

Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 18, 2013

T. Terriberry
Mozilla Corporation
October 15, 2012

Coding Tools for a Next Generation Video Codec
draft-terriberry-codingtools-00

Abstract

This document proposes a number of coding tools that could be incorporated into a next-generation video codec.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 18, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	3
2.	Entropy Coding	4
2.1.	Non-binary Arithmetic Coding	5
2.2.	Non-binary Context Modeling	5
2.3.	Simple Experiment	9
3.	Reversible Integer Transforms	10
3.1.	Lifting Steps	10
3.2.	4-Point Transform	13
3.3.	Larger Transforms	16
4.	Development Repository	17
5.	IANA Considerations	18
6.	Acknowledgments	19
7.	Informative References	20
	Author's Address	21

1. Introduction

One of the biggest contributing factors to the success of the Internet is that the underlying protocols are implementable on a royalty-free basis. This allows them to be implemented widely and easily distributed by application developers, service operators, and end users, without asking for permission. In order to produce a next-generation video codec that is competitive with the best patent-encumbered standards, yet avoids patents which are not available on an open-source compatible, royalty-free basis, we must use old coding tools in new ways and develop new coding tools. This draft documents some of the tools we have been working on for inclusion in such a codec. This is early work, and the performance of some of these tools (especially in relation to other approaches) is not yet fully known. Nevertheless, it still serves to outline some possibilities an eventual working group, if formed, could consider.

2. Entropy Coding

The basic theory of entropy coding was well-established by the late 1970's [[Pas76](#)]. Modern video codecs have focused on Huffman (or "Variable-Length Codes"/VLCs) and binary arithmetic coding. Huffman codes are limited in the amount of compression they can provide and the design flexibility they allow, but as each code word consists of an integer number of bits, their implementation complexity is very low, so they were provided at least as an option in every video codec up through H.264. Arithmetic coding, on the other hand, uses code words that can take up fractional parts of a bit, and are more complex to implement. However, the prevalence of cheap, H.264 High Profile hardware, which requires support for arithmetic coding, shows that it is no longer so expensive that a fallback VLC-based approach is required. Having a single entropy-coding method simplifies both up-front design costs and interoperability.

However, the primary limitation of arithmetic coding is that it is an inherently serial operation. A given symbol cannot be decoded until the previous symbol is decoded, because the bits (if any) that are output depend on the exact state of the decoder at the time it is decoded. This means that a hardware implementation must run at a sufficiently high clock rate to be able to decode all of the symbols in a frame. Higher clock rates lead to increased power consumption, and in some cases the entropy coding is actually becoming the limiting factor in these designs.

As fabrication processes improve, implementers are very willing to trade increased gate count for lower clock speeds. So far, most approaches to allowing parallel entropy coding have focused on splitting the encoded symbols into multiple streams that can be decoded independently. This "independence" requirement has a non-negligible impact on compression, parallelizability, or both. For example, H.264 can partition frames into "slices" which might cover only a small subset of the blocks in the frame. In order to allow decoding these slices independently, they cannot use context information from blocks in other slices (harming compression). Those contexts must adapt rapidly to account for the generally small number of symbols available for learning probabilities (also harming compression). Furthermore, an encoder must specially format the stream to use multiple slices per frame to allow any parallel entropy decoding at all. Encoders rarely have enough information to evaluate this "compression efficiency" vs. "parallelizability" trade-off, since they don't generally know the limitations of the decoders for which they are encoding. The same set of drawbacks apply to the DCT token partitions in VP8 [[RFC6386](#)].

Terriberry

Expires April 18, 2013

[Page 4]

2.1. Non-binary Arithmetic Coding

Instead, we propose a very different approach: use non-binary arithmetic coding. In binary arithmetic coding, each decoded symbol has one of two possible values: 0 or 1. The original arithmetic coding algorithms allow a symbol to take on any number of possible values, and allow the size of that alphabet to change with each symbol coded. Reasonable values of N (for example, $N \leq 16$) offer the potential for a decent throughput increase for a reasonable increase in gate count for hardware implementations.

