Fast Arithmetic Coding (FastAC) Implementations

Amir Said

1 Introduction

This document describes our fast implementations of arithmetic coding, which achieve optimal compression and higher throughput by better exploiting the great numerical capabilities of the current processors. References [1, 2] contain descriptions of the particularities of the coding methods used, and reference [3] presents the experiments and speed measurements used for optimizing our code.

Objectives

During the development of our program we had the following objectives:

1. Its interface must be simple enough so that it can used by programmers without compression expertise.
2. Its performance should represent the state-of-the-art of arithmetic coding in terms of speed, compression efficiency, and features.
3. It must be versatile, portable, and reliable enough to be used in more advanced coding projects.
4. The program should be clear and simple enough so that students in a first course on coding can learn about arithmetic coding, and truly understand how it works.

While we can achieve all the practical objectives (1–3) with a single version, we believe that for educational purposes it is better to provide more than one example, so we finally decided to have four versions. They are all interchangeable, and each, from the simplest to the most optimized, can provide some more insight in the arithmetic coding process.

The user only interested in using good arithmetic coding functions inside an application can employ (and possibly modify) only the first, which provides the best combination of efficiency and portability.

Document organization

This document is divided in the following parts. Section 2 presents the rationale for using object-oriented programming in compression applications, and describes our C++ classes implementing data-source models, and the arithmetic encoder and decoder. Next, in Section 3, we provide more details on how to use our programs by using them on a simple compression problem. The functions implementing an encoder and decoder are presented and we comment on the programming choices, how the coding functions work, and how the C++ classes are used. In Section 4 we explain the differences between the four arithmetic coding implementations, and at the end, in Section 5, we explain the purpose of three programs that we use to demonstrate, on real compression applications, how to use and test our implementations.

Program distribution

All the programs are in a single zip file called `FastAC.zip`, which contains 7 subdirectories. Under the subdirectory `AC_Versions` there are 4 subdirectories with each of the implementation files, which are all called `arithmetic_codec.h` and `arithmetic_codec.cpp` (see Section 4). The documentation files, `FastAC_Readme.pdf` (this file), and `FastAC_QA.pdf` (common questions), are in the root directory.

The root directory also contains files that define a MS VC++ workspace (`FastAC.dsw`, `FastAC.ncb`, and `FastAC.opt`) with 3 projects, called `acfile`, `acwav`, and `test`. Each project subdirectory contains a different program as an example of how to use our code (see Section 5). The desired implementation files (`arithmetic_codec.h` and `arithmetic_codec.cpp`) should be copied to the root directory before compilation, and it is necessary to recompile the whole project ("rebuild all").

2 Interface of the Arithmetic Coding C++ Classes

Because there are innumerable types of data that need to be compressed, but only a few coding methods, there are many advantages in separating the processes of *source modeling* from *entropy coding* [1, 2]. While in practice it is not possible to completely separate the two, with the current object-oriented programming (OOP) tools it is possible to write programs that eliminate the need to understand the details of how the data-source models are actually implementations and used for coding.

For example, for coding purposes the only information needed for modeling a data source is its number of data symbols, and the probability of each symbol. Thus, for OOP purposes, that is the only type of data abstraction needed. It is true that during the actual coding process what is used is data that is computed from the probabilities, i.e., the optimal codewords or coding functions, but we can use OOP to encapsulate that data and those functions in a manner that is completely transparent to the user.

Model classes

We provide four C++ classes for modeling data sources, with definitions in the files called `arithmetic_codec.h`, and implementations in the files called `arithmetic_codec.cpp`. Note that there are four versions of these files, each corresponding to a different implementation of arithmetic coding, and in a separate subdirectory. The differences between them are explained in Section 4. We use the OOP data hiding principle in this document too. The public class information in all the files is the same, and is shown and explained here. The private data and functions—different for each file—are not shown because they do not affect their use.

All the class definitions are shown in Figure 2. The first class, called `Static_Bit_Model`, is meant for sources with only two symbols (0 and 1). The only information needed for modeling is the probability of the symbol $s = 0$, which is defined using the function `set_probability_0`. The second class, called `Static_Data_Model` can be used for sources with a number of symbols between 2 and 2^{11} symbols or between 2 and 2^{14} symbols, depending on the precision of the implementation. The function `set_distribution` completely defines the source by setting the number of data symbols and their probabilities. If this function receives a pointer to probabilities equal to zero it assumes that all symbols are equally probable. If the data source has M symbols, they must be integers in the range $[0, M - 1]$.

