universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Science Data Processing for SMILE/SXI“

verfasst von / submitted by

Dipl. Ing. Berndt Dorninger, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2022 / Vienna, 2022

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066861
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Astronomie
Betreut von / Supervisor:	Univ. Prof. Mag. Dr. Franz Kerschbaum
Mitbetreut von / Co-Supervisor:	Mag. Dr. Roland Ottensamer

Acknowledgements

I was born in 1957, the year of the first space flight. Astronomy was always fascinating for me. Nevertheless I also was interested in the growing field of electronics and computer science where I have worked for many years. Combining these two fields of research enriches my life now.

Many thanks to my family and all my friends who encourage me to make my way towards the field of astronomy in a professional manner in my old days.

I want to express my sincere gratitude to my supervisor Univ.-Prof. Mag. Dr. Franz Kerschbaum who offered me the unique opportunity for this master's thesis and to become a member of his outstanding team of the SMILE-mission.

Furthermore, I want to thank my co-supervisor Mag. Dr. Roland Ottensamer for all the helpful discussions, advices and inspiring hints during the project.

Many thanks also to Verena Baumgartner for correcting typos and improve the English phrasing in some places.

Last but not least, I would like to thank all my colleagues at the Institute of Astrophysics, who were always there to help and advise me.

Abstract

The aim of the master's thesis is to adapt and expand the science data processing already developed for CHEOPS (CHaracterising ExOPlanet Satellite) to the SMILE mission.

SMILE, the Solar Wind Magnetosphere Ionosphere Link Explorer, is a joint mission of the European Space Agency and the Chinese Academy of Sciences involving contributors and companies from about 20 countries. This includes Austria with the IfA in Vienna and the IWF in Graz.
It is planned to launch a satellite into Earth orbit that will study the interaction between the solar wind, the Earth's magnetic field and the ionosphere. In order to better understand these processes, it is necessary to simultaneously study the processes in the day-side magnetopause, i.e. the area where the solar wind and the magnetosphere meet, the polar caps, where the sun particles interact directly with the earth's atmosphere, and the so-called auroral oval, that region where Auroras are common to watch.

For this purpose, SMILE has four different devices on board:
A magnetometer to measure the magnetic field, an ion detector for light ions, a UV camera to observe the polar region and a detector for soft X-rays that are produced in the so-called magnetosheath.
This detector, also known as a Soft X-ray Imager (SXI), detects the soft X-ray radiation that occurs during the interaction between the solar wind and the magnetosphere. It has two 4510 x 4510 pixel Charged Coupled Devices (CCD's) that resolve the X-rays with 16-bit depth (18 µm per pixel). The detector is normally operated in 6x6 binning mode (108 µm per pixel). There is also a 24x24 binning mode to detect high-energy UV light.
Science data processing is essential to make the resulting amounts of raw data scientifically accessible. A lot of data is already processed on board the satellite and the data sent to the ground station is optimally compressed beforehand with the least possible loss of information, since the transmission capacities are low.

This master's thesis should first give an insight into the astrophysical background of the SMILE mission as well as the objectives, scientific requirements and technical framework conditions.
The Soft X-ray Imager with its hardware and software environment is presented in detail and it is explained which further requirements are required for science data processing.

Based on this, the necessary adaptation work in the compress/decompress programs already in use at CHEOPS must be carried out for the application of science data processing at SMILE. The data simulator DaSi has to be adapted accordingly, a test strategy

has to be worked out and the necessary tests have to be carried out.
An additional extension of science data processing should also enable data compression for event detection data. Event detection is carried out directly in the control electronics of the Soft X-ray Imager's Charged Coupled Devices. The events caused by the X-rays are detected in the images and a data set of 5x5 pixels is generated for each event. This is the data that is mainly used and compression makes sense, even if the image data has already been reduced by event detection. For this purpose, the results of the previously known compression methods are used and a new method specially tailored to events is developed.
A compression method based on Golomb coding is currently being developed for the PLATO mission (PLAnetary Transits and Oscillation of stars). Making a comparison with the compression algorithms already been implemented at CHEOPS is interesting. Derived from this, the Golomb coding is implemented as an extension of the compression method for SMILE.

The possible uses of the new operating system UVIE FlightOS developed at the institute are to be evaluated and a corresponding implementation has to be made in the Compress programs.

Finally, it must also be ensured that the science data processing does not just run only in standalone mode, but can be built directly into the application software of the satellite and the application software of the ground station. This requires the development of appropriate interface modules.

During the work on this master's thesis, more than 3000 lines of code were created for programs and procedures, mainly in the programming languages C and C++. In addition, some auxiliary programs were written in Python.

Kurzfassung

Das Ziel der Masterarbeit ist es, das Science Data Processing von CHEOPS (CHaracterising ExOPlanet Satellite) für die SMILE-Mission anzupassen und zu erweitern.

SMILE, der Solarwind Magnetosphere Ionosphere Link Explorer, ist eine gemeinsame Mission der European Space Agency mit der Chinesischen Akademie der Wissenschaften wobei Mitwirkende und Firmen aus etwa 20 Staaten beteiligt sind. Darunter auch Österreich mit dem IfA in Wien und dem IWF in Graz.
Es ist geplant, einen Satelliten in eine Erdumlaufbahn zu bringen, der das Zusammenspiel zwischen Sonnenwind, Erdmagnetfeld und Ionosphäre untersuchen soll. Um diese Vorgänge besser zu verstehen, ist es notwendig, gleichzeitig die Abläufe in der tagseitigen Magnetopause, also dem Bereich, wo Solarwind und Magnetosphäre aufeinander treffen, den Polarkappen, wo die Sonnenpartikel direkt mit der Erdatmosphäre wechselwirken, und dem sogenannten auroral oval, jener Region wo Polarlichter häufig vorkommen, zu beobachten.

Zu diesem Zweck hat SMILE vier verschiedene Geräte an Bord:
Ein Magnetometer zur Messung des Magnetfeldes, einen Ionendetektor für leichte Ionen, eine UV-Kamera, die die Polregion beobachten soll und einen Detektor für weiche Röntgenstrahlung, die in der sogenannten Magnetosheath entsteht.
Dieser Detektor, auch Soft X-ray Imager (SXI) genannt, detektiert die weiche Röntgenstrahlung, die bei der Interaktion Sonnenwind / Magnetosphäre entsteht. Er hat zwei 4510 x 4510 PixeI große Charged Coupled Devices (CCD's), die mit 16-bit Tiefe die Röntgenstrahlung auflösen (18 µm pro pixel). Der Detektor wird normalerweise im 6x6 binning mode betrieben (108 µm pro pixel). Zusätzlich gibt es noch einen 24x24 binning modus um hochenergetisches UV-Licht zu detektieren.
Um die dabei entstehenden Rohdatenmengen wissenschaftlich zugänglich zu machen, ist das Science Data Processing unumgänglich. Dabei werden viele Daten bereits an Bord des Satelliten verarbeitet und außerdem die an die Bodenstation gesendeten Daten zuvor bei möglichst geringem Informationsverlust optimal komprimiert, da die Übertragungskapazitäten gering sind.

Diese Masterarbeit soll zunächst einen Einblick in die astrophysikalischen Hintergründe der SMILE-Mission geben sowie die Zielsetzungen, wissenschaftlichen Anforderungen und technischen Rahmenbedingungen geben.
Der Soft X-ray Imager mit seiner Hardware- und Software-Umgebung wird dabei im Detail vorgestellt und welche weiterführenden Anforderungen sich daraus für das Science Data Processing ergeben.

Darauf basierend sind für die Anwendung des Science Data Processing bei SMILE die notwendigen Anpassungsarbeiten in den, bereits bei CHEOPS im Einsatz befindlichen, Compress/Decompress-Programmen vorzunehmen. Der Datensimulator DaSi ist entsprechend zu adaptieren, eine Teststrategie auszuarbeiten und die notwendigen Tests durchzuführen.
Eine zusätzliche Erweiterung des Science Data Processing soll die Datenkompression auch für die Event Detection Daten ermöglichen. Event Detection wird direkt in der Steuerelektronik der Charged Coupled Devices des Soft X-ray Imager durchgeführt. Dabei werden die Events, die durch die Röntgenstrahlung entstehen, in den Bildern detektiert und pro Event ein Datensatz von 5x5 Pixel generiert. Diese sind die hauptsächlich genutzten Daten und eine Kompression ist durchaus sinnvoll, auch wenn die Imagedaten durch die Event Detection bereits reduziert wurden. Dazu werden die Ergebnisse der bisher bekannten Kompressionsverfahren herangezogen und ein neues, speziell auf Events zugeschnittenes, Verfahren entwickelt.
Ein auf Golomb-coding basierendes Kompressions-Verfahren wird derzeit für die PLATO-Mission (PLAnetary Transits and Oscillation of stars) entwickelt. Interessant ist ein Vergleich mit den Verfahren, die bereits bei CHEOPS implementiert wurden. Daraus abgeleitet, soll das Golomb-coding als Erweiterung der Kompressionsverfahren für SMILE implementiert werden.

Die Einsatzmöglichkeiten des am Institut entwickelten neuen Betriebssystems UVIE FlightOS sind zu evaluieren und eine entsprechende Implementierung in den Compress-Programmen vorzunehmen.

Schließlich ist auch dafür zu sorgen, dass das Science Data Processing nicht nur im standalone-Modus läuft, sondern direkt in die Application Software des Satelliten und in die Anwendungs-Software der Bodenstation eingebaut werden kann. Dafür ist die Entwicklung entsprechender Interface-Module notwendig.

Im Zuge dieser Masterarbeit entstanden mehr als 3000 Programmzeilen an Programmen und Prozeduren hauptsächlich in den Programmiersprachen C und C++. In Python wurden zusätzlich einige Hilfsprogramme geschrieben.

Contents

List of Tables

List of Figures

Listings

1. Introduction

The Sun appears as a source of energy in various forms for us, living on the planet Earth. But some interactions between the emissions of our sun with the environment of our planet are sometimes disturbing our modern life. One of these phenomena is the interaction between the solar wind and the magnetosphere and ionosphere of the Earth. It causes such beautiful events like the aurora borealis but has also a negative influence on modern telecommunication systems and electric power distribution networks and can even lead to intensive corrosion of pipelines in northern regions of America, Asia and Europe.

For a better understanding of these interactions a new space mission called SMILE was created (Branduardi-Raymont et al. 2018).
SMILE, the Solar Wind Magnetosphere Ionosphere Link Explorer, is a joint mission between European Space Agency (ESA) and the Chinese Academy of Sciences (CAS) involving scientists and companies from about 20 countries. This includes Austria with the Institute for Astrophysics (IfA) of the University of Vienna and the Institut für Weltraumforschung (IWF) in Graz.
A satellite will be put into orbit to investigate the interaction between the solar wind, the earth's magnetic field and the ionosphere. Phenomena like the aurora borealis are well known. However, in addition to these visually beautiful phenomena, there are also some less favourable phenomena such as radio interference, influencing the power grid and even increasing corrosion in pipelines due to the induced flow of electricity, especially in regions close to the polar. Therefore, e.g., Canada is also very interested in the mission.

In detail, the aim of the mission is to better understand the basic properties and processes of the daytime interaction between solar wind and magnetosphere or ionosphere and how the formation of substorms takes place.
A magnetospheric partial storm (substorm) is a short-term disturbance in the magnetosphere of a planet, which causes the release and redirection of energy from the "tail", the "lobes" of the magnetosphere into the near-polar ionosphere. Visually it appears as a sudden lighting up and increased movement of aurora arcs. Such partial storms occur over a period of a few hours, are mainly observed in the polar region, introduce few particles into the radiation belt and are relatively frequent, being often only a few hours apart. They arise when a plasmoid becomes detached in the magnetic field tail, which means there is a reconnection of the field lines.
This can also happen in times when no magnetic storm from the sun hits the earth caused by the interaction of the earth's magnetic field with the IMF, the interplanetary magnetic field (Gallagher, D. 2002).

1. Introduction

The solar wind flows around the earth at supersonic speed, its flow speed is greater than the speed with which disturbances of the density or the pressure in the solar wind move (speed of sound). It is braked to subsonic speed at the bow shock wave. The area between the bow shock wave and the magnetopause is called magnetosheath. Part of the solar wind is also reflected on the bow shock wave, so that a pre-shock develops. The magnetic field lines are closed on the day side. On the night side they are extended far into space, here the "lobes" are formed. The so-called "cusps" form the transition. Here the particles of the solar wind can penetrate into the earth's atmosphere.

