Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Christoffer Boothby

An implementation of a compression algorithm for hyperspectral images. A novelty of the CCSDS 123.0-B-2 standard

Master's thesis in Electronic Systems Design
Supervisor: Milica Orlandic
June 2020

NTNU
Norwegian University of Science and Technology

Christoffer Boothby

An implementation of a compression algorithm for hyperspectral images. A novelty of the CCSDS 123.0-B-2 standard

Master's thesis in Electronic Systems Design
Supervisor: Milica Orlandic
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Acknowledgement

I would like to thank my supervisor Milica Orlandic for giving me and others the support throughout the project thesis and master thesis. Willing to share knowledge, help others and push for new technology is a huge benefit to the world. She has given me huge motivation to further work with complicated hardware and the huge possibilities of FPGA. I would also like to give my thanks to Sivert Bakken for the tremendous help he has been able to provide when working with this project. Being able to provide insightful help when algorithms suddenly would not work. I would like to express gratitude to Evelyn Honoré-Livermore for taking the effort to create a team and the opportunity for NTNU to have a space related project. The effort to go into the space industry is difficult and tiring and the credits should not go without recognition. Finally i would like to give my thanks to the whole HYPSO team for creating a great working environment and all the work we went through together.

Christoffer Boothby

Abstract

For many years in the space industry the usage of hyperspectral imaging has been used to observe the Earth for research and industrial uses. The usage of hyperspectral imaging allows researchers to analayze the wide spectrum of wavelengths instead of regular colors. In particular the possibility to observe algae blooms or oil spills with an hyperspectral camera. These issues gave birth to the mission Hyperspectral Smallsat for Ocean Observation at Norwegian University of Science and Technology which is to launch a cubesat into space to observe the earth using a hyperspectral camera. Due to the limited speeds of the transmission antenna on the cubesat introduces the requirement of image compression algorithms to reduce the demand on the antenna. This thesis will implement two standards developed by Consultative Committee for Space Data Systems for compression of hyperspectral images. The first standard CCSDS 123.0-B-1 published in 2015 provides a method for lossless compression. The standard is already implemented as a FPGA solution in the on-board processing unit and required a software backup solution in the programming langauge C. The second standard CCSDS 123.0-B-2 recently published in 2019 is a new near-lossless compression algorithm of hyperspectral imaging. This thesis will focus on a software implementation of CCSDS 123.0-B-2 in the programming language C and further research the possible compression rates and image quality from this standard. With the possibility to control the amount of near-lossless in the compression algorithm allows a user to select the required quality of an image. The results from the implementation presents the possibility to achieve compression rates by using a lossless method. Achieving a typically compression rate of 2-3 when using the lossless method. By increasing the near-lossless the compression rates increases to beyond 100 when using the new hybrid encoder. A comparison with the sample adaptive encoder is presented and it peaks typically at a compression rate of 14-16. This does however come at the cost of image quality where the Peak signal-to-noise ratio (PSNR) ranges from 90 to 20 depending of the level of near-lossless, and the type of image. The implementation still requires some future work and verification to determine a functioning compression algorithm as the introduction of near-lossless creates a difficult problem of testing.

Sammendrag

I romindustrien har hyperspektrale kameraer vært kritisk for observasjoner av jorda for romforskining og industriell bruk. Bruken av hyperspektrale kameraer gir forskere muligheten å analysere det elektromagnetiske spekteret istedenfor å bruke vanlige farge kamera. Hyperspektrale kameraer gir muligheten for å observere for eksempel algevekst eller oljesøl. Disse problemene ga motivasjon for å danne prosjektet Hyperspectral Smallsat for Ocean Observation hos Norges teknisk-naturvitenskapelige universitet (NTNU) med formål å lage en satelitt og sende den til verdensrommet. Satelitten i en form av en cubesat vil observere jorden ved bruk av et hyperspektral kamera. Et problem ved å bruke cubesat er begrensningene på antennen for å overføre data til jorden. Dette gir motivasjon til å implementere en komprimerings algoritme av hyperspektrale bilder. Denne masteroppgaven vil implementere 2 standarder laget av Consultative Committee for Space Data Systems for komprimering av hyperspektrale bilder. Den første standarden CCSDS 123.0-B-1 er publisert i 2015 for tapsfri komprimering av hyperspektrale bilder. I prosjektet er denne standarden allerede implementert i form av en FPGA løsning og krevde derfor en implementasjon i programmeringspråket C som en sikkerhet. Den andre standarden CCSDS 123.0-B-2 som er nylig publisert i 2019 gir en mulighet for komprimering med tap av kvalitet. Tap av kvalitet kan gi bedre komprimeringsrater i forhold til tapsfri komprimering. I tilegg gir standarden en mulighet til å kontrollere graden av kvalitetestap for bedre kompresjonsrater. Denne masteroppgaven vil fokusere på en implementasjon av CCSDS 123.0-B-2 i programmeringsspråket C. I tilegg vil oppgaven gå dypere i de mulige kompresjonsratene og bildekvaliteten som medfører av standarden. Det esulteres i en kompresjonsrate på 2-3 ved bruk av tapsfri komprimering. Ved å øke tapet av bildekvalitet vil kompresjonsraten gå vel over 100 når det brukes den nye "hybrid encoder". I forhold til den gamle bruken av "sample adaptive encoder" som når maksimalt en kompresjonsrate på 14-16. Kvaliteten av bildene går derimot ned fra det orginale bildet til en Peak signal-to-noise ratio som vil typisk være fra 90 til 20. Dette varierer av hvor hvilken grad komprimerings tapet er valgt, og hvilket bilde som er brukt. Implementasjonen av standarden krever videre arbeid og verifisering for å kunne fastslå en korrekt komprimerings algoritme. Siden introduksjonen av å kontrollere kvalitetstapet gjør det vanskelig for verfisering om resultatene er korrekte.

Contents

E. Code Tables and Flush Tables

List of Tables

List of Figures

Acronyms

AXI Advanced eXtensible Interface.

BIL Band Interleaved by Line.

BIP Band Interleaved by Pixel.

BSQ Band Sequential.

CCSDS Consultative Committee for Space Data Systems.

CPU Central Processing Unit.

CubeDMA A direct memory access fpga core for hyperspectral cubes.

DDR Double Data Rate.

DMA Direct Memory Access.

FPGA Field-Programmable Gate Array.

HICO Hyperspectral Imager for the Costal Ocean.

HYPSO Hyperspectral Smallsat for Ocean Observation.

IP Intellectual Property.

NTNU Norwegian University of Science and Technology.

PSNR Peak signal-to-noise ratio.

SoC System on Chip.

1. Introduction

In the recent decades the cost for launching spacecrafts into space have been reduced. Launching a satellite in 1988 on an Ariane 44 rocket would cost $17900 per kg, and today SpaceX has the possibility to cut the costs down to $2720 per kg on a Falcon 9 as advertised[6]. With these reduction of costs gives the opportunity for low cost projects to launch into space. In particular the niche market of cubesats is emerging. A cubesat has the basic form of a 10cm x 10cm x 10cm cube to provide an alternative method of launching a payload into space in comparison to larger satellites. This allows multiple projects to launch in a single rocket for reducing the total project cost. However using a small satellite does come at the cost of available hardware, solar cell power generation, the maximum size of an antenna and the problems of dissipating heat in vacuum. A major challenge in a cubesat is to increase the antenna transmission speed which comes at the cost of heat generation and power usage. In addition to this there is a lack of high rate radio transmission antennas that can fit into a cubesat which makes providing high-rate transmission difficult. As such the possible speed a cubesat can typically achieve is a couple of Kb/s to 10Mbps+.[7]

With the possibility to use smallsats for science missions allowed a group called NTNU SmallSat to create a satellite for observing oceanographic phenomena. NTNU SmallSat created a mission called Hyperspectral Smallsat for Ocean Observation with the objective to observe the ocean through a hyperspectral camera. The camera allows scientists to observe the earth from different wavelength for research. In particular the observations from this camera allows to easily discover oil spills, algae blooms or boats, and much more. However the images produced from the camera will create files that can be hundreds of megabytes which is going to take a lot of time to download with a slow antenna. Primarily because the HYPSO satellite only has the possibility of a couple of Mbps. To offload the requirements on the antenna and to reduce the size of hyperspectral images is the usage of compression algorithms for high-speed computation. The sole goal of reducing the captured hyperspectral images to a file that is smaller in size than the original. By using compression algorithm on large amounts of data requires for a high-speed hardware but at a low cost. A good solution to these problems was the usage of System on Chip with an Field-Programmable Gate Array built inside. HYPSO uses an Zynq-7000 SoC as the on-board processing unit for the satellite. As the hardware is defined for the mission it was still a requirement to use developed compression algorithms. By using a compression algorithm on the satellite allows the possibility to transfer files at a reduced time. From this particular issue allowed a group called Consultative Committee for Space Data Systems to develop the standard CCSDS 123 for compression of hyper-spectral images. Two standards is published by CCSDS regarding compression of hyper-spectral. CCSDS 123.0-B-1 which is published in 2015 designed for compression of lossless images.[3] The second standard published in 2019 called CCSDS 123.0-B-2 is a new standard to achieve higher compression in the form of near-lossless.[4] This thesis will implement both standards in the programming language C for research usage, and the study of the new compression algorithm CCSDS 123.0-B-2 in future missions. In a previous master thesis by Johan Fjeldtvedt developed a FPGA version of CCSDS 123.0-B-1 for usage on the on-board processing unit in HYPSO.[2] If this would fail then the mission could not continue. This possible issue grew the requirement to create a backup solution as a C version

of CCSDS 123.0-B-1.

1.1. Chapter outline

Three main problems has been focused in this thesis and will be split into its separate chapters. The three problems detailed in this thesis is the implementation of CCSDS 123.0-B-1, and the improvements of the project thesis in [1] of CCSDS 123.0-B-2. The third problem is the required changes and improvements to the system for HYPSO. This thesis is structured as:

- **Chapter 2** will describe the necessary background to understand the CCSDS 123 standards and the project thesis which implemented CCSDS 123.0-B-2.
- **Chapter 3** will describe the C implementation of CCSDS 123.0-B-1
- **Chapter 4** details the improvement and fixes for completing the software that was done in this thesis.
- **Chapter 5** will detail the improvements and changes that was required for the HYPSO system.
- **Chapter 6** presents the results of the implementations and the verification on the software.
- **Chapter 7** is the final chapter where a discussion of the results will be presented with an conclusion and the required future work of the implementations.

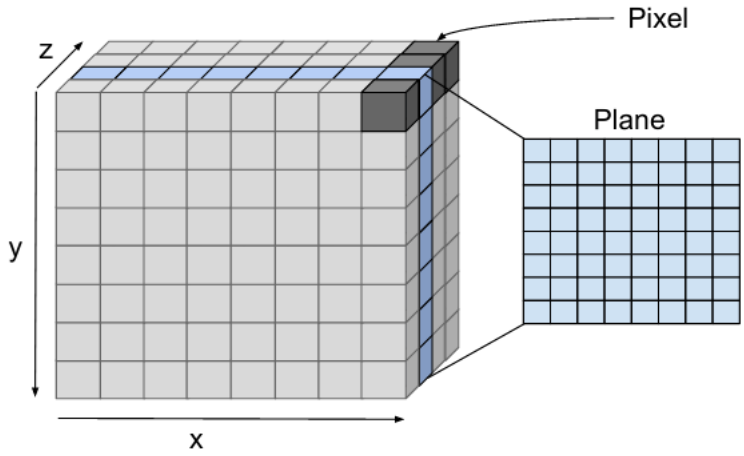
2. CCSDS 123 Project Thesis

As this master thesis is a continuation of a project thesis in [1] and this chapter will summarise the background information and the implementation of the project thesis. The project thesis implemented a compression algorithm of hyper-spectral images using the compression algorithm CCSDS 123. As the software was not completely finished in the project thesis, Chapter 4 will present the improvements done in this thesis for completing the software. Consultative Committee for Space Data Systems is an organisation founded by the major space agencies to develop standards for space data and informations systems [8]. The organistation has among the many publications, published a hyper-spectral compression algorithms for compressing the total data usage of the images. The compression algorithm has two versions that has been published in 2012 and 2019; CCSDS 123.0-B-1 and CCSDS 123.0-B-2. CCSDS 123 Issue 1 or the official name CCSDS 123.0-B-1 is a lossless compression algorithm that has the objective of not losing any information of the original image. CCSDS 123 Issue 2 or the official name CCSDS 123.0-B-2 is a near-lossless compression algorithm that will introduce lossy into the compressed images to achieve higher compression rates. This has the great benefit of reducing the demand of high data-rates on a transmission medium. Section 2.1 will present a summary of the background information regarding CCSDS 123 Issue 1 which is written in more detail in the project thesis [1] or the original published paper [9]. While Issue 1 and Issue 2 is similar there are some differences that are needed to be explained to understand the new additions.

2.1. CCSDS 123 Issue 1

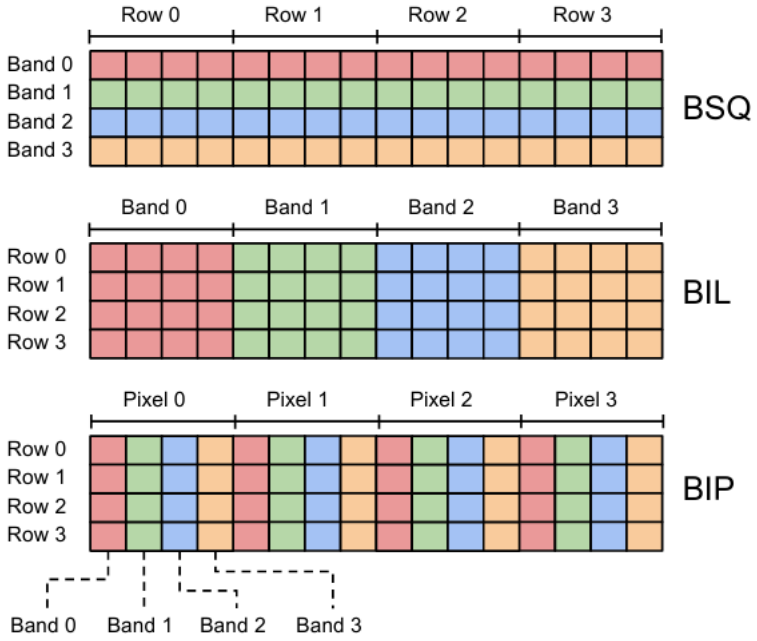
The CCSDS 123 Issue 1 standard will use the original CCSDS 123 Issue 1 as its main source of information, this source is the official published standard in [3]. Spectral images will be covering three dimensional images along the x, y and z axis where the x and y coordinates will be the spatial coordinates in an image. The z axis will be the spectral information of an image. Such a cube can be shown in Figure 2.1. This thesis will use the words sample or pixel to reference each individual coordinate in this cube. A sample is defined as $s_{z,y,x}$ where s is the image cube, and a pixel will also be defining this sample. The usage of the term pixel will differ from traditional use as it defines coordinates on a computer monitor. These pixels do represent the Red, Green and Blue (RGB) color scheme, or Cyan, Magneta, Yellow and Key(CMYK) to represent color. One could extract the wavelengths that represents the colors from a spectral cube and present them as such.

Figure 2.1.: Cube Dimensions of an image cube [2]



Images can be stored and computed in three different orders: Band Interleaved by Pixel which will store all the bands z for each pixel x, y in order, Band Interleaved by Line will store a spatial row x for each band for all rows and Band Sequential the spatial for each band are stored in order in similar to a normal image. This can be visually shown in Figure 2.2.

Figure 2.2.: Illustration of the three different sample orderings of spectral images [2].



As it is mentioned the cubes will represent coordinates in a three dimensional cube. Each of the axis x, y, z is limited constrained within $0 \leq x \leq N_x - 1$, $0 \leq y \leq N_y - 1$ and $0 \leq z \leq N_z - 1$, where each dimension N_x, N_y and N_z is specified in the integer range of $1 \leq N \leq 2^{16}$ [3]. Each sample is also constrained to a fixed bit-depth or also known as color-depth which defines how many bits to represent each individual sample. The bit size or also known as dynamic range are defined by the user specified parameter *D* in the range of $2 \leq D \leq 16$. The dynamic range is often constrained by the camera that will be capturing the images. The captured image samples $s_{z,y,x}$ can be signed or unsigned integers where the minimum value of a sample is s_{min}, middle value is s_{mid} and the maximum value s_{max} defines the range of the camera values. The unsigned minimum, middle and maximum value is defined by Equation 2.1 and for signed minimum, middle and maximum value this is defined by Equation 2.2.

$$s_{min} = 0, s_{max} = 2^{D-1}, s_{mid} = 2^D - 1 \tag{2.1}$$

$$s_{min} = -2^{D-1}, s_{max} = 2^{D-1} - 1, s_{mid} = 0 \tag{2.2}$$

2.1.1. Predictor

Predicition involves the method to calculate the prediction residual $\hat{s}_{z,y,x}$ and the mapped prediction residual $\delta_{z,y,x}$ from the input image $s_{z,y,x}$. The calculation of these values is done by calculating a local sum $\sigma_{z,y,x}$ which will be described in section 2.1.1.1. The local sum values are used in the calculation of local differences $d_{z,y,x}$, and the directional local differences $d^N_{z,y,x}$, $d^W_{z,y,x}$ and $d^{NW}_{z,y,x}$ as this will be explained in section 2.1.1.2. CCSDS 123 presents two modes; **reduced** and **full** for compression. Under Full mode the central differences and directional local differences will be used for calculation. If reduced is chosen then only central local differences will be used. These modes are to improve the possible compression rates which CCSDS will produce. [3]

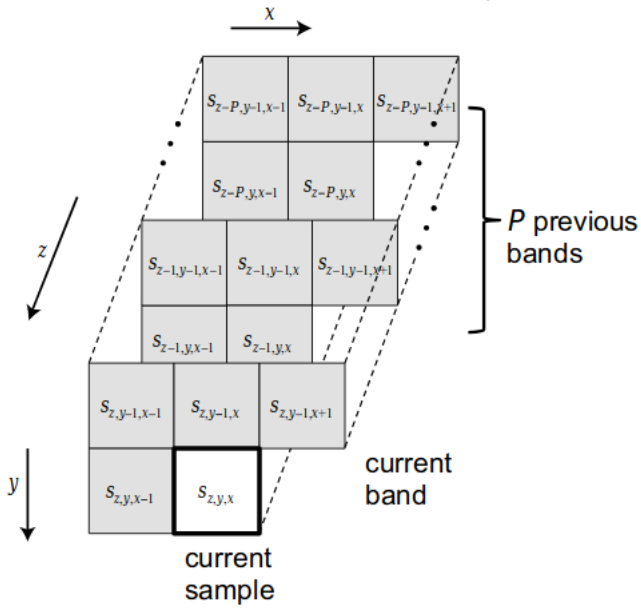
2.1.1.1. Local sum

CCSDS 123 Issue 1 proposes two methods for calculating the local sums; neighbour-oriented and column-oriented. The local sums will calculate a sum around a given sample as this is shown in Figure 2.4 or Equation 2.4. Using column-oriented local sum is not recommended when FULL mode is used and should therefore use neighbour-oriented local sum. Note that the localsum $\sigma_{z,0,0}$ is not used and not necessary to calculate. The corner case for neighbor oriented cover this as this is shown in Equation 2.3, and for column-oriented in Equation 2.4.

$$\sigma_{z,y,x} = \begin{cases} s_{z,y,x-1} + s_{z,y-1,x-1} + s_{z,y-1,x-1} + s_{z,y-1,x+1} & \text{if } y > 0 \text{ and } 0 < x < N_x - 1 \\ 4s_{z,y,x-1} & \text{if } y = 0 \text{ and } x > 0 \\ 2(s_{z,y-1,x} + s_{z,y-1,x+1}) & \text{if } y > 0 \text{ and } x = 0 \\ s_{z,y,x-1} + s_{z,y-1,x-1} + 2s_{z,y-1,x} & \text{if } y > 0 \text{ and } x = N_x - 1 \end{cases} \tag{2.3}$$

$$\sigma_{z,y,x} = \begin{cases} 4 * s_{z,y-1,x} & \text{if } y > 0 \\ 4 * s_{z,y,x-1} & \text{if } y = 0 \text{ and } x > 0 \end{cases} \tag{2.4}$$

Figure 2.3.: Neighborhood of a given sample $s_{z,y,x}$ [3]



2.1.1.2. Local differences

The local differences for each coordinate x,y,z is the difference between $\sigma_{z,y,x}$ and the sample $s_{z,y,x}$. The calculation of the central local diffrence is as shown in Equation 2.5. Similar to the localsum the local difference for $d_{z,0,0}$ is not defined, except for the directional local diffrences. The calculation of the directional local diffrences is as shown in Equation 2.6, Equation 2.7 and Equation 2.8. The directional local diffrences denoted by *N*, *W* and *NW* will denote the compass points in reference to a sample. The directional local differences will only be used when the prediction mode is used in full mode.

$$d_{z,y,x} = 4s_{z,y,x} - \sigma_{z,y,x} \tag{2.5}$$

$$d^{N}_{z,y,x} = \begin{cases} 4s_{z,y-1,x} - \sigma_{z,y,x} & \text{if } y > 0 \\ 0 & \text{if } y = 0 \end{cases} \tag{2.6}$$

$$d^{W}_{z,y,x} = \begin{cases} 4s_{z,y,x-1} - \sigma_{z,y,x} & \text{if } x > 0 \text{ and } y > 0 \\ 4s_{z,y-1,x} - \sigma_{z,y,x} & \text{if } y > 0 \text{ and } y > 0 \\ 0 & \text{if } y = 0 \end{cases} \tag{2.7}$$

$$d^{NW}_{z,y,x} = \begin{cases} 4s_{z,y-1,x-1} - \sigma_{z,y,x} & \text{if } x > 0 \text{ and } y > 0 \\ 4s_{z,y-1,x} - \sigma_{z,y,x} & \text{if } x = 0 \text{ and } y > 0 \\ 0 & \text{if } y = 0 \end{cases} \tag{2.8}$$

Figure 2.4.: Neighbor oriented local sum[3]

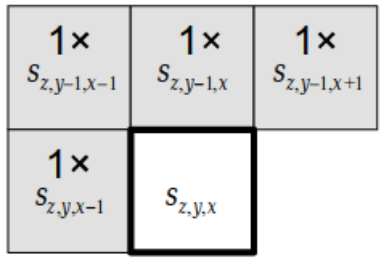
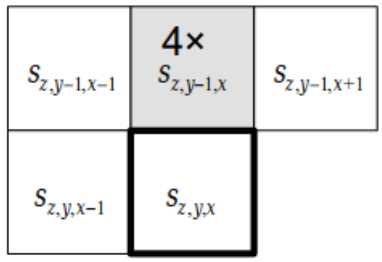


Figure 2.5.: Column-oriented local sum[3]



Prediction will use a number of preceding spectral bands defined by the user-specified parameter P, which is specified to be an integer in the range $0 \leq P \leq 15$. However if the current band z is less than P then the preceding bands will only look at z bands as this is defined in Equation 2.9.

$$P_z^* = min\{z, P\} \tag{2.9}$$

The calculation of the predicted sample requires the local difference vector $\mathbf{U}_z(t)$ of the P_z^* preceding bands. Under full mode the directional local differences will also be included in the vector as this is shown in Equation 2.10. Under reduced mode only the P_z^* preceding central local differences is used in the vector as shown in Equation 2.11.

$$\mathbf{U}_z(t) = \begin{bmatrix} d_z^N(t) \\ d_z^W(t) \\ d_z^{NW}(t) \\ d_{z-1}(t) \\ d_{z-1}(t) \\ \cdots \\ d_{z-P_z^*}(t) \end{bmatrix} \tag{2.10}$$

$$\mathbf{U}_z(t) = \begin{bmatrix} d_{z-1}(t) \\ d_{z-1}(t) \\ \cdots \\ d_{z-P_z^*}(t) \end{bmatrix} \tag{2.11}$$

2.1.1.3. Weight Vector

Similar to the local difference vector a weight vector $\mathbf{W}_z(t)$ of the same size of P is also required. Each element $\omega_z(t)$ in the vector is constrained to be in a range as defined in Equation 2.12, and the calculation of these values will be described in section 2.1.1.5. The user defined parameter Ω is the weight resolution of the weighs ω and is defined in the range $4 \leq \Omega \leq 19$.

$$\omega_{min} = -2^{\Omega+2}, \omega_{max} = 2^{\Omega+2} - 1 \tag{2.12}$$

Under full prediction mode the weight vectors $\mathbf{W}_z(t)$ will also include directional wights similar to local differences. This is as shown in Equation 2.13.

