**<https://www.eetimes.com/document.asp?doc_id=1274485>**

# An introduction to different rounding algorithms

[**EE Times**](https://www.eetimes.com/document.asp?doc_id=1274485)

1/4/2006 08:00 PM EST   
[Post a comment](https://www.eetimes.com/document.asp?doc_id=1274485#msgs)

NO RATINGS

* [LOGIN TO RATE](javascript:void(0))

https://m.eet.com/images/images/spacer.gif

[inShare](javascript:void(0);)

***Editor's Note:* I just thought I should mention that there's an ongoing, ever-evolving, and dramatically different presentation of this topic entitled**[Rounding 101](http://www.clivemaxfield.com/diycalculator/sp-round.shtml)**on my personal website.**

It is often said that it's only when you try to explain something to someone else that you come to realize that there are "holes" in your understanding of the topic in question. Such was the case when it came to presenting the concept of rounding in our recently-published book: [How Computers Do Math](http://www.amazon.com/gp/product/0471732788) (ISBN: 0471732788) featuring a virtual [8-bit](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=8-bit&x=&y=) computer-calculator called the [DIY Calculator](http://www.diycalculator.com/). In fact, the mind soon boggles at the variety and intricacies of the rounding algorithms that may be used for different applications.

For example, we have *round-up*, *round-down*, *round-toward-nearest*, *arithmetic rounding*, *round-half-up*, *round-half-down*, *round-half-even*, *round-half-odd*, *round-toward-zero*, *round-away-from-zero*, *round-ceiling*, *round-floor*, *truncation* (chopping), *round-alternate*, and *round-random* (stochastic rounding), to name but a few. What makes things even more exciting is that some of these terms can refer to the same thing (or not, depending on the application and to whom you are talking). Similarly, some rounding schemes work one way on sign-magnitude values (like standard decimal numbers and sign-magnitude binary numbers) and a different way on complement representations (like twos complement [signed binary] and tens complement [signed decimal]).

But, nothing daunted, we are going to rip the veils asunder and discover more about rounding than most of us ever wanted to know. We will commence by introducing a smorgasbord of basic concepts. Then, in order to provide some "meat," Tim Vanevenhoven at [AccelChip](http://www.accelchip.com/" \t "_new) was kind enough to create and run some test cases in MATLAB® from [The MathWorks](http://www.themathworks.com/) to illustrate the types of errors associated with different rounding schemes applied at various stages throughout a fixed-point digital filter implemented in hardware (the MATLAB source files are available for you to [download](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=download&x=&y=) and play with as described later in this article).

**Rounding in decimal**  
We'll start the ball rolling by considering the various rounding schemes in the context of the decimal numbers we know and love so well. The most fundamental fact associated with rounding is that it involves transforming some quantity from a greater precision to a lesser precision; for example, rounding a reasonably precise value like $3.21 to the nearest dollar would result in $3.00, which is a less precise entity.

Given a choice, we would generally prefer to use a rounding [algorithm](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=algorithm&x=&y=) that minimizes the effects of this loss of precision, especially in the case where multiple processing iterations – each involving rounding – can result in "creeping errors" (by this we mean that errors increase over time due to performing rounding operations on data that has previously been rounded). However, in the case of hardware implementations targeted toward tasks such as digital signal [processing](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=processing&x=&y=) (DSP) algorithms, for example, we also have to be cognizant of the overheads associated with the various rounding techniques so as to make appropriate design trade-offs.

For the purposes of the following discussions, we will assume that the goal is to round to an integer value. In real-world applications we might wish to round to any particular digit (usually a fractional digit), but the principles are exactly the same.

A summary of the actions of the main rounding modes as applied to standard (sign-magnitude) decimal values is provided in [*Table 1*](https://m.eet.com/media/1061419/max-round-tab01.gif).

**Round-toward-nearest:** This is perhaps the most intuitive of the various rounding algorithms. In this case, values such as 3.1, 3.2, 3.3, and 3.4 would round down to 3, while values of 3.6, 3.7, 3.8, and 3.9 would round up to 4. The trick, of course, is to decide what to do in the case of the half-way value 3.5. In fact, *round-toward-nearest* may be considered to be a superset of two complementary options known as *round-half-up* and *round-half-down*, each of which treats the 3.5 value in a different manner as discussed below.