In order to better understand these processes, it is necessary to simultaneously examine the processes in the daytime magnetopause, i.e. the area where the solar wind and the magnetosphere meet, the polar caps, where the sun particles interact directly with the earth's atmosphere, and the so-called auroral oval, the region where the aurora borealis is commonly observed.
For this purpose, SMILE has 4 different devices on board, a magnetometer to measure the magnetic field, an ion detector for light ions, a UV camera that is supposed to observe the polar region and a detector for soft X-ray radiation that is created in the so-called magnetosheath (Dennerl, K. et al. 2012; Sibeck, D. 2018; Fei H. et al. 2014; Kuntz 2018; Sun et al. 2019; Sibeck et al. 2018).
For one of these instruments, the SXI (Soft X-ray Imager), several contributions are developed in Austria. The Institut für Weltraumforschung (IWF) equip the mission with the onboard computer-hardware and the Institute for Astrophysics provides the Instrument Application software.

Roughly speaking, Science Data Processing is about filtering out scientifically relevant data from the raw dataset and then packing it into packets that can be transmitted to the Earth via the thin communication channel (Ottensamer, R. 2009), (Ottensamer R. 2018). Basically there are 2 ways:

- With intelligent preprocessing of the data only those that are of interest need to be transmitted.
- Compress the user data for easier handling for storage and transmission.

You can of course also combine these two methods and get very compact datasets.
The resources in the satellite are limited in terms of computing and storage capacity. In addition, it is always necessary to transfer entire raw data packets, e.g. for calibration or to find new scientific knowledge that the onboard software does not cover with its algorithms.

A little background knowledge about SDP would be helpful:
One should be familiar with the numerical methods of astronomy and parts of information theory, i.e. data statistics, various distribution functions, correlation, covariance, entropy in the sense of information theory. Encoding mechanisms are of course important (a popular example would be the Morse code). It is also good to know something about signal quality and image processing (Meffert & Hochmuth 2018).

Entropy is a way of evaluating information on one hand and noise on the other hand in a signal. This noise has to be characterized so that the useful signal can be filtered out (Shannon 1948).

When implementing solutions for Science Data Processing you should be familiar with development processes (ESA has its own specifications), of course with programming (especially Python, C and C++ as programming languages) and some hardware background such as detectors, readout electronics and their interfaces.

How does the processing in the SMILE-SXI work?
The CCD's are controlled by the Front-End Electronics (FEE). This is where the images are read out, an initial image processing and transfer to the Data Processin Unit (DPU) takes place (Reimers Ch. 2019; Smit S.J.A. 2019; Mecina, M. et al. 2020).
The Event Detection Unit (EDU) is also located in the FEE. It can already recognize X-ray events (single pixel threshold or local maximum) and deliver imagettes (image sections 5x5 pixels) that only include these events. There is also a monitoring of the solar radiation in order to close the shutter if necessary and to protect the instrument.
Beside the events that are triggered by the X-rays also so called cosmics are formed on the CCDs by high-energy particles. X-ray events are of interest here in terms of their number and distribution and their appearance. Cosmics are a disturbing factor for the measurements.
The instrument software that controls and monitors the entire system is located in the DPU. The data sorting algorithm can detect false events that are not caused by the X-rays and this is where also the compression unit of the science data processing is located. The compressed and packetdata is then handed over from the DPU to the service module of the satellite where it is temporarily stored and sent to the ground station in the next X-band communication mode.

The compression software consists of several parts that can be switched one after the other, whereby each step fulfills its own task and can be specially parameterized.
Everyone knows a few examples of lossy compression methods from daily practice: MP3, JPEG, for example, but also voice transmission in cellular networks in which the codec only transmits the coefficients and thus the voice is synthesized again. Zip is e.g. a lossless compression method. Sure, you want to have all the data again with unzip.

If you what to learn more about SMILE just have a look at the Official ESA-SMILE webpage: `https://sci.esa.int/web/smile` or the project homepage of IfA for SMILE: `https://space.univie.ac.at/projekte/smile/`.
The goal of this master's thesis is to contribute to the implementation of the SDP for the SMILE-SXI.
The IfA has a long lasting experience in SDP for space missions and CHEOPS is the actual one using software of the IfA. This CHEOPS-SDP can be used as a base to derive the SMILE-SXI-SDP.

Also a Data-Simulator "DaSi" is available to create test data sets and just has to be modified in some details to easily create the necessary amount of data-space (Seelig, J. 2020).

The SMILE mission will also be used to introduce a new operating system which offers the possibility to control the individual parts of a flight SW-system (Luntzer A. 2017a), (Luntzer A. 2017b).

In addition to the compression algorithms of CHEOPS reused for SMILE, the Golomb coding from the PLATO mission (Loidolt D. 2021) was also incooperated in the SMILE-SDP.

The outline of the master's thesis is as follows:
In chapter 2 the astrophysical background of the SMILE mission is described with the main emphasis on the soft X-ray emissions.
Chapter 3 will give an overview of the SMILE mission and its technical environment.
Chapter 4 contains a brief description of Science Data Processing.
In the first step, I worked with full frames created with the Data-Simulator "DaSi" to compare several compression variations. The structure and creation of the FITS-files used for simulations are described in chapter 5. Here also the second kind of datasets which are used in SMILE, the event detection data, is described and a new Data Simulator "EDSim" is introduced, which is able to derive a set of event detection data packages out of the fits-files provided by the Data-Simulator "DaSi".
The first adaptation of the SDP to SMILE requirements (SW-Version 1.1) is shown and explained in chapter 6 together with a new data reduction method for event detection data while chapter 7 shows the test strategy. Chapter 8 gives the results of the tests with full frames and event detection data sets including the compression rates and runtimes.
Chapter 9 contains the usage of the new UVIE FlightOS proc_chain (SW-Version 1.2) which shall accelerate the compression especially in multicore environment.
Golomb coding is used in the PLATO project. Chapter 10 outlines an investigation of the usage of Golomb coding for SMILE and the implementation (SW-Version 1.3).
It should be noted, that SW-Version 1.3 can be used with or without the UVIE-FlightOS proc_chain just by only exchanging the main-procedure of the compression.
Afterwards in chapter 11 a short description is given how to integrate the Compression package into the SMILE-DPU emulation environment for testing purposes (SW-Version1.4) and later on into the real SW-environment of the InFlight Application-SW (IASW) in the DPU on one side and also into the Software-environment of the Ground-Segment. Figure D.1 in appendix D shows the development path of the SDP-software.
Finally chapter 12 contains a conclusion together with an outlook to possible further optimizations.
All programs and procedures mentioned in this masters thesis can be found in the git-source-repository *smile@herschel.astro.univie.ac.at:1722/home/smile/OBSW.git* in the branch *berndt*. The corresponding direcory-tree is shown in appendix C.

2. Astrophysical background

In this chapter, I want to give a short glimpse on the astrophysical aspects which are important for the SMILE mission, in particular the SXI observations.

2.1. Magnetosphere

The magnetosphere is the space around an astronomical object in which charged particles are influenced by its magnetic field. The outer limit is called the magnetopause. The inner boundary to the neutral atmosphere is the ionosphere (Voigt 2012).
Only stars or planets with an internal dynamo have a magnetic field. In the surroundings of the planetary body, the magnetic field resembles a magnetic dipole. Further outside, the field lines can be distorted by conductive plasma that comes from the sun (as solar wind) or nearby stars. The earth's active magnetosphere weakens the effects of solar radiation and cosmic rays, which can have harmful effects on living beings.
Many planets have an magnetosphere like Jupiter or our Earth. Even Mars has a magnetosphere but it is very weak. A magnetosphere provides a shielding mechanism against the solar wind which will have a huge influence on the atmosphere of a planet otherwise and can even blow it away.

A planetary magnetosphere is primarily formed by the magnetic field carried by the solar wind. The solar wind reaches a speed of 300 to 800 km/s near the Earth and has a density of 3 to 10 particles per cubic centimeter. The interplanetary magnetic field (IMF) of about 4 nT contains a nearly collision-free, low-density plasma. The solar wind compresses the magnetosphere on the day side to about ten earth radii (about 60,000 km) and pulls it apart on the night side to form a magnetic tail that can reach a distance of about one hundred earth radii (600,000 km).
Over Earth's equator, the magnetic field lines become almost horizontal, then return to reconnect at high latitudes. However, at high altitudes, the magnetic field is significantly distorted by the solar wind and its solar magnetic field. The magnetopause exists at a distance of several hundred kilometers above Earth's surface. Earth's magnetopause has been compared to a sieve because it allows solar wind particles to enter.

The solar wind flows around the earth at supersonic speed, that means its flow speed is bigger than the speed with which disturbances of the density or the pressure in the solar wind move (speed of sound). It is braked to subsonic speed at the bow shock wave. The area between the bow shock wave and the magnetopause is called magnetosheath. Part of the solar wind is also reflected on the bow shock wave, so that a pre-shock develops. The

magnetic field lines are closed on the day side, while on the night side they are extended far into space, here the "lobes" are formed. The so-called "cusps" form the transition. Here the particles of the solar wind can penetrate into the Earth's atmosphere.
The shape of the magnetopause, however, is not static, but changes very strongly over time while the tail literally "flutters" in the solar wind due to the changing magnetic field direction of the solar wind, since the expansion on the dayside is dependent on the momentum of the solar wind. Figure 2.1 shows the structure of the magnetosphere where the solar wind hits the earth's magnetic field from the left. Reconnections occur, which cause the earth's magnetic field lines to migrate from the left into the tail area to the right.

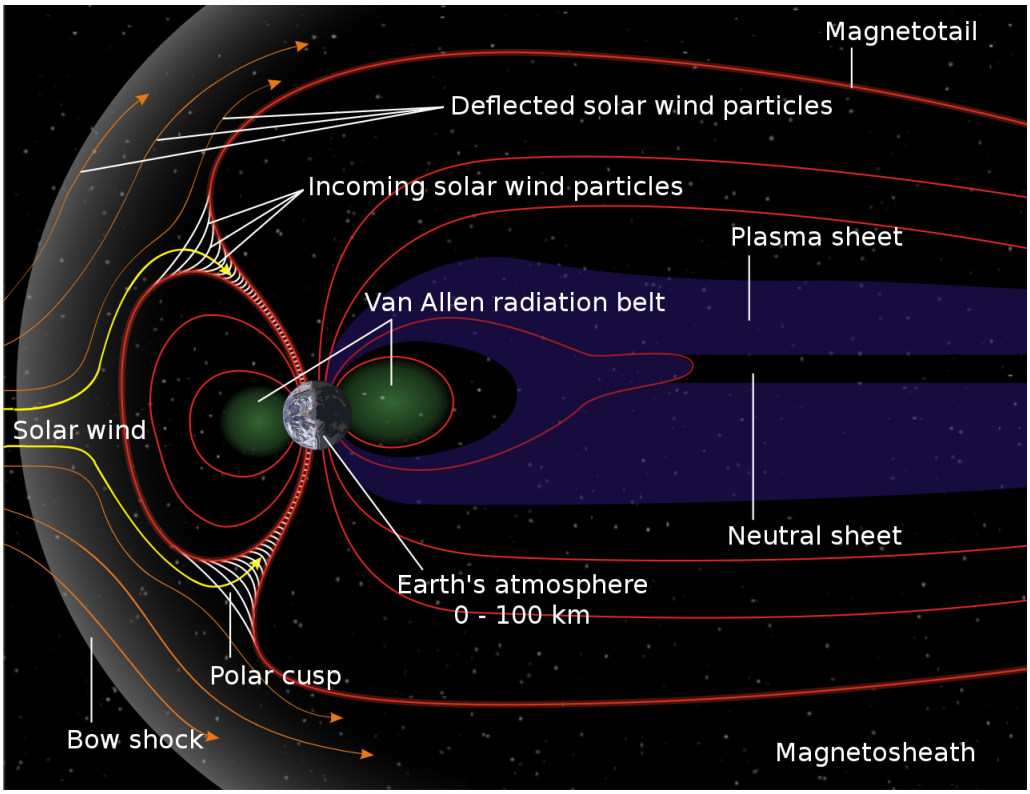


Figure 2.1.: Solar wind and magnetosphere (Gallagher, D. 2002)

Kelvin–Helmholtz instabilities occur when large swirls of plasma travel along the edge of the magnetosphere at a different velocity from the magnetosphere, causing the plasma to slip past. This results in magnetic reconnection, and as the magnetic field lines break and reconnect, solar wind particles are able to enter the magnetosphere. On Earth's nightside, the magnetic field extends in the magnetotail, which lengthwise exceeds 6,300,000 kilometers. Earth's magnetotail is the primary source of the polar aurora. Also, NASA scientists have suggested that Earth's magnetotail might cause "dust storms" on the Moon by creating a potential difference between the dayside and the nightside.

