A Low Complexity Block Implementation of Arithmetic Codes

On-Board Payload Data Compression Workshop
OBPDC 2008
26-27 June 2008 at ESTEC,
Noordwijk, Netherlands

S. M. R. Dibaj, F. Lahouti[1]

Wireless Multimedia Communications Laboratory
School of Electrical and Computer Engineering
University of Tehran, Tehran, Iran
Emails: m.dibaj@ece.ut.ac.ir, lahouti@ut.ac.ir

Abstract— **Arithmetic coding provides a high compression performance and is a key element for improved coding efficiency of recent multimedia communication standards such as H.264, and JPEG2000. In this article, we present a new block implementation of arithmetic codes that facilitates high speed coding, by eliminating the computationally complex interval renormalization. The proposed method operates on fixed length input and is set up based on following the decoder operations (intervals) at the encoder. Subsequently, for simultaneous truncation of arithmetic coding of an input block at the encoder and decoder, a few terminating bits are transmitted. As a result, the correct final interval is instantaneously obtained at the decoder, when the input block ends. Our extensive simulation results show that the redundancy of the proposed method with respect to entropy is small, while it maintains a very low complexity.**

Index Terms- **Arithmetic coding, variable length coding, entropy coding, data compression**

I. Introduction

Entropy coders are one of the important components of most compression standards. Although, arithmetic coding (AC) provides higher coding efficiency in this class, but many earlier generations of image and video coding standards such as JPEG, H.263, and MPEG-2 use Huffman codes for entropy coding due to its simplicity. Recently, AC has been subject to a higher interest and has been adopted in the latest standards such as JPEG2000, H.264 and MPEG-4. Besides the high compression performance of AC as Cleary *et al.* [1] have noted, its manageable extension to the case with adaptive source models is also a noteworthy feature. Moreover, recent advancements in hardware solutions for implementation of AC have been an essential motivating factor.

Nevertheless, there are several issues that make the widespread implementation of AC in different applications challenging. A primary issue is the efficient and practical implementation of AC. The important arithmetic encoding tasks include (disregarding source modeling): (a) interval update and arithmetic operations, (b) carry propagation and bit moves and (c) interval renormalization. In the last decade, due to the limitations of processors, interval update and arithmetic operations have been investigated more closely. For example the Q coder [2] and its variant [3] replace the multiplication with a pre-calculated table, provided that the current interval length is maintained in the range [0.75, 1.5) and can be approximated by one. Another method is the Skew coder proposed by Langdon and Rissanen [4], in which the multiplication is replaced with a simple shift. Specifically, in this method, the probability of the Least Probable Symbol (LPS) is approximated by 2^{-k}, where k is a positive integer. Consequently multiplication by LPS is replaced with a shift by k bits. However, at present, as the multiplication can be carried out with high precision in a single CPU clock, from a speed perspective, it is no longer advantageous to replace the multiplication with bit shifts or table-based approximations. Therefore, new research fronts for the speed gain are the interval renormalization, symbol search, and adaptation [5].

[1] This research has been supported in part by the Iran Telecommunications Research Center (ITRC).

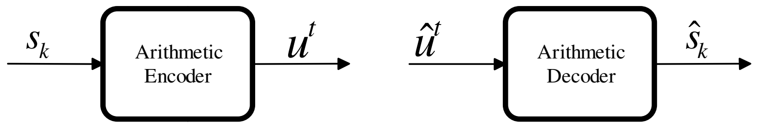


Figure 1: Arithmetic Coding System Model

Another problem in AC, as in other typical variable length codes, is the vulnerability to error. A one bit error in the encoded sequence, often propagates in the decoding of the rest of the sequence, and may lead to a different number of decoded symbols when compared to those encoded. Also, AC is not partially decodable and it produces a codeword for the entire message. Therefore, we must decode the entire message for access to a certain part of it. Moreover, we cannot say which part of the codeword refers to which part of the message. A solution that is proposed for solving these problems is the block arithmetic coding. Teuhola and Raita proposed a variable length input to fixed length output arithmetic block coding (ABC) [6], [7]. The ABC divides the input sequence into variable length substrings by splitting the interval iteratively in proportion to probability of symbols, while allowing possible collisions. When collision occurs the encoding procedure is terminated and then fixed length codewords are assigned to substrings. Boncelet [8] suggests a Block Arithmetic Coding (BAC), which parses the set of output codewords with fixed length recursively into non-empty disjoint sets of size equal to the number of source alphabet. This splitting is continued until the subset has fewer than *n* codewords.

