# Cast Binary Point Analysis

Analyze cast

y = a

where the types of u and y are the following binary-point scaled fixed-point data types.

ntu =

DataTypeMode: Fixed-point: binary point scaling

Signedness: Signed

WordLength: 16

FractionLength: 10

nty =

DataTypeMode: Fixed-point: binary point scaling

Signedness: Signed

WordLength: 8

FractionLength: 4

**Extreme values of input u**

Data type is numerictype(1,16,10)

Real World Notation: Binary Point

Value

31.9990234375 = 011111.1111111111

0.0009765625 = 000000.0000000001

0 = 000000.0000000000

-0.0009765625 = 111111.1111111111

-32 = 100000.0000000000

**Extreme values of output y**

Data type is numerictype(1,8,4)

Real World Notation: Binary Point

Value

7.9375 = 0111.1111

0.0625 = 0000.0001

0 = 0000.0000

-0.0625 = 1111.1111

-8 = 1000.0000

# Overflow Handling

Overflows can be handled in either of two ways.

The first is to saturate out of ranges values on the right side to the maximum representable output value and to saturate out of ranges values on the left side to the minimum representable output value.

The second is to wrap around like on the hands of a clock.

Depending on rounding mode the number of overflows possible and which specific values overflow can be slightly different.

**numRepresentableInputs** **numInputsThatOverflow** **ratioRepresentableInputsOverflow**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Floor**  65536 49152 0.75

