

# CS7491 3D Complexity Lecture Notes

Lecture by Jarek Rossignac  
Scribed by Brandon Beck and Jaeil Choi  
College of Computing, Georgia Tech

Jan 20, 2004

## 1 Arithmetic encoder - revisited

### Encoding one symbol using Arithmetic encoder

1. Split the interval  $[0, 1]$  into  $n$  segments which corresponds to the symbols  $s_i$  ( $i = 1, 2, \dots, n$ ), with size of the probability  $p_i$ .  $0 < p_i < 1$ ,  $\sum p_i = 1$
2. Use a string of bits to reduce the interval by half (binary subdivision).
3. Normalize the whole interval, so that we have current interval fitted to  $[0,1]$ .
4. If current interval falls completely inside a slot of any symbol, we know that this string of bits represents that symbol.

Number of bits needed to encode a symbol :  $\lceil \log(\frac{1}{p_i}) \rceil$

Number of bits needed in the worst case :  $\log(\frac{1}{p_i}) + 1$

### Comparison with Huffman code

Example : 3 symbols with uniform probability ( $p_1 = p_2 = p_3 = 1/3$ )

Arithmetic encoding : 7 bits.  $s_1 = 00, s_2 = 011, s_3 = 11$

Huffman coding : 5 bits.  $s_1 = 00, s_2 = 01, s_3 = 1$

Is Huffman coding always better ? No. When we send a message consisting of  $n$  symbols using Arithmetic encoding, we can reduce the number of bits needed to  $nE + 1$ , not  $nE + n$ , in the worst case ( $E = \sum(p_i \log(\frac{1}{p_i}))$  is entropy). On the other hand, Huffman encoding could cost you as much as  $nE + n$ .

Arithmetic encoding tries to reduce the  $n$  errors to 1 by computing combined probability for all possible strings of length  $n$ .

$$\begin{aligned} P(s_0, s_0, s_0, s_0, \dots, s_0, s_0) &= p_0^n \\ P(s_0, s_0, s_0, s_0, \dots, s_0, s_1) &= p_0^{n-1} p_1 \end{aligned}$$

And it send a bit string which can identify the small interval of a particular message in the space of all possible messages. Actually, we do not need to pre compute all probabilities of length  $n$ . We only need to keep numerical accuracy between binary interval  $[0, 1]$  and probability interval.

(Scribe's comment : When we encode/decode symbols one by one using Arithmetic encoding, we use smaller interval inside the slot of the previous symbol as starting interval for the next symbol. This accumulated savings makes the difference. On the other hand, in Huffman code, it's equivalent to starting from  $[0,1]$  interval every time.)

To summarize, Arithmetic Encoding

- can reduce the total number of bits for a message to  $(nE + 1)$
- may suffer from numerical errors
- may need to read the next symbol to decide a bit for the current symbol
- need to send the total length of the message first

## 2 Lossy Compression

Remind the whole compression procedure :

Normalize	$[0, 1.0]$	
Quantize	$[0, 2^b - 1]$	
Predict	$G$	
Encode residue	$d = (V - G)$	(hopefully small values, with biased distribution)

We are dealing with accuracy v.s. file size tradeoff here. Note that simplification does not always reduce transmission time if reducing causes guesses to be worse.

### Simplification of curve

Approaches to simplify :

- Fourier transform: Apply lowpass filters, remove high frequencies, and transmit fourier coefficients only.
- Edge collapse: combine two vertices into one optimal vertex.  
Optimal position ?  $(a, b, (a + b)/2)$ , preserve area, minimize haudorff)
- Vertex clustering: Using notion of a cluster (area). All vertices in the same cluster can be combined into one vertex.  
(example of cluster : regular grid)
- Vertex decimation: remove some vertices

When you do edge collapse one at a time, you may accumulate the error. For vertex clustering, simplification is done inside each cluster, and maximum displacement is bounded by the length of diagonal of the grid.

For vertex clustering, we can use quantization. You need to walk along the curve, and if  $v_i$  and  $v_{i+1}$  have the same quantized value, they are in the same cluster. And finally, replace the cluster with a single representative vertex inside the cell.

To summarize, vertex clustering

- is very cheap
- have a bound on total error less than the diagonal of the grid
- does not have much control over the number of vertices
- is not effective for slanted lines (results in zigzags)