Binary coding allows a number of computational simplifications. For example, for each coded symbol, the set of valid code points is partitioned in two, and the decoded value is determined by finding the partition in which the actual code point that was received lies. This can be determined by computing a single partition value (in both the encoder and decoder) and (in the decoder) doing a single comparison. A non-binary arithmetic coder partitions the set of valid code points into multiple pieces (one for each possible value of the coded symbol). This requires the encoder to compute two partition values, in general (for both the upper and lower bound of the symbol to encode). The decoder, on the other hand, must search the partitions for the one that contains the received code point. This requires computing at least $O(\log N)$ partition values.

However, coding a parameter with N possible values with a binary arithmetic coder requires $O(\log N)$ symbols in the worst case (the only case that matters for hardware design). Hence, this does not represent any actual savings (and actually represents an increase in the number of encoder partition value computations). In addition, there are a number of overheads that are per-symbol, rather than per-value. For example, renormalization (which enlarges the set of valid code points after partitioning has reduced it too much), carry propagation (to deal with the case where the high and low ends of a partition straddle a bit boundary), etc., are all performed on a symbol-by-symbol basis. Since a non-binary arithmetic coder codes a given set of values with fewer symbols than a binary one, it incurs these per-symbol overheads less often. This suggests that a non-binary arithmetic coder can actually be more efficient than a binary one.

2.2. Non-binary Context Modeling

The other aspect that binary coding simplifies is probability modeling. In arithmetic coding, the size of the sets the code points are partitioned into are (roughly) proportional to the probability of each possible symbol value. Estimating these probabilities is part of the coding process, though it can be cleanly separated from the

Terriberry

Expires April 18, 2013

[Page 5]

task of actually producing the coded bits. In a binary arithmetic coder, this requires estimating the probability of only one of the two possible values (since the total probability is 1.0). This is often done with a simple table lookup that maps the old probability and the most recently decoded symbol to a new probability to use for the next symbol in the current context. The trade-off, of course, is that non-binary symbols must be "binarized" into a series of bits, and a context (with an associated probability) chosen for each one.

In a non-binary arithmetic coder, the decoder must compute at least $O(\log N)$ cumulative probabilities (one for each partition value it needs). Because these probabilities are usually not estimated directly in "cumulative" form, this can require computing $(N-1)$ non-cumulative probability values. Unless N is very small, these cannot be updated with a single table lookup. The normal approach is to use "frequency counts". Define the frequency of value k to be

$$f[k] = A \times \text{the number of times } k \text{ has been observed} + B$$

where A and B are parameters (usually $A=2$ and $B=1$ for a traditional Krichevsky-Trofimov estimator). The resulting probability, $p[k]$, is given by

$$f_t = \sum_{k=0}^{N-1} f[k]$$

$$p[k] = \frac{f[k]}{f_t}$$

When f_t grows too large, the frequencies are rescaled (e.g., halved, rounding up to prevent reduction of a probability to 0).

When f_t is not a power of two, partitioning the code points requires actual divisions (see [\[RFC6716\] Section 4.1](#) for one detailed example of exactly how this is done). These divisions are acceptable in an audio codec like Opus [\[RFC6716\]](#), which only has to code a few hundreds of these symbols per second. But video requires hundreds of thousands of symbols per second, at a minimum, and divisions are still very expensive to implement in hardware.

There are two possible approaches to this. One is to come up with a replacement for frequency counts that produces probabilities that sum to a power of two. Some possibilities, which can be applied individually or in combination:

Terriberry

Expires April 18, 2013

[Page 6]

1. Use probabilities that are fixed for the duration of a frame. This is the approach taken by VP8, for example, even though it uses a binary arithmetic coder. In fact, it is possible to convert many of VP8's existing binary-alphabet probabilities into probabilities for non-binary alphabets, an approach that is used in the experiment presented at the end of this section.
2. Use parametric distributions. For example, DCT coefficient magnitudes usually have an approximately exponential distribution. This distribution can be characterized by a single parameter, e.g., the expected value. The expected value is trivial to update after decoding a coefficient. For example

$$E[x[n+1]] = E[x[n]] + \text{floor}(C*(x[n] - E[x[n]]))$$

produces an exponential moving average with a decay factor of $(1-C)$. For a choice of C that is a negative power of two (e.g., $1/16$ or $1/32$ or similar), this can be implemented with a two adds and a shift. Given this expected value, the actual distribution to use can be obtained from a small set of pre-computed distributions via a lookup table. Linear interpolation between these pre-computed values can improve accuracy, at the cost of $O(N)$ computations, but if N is kept small this is trivially parallelizable, in SIMD or otherwise.