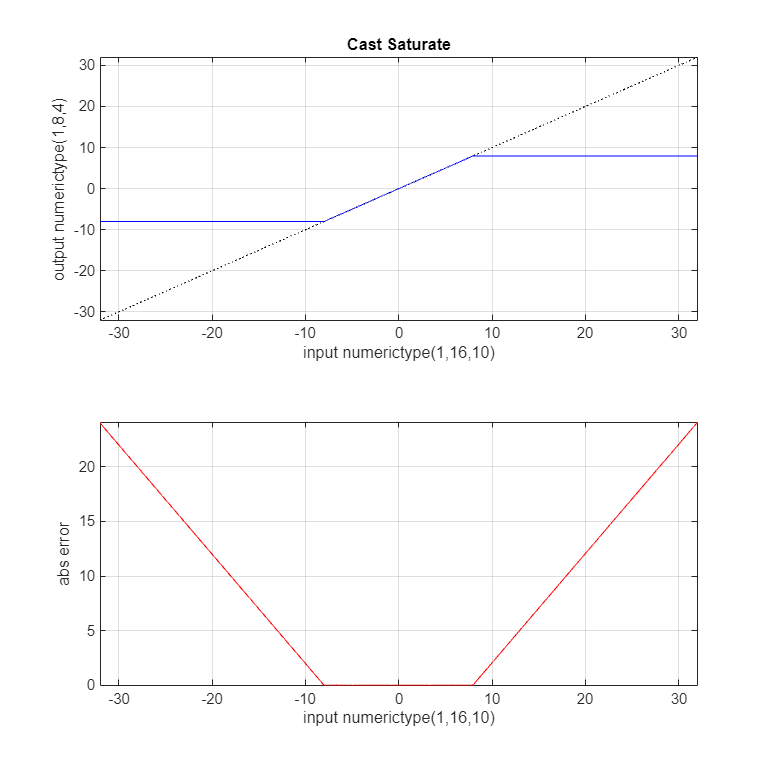
**Nearest**  65536 49152 0.75

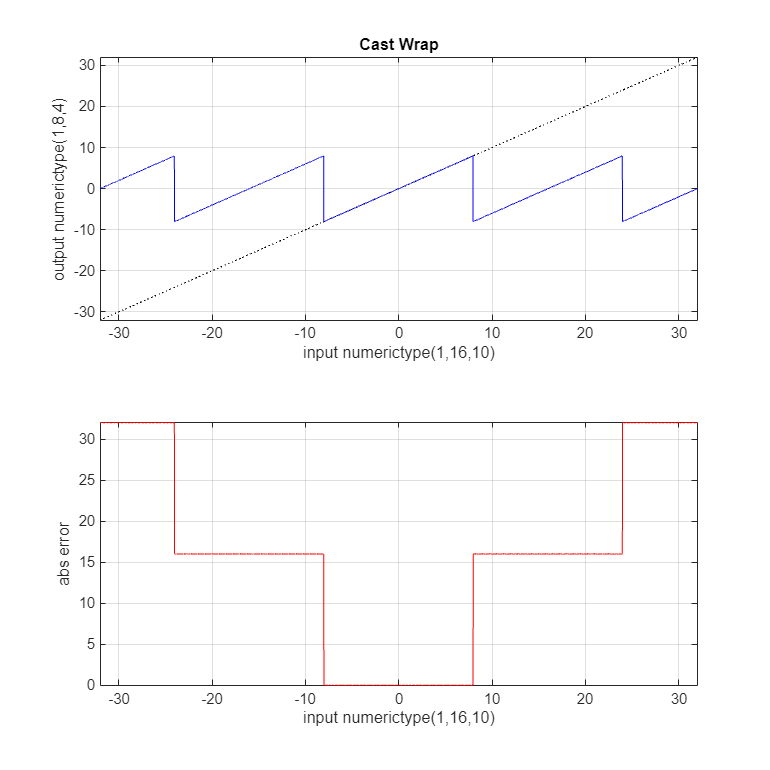
**Zero**  65536 49089 0.749038696289062

**Ceiling**  65536 49152 0.75

**Round**  65536 49153 0.750015258789062

**Convergent** 65536 49152 0.75





# Nearest Rounding (nearest with ties toward +Inf)

With nearest rounding, rounding error will be in the range (-0.5, 0.5] bits. In real-world values, the rounding error is (-0.5\*SlopeY, 0.5\*SlopeY].

In simple terms, the absolute rounding error is always half a bit or less.

Exact ties will round up, i.e. in the direction of +Inf.

For fixed-point to fixed-point casts, a big advantage of nearest rounding is that among rounding methods that give half a bit or less of error, nearest requires the smallest and fastest code. Rounding to Floor is even leaner, but it doubles the worst case rounding error. Compare code examples below.