In this paper, we introduce a Block Input implementation of Arithmetic Coding (BIAC) that eliminates bit moves and interval renormalization in AC operations for short blocks. This amounts to substantial savings in computational complexity and is achieved at the price of only approximately a one bit redundancy per block when compared to an ideal AC.

This paper is organized as follows. Section II describes the principles of AC and its practical implementation. In section III, the proposed method for efficient implementation of AC is presented. Experimental results and comparisons are presented in Section IV. Conclusions are given in section V.

II. Principles of Arithmetic Coding

Fig. 1 demonstrates the block diagram of typical arithmetic coding. Note that a subscript indicates the index of an input symbol to AC and a superscript shows that of the AC output bit in the rest of this paper. The AC operates by recursive partitioning of intervals, based on the symbol probabilities. The interval is initially set to $[0, 1]$ and is denoted by:

$$\phi(s_k) = [low_k, high_k), \tag{1}$$

where low and $high$ are respectively, the lower and upper bounds of the current interval and are updated by the following recursive equations:

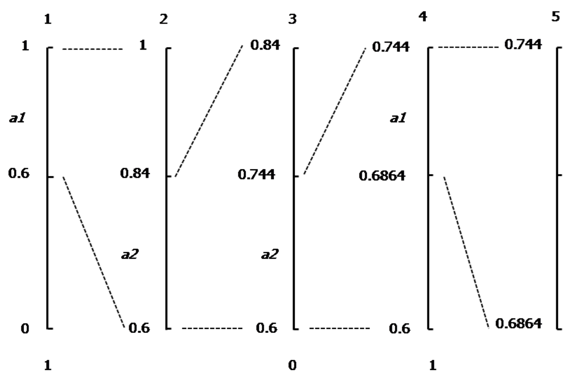


Figure 2: Arithmetic Coding Example

$$low_k = low_{k-1} + (high_{k-1} - low_{k-1}) \times c(s_k),$$
$$high_k = low_k + (high_{k-1} - low_{k-1}) \times p(s_k), \tag{2}$$

where $c(s_k)$ and $p(s_k)$ are respectively the cumulative and the probability distribution functions of s_k.

When the number of input symbols increase, a higher precision is required to accurately present very small real numbers. To avoid this and prevent the current interval becoming too small, often a periodic renormalization is considered. The most popular renormalization scheme is introduced by Witten *et al.* in Communications of the ACM, and is known as the CACM renormalization [9].

In binary representation of real numbers, when a number lies in $[0, 0.5]$ the first bit is zero, whereas for any number in $[0.5, 1]$ the first bit is one. In CACM, when the entire current interval is below or above 0.5 (settled), the first bit is emitted and the current interval is doubled (a bit shift). The carry over problem appears when the current interval straddles 0.5. This problem is solved in CACM with a specific treatment usually labeled as *scale3* procedure. The *scale3* doubles the current interval when it straddles the midpoint but not out of $[0.25, 0.75]$ and saves the *r*, where *r* is the number of scalings without emitting any bit. When a bit is emitted then it is followed by *r* opposite bits. This technique guarantees that the current interval always satisfies $low_k < 0.25 < 0.5 \leq high_k$ or $low_k < 0.5 < 0.75 \leq high_k$.

Table1: Binary Arithmetic Coding Algorithm

Encoder

```
1.$high_1$←1, $low_1$←0, $t$←0, $r$←0
2.for $k$ = 1 to $N$ do
  2.1.if $s_k = 0$
      then $low_k \leftarrow low_{k-1} + (high_{k-1} - low_{k-1}) \times p(s_k)$
      else $high_k \leftarrow low_{k-1} + (high_{k-1} - low_{k-1}) \times p(s_k)$
  2.2.while $high_k \leq 0.5$ OR $low_k \geq 0.5$ OR $0.25 \leq low_k < high_k \leq 0.75$
      2.2.1.if $high_k < 0.5$ (scaling1)
            $u^t \leftarrow 0$, $t$←$t$+1, $low_k \leftarrow 2 \times low_k, high_k \leftarrow 2 \times high_k$
            2.2.1.1while $r > 0$
                  $u^t \leftarrow 1$, $t$←$t$+1, $r$← $r$-1,
      2.2.2.if $low_k \geq 0.5$ (scaling2)
                  $u^t \leftarrow 1$, $t$←$t$+1,$low_k \leftarrow 2 \times (low_k - 0.5), high_k \leftarrow 2 \times (high_k - 0.5)$
            2.2.2.1.while $r > 0$
                  $u^t \leftarrow 0$, $t$←$t$+1, $r$← $r$-1,
      2.2.3.if $0.25 \leq low_k < high \leq 0.75$ (scaling3)
                  $r$← $r$+1, $low_k \leftarrow 2 \times (low_k - 0.25), high_k \leftarrow 2 \times (high_k - 0.25)$,
```

