DECOMPRESSION OF CORRUPT JPEG2000 CODESTREAMS

Ali Bilgin, Zhenyu Wu, and Michael W. Marcellin

Dept. of Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721.

ABSTRACT

In this paper, we investigate the error resilience properties of JPEG2000. Through a careful analysis of the structure of JPEG2000 codestreams, we identify the dependencies among coding passes of a codeblock. In our analysis, we consider the effects of mode variations provided by the standard for low-memory, low-complexity implementations. The proposed methods are derived using the existing dependency structure of coding passes and do not require a substantial increase in computational capabilities of the decoder. Experimental results indicate that these methods can improve error resilience performance substantially.

1. INTRODUCTION

With the explosive growth of multimedia applications, the transmission of multimedia products over lossy communication channels such as the Internet or wireless networks has become an important area of research. When compressed images are transmitted over such channels, the compressed codestreams received at the decoder can contain errors. It is highly desirable that the decoder be able to reconstruct images using these noisy codestreams.

Many image compression methods rely on efficient entropy coders that are dependent on the state of the system. Such coders are very sensitive to errors in the codestream. Thus, the operation of high performance image codecs over lossy communication channels requires careful design to avoid complete failure when the codestream gets corrupted.

JPEG2000 is the newest international image compression standard [1, 2]. It offers state-of-the-art compression performance as well as improved functionality over previous compression standards. In this paper, we analyze the error resilience properties of JPEG2000. The JPEG2000 standard offers several mechanisms to combat errors in the codestream. Through a careful analysis of the encoding and decoding procedures, we identify sections of the codestream that can be salvaged when errors occur. We also explore the effects of external error detection mechanisms on the error resilience properties of JPEG2000. The JPEG2000 standard offers some mode variations that were included to enable low-complexity, low-memory implementations. We illustrate how these mode variations can affect error resilience.

This paper is organized as follows: In the next section, we provide a brief overview of JPEG2000. In Section 3, error resilience properties of the JPEG2000 standard are analyzed. We provide experimental results in Section 4.

2. OVERVIEW OF JPEG2000

In this section, we provide an overview of the JPEG2000 algorithm. Our goal here is not to provide a comprehensive overview. We intend to review only those properties and features that are needed to understand the remainder of the paper. A more balanced review can be found in [3]. For a comprehensive overview of JPEG2000, the interested reader is referred to [1, 2].

In JPEG2000, the input image is first divided into non-overlapping rectangular tiles. If the image has multiple components, an optional component transform can be applied to decorrelate the components. The samples of each component that fall into a particular tile are referred to as a *tile-component.* Each tile-component is then transformed using a wavelet transform and the wavelet subbands are partitioned into several different geometric structures. These geometric structures are instrumental in enabling low memory implementations and providing spatial random access. As we will discuss later, they also contribute to the error resilience of the codestream.

The smallest geometric structure in JPEG2000 is the *codeblock.* Codeblocks are formed by partitioning the wavelet subbands, and the codeblocks of particular resolutions are grouped together to form *precincts*. Once the wavelet subbands are quantized, each codeblock is compressed individually using a bitplane coder. The bitplane coder makes three passes over each bitplane of a codeblock. These passes are referred to as *coding passes*. The compressed data from each codeblock can be regarded as an embedded bitstream. The codestream is then comprised of a different number of coding passes from each individual codeblock bitstream.

The first pass in a new bitplane is called the significance propagation pass. Let us define $\sigma[\mathbf{n}]$ as the " significance state" of a sample at location $\mathbf{n} = [n_1, n_2]$ during the coding process. $\sigma[\mathbf{n}]$ is reset to zero for all samples at the start of the coding process and it is set to one as soon as the first non-zero magnitude bit of the sample is coded. A bit is coded in the significance propagation pass if its location is not significant, but at least one of its eight-connected neighbors is significant. If the bit that is coded is 1, the sign bit of the current sample is coded and $\sigma[\mathbf{n}]$ is set to 1 immediately. The scan pattern followed for the coding of bitplanes, within each codeblock (in all subbands), is shown in Figure 1. This scan pattern is followed in each of the three coding passes.