2.1.1. Structure of the magnetosphere

The structure of the magnetosphere is described in many articles. A brief summary can be found in (Gallagher, D. 2002) and his subsequent articles which I am using here for illustration. Figure 2.2 shows a schematic cartoon of the structure of the magnetosphere explaining the location of the following areas:

1) Bow shock
2) Magnetosheath
3) Magnetopause
4) Magnetosphere
5) Northern tail lobe
6) Southern tail lobe
7) Plasmasphere

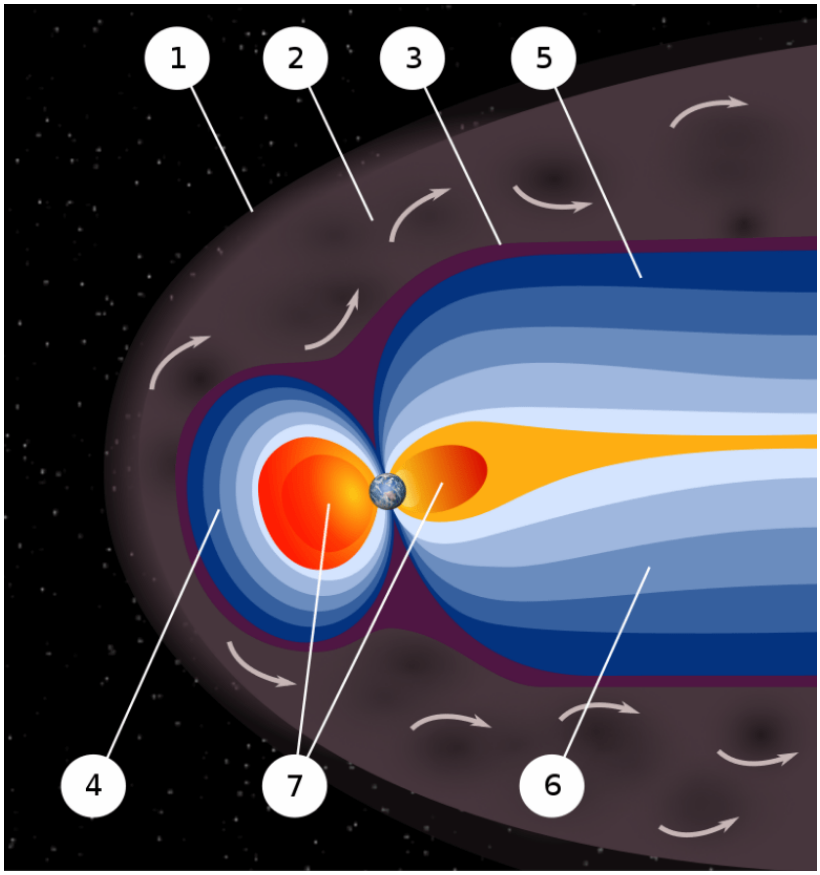


Figure 2.2.: Structure of the magnetosphere (Gallagher, D. 2002)

Bow shock

The bow shock forms the outermost layer of the magnetosphere. It is the boundary between the magnetosphere and the ambient medium. For stars, this represents the boundary between the stellar wind and interstellar medium. For a planet the speed of the solar wind decreases at that point as it approaches the magnetopause.
Earth's bow shock is about 17 kilometers thick and located about 90,000 kilometers from Earth.

Magnetosheath

The magnetosheath is the region of the magnetosphere between the above mentioned bow shock and the magnetopause. It is formed mainly from shocked solar wind and contains a small amount of plasma from the magnetosphere.
It is an area of high particle energy flux, where the direction and magnitude of the magnetic field varies randomly which is caused by the collection of solar wind gas that has undergone thermalization. It acts like a bumper block between the pressure from the flow of the solar wind and the barrier of the magnetic field from the object.
The shocked solar wind plasma flows around the magnetosphere through the magnetosheath. A relatively sharp transition from dense, shocked, solar wind plasmas to tenuous magnetospheric plasmas marks the magnetopause.

Magnetopause

The magnetopause is the area of the magnetosphere equalizing the pressure from the planetary magnetic field and the pressure from the solar wind. Because solar wind as well as planetary magnetic field contain magnetized plasma, the interactions between them are complex.

The structure of the magnetopause depends upon the Mach number of the plasma, the ratio of the plasma pressure to the magnetic pressure of the plasma (known as β), and the magnetic field. So the magnetopause changes size and shape as the pressure from the solar wind fluctuates.

Magnetotail

On the other side of the Earth, opposite to the compressed magnetic field there is the magnetotail, where the magnetosphere extends far beyond into space.
It contains two lobes, called the northern and southern tail lobe. Magnetic field lines in the northern tail lobe point towards the object while those in the southern tail lobe point away. The tail lobes carry only few charged particles opposing the flow of the solar wind. The two lobes are separated by a plasma sheet, an area where the magnetic field is weaker, and the density of charged particles is higher.

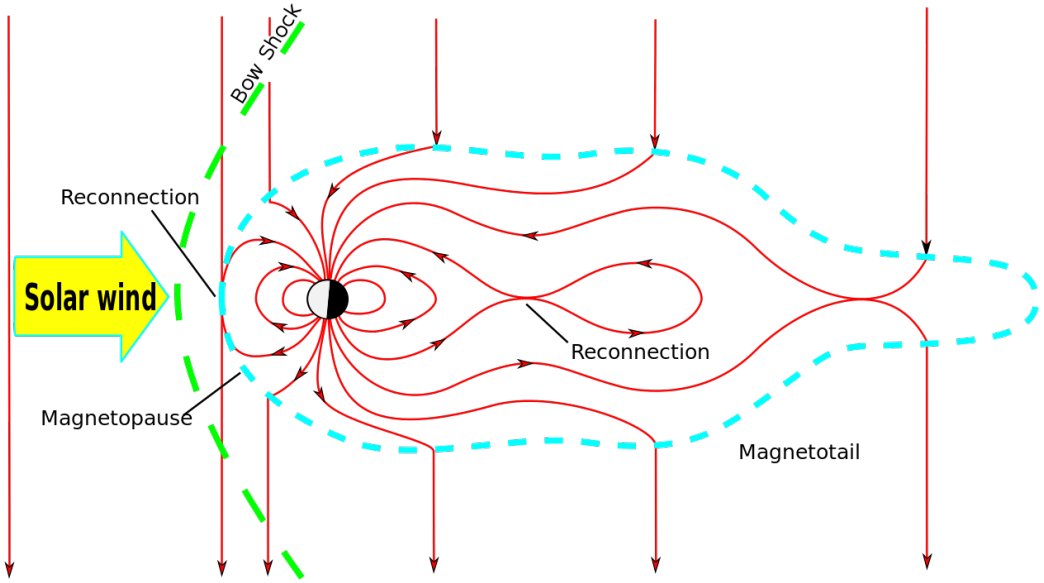


Figure 2.3.: Magnetosphere in the near-Earth space environment. (Wiki-GS 2001)

2.1.2. Geomagnetic storm

A geomagnetic storm is defined by the changes it causes in the geomagnetic field (Wiki-GS 2001). Among other things, the disturbance storm time index (Dst-Index) is used for the classification. This value is determined every hour and is available in almost real time. There are many influences on the magnetic field, so fluctuations in magnetic flux density of ± 20 nT are normal. For comparison: In Central Europe the horizontal component of the normal earth's magnetic field is around 20 μT.
The disturbance is triggered by shock wave fronts from the solar wind, which are caused by solar flares or coronal mass ejections (KMA) and take about 24 to 36 hours to reach Earth. It lasts about 24 to 48 hours, in individual cases several days - depending on the disturbances on the sun. The impact of the shock front, consisting of electrically charged particles, on the magnetosphere leads to a weakening of the earth's magnetic field, which reaches its minimum after about twelve hours.
The complete process is as follows: The Sun's magnetic field lines extend far into space and eventually reconnect. Plasma is thrown into space and the reconnected magnetic field moves with this plasma until it finally interacts with the Earth's magnetic field and thus changes it. In figure 2.3 the vertical lines represent the magnetic field of the plasma coming from the Sun and it shows how the interaction starts at the dayside with a reconnection between the 2 magnetic fields. Charged particles are transported along the altered field lines of the Earth's magnetic field into the Earth's atmosphere and generate the aurora borealis. The changes in the Earth's magnetic field continue on the night side of the Earth, extend it far into space and now also lead to reconnections there, whereby the energies released in the process have a retroactive effect on the Earth.

A geomagnetic storm is typically divided into three phases:

- Initial phase
  The initial phase is characterized by a weakening of the magnetic field by around 20–50 nT within a few dozen minutes. Not every storm event is preceded by such an initial phase, and conversely, not every such disturbance of the magnetic field is followed by a magnetic storm.

- Storm phase
  The storm phase begins when the disturbance is greater than 50 nT, which is an arbitrarily drawn limit. In the course of a typical magnetic storm, the disturbance continues to grow. The strength of a geomagnetic storm is called "moderate" if the maximum disturbance is less than 100 nT, as "intense" if the disturbance does not exceed 250 nT and otherwise as "super storm".
  A maximum attenuation of about 650 nT is rarely exceeded, which corresponds to about three percent of the normal value. The phase lasts a few hours and ends as soon as the strength of the disturbance decreases, i.e. the Earth's magnetic field begins to grow again to its typical strength.

- Recovery phase
  The recovery phase ends when normal value is reached and can last between 8 hours and a week.

Figure 2.4 shows an MHD-model of a geomagnetic storm and it is clearly visible, that during the storm (right side) the magnetosphere of the Earth is changing in comparison to its expansion before the storm (left side).

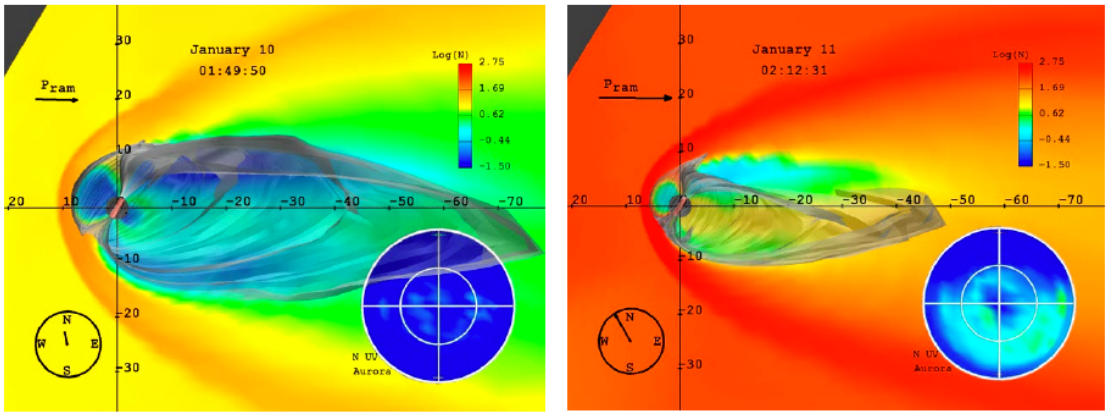


Figure 2.4.: Storm MHD model simulation based on data of a storm on 10 January 1997. Left: Magnetosphere before the storm and right: Magnetosphere during the storm. The plasma density is shown in color in the background and the Earth's magnetic field is shown in grey. (Branduardi-Raymont et al. 2018)

Substorms

A magnetospheric partial storm (substorm) is a short-term disturbance in the magnetosphere of a planet (see figure 2.5 A to E). It causes the release and redirection of energy from the dayside reconnection (A) influencing a magnetic flux which convect over the poles (B) and is stored as magnetic energy in the magnetotail lobes (C). This stored energy accumulates (D) until an explosive release returns closed flux to Earth into the near-polar ionosphere (E). Substorms may result from changes in the external driving of the magnetosphere and/or internal magnetotail instabilities.

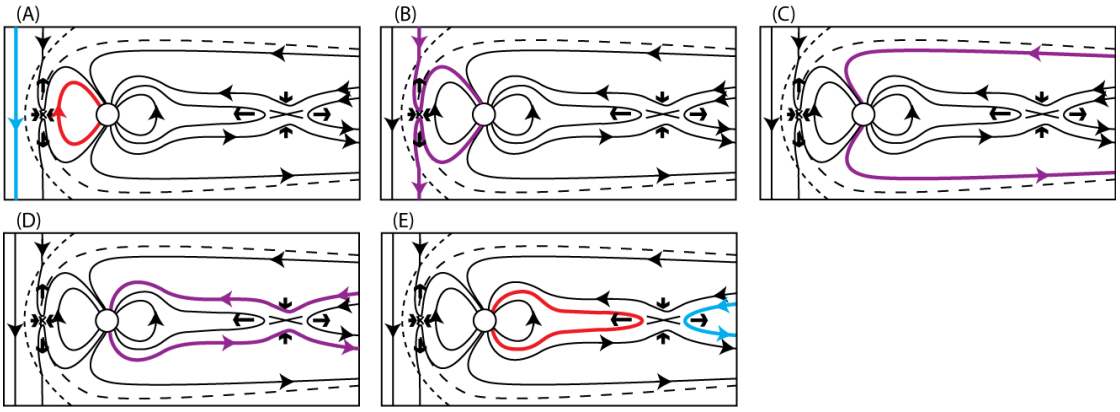


Figure 2.5.: Substorm phases. (Branduardi-Raymont et al. 2018)

Visually it appears as a sudden lighting up and increased movement of aurora arcs. Such partial storms occur over a period of a few hours, are mainly observed in the polar region, introduce few particles into the radiation belt and are relatively frequent, being often only a few hours apart. They arise when a plasmoid becomes detached in the magnetic field tail, in particular there is a reconnection of the field lines. This can also happen in times when no magnetic storm from the sun hits the earth. One cause could be the interaction of the earth's magnetic field with the IMF, the interplanetary magnetic field.