3. Change the frequency count update mechanism so that ft is constant. For example, let

$$fl[k] = \frac{\sum_{i=0}^{k-1} f[i]}{k}$$

be the cumulative frequency of all symbol values less than k and

$$e[i][k] = \begin{cases} 0, & k \leq i \\ 1, & k > i \end{cases}$$

be the elementary change in the cumulative frequency count $fl[k]$ caused by adding 1 to $f[i]$. Then one possible update formula after decoding the value i is

$$fl[k]' = fl[k] - \text{floor}(D*fl[k]) + k + F*e[i][k]$$

where D is a negative power of two chosen such that $\text{floor}(D*ft) == (N + F)$. This ensures that $ft == fl[N] == fl[N]'$

is a constant. This requires $O(N)$ operations, but the arithmetic is very simple (given the freedom to choose D and F , and to some extent N), and trivially parallelizable, in SIMD or otherwise. The downside is the addition of the value k at each step. This is necessary to ensure that the probability of an individual symbol ($fl[k+1] - fl[k]$) is never reduced to zero. However it is equivalent to mixing in a uniform distribution with counts that are otherwise an exponential moving average. That means that ft and F must be sufficiently large, or there will be an adverse impact on coding efficiency. The upside is that $F \cdot e[i]$ may be replaced by any monotonically non-decreasing vector whose N th element is F . That is, instead of just incrementing the probability of symbol i , it can increase the probability of values that are highly correlated with i . E.g., this allows decoding value i to apply a small probability increase to the neighboring values $(i-1)$ and $(i+1)$, in addition to a large probability increase to the value i . This may help, for example, in motion vector coding, and is much more sensible than the approach taken with binary context modeling, which often does things like "increase the probability of all even values when decoding a 6" because the same context is always used to code the least significant bit.

The other approach is to change the function used to partition the set of valid code points so that it does not need a division, even when ft is not a power of two. Let the range of valid code points in the current arithmetic coder state be $[L, L+R)$, where L is the lower bound of the range and R is the number of valid code points. Assume that $ft \leq R < 2 \cdot ft$ (this is easy to enforce with the normal rescaling operations used with frequency counts). Then one possible partition function is

$$r[k] = fl[k] + \min(fl[k], R - ft)$$

so that the new range after coding symbol k is $[L+r[k], L+r[k+1])$.

This is a variation of the partition function proposed by [\[SM98\]](#). The size of the new partition ($r[k+1] - r[k]$) is no longer truly proportional to $R \cdot p[k]$. It can be off by up to a factor of 2. However, if the probabilities are accurate and the symbols being coded are independent, the average inefficiency introduced can be as low as $\log_2(\log_2(e) \cdot 2/e) \sim 0.0861$ bits per symbol. The peak error can be as large as one bit per symbol, however. This error can, of course, be reduced by coding fewer symbols with larger alphabets. In practice the overhead is roughly equal to the overhead introduced by other approximate arithmetic coders like H.264's CABAC.

Terriberry

Expires April 18, 2013

[Page 8]

2.3. Simple Experiment

As a simple experiment to validate the non-binary approach, we compared a non-binary arithmetic coder to the VP8 (binary) entropy coder. This was done by instrumenting `vp8_treed_read()` in `libvpx` to dump out the symbol decoded and the associated probabilities used to decode it. This data only includes macroblock mode and motion vector information, as the DCT token data is decoded with custom inline functions, and not `vp8_treed_read()`. This data is available at https://people.xiph.org/~tterribe/daala/ec_test0/ec_tokens.txt. It includes 1,019,670 values encode using 2,125,995 binary symbols (or 2.08 symbols per value). We expect that with a conscious effort to group symbols during the codec design, this average could easily be increased.