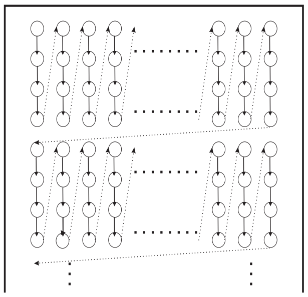


Figure 1: Scan pattern for bitplane coding.

The bits of each coding pass are arithmetically coded using context models depending on the coding pass and the subband type. For significance coding, 9 different contexts are used. The context label $\kappa^{sig}[\mathbf{n}]$ is dependent on the significance states of the eight-connected neighbors of the current bit. $\kappa^{sig}[\mathbf{n}]$ then determines the probability estimate that will be used in arithmetic coding.

The second pass is the magnitude refinement pass. In this pass, all bits from locations that became significant in a previous bitplane are coded. As the name implies, this pass refines the magnitudes of the samples that became significant in previous bitplanes. The magnitude refinement context label, $\kappa^{mag}[\mathbf{n}]$, is selected based on $\kappa^{sig}[\mathbf{n}]$ and the value of the significance state variable $\sigma[\mathbf{n}]$ delayed by one bitplane, $\overline{\sigma}[\mathbf{n}]$.

The third and final pass is the clean-up pass, which takes care of any bits not coded in the first two passes. By definition, this pass is a " significance" pass, so significance coding, as described for the significance propagation pass above, is used to code the samples in this pass. Unlike the significance propagation pass, however, a run coding mode may also occur in this pass. Run coding occurs when all four locations in a column of the scan (see Figure 1) are insignificant and each has only insignificant neighbors. A single bit is then coded to indicate whether the column is identically zero or not. If not, the length of the zero run (0 to 3) is coded, reverting to the " normal" bit-by-bit coding for the location immediately following the 1 that terminated the zero run.

For the purposes of forming the codestream, compressed data from each precinct are arranged to form *packets*. Packets play an important role in the organization of data within JPEG2000 codestream. Each packet contains a header and a body. The packet header contains information about the contribution of each codeblock in the precinct into the packet, and the body contains coding passes of codeblocks. Packets that belong to a particular tile are grouped together to form a *tile-stream*, and tile-streams are grouped together to form the JPEG2000 codestream. Similar to packets, tile-streams are comprised of a header and a body. It is possible to break tile-streams into multiple *tile-parts*. In this case, the first tile-part contains a tile header and the remaining tile-parts contain tile-part headers. Tile-parts allow the progression concepts to be extended to the entire image. There is also a main header at the beginning of the codestream. The structure of a JPEG2000 codestream is illustrated in Figure 2. The EOC marker denotes the end of the codestream.

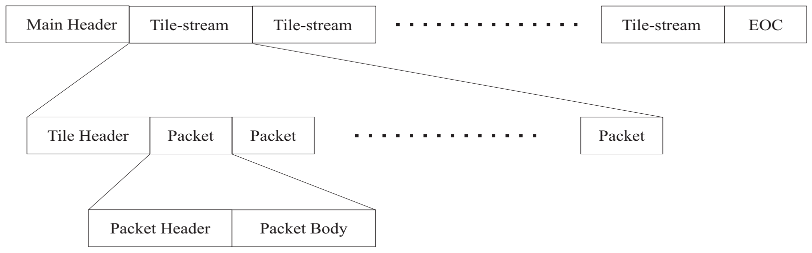


Figure 2: A simple JPEG2000 codestream.

3. ERROR RESILIENCE OF JPEG2000 CODESTREAMS

Entropy coding in JPEG2000 is achieved using a context-based arithmetic bitplane coder. The operation of this coder is highly dependent on the state of the system, and it is crucial to maintain synchronization between the encoder and the decoder. A single bit error in the arithmetically coded segments of the bitstream can destroy this synchronization, and could result in erroneous decompression. To combat this problem, several error resilience tools are provided within JPEG2000.

The partitioning of the codestream into different segments is the first line of defense. In terms of error resilience, this partitioning aims to isolate errors made in one segment to that particular segment, and prevent error propagation across segment boundaries. This isolation occurs at several levels, since the codestream is organized in a hierarchical fashion as illustrated in Figure 2.