$$\mathbf{W}_z(t) = \begin{bmatrix} \omega_z^N(t) \\ \omega_z^W(t) \\ \omega_z^{NW}(t) \\ \omega_z^{(1)}(t) \\ \omega_z^{(2)}(t) \\ \cdots \\ \omega_z^{(P_z^*)}(t) \end{bmatrix} \tag{2.13}$$

For reduced mode the weight vector will only include a vector of P_z^* weights. The calculation of these is as shown in section 2.1.1.5

$$\mathbf{W}_z(t) = \begin{bmatrix} \omega_z^{(1)}(t) \\ \omega_z^{(2)}(t) \\ \cdots \\ \omega_z^{(P_z^*)}(t) \end{bmatrix} \tag{2.14}$$

2.1.1.4. Predicted Sample

The local difference vector and weight vector will be multiplied as this is shown in Equation 2.15. If the preceding bands is zero then this value will always be zero.

$$\hat{d} = \mathbf{W}_z^T(t)\mathbf{U}_z(t) \tag{2.15}$$

For calculation of the predicted sample value a function $mod_R^*[x]$ is used to store the value in an R-bit two's bit complement. The calculation of this is as shown in Equation 2.16. The user parameter R used in equation 2.16 is constrained to be an integer in the range of $max\{32, D + \Omega + 2 \leq 64\}$. Note that a higher value of R can prevent a possible overflow which is detailed more in [10].

$$mod_R^*[x] = ((x + 2^{R-1}) mod 2^R) - 2^{R-1} \tag{2.16}$$

The calculation of the scaled predicted value is done as shown in Equation 2.17, it can be shown the corner cases for $t = 0$ where the localsums and local differences is not used.

$$\tilde{s}_z(t) = \begin{cases} clip\Big(\Big\lfloor \frac{mod^*_R[\hat{d}_z(t)+2^{\Omega}(\sigma_z(t)-4s_{mid}]}{2^{\Omega+1}} \Big\rfloor + 2s_{mid} + 1, \{2s_{min}, 2s_{max}+1\}\Big) & t > 0 \\ 2s_{z-1}(t) & t = 0, P > 0, z > 0 \\ 2s_{mid} & t = 0 \text{ and } (P = 0 \text{ or } z = 0) \end{cases} \tag{2.17}$$

As shown in the calculation of the scaled predicted sample they will be used in a clip function which will limit the output value within a range. This function is as described in Equation 2.18.

$$clip(x, \{x_{min}, x_{max}\}) = \begin{cases} x_{min} & \text{if } x < x_{min} \\ x & \text{if } x_{min} \leq x \leq x_{max} \\ x_{max} & \text{if } x > x_{max} \end{cases} \tag{2.18}$$

Finally the predicted sample value is calculated by Equation 2.19.

$$\hat{s}_z(t) = \frac{\tilde{s}_z(t)}{2} \tag{2.19}$$

2.1.1.5. Weight update

After the calculation of the predicted sample value the weight vector will be updated. This is done by first calculating an scaled prediction error $e_z(t)$ as shown in Equation 2.20.

$$e_z(t) = 2s_z(t) - \tilde{s}_z(t) \tag{2.20}$$

Weights will update to according to the image statistics during computation, this is done by calculating a weight update factor $p(t)$ in Equation 2.21. The values will be clipped using the function in Equation 2.18 by the user-defined parameters v_{min}, v_{max}. t_{inc} is a user defined parameter defining the rate of incriminating of the statistics. These parameters are to be in the range of $-6 \leq v_{min} \leq v_{max} \leq 9$ and $2^4 \leq t_{inc} \leq 2^{11}$.

$$p(t) = clip\Big(v_{min} + \frac{t - N_x}{t_{inc}}, \{v_{min}, v_{max}\}\Big) + D - \Omega \tag{2.21}$$

Finally all the weight vector will be updated as this is shown in Equation 2.22. The function $sgn^+(x)$ is defined in Equation 2.23. Finally the weights will be clipped withing the weight resolution as specified in Equation 2.12.

$$\mathbf{W}_z(t+1) = clip\Big(\mathbf{W}_z(t) + \Big\lfloor \frac{1}{2}(sgn^+[e_z(t)] \cdot 2^{-p(t)} \cdot \mathbf{U}_z(t) + 1)\Big\rfloor, \{\omega_{min}, \omega_{max}\}\Big) \tag{2.22}$$

$$sgn^+(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases} \tag{2.23}$$

2.1.1.6. Mapped Prediction residual

Finally in the prediction stage is the calculation of the mapped prediction residuals $\delta_z(t)$ as this is calculated by Equation 2.24. Note that the mapped prediction residuals will be an unsigned integer even if the samples are signed. This will create a similar result for both signed and unsigned images.

$$\delta_z(t) = \begin{cases} |\Delta_z(t)| + \theta_z(t) & \text{if } |\Delta_z(t)| > \theta_z(t) \\ 2|\Delta_z(t)| & \text{if } 0 \leq (-1)^{\bar{s}_z(t)}\Delta_z(t) \leq \theta_z(t) \\ 2|\Delta_z(t)| - 1 & \text{if otherwise} \end{cases} \tag{2.24}$$

$$\Delta_z(t) = s_z(t) - \hat{s}_z(t) \tag{2.25}$$

$\theta_z(t)$ is defined by equation 2.26.

$$\theta_z(t) = min\{\hat{s}_z(t) - s_{min}, s_{max} - \hat{s}_z(t)\} \tag{2.26}$$

2.1.2. Sample adaptive encoder

The sample adaptive encoder will encode mapped prediction residuals $\delta_z(t)$ to reduce the number of bits representing each value. This is done by using variable-length binary codeswords which uses golomb-power-of 2 codes. The variable-length codes will be output according the the statistics of the images which is updated by the mapped prediction residuals. Each output codeword is constrained by the user-defined parameter U_{max} to limit the maximum output codeword to be represented by $U_{max} + D$ bits. U_{max} is a user-defined parameter defined in the range of $8 \leq U_{max} \leq 32$. The encoding of the mapped residuals $\delta_z(t)$ is calculated by the integer $k_z(t)$ which allows a range of codes that will satisfy $0 \leq k_z(t) \leq D - 2$. Each code-word is calculated by Equation 2.27 where delta is represented by the quotient *u* and the remainder *r*.

$$\delta = u \cdot 2^k + r \tag{2.27}$$

The quotient is calculated by Equation 2.28 and the remained is calculated by Equation 2.29.

$$u = \left\lfloor \frac{\delta}{2^k} \right\rfloor \tag{2.28}$$

$$r = \delta \mod 2^k \tag{2.29}$$

The codeword of $\delta_z(t)$ is determined by the value of $u_z(t)$. The output code for a given mapped prediction residual will be determined as such:

- If $u_z(t) < U_{max}$ then the codeword shall consist of a unary encoding of u, the binary representation shall be $u_z(t)$ zeros followed by a one. Then it shall be followed by $k_z(t)$ least significant bits of $\delta_z(t)$.
- If $u_z(t) > U_{max}$ then the binary represented codeword shall consist of U_{max} zeros followed by $\delta_z(t)$ of bit size *D*.

The encoding will update according to statistics of the mapped prediction residuals to reduce the necessary bits to represent each value. The code-words requires the value k which is calculated according to Equation 2.30. This value updates according to the accumulator $\Sigma_z(t)$ and counter$\Gamma_z(t)$. The ratio of

$\frac{\Sigma_z(t)}{\Gamma_z(t)}$ is the mean estimate of the image statistics which is used to calculate the code-word.

$$2^{k_z(t)} \leq \frac{\Sigma_z(t)}{\Gamma(t)} + \frac{49}{128} \tag{2.30}$$

The value of $k_z(t)$ with base 2 is calculated as shown in Equation 2.31.

$$k_z(t) \leq \log_2 \left(\Sigma_z(t) + \left\lfloor \frac{49}{2^7} \Gamma(t) \right\rfloor \right) \tag{2.31}$$

After each calculation of a code-word the statistics of the image will be updated as this is shown in Equation 2.32, and Equation 2.34. Accumulator $\Sigma_z(t)$ will carry the information for the value of the mapped prediction residuals, and the counter will carry the number of samples computed. The accumulator is initialised as shown in Equation 2.35 where the parameter $k_z^{'}$ shall be in the range of $0 \leq k_z^{'} \leq D-2$. The counter is initialised as shown in Equation 2.33 where the user defined parameter γ_0 is to be defined in the range of $1 \leq \gamma_0 \leq 8$. These values are limited to the user-defined parameter γ^* which is defined to be in the range $max\{4, \gamma_0 + 1\} \leq \gamma^* \leq 9$. Once the accumulator reaches the value of $2^* - 1$ it will divide by 2 to "reset" the accumulator and counter.

$$\Sigma_z(t) = \begin{cases} \Sigma_z(t-1) + \delta_z(t-1) & \text{if } \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \frac{\Sigma_z(t-1) + \delta_z(t-1) + 1}{2} & \text{if } \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \tag{2.32}$$

$$\Gamma(1) = 2^{\gamma_0} \tag{2.33}$$

$$\Gamma(t) = \begin{cases} \Gamma(t-1) + 1 & \text{if } \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \frac{\Gamma(t-1)+1}{2} & \text{if } \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \tag{2.34}$$

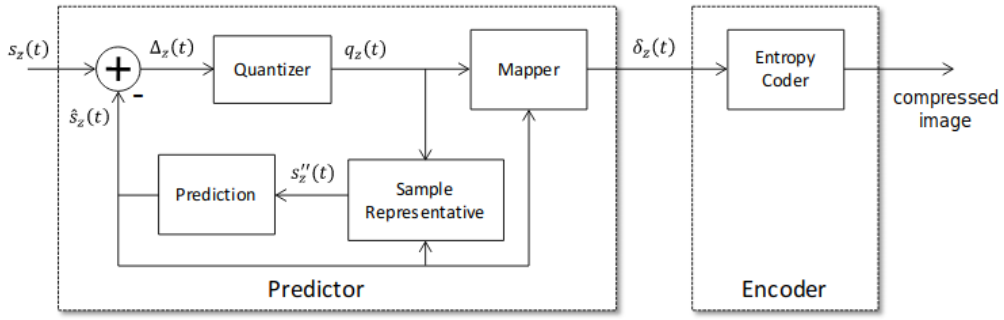
$$\Sigma_z(1) = \left\lfloor \frac{1}{2^7} \left(3 \cdot 2^{k_z^{'}+6}\right) \Sigma(1) \right\rfloor \tag{2.35}$$

2.2. CCSDS 123 Issue 2 Additions

This section will describe the new additions that has been made for the CCSDS 123 Issue 2 algorithm that was published in 2019 [4]. As mentioned this algorithm introduces a near-lossless compression algorithm that will achieve higher compression rates compared to Issue 1 which will benefit a space mission with limited hardware. The main source of information of the CCSDS 123 Issue 2 compression standard is described in [4]. CCSDS 123 Issue 2 will compress images similar to Issue 1 where it will first compute a value in prediction followed by an encoding the mapped quantizer index $\delta_z(t)$ to the output bit-stream. The prediction stage uses some similar elements to issue 1 but is changed to accommodate a near-lossless aspect of prediction. CCSDS 123 Issue 2 introduces a method to control the amount of lossy an image becomes during compression which is a huge benefit. For encoding it is still possible to use the sample adaptive encoder as this is described for Issue 1 in section 2.1.2, but a new encoder is introduced called the hybrid-encoder. This encoder is a hybrid of the sample-adaptive encoder which is inteded for a lower-entropy distribution mapped quantizer index $\delta_z(t)$. This primarily occurs because

of the near-lossless of prediction. The hybrid encoder will encode values as either lower-entropy or high-entropy depending on the statistics of the mapped quantizer index. The Low-entropy encoding is 16 variable-to-variable length codes that contains fixed output codes depending on the input. The High-entropy encoding is similar to the sample adaptive encoder. The Issue 2 top level design is shown in Figure 2.6 where the new additions of a quantizer and sample representative is the primary new changes to the prediction stage.

Figure 2.6.: Schematic[4]



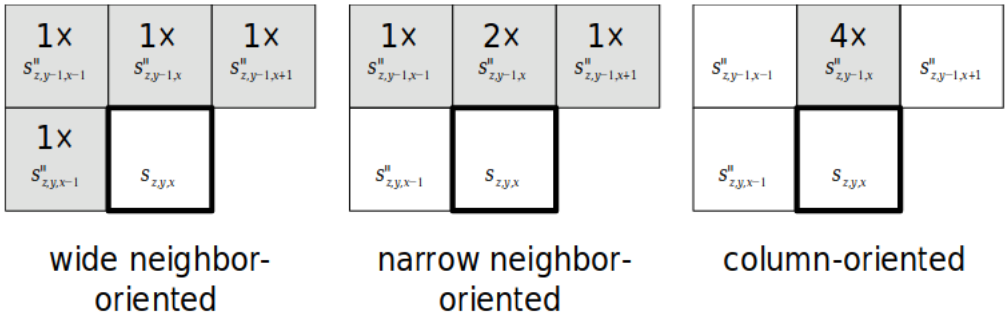
2.2.1. Predictor

CCSDS 123 Issue 2 prediction is similar to Issue 1 where it will compute a local sum, local difference and using weights for computation. The new introduction however is that computation is no longer around a sample $s_{z,y,x}$ but using sample representatives $\overset{"}{s}_{z,y,x}$ for near loss-less compression. The computation creates a data dependency by using previously computed sample representatives from the sample representation stage. The prediction is similar to what is in issue 1 but with some small changes that will be mentioned. In addition to this the quantizer and sample representative is the new additions to Issue 2 which provides the possibility for near-lossless compression. Finally the mapper is similar to the computation of mapped prediction residuals in issue 1 but with some small changes.

2.2.1.1. Local sum

For prediction it is required to compute the localsum $\sigma_{z,y,x}$ which is a weighted sum of the previous sample representatives $\overset{"}{s}_{z,y,x}$. Using the previous sample representatives allows the prediction stage to compress a sample for computation of the next sample. Issue 2 Local sum introduces four local sums where two new is introduced compared to the previous issue. The local sums can be computed with neighbour-oriented or column oriented. Issue 2 introduces the possibility to do wide or narrow local sums where wide oriented will include previous bands for computation. The different localsums can be shown in Figure 2.7.

Figure 2.7.: Sample dependency for local sums[4]



Wide-neighbor oriented shown in Equation 2.36 is similar to issue 1 as shown in Equation 2.3 with the exception of the local sum is using the sample representatives.

$$\sigma_{z,y,x} = \begin{cases} s''_{z,y,x-1} + s''_{z,y-1,x-1} + s''_{z,y-1,x-1} + s''_{z,y-1,x+1} & \text{if } y > 0 \text{ and } 0 < x < N_x - 1 \\ 4s''_{z,y,x-1} & \text{if } y = 0 \text{ and } x > 0 \\ 2(s''_{z,y-1,x} + s''_{z,y-1,x+1}) & \text{if } y > 0 \text{ and } x = 0 \\ s''_{z,y,x-1} + s''_{z,y-1,x-1} + 2s''_{z,y-1,x} & \text{if } y > 0 \text{ and } x = N_x - 1 \end{cases} \tag{2.36}$$

Narrow-neighbor is a new addition to Issue 2 which includes previous bands for computing the local sums. This is as shown in Equation 2.37.

$$\sigma_{z,y,x} = \begin{cases} s''_{z,y-1,x-1} + 2s''_{z,y-1,x} + s''_{z,y-1,x+1} & \text{if } y > 0 \text{ and } 0 < x < N_x - 1 \\ 4s''_{z-1,y,x-1} & \text{if } y = 0 \text{ and } x > 0 \text{ and } z > 0 \\ 2(s''_{z,y-1,x} + s''_{z,y-1,x+1}) & \text{if } y > 0 \text{ and } x = 0 \\ 2(s''_{z,y-1,x-1} + s''_{z,y-1,x}) & \text{if } y > 0 \text{ and } x = N_x - 1 \\ 4s_{mid} & \text{if } y = 0 \text{ and } x > 0 \text{ and } z = 0 \end{cases} \tag{2.37}$$

Wide column oriented is similar to issue 1 but as similar to wide neighbor it will use sample representatives instead of samples. The Equation 2.38 shows how it is done.

$$\sigma_{z,y,x} = \begin{cases} 4s''_{z,y-1,x} & \text{if } y > 0 \\ 4s''_{z,y,x-1} & \text{if } y = 0 \text{ and } x > 0 \end{cases} \tag{2.38}$$

Finally the narrow column local sum which is a new addition to issue 2 will also include previous bands for computing as this is shown in Equation 2.39.

$$\sigma_{z,y,x} = \begin{cases} 4s''_{z,y-1,x} & \text{if } y > 0 \\ 4s''_{z-1,y,x-1} & \text{if } y = 0 \text{ and } x > 0 \text{ and } z > 0 \\ 4s_{mid} & \text{if } y = 0 \text{ and } x > 0 \text{ and } z = 0 \end{cases} \tag{2.39}$$

2.2.1.2. Local differences

The local differences for Issue 2 does not use the samples $s_{z,y,x}$ but instead uses the sample representatives $\ddot{s}_{z,y,x}$. The central local difference is required to be calculated by Equation 2.40 and create a vector of P_z^* local differences from previous bands. If full mode is chosen then the computation will include the compass local differences denoted with North, West or Northwest. The local differences for North, West and Northwest calculations are computed respectively by Equation 2.41, Equation 2.42 and Equation 2.43.

$$d_{z,y,x} = 4\ddot{s}_{z,y,x} - \sigma_{z,y,x} \tag{2.40}$$

$$d^N_{z,y,x} = \begin{cases} 4\ddot{s}_{z,y-1,x} - \sigma_{z,y,x} & \text{if } y > 0 \\ 0 & \text{if } y = 0 \end{cases} \tag{2.41}$$

$$d^W_{z,y,x} = \begin{cases} 4\ddot{s}_{z,y,x-1} - \sigma_{z,y,x} & \text{if } x > 0 \text{ and } y > 0 \\ 4\ddot{s}_{z,y-1,x} - \sigma_{z,y,x} & \text{if } y = 0 \text{ and } y > 0 \\ 0 & \text{if } y = 0 \end{cases} \tag{2.42}$$

$$d^{NW}_{z,y,x} = \begin{cases} 4\ddot{s}_{z,y-1,x-1} - \sigma_{z,y,x} & \text{if } x > 0 \text{ and } y > 0 \\ 4\ddot{s}_{z,y-1,x} - \sigma_{z,y,x} & \text{if } x = 0 \text{ and } y > 0 \\ 0 & \text{if } y = 0 \end{cases} \tag{2.43}$$

2.2.1.3. Weight Vector

A weight vector $\mathbf{W}_z(t)$ of size P_z^* is used for prediction in the same way this is done in Issue 1. Each weight $\omega_z(t)$ has a bit-size defined by the user defined parameter Ω, and it is constrained to be within $4 \leq \Omega \leq 19$. The Weight vector will also include directional weights if FULL mode is chosen. The updating of the weights will be mentioned in section 2.2.1.7 where there are some changes compared to issue 1.

2.2.1.4. Prediction calculation

The method to calculate the predicted sample value $\hat{s}_z(t)$ is different from the method presented in issue 1. First it is done by calculating a high resolution prediction sample, followed by calculating the double resolution sample and finally the predicted sample value. The calculation of the high resolution predicted sample requires the multiplication of local difference vector and weight vectors as done in Issue 1 in Equation 2.15. After this is done then the calculation of the high resolution sample $\check{s}_z(t)$ is done as shown in Equation 2.44.

$$\check{s}_z(t) = clip\left(mod^*_R[\hat{d}_z(t) + 2^{\Omega}(\sigma_z(t) - 4s_{mid}) + 2^{\Omega+2}s_{mid} + 2^{\Omega+1}, \{2^{\Omega+2}s_{min}, 2^{\Omega+2}s_{max} + 2^{\Omega+1}\right) \tag{2.44}$$

The double resolution error is calculated as shown in Equation 2.45.

$$\tilde{s}_z(t) \begin{cases} \left\lfloor \frac{\check{s}_z(t)}{2^{\Omega+1}} \right\rfloor & \text{if } t > 0 \\ 2s_{z-1}(t) & \text{if } t = 0 \text{ and } P > 0 \text{ and } z > 0 \\ 2s_{mid} & \text{if } t = 0 \text{ and} (P = 0 \text{ or } z = 0) \end{cases} \tag{2.45}$$

Finally the predicted sample value $\hat{s}_z(t)$ is calculated as shown in 2.46.

$$\hat{s}_z(t) = \left\lfloor \frac{\tilde{s}_z(t)}{2} \right\rfloor \tag{2.46}$$

2.2.1.5. Quantization

Quantization is a new addition to CCSDS 123 Issue 2 which involves in calculating the quantizer index $q_z(t)$. First the sample residual $\Delta_z(t)$ is required which is the difference between sample $s_z(t)$ and the predicted sample value $\hat{s}_z(t)$, and is calculated as shown in Equation 2.47.

$$\Delta_z(t) = s_z(t) - \hat{s}_z(t) \tag{2.47}$$

The quantizer index is the quantization of the $\Delta_z(t)$ which is using a uniform quantizer with a step size $2m_z(t)$. The calculation of the quantizer index is as shown in Equation 2.48 wher e $m_z(t)$ is the fidelity control of maximum error. This will be more detailed in paragraph 2.2.1.5.1 where it is possible to control the maximum error with different methods.

$$q_z(t) = \begin{cases} \Delta_z(0) & \text{if } t = 0 \\ sgn(\Delta_z(t)) \left\lfloor \frac{|\Delta_z(t)| + m_z(t)}{2m_z(t)+1} \right\rfloor & \text{if } t > 0 \end{cases} \tag{2.48}$$

2.2.1.5.1. Fidelity control CCSDS 123 Issue 2 introduces a method to control the near-lossless compression of images by using fidelity control which involves controlling the value $m_z(t)$. By using $m_z(t) = 0$ would compress images lossless and be reversible when decompressing. However when $m_z(t)$ increases it is not possible to reproduce and provides loss when decompressing. There are three methods to control the value $m_z(t)$ by using absolute error, relative error or a combination of both. Absolute error will define a fixed $m_z(t)$ as shown in Equation 2.49 where the user-defined parameter $a_z(t)$ is limited to be in the range $0 \leq a_z \leq 2^{D_A} - 1$. D_A is a value defined to be in the range $1 \leq D_A \leq min\{D-1, 16\}$. The value $a_z(t)$ can be band-independent where the value is the same for each band, or band-dependent where each band has a unique $a_z(t)$ defined.

$$m_z(t) = a_z \tag{2.49}$$

Second option is to use relative error which will adjust based on the predicted sample value $\hat{s}_z(t)$ as shown in Equation 2.50. A relative error r_z is defined to be in the range $0 \leq r_z \leq 2^{D_R} - 1$, with the relative error bit depth D_R is defined to in the range $1 \leq D_R \leq min\{D-1, 16\}$. Relative error r_z can be band-independent where all bands use the same r_z or band-dependent where each band may use

unique r_z.

$$m_z(t) = \left\lfloor \frac{r_z|\hat{s}_z(t)}{2^D} \right\rfloor \tag{2.50}$$

The final possible fidelity control is the combination of absolute and relative error as shown Equation 2.51. This allows also for using band-independent or band-dependent error values, but also the possibility to mix both.

$$m_z(t) = min(a_z, \left\lfloor \frac{r_z|\tilde{s}_z(t)|}{2^D} \right\rfloor) \tag{2.51}$$

The updating of $m_z(t)$ has the possibility to be done less frequent by using an periodic error limit updater. This can be set by the user-defined parameter *u* which is an integer in the range $0 \leq u \leq 9$. This will limit the value $m_z(t)$ to update every 2^u frames instead of every frame.