2.2. Sources of X-ray radiation

There are several kinds of X-ray radiation sources also producing several different kinds of X-ray radiation from soft to hard X-rays (Dennerl, K. 2013).

2.2.1. Thermal emission

The sun's X-rays are generated in its corona, which contains very thin plasma that is several million degrees hot. This is the only place in the solar system that has a sufficiently high temperature to thermaly generate X-rays. A special feature is the recently observed flight of a comet (C / 2011 W3) through the inner solar corona (Raymond et al. 2018), in

which the gas of the comet was so strongly ionized that it could be observed in X-ray light due to its thermal emission.

2.2.2. Scattering of solar X-ray photons

Since the sun illuminates its surroundings not only with visible light, but also with X-rays, this process should take place with all objects that are hit by solar X-rays (Trümper, J. 1999). Since it has small effective cross sections, however, it is only effective for objects with sufficient density. Such objects are the atmospheres or surfaces of planets, planetary moons and planetary rings as well as of asteroids and comet nuclei. Evidence has been found so far for the planets Mercury, Venus, Earth, Mars, Jupiter and Saturn with its ring, the Earth's moon and for the asteroids 433 Eros and 25143 Itokawa (Dennerl, K. 2013). In the case of gases with very low densities, such as the exospheres of planets and comet atmospheres or even the interstellar gas that flows through our solar system, it plays almost no role.

2.2.3. Energetic electrons and ions

In addition to photons, our sun also emits charged particles, the so-called solar wind. Part of it are energetic electrons which, when deflected, can generate X-ray "Bremsstrahlung". Sufficiently strong magnetic fields, such as those in the magnetospheres of Earth and Jupiter, are required for this. In this way, X-rays are generated in the polar regions. A special feature is Jupiter, which, due to its high magnetic field, is able to "independently" accelerate electrons in its magnetosphere to such an extent that they can generate X-rays. The electrons accelerated in this way are probably a cause of the X-rays from the Io-Plasmatorus, a ring around Jupiter at the orbit of its moon Io, which is fed by its volcanism. Energetic electrons can also cause an X-ray emission through impact ionization, which in this case consists of the characteristic lines of the atoms hit. This effect is observed in the polar regions of the Earth's atmosphere as well as on the night side of Mercury.
Due to its high magnetic field, Jupiter is also able to accelerate ions to such an extent that they can generate X-rays via Bremsstrahlung or shock excitation. Indications of this additional radiation component were found in the X-ray spectrum of Jupiter. The direct impact of energetic ions on the surfaces of Jupiter's moons Io and Europa is seen as a possible cause of their X-ray radiation (Dennerl, K. 2013).

2.2.4. Reloading between highly ionized solar wind atoms and neutral gas

The solar wind, which essentially consists of protons, helium nuclei and electrons, contains about one per thousand heavier atoms, which are highly ionized when passing through the hot solar corona and, as a result of the subsequent rapid expansion, maintain this degree of ionization until they hit matter. When they recombine with electrons, X-rays are emitted. Since radiation occurs only at certain energies specified by the recharging process, it contains information about the degree of ionization and the chemical composition of the

solar wind. For the charge reversal process the interaction cross-sections are considerably larger than those of the other X-ray emission processes (Kuntz 2018). Due to this fact, this process unfolds its full strength when the solar wind encounters extensive gas clouds of low density, such as those found in comets, the exospheres of Venus, Earth and Mars and the entire heliosphere. A special case here is also Jupiter, which, due to its high magnetic field, is able to generate its own ions in addition to the solar wind ions. X-ray radiation caused by this charge exchange is now the main field of interest of the SMILE-SXI.

2.3. Solar Wind Charge Exchange (SWCX)

The particles of the solar wind hit the gas, mostly hydrogen, in the area of the magnetosheath. Charge exchange occurs between the ions and neutral atoms such as hydrogen. The ions grab the electrons from the neutral atoms, so to speak.
The ion has now a high energy level. During the subsequent relaxation of the ion, i.e. the transition of the electron towards an orbit with lower energy level, the energy difference is emitted as soft X-ray or extreme ultra violet radiation (Dennerl, K. 2010).
This is a different physical process than the "Bremsstrahlung", where the energy released by the braking process of electrons is radiated. The process is called "Solar Wind Charge Exchange" (SWCX) (Kuntz 2018).

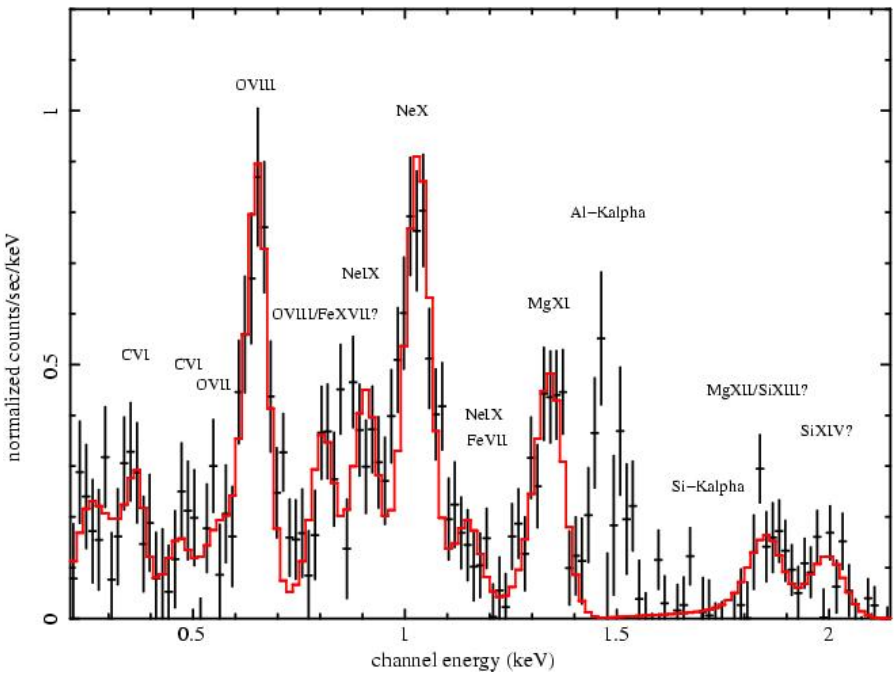


Figure 2.6.: Spectrum of the ions in the magnetosheath (Carter & Sembay 2008)

Figure 2.6 shows a spectrum of the ions in the magnetosheath; which is mainly oxygen, neon and carbon, all of them multi-ionized. The solar wind shows 2 different speed

components: The slower interstream solar wind (150-400 km/s corresponds to 100-900 eV/u (electron volts per mass unit of impact energy)) and the faster almost solar wind (500-800 km/s corresponds to 1300-3000 eV/u) of the coronary holes (Weigert et al. 2016). The first evidence of the X-rays triggered by this was obtained with ROSAT in the course of observing comets.

2.3.1. Soft X-ray emission

Soft X-ray emission is generated throughout the terrestrial magnetosheath as a consequence of charge transfer collisions between heavy solar wind ions (see figure 2.6) and geocoronal neutral (gas) atoms. The solar wind ions resulting from these collisions are left in highly excited states and emit extreme ultraviolet or soft X-ray photons. This X-ray emission can be used to remotely sense the solar wind flow around the magnetosphere. Also bow shock and magnetopause could be located from these images (Dennerl, K. et al. 2012). Similar X-rays are produced in the heliosphere. To get a clear picture of the soft X-ray emission produced in the Earth's magnetosphere it is necessary to eliminate this background emission caused by the heliosphere.

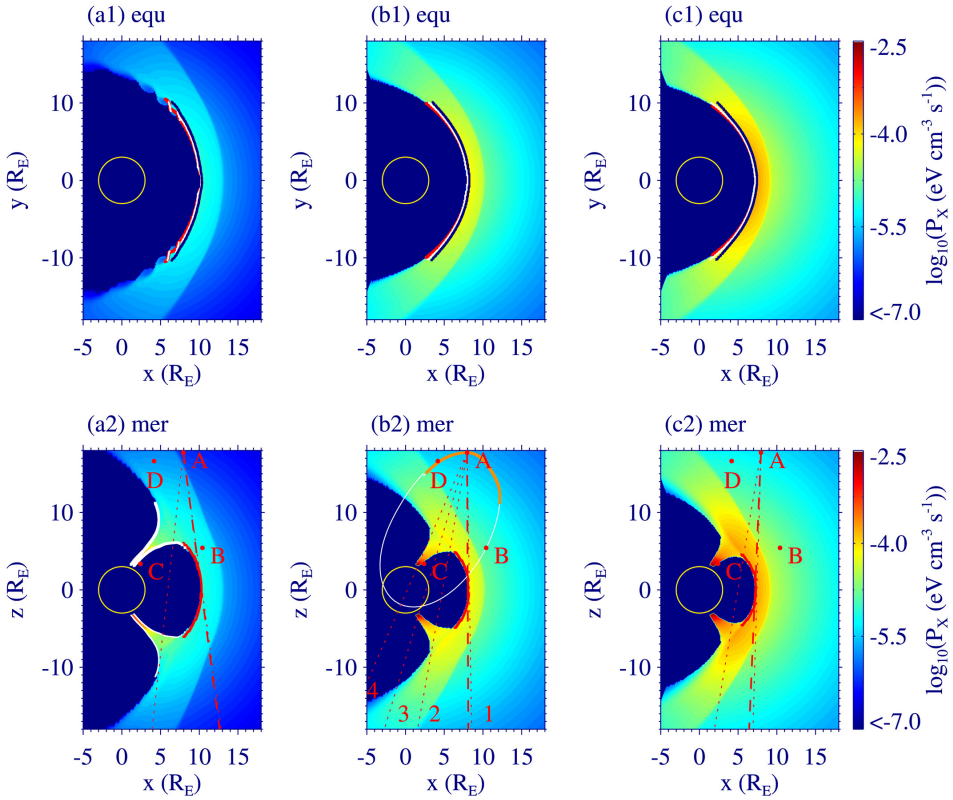


Figure 2.7.: Strength distribution of the X-rays in the equatorial and perpendicular directions. (Sun et al. 2019)

A couple of simulations with the strength distribution of the X-rays in the equatorial and perpendicular directions is shown in figure 2.7. The highest intensity occurs in those regions where the magnetic field lines are most strongly deflected by the solar wind.

2.3.2. EUV emission

Beside the soft X-ray emissions, there is also extreme ultraviolet (EUV) emission produced by solar wind charge exchange (SWCX) in the Earth's magnetosheath. These EUV have a wavelength of 30 nm (or 300 eV/u).

Such kind of emissions are depending on flow speed, the density, the temperature, and the He^{2+}/H^{+} density ratio of the solar wind. Their intensity is highly variable (Fei H. et al. 2014).