Below mid-point, round down

Type Real World Notation: Binary Point

Value

numerictype(1,16,10) 7.9052734375 = 000111.1110011111

numerictype(1,16,4) 7.875 = 000000000111.1110

numerictype(1,8,4) 7.875 = 0111.1110

At mid-point, round up

Type Real World Notation: Binary Point

Value

numerictype(1,16,10) 7.90625 = 000111.1110100000

numerictype(1,16,4) 7.9375 = 000000000111.1111

numerictype(1,8,4) 7.9375 = 0111.1111

Above mid-point, round up

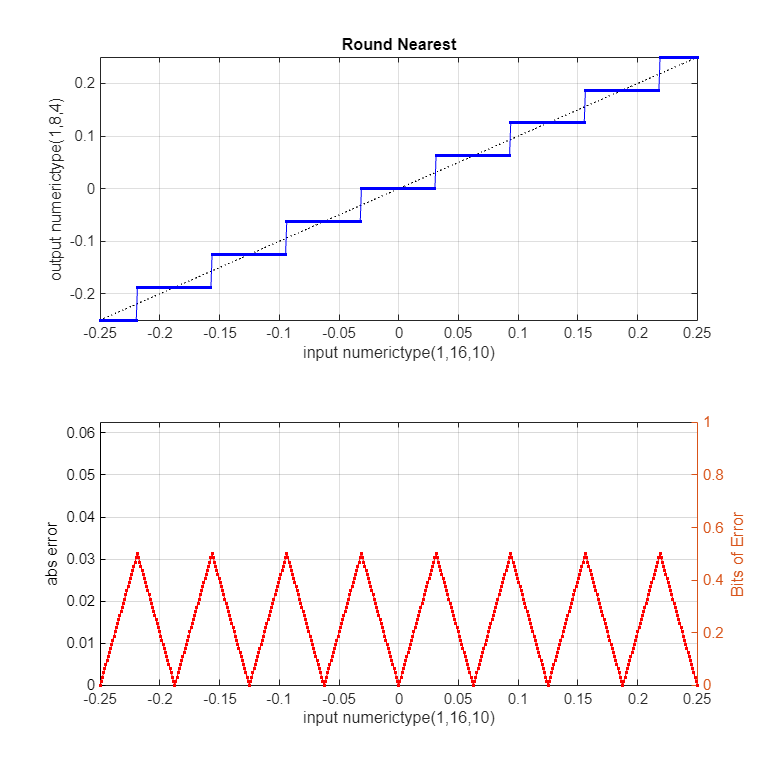
Type Real World Notation: Binary Point

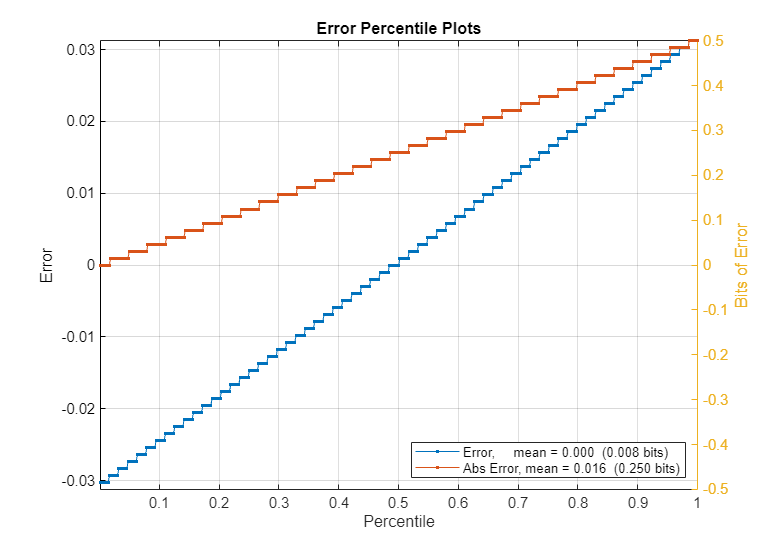
Value

numerictype(1,16,10) 7.9072265625 = 000111.1110100001

numerictype(1,16,4) 7.9375 = 000000000111.1111

numerictype(1,8,4) 7.9375 = 0111.1111





# Convergent Rounding (nearest with ties to even)

Convergent rounding is like nearest rounding, except ties are rounded to the nearest even output. Even means the last bit of the output is 0.

With convergent rounding, rounding error will be in the range [-0.5, 0.5] bits. In real-world values, the rounding error is [-0.5\*SlopeY, 0.5\*SlopeY].

An advantage of Convergent over Nearest is that Convergent will have near zero average rounding error (under mild assumption regarding input distribution). If the input is feed to calculations that accumulate errors such as integration, then Convergent is least likely to have significant error drift build up. If the downstream operations do not accumulate or there are corrections from an outer feedback loop, then average error may not be of importance.