JPEG2000 provides a mechanism where the packet headers can be extracted from every packet and stored in tile-part headers or the main header. This is referred to as *packed packet headers*. Packed packet headers can provide significant advantages for error resilience if the main and tile-part headers can be transmitted in an error-free fashion. Since each packet header contains the lengths in bytes of all coding passes in that packet (when used with the RESTART mode, described in the next section), the decoder can utilize this information to isolate errors.

3.1. Error Detection and Resynchronization

To complement the hierarchical data partitioning, JPEG2000 provides several mechanisms for error detection and resynchronization. One such mechanism is the byte-stuffing procedure in JPEG2000. Through the use of this byte-stuffing procedure, the JPEG2000 arithmetic coder does not produce certain values (**0xFF90** through **0xFFFF**) inside coding passes. These values, called *delimiting marker codes*, are reserved for codestream markers. Unexpected detection of one of these values would indicate that an error has occurred.

Some of the error detection and resynchronization mechanisms are enabled by mode variations. These alternatives to the default mode of the codec allow additional capabilities in exchange for some small losses in compression efficiency. Mode variations are controlled by flags that are signaled inside headers. Although some of these modes were not designed with the intent to improve error resilience, we will discuss how each mode affects error resilience.

When the RESET mode is used, the context states (i.e., the probabilities used in arithmetic coding) are reset to their initial values at the end of each coding pass. If the RESET switch is not specified, this initialization occurs only prior to the first coding pass of a codeblock.

The RESTART switch causes the MQ arithmetic coder to be restarted at the beginning of each coding pass. Thus, when this mode is utilized, every coding pass has its own MQ codeword segment. It should be noted that the length of each of these segments is signaled in the packet header. If the RESTART switch is not specified, the arithmetic coder is restarted only at the beginning of each codeblock.

The goal of the BYPASS mode is to provide reduced complexity at high rates with little loss in compression efficiency. When this mode is selected, the MQ coder is bypassed during the first (significance propagation) and second (magnitude refinement) coding passes (after the first 10 coding passes of a codeblock), and the binary symbols are stored in raw segments. To avoid the appearance of the delimiting marker codes inside these raw segments, a simple bit-stuffing procedure is adopted.

It is also possible to use the ERTERM mode. When the ERTERM mode is utilized, the encoder adopts a predictable termination policy for each MQ and/or raw codeword segment. Then, the decoder can detect that an error has occurred in an arithmetically coded codeword segment.

The CAUSAL mode was defined by the standard to allow parallel processing of coding passes, when used in conjunction with RESET and RESTART modes. However, as we will see in the following sections, it also has implications in error resilience. When the CAUSAL mode is utilized, the context formation process is modified slightly. Recall that the scan pattern of samples within a codeblock, as shown in Figure 1, was a 4-sample stripe-oriented scan. In CAUSAL mode, the samples within a given stripe are encoded without depending on the values of future stripes. Thus, when the contexts are formed, all samples from future stripes are treated as insignificant.

3.2. Data Recovery

As discussed in Section 2, entropy coding in JPEG2000 is performed independently on each codeblock. Three passes are performed over each bitplane of a codeblock. This is illustrated in Figure 3 [1]. When a decoder detects an error in a given coding pass, current practice is to discard the current and all future coding passes of the current codeblock. In this section, we illustrate that this practice results in suboptimal performance, and more information can be salvaged from the codestream. We observe that even though there might be errors in earlier coding passes, some (or all) future coding passes might still be completely or partially decodable, depending on certain modes used during the creation of the codestream.

We now analyze what parts of the codeblock bitstream can be recovered under different scenarios. It is important to note that external information on the locations of the errors (when available) can be utilized during decoding. Such information can be gathered from channel codes that are applied to a JPEG2000 codestream prior to transmission. For packet-switched networks, such as the internet, this information may be obtained from the transport layer. Of particular interest is the location of the first error within a coding pass. If this information is available, partial decompression of current and future coding passes can be enhanced.

We break our analysis into two sections. We first consider the case where no external information on the location of errors is available. In this case, error detection is performed using the

[1] It is important to mention that although the bitstreams for each coding pass are marked separately in Figure 3, they are not seperately identifiable in this fashion in the codestream, unless the RESTART marker is utilized.