2.2.1.6. Sample representation

Sample representation is the calculation of the sample representative value $\overset{"}{s_z}(t)$ to be used in following calculations of predicted sample values. First the clipped bin center is calculated as shown in Equation 2.52. If lossless is used then $s_z(t) = \overset{'}{s_z}(t)$, but if $m_z(t) \neq 0$ then the reconstruction of $s_z(t)$ will be at most $m_z(t)$.

$$\overset{'}{s_z}(t) = clip\Big(\tilde{s}_z(t) + q_z(t)(2m_z(t)+1), \{s_{min}, s_{max}\}\Big) \tag{2.52}$$

The double resolution sample representative is calculated as shown in Equation 2.53 which introduces the user-defined parameters damping ϕ_z, offset Ψ and resolution Θ. Resolution Θ is defined by in the range $0 \leq \Theta \leq 4$, and ϕ_z is defined in the range $0 \leq \phi_z \leq 2^\Theta - 1$. The offset Ψ is defined in the range $0 \leq \Psi \leq 2^\Theta - 1$. If lossless is chosen then $\Psi = 0$-

$$\tilde{\overset{"}{s}}_z(t) = \left\lfloor \frac{4(2^\Theta - \phi_z) \cdot (\overset{'}{s_z}(t) \cdot 2^\Omega - sgn(q_z(t)) \cdot m_z(t) \cdot \Psi_z \cdot 2^{\Omega-\Theta}) + \phi_z \cdot \check{s}_z(t) - \psi_z \cdot 2^{\Omega+1}}{2^{\Omega+\Theta+1}} \right\rfloor \tag{2.53}$$

Finally the sample representation of $s_z(t)$ is calculated as shown in Equation 3.2.

$$\overset{"}{s_z}(t) \begin{cases} s_z(0) & \text{if } t = 0 \\ \left\lfloor \frac{\overset{"}{s_z}(t)+1}{2} \right\rfloor & \text{if } t > 0 \end{cases} \tag{2.54}$$

2.2.1.7. Weight update

The weight update involves the calculation of the next weights for $t+1$ which is done the same way as Issue 1 in section 2.1.1.5. First is the calculation of dobule resolution error $e_z(t)$ as shown in Equation 2.55 using the clipped bin center as calculated in Equation 2.52.

$$e_z(t) = 2\overset{'}{s_z}(t) - \tilde{s}_z(t) \tag{2.55}$$

Similar to Issue 1 the prediction will update to the statistics of the image for the weight update control as shown in Equation 2.56.

$$p(t) = clip\Big(v_{min} + \left\lfloor \frac{t - N_x}{t_{inc}} \right\rfloor \{v_{min}, v_{max}\}\Big) + D + \Omega \tag{2.56}$$

The weight update will be done in a similar method as defined in issue 1 except with the introduction of the user-defined intra-band offset parameter ς_z^i and the intra-band offset parameter ς_z^*. Central weights will use the intra-band offset parameter for calculation as shown in Equation 2.57 where the parameter is defined to be in the range of $-6 \leq \varsigma_z^i \leq 5$. The directional weights North, West and Northwest will use the parameter ς_z^* as defined in Equation 2.58, Equation 2.59 and Equation 2.60. ς_z^* is defined to be in the range $-6 \leq \varsigma_z^* \leq 5$.

$$\omega^{(i)}(t+1) = clip\Big(\omega_z^{(i)}(t) + \Big\lfloor \frac{1}{2}\Big(sgn^+[e_z(t) \cdot 2^{-(p(t)+\varsigma_z^i)} \cdot d_{z-i}(t) + 1\Big\rfloor, \{\omega_{min}\omega_{max}\}\Big) \tag{2.57}$$

$$\omega^{N}(t+1) = clip\Big(\omega_z^{N}(t) + \Big\lfloor \frac{1}{2}\Big(sgn^+[e_z(t) \cdot 2^{-(p(t)+\varsigma_z^*)} \cdot d_{z-i}^N(t) + 1\Big\rfloor, \{\omega_{min}\omega_{max}\}\Big) \tag{2.58}$$

$$\omega^{W}(t+1) = clip\Big(\omega_z^{W}(t) + \Big\lfloor \frac{1}{2}\Big(sgn^+[e_z(t) \cdot 2^{-(p(t)+\varsigma_z^*)} \cdot d_{z-i}^W(t) + 1\Big\rfloor, \{\omega_{min}\omega_{max}\}\Big) \tag{2.59}$$

$$\omega^{NW}(t+1) = clip\Big(\omega_z^{NW}(t) + \Big\lfloor \frac{1}{2}\Big(sgn^+[e_z(t) \cdot 2^{-(p(t)+\varsigma_z^*)} \cdot d_{z-i}^{NW}(t) + 1\Big\rfloor, \{\omega_{min}\omega_{max}\}\Big) \tag{2.60}$$

2.2.1.8. Mapped quantizer index

The final stage of prediction is the computation of the mapped quantizer index which is calculated in Equation 2.61. This is similar to Issue 1 except using the quantizer indexes.

$$\delta_z(t) = \begin{cases} |q_z(t)| + \theta_z(t) & \text{if } |q_z(t)| > \theta_z(t) \\ 2|q_z(t)| & \text{if } 0 \leq (-1)^{\bar{s}_z(t)} q_z(t) \leq \theta_z(t) \\ 2|q_z(t)| - 1 & \text{if otherwise} \end{cases} \tag{2.61}$$

$\theta_z(t)$ is calculated by Equation 2.62 where the new addition is including the maximum error value $m_z(t)$ for calculation.

$$\theta_z(t) = \begin{cases} min\{\hat{s}_z(0) - s_{min}, s_{max} - \hat{s}_z(0)\} & \text{if t = 0} \\ min\{\Big\lfloor \frac{\hat{s} - s_{min} + m_z(t)}{2m_z(t)+1} \Big\rfloor, \Big\lfloor \frac{s_{max} - \hat{s}_z(t) + m_z(t)}{2m_z(t)+1} \Big\rfloor\} & \text{if } t > 0 \end{cases} \tag{2.62}$$

2.2.2. Hybrid entropy encoder

The hybrid encoder is a new addition for CCSDS 123 Issue 2 which encodes mapped quantizer indexes $\delta_z(t)$ to an output bit-stream by reducing the amount of bits necessary to represent them. The hybrid encoder as mentioned is a hybrid of the sample adaptive encoder but is adapted to a different image pattern because of the near-lossless compression. The hybrid encoder will as mentioned encode in high-entropy or low-entropy encoding. High-entropy encoding is similar to the sample adaptive encoder and will produce an output code-word immediately. Low-entropy code will collect more mapped prediction samples before producing an output code-word which allows for multiple inputs to be represented fewer bits. The

selection of high-entropy or low-entropy is based on the image statistics and a threshold to determine low-entropy encoding. The image statistics is determined by the counter $\Gamma_z(t)$ and accumulator $\tilde{\Sigma}_z(t)$ similar to the sample adaptive encoder. The counter $\Gamma_z(t)$ is initialised as shown in Equation 2.63 by the user defined parameter $\gamma 0$ in the range of $1 \leq \gamma_0 \leq 8$.

$$\Gamma(0) = 2^{\gamma_0} \tag{2.63}$$

The accumulator will be initialised by the user-defined parameter $\tilde{\Sigma}_z(0)$ in the range of $0 \leq \tilde{\Sigma}_z(0) \leq 2^{D+\gamma_0}$. The accumulator will be updated based on the encode mapped quantizer index as shown in Equation 2.64. When the accumulator rescales the least significant bit of $\tilde{\Sigma}_z(t)$ will be appended to the bitstream for reconstructing the accumulator during decompression.

$$\tilde{\Sigma}_z(t) = \begin{cases} \tilde{\Sigma}_z(t-1) + 4\delta_z(t) & \text{if } \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \left\lfloor \frac{\tilde{\Sigma}_z(t-1) + 4\delta_z(t) + 1}{2} \right\rfloor & \text{if } \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \tag{2.64}$$

The counter is update as shown in Equation 2.65.

$$\Gamma(t) = \begin{cases} \Gamma(t-1) + 1 & \text{if } \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \left\lfloor \frac{\Gamma(t-1)+1}{2} \right\rfloor & \text{if } \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \tag{2.65}$$

The next step is determine if the mapped quantizer index is a high-entropy or low entropy number. This is done by determining calculating the threshold where this occurs, and the calculation of the threshold is as shown in Equation 2.66. If the calculation is above the treshold T_0 then it is determined to be a high-entropy code. T_0 is the value for code index 0 determined by the treshold table for low entropy codes as shown in Table 2.1. Equation 2.67 shows a simpler way to determine if the treshold is satisfied.

$$\tilde{\Sigma}_z(t) \cdot 2^{14} \geq T_0 \cdot \Gamma(t) \tag{2.66}$$

$$\frac{\tilde{\Sigma}_z(t) \cdot 2^{14}}{\Gamma(t)} \geq T_0 \tag{2.67}$$

2.2.2.1. High-Entropy

If the treshold is satisfied as shown in Equation 2.67 then the encoding will be using high-entropy codes. The high entropy encoding as mentioned is similar to the sample adaptive encoding in section 2.1.2. However CCSDS 123 Issue 2 changes this by writing the output in a reverse order as such:

- If $u_z(t) < U_{max}$ then the codeword shall consist of the $k_z(t)$ least significant bits of $\delta_z(t)$, followed by a 'one' bit, and followed by $u_z(t)$ 'zeros'.
- If $u_z(t) > U_{max}$ then the binary represented codeword shall consist of $\delta_z(t)$ of size *D* followed by U_{max} 'zeros'.

2.2.2.2. Low-Entropy

When the statistics for the current mapped quantizer index would be below the threshold value T_0 then it would be determined as a low entropy encoding. Encoding these numbers are done by a set of 16 inputs,

binary output and different fixed codes for each input. These 16 codes is as shown in Table 2.1 where each different code is differed by the code index *i*. Each code has a input symbol limit which is used to determine if the numbers was an unlikely input and it sets the maximum value within each codetable that is possible. Each code is also determined by the statistics in similar manner to determine if a code is high entropy or low entropy, and each code index *i* has a threshold value T_i as shown in Table 2.1. One speciality about the low entropy encoding is to allow for multiple inputs to fewer outputs and it is done by storing a sequence of active prefix AP_i. During encoding of low entropy numbers a mapped quantizer index will be appended to the sequence and to be checked for a valid output. Appendix E provides 16 code tables and flush tables used for checking of these inputs to output codes. If an input is valid then the corresponding output code will be appended to the bit-stream. For example if the active prefix for index 15 has stored 256 zeros then the valid output is a 1 bit written to the output stream. If each of those zeros was represented with 16 bits then the compression rate is huge, but this is not the case for images with varying statistics. Choosing the correct code index for a mapped quantizer index is chosen

Table 2.1.: Low entropy code input symbol limit and threshold.

Code Index, i	Input Symbol Limit, L_i	Treshold, T_i	Active Prefix (AP_i)
0	12	303336	
1	10	225404	
2	8	166979	
3	6	128672	
4	6	95597	
5	4	69670	
6	4	50678	
7	4	34898	
8	2	23331	
9	2	14935	
10	2	9282	
11	2	5510	
12	2	3195	
13	2	1928	
14	2	1112	
15	0	408	

by satisfying the equation Equation 2.68 and choose the largest code index in Table 2.1. If a threshold value is calculated to be e.g 409 then the code is determined for code index 14.

$$\tilde{\Sigma}_z(t) \cdot 2^{14} < T_i \cdot \Gamma(t) \tag{2.68}$$

Equation 2.68 can be rewritten as Equation 2.69.

$$\frac{\tilde{\Sigma}_z(t) \cdot 2^{14}}{\Gamma(t)} < T_i \tag{2.69}$$

After a code index is chosen then it must be determined if the input was an unlikely number. The next input symbol $\iota_z(t)$ is chosen by Equation 2.70 where the hexadecimal representation of the mapped

quantizer index $\delta_z(t)$ is stored. However if the mapped quantizer index $\delta_z(t)$ is larger than the input symbol limit from Table 2.1 then an X will be used instead, and this number is therefore considered unlikely. If an X is appended then $\delta_z(t) - L_i - 1$ will be encoded to the bitstream. A '1' bit followed by the D-bit value of $\delta_z(t) - L_i - 1$ followed by U_{max} 'zeros' will be appended to the bitstream.

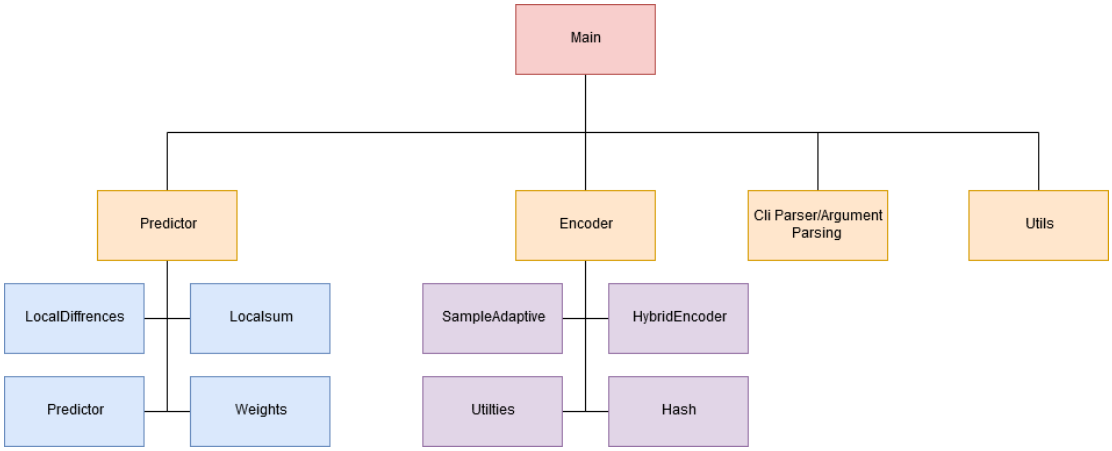
$$\iota_z(t) = \begin{cases} \delta_z(t) & \text{if } \delta_z(t) \leq L_i \\ X & \text{if } \delta_z(t) > L_i \end{cases} \tag{2.70}$$

Each input symbol $\iota_z(t)$ will be appended to their corresponding code index chosen by the image statistics. If after appending a code index there is a corresponding active-prefix , then the output code will be written to the bitstream. The active prefix will also be reset to a null sequence. Appendix E gives an example of these codewords and the corresponding output code word, these tables are extracted from the CCSDS 123 Issue 2 standard in [4]. After the encoding of the image is done then the remaining active prefixes AP_i will be flushed to the bitstream by using a set of 16 flush-tables. Each remaining active prefix will have a corresponding output code that will be written to the bitstream. Appendix E gives an example of the flush tables for table 2 where each active prefix has a corresponding flush word. Finally the final accumulator value $\tilde{\Sigma}_z(N_x \cdot N_y - 1)$ will be written to the bitstream of $2 + D + \gamma^*$ bits. A '1' will be written after the accumulator value followed by fill bits to make the compressed image a multiple of the output word size.

2.3. CCSDS 123 Issue 2 Software Implementation

This section will explain some details regarding the software implementation that was made in the project thesis in [1]. Note that the additions in 4 will provide changes to the software that have improved or fixed problems that occurred for this implementation. The overall structure of the software has not been changed and can be shown in figure 3.2. The figure shows how the folder structure and modular design of the software and the location to find the different implementations of CCSDS Issue 2. The compression

Figure 2.8.: Modular Software Implementation[1]



algorithm involves two steps to compress images at the top level design which is to predict samples and then to encode them into a bit-stream. Figure 2.9 provides the code procedure from the top level on how this is done. First the software will parse arguments provided to the command line interfaces. These primarily contain the user-defined arguments for the CCSDS 123 Issue 2 algorithm. These arguments can be shown in table 3.1 where each CCSDS 123 Issue 2 argument can be shown. The implementation of the argument parsing uses the ArgP library which is included in the standard GNU C library. After the argument has been parsed then it is possible to determine the image size and read every sample into memory for compression. Specifically the x, y and z arguments provide the image size.

Figure 2.9.: Procedure for compression[1]

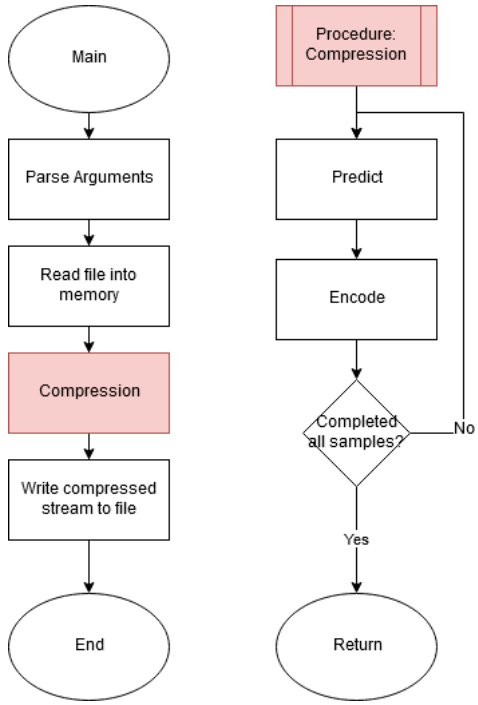


Table 2.2.: User defined parameters for arguments[1]

Argument Description	Argument	Allowed Values
Full Prediction Mode	-f	NONE
Debug Mode	–debug	NONE
Register size, R	-r	$max\{32, D + \Omega + 2\} \leq R \leq 64$
Sample resolution	-f	N/A
Dynamic Range	-d	$2 \ldots 16$
Prediction Bands, P	-p	$0 \leq P \leq 15$
Weight resolution, Ω	N/A	$4 \leq \Omega \leq 19$
Weight interval, t_{inc}	N/A	$2^4 \leq t_{inc} \leq 2^{11}$
Vmin	-v	$-6 \leq v_{min} \leq v_{max} \leq 9$
Vmax	-V	$-6 \leq v_{min} \leq v_{max} \leq 9$
Image size X	-x	$1 \leq X \leq 2^{16} - 1$
Image size Y	-y	$1 \leq Y \leq 2^{16} - 1$
Image size Z	-z	$0 \leq Z \leq 2^{16} - 1$
Word Size	N/A	Computer word size, 8 = 64-bit
Initial Accumulator, Sample Adaptive, $\Sigma_z(1)$	N/A	$0 \leq K \leq min(D - 2, 14)$
Initial counter, γ^0	N/A	$1 \leq \gamma^0 \leq 8$
Counter rescaling, γ^*	N/A	$max(4, \gamma_0 + 1) \leq \gamma^* \leq 11$
U_{max}	N/A	$8 \leq U_{max} \leq 32$

As mentioned the compression will predict samples and encode them into the bit-stream. The objective of the prediction is to reduce the image samples to smaller numbers to benefit a greater compression rate when encoding. The introduction of quantization and lossy compression would create a much lower entropy number which could benefit the compression rate even more. The prediction follows the prediction algorithm as described in section 2.2.1 and the overall procedure is done as follows:

- Compute the local sum for $t > 0$ (Will store all the sum in memory)
- Compute the local differences and create the diffrences vector for $t > 0$
- Compute high resolution prediction sample for $t > 0$
- Compute double resolution prediction sample
- Compute the predicted sample value
- Compute the quantization value
- Compute the clipped bin center
- Compute the sample representation and store it (Will store all sample rep in memory)
- Weight update/Weight init
- Compute and return the mapped quantizer index

The code for the procedure of prediction is as shown in listing 2.1, note that some changes to prediction has been made in chapter 4, but the overall structure remains the same. This code will calculate everything specified for the prediction stage as this was described in section 2.2.1. This top level code will calculate the prediction quantizer index for each sample $s_{z,y,x}$.

```
uint32_t predict(uint32_t * inputSample, uint16_t x, uint16_t y, uint16_t z, struct
    arguments * parameters, uint32_t * sampleRep, int32_t * localsum,
int32_t * diffVector, int32_t * weights, int32_t sMin, int32_t sMax, int32_t sMid,
    uint32_t maximumError, uint32_t sampleDamping, uint32_t sampleOffset, uint32_t
    interbandOffset, int32_t intrabandExponent) {
  /*
    Calculate local sum and build up the diffrential vector at a given sample.
  */
  int64_t highResSample = 0;
  if(x+y != 0) {
    narrowNeighborLocalSum(sampleRep,localsum, sMid, x,y,z,parameters);
    BuildDiffVector(sampleRep,localsum,diffVector,x,y,z,parameters);
    highResSample = computeHighResPredSample(localsum, weights, diffVector, sMid, sMin
    , sMax, x, y, z, parameters);
  }
  /*
    Step for calculating prediction sample and doubleResPredSample
  */
  int64_t doubleResPredSample = 0; // Calculated inside function
    computePredictedSample
  int64_t predictedSample = computePredictedSample(inputSample, &doubleResPredSample,
    localsum, highResSample, sMid, sMin, sMax, x, y, z, parameters);
  /*
    Quantization
  */
  int32_t quantizerIndex = quantization(inputSample, predictedSample, maximumError, x,
     y, z, parameters);
  /*
    Sample Representation
  */
  int32_t clippedBin = clippedBinCenter(predictedSample, quantizerIndex, maximumError,
     sMin, sMax);
  sampleRep[offset(x,y,z, parameters)] = sampleRepresentation(inputSample, clippedBin,
     predictedSample, quantizerIndex, maximumError, highResSample, sampleDamping,
    sampleOffset, x, y, z, parameters);
  /*
    Weight Update
  */
  if(x+y == 0) {
    initWeights(weights, z, parameters);
  } else {
    int64_t doubleResError = (clippedBin << 1) - doubleResPredSample;
    updateWeightVector(weights, diffVector, doubleResError, x, y, z, interbandOffset,
    intrabandExponent, parameters);
  }
  int32_t residual = computeMappedQuantizerIndex(quantizerIndex, predictedSample,
    doubleResPredSample, sMin, sMax, maximumError, x, y, z, parameters);
  return residual;
}
```

Listing 2.1: Prediction function

This procedure will iterate over each sample in a BSQ/BIL or BIP order and compute the mapped quantizer index $\delta_z(t)$ for encoding. After the prediction is finished then there are two encoding methods which is the hybrid encoder or the sample adaptive encoder. The objective for these is to reduce the number of bits needed to represent each mapped quantizer index. When the sequences of mapped quantizer indexes are low entropy numbers or close to 0 then these will affect the compression rate by reducing the number of bits needed. One renown encoding of numbers is the Huffman Codes which could work as an encoder for CCSDS 123, but the Huffman Coding is not effective in hardware as it would require to store all the mapped quantizer indexes before encoding[4, 11].

Concerning the hybrid encoder it still remained to fix the problems regarding the implementation where the program could not find specific codes. This particular issue will be further detailed in section 4.5. Instead of using the hybrid encoder the software could use the sample adaptive encoder. The sample adaptive encoder was completed and functional to compress the mapped quantizer indexes. The implementation of the sample adaptive encoder follows the procedure as described in section 2.1.2, with the C code implementation as shown in listing 3.6.

```
1  int encodeSampleAdaptive(unsigned long sampleToEncode, unsigned int * counter,
       unsigned int * accumulator, int x, int y, int z, unsigned int * totalWrittenBytes,
        unsigned int * numWrittenBits, FILE * fileToWrite, struct arguments * parameters)
        {
2      if(y == 0 && x == 0) {
3          writeBits(sampleToEncode, parameters->dynamicRange, numWrittenBits,
       totalWrittenBytes, fileToWrite);
4      } else {
5          int64_t kValue = log2((accumulator[z] + ((49*counter[z]) >> 7)) / counter[z]);
6          kValue = kValue < 0 ? 0 : kValue;
7          kValue = kValue > (parameters->dynamicRange - 2) ? parameters->dynamicRange -
       2 : kValue;
8          uint64_t uValue = sampleToEncode >> kValue;
9          if(uValue < parameters->uMax) {
10             writeBits(0, uValue, numWrittenBits, totalWrittenBytes, fileToWrite);
11             writeBits(1, 1, numWrittenBits,totalWrittenBytes, fileToWrite);
12             writeBits(extractBits(sampleToEncode, kValue), kValue, numWrittenBits,
       totalWrittenBytes, fileToWrite);
13         } else {
14             writeBits(0, parameters->uMax, numWrittenBits, totalWrittenBytes,
       fileToWrite);
15             writeBits(sampleToEncode, parameters->dynamicRange, numWrittenBits,
       totalWrittenBytes, fileToWrite);
16         }
17
18         if(x+y != 0) {
19             if(counter[z] < ((1 << parameters->yStar) - 1)) {
20                 accumulator[z] += sampleToEncode;
21                 counter[z]++;
22             } else{
23                 accumulator[z] = (accumulator[z] + sampleToEncode + 1) >> 1;
24                 counter[z] = (counter[z] + 1) >> 1;
25             }
26         }
27     }
28
29     return 0;
```

```
30 }
```

Listing 2.2: Prediction function

The resulting compression rates achieved by the sample adaptive encoder as shown in table 2.3 did provide good results of compression rates using the sample adaptive encoder. Note that these results are lossless which saved roughly 50% of data sizes. The introduction of lossy compression which is added in chapter 4 could yeild higher compression rates at the cost of quality of image. The software presented

Table 2.3.: Compression rate with sample adaptive

Dataset	Total Size	Compressed Size	Compression Ratio
HICO L2_1.BSQ	133.1 MB	34.9 MB	3.81
HICO L2_2.BSQ	133.1 MB	47.8 MB	2.78
HICO L2_3.BSQ	133.1 MB	49.2 MB	2.71
HICO L2_4.BSQ	133.1 MB	48.1 MB	2.77
HICO L2_6.BSQ	133.1 MB	64.3 MB	2.07

in [1] as mentioned was not completely finished, and it would still require improvements to provide a feasible implementation. The most important work that remained is presented as such:

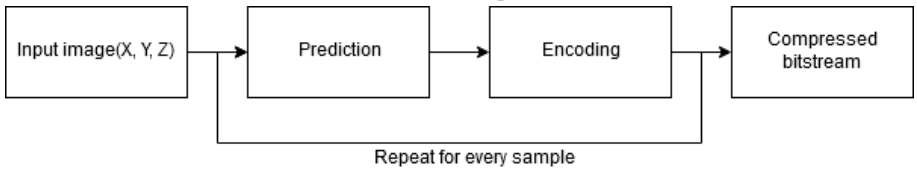
- Support different bit sizes, D. The implementation only supports 16-bit image data.
- The software implementation requires to support signed numbers and the implementation only supports unsigned.
- Argument parsing does not support all user-defined parameters, some of these are set to a default value in the argument parser.
- Currently the software only support BSQ. An important feature is to support BIP and BIL.
- The hybrid encoder requires future work to solve the issues regarding the codewords and a better testing solution. A decoder is a possible testing solution.
- The software implementation needs a decompression implementation to be able to read a compressed stream. The sample adaptive already has a possible decoder solution from issue 1, but the predictor for CCSDS 123 Issue 2 does not have a depredictor and requires an implementation.