At mid-point, round to even

Type Real World Notation: Binary Point

Value

numerictype(1,16,10) 7.84375 = 000111.1101100000

numerictype(1,16,4) 7.875 = 000000000111.1110

numerictype(1,8,4) 7.875 = 0111.1110

At mid-point, round to even

Type Real World Notation: Binary Point

Value

numerictype(1,16,10) 7.90625 = 000111.1110100000

numerictype(1,16,4) 7.875 = 000000000111.1110

numerictype(1,8,4) 7.875 = 0111.1110

Below mid-point, round down

Type Real World Notation: Binary Point

Value

numerictype(1,16,10) 7.9052734375 = 000111.1110011111

numerictype(1,16,4) 7.875 = 000000000111.1110

numerictype(1,8,4) 7.875 = 0111.1110

Above mid-point, round up

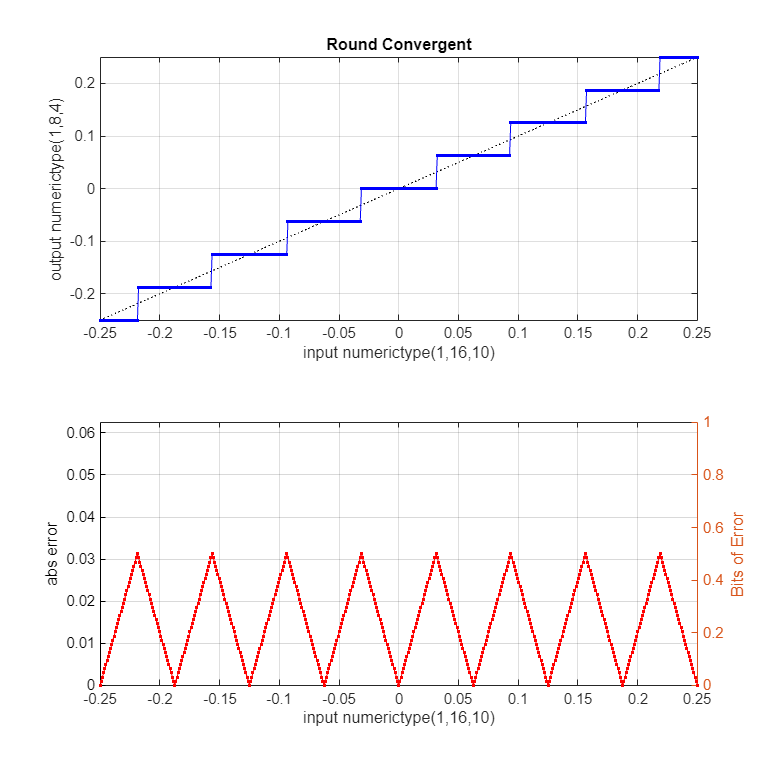
Type Real World Notation: Binary Point

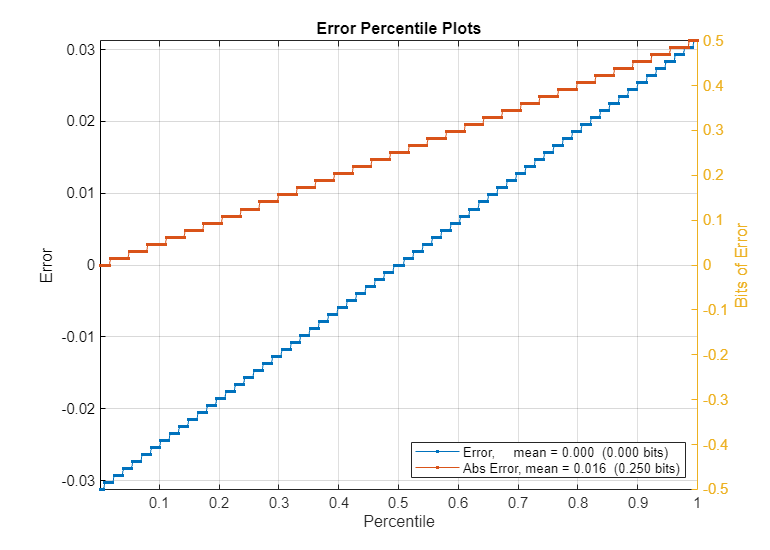
Value

numerictype(1,16,10) 7.9072265625 = 000111.1110100001

numerictype(1,16,4) 7.9375 = 000000000111.1111

numerictype(1,8,4) 7.9375 = 0111.1111





# Floor Rounding

With floor rounding, rounding error will be in the range (-1, 0] bits. In real-world values, the rounding error is (-SlopeY, 0].

In simple terms, the absolute rounding error is always less than one bit.

For fixed-point to fixed-point casts, a big advantage of floor rounding is that it requires the smallest and fastest code of all the rounding methods. Compare code examples below.