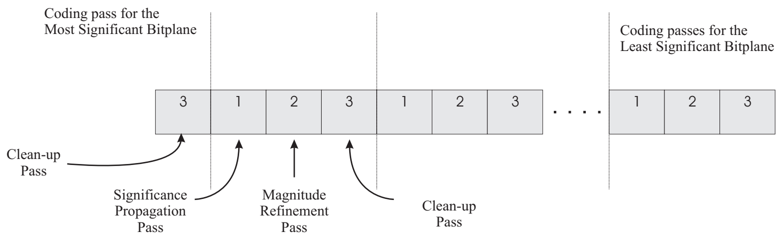

Coding pass for the Most Significant Bitplane
Coding passes for the Least Significant Bitplane
3
1
2
3
1
2
3
. . . .
1
2
3
Clean-up Pass
Significance Propagation Pass
Magnitude Refinement Pass
Clean-up Pass

Figure 3: A codeblock bitstream.

internal mechanisms of JPEG2000. We then consider external error detection. It should be noted that in all cases, we assume the RESTART switch has been set at the encoder. This will allow us to identify individual coding passes and parse a codeblock bitstream into individual coding passes at the decoder.

3.2.1. Without External Error Detection

Let us consider four different scenarios, depending on whether the RESET and the BYPASS switches were set to create the codestream at the encoder. For each of the four cases, let us consider errors that have occurred in significance propagation, magnitude refinement, and clean-up passes separately. This results in twelve different cases which we analyze individually:

Error in Significance Propagation {*RESET=False, BYPASS=False*} or {*RESET=True, BYPASS =False*} : When there is an error in a significance propagation pass, κ^{sig} and $\sigma[j]$ get corrupt. Thus, none of the following coding passes can be decoded. See Figure 4-(a).

Error in Significance Propagation {*RESET=False, BYPASS=True*} or {*RESET=True, BYPASS =True*} : In this case, κ^{sig} and $\sigma[j]$ get corrupt. Although no future significance propagation or clean-up passes can be decoded, one more magnitude refinement pass can be used since the bypass option turns off arithmetic coding. See Figure 4-(b).

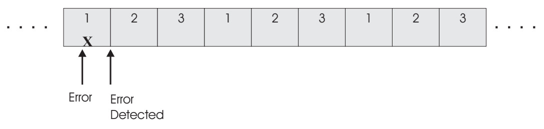
Error in Magnitude Refinement {*RESET=False, BYPASS=False*} : In this case although κ^{sig} will not be corrupt, the state information used in arithmetic coding for magnitude refinement passes will be corrupt. Thus, all following magnitude refinement coding passes should be discarded. However, the two other types of coding passes can be decoded. See Figure 4-(c).

Error in Magnitude Refinement {*RESET=False, BYPASS=True*} or {*RESET=True, BYPASS =True*} : Since arithmetic coding is not utilized and the bits are stored raw, a bit error only affects a single sample [2]. Thus, the decoder should continue decoding the current and all future coding passes. See Figure 4-(d).

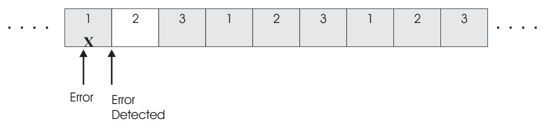
Error in Magnitude Refinement {*RESET=True, BYPASS=False*} : An error in magnitude refinement pass will corrupt κ^{mag}. Since neither the significance propagation nor the clean-up passes are dependent on κ^{mag}, all future such passes can be decoded. Furthermore, since the reset switch will reinitialize the κ^{mag} for the next magnitude refinement coding pass, all future magnitude refinement coding passes can be decoded as well. See Figure 4-(e).

Error in Clean-up: κ^{sig} and $\sigma[j]$ get corrupt and all following coding passes should be discarded. See Figure 4-(f).

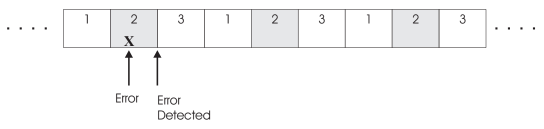
[2]Except for the case where byte stuffing needs to be used to avoid producing restricted marker codes



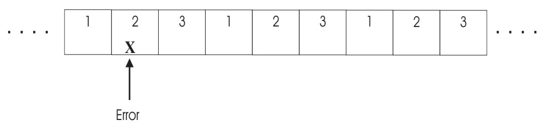
(a) Error in Significance Propagation $\{$*RESET=False, BYPASS=False*$\}$ or $\{$*RESET=True, BYPASS=False*$\}$.