We then implemented both the regular VP8 entropy decoder (in plain C, using all of the optimizations available in `libvpx` at the time) and a multisymbol entropy decoder (also in plain C, using similar optimizations), which encodes each value with a single symbol. For the decoder partition search in the non-binary decoder, we used a simple for loop ($O(N)$ worst-case), even though this could be made constant-time and branchless with a few SIMD instructions such as (on x86) `PCMPGTW`, `PACKUSWB`, and `PMOVMASKB` followed by `BSR`. The source code for both implementations is available at https://people.xiph.org/~tterribe/daala/ec_test0/ec_test.c (compile with `-DEC_BINARY` for the binary version and `-DEC_MULTISYM` for the non-binary version).

The test simply loads the tokens, and then loops 1024 times encoding them using the probabilities provided, and then decoding them. The loop was added to reduce the impact of the overhead of loading the data, which is implemented very inefficiently. The total runtime on a Core i7 from 2010 is 53.735 seconds for the binary version, and 27.937 seconds for the non-binary version, or a 1.92x improvement. This is very nearly equal to the number of symbols per value in the binary coder, suggesting that the per-symbol overheads account for the vast majority of the computation time in this implementation.

3. Reversible Integer Transforms

Integer transforms in image and video coding date back to at least 1969 [[PKA69](#)]. Although standards such as MPEG2 and MPEG4 Part 2 allowed some flexibility in the transform implementation, implementations were subject to drift and error accumulation, and encoders had to impose special macroblock refresh requirements to avoid these problems, not always successfully. As transforms in modern codecs only account for on the order of 10% of the total decoder complexity, and, with the use of weighted prediction with gains greater than unity and intra prediction, are far more susceptible to drift and error accumulation, it no longer makes sense to allow a non-exact transform specification.

However, it is also possible to make such transforms "reversible", in the sense that applying the inverse transform to the result of the forward transform gives back the original input values, exactly. This gives a lossy codec, which normally quantizes the coefficients before feeding them into the inverse transform, the ability to scale all the way to lossless compression without requiring any new coding tools. This approach has been used successfully by JPEG XR, for example [[TSSRM08](#)]. Even when the goal is still lossy compression, the ability to selectively enable lossless compression for some parts of the frame can be beneficial. For example, in very dark areas, single pixel value differences are sometimes visible. In very still areas (particularly from synthetic content), coding a region losslessly allows it to perfectly predict that region in many future frames.

Such reversible transforms can be constructed using "lifting steps", a series of shear operations that can represent any set of plane rotations, and thus any orthogonal transform. This approach dates back to at least 1992 [[BE92](#)], which used it to implement a four-point 1-D Discrete Cosine Transform (DCT). Their implementation requires 6 multiplications, 10 additions, 2 shifts, and 2 negations, and produces output that is a factor of $\sqrt{2}$ larger than the orthonormal version of the transform. The expansion of the dynamic range directly translates into more bits to code for lossless compression. Because the least significant bits are usually very nearly random noise, this scaling increases the coding cost by approximately half a bit per sample.

3.1. Lifting Steps

To demonstrate the idea of lifting steps, consider the two-point transform

$$\begin{bmatrix} y0 \\ \\ y1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ & \\ -1 & 1 \end{bmatrix} \begin{bmatrix} x0 \\ \\ x1 \end{bmatrix}$$

This can be implemented up to scale via

$$y0 = x0 + x1$$

$$y1 = 2*x1 - y0$$

and reversed via

$$x1 = (y0 + y1) >> 1$$

$$x0 = y0 - x1$$

Both $y0$ and $y1$ are too large by a factor of $\sqrt{2}$, however.