These necessary fixes will be addressed in chapter 4.

3. CCSDS 123 Issue 1 implementation

As the HYPSO mission will launch with the CCSDS 123 Issue 1 standard running on the on-board processing pipeline, it was necessary to implement a software solution as a backup if the FPGA on-board did not operate correctly. This chapter will present two different versions of the software implementation but very similar implementations. The embedded version which will run as the software backup is presented in section 3.3 which will only implement the compression of hyper-spectral images and is integrated into the HYPSO software running on-board. Section 3.1 will present the normal implementation of the compression standard which is a more user friendly program to be run on a standard computer. This implementation will also include a decompression implementation to do decompression of hyper-spectral images. Both implementation follow the iteration of computing the prediction stage and encoding a sample into the compressed bitstream. Figure 3.1 presents the general idea of how this process works where the input image is a stream of samples according to the BIP, BIL or BSQ standard.

Figure 3.1.: Top Level



3.1. Standard Version

A software similar to the one made in the project thesis as described in section 2.3 was made for testing and decompression of the compressed images. The software for Issue 1 uses the same library and the same structure as the software but has removed any CCSDS 123 Issue 2 implementations. This implementation can be found in Appendix D. The software uses the ArgP library to parse arguments into the program and these arguments are described as follows:

Table 3.1.: User defined parameters for arguments[1]

Argument Description	Argument	Allowed Values	Default
Full Prediction Mode	-f	NONE	Reduced
Debug Mode	–debug	NONE	OFF
Register size, R	-r	$max\{32, D + \Omega + 2\} \leq R \leq 64$	N/A
Sample resolution	-f	N/A	N/A
Dynamic Range	-d	$2 \ldots 16$	N/A
Prediction Bands, P	-p	$0 \leq P \leq 15$	N/A
Weight resolution, Ω	N/A	N/A	N/A
Weight interval, t_{inc}	N/A	N/A	N/A
Vmin	-v	$-6 \leq v_{min} \leq v_{max} \leq 9$	N/A
Vmax	-V	$-6 \leq v_{min} \leq v_{max} \leq 9$	N/A
Image size X	-x	$1 \leq X \leq 2^{16} - 1$	N/A
Image size Y	-y	$1 \leq Y \leq 2^{16} - 1$	N/A
Image size Z	-z	$0 \leq Z \leq 2^{16} - 1$	N/A
Θ	N/A	N/A	0
ω_{min}	N/A	N/A	N/A
ω_{max}	N/A	N/A	N/A
wordSize	N/A	N/A	N/A
initialAccumulator, $\check{\Sigma}_z(0)$	N/A	N/A	2^6
U_{max}	N/A	N/A	N/A

3.1.1. Prediction

This section will describe how the prediction is implemented in the programming language C. This implementation will describe the algorithms described in section 2.1 regarding CCSDS Issue 1. As listing 3.1 shows the implemented code for doing the prediction. This involves creating the local differences as according to section 2.1.1.2 by iterating through *P* previous bands.

```
uint16_t predict(uint16_t * inputSample, uint16_t x, uint16_t y, uint16_t z, struct
    arguments * parameters, int64_t * diffVector, int32_t * weights) {
  /*
    Calculate local sum and build up the diffrential vector at a given sample.
  */
  if(x+y != 0) {
    BuildDiffVector(inputSample, diffVector, x, y, z, parameters, wideNeighborLocalSum
    );
  }
  /*
    Step for calculating prediction sample and scaledPredicted
  */
  int32_t scaledPredicted = computeScaledPredicted(inputSample, weights, diffVector,
    wideNeighborLocalSum(inputSample, x, y, z, parameters), x, y, z, parameters);

  /*
    Update weights
  */
```

```
  if(x+y == 0) {
    initWeights(weights, z, parameters);
  } else {
    int64_t doubleResError = (inputSample[offset(x,y,z,parameters)] << 1) -
    scaledPredicted;
    updateWeightVector(weights, diffVector, doubleResError, x, y, z, parameters);
  }
  return computeMappedResidual(inputSample[offset(x,y,z,parameters)], scaledPredicted,
     parameters);
}
```

Listing 3.1: Precition stage of CCSDS 123 Issue 1

The iteration of the previous bands is as shown in listing 3.2 which shows how it is done for the central local differences and the directional local diffrences. Note that only the directional local differences is calculated when Full mode is used during compression. The function creates the local diffrences as shown in equation 2.10 or equation 2.11.

```
void BuildDiffVector(uint16_t * sample,  int64_t * diffVector, uint16_t x, uint16_t y,
     uint16_t z, struct arguments * parameters, int32_t(*localSumFunc)(uint16_t *,
    uint16_t, uint16_t, uint16_t, struct arguments *)) {
  int currentPredBands = z < parameters->precedingBands ? z : parameters->
    precedingBands;
  if (z > 0) {
    for (int i = 0; i < currentPredBands; i++) {
      diffVector[i] = (sample[offset(x,y,z-i-1,parameters)] << 2) - localSumFunc(
    sample, x, y, z-i-1, parameters);
    }
  }
  if (parameters->mode == FULL) {
    if(x+y != 0) {
      diffVector[parameters->precedingBands] = northLocalDiffrence(sample, x, y, z,
    localSumFunc(sample, x, y, z, parameters), parameters);
      diffVector[parameters->precedingBands+1] = westLocalDiffrence(sample, x, y, z,
    localSumFunc(sample, x, y, z, parameters), parameters);
      diffVector[parameters->precedingBands+2] = northwestLocalDiffrence(sample, x, y,
     z, localSumFunc(sample, x, y, z, parameters), parameters);
    }
  }
}
```

Listing 3.2: Building the Local Difference function

For calculating the scaled predicted sample as the formula is specified in equation 2.19 it is done as shown in listing 3.3. Note that the left shifting can cause severe issues when done on negative numbers, this is circumvented by left shifting a positive number. Multiplying the positive number by negative 1 if the end result will be negative.

```
int32_t computeScaledPredicted(uint16_t * sample, int32_t * weightVector, int64_t *
    diffVector, int32_t localsum,
  uint16_t x, uint16_t y, uint16_t z, struct arguments * parameters) {

  int32_t scaledPredicted = 0;
  if(x+y == 0) {
    if(z == 0 || parameters->precedingBands == 0) {
      scaledPredicted = parameters->sMid << 1;
    } else {
```

```
        scaledPredicted = sample[offset(x,y,z-1,parameters)] << 1;
    }
  } else {
    int32_t diffPredicted = innerProduct(weightVector, diffVector, z, parameters);
    /// Shifting of negative numbers can cause severe issues. Thus shifting of the
    absolute value and then make it negative.
    int64_t tmpValue = localsum - (parameters->sMid << 2);
    int sgn = tmpValue < 0;
    tmpValue = abs(tmpValue) << parameters->weightResolution;
    tmpValue = sgn ? -1 * tmpValue : tmpValue;
    int64_t highResSample = modR(diffPredicted + tmpValue, parameters->registerSize);
    ///
    highResSample = highResSample >> (parameters->weightResolution+1);
    highResSample += ((parameters->sMid<<1) + 1);

    int32_t lowerbounds = parameters->sMin<<1;
    int32_t higherbounds = (parameters->sMax<<1) + 1;
    scaledPredicted = clip(highResSample, lowerbounds, higherbounds);
  }
  return scaledPredicted;
}
```

Listing 3.3: Computing the scaled predicted sample

Finally the weights will be updated as specified in section 2.1.1.5 where listing 3.4 shows the implementation.

```
void updateWeightVector(int32_t * weights, int64_t * diffVector, int32_t error,
    uint16_t x, uint16_t y, uint16_t z, struct arguments * parameters) {

    int64_t weightLimit = 0x1 << (parameters->weightResolution + 2);
    int signError = error < 0 ? -1 : 1;
    uint64_t scalingExp = 0;
    scalingExp = ((y-1)*parameters->xSize+x) / (1<<parameters->weightInterval);
    scalingExp = clip(parameters->weightMin + scalingExp, parameters->weightMin,
    parameters->weightMax);
    scalingExp += parameters->dynamicRange - parameters->weightResolution;
    int currentPredictionBand = z < parameters->precedingBands ? z : parameters->
    precedingBands;

    long tmp = 0;
    for(int i = 0; i < currentPredictionBand; i++) {
        if((scalingExp) > 0) {
            tmp = (signError * diffVector[i]) >> (scalingExp);
        } else {
            tmp = (signError * diffVector[i]) << (-1*(scalingExp));
        }
        tmp = (tmp + 1) >> 1;
        weights[i] = clip(weights[i] + tmp, (-1 * weightLimit), (weightLimit - 1));
    }
    if(parameters->mode == FULL) {
        for (int i = 0; i < 3; i++) {
            if((scalingExp) > 0) {
                tmp = (signError * diffVector[parameters->precedingBands + i]) >> (
    scalingExp);
            } else {
                tmp = (signError * diffVector[parameters->precedingBands + i]) <<
    (-1*(scalingExp));
```

```
            }
            tmp = (tmp + 1) >> 1;
            weights[parameters->precedingBands + i] = clip(weights[parameters->
    precedingBands + i] + tmp, (-1 * weightLimit), (weightLimit-1));
        }
    }
}
```

Listing 3.4: Updating Weights

The final stage of the prediction is calculating the mapped residual which calculates as specified in section 2.1.1.6. Listing 3.5 shows the implementation.

```
uint16_t computeMappedResidual(uint16_t sample, uint32_t scaledPredicted, struct
    arguments * parameters) {
  uint16_t predictedSample = scaledPredicted >> 1;
  int32_t delta = sample - predictedSample;

  int32_t temp1 = predictedSample - parameters->sMin;
  int32_t temp2 = parameters->sMax - predictedSample;
  int32_t omega = temp1 > temp2 ? temp2 : temp1;

  if (abs(delta) > omega) {
    return abs(delta) + omega;
  } else if (scaledPredicted % 2 == 0 && delta >= 0 || scaledPredicted % 2 != 0 &&
    delta <= 0) {
    return abs(delta) << 1;
  } else {
    return (abs(delta) << 1) - 1;
  }
}
```

Listing 3.5: Computing the mapped prediction residual

3.1.2. Encoding

The final step of the compression is to encode each of the samples produced from the prediction stage, and this will in turn reduce the overall size of the total image. The objective of the encoder is to reduce the amount of bits needed to represent each of the samples. The Sample adaptive encoder is implemented in listing 3.6 which implements the specified algorithm in section 2.1.2. The function *writeBits* will as the name explains only write a set amount of bits to the the output bitstream.

```
int encodeSampleAdaptive(uint32_t sampleToEncode, uint16_t * counter, uint64_t *
    accumulator, uint16_t x, uint16_t y, uint16_t z, unsigned int * totalWrittenBytes,
     unsigned int * numWrittenBits, uint8_t * compressedImage, struct arguments *
    parameters) {
    if(x == 0 && y == 0) {
        initSampleEncoder(z, counter, accumulator, parameters);
        writeBits(sampleToEncode, parameters->dynamicRange, numWrittenBits,
    totalWrittenBytes, compressedImage);
    } else {
        long kValue = log2((accumulator[z] + ((49*counter[z]) >> 7)) / counter[z]);
        kValue = kValue < 0 ? 0 : kValue;
        kValue = kValue > (parameters->dynamicRange - 2) ? parameters->dynamicRange -
    2 : kValue;
        long uValue = sampleToEncode >> kValue;
```

```
        if(uValue < parameters->uMax) {
            writeBits(0, uValue, numWrittenBits, totalWrittenBytes, compressedImage);
            writeBits(1, 1, numWrittenBits,totalWrittenBytes, compressedImage);
            writeBits(extractBits(sampleToEncode, kValue), kValue, numWrittenBits,
    totalWrittenBytes, compressedImage);
        } else {
            writeBits(0, parameters->uMax, numWrittenBits, totalWrittenBytes,
    compressedImage);
            writeBits(sampleToEncode, parameters->dynamicRange, numWrittenBits,
    totalWrittenBytes, compressedImage);
        }
        if(x+y != 0) {
            if(counter[z] < ((1 << parameters->yStar) - 1)) {
                accumulator[z] += sampleToEncode;
                counter[z]++;
            } else{
                accumulator[z] = (accumulator[z] + sampleToEncode + 1) >> 1;
                counter[z] = (counter[z] + 1) >> 1;
            }
        }
    }
    return 0;
}
```

Listing 3.6: Implemented sample adaptive encoder

3.2. Decompression

The CCSDS 123 Issue 1 standard was also made to be decompressing by decoding and using prediction to reproduce the original samples. This section will describe the implementation to decompress the images produced in the compression software. Decompression is done in a different manner from compression because compressed images can differ based on the parameters chosen. First the decompression tool will read the image header that has been written at the top of the bitstream that contains the parameters that was parsed in the argument parser. Listing 3.7 shows the top level decompression of images which involves a decoding and a unprediction step.

```
int decompressImage(FILE * compressedImage, FILE * decompressedImageFile, struct
    arguments * parameters) {
    for (uint16_t z = 0; z < parameters->zSize; z++) {
      for (uint16_t y = 0; y < parameters->ySize; y++) {
        for (uint16_t x = 0; x < parameters->xSize; x++) {
                    uint32_t tempResidual = decodeSampleAdaptive(counter, accumulator,
     x, y, z, compressedImage, parameters);
                    originalImage[offset(x,y,z, parameters)] = unPredict(originalImage
    , tempResidual, x, y, z, parameters, diffVector, weights);
                    if(parameters->pixelType == SIGNED) {
                        int16_t signedSample = originalImage[offset(x,y,z, parameters)
    ] - parameters->sMid;
                        fwrite(&signedSample, 2, 1, decompressedImageFile);
                    } else {
                        fwrite(&originalImage[offset(x,y,z, parameters)], 2, 1,
    decompressedImageFile);
                    }
        }
```

```
        }
    }
}
```

Listing 3.7: Decompression function

3.2.1. Decode

Decoding involves reproducing the mapped prediction samples produced by the sample adaptive encoder as described in section 2.1.2. The step to reproduce the samples invovlves reading the uValue $u_z(t)$ that was written to the bitstream as a value of N zeros until the '1' bit. Thus it can be determined how man bits to be read to fetch the encoded mapped prediction sample from the bitstream. Finally the accumulator and counter will be updated as described by the sample adaptive encoder. Listing 3.8 shows the implementation of the decoder whch will perform the operation as mentioned.

```
uint32_t decodeSampleAdaptive(uint16_t * counter, uint64_t * accumulator, uint16_t x,
    uint16_t y, uint16_t z, FILE * compressedImage, struct arguments * parameters) {
    uint32_t tempSample = 0;
    if(x == 0 && y == 0) {
        initSampleEncoder(z, counter, accumulator, parameters);
        tempSample = readBits(parameters->dynamicRange, compressedImage);
        return tempSample;
    } else {
        long kValue = log2((accumulator[z] + ((49*counter[z]) >> 7)) / counter[z]);
        kValue = kValue < 0 ? 0 : kValue;
        kValue = kValue > (parameters->dynamicRange - 2) ? parameters->dynamicRange -
    2 : kValue;

        uint16_t uValue = readNZeros(parameters->uMax, compressedImage);
        if(uValue > parameters->uMax) {
            tempSample = readBits(parameters->dynamicRange, compressedImage);
        } else {
            uint16_t tempBits = readBits(kValue, compressedImage);
            tempSample = (uValue << kValue) | tempBits;
        }

        if(counter[z] < ((1 << parameters->yStar) - 1)) {
                accumulator[z] += tempSample;
                counter[z]++;
            } else{
                accumulator[z] = (accumulator[z] + tempSample + 1) >> 1;
                counter[z] = (counter[z] + 1) >> 1;
        }
        return tempSample;
    }

}
```

Listing 3.8: Sample adaptive decoding of bitstream

3.2.2. UnPredict

Unprediction involves the step to reproduce a sample $s_{z,y,x}(t)$ from a mapped prediction residual after it has been decoded. Equation 2.25 shows how to produce the delta prediction residual in the prediction

stage, but for unprediction this step requires to reproduce the sample residual $\Delta_z(t)$. This can be done by inversing the mapped prediction residual computed in section 2.1.1.6 as shown in Equation 3.1[10]. The prediction stage will produce a scaled prediction and prediction sample which is used in the equation. $\theta_z(t)$ is calculated in a similar manner as shown in Equation 2.26.

$$\Delta(t) = \begin{cases} (\theta_z(t) - \delta_z(t))sgn^+(\hat{s}_z(t) - s_{mid}) & \text{if } \delta_z(t) > 2\theta_z(t) \\ \frac{\delta_z(t)+1}{2}(-1)^{\bar{s}_z(t)+\delta_z(t)} & \text{if } \delta_z(t) \leq 2\theta_z(t) \end{cases} \tag{3.1}$$

Finally the sample $s_z(t)$ can be reproduced by inverting the equation Equation 2.25 as this is shown in Equation 3.2.

$$s_z(t) = \Delta_z(t) + \hat{s}_z(t) \tag{3.2}$$

The implementation of the unprediction stage as mentioned is shown in Listing 3.9.

```
uint32_t unPredict(uint16_t * inputSample, uint16_t mappedResidual, uint16_t x,
    uint16_t y, uint16_t z, struct arguments * parameters, int64_t * diffVector,
    int32_t * weights) {
    /*
    Calculate local sum and build up the diffrential vector at a given sample.
  */
  if(x+y != 0) {
    BuildDiffVector(inputSample, diffVector, x, y, z, parameters, wideNeighborLocalSum
    );
  }
  /*
    Step for calculating prediction sample and scaledPredicted
  */
  int32_t scaledPredicted = computeScaledPredicted(inputSample, weights, diffVector,
    wideNeighborLocalSum(inputSample, x, y, z, parameters), x, y, z, parameters);
  int32_t delta = inverseMappedResidual(mappedResidual, scaledPredicted>>1,
    scaledPredicted, parameters);
  uint16_t sample = (scaledPredicted>>1) + delta;
  /*
    Update weights
  */
  if(x+y == 0) {
    initWeights(weights, z, parameters);
  } else {
    int32_t doubleResError = (sample << 1) - scaledPredicted;
    updateWeightVector(weights, diffVector, doubleResError, x, y, z, parameters);
  }
  return sample;
}

int32_t inverseMappedResidual(uint32_t mappedResidual, uint64_t predictedSample,
    uint32_t scaledPredicted, struct arguments * parameters) {
    int64_t omega = 0;
  int64_t temp1 = predictedSample - parameters->sMin;
  int64_t temp2 = parameters->sMax - predictedSample;

    omega = temp1 > temp2 ? temp2 : temp1;

  if(mappedResidual > (omega<<1)) {
    int sgn = sgnPlus(predictedSample-parameters->sMid);
    return (omega-mappedResidual)*sgn;
```

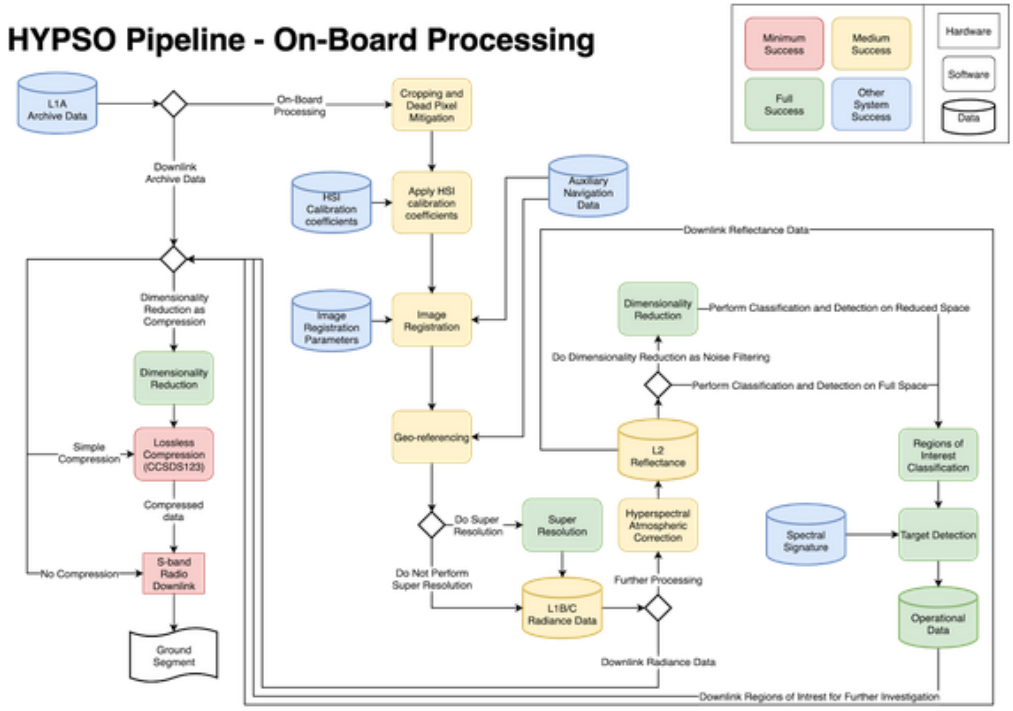
```
36    } else {
37      if((mappedResidual+scaledPredicted) % 2 == 0) {
38        return ((mappedResidual+1) / 2);
39      } else {
40        return -1 * ((mappedResidual+1) / 2);
41      }
42    }
43  }
```

Listing 3.9: Unprediction top level function

3.3. Embedded Version

The embedded version is implemented as a function in the HYPSO software running on-board HYPSO mission, it only contains the compressImage function as similar to the one implemented in Section 3.1 but with limited capabilities to change parameters in its current status. The primary design focus is using the user defined parameters similar to the one implemented in the FPGA version as this is defined during synthesis. Figure 3.2 shows the total design of the on-board processing but the focus is on the Lossless Compression where it used as a stage in the pipeline before transmission or local storage. For the satellite it is very important to use the compression to reduce the pressure on the transmission through the on-board antenna. Uncompressed images can become a problem for slow speeds on the antenna.

