A Fast Renormalisation Me

Method for Arithmetic Co

Coding

Michael Schindler

Remark:

Since submission of the abstract page to print an old article by G. N. N. Martin submitted to the Data Recording Conference, Southampton in March 1979 has been brought to my notice. That article presents range encoding, which can be seen as arithmetic coding with renormalisation in larger units like it is presented here.

However this poster still provides interesting comparison with current implementations and may help to spread the knowledge, so I decided to present it.

I also added some profiling data to show that the actual entropy coding no longer is the bottleneck.

Proposed here is a renormalisation method for arithmetic coding. Unlike standard methods like the one given by A. Moffat et alii on DCC95 this one does not require that the first set bit of the range is on a fixed position.

This allows to delay renormalisation until an entire byte can be processed at once, thus getting rid of overhead inside the renormalisation loop and any bitwise input/output.

The drawback is that the shift+add method for division will not work. Furthermore the guaranteed size of range is smaller, reducing the values allowable for the total frequency count or increasing coding loss.

The code presented here was written to match the verbal description, especially for carry detection there are better options, such as accumulating the sum not in the low variable directly.

At any point during arithmetic coding the output consists of four parts.

1. There is the part already written, which can't be changed any more.
2. There is one digit (bit, byte, whatever) that may be changed by at most one carry when adding to the lower end of the interval. There will never be two carries since the range when fixing that digit was $\leq$ 1unit. Two carries would require a range > 1 unit.
3. There is a (possibly empty) block of digits that pass on a carry (1 in binary, 9 in decimal, 255 for bytes, …) that are represented by a counter counting their number.
4. The last part is represented by the low variable (out_L) of the encoder.

When doing encoding renormalisation the following can happen:

- No renormalisation is needed since the range is in the desired interval.
- The low end plus the range (this is the upper end of the interval) will not produce any carry. In this case the second and third part can be output, they will never change. The digit produced will become part two, and part three will be empty.
- The low end has already produced a carry. Here the (changed) second and third part can be output; there will not be another carry. Set the second and third part as before.
- The digit produced will pass on a possible future carry, so it is added to the third block.

The difference from standard (DCC95) method is that part two, which may be changed by a carry, has to be stored explicitly. With binary output this part is always 0 since the ones are always added to the carry passing block. The advantage over the standard method is that the renormalisation is less often needed, and if it is needed it is mostly the no carry possible branch.

The difference from Martin's Range encoding is that the delayed digits are resolved only if a renormalisation is needed, thus simplifying the method.

Implementation is straight forward, the test code that produced the tables is available at http://www.compressconsult.com/rangecoder/

Statistics for world192.txt out of Canterbury Corpus (2473400 bytes), profiling data on a Pentium 166 using HiProf, program counter sampling.

Compression	Old renormalisation		New renormalisation	
Reading	0.38	7.8%	0.69	20.1%
Writing	0.32	6.6%	0.38	10.9%
Arithm. Coding	2.69	55.0%	1.02	29.5%
Model query	0.19	3.9%	0.19	5.3%
Model update	0.48	9.8%	0.35	10.1%
Other (main loop, open)	0.83	16.9%	0.84	24.1%
Total time	4.89 sec	30.3MB/min	3.47 sec	42.8MB/min
Filesize	1568063 byte		1568078 byte	

Decompression	Old renormalisation		New renormalisation	
Reading	0.40	6.1%	0.27	5.4%
Writing	0.65	10.0%	0.59	11.6%
Arithm. Decoding	3.07	46.7%	1.18	23.5%
Arithm. Update	0.65	10.0%	0.58	11.5%
Model search + query	0.77	11.7%	0.72	14.4%
Model update	0.45	6.8%	0.40	7.9%
Other (main loop, open)	0.57	8.7%	1.31	25.7%
Total time	6.56 sec	22.6MB/min	5.05 sec	29.4MB/min

The reading, writing and other time vary significantly with each run. They depend on other processes, load and physical position of data. The coder and model values are repeatable within small areas.

The following table was produced using linux on the same PC (166Mhz Pentium) as before, taking user+system time for the same world192.txt file.

Mbyte/minute	Old renormalisation	New renormalisation
Compress	36.9	67.5
Decompress	28.4	40.7

The statistics model used for all experiments is a simple order 0 model. It updates only periodically and keeps the total frequency count a power of two. This was done to speed up the model; otherwise slow model operation would outweight all other effects. The faster operation of the coder here compared to the one presented on this conference by Lang Stuiver and Alistair Moffat is mostly due to this faster model – it even compensates the performance difference between a Pentium Pro233 and a Pentium 166.