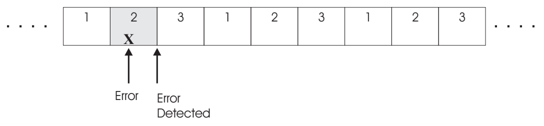
(b) Error in Significance Propagation $\{$*RESET=False, BYPASS=True*$\}$ or $\{$*RESET=True, BYPASS=True*$\}$.



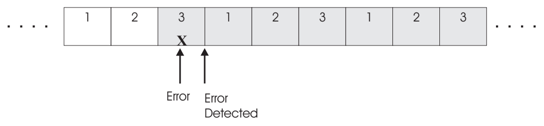
(c) Error in Magnitude Refinement $\{$*RESET=False, BYPASS=False*$\}$.



(d) Error in Magnitude Refinement $\{$*RESET=False, BYPASS=True*$\}$ or $\{$*RESET=True, BYPASS=True*$\}$.



(e) Error in Magnitude Refinement $\{$*RESET=True, BYPASS=False*$\}$.



(f) Error in Clean-up

Figure 4: Coding pass dependencies (Without External Error Detection). The portions of the bitstream that need to be discarded are shaded.

3.2.2. With External Error Detection

When an external mechanism for error detection is available, the information on the locations of the errors can be utilized to improve the performance of JPEG2000. The location of the first error within a coding pass is particularly important. If this location is known or can be estimated, the error free section of the coding pass can be decompressed. We refer to this as *partial decoding*. Furthermore, the sections of future coding passes that are only dependent on the error free portion of the current coding pass can also be decompressed.

Similar to the previous section, we observe that partial decoding is dependent on the switches used to create the codestream. In addition to the RESTART, RESET, and BYPASS switches, the CAUSAL switch also plays an important role in this case. We assume that the RESTART switch is used for all codestreams, and analyze the effects of the RESET and BYPASS switches on partial decoding first. We will then analyze how the CAUSAL switch affects partial decoding.

Error in Significance Propagation $\{$*RESET=False, BYPASS=False*$\}$: When there is an error in a significance propagation pass, κ^{sig} and $\sigma[j]$ get corrupt. Although none of the future coding passes can be decoded completely, one more magnitude refinement pass can be decoded partially. The portion of the magnitude refinement pass that can be decoded is determined by the uncorrupt portion of κ^{sig} and $\sigma[j]$. See Figure 5-(a).

Error in Significance Propagation $\{$*RESET=False, BYPASS=True*$\}$: In this case, $\sigma[j]$ get corrupt. Although no future significance propagation or clean-up passes can be decoded completely, one more magnitude refinement pass can be decoded completely. Furthermore, the following clean-up, significance propagation, and magnitude refinement passes can be decoded partially. See Figure

5-(b).

Error in Significance Propagation $\{$*RESET=True, BYPASS=False*$\}$: κ^{sig} and $\sigma[j]$ get corrupt. However, since the arithmetic coding states are reset for every coding pass, partial decoding of all following coding passes is possible. See Figure 5-(c).

Error in Significance Propagation $\{$*RESET=True, BYPASS=True*$\}$: In this case, κ^{sig} and $\sigma[j]$ get corrupt. One more magnitude refinement pass can be decoded completely and all remaining passes can be decoded partially. See Figure 5-(d).

Error in Magnitude Refinement$\{$*RESET=False, BYPASS=False*$\}$: In this case although κ^{sig} will not be corrupt, the state information used in arithmetic coding for magnitude refinement passes will be corrupt. Thus, all following magnitude refinement coding passes should be discarded. However, the two other types of coding passes can be decoded completely. See Figure 5-(e).

Error in Magnitude Refinement $\{$*RESET=False, BYPASS=True*$\}$ or $\{$*RESET=True, BYPASS =True*$\}$: Since arithmetic coding is not utilized and the bits are stored raw, a bit error only affects a single sample (Except for the case, where byte stuffing needs to be used to avoid producing restricted marker codes). Thus, the decoder should continue decoding the current and all future coding passes. See Figure 5-(f).