Figure 3.2.: Hypso Pipeline



Listing 3.10 provides the top level function *compressImage* as implemented in the pipeline. The implementation is at the top level shown in the figure 3.1. In the for loop for line 35 to 45 it will predict and encode every individual pixel in the image. Similar to the implementation of CCSDS 123 Issue 2 with argument parsing it would initialise default parameters to be used for the algorithm. This is different in this implementation as these parameters are initialised by the HYPSO software. The intention for this is that the parameters will not change often as the mission is operational. Once the satellite has been launched there is the possibility of downloading an uncompressed image and optimise the parameters to provide the optimal results for the mission. The second point is to be similar to the FPGA at every time to provide similar results, but if the FPGA changes then it is possible to update the HYPSO software.

```
1  int compressImage(
2      uint16_t * image,
3      uint8_t * compressedImage,
4      unsigned int * totalWrittenBytes,
5      compr_config_sw_t * parameters) {
6      int error = 0;
7
8      // Allocate memory. If any of these are null then it should clean up.
9    int32_t * weights = malloc((parameters->mode != REDUCED ? parameters->precedingBands
       +3 : parameters->precedingBands) * sizeof(int32_t));
10   int32_t * diffVector = malloc((parameters->mode != REDUCED ? parameters->
```

```
    precedingBands+3 : parameters->precedingBands) * sizeof(int32_t ));
  uint16_t * counter = malloc(sizeof(uint16_t)*parameters->zSize);
  uint64_t * accumulator = malloc(sizeof(uint64_t)*parameters->zSize);


    if (weights == NULL) {
        error = 1;
        goto error;
    }
    if (diffVector == NULL) {
        error = 1;
        goto error;
    }
    if (counter == NULL) {
        error = 1;
        goto error;
    }
    if(accumulator == NULL) {
        error = 1;
        goto error;
    }
  unsigned int numWrittenBits = 0;

    // Add the image header
    writeImageHeader(&numWrittenBits, totalWrittenBytes, compressedImage, parameters);
    for (uint16_t y = 0; y < parameters->ySize; y++) {
        for (uint16_t x = 0; x < parameters->xSize; x++) {
            for (uint16_t z = 0; z < parameters->zSize; z++) {
                uint32_t residuals = predict(image, x, y, z, parameters, diffVector,
    weights);
                encodeSampleAdaptive(residuals, counter, accumulator, x, y, z,
    totalWrittenBytes, &numWrittenBits, compressedImage, parameters);
                //printf("    x: %i, y: %i, z: %i\n", x, y, z);
            }
        }
    }
  /*
    Fill the final bits to reach the word size of the computer
  */
  fillBits(&numWrittenBits, totalWrittenBytes, compressedImage, parameters);

    // Free on NULL pointers is regarded as NOP
error:


  free(accumulator);
  free(counter);
  free(weights);
  free(diffVector);
  return error;
}
}
```

Listing 3.10: CCSDS 123 Issue 1 Embedded Implementation

4. CCSDS 123 Issue 2 Improvements

As it was mentioned in section 2.2 there was still features that was required to be made for the software to be functioning properly and to perform the baseline compression of near-lossless. This chapter will describe the new improvements to this software that was made for this thesis. The software implementation can be found in Appendix D. The main improvements that was required was described in the future work in section 2.2 and described as such:

- Support different bit sizes, D. The implementation only supports 16-bit image data.
- The software implementation requires to support signed numbers and the implementation only supports unsigned.
- Argument parsing does not support all user-defined parameters, some of these are set to a default value in the argument parser.
- Currently the software only support BSQ. An important feature is to support BIP and BIL.
- The hybrid encoder requires future work to solve the issues regarding the codewords and a better testing solution. A decoder is a possible testing solution.
- The software implementation needs a decompression implementation to be able to read a compressed stream. The sample adaptive already has a possible decoder solution from issue 1, but the predictor for CCSDS 123 Issue 2 does not have a depredictor and requires an implementation.

This chapter will address these improvements that was required to perform a baseline compression of near-lossless, and it will not include all features of the CCSDS 123 Issue 2 standard.

4.1. Supporting different bit sizes

Supporting bit-sizes larger than 16-bits would require changing the types used in the compression software to accommodate the bitsizes. The CCSDS 123 Issue 1 green paper in [10] addresses the bit-sizes for CCSDS 123 Issue 1, but Issue 2 changes this because the maximum bit size is 32-bit. In the programming language C it would require to use 32-bit or 64-bit types to expand the bit-sizes to prevent overflows. A bug with overflow was still occurring even if the types are changed. Listing 4.1 shows the implementation of the scaled predicted sample which had an overflow bug which might not be noticeable at first. The primary problem is by left shifting the sample in line 6. This line will left shift a 32-bit sample value by 1 and store the result in 32-bit register. Followed by this the resulting 32-bit register will be stored into the 64-bit variable doubleResPredSample. If the result of the left shift overflows then it is because of not left shifting a 64-bit register.

```
1 uint64_t computePredictedSample(uint32_t * sample, uint64_t * doubleResPredSample,
      uint64_t highResPredSample, uint16_t x, uint16_t y, uint16_t z, struct arguments *
      parameters) {
```

```
  if(x+y == 0) {
    if(z == 0 || parameters->precedingBands == 0) {
      (*doubleResPredSample) = parameters->sMid<<1;
    } else {
      (*doubleResPredSample) = sample[offset(x,y,z-1,parameters)]<<1;
    }
  } else {
    (*doubleResPredSample) = highResPredSample >> (parameters->weightResolution+1);
  }
  return (*doubleResPredSample)>>1;
}
```

Listing 4.1: Overflow bug

A simple solution used in the C programming language is by using the method casting which will promote one of the variables into a selected size. The result for the selected size will therefore promote all variables used in the calculation. Listing 4.2 fixes this issue due to the specification of C99 integer promotion rules will promote the results into a 64-bit result [12].

```
uint64_t computePredictedSample(uint32_t * sample, uint64_t * doubleResPredSample,
    uint64_t highResPredSample, uint16_t x,
  if(x+y == 0) {
    if(z == 0 || parameters->precedingBands == 0) {
      (*doubleResPredSample) = parameters->sMid<<1;
    } else {
      (*doubleResPredSample) = (uint64_t)sample[offset(x,y,z-1,parameters)]<<1;
    }
  } else {
    (*doubleResPredSample) = highResPredSample >> (parameters->weightResolution+1);
  }
  return (*doubleResPredSample)>>1;
}
```

Listing 4.2: Fixed overflow bug

4.2. Support for signed integers

Another feature is to implement the functionality to read signed integer images as there is a possibility a camera will capture in signed integers. Instead of creating multiple versions to support integer casting there was a simpler solution to convert signed integers to unsigned integers. Because the bit-size is already known at the beginning it was simply to convert signed integers to unsigned integers by adding the value s_{mid} to an signed integer and store the result in an unsigned integer. Using the unsigned value of s_{mid} as calculated in section 2.1 where the D-bit value is the bit-size. The implementation of this is as shown in Listing 4.3.

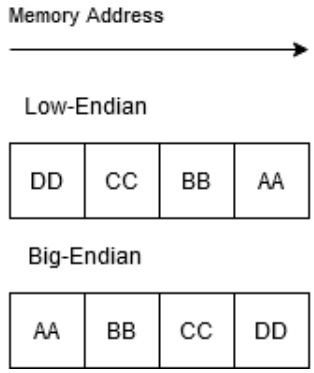
```
if(parameters->pixelType == SIGNED) {
    samples[readbytes] = buffer + parameters->sMid;
} else {
    samples[readbytes] = buffer;
}
```

Listing 4.3: Conversion of signed integers

4.3. Conversion of Endianess

A feature that was not required at the future work but was implemented anyways is the conversion of big-endian numbers to low-endian. An image created by a big-endian machine without conversion used on a low-endian machine would make the compression produce unexpected or worse results due to how the samples are stored in memory. A big-endian machine stores the most significant byte as the first byte with an increasing order, and low-endian will store the most-significant byte at the end. For example the 32-bit value $0xAABBCCDD$ would be stored as DD, CC, BB and AA in memory. This is visually represented in Figure 4.1 where the memory address shows an increasing order towards the right and the different Endianess and how they store $0xAABBCCDD$.

Figure 4.1.: Big-endian vs Low Endian in memory



Converting a big-endian number to a low-endian is performed by shift operations which moves the bytes to their proper locations. AA would be right shifted to the right by 24 places, BB is shifted to the right by 8 places, CC is left shifted by 8 places and finally DD is left shifted by 24 places. GCC provides a builtin function which provides this functionality as shown in Listing 4.4 which will swap a 32-bit integer.

```
#define __bswap_constant_32(x) \
     ((((x) & 0xff000000) >> 24) | (((x) & 0x00ff0000) >>  8) |                  \
      (((x) & 0x0000ff00) <<  8) | (((x) & 0x000000ff) << 24))
```

Listing 4.4: Converting endianess

4.4. Support for varying image order and encoding order

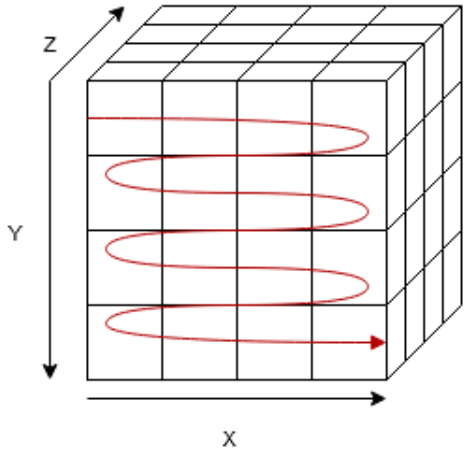
Supporting BSQ, BIL and BIP introduces two problems for compression to perform. First is reading images that are in such order which they are stored in memory as they are specified in section 2.1. As the programming language C does not allow to easily use a 3-dimensional array without pointer arithmetic it was simpler to use a one-dimensional array, and to use the parameters x, y, z to denote the coordinates within this one dimensional array. An offset function was made to perform the conversion from BSQ,BIL or BIP to 1-Dimensional coordinate. This function is as shown in Listing 4.5 which will read the image decided by the user.

```
int offset(uint16_t x, uint16_t y, uint16_t z, struct arguments * args) {
    if(args->imageOrder == BSQ) {
        return (z * args->xSize * args->ySize) + (y * args->xSize) + x; // BSQ
    } else if (args->imageOrder == BIP) {
        return (args->xSize*args->zSize*y) + (args->zSize*x) + z; // BIP
    } else {
        return (args->xSize*args->zSize*y) + (args->xSize*z) + x; // BIL
    }
}
```

Listing 4.5: Offset function for reading different image orders

Compression also needs to support encoding in BSQ, BIL and BIP which describes the order of encoding pixels. This does not affect what the input image order is stored in as the compression tool can read an BSQ image and encode it as BIP, or any other combination. The CCSDS 123 standard allows for encoding sub-frames within frames but this was not necessary to use and was neglected from the implementation. Encoding in the different orders is simply the way the top level for loop iterates through the image for encoding. For example the encoding of BSQ would iterate through the spatial (x, y) coordinates first followed by each band z as this is shown in Figure 4.2.

Figure 4.2.: Big-endian vs Low Endian in memory



The pseudo-code performing this encoding order is as shown in Listing 4.6 which performs this encoding order.

```
for (uint16_t z = 0; z < parameters.zSize; z++) {
  for (uint16_t y = 0; y < parameters.ySize; y++) {
    for (uint16_t x = 0; x < parameters.xSize; x++) {
        //Predict/Encode
    }
  }
}
```

Listing 4.6: Encoding of BSQ

BIP encoding order is the encoding of spectral z followed by spatial x first and finally the spatial coordinate z. The pseudo code for performing this encoding order is as shown in Listing 4.7.

```
for (uint16_t y = 0; y < parameters.ySize; y++) {
  for (uint16_t x = 0; x < parameters.xSize; x++) {
    for (uint16_t z = 0; z < parameters.zSize; z++) {
    //Predict/Encode
    }
  }
}
```

Listing 4.7: Encoding of BIP

Finally the encoding order for BIL is the encoding of spatial coordinate x followed by the spectral coordinate z and finally the spatial coordinate y. The pseudo code for BIL encoding order is as shown in Listing 4.8.

```
for (uint16_t y = 0; y < parameters.ySize; y++) {
  for (uint16_t z = 0; z < parameters.zSize; z++) {
    for (uint16_t x = 0; x < parameters.xSize; x++) {
    //Predict/Encode
    }
  }
}
```

Listing 4.8: Encoding of BIL

4.5. Fixing the hybrid encoder

The hybrid encoder was developed and almost finished in the project thesis as mentioned in section 2.3, but there was an issue regarding the code tables where at some points it did not provide the correct codes at the output. The bug occured because of a problem with the implementation and error handling. The problem with the implementation is that there are 12 arrays of 258 characters that are declared as a global variable as this is shown in Listing 4.9. The activeprefix will be appended with the hexadecimal character according to the standard described in section 2.2 but a problem will occur if there is no valid code and never will be reset. If a code is never found then it will be appended forever until it will overwrite the string terminator and start overwriting other active-prefixes. This problem occured especially with codetable 12 where it never stopped and with some more research there was a problem with the documentation of [4]. Codetable 12 has 106 unique codes as this is shown in Appendix E, but it should be 109 codes as provided in [13]. After contacting Ian Blanes which is one of the co-auhtors on [13] it was true that there is some missing codes. Thankful to this contact it was provided a location of the correct code tables which is provided in [14] which can also be found included in Appendix D.

```
char activePrefix[16][258] = {"", "", "", "", "", "", "", "" , "" , "", "" , "" , "" ,
    "", "" ,""};
uint8_t codeIndex[16] = {12, 10, 8, 6, 6, 4, 4, 4, 2, 2, 2, 2, 2, 2, 2, 0};
```

Listing 4.9: Array of activeprefixes

4.6. Decompression

With the limited time before delivering the thesis the development of a hybrid decoder was not possible and is therefore neglected. Even so the unprediction stage was still feasible to implement.

4.6.1. Unpredict

After decoding of a mapped quantizer index then it possible to reconstruct $s_z(t)$ in the unprediction step. However it is not entirely possible to reconstruct $s_z(t)$ if lossy is chosen where $m_z(t) \neq 0$ and the original $s_z(t)$ is not similar to the reconstructed one. Unprediction uses the prediction stage to use the same calculations as this will reproduce the statistics as if it were in compression. There is some differences as the step will require to reverse steps that was done in compression and this will be described in this subsection. The software top level calculation differs from the one mentioned in compression and the unprediction is performed as such:

- Compute the local sum for $t > 0$ (Will store all the sum in memory)
- Compute the local differences and create the diffrences vector for $t > 0$
- Compute high resolution prediction sample for $t > 0$
- Compute double resolution prediction sample
- Compute the predicted sample value
- Compute the fidelity control for $m_z(t)$
- Compute the inverse mapped quantizer index to reproduce $q_z(t)$ from $\delta_z(t)$
- Compute the dequantization to reproduce $\Delta_z(t)$ from $q_z(t)$
- Compute the clipped bin center
- Weight update/Weight init
- For t = 0 return $s_z(t) = \Delta_z(t) + \hat{s}_z(t)$.
- For t > 0 return the clipped bin center

The introduction to the unprediction is the inverse of mapped quantizer index to reproduce $q_z(t)$ and dequatization to reproduce $\Delta_z(t)$. The other implementations reuses the implementations for compression from section 2.2 and is therefore negltected in this section. The decoding of the bitstream will provide a mapped quantizer index $\delta_z(t)$ that is used in reproducing the original sample $s_z(t)$. After the step of prediction then the inverse of mapped quantizer index to calculate the quantizer index $q_z(t)$ is possible. The equation for calculation of the mapped quantizer index is already introduced in Equation 2.61 where the reverse for calculating the quantizer index $q_z(t)$ is in Equation 4.1.

$$q_z(t) = \begin{cases} (\theta_z(t) - \delta_z(t))sgn^+(\hat{s}_z(t) - s_{mid}) & \text{if } \delta_z(t) > 2\theta_z(t) \\ \frac{\delta_z(t)+1}{2}(-1)^{\tilde{s}_z(t)+\delta_z(t)} & \text{if } \delta_z(t) \leq 2\theta_z(t) \end{cases} \tag{4.1}$$

```
1 int64_t inverseMappedResidual(uint32_t mappedResidual, uint64_t predictedSample,
      uint64_t doubleResPredSample, uint32_t maximumError, uint16_t x, uint16_t y,
      uint16_t z, struct arguments * parameters) {
2     int64_t omega = 0;
3   int64_t temp1 = predictedSample - parameters->sMin;
4   int64_t temp2 = parameters->sMax - predictedSample;
```

```
  if (x == 0 && y == 0) {
    omega = temp1 > temp2 ? temp2 : temp1;
  } else {
    temp1 = ((temp1 + maximumError) / ((maximumError << 1) + 1));
    temp2 = ((temp2 + maximumError) / ((maximumError << 1) + 1));
    omega = temp1 > temp2 ? temp2 : temp1;
  }

  if(mappedResidual > (omega<<1)) {
    int sgn = sgnPlus(predictedSample->parameters->sMid);
    return (omega-mappedResidual)*sgn;
  } else {
    if((mappedResidual+doubleResPredSample) % 2 == 0) {
      return ((mappedResidual+1)>> 1);
    } else {
      return -1 * ((mappedResidual+1)>> 1);
    }
  }
```

Listing 4.10: Dequantization

Calculating the predicted residual $\Delta_z(t)$ imposes some problems when reconstructing the original sample $s_z(t)$. Reversing the equation for calculating the quantizer index in Equation 2.48 is problematic because of the maximum error $m_z(t)$ where the reversed equation is in Equation 4.2. The introduction of $\pm m_z(t)$ provides an error where it is not possible to determine if it should be added or subtracted from the final answer, and this provides no more than $m_z(t)$ units of error when reconstructing samlpe $s_z(t)$. Therefore the implementation is neglegting this from the final equation as this is shown in Equation 4.3.

$$\Delta(t) = \begin{cases} q_z(t) & \text{if } t = 0 \\ 2 * q_z(t) * m_z(t) + q_z(t) \pm m_z(t) & \text{if } t > 0 \end{cases} \tag{4.2}$$

$$\Delta(t) = \begin{cases} q_z(t) & \text{if } t = 0 \\ 2 * q_z(t) * m_z(t) + q_z(t) & \text{if } t > 0 \end{cases} \tag{4.3}$$

Listing 4.11 shows the implementation of Equation 4.3.

```
int64_t deQuantizizer(int64_t quantizerIndex, uint32_t maximumError, uint16_t x,
    uint16_t y) {
  if(x+y == 0) {
    return quantizerIndex;
  } else {
    return (maximumError * quantizerIndex<<1) + quantizerIndex;
  }
}
```

Listing 4.11: Dequantization

After the delta residual has been calculated then it is possible to reproduce the sample $s_z(t)$ from $\Delta_z(t)$. By reversing the equation in Equation 2.47 as this is shown in Equation 4.4.

$$s_z(t) = \Delta_z(t) + \hat{s}_z(t) \tag{4.4}$$

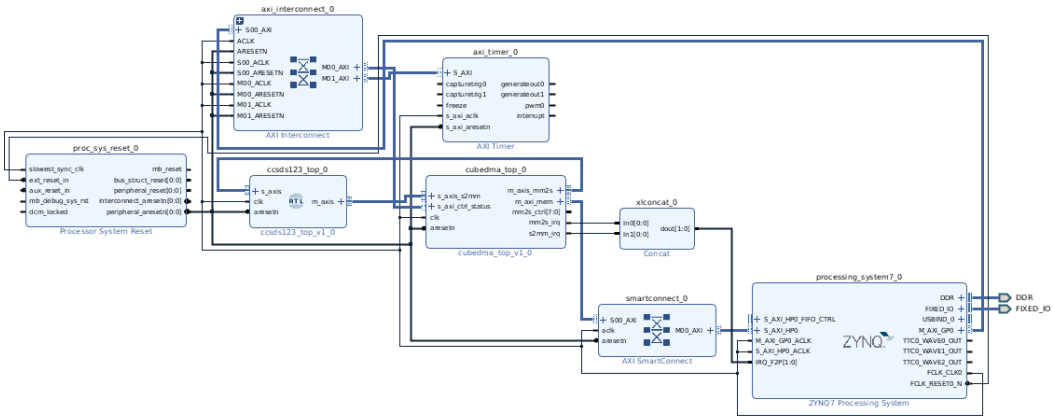
For t>0 then the reproduction of sample $s_z(t)$ is the calculated clipped bin center.

5. Hypso on-board processing testing and changes

As HYPSO mission is close for launch there was several tasks that was necessary for making a successful mission on the on-board processing pipeline. A Direct Memory Access fpga core and a CCSDS 123 Issue 1 core on the FPGA has been previously made by Johan Fjeldtvedt in his master thesis on HYPSO, where this thesis can be found in [2]. Multiple publications regarding this implementation of CubeDMA and the CCSDS 123 Issue 1 core has also been made in [15, 16, 17]. During the development the integration of these systems has been primarily in a embedded system. It required to be implemented into the linux system on the HYPSO satellite. One problem is that there had not been any integration tests in the HYPSO system and it was necessary to perform the tests. This chapter will go through how the system is designed and the necessary Linux kernel driver that is made to accommodate the pipeline. The system is run on an ZYNQ 7000 SoC that is built up using an ARM CPU and a Xilinx FPGA. Xilinx has developed a software for developing on their FPGAs called Vivado. It is assumed the reader has some knowledge regarding the tool Vivado.

The integrated system on the FPGA which is developed by the HYPSO team is shown in figure 5.1. The system presents the integration of CCSDS and CubeDMA as a custom IP and integrated with the CPU. Note that the CubeDMA has full access to the Double Data Rate memory through the AXI_HP0 ports which is a slave port using the Advanced eXtensible Interface developed by ARM. CubeDMA is also configured by the Master General Purpose port, and after cubeDMA has been started it will transfer data from DDR memory to CCSDS, and from CCSDS to DDR memory. This system is used for all verifications of hardware with the exception of CubeDMA tests. This test has removed CCSDS from the FPGA and connects the port s_axis_s2mm port of CubeDMA to m_axis_mm2s port of CubeDMA as shown in Figure 5.1. The system does have support for interrupts of completed transfers of CubeDMA but these are not used to lower the necessary development time on the kernel driver HYPSO mission it was deemed unnecessary.

Figure 5.1.: Zynq System Setup in Vivado



5.1. Communication with CubeDMA

Due to the troubles with integrating and communicating with CubeDMA it was necessary to verify and rewrite the code for communication with this module. This section will describe what CubeDMA does and how to use it. As mentioned the CubeDMA is an FPGA core to perform direct memory accesses to offload these operations from the CPU. Allowing the CPU to continue working on other workloads while waiting for the DMA to finish. The usage of a DMA can be very beneficial when there are many memory accesses necessary, and for hyperspectral images this can be immense. The CubeDMA was designed to transfer hyperspectral images, pixel by pixel, from the memory to a destination, and the destination can be a computation core or itself. The focus of CubeDMA will be the software side of using this core and the knowledge regarding the hardware is not necessary. Details regarding the hardware design is referred to the thesis in[2], which also details how to use it in an embedded environment but not a Linux system.

Programming the CubeDMA is allowed through memory mapped registers stored inside the FPGA core allowing a CPU to communicate and control the CubeDMA. These memory mapped registers are fixed addresses which a CPU will translate as a communication to the FPGA instead of as a normal memory access. The CPU translation map can be shown in Figure 5.2 which contains all the address translations. The addresses which will translate to communication to the FPGA is PL AXI slave port 0 which is addresses **0x4000_0000** to **0x8000_0000** where 1 is not used. The specific address in this range can be customised in the Vivado tool for configuring the system but in the HYPSO system it is currently fixed to **0x4C00_0000** to **0x4C00_FFFF**. This means 64k of 32-bit addresses can be used, but not all are used as this was the default by Vivado.

Figure 5.2.: Zynq 7000 CPU memory map[5]

Start Address	Size (MB)	Description
0x0000_0000	1,024	DDR DRAM and on-chip memory (OCM)
0x4000_0000	1,024	PL AXI slave port #0
0x8000_0000	1,024	PL AXI slave port #1
0xE000_0000	256	IOP devices
0xF000_0000	128	Reserved
0xF800_0000	32	Programmable registers access via AMBA APB bus
0xFA00_0000	32	Reserved
0xFC00_0000	64 MB - 256 KB	Quad-SPI linear address base address (except top 256 KB which is in OCM), 64 MB reserved, only 32 MB is currently supported
0xFFFC_0000	256 KB	OCM when mapped to high address space

As the addresses are defined then it is necessary to know how to communicate with CubeDMA. Table 5.1 and Table 5.2 describes each registers address and the description. Not all registers are used and the focus will be on the implementation for HYPSO. The source part of the CubeDMA is described by Table 5.1 where the most important registers are; Control, Status, Base address, Cube Dimension and row size. Control register allows to start the DMA engine which will make the DMA transfers begin. If a transfer is complete then the status register will notify if it is done or if an error is triggered. As the DMA has access to the DDR memory that the CPU also uses it is necessary to define a base address from where the DMA will read. This is defined through the base address register. The DMA will handle how many addresses to read through the cube dimension register which is the total size of the hyperspectral image. Width describes the X axis, height is the Y axis and depth is the Z axis in a spectral cube. Finally the row size is necessary for the CubeDMA to function which is the multiplication X*Z. Finally once these registers are defined it is necesarry for configuring CubeDMA for writing to a destination which has fewer registers. As this is shown in Table 5.2 the registers are Control, Status, Base Address and Received length. Control registers will allow the core to write to a destination. Status register will notify if a transfer is done or an error has occurred. It is important to define the base address which the CubeDMA will write to a destination address. Finally the received length address will define the number of bytes which has been written to the CubeDMA core which can be less or larger than the cube size register defined. With these registers defined then it is possible to use the CubeDMA.