Everything rounds down

Type Real World Notation: Binary Point

Value

numerictype(1,16,10) 7.8759765625 = 000111.1110000001

numerictype(1,16,4) 7.875 = 000000000111.1110

numerictype(1,8,4) 7.875 = 0111.1110

Everything rounds down

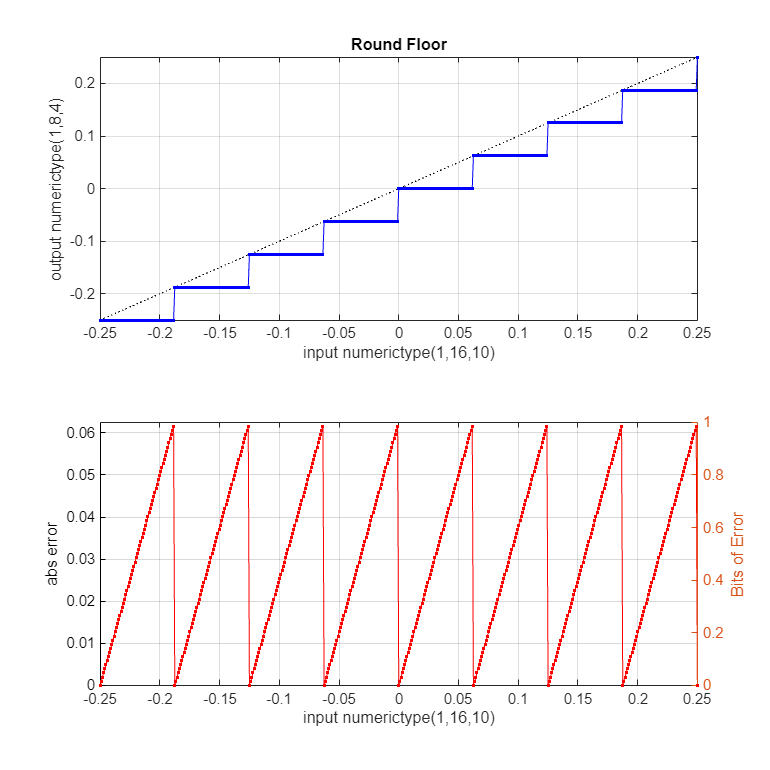
Type Real World Notation: Binary Point

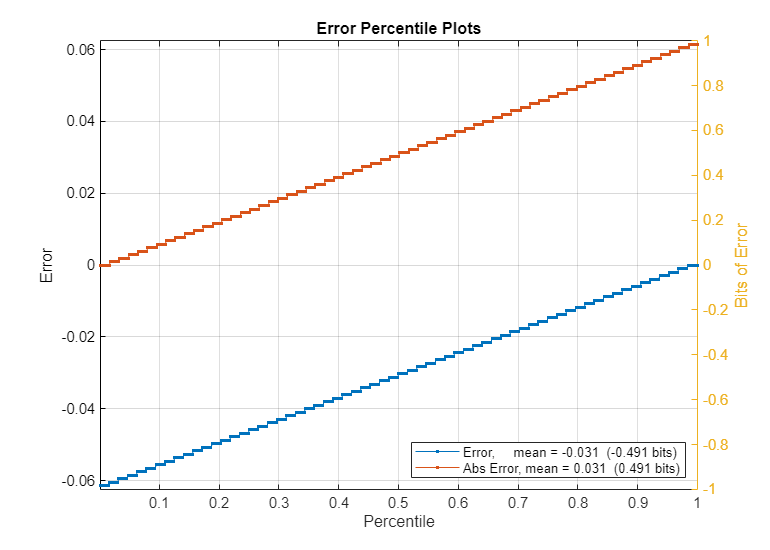
Value

numerictype(1,16,10) 7.9365234375 = 000111.1110111111

numerictype(1,16,4) 7.875 = 000000000111.1110

numerictype(1,8,4) 7.875 = 0111.1110





# Ceiling Rounding

With ceiling rounding, rounding error will be in the range [0, 1) bits. In real-world values, the rounding error is [0, SlopeY).

In simple terms, the absolute rounding error is always less than one bit.

Ceiling rounding is rarely used. It is costlier than Floor without numerical advantages. It is mainly used in specialized applications like interval analysis.