Even though these models are called *static*, the symbol probabilities of these classes can be changed any time while coding, as long as the decoder uses the same sequence of probabilities. Their main limitation is that they only use the probabilities provided by the user. In most situations we do not know *a priori* the probabilities and we have to use estimates. Since the coding process can be quite complex, we want to avoid using two passes, one only for gathering statistics, and another for coding.

The solution is provided by *adaptive* models, which simultaneously code and update probability estimates. Figure 2 shows the definition of class `Adaptive_Bit_Model` for binary sources. It starts assuming that both symbols are equally probable, and updates the probability estimates as the symbols are coded. Its only public function is `reset`, which restarts the estimation process. The last class, `Adaptive_Data_Model`, is for sources with a larger number of symbols. Before it is used for coding it needs to know the number of data symbols, which is defined using the function `set_alphabet`. It also starts assuming that all symbols have equal probability, and has a function called `reset` to restart the estimation process.

We have two types of classes because there are special optimizations that work only for binary models. In consequence, coding with binary models is typically 50% faster than using the general model with the number of symbols set to two.

Encoder and decoder (codec) class

For simplicity we use a single class, called `Arithmetic_Codec`, to encapsulate both the encoder and the decoder functions. Its definition is shown in Figure 2. We use a framework in which all compressed data is saved to and fetched from a memory buffer, and only periodically written to or read from a file. Thus, before we can start encoding and decoding, it is necessary to define the memory location where we want the compressed data to be stored.

```
class Static_Bit_Model
{
public:
   Static_Bit_Model(void);
   void set_probability_0(double);             // set probability of symbol 0
};

class Static_Data_Model
{
public:
   Static_Data_Model(void);
  ~Static_Data_Model(void);
   void set_distribution(unsigned number_of_symbols,
                         const double probability[] = 0); // 0 means uniform
   unsigned model_symbols(void) { return data_symbols; }
};

class Adaptive_Bit_Model
{
public:
   Adaptive_Bit_Model(void);
   void reset(void);                            // restart estimation process
};

class Adaptive_Data_Model
{
public:
   Adaptive_Data_Model(void);
  ~Adaptive_Data_Model(void);
   Adaptive_Data_Model(unsigned number_of_symbols);
   void set_alphabet(unsigned number_of_symbols);
   void reset(void);                            // restart estimation process
   unsigned model_symbols(void) { return data_symbols; }
};
```

Figure 1: Definition of classes for supporting static and adaptive data-source models.

```
class Arithmetic_Codec
{
public:
   Arithmetic_Codec(void);
  ~Arithmetic_Codec(void);
   Arithmetic_Codec(unsigned max_code_bytes,
                    unsigned char * user_buffer = 0);       // 0 = assign new

   unsigned char * buffer(void)  return code_buffer;

   void set_buffer(unsigned max_code_bytes,
                   unsigned char * user_buffer = 0);        // 0 = assign new

   void start_encoder(void);
   void start_decoder(void);
   void read_from_file(FILE * code_file);    // read code data, start decoder

   unsigned stop_encoder(void);              // returns number of bytes used
   unsigned write_to_file(FILE * code_file); // stop encoder, write code data
   void     stop_decoder(void);

   void     put_bit(unsigned bit);
   unsigned get_bit(void);

   void     put_bits(unsigned data, unsigned number_of_bits);
   unsigned get_bits(unsigned number_of_bits);

   void     encode(unsigned bit,
                   Static_Bit_Model &);
   unsigned decode(Static_Bit_Model &);

   void     encode(unsigned data,
                   Static_Data_Model &);
   unsigned decode(Static_Data_Model &);

   void     encode(unsigned bit,
                   Adaptive_Bit_Model &);
   unsigned decode(Adaptive_Bit_Model &);

   void     encode(unsigned data,
                   Adaptive_Data_Model &);
   unsigned decode(Adaptive_Data_Model &);
};
```

Figure 2: Definition of the class for arithmetic encoding and decoding.

