# FastAC: Common Questions

**Q.** *Can I trust these coding programs for using them in my thesis, application, etc.?*
**A.** The arithmetic coding implementations are provided together with a program for testing and benchmarking to create confidence on the code. They had been tested on billions of simulations with no decoding error. However, as with any software, it is possible that some small detail was forgotten when we copied and simplified the programs, and it is theoretically possible that an error may occur under *extremely rare conditions*. The integer-based versions should be correct always. The floating-point version is meant for educational purposes only, so we didn't spend much time looking for possible incorrect decoding under all possible extreme conditions (the analysis of floating point arithmetic is considerably more complicated.)

**Q.** *Why four different versions? Why so much code is repeated in all the versions?*
**A.** We have the objective of providing useful implementations that are also clear enough to be didactic. The floating-point version is obviously a special case. The different integer-based versions are meant to show a few of the many possible implementation choices and trade-offs. However, for practical applications the user should choose one, and it is better if it is in only two files. There is a lot of common code, but we do not like the idea of spreading the code in many files, or making the programs harder to understand due to many options for conditional compilation, or creating a complicated interface with many classes.

**Q.** *Is the C++ source code portable across different computer platforms? Why C++?*
**A.** Except for the version that uses a few lines of Intel x86 assembler code to retrieve the 64-bit multiplication result, all versions should compile and run in any OS & computer. The advantages of C++ over C are now very well established, and we believe our application fits the object-oriented approach very well. (Even development tools for embedded systems now support C++.)

**Q.** *Is the compressed data stream portable across different computer platforms?*
**A.** There is no byte-order or similar problems with the code: everything that is computed using integers (including the 64-bit multiplication results) should be 100% portable. However, what is computed using floating point numbers may not be, since different types of FPU may yield slightly different results, and even tiny differences are magnified by the renormalization until they result in incorrect decoding. This includes the whole version that uses floating point arithmetic, and may also include the initialization code for the fixed model implementations. (This problem can be solved by having probabilities defined using integers—easy, but not very intuitive).

**Q.** *Why the programs have limits on probabilities?*
**A.** All arithmetic coding implementations have strict limits on the smallest probability they can support. The data models in the versions using 32-bit products have a minimum value equal to $2^{-15}$. In some special cases that need more precision, it can be extended to $2^{-18}$ without losing much in compression. The version using 64-bit products has a minimum value equal to $2^{-24}$ (definitely beyond practical needs), and the floating-point version will work with even

smaller values. However, the program rejects values that are not as small as the absolute minimum because we assumed that symbols with probability $2^{-20}$ are not exactly what you expect in real coding applications, and it is more likely that such numbers represents a user mistake.

**Q.** *So, it is possible that the programs refuse some parameters, even though thy will not cause malfunction or incorrect decoding?*
**A.** Yes. When the programs identify cases that are too unusual, and also some that would result in correct, but less efficient compression (unusual too), they abort execution and put out an error message. The objective of the program is to provide convenience when coding, which means good detection of common mistakes, and not necessarily support for extreme cases.

**Q.** *Why the demo program crashes after I change the version of the arithmetic codec?*
**A.** The file and class names are all the same, so the compiler may not know it is a new, different version. You need to recompile the whole project ("rebuild all").

**Q.** *My program is not decoding correctly. What should I do?*
**A.** First try to compile in debug mode (`_DEBUG = 1`). This will enable a few more checks for common mistakes. Second, consider if there is no corruption to the compressed data bytes. Next, replace the current version with another (e.g., replace the integer with the floating-point version): if the problem remains it should not be in the coding functions. Consider that if your program is not stressing the codecs beyond what is done in the simulations, it is very improbable that the problem is actually in the codec. If the problem occurs in only one of the versions/modes, and you are absolutely convinced it should be a problem in the codec, then let us know.

**Q.** *Why am I getting coding times that are different from the published results?*
**A.** First, be sure you compile the program using all the optimizations for fastest execution ("release" mode for MS VC++ with inline enabled, options "-oX" for gcc, etc.).

**Q.** *Why am I still getting coding times that are different from the published results?*
**A.** Maybe you discovered the effects of the Mysterious Deliberations of the Mystic Compiler! You can even find that the code with more instructions runs faster! This happens because processors are beginning to have more pipelines that an oil refinery, and they try to have all results computed even before you know you need them. We have the fastest code when all the planets, I mean, instructions, are properly aligned. Unfortunately, even if the compilers are aware of these facts, the executable optimization decisions depend on the compiler's whim. For instance, they panic when they find assembler code (as in our 64-bit version), and consequently may disable all optimizations for days. Before getting the results for publication we tried to eliminate these problems, but we didn't want to make the public code unreadable due to the tricks needed to deal with this problem.

**Q.** *How good is the compression of the demo programs?*
**A.** They are OK. Surely not state-of-the-art, but they may be good enough for practical use! Their limitations are in modeling and signal processing, and not in the coding process. All our arithmetic coding implementations yield practically optimal compression.