Letters

Danger in Floating-Point-to-Integer Conversion

I have run across a classic bug in two recently written and widely used audio programs. It occurred to me that if the very smart people who wrote these programs made this mistake, then it might be widespread. I checked Csound and found the bug there too, which I think is a good indication that the bug is in fact common. The bug is a direct result of the default floating-point-to-integer conversion in the C programming language, so it affects many programs and languages implemented in C. I want to describe the problem and some solutions.

Most audio programs convert floating point samples to integer samples and write them to a file or an audio output device. I will assume that "correct" behavior is to round to the nearest integer value. Dealing with scale factors and overflow are also important issues, but there is no standard and the best approach may depend on the application. I will limit my discussion to rounding, which is where this bug occurs.

The natural way to implement the conversion is to scale each floating point sample to some appropriate range (−32767 to 32767) and assign it to a signed 16-bit integer as follows:

```
float f; /* assume −32768 <= f <= 32767 */
int i; /* could also be ``short int'' */
i = (int) f; /* ``(int) f'' means ``convert
 f to an int'' */
```

The default float-to-integer conversion in C does not round to the nearest integer, but instead truncates toward zero. That means that signal values in the open interval (–1.0, 1.0) are all converted to zero (0). This interval is twice as large as the interval mapping to any other integer, and this introduces a nonlinear distortion into the signal.

This is not just an issue of truncation versus rounding. It is well known that rounding to the nearest integer can be achieved by adding 0.5 and rounding down, but the following C assignment is incorrect:

```
i = (int) (f + 0.5);
```

C does not round negative numbers down, so values in the interval (−1.5, 0.5) are converted to zero. In contrast, a correct conversion should map only the interval (−0.5, 0.5) to zero.

There are several ways to perform rounding for audio, and, surprisingly, proper rounding can be faster than the default conversion in C. The direct implementation is to treat positive and negative numbers as different cases:

```
float f; /* assume −32768 <= f <= 32767 */
int i; /* could also be ``short int'' */
if (f > 0) { i = (int) (f + 0.5); }
else { i = (int) (f − 0.5) }
```

This code has the problem of taking a branch, which is very slow relative to arithmetic on modern processors. However, this is a good approach if you can combine the rounding with testing for peak values and clipping out-of-range values, which also treat positive and negative samples separately.

An elegant approach, suggested by Phil Burk, the developer of JSyn and co-developer of PortAudio, is to offset the sample values to make them all positive, perform rounding, and then shift back. Note that I add an extra 0.5 before truncating to simulate rounding behavior:

```
i = (((int) (f + 32768.5)) − 32768)
```

This also produces correct results. These last two algorithms essentially work around the default C conversion semantics, but unfortunately, the conversion itself is slow in most C implementations. Erik de Castro Lopo describes this in detail and offers solutions (see meganerd.com/FPcast/) that avoid using the default conversion altogether, thereby achieving substantially better performance. Intel offers an optimized signal processing library (developer.intel.com/software/products/perflib/spl/index.htm) that includes fast rounding and conversion functions. Finally, there is an interesting conversion method described on page 91 of Dannenberg and Thompson, "Real-Time Software Synthesis on Superscalar Architectures" (*CMJ* 21:3), although this is reportedly not the best method on an ix86 processor.

And now, if you will excuse a brief plug, it amazes me that after decades of software development in computer music, our tiny community tries to support so many different implementations of basic functions for music processing. Surely if we could share a common, portable code base, problems like rounding errors would be less common and solutions could be more readily shared. To this end, I invite all interested readers to join a discussion at www.create.ucsb.edu/mailman/listinfo/media_api and to join in the PortMusic effort (www.cs.cmu.edu/~music/portmusic).

Roger B. Dannenberg
Carnegie Mellon University
rbd@cs.cmu.edu