**Round-half-up:** This algorithm, which may also be referred to as *arithmetic rounding*, is the one that we typically associate with the rounding we learned at grade-school. In this case, a half-way value such as 3.5 will round up to 4. One way to view this is that, at this level of precision and for this particular example, we can consider there to be ten values that commence with a 3 in the most-significant place (3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9). On this basis, it intuitively makes sense for five of the values to round down and for the other five to round up; that is, for the five values 3.0 through 3.4 to round down to 3, and for the remaining five values 3.5 through 3.9 to round up to 4.

The tricky point with the *round-half-up* algorithm arrives when we come to consider negative numbers. In the case of the values -3.1, -3.2, -3.3, and -3.4, these will all round to the nearest integer, which is -3; similarly, in the case of values like -3.6, -3.7, -3.8, and -3.9, these will all round to -4. The problem arises in the case of -3.5 and our definition as to what "up" means in the context of *round-half-up*. Based on the fact that a value of +3.5 rounds up to +4, most of us would intuitively expect a value of -3.5 to round to -4. In this case, we would say that our algorithm was *symmetric* for positive and negative values.

However, some applications (and mathematicians) regard "up" as referring to positive infinity. Based on this, -3.5 will actually round to -3, in which case we would class this as being an *asymmetric*implementation of the *round-half-up* algorithm. For example, the *round* method of the [Java](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=Java&x=&y=) Math [Library](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=Library&x=&y=) provides an asymmetric implementation of the *round-half-up* algorithm, while the *round*function in MATLAB provides a symmetric implementation. (Just to keep us on our toes, the *round* function in Visual Basic for Applications 6.0 actually implements the *round-half-even* [Banker's rounding] algorithm discussed below.)

**Round-half-down:** This acts in the opposite manner to its *round-half-up* counterpart. In this case, a half-way value such as 3.5 will round down to 3. Once again, we run into a problem when we come to consider negative numbers, depending on what we assume "down" to mean. In the case of a symmetric implementation of the algorithm, a value of -3.5 will round to -3. By comparison, in the case of an asymmetric implementation of the algorithm, in which "down" is understood to refer to negative infinity, a value of -3.5 will actually round to -4.

As a point of interest, the symmetric versions of rounding algorithms are sometimes referred to as "Gaussian implementations." This is because the theoretical frequency distribution known as a Gaussian distribution – which is named for the German mathematician and astronomer Karl Friedrich Gauss (1777-1855) – is symmetrical about its mean value.

**Round-half-even:** If half-way values are always rounded in the same direction (for example 3.5 rounds to 4 and 4.5 rounds to 5), the result can be a bias that grows as more rounding operations are performed. One solution toward minimizing this bias is to sometimes round up and sometimes round down.

In the case of the *round-half-even* algorithm (which is often referred to as *Banker's Rounding* because it is commonly used in financial calculations), half-way values are rounded toward the nearest *even*number. Thus, 3.5 will round *up* to 4 and 4.5 will round *down* to 4. This algorithm is, by definition, symmetric for positive and negative values, so both -3.5 and -4.5 will round to -4.

In the case of data sets that feature a relatively large number of "half-way" values (financial records provide a good example of this), the *round-half-even* algorithm performs significantly better than the *round-half-up* scheme in terms of total bias. However, in the case of data sets containing a relatively small number of "half-way" values – such as real-world values being applied to DSP algorithms – the overhead involved in performing the *round-half-even* algorithm in hardware does not justify its use (see also the filter examples shown later in this paper).

**Round-half-odd:** This is the theoretical counterpart to the *round-half-even* algorithm, in which half-way values are rounded toward the nearest odd number. In this case, 3.5 will round to 3 and 4.5 will round to 5 (similarly, -3.5 will round to -3, and -4.5 will round to -5). The reason we say "theoretical" is that, in practice, the *round-half-odd* algorithm is rarely (if ever) never used because it will never round to zero (rounding to zero is often a desirable attribute for rounding algorithms).

**Round-alternate:** Also known as *alternate rounding*, this is similar in concept to the *round-half-even* and *round-half-odd* schemes discussed above, in that the purpose of the *round-alternate*algorithm is to minimize the bias that can be caused by always rounding half-way values in the same direction.

In the case of the *round-half-even* approach, for example, it would be possible for a bias to occur if the data being processed contained a disproportionate number of odd and even half-way values. One solution is to use the *round-alternate* algorithm, in which the first half-way value is rounded up (for example); the next is rounded down, the next up, the next down, and so on.