Simulations show the dependencies on flow speed (figure 2.8) and density (figure 2.9):

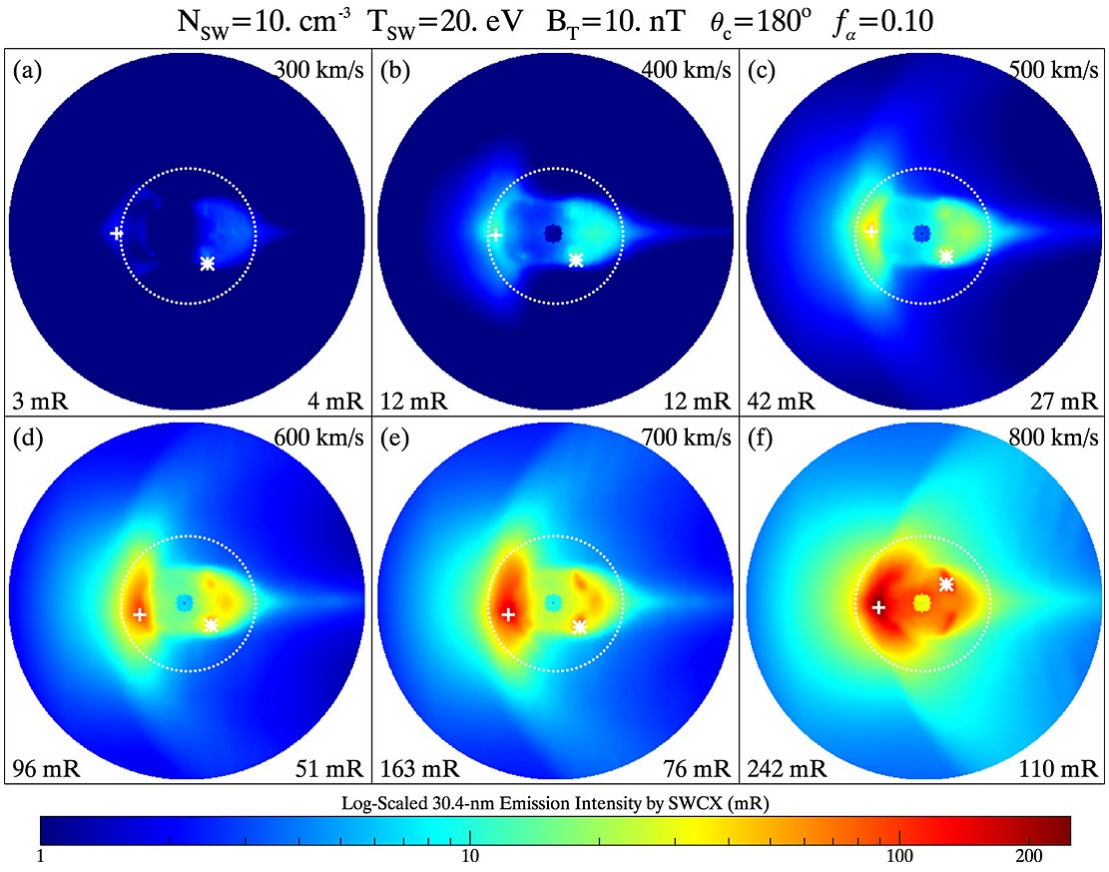


Figure 2.8.: Dependence of SWCX on flow speed (Sibeck et al. 2018)

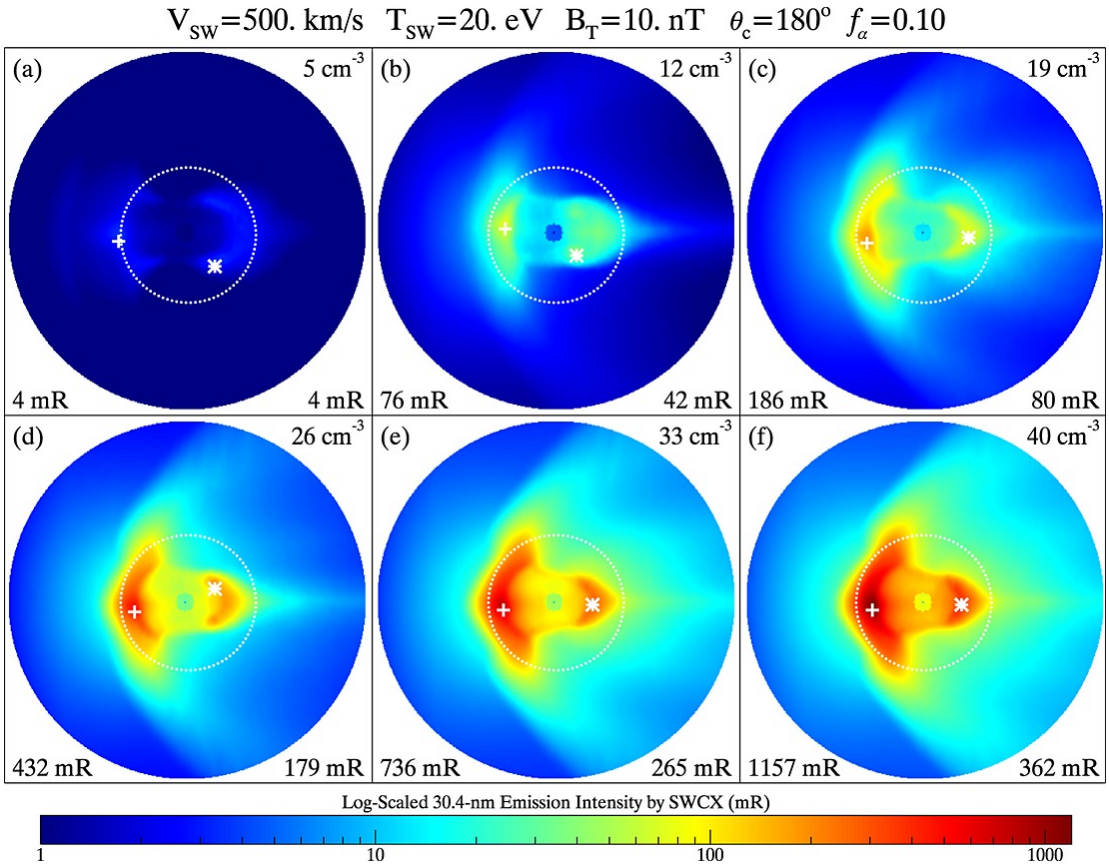


Figure 2.9.: Dependence of SWCX on solar wind particle density (Sibeck et al. 2018)

So both, extreme ultra violet (EUV) and soft X-ray emission, are caused by the same physical processes and it is just a matter of the energy of the solar wind particles and their excited states, which kind of radiation will be emitted. SMILE SXI offers the possibility to detect both kind of radiation by changing the binning, using 6x6 for soft X-ray and 24x24 for EUV.

2.3.3. Expectations of the SMILE mission

SMILE shall investigate the interactions between the solar wind and the magnetosphere of the Earth. In particular when looking at the SXI two kinds of astrophysical events are forming the field of interest, the soft X-ray events and the extreme ultraviolet emission both caused by the solar wind charge exchange (Branduardi-Raymont et al. 2018).
The CCDs of the SXI are able to detect both kind of radiation by adjusting the binning of the CCDs. 6x6 binning will be used for soft X-ray and 24x24 binning for the EUV.
Beside these events, there are additional impacts on the CCDs caused by the cosmic radiation, i.e. high energy particles, which also induce effects disturbing the measurements. So for a correct analysis of the events an event detection is necessary.

3. SMILE Mission

SMILE (Solar Wind Magnetosphere Ionosphere Link Explorer) is a new, independent mission to observe the solar wind-magnetosphere coupling through simultaneous X-ray imaging of the magnetosheath and the pole cusps, UV imaging of global aurora distributions and in situ solar wind/magnetosheath plasma and magnetic field measurements using four scientific instruments.

The solar wind charge exchange (SWCX) process produces X-ray emissions which was first observed on comets and later on found also near the Earth's magnetosphere. So X-ray images of the magnetosheath and cusps will be made to investigate the interaction between solar wind and magnetosphere of the Earth.

One of the scientific goals of SMILE is to record the substorm cycle via X-rays imaging of the dayside magnetosheath and to investigate the consequences on the night side with UV images of the aurora.

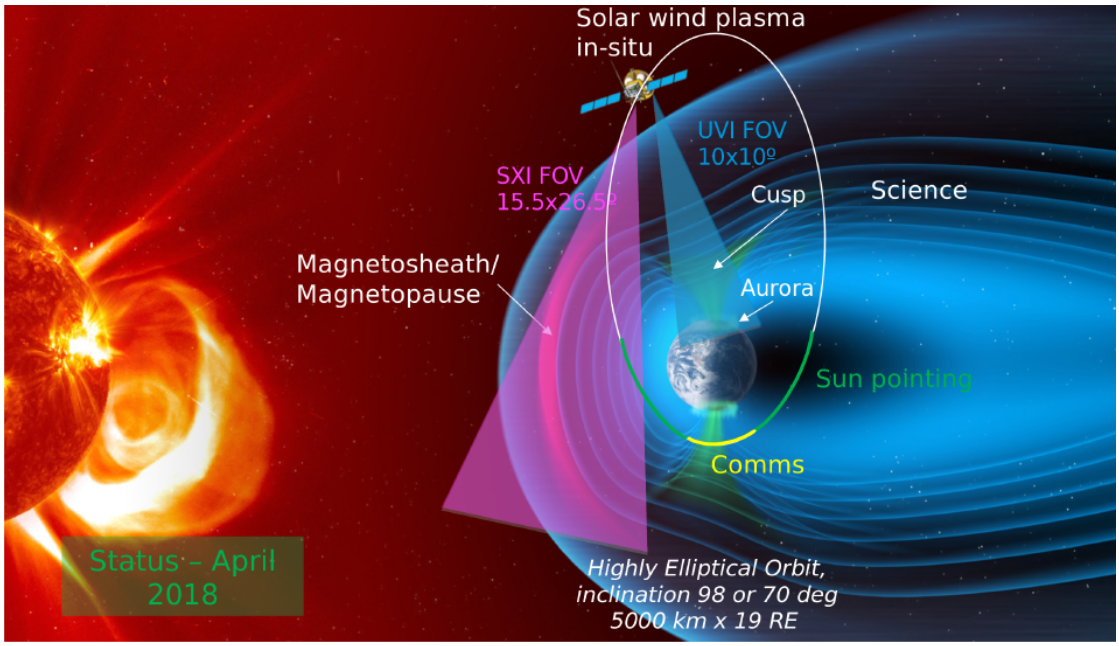


Figure 3.1.: SMILE Mission with Observation View of SXI (magenta) and UVI (blue) (SMILE-team 2018)

SMILE is a joint mission between ESA and the Chinese Academy of Sciences involving scientists and companies from about 20 countries including Austria with the IfA in Vienna and the IWF in Graz. An overview of the mission itself, its scientific goals, the satellite and its instruments is given in (Branduardi-Raymont et al. 2018). Here I just want to mention the most important bullet points out of this overview.

3.1. Mission

The start of SMILE was initially planned for November 2023. Due to some reasons it will probably be postponed for several month. The launch will take place with an Ariane or Vega rocket from Kourou, ESA's spaceport.

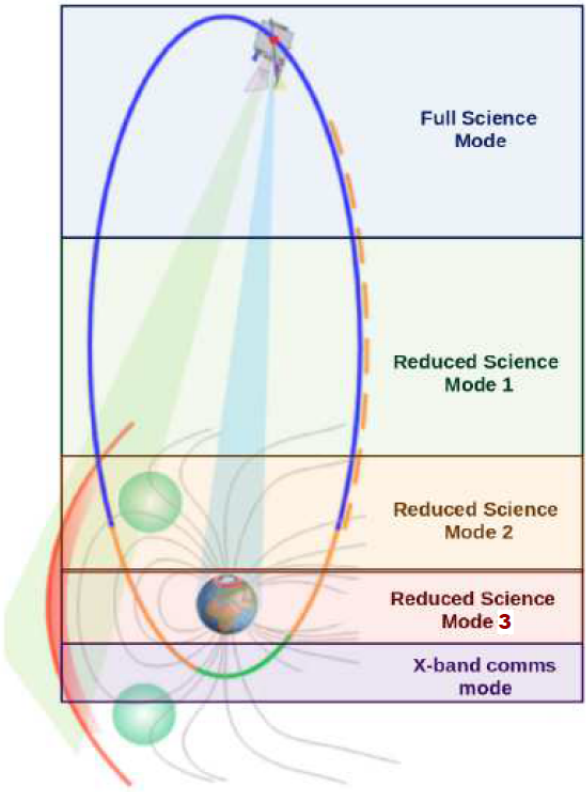


Figure 3.2.: SMILE orbit (SMILE-team 2018)

In order to be able to observe the polar regions and at the same time the interaction area in the magnetosphere, the satellite will move in a strongly eccentric orbit with high inclination ($90 \pm 27°$) around the earth. In the area closest to the earth at about 5,000 km, it gets rid of the data, with a time window of about 10-11 hours available. The total orbital time is about 51 hours.

In the area furthest from earth with an apogee of over 120,000 km (i.e. about a weak third of the distance between earth and moon), all of the instruments will work in the so-called "Full Science Mode".
In the Reduced Science Modes, some devices are switched off to protect them from excessive radiation. In mode 1, UVI is switched off, in mode 2 the X-ray detector is shut down and in mode 3, the area close to the earth, only the magnetic field and ion detectors are running. Communication takes place in the X-band (8.4 GHz) during the orbital path closest to the earth.