Everything rounds up

Type Real World Notation: Binary Point

Value

numerictype(1,16,10) 7.8759765625 = 000111.1110000001

numerictype(1,16,4) 7.9375 = 000000000111.1111

numerictype(1,8,4) 7.9375 = 0111.1111

Everything rounds up

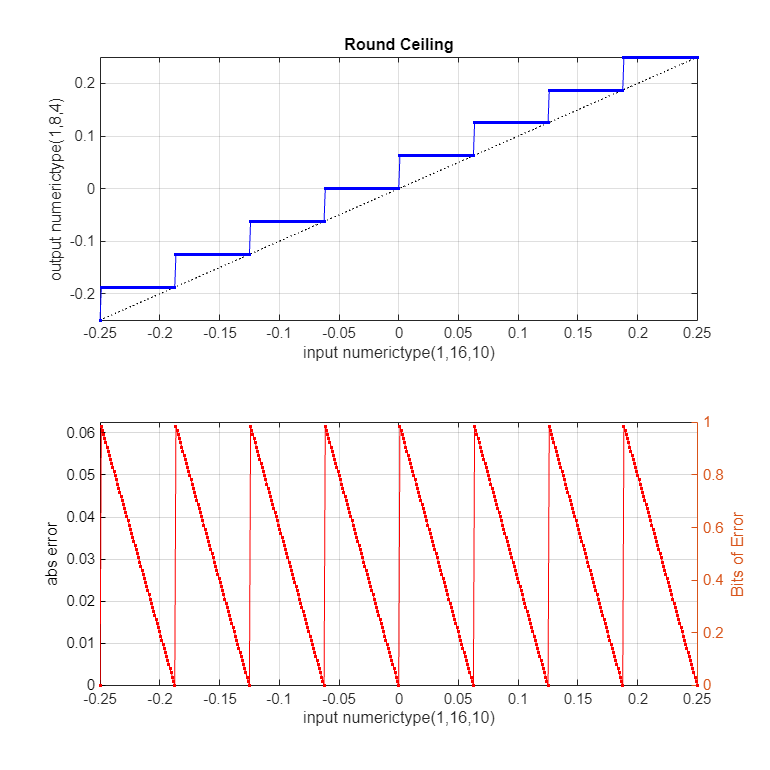
Type Real World Notation: Binary Point

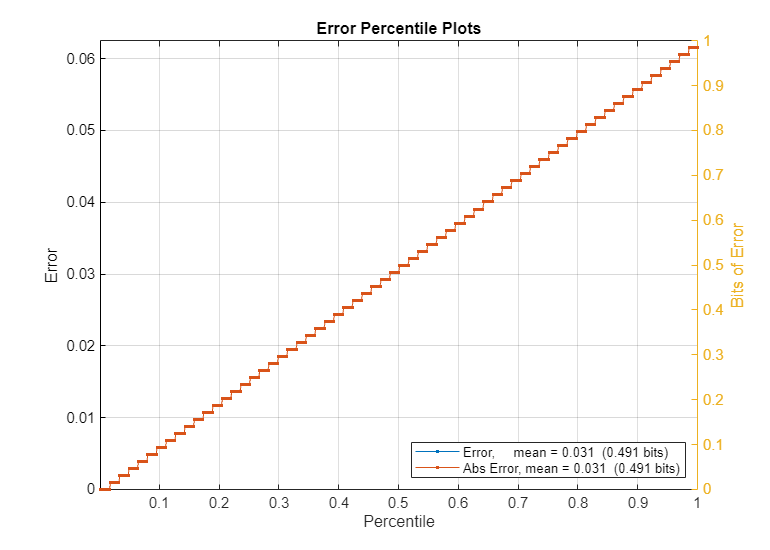
Value

numerictype(1,16,10) 7.9365234375 = 000111.1110111111

numerictype(1,16,4) 7.9375 = 000000000111.1111

numerictype(1,8,4) 7.9375 = 0111.1111





# Zero Rounding

With zero rounding, rounding error will be in the range (-1, 1) bits. In real-world values, the rounding error is (-SlopeY, SlopeY).

In simple terms, the absolute rounding error is always less than one bit.

For fixed-point to fixed-point casts, zero rounding is rarely used because it is costlier than Floor and does not have numerical advantages.

For floating-point to fixed-point casts, zero rounding is widely used. The C standard mandates that rounding from floating-point to integer will be round to zero. This gives a big efficiency advantage to zero rounding in floating-point to integer and fixed-point types.

For fixed-point division, zero rounding is also often advantageous. The C99 standard mandates that signed integer division must round toward zero. (Note, for unsigned round to Floor and Zero are identical.)