Table 5.1.: Part 1 of memory map registers for CubeDMA[2]

Field	Description	Unit	Bits
Control register (0x00)			
Start	Core starts transfer when this bit transitions from 0 to 1		0
Block-wise mode	Cube is read in blocks of specified size		2
Plane-wise mode	Cube is read planewise, with a given number of planes in parallel		3
Error IRQ enable	Trigger IRQ when error condition arises		4
Completion IRQ enable	Trigger IRQ when transfer is complete		5
Number of plane transfers	How many plane transfers to perform		15 - 8
Start offset	Plane offset to start transferring from	c	23 - 16
Status register (0x04)			
Transfer done	Indicates whether the transfer is completed		0
Error mask	Indicates which errors occured		3 - 1
Error IRQ flag	Set when IRQ was triggered due to error. Cleared when 1 is written to this bit.		4
Completion IRQ flag	Set when IRQ was triggered due to completion. Cleared when 1 is written to this bit.		5
Base address register (0x08)			
Base address	The address of the first component in the first pixel of the HSI cube	b	31 - 0
Cube dimension register (0x0C)			
Width	The width of the HSI cube	p	11 - 0
Height	The height of the HSI cube	p	23 - 12
Depth	The depth of the HSI cube	c	31 - 24
Block dimension register (0x10)			
Block width	$\log_2$ of the width of each block	p	3 - 0
Block height	$\log_2$ of the height of each block	p	7 - 4
Last block row size	Number of components in each row of the last block in a row	c	31 - 12
Row size register (0x14)			
Row size	Number of components in one row of the cube	c	19 - 0

Table 5.2.: Part 2 of memory map register for CubeDMA[2]

Field	Description	Unit	Bits
Control register (0x20)			
Start	Core starts transfer when this bit transitions from 0 to 1		0
Error IRQ enable	Trigger IRQ when error condition arises		4
Completion IRQ enable	Trigger IRQ when transfer is complete		5
Status register (0x24)			
Transfer done	Indicates whether the transfer is completed		0
Error mask	Indicates which errors occured		3 - 1
Error IRQ flag	Set when IRQ was triggered due to error. Cleared when 1 is written to this bit.		4
Completion IRQ flag	Set when IRQ was triggered due to completion. Cleared when 1 is written to this bit.		5
Base address register (0x28)			
Base address	The address of where to store the incoming stream data	b	31 - 0
Received length register (0x2C)			
Received length	The number of bytes received from start of transfer until TLAST was asserted	b	31 - 0

The address for the registers is the address defined within the parentheses (0xXX) which will define the offset address from the base address **0x4C00_0000**. This offset allows the programming language C to create an array to access each address individually. As long as the array is an 32bit then the increments array[0], array[1] and array[2] will be 0x4 each. Using this method in an embedded system without any operating system is a lot easier as the memory management is handled by the program. This is not the case for the HYPSO mission as it runs a Linux subsystem which allows to use varied methods such as using a Linux kernel driver. For this case however the easiest method was to directly use the Linux memory driver */dev/mem* to access the physical memory of the system. This is performed as shown in Listing 5.1 where mmaping will create a virtual address from the phsyical address CUBEDMA_BASE.

```
1 #define CUBEDMA_BASE 0x43C00000
2 int fd = open("/dev/mem", O_RDWR|O_SYNC);
3
4 deviceMem = (uint32_t *) mmap(NULL, getpagesize(), PROT_READ|PROT_WRITE, MAP_SHARED,
      fd, CUBEDMA_BASE);
```

Listing 5.1: Memory access to cubeDMA

5.2. Linux Kernel Driver

The Linux kernel driver has been previously made by Andreas Varntresk on a previous master thesis for the HYPSO mission, where this thesis can be found in [18], but due to limited time it had not been completed entirely and was prone for error. This section will not go into much details regarding the implemented kernel driver, but only point out some changes that was made to it to prevent kernel panics and general errors. One important note is that the Linux kernel used for this project has been changed to make sure the contiguous memory allocation of hyperspectral images is done correctly. A range of 256MiB has been reserved from the kernel to make sure the range is never used. This was done because there was some issues trying to integrate it without changing the linux kernel and proved to crash often. The reservation of the memory region is done by reducing the amount of memory "accessable" to the linux kernel from approximately 1 GB to 756 MB when creating the linux kernel. Adresses 0x0000_0000 to 0x3000_0000 is therefore available to the linux kernel, while the rest 0x3000_0000 to 0x4000_0000 is not available to the Linux kernel but can be used. Figure 5.2 shows the CPU addresses map that is available for the CPU to use where 0x0000_0000 to 0x4000_0000 is used for DDR DRAM. Table 5.3 shows how the memory region is structured. The whole memory region defines a 256MB of reserved storage but it does not mean it will use all of it. This is a safety prevention if the size of the images will change in the future. The code for this Linux device driver is as shown in Appendix B.

Table 5.3.: Memory structure of reserved memory

Memory Start	Memory End	Description
0x000_00000	0x3000_0000	Linux Ram
0x3000_0000	0x4000_0000	CubeDMA addresses

The information regarding how to make kernel drivers is huge and a lot of the information regarding this can always be outdated or changed as the linux is continuously updated. However the fundamentals of designing Linux device drivers for this project is gained from the book Linux Device Drivers [19].

5.2.1. Opening and closing the device

Inserting the kernel module driver into the Linux system at runtime can be done by using the command *insmod*. This function is a linux command that inserts linux kernel drivers at runtime. The command will run the MODULE_INIT of the kernel driver which calls the function *ebbchar_init* in the C code. This will do the standard init procedure of kernel drivers to insert it into the kernel at runtime. From the previous work of Andreas Varntresk there has not been much additions to this function other than adding the function *request_mem_region* which is a function that will reserve a memory region from the linux kernel. It serves the purpose of preventing kernel drivers of allocating memory on top of each other. Releasing this memory region can be done by using the function *release_mem_region* in the exit function of the kernel driver. The linux command for inserting the linux kernel driver is done as shown in Listing 5.2.

Listing 5.2: Linux command for inserting linux driver

```
$ insmod /lib/modules/4.19.0-xilinx-v2019.1/extra/cubedma.ko
```

Using the rmmod command in the linux terminal will remove the device from the linux kernel and requires proper memory cleanup to prevent memory leaks and other issues. An issue occured that it was not possible to do insmod after using the command rmmod to remove the driver. This was easily fixed by adding the function *memunmap* to the exit function as it was not cleaning up properly when tested. This function is used to cleanup the linux function *memremap* which is used when the driver is inserted. Once this was changed, the driver would properly cleanup and removed the potential memory leak in the driver. In the HYPSO system the removal of this kernel driver is performed as shown in Listing 5.3.

Listing 5.3: Linux command for removing linux driver

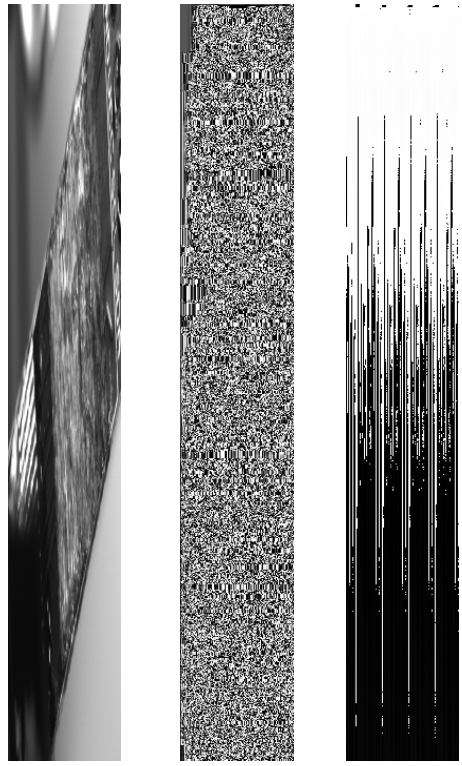
```
$ rmmod /lib/modules/4.19.0-xilinx-v2019.1/extra/cubedma.ko
```

6. Results

When testing the compression algorithms there is the possibility of testing on random data but this is problematic as the compression algorithm does not compress well on random noise at least for compression rates. Especially since the algorithm is not designed to compress random noise but rather hyperspectral images. Therefore for this thesis it was easier to use hyper-spectral images. The testing method used in this thesis is not to determine the optimal compression parameters, but rather to test for faults and correctness. The aim for this testing is to provide a well tested compression software for usage by others. For lossless compression the method to verify a compression algorithm is to run the compression software on a hyperspectral image or random data. If the image is able to decompress the image back to the original then the software is determined to be correct or at least working for that dataset. This does not eliminate the possibility of a small bug in the software that is not detected for other images.

6.1. CCSDS 123 Issue 1 Results

As the CCSDS 123 Issue 1 software was implemented on the HYPSO software as an embedded implementation and another as a standard version. It was chosen to focus the results on the HYPSO software. Decompressing images produced from the HYPSO software does require the standard software for verification. An example image produced from the HYPSO software is as shown in Subfigure 6.1(a). This is testing with the integrated system of HYPSO software where the camera in use is not the hyperspectral camera but a monochrome camera. As such it captures only monochrome image and each band is just another capture so the compression output from these captures will not be ideal. The compression algorithm CCSDS 123 Issue 1 is difficult to debug and verify primarily because of everything can compute correctly until a point in the compression. One of the faults can be presented in Subfigure 6.1(b) where the original image should be Subfigure 6.1(a). This particular issue was with compression under BIP mode as the weight vector was not set up properly where there was only one weight vector stored for all bands. Once each weight was set up to be using their own weight vector the problem was removed. Another issue that would occur is improperly initialising variables and vectors which could cause compression faults as this is shown in Subfigure 6.1(c). All these faults are primarily fault in the implementation of the algorithm where there is also the possibility of overflow issues, but the chosen integer types will cover all the possibility overflows as the bit-sizes are described in [10].



(a) Successful test

(b) Fault in implementation

(c) Another fault in implementation

Figure 6.1.: HYPSO compression test of CCSDS 123 Issue 1

6.2. CCSDS 123 Issue 2 Results

As the software did not implement the full standard with every possible feature provided by the standard it was decided to provide a general overview of what is supported in the software implementation. The overview is summarised for general features in Table 6.1, prediction features in Table 6.2, the fidelity control features in Table 6.3 and the encoder features in Table 6.4 where the feature is described and the status of the feature. Some features of the standard might not be described in the tables and it is fair to assume it is not implemented. Note that the software does not support decoding to other orders. Currently if one encodes a BSQ image in BIP order then the decoded image will become a BIP image.

Table 6.1.: General status of the software

General Features	Status
Error Handling to prevent illegal values	Not Implemented
Compression/Decompression of HSI images	Implemented
Reading BSQ/BIP/BIL	Implemented
Encoding BSQ/BIP/BIL	Implemented
Sub-frame Encoding	Not Implemented
Decoding BSQ/BIP/BIL to other encoding orders	Not Implemented
Config File for custom initializations	Not Implemented

Some features of the prediction was not implemented and this is summarised in Table 6.2. Primarily the parts for custom initialisation and initialisation for bands is not implemented. This was due to the practical issues with argument parsing with vector values which should have been in an config file.

Table 6.2.: Status of prediction

Prediction Features	Status	Note
Prediction according to the CCSDS 123 Standard	Implemented	
Wide Neighbor Oriented local sum	Implemented	
Narrow Neighbour local sum	Implemented	
Wide Column Local Sum	Implemented	
Narrow Column Local Sum	Implemented	
Init for damping and offset for each band, in sample representation	Not Implemented	Only support fixed values
Custom Weight initialization	Not Implemented	

Not all features of fidelity control for calculating $m_z(t)$ was not implemented and this is summarised in Table 6.3. The same problem with the implementation of the prediction is the vector values which would require a config file. Therefore the band dependent values was not implemented.

Table 6.3.: Status of Fidelity Control

Fidelity control Features	Status
Absolute Error	Implemented
Relative Error	Implemented
Absolute and Relative Error	Implemented
Band dependent error for Absolute/Relative	Not Implemented
Band independent error for Absolute/Relative	Implemented
Periodic Error Limit Updating	Not Implemented

Finally the status of the implementations of the encoders is summarised in Table 6.4 where it shows the varied encoders in the standard. As it is mentioned the hybrid decoder was not implemented because there was not enough time to work on it. Even so the hybrid encoder is implemented which can be used

for testing the new encoder and the greater benefits to compression. Note that the block adaptive encoder is not implemented as this was decided not to be used because of the greater benefits by using sample adaptive or hybrid.

Table 6.4.: Status of Encoder

Encoding Features	Status
Sample Adaptive Encoder	Implemented
Sample Adaptive Decoder	Implemented
Custom init of Accumulators/Counters	Not Implemented
Hybrid Encoder	Implemented
Hybrid Decoder	Not Implemented
Block Adaptive Encoder/Decoder	Not Implemented

Testing the software proved to be difficult primarily due to the size of the software where a lot of the calculations could prove correct for a small data set while failing on a larger data set. One could calculate the numbers manually using the mathematical equations but that would still not be feasible because corner case errors could potentially not occur. A more thorough testing scheme is to use hyperspecral images. For the testing of the software implementation the testing involves three data images of varied sizes and locations for testing. Two of the images are provided by CCSDS which is located in [14] where multiple test data is provided of varied entropy, and one image was fetched from the Hyperspectral Imager for the Costal Ocean spectrometer located on ISS[20]. For lossless compression of the software the verification method of the software is to compress and decompress itself where the original image can be compared to the decompressed image. This method is not entirely robust as a low entropy image might not trigger some corner cases or create large/small enough numbers for possible failures in the software. Using multiple different images of varied entropy is important for the verification. These verification methods does not work when using lossy compression as the decompressed image will not be equal to the original image. For lossy compression a well known metric for verification of image quality is using the Peak signal-to-noise ratio which provides an objective numerical value to determine quality. First a mean squared error is calculated which gives an average error based on the original image *A* and the compressed image *B*. This is calculated by Equation 6.1.

$$MSE = \frac{1}{xyz}\sum_{i=0}^{x-1}\sum_{j=0}^{y-1}\sum_{k=0}^{z-1}[A(i,j,k) - B(i,j,k)]^2 \tag{6.1}$$

PSNR is calculated by Equation 6.2 where *MAX* is the maximum value possible in the image which is $2^{16} - 1$ for a 16-bit unsigned image. Note that MSE will be 0 when the images are identical which yeilds a division by zero where the PSNR value is infinite. A lower value of PSNR means lower quality.

$$PSNR = 10 * \log(\frac{MAX^2}{MSE}) \tag{6.2}$$

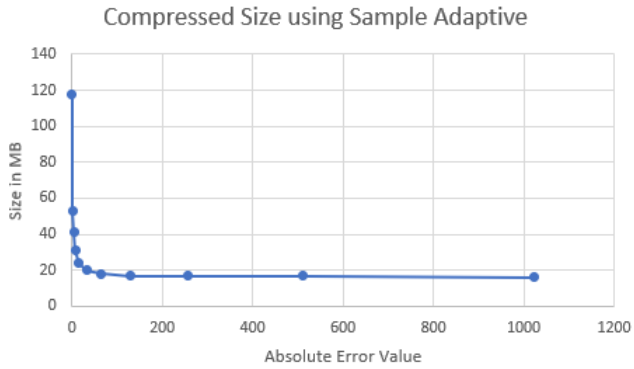
The parameters used for testing all images is as shown in Table 6.5 where the focus of this testing is the varying maximum error under absolute fidelity control. Using the band independent method where all error values are using the same number. The first data set to look on is the HICO image which is a low entropy image with a very flat structure in the image as this is shown on the lossless compressed

image in Figure 6.7. The original image is a 250MB file which is compressed using a varying absolute maximum error x for values 0 to 10 which is incremented by 2^x to provide a fast image degradation. The varying compressed sizes when using the sample adaptive encoder is as shown in Figure 6.2 where a larger absolute error will reduce the compressed size of the image.

Table 6.5.: Parameters used in Testing

Parameter	Value
Prediction mode	REDUCED
Dynamic Range	16
Weight resolution	4
Preceding bands	0
V_{min}	-5
V_{max}	-4
$\log_2 Tinc$	4
Register size, R	64
Local sum	Wide Neighbour unless specified
Θ	0
Error type	ABSOLUTE
Maximum Error $m_z(t)$	Varying
interband offset	0
intraband offset	0
sampleoffset	0
sample damping	0
intraband exponent	0
U_{max}	14
Initial Accumulator value	0
Initial counter value	7
Counter rescaling	9

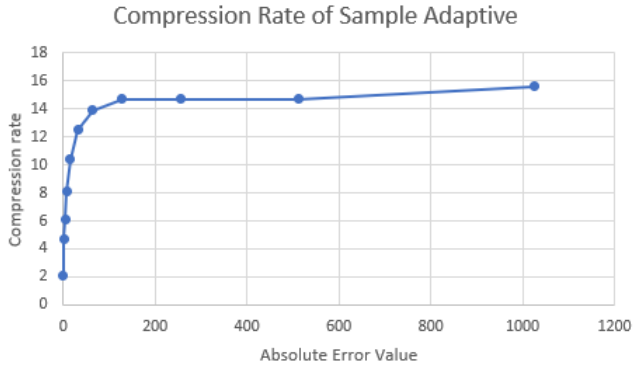
Figure 6.2.: The compressed size of the original image compared to an increasing absolute error for sample adaptive encoder



A different look on the compression of an image is the compression rate **CR** of the compression which provides the total factor of compression. This is calculated by Equation 6.3 which is the division of the original image size by the compressed image size. The compression rate using the sample adaptive encoder with varying absolute error is as shown in Figure 6.3.

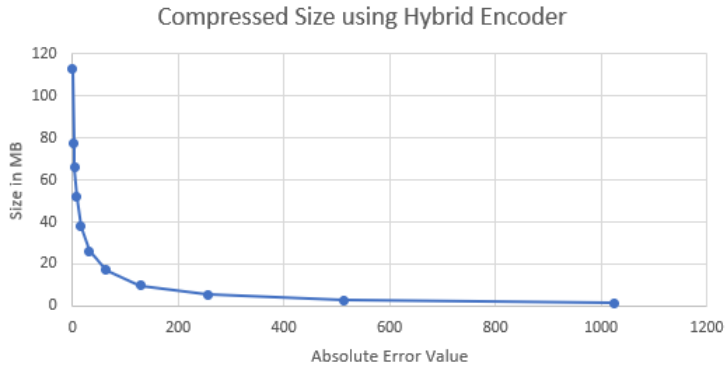
$$CR = \frac{Original\ Size}{Compressed\ Size} \tag{6.3}$$

Figure 6.3.: The compression rate compared to an increasing absolute error for the sample adaptive encoder



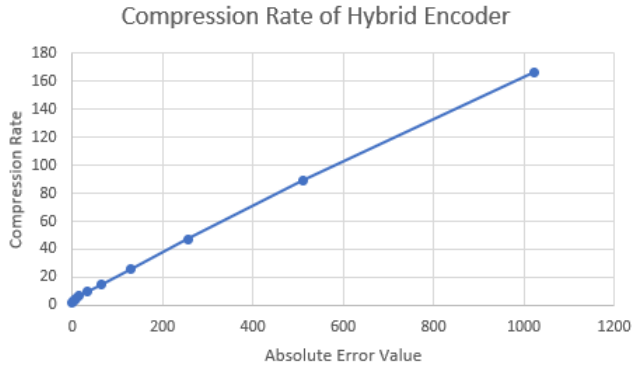
While the hybrid encoder does not have a working decoder it was still feasible to take a look on the statistics of the hybrid encoder on how well it would encode compared to the sample adaptive. Figure 6.4 provides the compressed size of the hybrid encoder which would yield smaller compressed sizes.

Figure 6.4.: The compressed size of the original image compared to an increasing absolute error for the hybrid encoder



In other words compression rate would benefit greatly using the hybrid encoder as this is shown in Figure 6.5 which looked to be going in an linear compression rate. This compression rate however would peak at some value when the quantization would be too large and make each pixel 0 because of division of large values using integers which would make the values 0.

Figure 6.5.: The compression rate compared to an increasing absolute error for the hybrid encoder



Lossy compression does come at the cost of image quality when the absolute error value would increase and cause a degredation of the compressed images. The trend of the PSNR is shown in Figure 6.6 where absolute values 0, 2, 4, 16, 32, 64 and 128 saw no severe image degredation as this is shown respectively in Figure 6.7, Subfigure 6.8(a), Subfigure 6.8(b), Subfigure 6.8(d), Subfigure 6.8(e), Subfigure 6.9(a) and Subfigure 6.9(b). Most edges and distinct objects within the images would slowly become quantized and become similar to the surroundings. For absolute error 256, 512 and 1024 the images would degrade severely as this is shown in Subfigure 6.9(c), Subfigure 6.9(d) and Subfigure 6.9(e) respectively. The possible reason for a comparable large and similar PSNR value for these images is the surrounding water which does not become too much damaged. It should be noted that using different types of local sums provides different image quality where for all these images are using wide local

sums. Looking at a specific spectral band using narrow local sum with absolute error 1024 as this is shown in Subfigure 6.10(a). The image preserves some of the image but no PSNR increase where the narrow local sum is at 41.58 compared to 43 with wide local sum. Looking at spectral band 10 however shows most of the image has been degraded severly as this is shown in Subfigure 6.10(c). Comparably looking at wide local sum with absolute value 1024 as shown in Subfigure 6.10(b) where the land has noticeably been degraded in the image.