It is also possible to implement any rotation by an angle t , including the orthonormal scale factor, by decomposing it into three steps:

$$u0 = x0 + \frac{\cos(t) - 1}{\sin(t)} * x1$$

$$y1 = x1 + \sin(t)*u0$$

$$y0 = u0 + \frac{\cos(t) - 1}{\sin(t)} * y1$$

By letting $t=-\pi/4$, we get an implementation of the first transform that includes the scaling factor. To get an integer approximation of this transform, we need only replace the transcendental constants by fixed-point approximations:

$$u0 = x0 + ((27*x1 + 32) >> 6)$$

$$y1 = x1 - ((45*u0 + 32) >> 6)$$

$$y0 = u0 + ((27*y1 + 32) >> 6)$$

This approximation is still perfectly reversible:


```
u0 = y0 - ((27*y1 + 32) >> 6)
```

```
x1 = y1 + ((45*u0 + 32) >> 6)
```

```
x0 = u0 - ((27*x1 + 32) >> 6)
```

Each of the three steps can be implemented using just two ARM instructions, with constants that have up to 14 bits of precision (though using fewer bits allows more efficient hardware implementations, at a small cost in coding gain). However, it is still much more complex than the first approach.

We can get a compromise with a slight modification:

```
y0 = x0 + x1
```

```
y1 = x1 - (y0 >> 1)
```

This still only implements the original orthonormal transform up to scale. The y_0 coefficient is too large by a factor of $\sqrt{2}$ as before, but y_1 is now too small by a factor of $\sqrt{2}$. If our goal is simply to (optionally quantize) and code the result, this is good enough. The different scale factors can be incorporated into the quantization matrix in the lossy case, and the total expansion is roughly equivalent to that of the orthonormal transform in the lossless case. Plus, both steps together require just two ARM instructions, total.

However, if instead we want to apply additional transformations to the data, or use the result to predict other data, it becomes much more convenient to have uniformly scaled outputs. For a two-point transform, there is little we can do to improve on the three-multiplications approach above. However, for a four-point transform, we can use the last approach and arrange multiple transform stages such that the "too large" and "too small" scaling factors cancel out, producing a result that has the true, uniform orthonormal scaling. To do this, we need one more tool that implements the following transform:

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} \cos(t) & -\sin(t) \\ \sin(t) & \cos(t) \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

This takes unevenly scaled inputs, rescales them, and then rotates them. Like an ordinary rotation, it can be reduced to three lifting steps:

$$\begin{aligned}
 u_0 &= x_0 + \frac{2\cos(t) - \sqrt{2}}{\sin(t)} * x_1 \\
 y_1 &= x_1 + \frac{1}{\sqrt{2}} * \sin(t) * u_0 \\
 y_0 &= u_0 + \frac{\cos(t) - \sqrt{2}}{\sin(t)} * y_1
 \end{aligned}$$

As before, the transcendental constants may be replaced by fixed-point approximations without harming the reversibility property.

3.2. 4-Point Transform

Using the tools from the previous section, we can design a reversible integer four-point DCT approximation with uniform, orthonormal scaling. This requires 3 multiplies, 9 additions, and 2 shifts (not counting the shift and rounding offset used in the fixed-point multiplies, as these are built into the multiplier). This is significantly cheaper than the [\[BE92\]](#) approach, and the output scaling is smaller by a factor of $\sqrt{2}$, saving half a bit per sample in the lossless case. By comparison, the four-point forward DCT approximation currently in the libvpx experimental branch, which is not reversible, uses 4 multiplies and 8 additions. Its multipliers also require 28-bit accumulators, whereas this proposal can use much smaller multipliers without giving up the reversibility property. The total dynamic range expansion is 1 bit: inputs in the range $[-256, 255)$ produce transformed values in the range $[-512, 510)$. This is the smallest dynamic range expansion possible for any reversible transform constructed from mostly-linear operations. It is possible to make reversible orthogonal transforms with no dynamic range expansion by using "piecewise-linear" rotations [\[SLD04\]](#), but each step requires a large number of operations in a software implementation.

Pseudo-code for the forward transform follows:


```
Input:  x0, x1, x2, x3
Output: y0, y1, y2, y3
/* Rotate (x3, x0) by -pi/4, asymmetrically scaled output. */
t3  = x0 - x3
t0  = x0 - (t3 >> 1)
/* Rotate (x1, x2) by pi/4, asymmetrically scaled output. */
t2  = x1 + x2
t2h = t2 >> 1
t1  = t2h - x2
/* Rotate (t2, t0) by -pi/4, asymmetrically scaled input. */
y0  = t0 + t2h
y2  = y0 - t2
/* Rotate (t3, t1) by 3*pi/8, asymmetrically scaled input. */
t3  = t3 - (45*t1 + 32 >> 6)
y1  = t1 + (21*t3 + 16 >> 5)
y3  = t3 - (71*y1 + 32 >> 6)
```

Even though there are three asymmetrically scaled rotations by $\pi/4$, by careful arrangement we can share one of the shift operations (to help software implementations: shifts by a constant are basically free in hardware). This technique can be used to even greater effect in larger transforms.