**Round-random:** This may also be referred to as *random rounding*or *stochastic rounding*, where the term "stochastic" comes from the Greek *stokhazesthai*, meaning "to guess at." With this technique, in the case of "half-way" values, we effectively toss a metaphorical coin in the air and randomly (or pseudo-randomly) round the value up or down.

Although this technique typically gives the best overall result over a large number of calculations, it is only employed in very specialized applications, because the nature of this algorithm makes it difficult to implement and tricky to verify the results.

**Round-ceiling:** This refers to rounding towards positive infinity. In the case of a positive number, the result will remain unchanged if the digits to be discarded are all zero; otherwise it will be rounded up. For example, 3.0 will be rounded to 3, but 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9 will all be rounded to 4. By comparison, in the case of a negative number, the unwanted digits are simply discarded. For example, -3.0, -3.1, -3.2, -3.3, -3.4, -3.5, -3.6, -3.7, -3.8, and -3.9 will all be rounded to -3.

This algorithm (which is implemented using the *ceil* function in MATLAB), results in a cumulative positive bias and also requires additional logic when realized in hardware. For these reasons, the round-ceiling algorithm is not often used in hardware implementations. (In the case of non-hardware implementations, or during analysis of a system using software simulation, the *round-ceiling* approach is sometimes employed to determine the upper limit of the algorithm for use in diagnostic functions).

**Round-floor:** The counterpart to *round-ceiling*, this refers to rounding towards negative infinity. In the case of a negative number, the result will remain unchanged if the digits to be discarded are all zero; otherwise it will be rounded down. For example, -3.0 will be rounded to -3, but -3.1, -3.2, -3.3, -3.4, -3.5, -3.6, -3.7, -3.8, and -3.9 will all be rounded to -4. By comparison, in the case of a positive number, the unwanted digits are simply discarded. For example, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9 will all be rounded to 3.

This algorithm (which is implemented using the *floor* function in MATLAB), results in a cumulative negative bias. However, round-floor is "cheap" in terms of a hardware implementation since it involves only a simple truncation; this technique is therefore very often used with regard to hardware implementations. (In the case of non-hardware implementations, or during analysis of a system using software simulation, the *round-floor* approach is sometimes employed to determine the lower limit of the algorithm for use in diagnostic functions).

**Round-toward-zero:** As its name suggests, this refers to rounding in such a way that the result heads toward zero. For example, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9 will all be rounded to 3. Similarly, -3.1, -3.2, -3.3, -3.4, -3.5, -3.6, -3.7, -3.8, and -3.9 will all be rounded to -3.

Another way to think about this is that *round-toward-zero* (which is implemented using the *fix* function in MATLAB) acts in the same way as the *round-floor* algorithm for positive numbers and as the *round-ceiling* algorithm for negative numbers.

**Round-away-from-zero:** The counterpart to *round-toward-zero*, this refers to rounding in such a way that the result heads away from zero. For example, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9 will all be rounded to 4. Similarly, -3.1, -3.2, -3.3, -3.4, -3.5, -3.6, -3.7, -3.8, and -3.9 will all be rounded to -4.

Another way to think about this is that *round-away-from-zero* acts in the same manner as a *round-ceiling* algorithm for positive numbers and as a *round-floor* algorithm for negative numbers.

**Round-up:** The actions of this rounding [mode](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=mode&x=&y=) depend on what one means by "up". Some applications understand "up" to refer to heading towards positive infinity; in this case, *round-up* is synonymous for *round-ceiling*.

Alternatively, some applications regard "up" as referring to an absolute value heading away from zero; in this case, *round-up* acts in the same manner as the *round-away-from-zero* algorithm.

**Round-down:** The counterpart to the *round-up* algorithm, the actions of this mode depend on what one means by "down". Some applications understand "down" to refer to heading towards negative infinity; in this case, *round-down* is synonymous for *round-floor*.

Alternatively, some applications regard "down" as referring to an absolute value heading toward zero; in this case, *round-down* acts in the same manner as the *round-toward-zero* algorithm.

**Truncation:** Also known as *chopping*, truncation simply means discarding any unwanted digits. Although this would appear – on the surface – to be relatively simple, things become a little more interesting when we realize that the actions resulting from truncation vary depending on whether we are working with sign-magnitude, unsigned, or signed (complement) values. In the case of sign-magnitude values (like the standard decimal values we've been considering thus far), and also when working with unsigned binary values, the actions of truncation are identical to those of the *round-toward-zero* mode. In the case of signed binary values, however, truncation works somewhat differently for negative values (we'll return to this point shortly).