3.2. The satellite

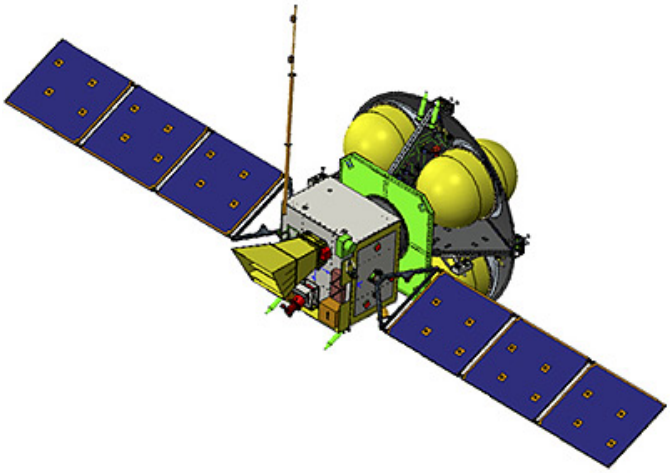


Figure 3.3.: SMILE Satellite, solar panels open (Branduardi-Raymont et al. 2018)

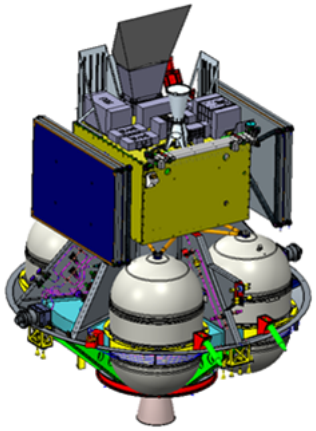


Figure 3.4.: SMILE Satellite, solar panels closed (Branduardi-Raymont et al. 2018)

Figures 3.3 and 3.4 show schematic drawings of the SMILE satellite. It consists of:

- The platform (PF) carrying the propulsion module (PM) recognizable by the 4 yellow fuel tanks and the service module (SVM) with the board computer.
- The payload module (PLM) which carries the scientific equipment.

The SVM provides the power for the satellite and also the X-band ground-link. Its board computer takes control of the whole spacecraft including the data flow from the scientific instruments (Branduardi-Raymont et al. 2018).

3.3. Scientific equipment

SMILE carries 4 scientific instruments, two in situ measuring instruments (LIA and MAG) and two imagers (UVI and SXI):

3.3.1. LIA - Light Ion Analyzer

LIA, the analyzer for light ions is supposed to detect solar wind and magnetosheath ions under various conditions by measuring the three-dimensional velocity distribution of the particles. This mainly concerns protons and helium nuclei, i.e. alpha particles. It has 2 top-hat-type electrostatic analysers and can sample the full 4 π three-dimensional distribution of the solar wind, measuring ions in the range of 0.05 to 20 keV at up 0.5 second time resolution.

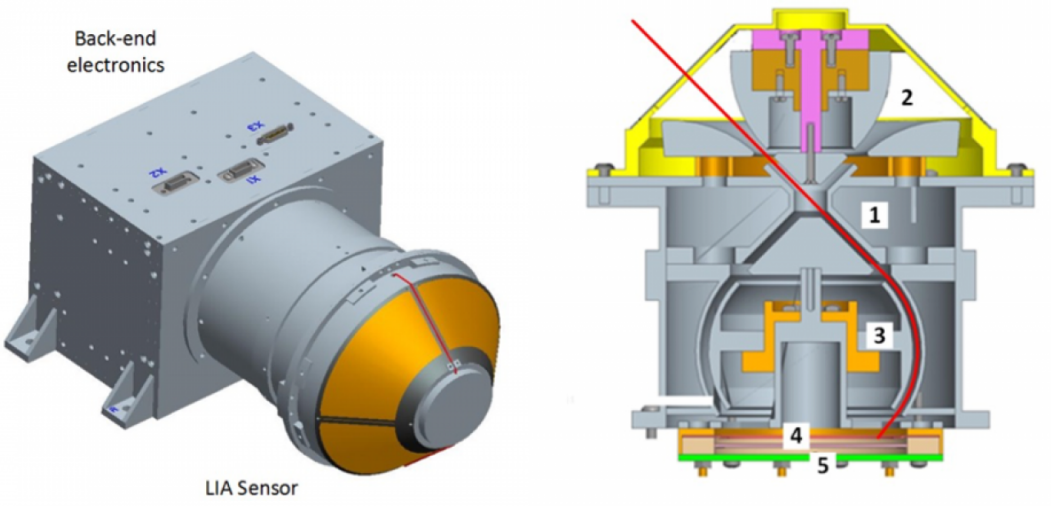


Figure 3.5.: SMILE Light Ion Analyzer (LIA)
Left panel: 3D rendition of LIA main sub-systems. Right panel: Cut-away view of the LIA electro-optical sensor. A typical path of a detected ion is shown as a red line. (Branduardi-Raymont et al. 2018)

The two LIA instruments are designed to operate simultaneously and synchronously throughout the entire science orbit. The LIA system has 2 different scientific operating modes: Normal Mode (NM) and Burst Mode (BM). Normal Mode will primarily be used to provide full 3-D moments of the plasma ions at a cadence of 2 seconds. In this mode, the sensor will scan 62 energy values for 8 Elevation angles, acquiring 24 Azimuth angles simultaneously with each sample. The instruments will be in this mode for most of the science orbit. Burst Mode is providing a mode to increase the time resolution from 2 seconds (NM) to 0.5 second but at the same time decrease the number of energy steps from 62 to 30 and reduce the number of Elevation steps and the Azimuth resolution. Burst Mode will primarily be used near apogee.

3.3.2. MAG - Magnetometer

The **MAG** magnetometer is there to measure the orientation and magnitude of the magnetic field in the solar wind and detects solar wind shocks or discontinuities by measuring the changes of the magnetic field which are caused by such kind of events. 2 tri-axial sensors will be mounted on a 3-m-long boom some 80 cm apart. MAG will measure the three space components of the magnetic field in the range +/- 12800 nT.

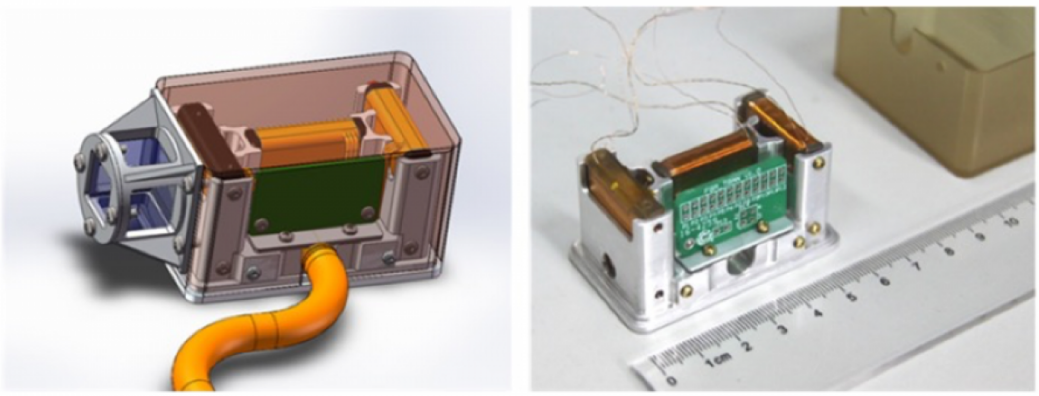


Figure 3.6.: SMILE Magnetometer (MAG) Sensor head.
Left panel: 3D rendition from CAD model. Right panel: Finished engineering model. (Branduardi-Raymont et al. 2018)

Each of the two MAG sensor heads comprises three ring-core fluxgate sensors oriented along three orthogonal axes, thus sampling the three spatial components of the local magnetic field. The fluxgate sensors are mounted on an aluminium platform and are covered with a polyamide lid. The design of the MAG sensor heads is illustrated in figure 3.6 (left side). Both sensor heads of MAG are designed identically.

3.3.3. UVI Ultra Violett Imager

UVI, the UV Imager, is, as the name suggests, a UV camera that is supposed to examine the northern polar regions. It takes a picture every 60 seconds with a 10-degree field resolution and has a 1024 x 1024 pixel CMOS detector. It will have a spatial image resolution at apogee of 150 km, and will use four thin film-coated mirrors to guide light into its detector. The imager is provided by Canada.

UVI consists of three partitions: UVI-Camera (UVI-C) and UVI-Electronics (UVIE) connected via a harness (UVI-H). This partitioning also represents the division of responsibilities between the Canadian (UVI-C & UVI-H) and China Academy of Science contributions (UVI-E). The UVI-C unit further contains contributions from CSL, Belgium in the form of mirror coating design and manufacture. To minimise risks for the assembly, a backup plan based on commercial filter coatings is pursued in parallel.

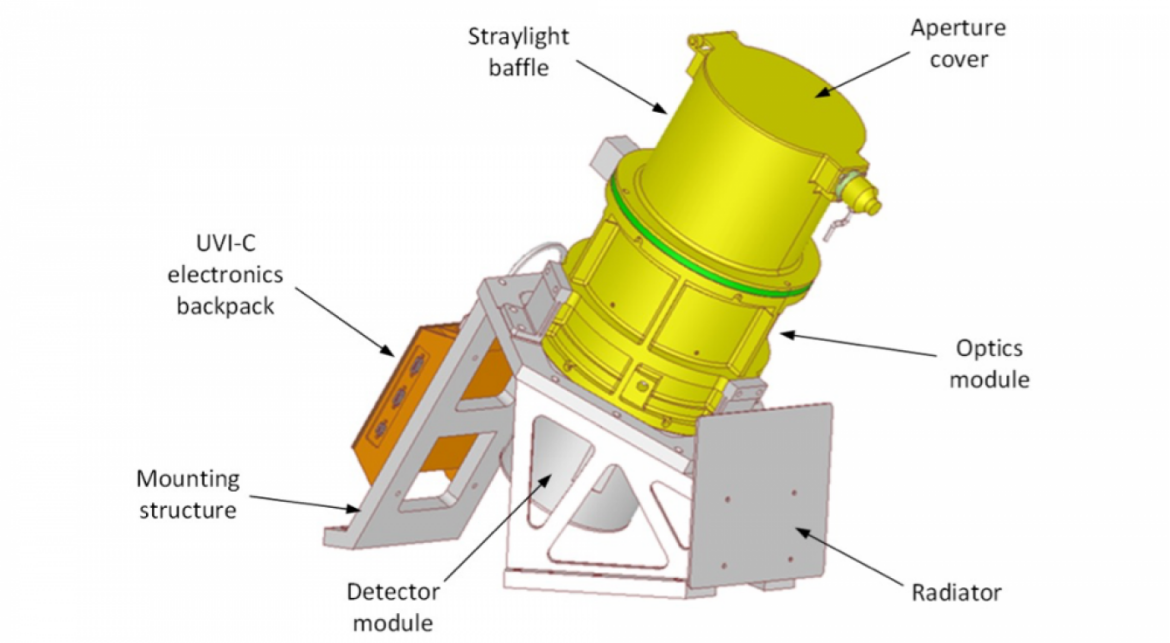


Figure 3.7.: SMILE UltraViolett Imager (UVI) (Branduardi-Raymont et al. 2018)

Simultaneously with the SXI observations the UVI instrument will monitor Earth's Northern aurora to link the processes of solar wind injection into the magnetosphere with those acting on the charged particles precipitating into the cusps and eventually the aurora.

3.3.4. SXI - Soft X-ray Imager

The Soft X-ray Imager **SXI** detects the soft X-ray radiation that arises from the interaction between the solar wind and the magnetosphere. Beside soft X-rays the imager is also able to detect Extreme Ultra Violet radiation and the CCDs of the SXI are able to change between 2 different binning modes to collect these 2 different kinds of radiation.

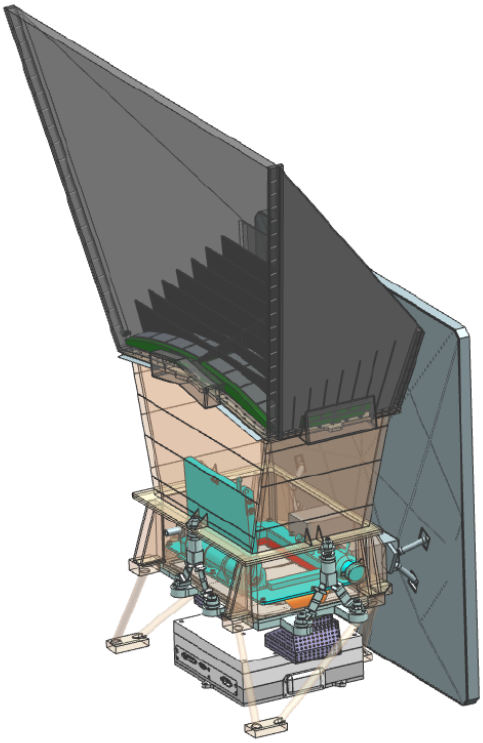


Figure 3.8.: SMILE Soft X-ray Imager (SXI) and Front End Electronics (FEE) (Branduardi-Raymont et al. 2018)

The Soft X-ray imager consists of four components:

- The SXI Telescope with the tube structure, the micro pore optic, straylight baffle, detector plane with the CCDs and a radiation shutter.
- The thermal control system with the radiator.
- The Front End Electronic (FEE) controlling the CCDs and also being able to detect the single X-ray events.
- The Back End Electronics carrying the Radiation Shutter Electronics and the Data Processing Unit (DPU) with the necessary software including also the Science Data Processing.