We must use a constructor or the function `set_buffer` to define an amount of memory equal to `max_code_bytes` in which the compressed data can be written or read. The function `buffer` returns a pointer to the first byte in this buffer. Note that we said *define*, not allocate. This depends on the parameter `user_buffer`. If it is zero then the class `Arithmetic_Codec` will allocate the indicated amount of memory, and later its destructor will free it. Otherwise, it will use the `user_buffer` pointer as the first position of the compressed data memory, assuming that the user is responsible for its allocation and for freeing it.

Before encoding data it is necessary call the function `start_encoder` to initialize and set the codec to an encoder mode. Next, data is coded using the version of the function `encode` corresponding to its data model. It is also possible to write an integer number of bits to the compressed stream using the functions `put_bit` or `put_bits`. If the data source has M symbols, the data must be unsigned integers in the range $[0, M-1]$. Similarly, the data with b bits should be in the range $[0, 2^b - 1]$.

Before the compressed data can be copied from its memory buffer it is necessary to call the function `stop_encoder`, which saves the final bytes required for correct decoding, and returns the total number of bytes used. Alternatively, the function `write_to_file` can be used to simultaneously stop the encoder and save the compressed data to a file. This function also writes a small header so that the decoder knows how many bytes it needs to read.

The decoding process is quite similar. Before it starts all the compressed data must to be copied to the memory buffer, and the function `start_decoder` must be called. If the compressed data had been saved to a file using the function `write_to_file`, then the function `read_from_file` must be used to read the data and start the decoder. The compressed data is retrieved using the corresponding `decode`, `get_bit`, or `get_bits` functions. When decoding is done, the function `stop_decoder` must be called to restore the codec to a default mode, so that it can be restarted to encode or decode.

3 Coding Example

Let us consider the problem of compressing the data used for line plots.[1] Each plot is defined by an array of structures of the type `Plot_Segment`, shown in the top of Figure 3. The first component, `line_color`, indicates the index of the color of the line, which is a number between 0 and 7. The meaning of the second and third component depends on the first. If `line_color = 0` then it contains the absolute (`x,y`) coordinate to where the plotter pen should move ("move to" instruction). If `line_color > 0` then it contains the (`x,y`) displacement from the current location, using a pen with the indicated color ("line to" instruction). We assume that `x` and `y` are in the interval $[-2^{15}, 2^{15} - 1]$.

For designing a new compression method we should consider that, even though in theory we can use a large number of adaptive models to form complex high-order models for any type of data, in practice it is better to consider if we can exploit the particularities of typical data. This happens because adaptive methods need time to get good estimates, which is not a problem if the number of models is reasonably small, but becomes a problem when we have, for example, many thousands of models.

[1] Surely obsolete, but it makes an interesting example.

For our plot-segment compression problem we can exploit the following facts:

- Consecutive line segments normally have the same color.
- New colors are expected after a "move to."
- Curves are plotted with many short segments.

One way to make use of the fact that something does not change very frequently is to first use binary information to indicate when changes occur, and only then code the new information. Figure 3 show an example of a coding function that uses this technique to code an array of structures of the type `Plot_Segment`.

Following the sequence of instructions in Figure 3, we see that we start defining one variable of the type `Arithmetic_Codec`. We assume that it will compress to a memory buffer that is provided from outside the function `Encode_Plot`, and use the proper constructor to set this buffer. Next we define four adaptive models. The first, `color_change_model`, is binary, and is used for coding color change information. The second, `color_model`, is for coding the actual color information, and thus we use the constructor informing that its number of data symbols is equal to 8. The third model, `short_line_model`, is binary, and is used to indicate when displacements are small. The fourth model is for coding short lines.

After the call to the function `start_encoder` we have the loop for coding all the plot segments. The information coded depends on the previous and current colors. If in the previous segment we had `line_color = 0` then we code the color information immediately, otherwise we first code the change information, and, only if there is color change we code the new color.

Next, we need to code the position or displacement (`x,y`). Since their range is quite large, we do not try to entropy code them all the time. If (`x,y`) represents absolute position (`line_color = 0`), or a large displacement, then we just save `x` and `y` as 16-bit nonnegative numbers. If the displacement magnitude is smaller than 128 then we first code the information that the segment is short, followed by `x` and `y`, which are converted to nonnegative number and coded using `step_model`.