Figure 6.6.: Chart of the peak signal to noise ratio compared to an increasing absolute error

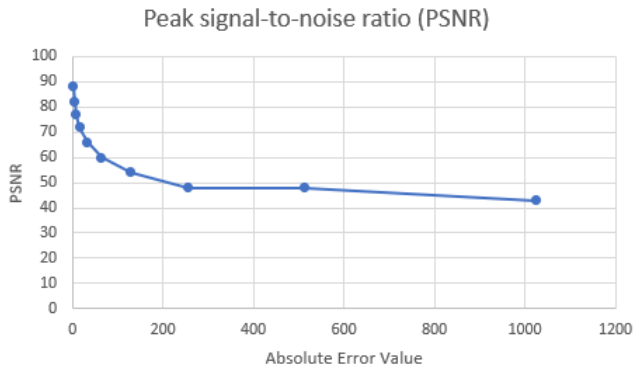
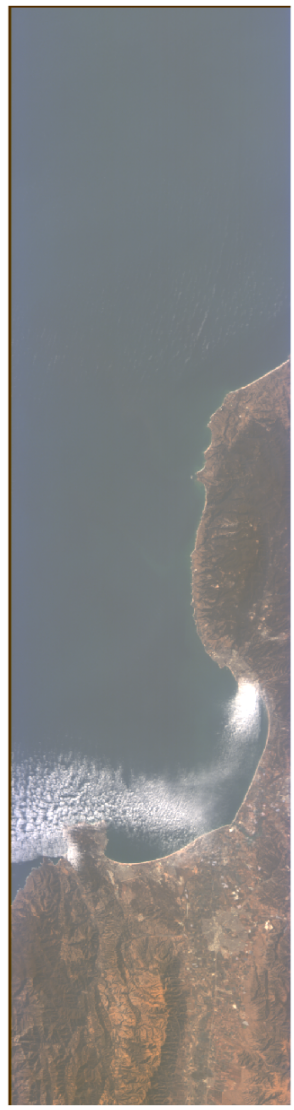
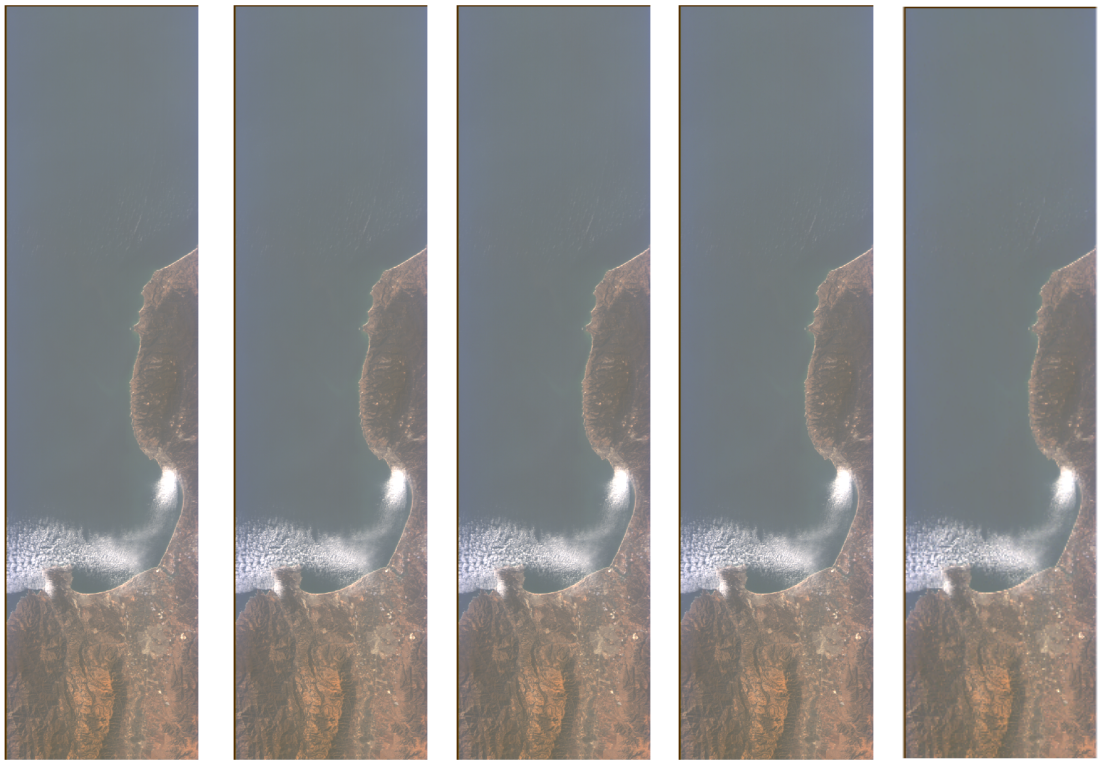


Figure 6.7.: Lossless compressed HICO image with Absolute Error = 0





(a) Absolute error 2 with PSNR 88

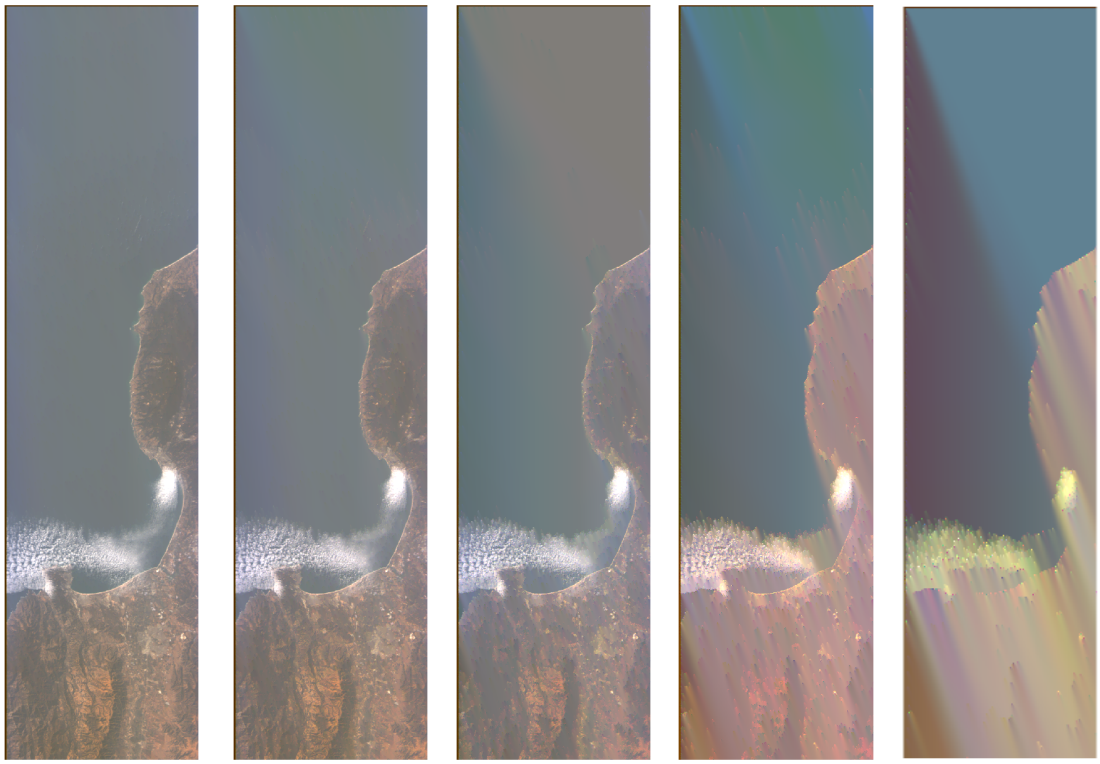
(b) Absolute error 4 with PSNR 82

(c) Absolute error 8 with PSNR 82

(d) Absolute error 16 with PSNR 77

(e) Absolute error 32 with PSNR 72

Figure 6.8.: Lossy compression of HICO images with absolute error



(a) Absolute error 64 with PSNR 66

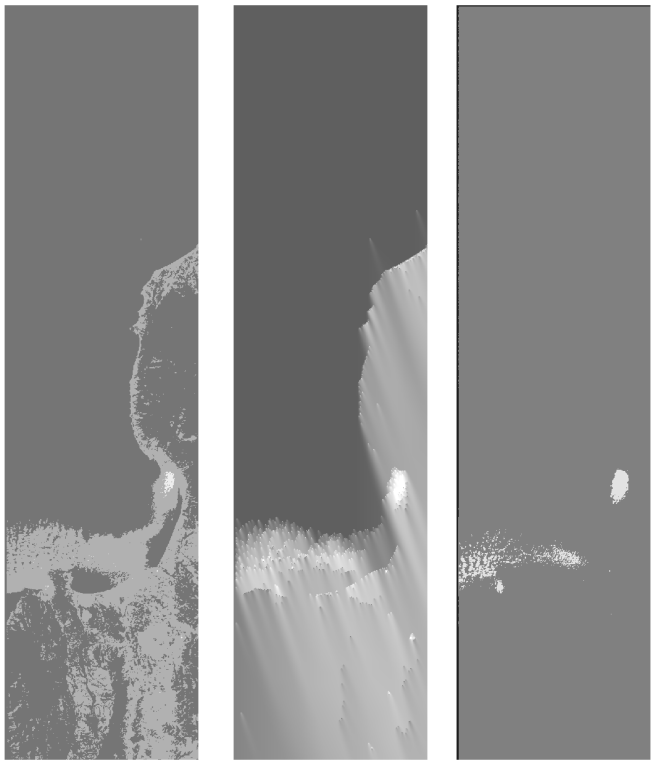
(b) Absolute error 128 with PSNR 60

(c) Absolute error 256 with PSNR 54

(d) Absolute error 512 with PSNR 48

(e) Absolute error 1024 with PSNR 43

Figure 6.9.: Lossy compression of HICO images with absolute error



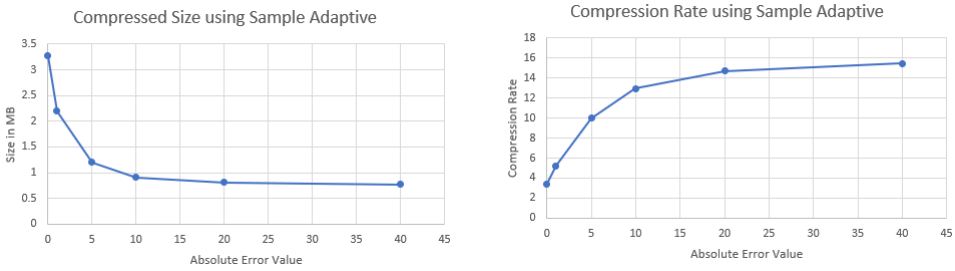
(a) Narrow local sum (b) Wide local sum (c) BAND 10 of narrow local sum

Figure 6.10.: Comparison of different local sums using absolute error 1024

As it can be seen in the HICO images distinct objects in the image could be interpreted which might have been primarily due to the nature of low entropy in the image. Due to this it would be beneficial to use a higher entropy image for compression where a landsat image of a mountain was used. The lossless compressed image of the mountain is as shown in Figure 6.11 where the original image is 12MB in size. The absolute error values used for the compression of this image is much lower compared to the HICO image as severe degradation of the image occurred for small values, and the values used is 0, 1, 5, 10, 20 and 40. The compressed sizes using the sample adaptive is as shown in Subfigure 6.12(a).

Figure 6.11.: Landsat image of a mountain lossless compressed of size X: 1024 Y: 1024 Z: 6

And the compression rate of the images using the sample adaptive encoder is shown in Subfigure 6.12(b).



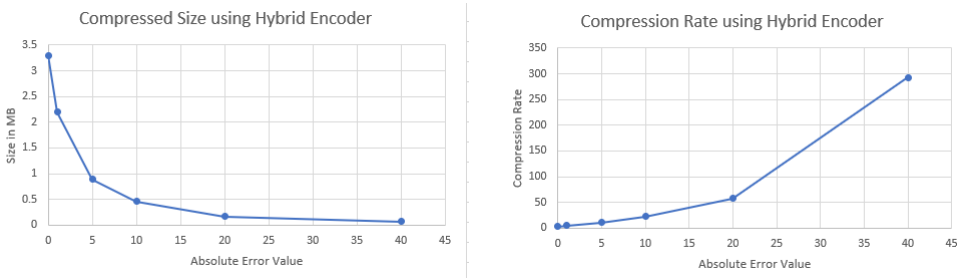
(a) The total compressed size when increasing absolute error

(b) Compression rate when increasing the absolute error

Figure 6.12.: Compressed size and compression rate results from varying absolute error using the sample adaptive encoder

Using the hybrid encoder provides greater compressed sizes for a larger absolute error value as this is shown in Figure 6.4. With the compression rate similar to the the HICO image causing an almost linear compression rate as this is shown in Subfigure 6.13(b). This linearity might occur because of sudden

jumps in incriminating of the absolute error on the absolute error.



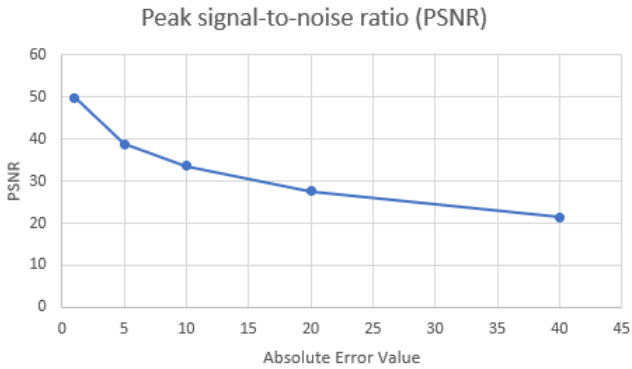
(a) The total compressed size when increasing absolute error

(b) Compression rate when increasing the absolute error

Figure 6.13.: Compressed size and compression rate results from varying absolute error using the hybrid encoder

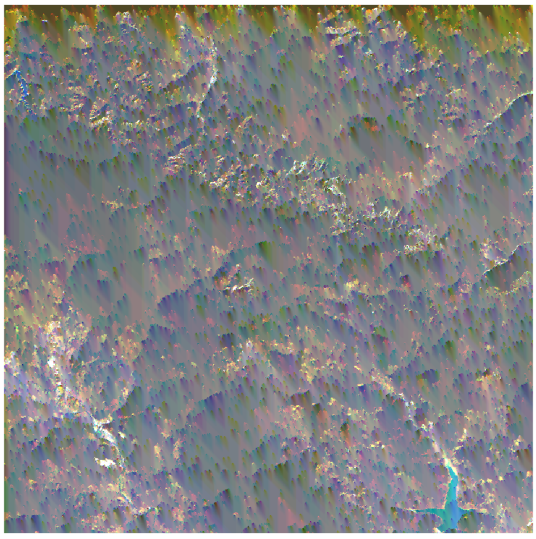
Finally the PSNR of the compression is as shown in Figure 6.14 where the images would degrade for larger absolute error values. Using the absolute error values 0, 10, 20 and 40 for the images which is shown in Figure 6.11, Subfigure 6.15(b), Subfigure 6.15(c) and Subfigure 6.15(d) respectively. Compared to the HICO images these images would degrade higher for very small increments of the absolute error value to the point of severe degradation as this is shown for absolute error 40. Using the narrow local sums also caused some degradation of the image as this is shown for absolute error 1 and 40 in Subfigure 6.16(a) and Subfigure 6.16(b) respectively.

Figure 6.14.: Peak signal-to-noise ratio compared to absolute error value

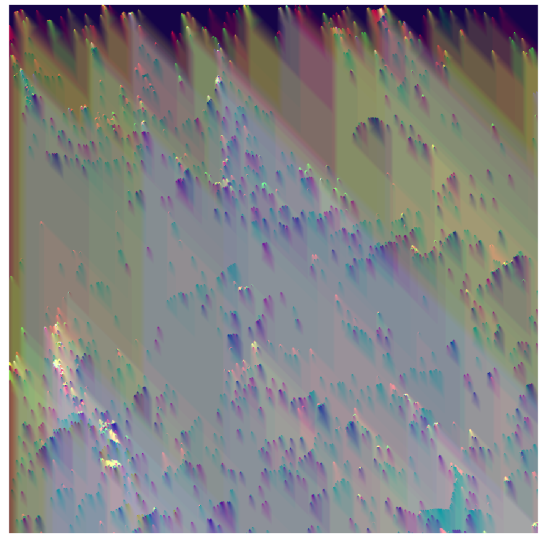


(a) Absolute error 1 with PSNR 49

(b) Absolute error 10 with PSNR 33



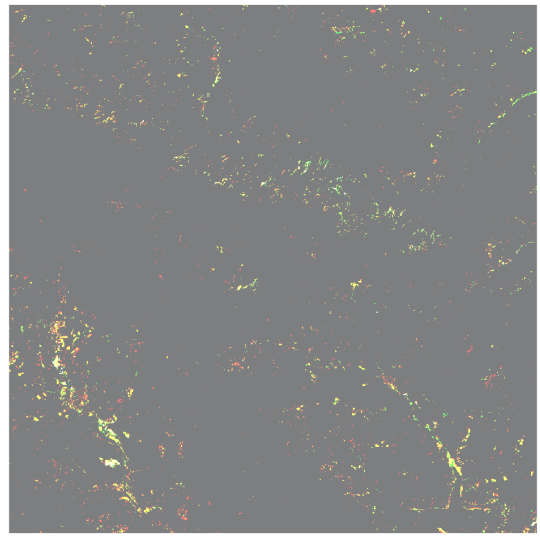
(c) Absolute error 20 with PSNR 27



(d) Absolute error 40 with severe image degradation

Figure 6.15.: Lossy compression of landsat mountain images with absolute error

(a) Absolute error 1 with PSNR 49 using narrow local sum



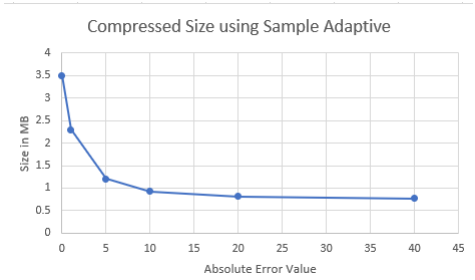
(b) Absolute error 40 with PSNR 23 using narrow local sum

Figure 6.16.: Lossy compression when using NARROW local sum

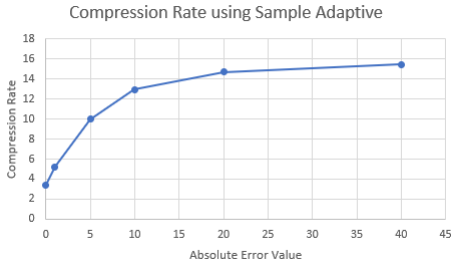
The last image used for testing is a landsat image of farmlands/forests to focus on the preservation of edges in an image when using the compression algorithm. A problem that might occur from using quantization in lossy compression is that colors might become overlapping and two unique parts in an image can become one. This can become clearly in an image with varied farmlands or forests as this is shown in the lossless compressed image Figure 6.17 where it is possible to look at unique parts in the image.

Figure 6.17.: Landsat image lossless compressed of size X: 1024 Y:1024 Z:6

Using similar absolute error values as for the landsat mountain image the compressed sizes is as shown in Subfigure 6.18(a) for absolute errors 0, 1, 5, 10, 20 and 40. The original image is 12MB in size which gives the compression rates as shown in Subfigure 6.18(b).



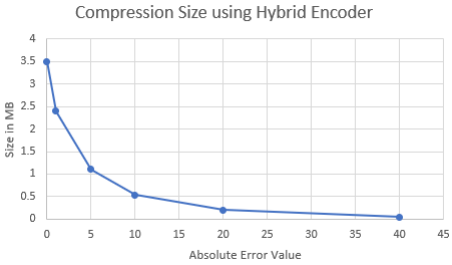
(a) The total compressed size when increasing absolute error



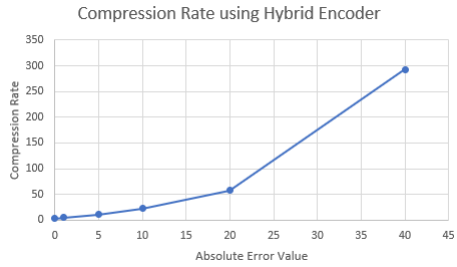
(b) Compression rate when increasing the absolute error

Figure 6.18.: Compressed size and compression rate results from varying absolute error using the sample adaptive encoder

Using the hybrid encoder gave better compressed sizes compared to the sample adaptive as this is shown in Subfigure 6.19(a). The compression rate of the hybrid encoder is as shown in Subfigure 6.19(b).



(a) The total compressed size when increasing absolute error

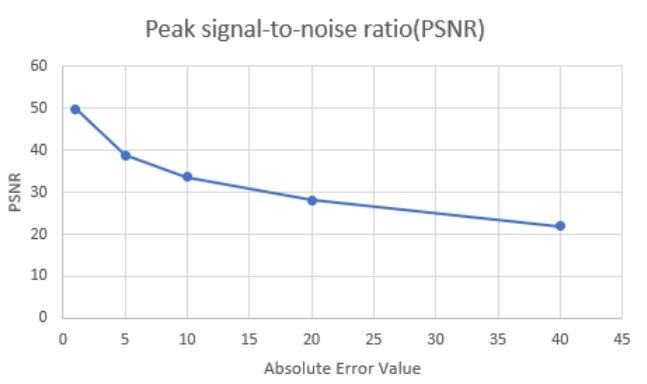


(b) Compression rate when increasing the absolute error

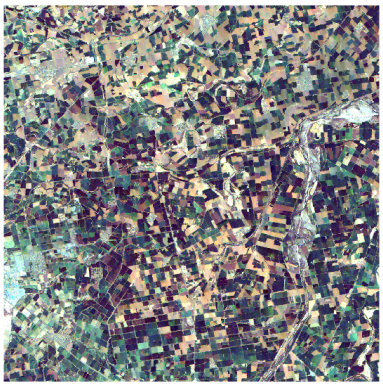
Figure 6.19.: Compressed size and compression rate results from varying absolute error using the hybrid encoder

The Peak signal-to-noise ratio similar to the landsat mountain results would suffer from a higher absolute error value as this is shown in Figure 6.20. The images produced from the varying absolute error is as shown in Subfigure 6.21(a), Subfigure 6.21(b), Subfigure 6.21(c), Subfigure 6.21(d) and Subfigure 6.21(e) for absolute values 0, 10, 20, 30 and 40. It is noticeably the image would suffer with a higher absolute error value to the point where parts in the image would no longer be distinguishable as this is shown for compression with absolute error 40.

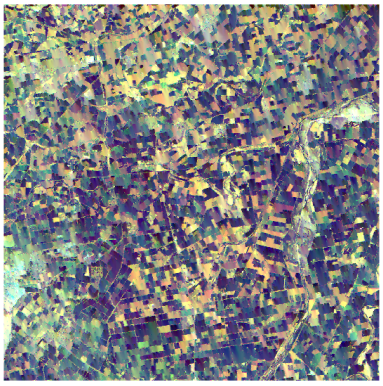
Figure 6.20.: The Peak signal-to-noise ratio of the compressed image compared to absolute error value



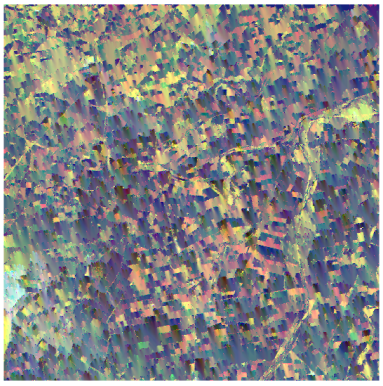
(a) Absolute error 1 with PSNR 50



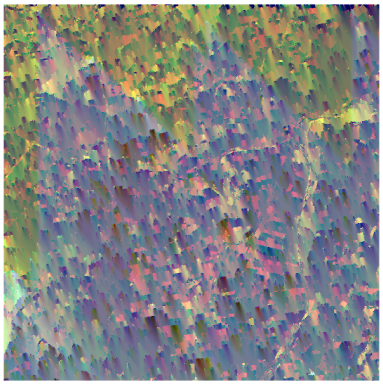
(b) Absolute error 10 with PSNR 33



(c) Absolute error 20 with PSNR 28



(d) Absolute error 30 with PSNR 24

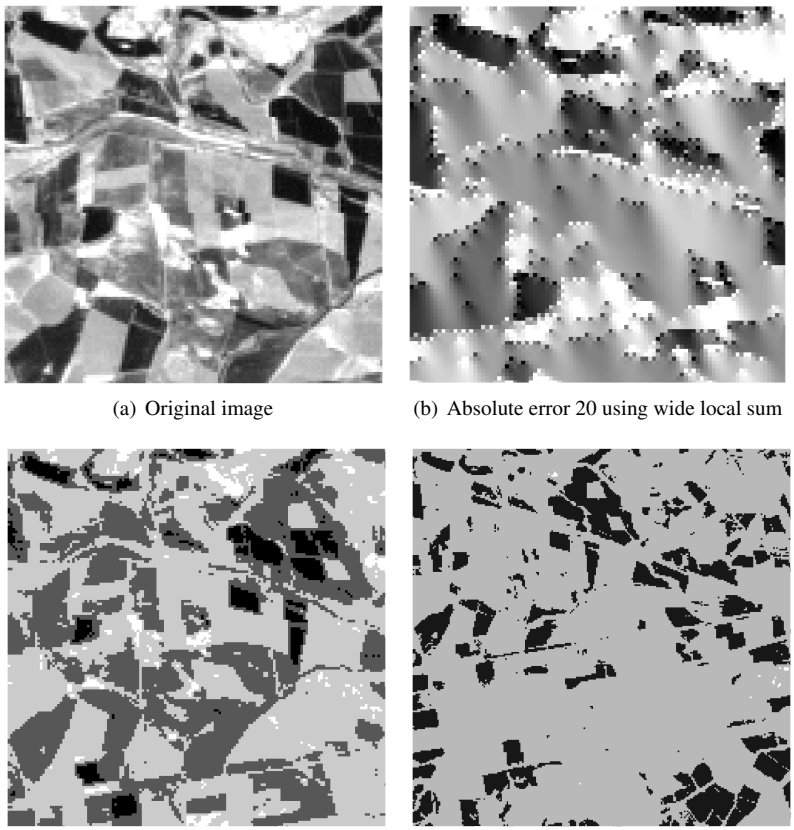


(e) Absolute error 40 with PSNR 21

Figure 6.21.: Lossy compression of landsat images with absolute error

Another issue that might arise is how the shapes inside an image is kept when using the different

compression parameters when the absolute error does not damage the entire image. Using the top left corner as a reference as this is shown in Subfigure 6.22(a) looking at a specific band. Using the absolute error 20 and a wide local sum the results of this is as shown in Subfigure 6.22(b). Noticeably black spots appears in the image and the image appears to be smudged similar to previous images that have been compressed. This issue primarily becomes apparent when using the wide local sums and is not a problem when using the narrow local sums as shown in Subfigure 6.22(c). Focusing on the narrow local sum there is an interesting part of the compression where small objects are quantized. The roads or small edges are gone from the image and parts in the image that has a small difference in color has become one. Using a larger absolute error value of 40 the amount of unique colours in the image becomes a lot smaller as this is shown in Subfigure 6.22(d). Interestingly the dark colours in the original image is preserved from the light colours in the image.