Error in Magnitude Refinement$\{$*RESET=True, BYPASS=False*$\}$: An error in magnitude refinement pass will corrupt κ^{mag}. Since neither the significance propagation nor the clean-up passes are dependent on κ^{mag}, all such passes can be decoded. Furthermore, since the reset switch will reinitialize the κ^{mag} for the next magnitude refinement coding pass, all future magnitude refinement coding passes can be decoded as well. See Figure 5-(g).

Error in Clean-up $\{$*RESET=False, BYPASS=False*$\}$: When there is an error in a clean-up pass, κ^{sig} and $\sigma[j]$ get corrupt. Thus, all following coding passes should be discarded. See Figure 5-(h).

Error in Clean-up $\{$*RESET=False, BYPASS=True*$\}$: Although κ^{sig} or $\sigma[j]$ get corrupt, one more significance propagation and magnitude refinement coding passes can be partially decoded. See Figure 5-(i).

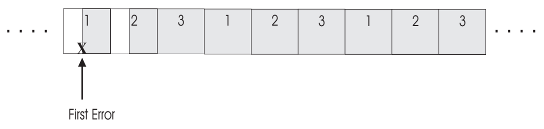
Error in Clean-up $\{$*RESET=True, BYPASS=False*$\}$ or $\{$*RESET=True, BYPASS=True*$\}$: κ^{sig} and $\sigma[j]$ get corrupt. However, since the arithmetic coding states are reset for every coding pass, partial decoding of all following coding passes is possible. See Figure 5-(j).

In all the above cases, the portion of the coding passes that can be salvaged is dependent on how the error propagates from one coding pass to another. This propagation is dependent on the order in which samples are visited during coding. Error propagation is also dependent on whether the codestream has been created using *CAUSAL=True* or *CAUSAL=False*. When the CAUSAL switch is turned on, the samples in future stripes are treated as insignificant during context formation. This affects error propagation. Thus, when an error occurs, the error propagation is much slower if *CAUSAL=True* versus when *CAUSAL=False*. The samples that can be salvaged in future coding passes can be identified through a careful analysis of this error propagation.

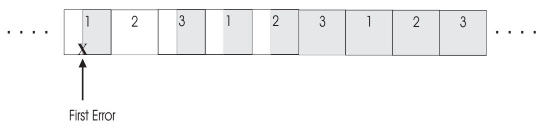
4. EXPERIMENTAL RESULTS

In this section, we provide experimental results to compare the performances of the presented methods. In our experiments, we have used the 512×512 Barbara, Goldhill, and Lenna images. We have used Kakadu as our JPEG2000 codec [5] with 64×64 codeblocks, and 128×128 precincts. In all the simulations, PPM markers were utilized, and the main header and the tile header were protected from errors. Each simulation was performed 1000 times over a Binary Symmetric Channel (BSC) with BER = 10^{-4}. We consider four different methods:

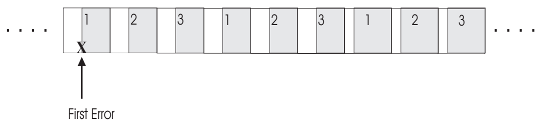
The first method is the one that is implemented in Kakadu [5]. In this method, the decoder is dependent on the internal error detection mechanisms of JPEG2000. Once an error is detected



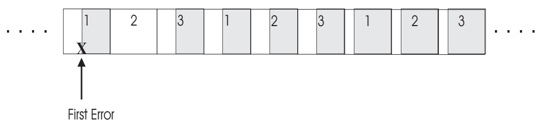
(a)
Error in Significance Propagation {*RESET=False, BYPASS=False*}.



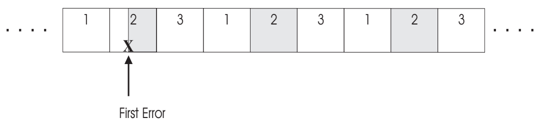
(b)
Error in Significance Propagation {*RESET=False, BYPASS=True*}.