Figure 3 shows the function `Decode_Plot` to decompress the data created by `Encode_Plot`. Note that it is very similar to the encoder, since it must reproduce all the sequences of decisions taken by the encoder. Thus, even though we have four types of information that can be coded, we have correct decoding because the sequence of models used by the decoder is identical to encoder's sequence.

4 Arithmetic Coding Versions

As explained in the introduction, for educational purposes we have written four different fully functioning implementations of arithmetic coding. In this section we present the main features of each implementation. For practical use we recommend Version 1, which is 100% portable and quite efficient. For those that want to learn how arithmetic coding works,

```
struct Plot_Segment
{
   int line_color, x, y;
};

int Encode_Plot(int plot_points,
                Plot_Segment seg[],
                int buffer_size,
                unsigned char * compressed_data)
{
   Arithmetic_Codec    ace(buffer_size, compressed_data);
   Adaptive_Bit_Model  color_change_model;
   Adaptive_Data_Model color_model(8);
   Adaptive_Bit_Model  short_line_model;
   Adaptive_Data_Model step_model(257);

   ace.start_encoder();
   int short_line, last_color, current_color = 1;

   for (int p = 0; p < plot_points; p++) {

       last_color = current_color;
       current_color = seg[p].line_color;

       if (last_color != 0)
          ace.encode(last_color != current_color, color_change_model);
       if ((last_color == 0) || (last_color != current_color))
          ace.encode(current_color, color_model);

       if (current_color == 0)
          short_line = 0;
       else {
          short_line = (abs(seg[p].x) <= 128) && (abs(seg[p].y) <= 128);
          ace.encode(short_line, short_line_model);
       }

       if (short_line) {
          ace.encode(seg[p].x + 128, step_model);
          ace.encode(seg[p].y + 128, step_model);
       }
       else {
          ace.put_bits(seg[p].x + 32768, 16);
          ace.put_bits(seg[p].y + 32768, 16);
       }
   }
   return ace.stop_encoder();   // return number of bytes used for compression
}
```

Figure 3: Definition of structure `Plot_Segment` for storing plot graphic data, and implementation of function `Encode_Plot`.

```
void Decode_Plot(int plot_points,
                 int data_bytes,
                 unsigned char * compressed_data,
                 Plot_Segment seg[])
{
   Arithmetic_Codec    acd(data_bytes, compressed_data);
   Adaptive_Bit_Model  color_change_model;
   Adaptive_Data_Model color_model(8);
   Adaptive_Bit_Model  short_line_model;
   Adaptive_Data_Model step_model(257);

   acd.start_decoder();
   int short_line, current_color = 1;

   for (int p = 0; p < plot_points; p++) {

       if (current_color == 0)
          current_color = acd.decode(color_model);
       else
          if (acd.decode(color_change_model))
             current_color = acd.decode(color_model);

       seg[p].line_color = current_color;

       if (current_color == 0)
          short_line = 0;
       else {
          short_line = acd.decode(short_line_model);

       if (short_line) {
          seg[p].x = int(acd.decode(step_model)) - 128;
          seg[p].y = int(acd.decode(step_model)) - 128;
       }
       else {
          seg[p].x = int(acd.get_bits(16)) - 32768;
          seg[p].y = int(acd.get_bits(16)) - 32768;
       }
   }
   acd.stop_decoder();
}
```

Figure 4: Implementation of function `Decode_Plot`.

inspect the code details, and possibly change the programs, we suggest following the inverse version order, i.e., start from Version 4, followed by 3, 2, and 1.

Version 1: 32-bit variables, 32-bit products

This version, which is in the directory `int_32_32`, is the most portable and usually the fastest, integrating all the main acceleration techniques. It is a good example of how an implementation can be both efficient and simple. It has some significant differences compared to the straightforward floating-point version, but it should not be difficult to understand how it works.

It uses 32-bit arithmetic all the time, which is exploited to have compression very near the optimal. Before multiplications it discards the least-significant bits to avoid overflow. Thus, the total number of bits of precision assigned to the interval length and probability must not exceed 32. In the current version the precision depends on the model. For binary models we have 19 bits for the length and 13 bits for probabilities, while for the other models, we have 17 bits for the length and 15 bits for probabilities.