Standard renormalisation

```
/* Code based on A. Moffat et al. DCC95 article code */
#define         Half            ((code_value)1 << (CODE_BITS-1))
#define         Quarter         (Half>>1)

#define OUTPUT_BIT(aco,b)                   \
do {if (!(aco)->bitsleft--)                 \
    {   outbyte((aco),(aco)->buffer);       \
        (aco)->bitsleft = 7;                \
    }                                       \
    (aco)->buffer = (aco)->buffer<<1 | b;\
} while (0)


#define INPUT_BIT(aco,target)               \
do {if (!(aco)->bitsleft)                   \
    {   (aco)->buffer = inbyte(aco);        \
        (aco)->bitsleft = 8;                \
    }                                       \
    (aco)->bitsleft--;                  \
    (target) = (target)<<1 | (((ac)->buffer>>(ac)->bitsleft)&1);\
} while (0)


/* BIT_PLUS_FOLLOW(b)
 * responsible for outputting the bit passed to it and an opposite number
 * of bit equal to the value stored in bits_outstanding */
#define BIT_PLUS_FOLLOW(aco,b)              \
do {   OUTPUT_BIT((aco),(b));               \
    while ((aco)->bytestofollow > 0)        \
    {   OUTPUT_BIT((aco),!(b));             \
        (aco)->bytestofollow--;             \
    }                                       \
} while (0)


/* ENCODE_RENORMALISE
 * output code bits until the range has been expanded to above QUARTER */
static Inline void enc_normalize( arithcoder *ac )
{   while (ac->range <= Quarter)
    {   if (ac->low >= Half)
        {   BIT_PLUS_FOLLOW(ac,1);
            ac->low -= Half;
        }
        else if (ac->low+ac->range <= Half)
        {   BIT_PLUS_FOLLOW(ac,0);
        }
        else
        {   ac->bytestofollow++;
            ac->low -= Quarter;
        }
        ac->low <<= 1;
        ac->range <<= 1;
    }
}
```

```
void done_encoding( arithcoder *ac )
{   int i,b;
    enc_normalize( ac );
    for (i = CODE_BITS-1; i>0; i--)
    {   b = (ac->low>>i);
        BIT_PLUS_FOLLOW(ac,b);
    }
    b = ac->low & 1;
    BIT_PLUS_FOLLOW(ac,b);
    outbyte(ac,ac->buffer);
}

/* DECODE_RENORMALISE
 * input code bits until range has been expanded to
 * more than QUARTER. Mimics encoder. */
static Inline void dec_normalize( arithcoder *ac )
{   while (ac->range <= Quarter)
    {   ac->range <<= 1;
        INPUT_BIT(ac,ac->low);
    }
}
```

New renormalisation

```
#define NOWARN
#define Top_value ((code_value)1 << CODE_BITS)
#define Bottom_value (Top_value >> 8)

static Inline void enc_normalize( arithcoder *ac )
{   while(ac->range <= Bottom_value)
    {   if (ac->low < 0xff<<SHIFT_BITS) /* no carry possible --> output*/
        {   outbyte(ac,ac->buffer);
            for(; ac->bytestofollow; ac->bytestofollow--)
                outbyte(ac,0xff);
            ac->buffer = (unsigned char)(ac->low >> SHIFT_BITS);
        } else if (ac->low >= Top_value) /* carry now, no future carry */
        {   outbyte(ac,ac->buffer+1);
            for(; ac->bytestofollow; ac->bytestofollow--)
                outbyte(ac,0);
            ac->buffer = (unsigned char)(ac->low >> SHIFT_BITS);
        } else
#ifdef NOWARN
            ac->bytestofollow++;
#else
            if (ac->bytestofollow++ == 0xffffffffL)
            {    fprintf(stderr,"Too many bytes outstanding\n");
                 exit(1);
            }
#endif
        ac->range <<= 8;
        ac->low = (ac->low << 8) & (Top_value-1);
    }
}
```

```
/* Finish encoding
 * actually not that many bytes need to be output, but who cares
 * I output them because decode will read them */
void done_encoding( arithcoder *ac )
{   unsigned char carry;
    int bitsleft;
        ac->low += ac->range >> 1;       /* select middle of interval */
        carry = ac->low >> CODE_BITS;
        outbyte(ac,ac->buffer+carry);    /* flush outstanding bytes   */
        carry = carry? 0 : 0xff;
        for (; ac->bytestofollow; ac->bytestofollow--)
                outbyte(ac,carry);
        /* output low */
        for (bitsleft = SHIFT_BITS; bitsleft>0; bitsleft-=8)
                outbyte(ac,(unsigned char)(ac->low >> bitsleft));
        outbyte(ac,(unsigned char)(ac->low << -bitsleft));
}


static Inline void dec_normalize( arithcoder *ac )
{   while (ac->range <=Bottom_value)
    {   ac->low = (ac->low<<8) | ((ac->buffer<<EXTRA_BITS)&0xff);
        ac->buffer = inbyte(ac);
        ac->low |= ac->buffer >> (8-EXTRA_BITS);
        ac->range <<= 8;
    }
}
```