The inverse transform is constructed by simply undoing each step in turn:

```
Input:  y0, y1, y2, y3
Output: x0, x1, x2, x3
/* Rotate (y3, y1) by -3*pi/8, asymmetrically scaled output. */
t3  = y3 + (71*y1 + 32 >> 6)
t1  = y1 - (21*t3 + 16 >> 5)
t3  = t3 + (45*t1 + 32 >> 6)
/* Rotate (y2, y0) by pi/4, asymmetrically scaled output. */
t2  = y0 - y2
t2h = t2 >> 1
t0  = y0 - t2h
/* Rotate (t1, t2) by -pi/4, asymmetrically scaled input. */
x2  = t2h - t1
x1  = t2 - x2
/* Rotate (x3, x0) by pi/4, asymmetrically scaled input. */
x0  = t0 - (t3 >> 1)
x3  = x0 - t3
```

Although the right shifts make this transform non-linear, we can compute "basis functions" for it by sending a vector through it with a single value set to a large constant (256 was used here), and the rest of the values set to zero. The true basis functions for a four-point DCT (up to five digits) are

Terriberry

Expires April 18, 2013

[Page 14]

```

[ y0 ]   [ 0.50000  0.50000  0.50000  0.50000 ] [ x0 ]
[ y1 ] = [ 0.65625  0.26953 -0.26953 -0.65625 ] [ x1 ]
[ y2 ]   [ 0.50000 -0.50000 -0.50000  0.50000 ] [ x2 ]
[ y3 ]   [ 0.27344 -0.65234  0.65234 -0.27344 ] [ x3 ]

```

The corresponding basis functions for our reversible, integer DCT, computed using the approximation described above, are

```

[ y0 ]   [ 0.50000  0.50000  0.50000  0.50000 ] [ x0 ]
[ y1 ] = [ 0.65328  0.27060 -0.27060 -0.65328 ] [ x1 ]
[ y2 ]   [ 0.50000 -0.50000 -0.50000  0.50000 ] [ x2 ]
[ y3 ]   [ 0.27060 -0.65328  0.65328 -0.27060 ] [ x3 ]

```

The mean squared error (MSE) of the output, compared to a true DCT, can be computed with some assumptions about the input signal. Let G be the true DCT basis and G' be the basis for our integer approximation (computed as described above). Then the error in the transformed results is

$$e = G.x - G'.x = (G - G').x = D.x$$

where $D = (G - G')$. The MSE is then [\[Que98\]](#)

$$\begin{aligned}
 \frac{1}{N} \sum E[e^T e] &= \frac{1}{N} \sum E[x^T D^T D x] \\
 &= \frac{1}{N} \sum E[\text{tr}(D x x^T D^T)] \\
 &= \frac{1}{N} \sum E[\text{tr}(D R_{xx} D^T)]
 \end{aligned}$$

where R_{xx} is the autocorrelation matrix of the input signal. Assuming the input is a zero-mean, first-order autoregressive (AR(1)) process gives an autocorrelation matrix of

$$R_{xx}[i, j] = \rho^{|i - j|}$$

for some correlation coefficient ρ . A value of $\rho = 0.95$ is typical for image compression applications. Smaller values are more normal for motion-compensated frame differences, but this makes surprisingly little difference in transform design. Using the above procedure, the theoretical MSE of this approximation is $1.230E-6$,

which is below the level of the truncation error introduced by the right shift operations. This suggests the dynamic range of the input would have to be more than 20 bits before it became worthwhile to increase the precision of the constants used in the multiplications.