3.4. Details of SXI

SXI detects the soft X-ray radiation that arises when the solar wind interacts with the magnetosphere. It has two CCDs with a size of 4510 x 4510 pixels each, which resolve the X-rays with a depth of 16 bits (18 µm per pixel). The detector is normally operated in 6x6 binning mode (108 µm per pixel). Soft X-ray radiation lays between 0.2 and 2.5 keV, i.e. in the range of 5 nm to 0.5 nm wavelength. 1 nm corresponds to 1.24 keV, ($\lambda = hc/E$).
SXI is also able to operate in a 24x24 binning mode to detect high-energy UV light (extreme ultra violet - EUV). It should also be noted that the actual image area is 3791 x 4510 pixels, the remaining of 719 x 4510 pixels is actually storage area for the readout cycle. In principle, the instrument can generate an image every 5 seconds.

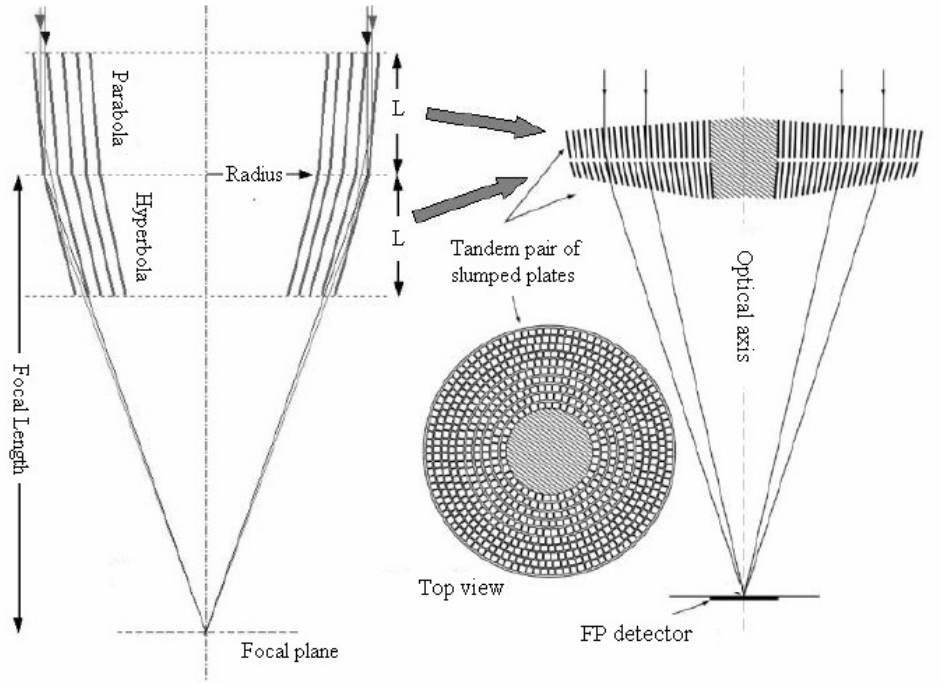


Figure 3.9.: Design principle of micro pore optics (Wallace, K. et al. 2007)

The optical field is quite large covering 15.5 to 26.5 degrees. Micro pore optics (MPO) are used to better focus the X-rays on the individual pixels (Mini Wolter). These kind of optics is also called "lobster eye" (figure 3.9) and the micropores can be stacked radially or arranged as square channels as it is done for SMILE-SXI (Branduardi-Raymont et al. 2018) with 40 μm MPO-poresize, 40 mm pore-length and 300 mm focal length. The same technology is used in the MIXS Instrument on the ESA BepiColombo Mission.
As always when focusing X-rays it is necessary to choose a grazing incidence optics otherwise the X-rays will not be reflected but will enter into the mirror-material. This principle is used in every Wolter-telescope but these kind of telescopes have large tubes

and in the SXI-implementation short tubes are necessary and single events shall be focused on one or in maximum a few pixels. Therefore the micro pore optics (which is acting like a 2-dimensional array of mini-Wolter-telescopes) is chosen.
The Institute for Astrophysics contributes with software for the data processing unit. The detector and readout unit are delivered by Great Britain, the IWF in Graz is providing the DPU hardware. Figure 3.10 shows the structure of the SXI.

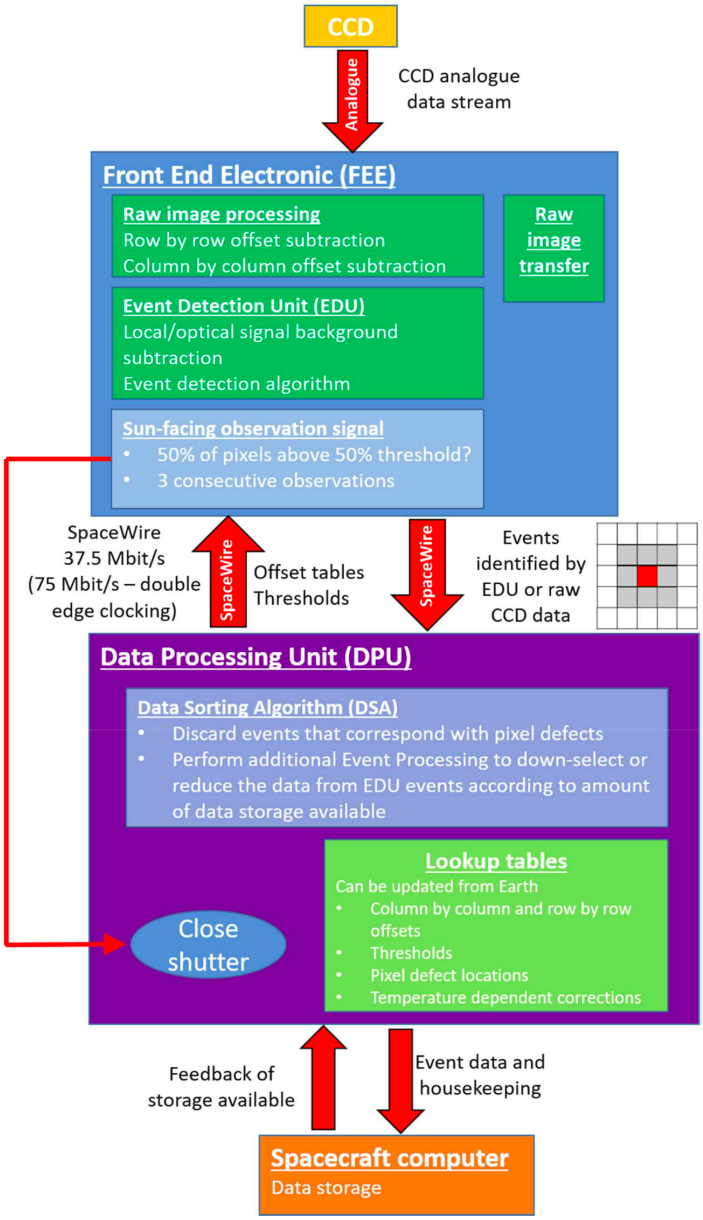


Figure 3.10.: SXI structure (Soman M. and Randall G. 2020)

The CCDs are controlled by the front end electronics (FEE). The FEE reads out the images and does an initial image processing and the transfer to the DPU. The FEE contains an event detection unit (EDU). It can already recognize X-ray events (single pixel threshold or local maximum) and deliver imagettes (image sections 5x5 pixels) that only include these events. Beside these single events, also complete images are transferred. There is also a monitoring of the solar radiation in order to close the shutter if necessary and to protect the instrument.

Apart from the X-ray events also cosmics are formed on the CCDs by high-energy particles. Only the X-ray events are of interest in terms of their number and spatial distribution and appearance. Cosmics are just disturbing the picture.
The instrument software that controls and monitors the entire system is located in the data processing unit (DPU). The data sorting algorithm can detect false events that are not caused by the X-rays and this is where also the compression unit of the science data processing is located. The finished data is then handed over from the DPU to the service module of the satellite, where it is temporarily stored and sent to the ground station in the next X-band communication mode.

Instrument Software

Figure 3.11 shows the rough structure of the SW-layers:

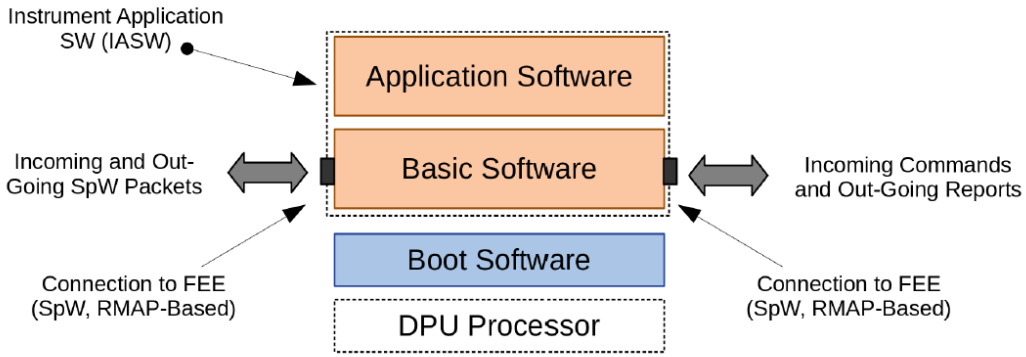


Figure 3.11.: SXI SW-layer (Mecina, M. et al. 2020)

The Boot Software on the DPU processor is used for starting up the system. The Basic Software takes care of the communication with other modules and, based on this, the Instrument Application Software (IASW) is controlling the instrument, does the housekeeping and the scientific data processing.

Science Data Processing (SDP) is a part of the IASW and the compress function is part of SDP.

4. Science Data Processing

4.1. Motivation

One of the problems for space missions is the growing amount of raw data collected by the bigger and bigger CCDs used for cameras and other image collecting instruments. Such big amount of data can hardly be transmitted to ground stations and it is necessary to use some kind of data processing before the relevant science data can be collected into a transmission channel fitting format.

Two main procedures for science data processing can be applied:

- Scientific processing of the data using appropriate scientific algorithms.
- Data compression to reduce the data for the transmission.

In this chapter mainly the second option, the data compression, is described and how it is used for the SMILE mission. Typically, a device that performs data compression is referred to as an encoder, and one that performs the reversal of the process (decompression) as a decoder.

4.2. Basics about compression

There is a lot of literature about data compression and many methods how it can be done. I just want to give a short overview of the most important fact but don't want to go into every detail. Overview and basic concepts of compression methods can be found in (Lelewer. D. A. & Hirschberg S. 2007), (Pu I. 2004) or (Sidoriv K. & Marshall D. 2006). The data compression is a process in which the amount of digital data is compressed or reduced. This reduces the storage space required and the time it takes to transfer the data. In communications engineering, the compression of messages from a source by a sender is known as source coding.

When compressing data, an attempt is always made to remove redundant information. For this purpose, the data is converted into a representation with which all - or at least most of the information can be presented in a shorter form. This process is carried out by an encoder and the process is known as compression. The reverse process is known as decompression.

One speaks of **lossless compression**, lossless coding or redundancy reduction when exactly the original data can be recovered from the compressed data. This is necessary,

for example, when compressing executable program files. Because most real-world data exhibits statistical redundancy lossless compression is possible. Well-known methods are run length coding, LZW (Lempel-Ziv-Welch algorithm) or Huffman coding. Among the most popular examples for lossless compression of images are GIF, PNG, TGA. A good compression rate can only be achieved with these methods if the images are convenient for these compression algorithms. This means that they should have the largest possible areas with the same color or the same pattern, for PNG possibly also with color gradients.

Here are some of these lossless image compression types:

- Portable Network Graphics (PNG) compresses images to keep them small by finding patterns in a photo and compressing them together. The compression is reversible. Once a PNG file is opened, the image is exactly restored. The PNG graphic format uses Deflate, a combination of LZ77 and Huffman coding, as the compression method. The good compression of PNG can be explained by the additional use of predictive coding.

- The graphic format GIF uses the Lempel-Ziv-Welch algorithm for compression and therefore usually does not achieve as good a compression as PNG. In addition, it can only display a maximum of 256 colors per single image. With GIF, the color information is stored in a color table. For simple drawings, black-and-white photographs, 256 colors or shades of gray are usually still sufficient today. More complex images such as color photos or drawings with extensive color gradients must therefore be reduced to 256 colors or less before being saved (color quantization). The resulting color levels or dithering effects are particularly annoying in large images.