[EMAIL THIS](https://www.eetimes.com/email.asp)

[PRINT](https://www.eetimes.com/document.asp?doc_id=1274485&print=yes)

[COMMENT](https://www.eetimes.com/document.asp?doc_id=1274485#msgs)

**PAGE 1 / 2**[**NEXT >**](https://www.eetimes.com/document.asp?doc_id=1274485&page_number=2)

# An introduction to different rounding algorithms

[**EE Times**](https://www.eetimes.com/document.asp?doc_id=1274485)

1/4/2006 08:00 PM EST   
[Post a comment](https://www.eetimes.com/document.asp?doc_id=1274485&page_number=2#msgs)

NO RATINGS

* [LOGIN TO RATE](javascript:void(0))

https://m.eet.com/images/images/spacer.gif

[inShare](javascript:void(0);)

**Hardware implementations and implications**  
The next point we need to consider is when and where we may wish to actually perform rounding operations. In order to do this, we first need to take a step back to define a couple of concepts, such as the differences between integer, fractional, fixed-point, and floating-point values.

Let's start with the term "fixed-point," which refers to a way of writing (or otherwise representing, including storing in the case of computers) numerical quantities with a predetermined number of digits and with the "point" located at a single, unchanging position (this would be the "decimal point" in the case of decimal numbers, the "binary point" in the case of binary numbers, and so forth).

For example, a fixed-point sign-magnitude decimal number supporting three digits to the left of the decimal point and two digits to the right could be used to represent values in the range "999.99 to +999.99. From this, we realize that integer representations are a special case of fixed-point in which there are no fractional digits. Similarly, purely fractional representations are a special case of fixed-point in which there are no integer digits.

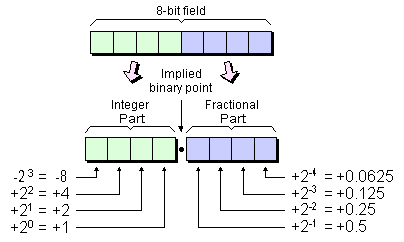
As opposed to fixed-point, a floating-point value is one that is expressed as a multiple of an appropriate power of the base of the number system, which thereby allows the point to move around. For example, using floating-point notation, the value 12.34 can also be written as 123.4 × 10-1, 1234 × 10-2, 12340 × 10-3, and so forth; and also as 1.234 × 101, 0.1234 × 102, 0.01234 × 103, and so forth.

So, one case where we might wish to perform rounding would be when working only with integers and dividing two integer values, such as 9 / 2 = 4.5, at which point we have to round the result back to an integer. Similarly, it may be necessary to perform a rounding operation after multiplying two fractional numbers together.

One very common case where rounding proves necessary is when converting from floating-point representations into their fixed-point counterparts. For example, DSP algorithms are typically first analyzed and evaluated using floating-point representations; these algorithms are subsequently recast into fixed-point representations for implementation in hardware. In this case, it is possible to experience what are known as *quantization errors* or *computational noise* (where the term "quantization" refers to the act of limiting the possible values of some quantity or magnitude into a discrete set of values or quanta.)

When we come to hardware implementations of rounding algorithms, the most common techniques are *truncation* (which is often referred to as *round-toward-zero*, but this would be incorrect in the case of signed binary values as discussed below), *round-half-up* (which is commonly referred to as *round-to-nearest*, but this would be inexact as discussed earlier in this article), and *round-floor* (rounding towards negative infinity).

In the case of unsigned binary representations, *truncation*, *round-half-up*, and *round-floor* work as discussed above. However, there are some interesting nuances when it comes to signed binary representations. Purely for the sake of discussion, let's consider an 8-bit signed binary fixed-point representation comprising four integer bits and four fractional bits *(Fig 1).*



*1. An 8-bit signed binary 4.4 fixed-point representation.*

For the purpose of this portion of our discussions (and for the sake of simplicity), let's assume that we wish to perform our various rounding algorithms so as to be left with only an integer value. Now, let's suppose that our 8-bit field contains a value of 0011.1000, which equates to +3.5 in decimal. In the case of a truncation (or chopping) algorithm, we will simply discard the fractional bits, resulting in an integer value of 0011; this equates to 3 in decimal, which is what we would expect.