This version saves full bytes during renormalization, and has two forms of decoding. The first, used when the number of symbols is large, is based on fast table look-up and needs divisions (which can be slow on some processors). The second form replaces the division with several multiplications, and is used when the number of symbols is small.

Version 2: 32-bit variables, 32-bit products, sorted symbols

The version in directory `int_32_32_sorted` is very similar to the previous versions, but it sorts the symbols according to probability to optimize some operations. If the source distribution is highly skewed, then this can be the fastest version, using only multiplications (no divisions). In other cases it is better to use the version with table look-up decoding. The static binary model in this version is also different: we provide an implementation that shows that for binary coders it is relatively easy to approximate multiplications with bit shifts (the speed improvement, however, is not very significant).

Version 3: 32-bit variables, 64-bit products

Many 32-bit processors (e.g., Pentium) compute the full 64 bits of a multiplication of two 32-bit integers. The problem is that the compilers do not give access to the register with the most significant bits. The version in directory `int_32_64` shows how to use a few lines of assembler code to multiply and extract those bits, and also for a 64-bit division used by table look-up decoding. It can be a bit slower, but mostly because the compilers don't know how to use assembler and optimize at the same time. On the other hand, it has a precision that guarantees virtually optimal compression even in extreme cases.

Version 4: floating-point arithmetic

This version, in directory `floating-point`, was created because we believe floating-point numbers provide a more intuitive way of understanding arithmetic coding. Students can follow the program's execution, and can see all the values in a scale that shows the true value of probabilities. The implementation is straightforward, without much effort to optimize speed (but it is not significantly worse than the others).

The implementation uses 48 bits of double precision floating-point numbers, and renormalization occurs when at least 16 bits are ready. Two extra tricks had to be used: a small offset is added to the code value while decoding (to deal with the hard-to-predict behavior of the least significant bits beyond the first 48), and some extra "leakage" for each coded symbol (to compensate for possible rounding and other factors).

5 Arithmetic Coding Demo Programs

The three programs with applications are in the files called `acfile.cpp`, `acwav.cpp`, and `test.cpp`, in the subdirectories with the same name. Here is their description.

`acfile.cpp`

This is an example of how to use arithmetic coding to compress any file (text, executables, etc.) using a relatively small number of adaptive models. Probably the most important lesson to be learned from this program is that our code is very easy to use, and enables writing a reasonably good compression program with small effort.

The program's usage for compressing and decompressing a file is, respectively

```
acfile -c file_name compressed_data_file
acfile -d compressed_data_file new_file
```

`acwav.cpp`

This is a slightly more complex example. It is for *lossless* compression of audio files. The audio file format supported is "wav", which is supported by all CD "ripping" programs. It uses some signal processing to improve compression (the reversible S+P transform), but the coding process is very similar to the first case, except that here we decompose each transform sample in a "bits+data" representation (same as VLI in JPEG & MPEG standards). The former is coded with contexts and adaptive models, while the bits of the latter are save directly.

The program's usage is similar. For compression and decompression use

```
acwav -c wav_file compressed_wav_file
acwav -d compressed_wav_file new_wav_file
```

test.cpp

This program is not really an application. It has functions to test and benchmark the speed of our arithmetic coding implementations (it is similar to the program used in [3]). Given a number of data symbols, it defines several values of source entropy, and for each value it generates millions of pseudo-random source samples. The times to encode and decode this data are measured, and it finally compares the decoded with the original to make sure the code is correct.

The usage is

```
test number_of_symbols
test number_of_symbols number_of_simulation_cycles
```

The first uses a default number of cycles equal to 10.

In this directory we also have the files `test_support.h` and `test_support.cpp`, which contain some functions required for the simulations, like the pseudo-random number generator.

References

[1] Amir Said, "Arithmetic Coding," in *Lossless Compression Handbook,* (K. Sayood, Ed.), Academic Press, San Diego, CA, 2003.

[2] Amir Said, *Introduction to Arithmetic Coding Theory and Practice,* Hewlett-Packard Laboratories Report, HPL–2004–76, Palo Alto, CA, April 2004 (http://www.hpl.hp.com/techreports/).

[3] Amir Said, *Comparative Analysis of Arithmetic Coding Computational Complexity,* Hewlett-Packard Laboratories Report, HPL–2004–75, Palo Alto, CA, April 2004 (http://www.hpl.hp.com/techreports/).