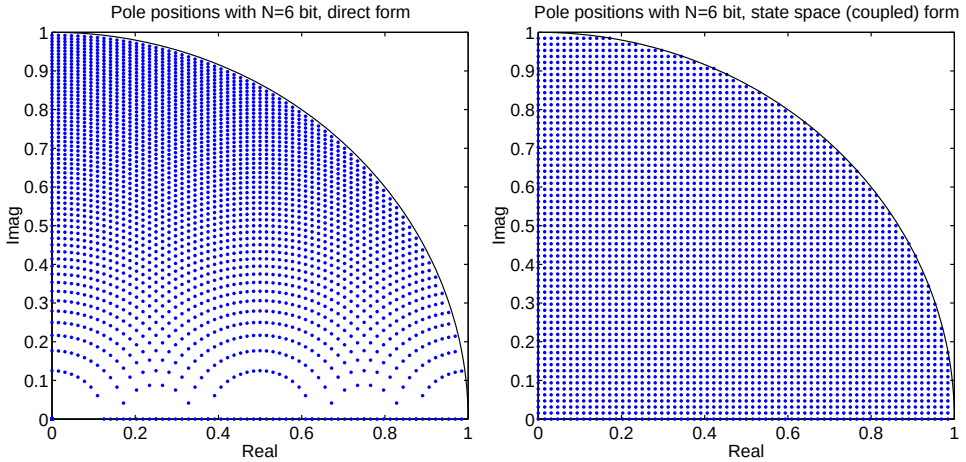
$$v_i(k+1) = \begin{pmatrix} \sigma_i & \omega_i \\ -\omega_i & \sigma_i \end{pmatrix} v_i(k) + \begin{pmatrix} \gamma_{i1} \\ \gamma_{i2} \end{pmatrix} y(k) \qquad i = 1, \ldots, n_c$$

$$u(k) = Dy(k) + \sum_{i=1}^{n_r} \gamma_i z_i(k) + \sum_{i=1}^{n_c} \delta_i^T v_i(k)$$

Matlab: `sysm = canon(sys,'modal')`

Multiple eigenvalues require Jordan blocks

Possible Pole Locations for Direct vs Modal Form



Short Sampling Interval Modification

In the state update equation

$$x(k+1) = \Phi x(k) + \Gamma y(k)$$

the system matrix Φ will be close to I if h is small. Round-off errors in the coefficients of Φ can have drastic effects.

Better: use the modified equation

$$x(k+1) = x(k) + (\Phi - I)x(k) + \Gamma y(k)$$

- Both $\Phi - I$ and Γ are roughly proportional to h
  - Less round-off noise in the calculations
- Also known as the δ-form

Short Sampling Interval and Integral Action

Fast sampling and slow integral action can give roundoff problems:

$$I(k+1) = I(k) + \underbrace{e(k) \cdot h/T_i}_{\approx 0}$$

Possible solutions:

- Use a dedicated high-resolution variable (e.g. 32 bits) for the I-part
- Update the I-part at a slower rate

General problem for filters with very different time constants

A Unifying Framework for Finite Wordlength Realizations

(Hilaire, Chevrel, Whidborn, *IEEE Trans. Circuits & Systems*, 2007)

Specialized implicit state-space form of a filter with input U, intermediate variable T, state X, output Y:

$$\begin{aligned} JT(k+1) &= MX(k) + NU(k) \\ X(k+1) &= KT(k+1) + PX(k) + QU(k) \\ Y(k) &= LT(k+1) + RX(k) + SU(k) \end{aligned}$$

Covers δ-form, direct forms, cascade/parallel forms, lattice filters, mixed q/δ, ...

Sensitivity towards coefficient quantization can be analyzed

Pulse-Width Modulation (PWM)

Poor D-A resolution (e.g. 1 bit) can often be handled by fast switching between levels + low-pass filtering

The new control variable is the duty-cycle of the switched signal