Decoder

```
1.$high_1$←1, $low_1$←0, $t$←$P$, $v \leftarrow \sum_{n=1}^{P} 2^{-n} u^n$
2.for $k$ = 1 to $N$ do
  2.1.if $v < low_{k-1} + (high_{k-1} - low_{k-1}) \times p(s_k)$
      then $s_k \leftarrow 0, high_k \leftarrow low_{k-1} + (high_{k-1} - low_{k-1}) \times p(s_k)$
      else $s_k \leftarrow 1, low_k \leftarrow low_{k-1} + (high_{k-1} - low_{k-1}) \times p(s_k)$
  2.2.while $high_k \leq 0.5$ OR 0.5 $low_k \geq 0.5$ OR 0.5 scale3 condition
      2.2.1.if $high_k \leq 0.5$ 0.5
            then $low_k \leftarrow 2 \times low_k, high_k \leftarrow 2 \times high_k, v \leftarrow 2 \times v$
      2.2.2.if $low_k \geq 0.5$
            then $low_k \leftarrow 2 \times (low_k - 0.5), high_k \leftarrow 2 \times (high_k - 0.5),\ v \leftarrow 2 \times (v - 0.5)$
      2.2.3.else (scale3 condition)
            then $low_k \leftarrow 2 \times (low_k - 0.25), high_k \leftarrow 2 \times (high_k - 0.25),\ v \leftarrow 2 \times (v - 0.25)$
  2.3.$t$←$t$+1, $v \leftarrow v + 2^{-P} \times u^t$
```

For binary sources, the equations in (2) are very simple. Fig. 2 illustrates the binary arithmetic coding (BAC), where the source alphabet is $\{a1, a2\}$ with the probabilities {0.4, 0.6}, respectively. This example shows that the current interval is split in proportion to the probability of symbols with one dividing point. This dividing point is used to update the next interval as follows:

$$\begin{cases} low_k \leftarrow \ low_{k-1} + (high_{k-1} - low_{k-1}) \times p(s_k)\ ,\ s_k = 0 \\ high_k \leftarrow \ low_{k-1} + (high_{k-1} - low_{k-1}) \times p(s_k)\ ,\ s_k = 1 \end{cases} \tag{3}$$

In Table 1 the procedure of encoding (and decoding) of BAC is presented. Following the algorithm for the example of Fig. 2, we have $[0.6864, 0.744]$ as the final interval of encoding $\{a1, a2, a2, a1\}$ and any number that is placed in this interval may be sent as the final codeword.

For decoding of a sequence, we must reproduce the encoding process at the decoder. Generally, any number in the final interval can be assumed as the encoding result. Let us consider 0.7 as the codeword for the example of Fig. 2 First, the interval $[0, 1]$ is split proportionately with the probability of symbols. Since 0.7 lies in $[0.6, 1]$ the first symbol is $a2$a2. Now the current interval, [0.6, 1], is to be split. The number 0.7 is now in $[0.6, 0.84]$, therefore the second symbol is $a1$ and so on.

For practical purposes, usually an integer implementation of AC is used. In this case $[0, 1]$ is mapped to $[0, 2^P)$, where P is the precision of arithmetic encoder and refers to the register size for saving $[low_k, high_k)$. In this manner, the decoding procedure is somewhat different. The decoder can initiate the decoding, only when the buffer becomes full. Then the stored codeword in the buffer, is compared with the dividing point used by (4), for determining the encoded symbol as in 2.1 in decoder of Table 1. After decoding each symbol the renormalization is done by shifting all the variables (*high, low, v*) and LSB of these variables are filled with 1, 0, and u^t, respectively.