Next, assume a value of 1100.1000. The integer portion of this equates to -4, while the fractional portion equates to +0.5, resulting in a total value of -3.5. Thus, when we perform a truncation and discard our fractional bits, we end up with an integer value of 1100, which equates to -4 in decimal. Thus, in the case of a signed binary value, performing a truncation operation actually results in implementing a *round-floor* algorithm.

One of the reasons the *round-half-up* algorithm is popular for hardware implementations is that it doesn't require up to perform a comparison operation. Instead, all we have to do is to add a value of 0.5 and truncate the result. (Note that, when we say "add 0.5," this is based on our earlier assumption that we are rounding to the nearest integer; by comparison, if we were rounding to the nearest half, we would add 0.25 [which is 0.01 in binary] and truncate; and so forth.)

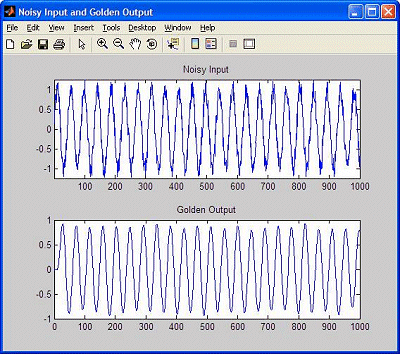
For example, suppose that our 8-bit field contains a value of 0011.0110, which equates to 3.375 in decimal. In this case, if we add a value of 0000.1000 (which equates to 0.5 in decimal), the result is 0011.1110; so when we now truncate this result, we end up with 0011, which equates to 3 in decimal. And this is, of course, what we would expect, because 3.375 should round to 3 in the case of the round-half-up algorithm.

Now remember that, in the case of positive values, we expect the round-half-up algorithm to round to the next integer for fractional values of 0.5 and higher. Suppose we have an initial value of 0011.1000, which equates to 3.5. Adding 0000.1000 and truncating the result leaves us with 0100, or 4 in decimal, which is what we expect. Similarly, when we take an initial value of 0011.1100, which equates to 3.75 in decimal, adding 0000.1000 and truncating the result again leaves us with 0100, which is what we expect.

But what about negative values? For example, consider an initial value of 1100.1000. Once again, the integer portion equates to -4, while the fractional portion equates to +0.5, resulting in a total value of -3.5. In this case, adding 0000.1000 and truncating results in a value of 1101, which equates to -3 in decimal. From this we see that the simple *round-half-up* algorithm favored by many hardware implementations actually results in an *asymmetrical* realization of this rounding function.

**Applying different rounding schemes to a filter design**  
In order to provide some real-world examples, Tim Vanevenhoven at [AccelChip](http://www.accelchip.com/" \t "_new) was kind enough to create and run some test cases in MATLAB to illustrate the types of errors associated with the different rounding schemes applied at various stages throughout a digital filter.

These examples are based on a 32-tap low-pass FIR filter. The screenshot in *Fig 2* shows a noisy [input](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=input&x=&y=) applied to the filer (top) compared to the golden [output](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=output&x=&y=) from a floating-point version of the filter (bottom).



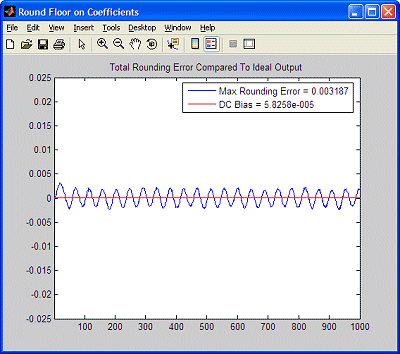
*2. Noisy input versus golden output.*

Next, a fixed-point version of the filter was created with the following characteristics:

* The input has 12 bits, 10 of which are fractional.
* The coefficients have 12 bits, 11 of which are fractional.
* The internal accumulator has 11 bits, 10 of which are fractional.

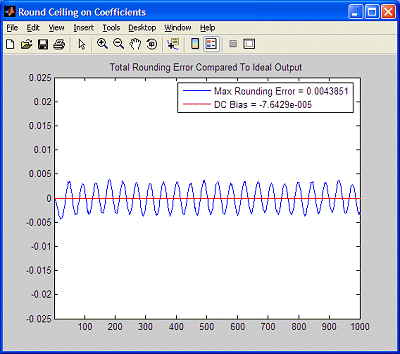
The idea is to apply some of the more common rounding algorithms to different portions of the filter so as to compare the quantization errors and computational noise associated with these schemes. Note that all of the quantization errors are derived from the rounding in this example, because it has been designed such that there is no overflow.