3.3. Larger Transforms

The same techniques can be applied to construct a reversible eight-point DCT approximation with uniform scaling using 11 multiplies, 29 additions, and 6 shifts. This is the minimum number of multiplies possible for an eight-point DCT with uniform scaling [[LLM89](#)] (though an approximation to the DCT can be made simpler by replacing the multiplies with shifts and adds). By comparison, the current eight-point transform in the libvpx experimental branch uses 16 multiplications and 28 additions. The output scaling is currently a factor of a $\sqrt{2}$ larger than orthonormal. While it should be theoretically possible to reduce this down to the orthonormal scaling, it appears to require additional multiplies, and means that the DC coefficient would no longer differ from a simple sum of the inputs by just a power of two. These trade-offs may ultimately be worthwhile to improve the lossless compression performance. With the current scaling, the dynamic range expansion is 2 bits, and the MSE is 1.689E-5.

Similarly, we have also constructed a reversible sixteen-point DCT approximation with uniform orthonormal scaling using 32 multiplies, 83 additions, and 17 shifts. This is just 1 multiply and 2 additions more than the (non-reversible, non-integer, but uniformly scaled) factorization in [[LLM89](#)]. The experimental branch of libvpx does not currently have a sixteen-point integer transform to compare with, though it has a floating-point transform which requires 46 multiplies (not counting factors of 2) and 72 additions (plus some scalings by factors of 2 and some negations). The dynamic range expansion is only 2 bits (the same as the eight-point transform), and the MSE is 7.444E-5.

Code for both of these transforms (as well as the four-point transform) is available in the development repository listed in [Section 4](#).

4. Development Repository

The tools presented here were developed as part of Xiph.Org's Daala project. They are available, along with many others in greater and lesser states of maturity, in the Daala git repository at <https://git.xiph.org/daala.git>. See <https://xiph.org/daala/> for more information.

5. IANA Considerations

This document has no actions for IANA.

6. Acknowledgments

Thanks to Nathan Egge, Gregory Maxwell, and Jean-Marc Valin for their assistance in the experimentation and in preparing this draft.

7. Informative References

- [RFC6386] Bankoski, J., Koleszar, J., Quillio, L., Salonen, J., Wilkins, P., and Y. Xu, "VP8 Data Format and Decoding Guide", [RFC 6386](#), November 2011.
- [RFC6716] Valin, JM., Vos, K., and T. Terriberry, "Definition of the Opus Audio Codec", [RFC 6716](#), September 2012.
- [BE92] Bruekers, F. and A. van den Enden, "New Networks for Perfect Inversion and Perfect Reconstruction", IEEE Journal on Selected Areas in Communication 10(1):129--137, January 1992.
- [LLM89] Loeffler, C., Ligtenberg, A., and G. Moschytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications", Proc. Acoustics, Speech, and Signal Processing (ICASSP'89) vol. 2, pp. 988--991, May 1989.
- [Pas76] Pasco, R., "Source Coding Algorithms for Fast Data Compression", Ph.D. Thesis Dept. of Electrical Engineering, Stanford University, May 1976.
- [PKA69] Pratt, W., Kane, J., and H. Andrews, "Hadamard Transform Image Coding", Proc. IEEE 57(1):58--68, Jan 1969.
- [Que98] de Queiroz, R., "On Unitary Transform Approximations", IEEE Signal Processing Letters 5(2):46--47, Feb 1998.
- [SLD04] Senecal, J., Lindstrom, P., and M. Duchaineau, "An Improved N-Bit to N-Bit Reversible Haar-Like Transform", Proc. of the 12th Pacific Conference on Computer Graphics and Applications (PG'04) pp. 371--380, October 2004.
- [SM98] Stuiver, L. and A. Moffat, "Piecewise Integer Mapping for Arithmetic Coding", Proc. of the 17th IEEE Data Compression Conference (DCC'98) pp. 1--10, March/April 1998.
- [TSSRM08] Tu, C., Srinivasan, S., Sullivan, G., Regunathan, S., and H. Malvar, "Low-complexity Hierarchical Lapped Transform for Lossy-to-Lossless Image Coding in JPEG XR/HD Photo", Applications of Digital Image Processing XXXI vol 7073, August 2008.

Author's Address

Timothy B. Terriberry
Mozilla Corporation
650 Castro Street
Mountain View, CA 94041
USA

Phone: +1 650 903-0800
Email: tterribe@xiph.org