III. Block Input Arithmetic Coding

The proposed BIAC is based on dividing the input sequence into fixed length blocks and encoding each block separately. For simplicity we focus on binary AC, but the suggested procedure can also be extended to non-binary

Table 2: Binary BIAC Algorithm

Encoder

```
1.$high_0$←1, $low_0$←0, $t$←1,
2.for $k$ = 1 to $L$ do
  2.1.if  $s_k = 0$
      then $low_k \leftarrow low_{k-1} + (high_{k-1} - low_{k-1}) \times p(s_k)$
      else $high_k \leftarrow low_{k-1} + (high_{k-1} - low_{k-1}) \times p(s_k)$
3. while  $high_k^t = low_k^t$
  $u_t$←$high_k^t$, $t$←$t$+1,
4.  $m$←$t$+1
5. while  $high_k^{t+1} = 1$ OR $low_k^{t+1} = 0$
  $t$←$t$+1,
6. if $high_k^{t+1} = 1$
  then  $t \leftarrow t+1$, $u^m$←1, $u^{m+1}$: $u^{t+m}$←0,
  else   $t \leftarrow t+1$, $u^m$←0, $u^{m+1}$: $u^{t+m}$←1,
```

Decoder

```
1.$high_1$ ← 1, $low_1$ ← 0, $t$←1, $mid \leftarrow p(s_k)$,  $k \leftarrow 1$,
2.while $k \neq L$ do
  2.1.if $u^t = 0$
      then $h_{t+1} \leftarrow (l_t + h_t)/2, t = t + 1$,
      else $l_{t+1} \leftarrow (l_t + h_t)/2, t = t + 1$,
  2.2.while $h_t \leq mid$ OR  $l_t \geq mid$ do
      2.2.1.if $h_t \leq mid$
            then $s_k \leftarrow 0, k \leftarrow k + 1, high_k \leftarrow low_{k-1} + (high_{k-1} - low_{k-1}) \times p(s_k)$
            else  $s_k \leftarrow 1, k \leftarrow k + 1, low_k \leftarrow low_{k-1} + (high_{k-1} - low_{k-1}) \times p(s_k)$
      2.2.2 $mid \leftarrow low_k + (high_k - low_k) \times p(s_k)$
```

cases. The encoding and decoding procedures are presented in Table 2. When a block is completely en/decoded then the en/decoder is reset to the initial state. For encoding a block of L symbols using the BIAC, an appropriate (terminating) codeword in the final interval is to be considered, so that the decoder can decode the entire block. Considering the example of Fig. 2, the final interval in binary representation with 8 bit resolution is $[10101111, 10111110]$. We need to select a number (codeword) in this interval to ensure correct decoding. The first three bits in *high* and *low* are equal and form the first segment of the codeword. Next, we have two choices for the subsequent segment of the codeword:

(i) We consider a 1 and a number of 0's until we reach the position of the second 1 in *high* following the first three bits. The 1 ensures that the transmitted codeword is greater than *low*, and the 0's indicate that the codeword is smaller than *high*. The terminated codeword is then transmitted as 101**10**.

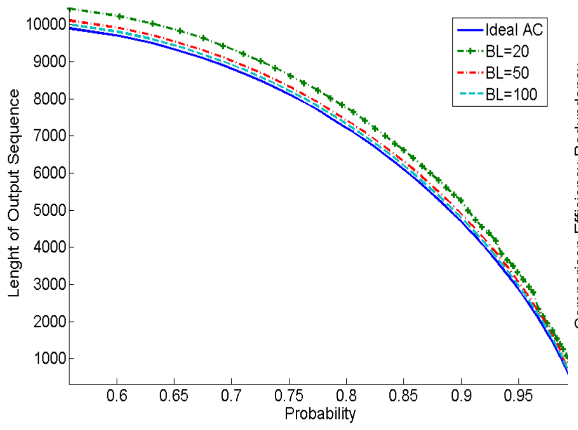
(ii) Alternatively, we consider a 0 followed by a number of 1's until we reach the position of the second 0 or the last bit in *low*. In this case, the terminated codeword is 101**01111**.