The first test involved performing a *round-floor* algorithm on the filter coefficients (note that a floating-point version of the accumulator was used in this test, so as to isolate any effects to the coefficients themselves). As shown in *Fig 3*, the result was a maximum rounding error of 0.003187 and a [DC](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=DC&x=&y=) bias of +5.8258e-005 as compared to the ideal (golden) output.



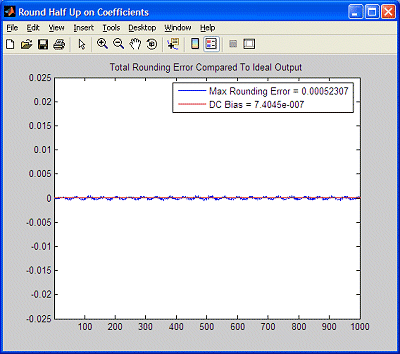
*3. Round-floor on coefficients.*

Next, we applied a *round-ceiling* algorithm to the filter coefficients, which resulted in a maximum rounding error of 0.0043851 and a DC bias of -7.6429e-005 *(Fig 4).*



*4. Round-ceiling on coefficients.*

We then applied a *round-half-up* algorithm to the coefficients; as illustrated in *Fig 5*, this resulted in a maximum rounding error of 0.00052307 (an order of magnitude better than the round-floor and round-ceiling algorithms) and a maximum bias of +7.4045e-007 (a two orders of magnitude improvement).



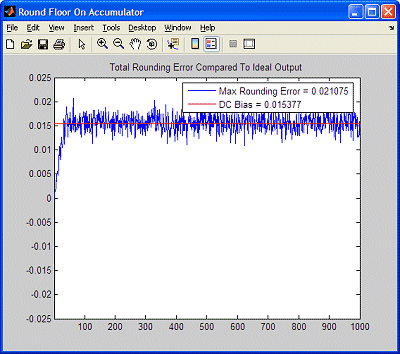
*5. Round-half-up on coefficients.*

Rounding the filter coefficients is an effective way to reduce the overall quantization error in a design. Of particular interest is the fact that – assuming the coefficients to be constant values – this doesn't add any additional hardware to the design. Thus, it would be interesting to experiment with alternative rounding schemes on the coefficients, such as *round-half-even*, *round-alternate*, and even *round-random*.

It's important to note that exactly how much the more sophisticated rounding schemes will improve the filter's performance versus the *round-floor* or *round-ceiling* algorithms is dependent on the actual values of the coefficients and "how far" the floating point values are from the nearest quantization steps. In this particular example, *round-ceiling* introduced more quantization error than *round-floor*; however, the results could be different with different coefficients.

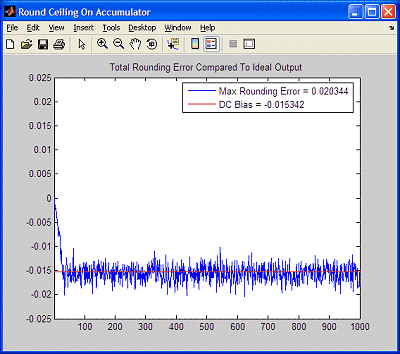
Rounding to the nearest neighbor (using the *round-half-up*algorithm in this example) will usually give the best results, because the absolute value of the difference between the floating-point and fixed-point coefficients will be minimized. In addition, some of the rounding errors will be positive and some will be negative and will – in a sense – tend to cancel each other out. By comparison, using *round-floor* or *round-ceiling* will shift all of the coefficients toward negative or positive infinity, resulting in the positive and negative DC biases reflected in *Figs 3* and *4*, respectively.

Our next experiment was to apply a *round-floor* algorithm to the entire datapath (except for the filter coefficients, which we maintained using the *round-half-up* algorithm from *Fig 5*). The result was a maximum rounding error of 0.021075 and a DC bias of +0.015377 as illustrated in *Fig 6*.