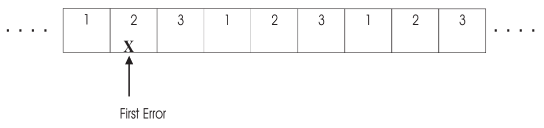
(c) Error in Significance Propagation {*RESET=True, BYPASS=False*}.



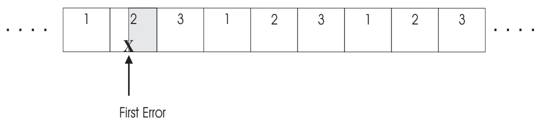
(d) Error in Significance Propagation {*RESET=True, BYPASS=True*}.



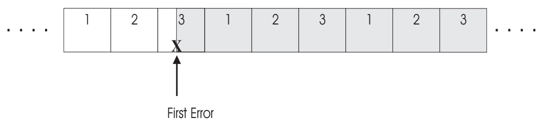
(e) Error in Magnitude Refinement {*RESET=False, BYPASS=False*}.



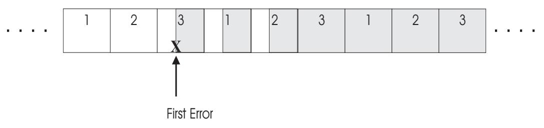
(f) Error in Magnitude Refinement {*RESET=False, BYPASS=True*} or {*RESET=True, BYPASS=True*}.



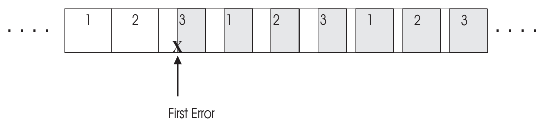
(g) Error in Magnitude Refinement {*RESET=True, BYPASS=False*}.



(h) Error in Clean-up {*RESET=False, BYPASS=False*}.



(i) Error in Clean-up {*RESET=False, BYPASS=True*}.



(j)
Error in Clean-up {*RESET=True, BYPASS=False*} or {*RESET=True, BYPASS=True*}.

Figure 5: Coding pass dependencies (With External Error Detection). The portions of the bitstream that need to be discarded are shaded.

in a coding pass, all remaining coding passes from the current codeblock are discarded. We refer to this method with the acronym KDU. The second method is referred to with the acronym ECP. Here, the decoder is relying on external error detection mechanisms to locate the erroneous coding passes. Once an error is found in a given coding pass, the decoder discards the remaining coding passes from that codeblock. The third method is referred to with the acronym PCP. In this method, external error detection mechanisms are utilized to detect the first byte in a codeblock codestream that contains an error. The decoder consumes all the bytes up to the erroneous one, and discards the remaining codeblock codestream. Notice that this method can decode a coding pass partially. The fourth and the final method is referred to with the acronym CCP. This method involves all of the data recovery ideas presented earlier. Once the decoder identifies the initial error in a codeblock codestream through external means, the decoding is performed up to the byte containing the first error. Then the decoder performs an analysis to identify which of the remaining coding passes can be decoded. This method allows partial decoding of future coding passes by taking error propagation into account.

We first compare the performance of different methods. Table 1 presents the average Peak Signal-to-Noise Ratio (PSNR) performance of different methods. In these simulations, the JPEG2000 codestreams were generated using the BYPASS, RESET, RESTART, CAUSAL, and ERTERM modes.

Table 1: Average PSNR performance of different methods over BSC with BER = 10^{-4}.

Image	Rate (bpp)	KDU	ECP	PCP	CCP
Barbara	0.25	25.49	25.43	25.84	26.28
	0.5	26.98	26.93	27.52	28.34
	1.0	27.76	27.64	28.41	29.57
Goldhill	0.25	27.84	27.83	28.15	28.62
	0.5	29.08	29.01	29.50	30.26
	1.0	29.88	29.76	30.39	31.49
Lenna	0.25	29.59	29.53	29.95	30.70
	0.5	30.54	30.43	30.98	32.00
	1.0	31.29	31.08	31.75	33.02

As illustrated in Table 1, the proposed methods can provide significant gains in average PSNR. It is worth mentioning that for most of the cases, the KDU and ECP simulations produced identical results. This suggests that the internal error detection mechanisms of JPEG2000 function well. However, notice that the ECP results are slightly lower than those of KDU. When an error occurs in the last bits of a codeword used for byte alignment, ECP identifes these as errors and stops further decompression. Since these " errors" are not detected by KDU, decompression continues resulting in slight increase in performance. It is possible for ECP (as well as PCP and CCP) to detect this error, although we have not yet added this detection into our software.