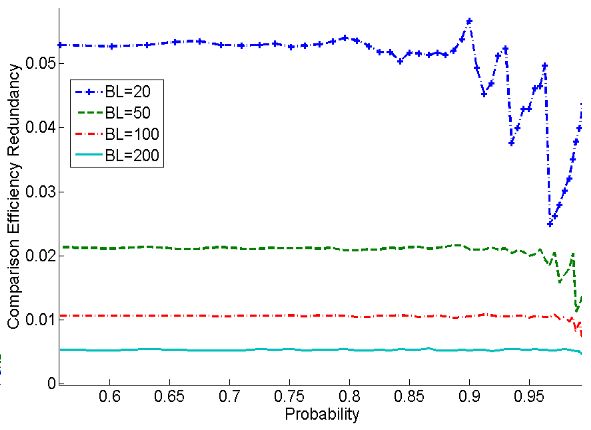
At the encoder presented in Table 2, the two described options for terminating the codeword are concurrently assessed and the one with the shorter length is selected.

To synchronize the encoding and decoding operations, we use a bit by bit decoding scheme which has also been used in [10] in the context of joint source channel decoding. To this end, two intervals are utilized: the first interval, as in the encoder, is constructed based on the input symbol probability denoted by $[low, high)$; the second interval $[l, h)$, is set up based on the received bits. The latter interval is initially set to $[0,1)$ and is updated as follows:

$$\begin{cases} l \leftarrow \frac{l+h}{2}, \ u_k = 0 \\ h \leftarrow \frac{l+h}{2}, \ u_k = 1 \end{cases} \quad (4)$$

depending on the received bits. If $[l, h)$ lies in one of the two intervals identified within $[low, high)$, one symbol is decoded, and the $[low, high)$ interval is updated. This procedure is continued iteratively until no symbols may be decoded ($[l, h)$ straddles the dividing point). Next, the following received bit is processed in the same manner, until a block of L symbols are decoded. Subsequently, the decoder is reset to the initial state.





(a) **(b)**

Fig.3 (a) Length of output for BIAC encoding of a sequence with different input block lengths as function of probability (b) Redundancy: difference between output sequence of BIAC and integer implementation of AC with CACM and normalized by *N* as function of probability.

IV.Experimental results

In this section, we analyze the effect of blocking on compression efficiency of the proposed BIAC. Simulations are performed for a binary source with i.i.d. distribution and input sequence with *N*=10000 symbols. Matlab code is run on a computer with 2.4GHz Intel Pentium IV processor and 512 MB RAM. The results are averaged over 20 runs and for different probabilities in the range of [0.5, 1] with a step size of 0.001.

a) Compression efficiency

Fig. 3(a) depicts the length of the sequence encoded using the BIAC with different input block lengths, *L*. Note that the performance difference between an integer implementation of AC with CACM (referred to simply as CACM) and the entropy is negligible. To clarify the effect of blocking in compression efficiency, Fig. 3(b) demonstrates the difference between the output sequence length of BIAC and CACM. It is evident that when *L* is increased, the compression efficiency is improved. Specifically, this improvement is linear with respect to *L* and is approximately 0.01 bit per input block bit. The compression efficiency of BIAC approaches that of integer implementation of AC with CACM for small entropy. Although a small source entropy causes certain roughness in curves. In this case, the codeword corresponding to each input block is much shorter than that in the case of a high source entropy. The lengths of these short codewords are very sensitive to the value of source probability. In Table 3 the probability of the sequences, which have a local peak in Fig. 3(b) is replaced with that of a local minimum. The redundancy is hence reduced and this reduction is more for lower source entropy.

As mentioned earlier, using BIAC, we can eliminate the renormalization with various *L* subject to their entropy: for example if *l* bits are used to represent the probability then the encoding can be done without any renormalization for blocks with:

$$L = (p - l)/H(x) \qquad (5)$$

where *H(x)* is the source entropy.