*6. Round-floor on entire datapath (except filter coefficients).*

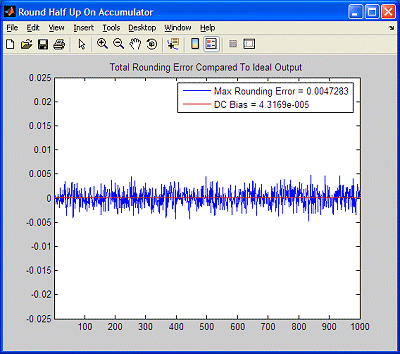
Next, we applied a *round-ceiling* algorithm to the accumulator while maintaining the *round-half-up* on the coefficients and the *round-floor* on the remainder of the datapath. This resulted in a maximum rounding error of 0.020344 and a DC bias of -0.015342 as illustrated in *Fig 7*.



*7. Round-ceiling on accumulator (RF on datapath, RHU on coefficients).*

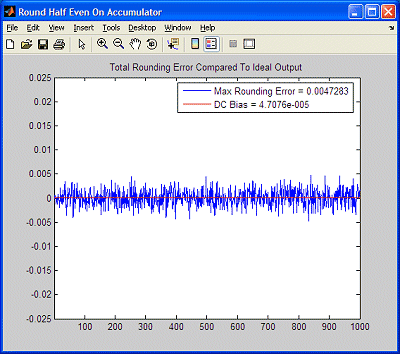
As we see, rounding the accumulator using *round-floor* or *round-ceiling* can – not surprisingly – introduce a DC bias into the output. These schemes also increase the overall error, because the rounding will cause creeping during the accumulation; the larger the number of accumulation iterations, the more the creep will add up.

In order to address this, we first modified the accumulator to use a *round-half-up* algorithm, resulting in a maximum rounding error of 0.0047283 and a DC bias of +0.000043169, as shown in *Fig 8*.



*8. Round-half-up on accumulator (RF on datapath, RHU on coefficients).*

We then changed this to the *round-half-even* algotithm (which MATLAB refers to as a "convergent round"), resulting in a maximum rounding error of 0.0047283 and a DC bias of +0.000047076, as shown in *Fig 9*.



*9. Round-half-even on accumulator (RF on datapath, RHU on coefficients).*

Both the *round-half-up* and round-half-even algorithms provided an order of magnitude improvement in the maximum rounding error as compared to the *round-floor* and *round-ceiling* functions; they both also provided three orders of magnitude improvement in the DC bias.

The interesting point here is that the results from the *round-half-up*and *round-half-even* algorithms were almost identical in this case (this tells us that the data in this example didn't contain many values that fell exactly on "half-way" [0.5] boundaries). However, a hardware implementation of the *round-half-even* would require the additional overhead of a comparison operation (to identify data values that exactly fell on 0.5 boundaries). Thus, the *round-half-up*algorithm, which doesn't require this comparison, gives a much better "bang for the buck" in this example, because the only additional "cost" in hardware is the inclusion of a simple adder.

There are of course many more experiments that one could perform along these lines. For those who are interested, Tim has kindly made the [MATLAB Source Files](https://m.eet.com/media/1061430/max-round-filter-source.zip) for the examples described here available for your delectation and delight. If, you come up with some additional rounding facts and considerations (for these examples or for your own test cases), please feel free to tell me about them, and maybe we'll write another article!

***Clive "Max" Maxfield****is president of*TechBites Interactive*, a marketing consultancy firm specializing in high technology. Max is the author and co-author of a number of books, including*Bebop to the Boolean Boogie*(An Unconventional Guide to Electronics),*The Design Warrior's Guide to FPGAs*(Devices, Tools, and Flows), and – most recently –*How Computers Do Math*(ISBN: 0471732788) featuring the pedagogical and phantasmagorical virtual*DIY Calculator*.*

*Widely regarded as being an expert in all aspects of computing and electronics (at least by his mother), Max was once referred to as "an industry notable" and a "semiconductor design expert" by someone famous who wasn't prompted, coerced, or remunerated in any way. Max can be reached at*[max@techbites.com](mailto:max@techbites.com).

[EMAIL THIS](https://www.eetimes.com/email.asp)

[PRINT](https://www.eetimes.com/document.asp?doc_id=1274485&print=yes)

[COMMENT](https://www.eetimes.com/document.asp?doc_id=1274485&page_number=2#msgs)

[**< PREVIOUS**](https://www.eetimes.com/document.asp?doc_id=1274485&page_number=1)**PAGE 2 / 2**