Negatives rounds up

Type Real World Notation: Binary Point

Value

numerictype(1,16,10) -0.1240234375 = 111111.1110000001

numerictype(1,16,4) -0.0625 = 111111111111.1111

numerictype(1,8,4) -0.0625 = 1111.1111

Negatives rounds up

Type Real World Notation: Binary Point

Value

numerictype(1,16,10) -0.0634765625 = 111111.1110111111

numerictype(1,16,4) -0.0625 = 111111111111.1111

numerictype(1,8,4) -0.0625 = 1111.1111

Positives round down

Type Real World Notation: Binary Point

Value

numerictype(1,16,10) 0.0634765625 = 000000.0001000001

numerictype(1,16,4) 0.0625 = 000000000000.0001

numerictype(1,8,4) 0.0625 = 0000.0001

Positives round down

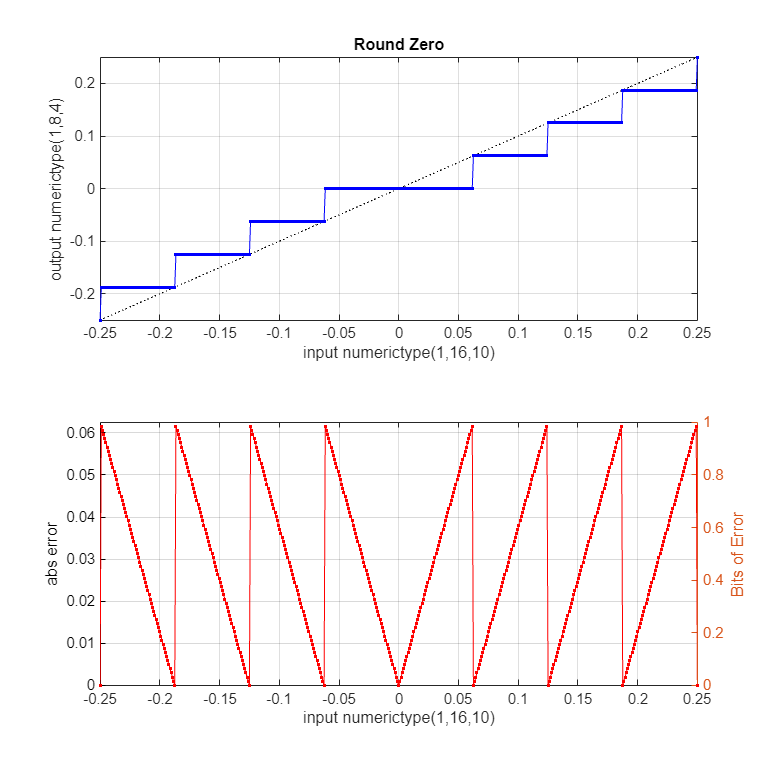
Type Real World Notation: Binary Point

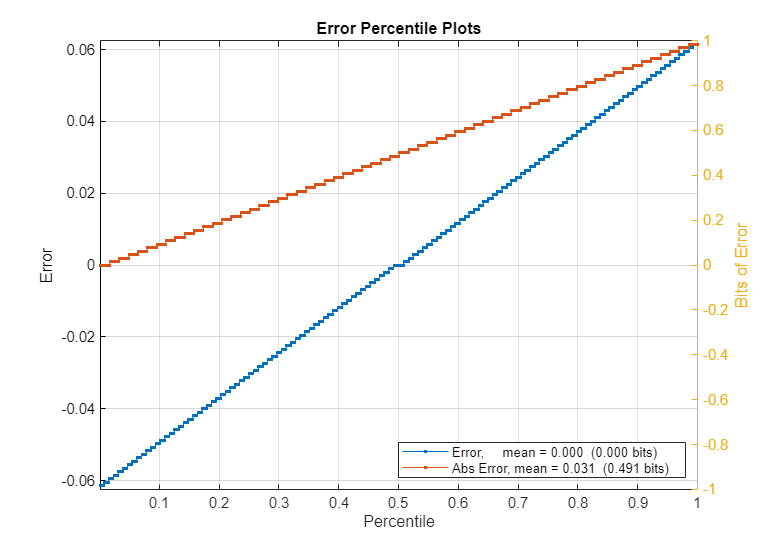
Value

numerictype(1,16,10) 0.1240234375 = 000000.0001111111

numerictype(1,16,4) 0.0625 = 000000000000.0001

numerictype(1,8,4) 0.0625 = 0000.0001





# Map Math in Real World Value to Operations on Stored Integers

Math in Real World Values is

Vy = Vu

The fixed-point scaling for each variable is

Vy = Sy \* Qy + By

Vu = Su \* Qu + Bu

where

V\_ are real-world values

Q\_ are stored integer vales

S\_ are scaling slopes

B\_ are scaling biases

Substituting the scaling equations into the real-world math equation gives.



Solving for the output stored integer value show the operations that must be done in code to implement the math.

Qy =



Keep in mind that the scaling will be constants in the real-time embedded code, so at worst, one multiplication by a constant and one addition by a constant are needed. Depending on how well the scaling of the input and output are "matched", the additive term may go away and the multiplicative term my reduce to an exact power of two. If the multiplicative term is an exact power of two, then a fast shift left or shift right can be used.

Since this case analysis is restricted to binary-point scaling, we can simplify things further.

S\_ = 2^E\_

B\_ = 0

where

E\_ is the scaling fixed exponent

So for binary-point scaling, the fixed-point scaling for each variable is

Vy = 2^Ey \* Qy

Vu = 2^Eu \* Qu

Substituting known scalings into these ideal math equation gives the following.

Vu =



Vy =





Now solving this set of equations for Qy describes the net-scaling and conceptual level operations on the stored integers.

Qy =



For the binary-point, a shift left or shift right is all that is needed for the ideal math.

Note, in addition to the ideal math, the implementation will need to handle overflows, rounding, and changing of container types.

# C code generation from MATLAB Coder

Let's look at the generated code needed for various combinations of overflow handling and rounding handling.

Keep an eye out of additional operations needed for rounding, usually and additive term.

Keep an eye out for if then else to handle saturation.

Also keep an eye on C type casts. So of these casts will be part of the C language rule of never doing math on types smaller than an int. Some up casts may be needed to prevent overflows in intermediate results. Down casts are usually where any overflow wrapping will occur.