Table 3: Comparison of redundancy (see Fig. 3 for definition) with exact & improved probabilities (*L*=20)

Exact			Improved		
Probability	Output length	Redundancy	Probability	Output length	Redundancy
0.9	5263	5.67	0.912	5177	4.81
0.93	4178	5.23	0.935	4050	3.95
0.959	2938	4.65	0.967	2871	3.08
0.963	2785	4.97	0.967	2566	2.78

Table 4: Comparison of relative redundancy (%) of different implementations of BAC

Probability	Skew coder	MQ coder	BIAC
0.6	3.36	3.8	2 (L=50)
0.8	1.9	4.5	1.7 (L=70)
0.9	0.9	4.7	2 (L=100)
0.99	1	4.2	1.4 (L=600)
0.997	0.7	4.5	1.3 (L=2500)

Table 5: Encoding time of BIAC and CACM per symbol (ns)

Entropy	CACM	BIAC (L=20)	BIAC (L=50)
0.6	17.9	3.36	1.58
0.8	12.67	3.16	1.51
0.9	8.12	2.9	1.42
0.99	3.38	2.36	1.21
0.999	2.93	2.25	1.08

Table 4 shows the relative redundancy, which is the deference between the code rate and the entropy for BIAC, MQ coder and skew coder. We use the results of [5] for MQ coder, which are provided in an identical setting. Skew coder performs very well in terms of the relative redundancy for compression of low entropy sources; however, it requires a renormalization procedure such as CACM. The relative redundancy of the MQ coder is about 4-5%. The BIAC relative redundancy is shown for various L, so that no renormalization is needed and better compression efficiency is provided. The results show that the proposed BIAC, when compared to other two methods, provides the smallest relative redundancy for the cases with moderate to high source entropy, and performs competitively for the cases with lower source entropy. For example, for a binary source with probability 0.997, the output sequence of BIAC, with a much smaller complexity, is only 2 bits longer than that of the skew coder.

b) Complexity considerations

In another experiment, we compare the speed of BIAC with CACM. The result of this experiment is presented in Table 5. It is evident that BIAC, due to elimination of the renormalizations, is very fast when compared to CACM. Moreover, the encoding time of CACM drops fast for smaller source entropies; as in this case, the number of renormalizations is reduced. However, even in this case CACM is slower than BIAC.

V. Conclusions

In this article, we presented a new block input implementation of arithmetic codes that facilitates high speed coding, by eliminating the computationally complex interval renormalization. The proposed method simultaneously truncates arithmetic coding of an input block at the encoder and decoder, by transmitting a few terminating bits. As a result, the correct final interval is instantaneously obtained at the decoder, when the input block ends. Our analysis and simulations show that the redundancy of the proposed method with respect to entropy is small, while offering substantial reductions in complexity. As the proposed implementation closely follows the original arithmetic coding algorithm, it is expected to be easily admissible to incorporating adaptation based on source statistics.

References

[1] J. Cleary, S. Irvine, and I. Rinsma-Melchert, “On the insecurity of arithmetic coding,” *Comput. Secur.* vol. 14, no. 2, pp. 167–180, 1995.

[2] W. B. P. al, “An overview of the basic principles of the Q-coder adaptive binary arithmetic coder,” *IBM J. Res. Develop.*, vol. 32, pp. 717–726, 1988.

[3] W. B. B. Pennebaker and J. L. Mitchell, *JPEG Still Image Data Compression Standard*. New York: Chapman and Hall, 1992.

[4] G.G. Langdon Jr. and J.J. Rissanen, “Compression of black-white images with arithmetic coding,” *IEEE Trans. Communications,* vol. 29(6), pp. 858–867, June 1981.

[5] A. Said “Comparative Analysis of arithmetic coding computational complexity*”, Proceedings of the Data Compression Conference,* March 2004.

[6] J. Teuhola and T. Raita, “Arithmetic coding into fixed-length codewords,” *IEEE Trans. Inform. Theory,* vol**.** 40, pp. 219-223**,** Jan. 1994.

[7] J. Teuhola and T. Raita, ‘‘Piecewise arithmetic coding,” *Proceedings of the Data Compression Conference* pp. 33-42, 1991.

[8] C. *G.* Boncelet, Jr., “Block arithmetic *coding for source compression*,” *IEEE Trans. Inform. Theory,* vol**.** 39, pp. 1546-1554**,** Jan. 1993..

[9] I. H. Witten, R. M. Neal, and J. G. Cleary, “Arithmeti*c coding for data compressi*on,” *Commun. ACM*, vol. 30**,** pp. 520–540, 1987.

[10] T. Guionnet and C. Guillemot, “Soft decoding and synchronization of arithmetic codes: *Application* to image transmission over noisy channels,” *IEEE Trans. Image Process.*, vol. 12, no. 12, pp. 1599–1609, Dec. 2003.