(a) Original image

(b) Absolute error 20 using wide local sum

(c) Absolute error 20 using narrow local sum

(d) Absolute error 40 using narrow local sum

Figure 6.22.: Top left corner of the original Landsat image

6.3. HYPSO mission verification

Veficiation of the compression algorithm CCSDS 123 Issue 1 on the FPGA is a bit more tricky to verify as it would require a decompression algorithm for verifying a working algorithm. Doing it in this way would require storing the original data and the compressed data for comparing after decompression is done. The decompression has to be done on an x86 machine as the decompression is intended to run on a desktop/server and not on an embedded system. It could run on an embedded system but would take unnecessary more time. Note that this verification method allows to use random data or hyper spectral images for testing but varied compression rates will vary. The test verifies the FPGA implementations and if the integration is working, and not the effect of compression rates. The sizes of the images in this test is therefore considered not important. Due to the necessity to verify CCSDS Issue 1 on the FPGA it gave the motivation to further develop and implement a decompression software in C as this was was described in chapter 3.

The first stage of the verification of the hardware was to verify the CubeDMA. There was some doubts whatever it was in fact moving data properly or the interface was working properly. The testing of the DMA engine is very simple and it involves by moving data from one location in memory to another location in memory through the DMA, if those regions are equal then the DMA has done a successful transfer. The code for verifying this is done in C code and is run on the target architecture which in this case is a ZYNQ 7000. The code is as listed in Appendix A, and the result from the code is a simple success or failure if the memory regions is equal or not.

Finally the verification in the FPGA is the CCSDS 123 core. The code presented in Appendix C will create some random data for uncompressed data and add image metadata, as this is specified for header files of the CCSDS 123 standard, to the first 0x13 bytes of compressed data. Adding the metadata is necessary as the FPGA implementation does not do this and would require manual parameters input into the decompression if so. Next procedure is to interface with the CubeDMA and transfer the whole memory region to the CCSDS on the FPGA and wait for completion from the FPGA. Once this is complete the uncompressed data and compressed data will be stored to their individual file for verification. These files needs to be transferred to an x86 machine for using the decompression software that this was described in Chapter 3. As not everyone will be using CCSDS 123 Issue 1 software it was decided to make a bash script to provide a easy verification of decompression. Using the CCSDS 123 Issue 1 software made in 3 it will compare the compressed image with the original image. Comparison in Linux can be done by using the *cmp* command which will compare each byte of a file. Note that the bash script requires the decompression software.

```
#!/bin/bash
echo "Compressed Image: $1"
echo "Original Image: $2"

red=`tput setaf 1`
green=`tput setaf 2`
reset=`tput sgr0`

./main.out -i $2 --DECOMPRESSION
if cmp DecompressedFile.raw $1; then
  echo "${green}Image $1 is equal to DecompressedFile.raw: Compression success"
else
  echo "${red}Images are not equal: Compression failed"
```

```
fi

rm DecompressedFile.raw
```

6.3.1. Verification of Memory Region

An issue that would occasionally occur is a kernel panic when working with the physical memory with a Linux operating system. Linux provides some methods to reserve memory in the RAM for using by a kernel driver or in user programs, but it is not always very documented or shared by others developers. It was then necessary to verify the physical ram was not used by other programs or used by the kernel. The issue might not always arise early after boot-up as memory allocation by Linux can be placed anywhere on the physical ram. A simple method to verify the physical memory is by using the linux device */dev/mem* which provides direct access to the physical memory. Listing 6.1 show how to allocate the memory by using the *mmap* function provided by linux. The result is a pointer to the physical memory region which can then be accessed and verified. Testing the memory reagion can be easily done by iterating through each address of the pointer and writing to the region. If the linux kernel would kernel panic during this process it would mean the memory region is not reserved. This verification provided the necessary help to restructure the linux kernel as described in Section 5.2 by reducing the linux kernel allowed memory, and moving the reserved region to the unused memory. This was decided because it was unceratin if the linux kernel respects reserving memory region within its available memory region. The reservation was primarily done by reserving with device trees as this is documented in [21]. A other option is to use the Contigious Memory Allocator provided by the Linux kernel which provides methods to allocate a contiguous memory dynamically.

```
int fd = open("/dev/mem", O_RDWR|O_SYNC);
uint32_t deviceMem = (uint32_t *) mmap(NULL, CUBESIZE,
                                PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x30000000);
```

Listing 6.1: Allocate Memory

7. Analysis

Finally the implementation of the software will be discussed and the troubles regarding the implementations. An analysis regarding the results of the software will be described. As mentioned the software also had some troubles so the necessary future work for these implementations will also be discussed. Finally the verification of HYPSO related work and results will also be discussed.

7.1. Issue1

The implementation of CCSDS 123 Issue 1 provided a solution for the HYPSO software which could perform a software compression if the FPGA would not be able to perform its job. In most cases this software implementation would not be used as the time to compress in the software is huge compared to the fpga version as this was studied in [1]. This noticeably becomes a problem when the hyperspectral image becomes large enough. The main reason for this large comparison in speed is because how an FPGA can speed up computation by pipelining. Pipelining in a digital design is the usage of registers to allow for parallel computation in a system as this will store the outputs from each computation into registers. This for a system that can pipeline each computation can cause a huge speedup where clocks run at hundreds of MHz. The FPGA implementation made by Johan Fjeldtvedt allows for a pipelined multicore system which allows for higher speedups where the results of his master thesis provided an FPGA core which could compute HSI cubes at 1.2Gbps[2] Even with the problems regarding the implementation of Issue 1 the implementation provides a functioning compression algorithm for the HYPSO mission to perform hyper-spectral compression and decompression. Compared to the FPGA version it was not expected to compete with a dedicated hardware implementation but as mentioned to serve as a backup. Due to this the main focus on the implementation and thesis regarding CCSDS 123 Issue 1 is to create a software to perform this role. Because of the implementation focus of the software there has not been a lot of testing regarding parameters and finding optimal parameters for the HYPSO mission. The approach to this will require to use images from the hyperspectral camera on-board the HYPSO satellite and download images which the HYPSO mission will focus on. The reason for this is to find optimal parameters after the satellite has launched. The hyperspectral images produced by HYPSO can be studied to find optimal compression parameters for optimal compression rates.

The implementation of the software did not go without its problems. As the results are presented there was some issues with the implementation of the predictor as it would create distorted data. Most of these problems occured primarily because of the large algorithm where if anything in the algorithm goes wrong then in most cases everything will be wrong. As the data depends on the previous computed sample then the fault can have occurred previously in the computation. The issues in the software would be from using the wrong integer type, data dependencies or misunderstanding the algorithm. Primarily these issues are from the programmers but the potential culprit could also be the usage of the programming language C. The language is a complex and detailed system which revolves around implicit programming where code needs to be explicit declared to make sure the compiler is following. A lot of the problems can

occur in the background without the programmer knowing what happens as this was the case for the software implementation with integer type casting. While there are detailed rules on how the compiler for C works it provides the necessary work for reading through the standard for C11 which increases the workload [12]. An important factor of this issue is how a team of a few people or magnitude of hundreds of engineers working together handles software design and programming. The possibility in a programming language to implement the same thing in hundreds of varied methods allows for misinterpretation. Solving the casting issue where a good approach could be to define a fixed method for performing a conversion. For example the programming language GO requires types to be defined explicitly and converted as such which allows for no implicit failure from a programmers perspective. A particular issue of CCSDS 123 Issue 2 was the usage of signed and unsigned integers when calculating the inverse of the mapped quantizer index which would cause an overflow issue and make an unsuccessful decompression. The total development time and debugging time could be severely reduced by creating the software in a higher level programming language instead of C. Alternatively is the usage of higher level programming languages where there is less complexity but often at the cost of speed, but as the software could not run well compared to an FPGA, it is negligible [1]. Using alternative programming language compared to C allows for less code which can give the programmer easier overlook of what is happening, this particular issue is primarily regarding the necessary explicit declaration in C where the problems might not occur obviously. Using a different programming language in an embedded environment does provide possible challenges when working with an operating system such as Linux. Some programming languages might not be suitable for performing system operations on an Linux system and could create new possible challenges. A different approach to this is to use C code for the system that requires C and a high level programming language for the non C code of the system. This approach is primarily focusing on how the whole system is designed in a software environment. Communication between the codes can be done by using the linux system with sockets, signals, pipes and files. In particular that HYPSO software is entirely built around C will slowly become bigger and bigger with new people working on it every year. A simple segmentation fault in the software can cause a severe crash to the total software which damages the mission. Creating the control logic, error handling and the system main glue code in its own module and code could potentially remove a total software crash. With this design a crash would not shutdown the whole software but only some sub part of it. This approach is only to prevent a fault from other parts in the software but does not remove the possible system faults from external sources. This redesign of the system will take time and will require a lot of work, but might be a good approach to new missions.

7.2. Issue 2

The new compression standard of CCSDS 123 Issue 2 introduced the possibility to compress hyperspectral images in a near-lossless scheme. Allowing the possibility of higher compression rates compared to the previous issue and allowing to use new hardware in space missions. Notable improvements as mentioned is the support for up to 32-bit sizes which gives it a possibility to use modern hardware. In particular for satellites the costs will increase for providing better and modern hardware while also increasing a demand on transmission bandwidth. This issue provides the growth of compression standards for reducing the necessary hardware to transmit hyperspectral images. As it might not have been obvious the primary benefit for the compression standard is the usage of fidelity control to allow the user of the standard to control the lossy compression. If for example one would like to perform low compression on few selected bands then this is possible to do. As the results has shown using the absolute method for fidelity control to perform the compression proved to increase the compression rates almost linearly for

the first part of absolute error values. This linearity could be perhaps of the sudden jump when increasing the absolute error value. Undoubtedly this came at the cost of image quality from looking at the images and from the PSNR perspective, but as this standard is very new it will still require to research using the compression standard for practical purposes. Another important improvement to the compression standard is the new hybrid encoder. This encoder as mentioned allows for compressing multiple samples into one code-word which reduces the amount of bits necessary to represent the same multiple samples. This improvement becomes apparent in results for compressed sizes of all images when using larger absolute error values, but the sample adaptive encoder did appear to compete for low absolute error values. This type of comparison was also performed in [13] where the same type of results was apparent when competing the hybrid encoder and the sample adaptive encoder.

A particular issue might arise when studying the results of the compressed images which for large absolute error values seemed to damage the image severely. The implementation as similar to the issue 1 implementation did not come without its problems. The C implementation of issue 1 was a copy from the implementation of issue 2 where bugs that appeared in one version did become apparent in the other. This grows the suspicion of a possible fault with lossy compression, but as there is no comparison to other software implementations there is the possibility of software faults. Now that does not remove the fact that it performs well when using lossy comprssion and there might be no issue at all. The implementation would still require further verification of the implementation from a third-party so that the results can be confirmed. The verification performed on the software has had the possibility to use test vectors from [14] but these has primarily been for supporting 32-bit and the verification of the hybrid encoder. It still remains to verify the usage fidelity control for compression.

Assuming the results are valid and there is no problem with the compression algorithm. The resulting images from the varied tests show that any user of the software or algorithm is required to find an optimal fidelity control for their usage. This will also grow the requirement for hardware to support dynamic parameters as the compression algorithm can work for some images and terribly for others. Even so the compression algorithm proves to be an effective algorithm that can compress images that are hundreds of MB to a lot less depending on the chosen fidelity control.

Using the compression standard for the FPGA does include some new design challenges for the implementation. A problem that will damage the possible compression speed on an FPGA is the introduction of sample representatives. The reason for this is that the calculation of predicted sample at t will require the sample representative from $t-1$. This is used for calcuating the localsum and followed by calculating the local difference. This problem arise for performing prediction under a BSQ/BIL encoding order, but almost similar problem will arise in a BIP encoding order. For the BIP encoding order the dependency is apparent when performing prediction mode with $P > 0$. The algorithm will require to use the preceding local difference $z - 1$ when performing prediction at band z, and the local difference again requires the local sum and the sample representation calculated in $z - 1$. This kind of problem will arise when attempting to pipeline or attempting an parallel implementation in an FPGA, or creating a multi-threaded software implementation. This issue will still require further research when attempting further speedups of the implementation.

As the status of the software is presented there still remains some work to be completed for fully implementing the algorithm. The most important remaining feature is the hybrid decoder which requires

some more research and work for completing it. Creating a decoder will allow the software to fully function with the hybrid encoder to gain the benefits of higher compression rates. The implementation of the software provides a basic implementation of the CCSDS 123 Issue 2 standard and all other features not implemented was deemed not necessary as mentioned. These features is therefore considered in the remaining work if it is required for future usage. The remaining work required is the further verification of fidelity control to make sure the near-lossless compression is true. A potential verification is to wait for other researchers to develop their software for comparison.

7.3. HYPSO

As mentioned the need for verifying the HYPSO hardware and communication with the hardware was a requirement that grew throughout this project. As it can be noticeable is the out of context this type verification is described in this thesis. This was still required to verify and get a working system for further development in this project. A description of the additions and changes was therefore required to be written about in this thesis, as well as the communication with CubeDMA. The communication with the CubeDMA implemented is using the basic features of the CubeDMA for transferring data. The DMA can still support different functionality but these functionalities is not used for this context. However if it is a required functionality then it can be implemented from the baseline communication from this thesis. From the resulting verification of the CubeDMA proved the results to be successful and a functional system.

Once the CubeDMA was properly tested and verified it was possible to begin verifying CCSDS 123 FPGA core. As mentioned it was required to implement a header to the FPGA compressed stream and for decompression to verify a functiong system. This was verified and properly working during the verification, but did not go without its troubles. During the verification a couple of issues was discovered. The compression core could not perform compression on images of varied sizes without requiring to synthesise a new core for the new parameters. This became a problem as capturing images of varied sizes can be a possibility for HYPSO. A solution is to capture images in multiples of a fixed size. E.g using X:512 Y:200 Z:128 where Y will be the focus of multiples. When the satellite moves along the orbit then the image can capture Y=200 images followed by 200 more and so on. If one wants to capture a size of Y=2000 images then it would require 10 image captures. Even with this problem the FPGA core provides a functioning compression core of CCSDS 123 Issue 1 for the HYPSO mission.

Bibliography

[1] Christoffer Boothby. A software implementation of the ccsds 123 issue 2 compression standard. a low-complexity lossless and near-lossless compression standard of multispectral and hyperspectral images, project thesis. 2019.

[2] Johan Fjeldtvedt. Efficient streaming and compression of hyperspectral images, master thesis. 2018.

[3] National Aeronautics and Space Administration. Lossless multispectral and hyperspectral image compression recommended standard. Blue/Silver book CCSDS 123.0-B-1, Consultative Committee for Space Data Systems, 2012. URL `https://public.ccsds.org/Pubs/123x0b1ec1s.pdf`.

[4] National Aeronautics and Space Administration. Low-complexity lossless and near-lossless multispectral and hyperspectral image compression. Blue book CCSDS 123.0-B-2, Consultative Committee for Space Data Systems, 2019. URL `https://public.ccsds.org/Pubs/123x0b2c1.pdf`.

[5] *Zynq-7000 SoC Techincal Refrence Manual, UG585*. Xilinx, 2018. URL `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`.

[6] Harry Jones. The recent large reduction in space launch cost. 48th International Conference on Environmental Systems, 2018.

[7] Scott E Palo. High rate communications systems for cubesats. In *2015 IEEE MTT-S International Microwave Symposium*, pages 1–4. IEEE, 2015.

[8] The consultative committee for space data systems (ccsds). URL `https://public.ccsds.org/default.aspx`.

[9] Aaron B. Kiely, Matthew Klimesh, Ian Blanes, Jonathan Ligo, Enrico Magli, Nazeeh Aranki, Michael Burl, Roberto Camarero, Michael Cheng, Sam Dolinar, and et al. *THE NEW CCSDS STANDARD FOR LOW-COMPLEXITY LOSSLESS AND NEAR-LOSSLESS MULTISPECTRAL AND HYPERSPECTRALIMAGE COMPRESSION*.

[10] National Aeronautics and Space Administration. Lossless multispectral and hyperspectral image compression. Green book CCSDS 120.0-G-3, Consultative Committee for Space Data Systems, 2015. URL `https://public.ccsds.org/Pubs/123x0b2c1.pdf`.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

[12] *SEI CERT C Coding Standard*. Carnegie Mellon University, 2016.

[13] Ian Blanes, Aaron Kiely, Miguel Hernández-Cabronero, and Joan Serra-Sagristà. Performance impact of parameter tuning on the ccsds-123.0-b-2 low-complexity lossless and near-lossless multispectral and hyperspectral image compression standard. *Remote Sensing*, 11(11):1390, 2019.

[14] Ccsds 123 sharepoint info. URL "https://cwe.ccsds.org/sls/docs/SLS-DC/123.0-B-Info".

[15] Milica Orlandić, Johan Fjeldtvedt, and Tor Arne Johansen. A parallel fpga implementation of the ccsds-123 compression algorithm. *Remote Sensing*, 11(6):673, 2019.

[16] Johan Fjeldtvedt, Milica Orlandić, and Tor Arne Johansen. An efficient real-time fpga implementation of the ccsds-123 compression standard for hyperspectral images. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 11(10):3841–3852, 2018.

[17] Johan Fjeldtvedt and Milica Orlandić. Cubedma–optimizing three-dimensional dma transfers for hyperspectral imaging applications. *Microprocessors and Microsystems*, 65:23–36, 2019.

[18] Andreas Varntresk. Assembly and testing of baselineprocessing chain, master thesis. 2019.

[19] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005. ISBN 0596005903.

[20] "hyperspectral imager for the coastal ocean". URL "http://hico.coas.oregonstate.edu/".

[21] Linux documentation of reserving memory. URL https://github.com/torvalds/linux/blob/master/Documentation/devicetree/bindings/reserved-memory/reserved-memory.txt.

A. Verification of CubeDMA

The code for verification of CubeDMA is attached as a delivery to this thesis. Alternatively with access it can be found in the SmallSat lab github page `https://github.com/NTNU-SmallSat-Lab`.

B. Linux Kernel Driver

The developed linux kernel driver is attached to this thesis. Alternatively with access it can be found in the SmallSat lab github page `https://github.com/NTNU-SmallSat-Lab`.

C. Verification of CCSDS FPGA

The implemented verification software of the CCSDS 123 on the FPGA is attached to this thesis.

D. Software Implementation

The software implementation is included as a zipped file with this thesis. Including Issue 1 and Issue 2 implementation of CCSDS 123.

E. Code Tables and Flush Tables

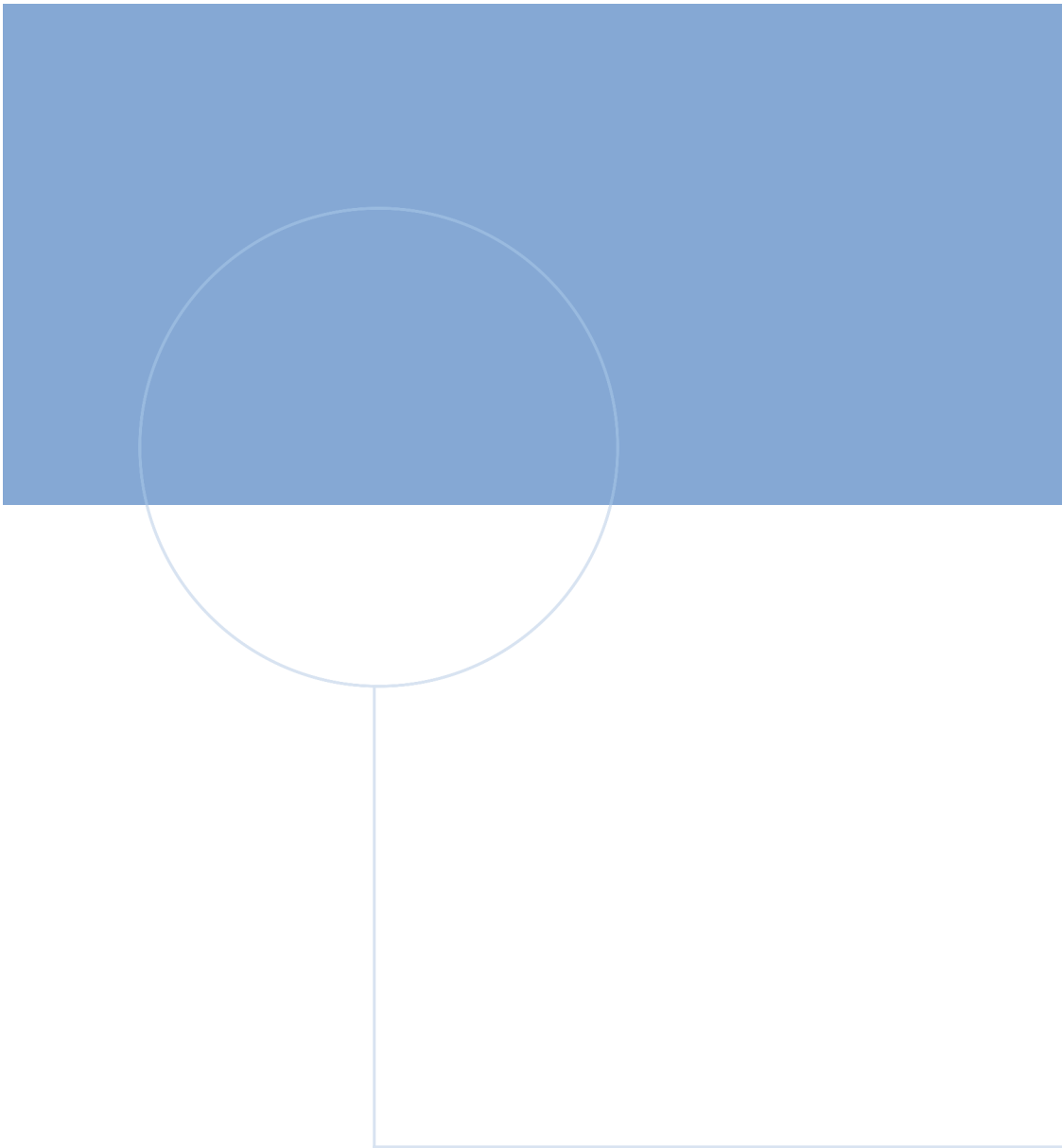
This appendix provides an example code table and flush table for CCSDS 123 Issue 2. Note that this is extracted from the PDF of the code tables as provided in [4], and it is liable for changes in the future.

Table B-1: Code Table for Low-Entropy Code 0

Input Codeword	Output Codeword	Input Codeword	Output Codeword	Input Codeword	Output Codeword
00	5'h19	09	8'h3B	5	4'h6
010	8'h57	0A	8'hBB	6	4'hE
011	8'hD7	0B	9'h00F	70	7'h0B
012	8'h37	0C	9'h10F	71	7'h4B
013	9'h0AF	0X	8'h7B	72	7'h2B
014	9'h1AF	1	3'h0	73	8'h47
015	9'h06F	2	3'h4	74	8'hC7
016	9'h16F	3	3'h2	75	8'h27
017	10'h03F	40	6'h1D	76	8'hA7
018	10'h23F	41	6'h3D	77	9'h0CF
019	11'h17F	42	6'h03	78	9'h1CF
01A	11'h57F	43	6'h23	79	10'h15F
01B	12'h1FF	440	9'h09F	7A	10'h35F
01C	12'h9FF	441	9'h19F	7B	11'h07F
01X	11'h37F	442	9'h05F	7C	11'h47F
020	8'hB7	443	10'h0BF	7X	10'h0DF
021	8'h77	444	10'h2BF	80	7'h6B
022	8'hF7	445	10'h1BF	81	7'h1B
023	9'h0EF	446	10'h3BF	82	7'h5B
024	9'h1EF	447	11'h2FF	83	8'h67
025	9'h01F	448	11'h6FF	84	8'hE7
026	9'h11F	449	12'h3FF	85	8'h17
027	10'h13F	44A	12'hBFF	86	8'h97
028	10'h33F	44B	13'h0FFF	87	9'h02F
029	11'h77F	44C	13'h1FFF	88	9'h12F
02A	11'h0FF	44X	12'h7FF	89	10'h2DF
02B	12'h5FF	45	7'h33	8A	10'h1DF
02C	12'hDFF	46	7'h73	8B	11'h27F
02X	11'h4FF	47	8'hFB	8C	11'h67F
03	6'h15	48	8'h07	8X	10'h3DF
04	6'h35	49	9'h08F	9	5'h01
05	6'h0D	4A	9'h18F	A	5'h11
06	6'h2D	4B	9'h04F	B	6'h05
07	7'h13	4C	9'h14F	C	6'h25
08	7'h53	4X	8'h87	X	5'h09

Table B-2: Flush Table for Low-Entropy Code 0

Active Prefix	Flush Word	Active Prefix	Flush Word	Active Prefix	Flush Word
(null)	1'h0	02	6'h1F	7	5'h07
0	2'h1	4	3'h3	8	5'h17
01	5'h0F	44	6'h3F		



NTNU

Norwegian University of
Science and Technology