## Wrapping Floor

Code generation successful: View report

#include "exFixptCastWrapFloor.h"

int8\_T exFixptCastWrapFloor(int16\_T a)

{

return (int8\_T)((int32\_T)a >> 6);

}

## Wrapping Nearest (Ties toward +Inf)

Code generation successful: View report

#include "exFixptCastWrapNearTieInf.h"

int8\_T exFixptCastWrapNearTieInf(int16\_T a)

{

return (int8\_T)(((int32\_T)a + 32) >> 6);

}

## Wrapping Round (Ties away from zero)

Code generation successful: View report

#include "exFixptCastWrapNearTieAway.h"

int8\_T exFixptCastWrapNearTieAway(int16\_T a)

{

return (int8\_T)(((int32\_T)a >> 6) +

((((int32\_T)a & 32) != 0) &&

((((int32\_T)a & 31) != 0) || ((int32\_T)a > 0))));

}

## Wrapping Convergent (Ties to even valued)

Code generation successful: View report

#include "exFixptCastWrapNearTieEven.h"

int8\_T exFixptCastWrapNearTieEven(int16\_T a)

{

return (int8\_T)((((int32\_T)a >> 6) +

((((int32\_T)a & 32) != 0) && (((int32\_T)a & 31) != 0))) +

((((int32\_T)a & 63) == 32) && (((int32\_T)a & 64) != 0)));

}

## Wrapping Ceiling

Code generation successful: View report

#include "exFixptCastWrapCeil.h"

int8\_T exFixptCastWrapCeil(int16\_T a)

{

return (int8\_T)(((int32\_T)a + 63) >> 6);

}

## Wrapping Zero

Code generation successful: View report

#include "exFixptCastWrapZero.h"

int8\_T exFixptCastWrapZero(int16\_T a)

{

int32\_T i;

if ((int32\_T)a < 0) {

i = 63;

} else {

i = 0;

}

return (int8\_T)((int32\_T)(int16\_T)((int32\_T)a + i) >> 6);

}

## Saturating Floor

Code generation successful: View report

#include "exFixptCastSatFloor.h"

int8\_T exFixptCastSatFloor(int16\_T a)

{

int16\_T i;

i = (int16\_T)((int32\_T)a >> 6);

if ((int32\_T)i > 127) {

i = (int16\_T)127;

} else if ((int32\_T)i < -128) {

i = (int16\_T)-128;

}

return (int8\_T)i;

}

## Saturating Nearest (Ties toward +Inf)

Code generation successful: View report

#include "exFixptCastSatNearTieInf.h"

int8\_T exFixptCastSatNearTieInf(int16\_T a)

{

int16\_T i;

i = (int16\_T)(((int32\_T)a + 32) >> 6);

if ((int32\_T)i > 127) {

i = (int16\_T)127;

} else if ((int32\_T)i < -128) {

i = (int16\_T)-128;

}

return (int8\_T)i;

}

## Saturating Round (Ties away from zero)

Code generation successful: View report

#include "exFixptCastSatNearTieAway.h"

int8\_T exFixptCastSatNearTieAway(int16\_T a)

{

int16\_T i;

i = (int16\_T)(((int32\_T)a >> 6) +

((((int32\_T)a & 32) != 0) &&

((((int32\_T)a & 31) != 0) || ((int32\_T)a > 0))));

if ((int32\_T)i > 127) {

i = (int16\_T)127;

} else if ((int32\_T)i < -128) {

i = (int16\_T)-128;

}

return (int8\_T)i;

}

## Saturating Convergent (Ties to even valued)

Code generation successful: View report

#include "exFixptCastSatNearTieEven.h"

int8\_T exFixptCastSatNearTieEven(int16\_T a)

{

int16\_T i;

i = (int16\_T)((((int32\_T)a >> 6) +

((((int32\_T)a & 32) != 0) && (((int32\_T)a & 31) != 0))) +

((((int32\_T)a & 63) == 32) && (((int32\_T)a & 64) != 0)));

if ((int32\_T)i > 127) {

i = (int16\_T)127;

} else if ((int32\_T)i < -128) {

i = (int16\_T)-128;

}

return (int8\_T)i;

}

## Saturating Ceiling

Code generation successful: View report

#include "exFixptCastSatCeil.h"

int8\_T exFixptCastSatCeil(int16\_T a)

{

int16\_T i;

i = (int16\_T)(((int32\_T)a + 63) >> 6);

if ((int32\_T)i > 127) {

i = (int16\_T)127;

} else if ((int32\_T)i < -128) {

i = (int16\_T)-128;

}

return (int8\_T)i;

}

## Saturating Zero

Code generation successful: View report

#include "exFixptCastSatZero.h"

int8\_T exFixptCastSatZero(int16\_T a)

{

int32\_T i;

int16\_T i1;

if ((int32\_T)a < 0) {

i = 63;

} else {

i = 0;

}

i1 = (int16\_T)((int32\_T)(int16\_T)((int32\_T)a + i) >> 6);

if ((int32\_T)i1 > 127) {

i1 = (int16\_T)127;

} else if ((int32\_T)i1 < -128) {

i1 = (int16\_T)-128;

}

return (int8\_T)i1;

}