To analyze these results further, consider the plot presented in Figure 6. In the figure, the KDU and CCP methods are compared at 1.0 bits/pixel using the Lenna image. The plot was obtained from the same 1000 realizations that were used to generate Table 1. This plot illustrates the distribution of the PSNR gain obtained by using CCP instead of KDU. The x-axis of the plot indicates PSNR (in dB), and the y-axis indicates the cumulative percentage of realizations. From this plot, we can infer that the gain provided by CCP in these simulations has been as large as 8.6 dB. We can also see that CCP provided a gain greater than 2.7 dB in roughly 20% of the cases. As mentioned in earlier, " errors" in stuffing bits that should be ignored are not ignored in the current version of our software. This results in KDU outperforming CCP in some instances as seen in the figure. We expect larger average gains when this feature is added to our software.

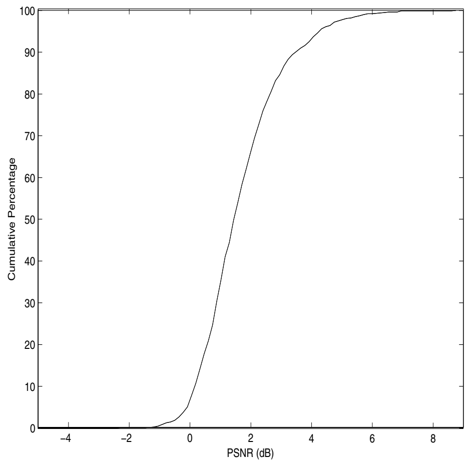


Figure 6: The distribution of the PSNR gain provided by CCP over KDU.

We present the average PSNR performance of KDU and CCP using different modes in Table 2. These results illustrate that the utilization of the CAUSAL mode can provide significant improvement in error resilience when the CCP approach is adopted. As explained earlier, this is due to the error propagation being slower in *CAUSAL = True* mode vs. *CAUSAL = False* mode.

Table 2: The average PSNR (in dB) performances of KDU and CCP over the BSC with BER = 10^{-4} using different modes.

Image	Rate (bpp)	KDU-CCP			
		RESET RESTART, ERTERM	RESET, RESTART ERTERM, CAUSAL	RESET, RESTART ERTERM, BYPASS	RESET RESTART, ERTERM CAUSAL, BYPASS
Barbara	0.25	25.54-26.21	25.53-26.39	25.52-26.10	25.48-26.28
	0.5	26.92-28.01	27.01-28.41	26.84-27.93	26.98-28.34
	1.0	27.69-29.27	27.73-29.75	27.82-29.20	27.76-29.57
Goldhill	0.25	27.86-28.53	27.83-28.65	27.95-28.54	27.84-28.62
	0.5	28.86-29.83	28.87-30.18	28.96-29.82	29.08-30.26
	1.0	29.60-30.94	29.82-31.64	29.75-30.93	29.88-31.48
Lenna	0.25	29.88-30.75	29.58-30.78	27.77-30.52	29.59-30.70
	0.5	30.61-31.82	30.62-32.34	30.78-31.80	30.54-32.00
	1.0	31.31-32.78	31.12-33.24	31.39-32.58	31.29-33.02

5. REFERENCES

[1] ISO/IEC 15444-1, " JPEG2000 Image Coding System" , 2000.

[2] D. S. Taubman and M. W. Marcellin, *JPEG2000: Image Compression Fundamentals, Practice and Standards*, Kluwer Academic Publishers, Massachusetts, 2002.

[3] M.W. Marcellin, M.J. Gormish, A. Bilgin, and M.P. Boliek, " An overview of JPEG-2000" , in *Data Compression Conference*, Mar. 2110, pp. 523– 581.

[4] ISO/IEC 14492, " Information Technology - Lossy/Lossless Coding Of Bi-Ievel Images" , 1999.

[5] *http://www.kakadusoftware.com/*.