- Windows Bitmap (BMP) is a file-format desigend by Microsoft. Windows bitmaps allow color depths of 1, 4, 8, 16, 24 or 32 bpp (bits per pixel), although not all bits actually have to be used with 16 and 32 bpp. The "biCompression" in the file-header indicated if and how the data is compressed (normaly no or run length coding compression is used).

- RAW - Many digital SLR cameras contain and store all light data received from the camera sensor. These file types are usually quite large. In addition, there are different versions of RAW and certain software may be required to manipulate the image files. Most camera raw formats are based on lossless JPEG.

With **lossy compression** or irrelevance reduction, the original data can usually no longer be exactly recovered from the compressed data, that is, part of the information is lost; the algorithms try to omit only "unimportant" information as far as possible. Such kind of methods are often used for image or video compression and audio data compression. Some examples for lossy compression are JPEG and JPEG 2000 but they can also be lossless.

In the JPEG process, the image is transformed with the help of a system of basic functions. With JPEG this is the discrete cosine transformation, while JPEG 2000 is using the wavelet transformation. In this way, another equivalent representation of the image is obtained, which consists of the coefficients for these basic functions. For JPEG these are the amplitudes of the socalled spatial frequencies, whereas for JPEG 2000 a kind of mean values over 4, 16, 64, a.s.o. pixels are used. These coefficients are changed by the quantization. Small coefficients disappear completely and the larger ones are set to the next best value. So they can be represented with fewer bits.

The biggest visual problems with excessive JPEG compression are compression artifacts caused by the formation of so-called block artifacts, which arise when the image is divided into small blocks, as well as ringing, a consequence of the adverse behavior of the discrete cosine transformation with hard color or brightness transitions.

Another lossy method is fractal image compression (TI 1997). It is based on the knowledge of chaos theory that almost every picture shows self-similarity. Here, groups of pixels of a certain size (e.g. 8×8 points) similar to groups of the next smaller size (4×4 points) are searched for in the same image. Instead of the actual groups, only the position of the similar reference groups which have less image data are then stored. This procedure essentially corresponds to a codebook procedure, with the difference that the codebook does not have to be saved separately, but is available in the same image. Since the search for similar groups can be quite time-consuming, neural networks are used here. However, the decoding is no more complex than with conventional methods.

Data compression occurs in most long-distance transmissions of digital data today. It helps to save resources when transferring or storing data by transforming it into a form that, depending on the application, is as minimal as possible. Only data that is redundant in some form can be compressed without loss. If there is no redundancy, for example in the case of completely random data, lossless compression is in principle impossible because of the Kolmogorov complexity. On the other hand, lossy compression is always possible: An algorithm arranges the data according to how important they are and then discards the "unimportant" ones. In the list of how important which components are, more and more can be discarded by shifting the "keep threshold" accordingly.

In the case of data compression, computational effort is required on both the sender and the receiver side in order to compress or restore the data. However, the computational effort is very different for different compression methods. Deflate and the Lempel-Ziv-Oberhumer-algorithm (LZO), for example, are very fast in compression and decompression, while the Lempel-Ziv-Markow-algorithm (LZMA), for example, achieves particularly extensive compression - and thus the smallest possible amount of data - with great effort, while compressed data can be converted back to its original form very quickly. So depending on the use case this will lead to a different approach of the used compression method.
Compression methods are therefore either optimized for data throughput, energy require-

ments or data reduction, and the aim of compression is therefore not always as compact as possible. The difference becomes clear in these examples:

- If video or sound recordings are broadcast live, compression and restoration must be carried out as quickly as possible. Loss of quality is justifiable if the maximum (possible) transmission rate is adhered to. This applies, for example, to telephone calls, where the conversation partner can often still be understood even if the sound quality is poor.
- If a single file is downloaded by countless users, a slow but very powerful compression algorithm is worthwhile. The reduced bandwidth during transmission easily makes up for the time required for compression.
- When backing up and archiving data, an algorithm must be used that may also be used in the distant future. In this case, only popular, tried and tested algorithms come into question, which sometimes do not have the best compression rates.
- The type of data is also relevant for the selection of the compression method. For example, the two compression programs gzip and bzip2 used on Unix-like operating systems have the properties that gzip only compresses blocks of 32,000 bytes, while bzip2 has a block size of 900,000 bytes. Redundant data is only compressed within these blocks.

Sometimes the data is transformed into a different representation before compression. This enables some methods to then compress the data more efficiently. This preprocessing step is called precoding.

4.2.1. Limits of compressibility

Overview and basic concepts of compression limits can be found in (Phamdo N. 2000).

Lossy compression

As described above, lossy compression is always possible - the threshold for what is considered "redundant" can be increased until only 1 bit remains. The boundaries are fluid and are determined by the use case.
With lossy image compression, details are increasingly lost or let's say become blurred, ultimately everything "blurs" into a surface with a uniform color or gray tone. An audio recording usually becomes dull and indistinct, after the greatest possible compression with most algorithms it would only have a simple sine tone. The so called compression artifacts are signal interference caused by lossy compression.

Lossless compression

In the case of lossless compression, much narrower limits apply, since it must be ensured that the compressed file can be transformed back into the original file. Basically the

lossless compression is using the redundancy principle. Due to the information content, it specifies a minimum number of bits that are required to encode a symbol. This is described by the entropy of the content. Lossless compression methods try to encode messages in such a way that they approximate their entropy as closely as possible.
The Kolmogorow Complexity deals with the smallest possible "instruction set" that is necessary to restore the original data from the compressed data. For example, the number "8000000000000000000000000000000000000000" can be compressed very easily: "Write 8 and then 40 zeros", which is a compression from 41 to 29 characters. Any number of digits after the decimal point of the circle number π can also be compressed with its calculation rule - whereby the compression algorithm would then have to recognize that it is π.

It should be noted that the recovery algorithm would also have to be added to the file size, since any compressed file without such an algorithm is worthless. It is also possible to add only some short parameters or notations to the file from which the ground station can derive and choose the appropriate algorithm to decompress the file. Of course the ground station has to have the algorithm available on her own because it will not be transmitted together with the file containing the compressed data.
For example the above number could also be compressed with "8e40", whereby the reader must then be aware of the method of restoration, namely the power notation. However, if a character string has no recognizable structure, then compression is not possible - the instructions should contain the unchanged original data.

Another reason for the incompressibility of some data is the so-called dovecote principle: If there are fewer nesting places for pigeons than there are pigeons in the dovecote, two or more pigeons must inevitably share a nesting place. One of 2^n possible pieces of information can be stored on an n-bit storage space, and consequently only one of half as much possible information can be stored on a storage space that is one bit smaller: 16 bits $\rightarrow 2^{16} = 65536$ possible information, 15 bits $\rightarrow 2^{15} = 32768$ possible information. Assuming that every possible file could be reduced by one bit, this would mean, according to the dovecote principle, that each storage space would have to contain two different compressed files at the same time. However, since there must be a reversible, unambiguous assignment between compressed and uncompressed files in lossless compression, this is forbidden.
If the dovecote principle did not apply, and if there was an algorithm that could compress any file by at least one bit, this could be applied recursively to the compressed file - any information could be reduced to 0 bits. In practice, data that has already been compressed can only be compressed again if a not 100% efficient algorithm was used in the previous run, which has not yet completely removed the redundancy (e.g. a very large file full of zeros is displayed twice with gzip compressed).

The conclusion from these two facts is that purely random data is most likely uncompressible since it is mostly unstructured and that much, but not all, of the data can be compressed.

4.3. Theoretical background

There are a lot of mathematical and information-theoretical backgrounds to deal with when we are talking about image-compression. A very good overview can be found in (Ottensamer, R. 2009).

Signal to noise ratio, the color of noise, statistical methods like mean values, standard deviation and variance, several distribution curves like Normal distribution, Poisson distribution or Laplace distribution and also terms like entropy and covariance & correlation will come along when we are dealing with compression. The central point is of what components the scientific data consists and what are the conclusions for compression, storage and transmission of the data.

Because there is a lot of literature about all these things I just want to give a short glimpse on the most important one used related to this master's thesis.

Entropy

In information theory the entropy is a metric to determine the information content of a sample. Entropy is a measure of the mean information content per character of a source that represents a system or an information sequence. In information theory, information is also referred to as a measure for eliminated uncertainty. In general, the more characters that are received from a source, the more information is obtained and, at the same time, the uncertainty about what could have been sent decreases.

The theory is based on the papers Shannon wrote already in 1948 (Shannon 1948).
Entropy is marked with the capital letter H and the maximal entropy with H_{max}. For a source of discrete random variables the entropy of a certain character x is defined as the expected value of the information content and summing it up gives the entropy of the source over all.

$$H(X) = -\sum_{x \in X} p(x) \ \ ld\, p(x) \tag{4.1}$$

$p(x)$ is the probability of x and $ld\, p(x)$ is the logarithmus dualis of the probability [1].

The definition of information content can be justified as follows:
If an event that has a certain probability of occurrence actually occurs, then a certain event is selected from a hypothetical set of equally probable, independent events $(1/p(x))$. In order to be able to differentiate between these events, one needs $-ld(p(x))$ binary bits. If the actual information content is now weighted with its $p(x)$, the mean information content of the sign is obtained. The smaller the probability of a character appearing, the higher its information.

[1] ld(x) = ln(x)/ln(2)

In general, the more characters are received from a source, the more information is obtained and, at the same time, the uncertainty about what could have been sent decreases. In lossless compression, the entropy of the noise defines the data rate (Ottensamer R. 2018).

A normalized measure for the entropy of any discrete distribution can be produced if the maximum possible entropy H_{max} is used for normalization, which is achieved with uniform distribution of $p(x)$. H/H_{max} as normalized entropy can have the value of 1 in maximum. For a binary distribution $H_{max} = 1$.

Shannon's original intention was to use entropy as the measure of the required bandwidth of a transmission channel.

Signal and noise

Normally a data set carries the signal that we want to detect together with noise. Noise is the disturbing factor in transmitting and signal detection. So the signal to noise ratio is one of the important issues to deal with and also how it is possible to characterize the noise (Russ 2006).
There are a lot of sources for noise in the information channel such as photon noise or quantum noise caused by the reason that photons are sent out as single events with unpredictable time intervals in between and readout noise of the detector bringing an uncertainty into the signal. It is only possible to use the information of a signal if we know how uncertain it is. Photon noise from the sources will be the second largest contributor to the data frame entropy and this will reduce the compression rate. We can assume a Gaussian distribution for the CCD read noise from the analog chain which is a large noise contributor to a single frame.

Two properties of noise which can be used to characterize it, are:
The Probability Density Function (histogram, PDF) and the Power Spectral Density (color, PSD).
The Probability Density Function gives the distribution of the probability of the noise-amplitude values. Common noise probability density functions we might encounter include Gaussian, Poisson, and negative binomial distributions.
The Power Spectral Density describes the distribution of power into frequency components composing that signal. It has dimension of power over frequency.

Noise can be characterized by its color:

- White noise means that noise is spread over the whole spectrum or in other words it is a signal with (nearly) constant spectral density.

- Pink noise shows a power spectral density inversely proportional to the frequency $(1/\nu)$.

- Purple noise shows a power spectral density which increases with the square of the frequency (ν^2).
- Brown or Red noise can be produced by integrating white noise and is that kind of noise produced by Brownian motion.

Technical issues: Quantization and Sampling

Another problem is caused by the technical frame conditions of signal processing. A normal analog signal (let's speak of the naturaly signal) has to be quantized in amplitude and sampled over time to be able to be processed. This is adding additional uncertainty (or "noise") to the data set. Just remember things like 16-bit quantization of signal amplitudes or the Nyquist criterion for sampling a signal over time.

Statistical properties

Signals can be characterized by their statistical properties like mean value μ, median value, standard deviation σ or variance. What we have to do is to transform the signal or dataset into a form which makes it possible to describe it with the help of the statistical properties.

In the first step the dataset is transformed into its probability density function (PDF) which means that for example when we process an image every appearance of a single pixel-value is counted and the amplitude is plotted depending on its number-count.
In the second step the statistical properties of the PDF are derived which are a measure for the signal over all. Also the PDF can be compared with several normal distributions to get the best fit as described in the next item.

Distributions

If we are looking at a signal it is also possible to characterize it by the distribution of the frequencies in the signal. For example photon noise follows a Poisson probability distribution and readout noise follows a normal distribution.

That means in particular that it is possible to transform the signal from the time-space into the frequency-space and find the best fitting with a distribution function like Gaussian or other. The same can be done with every sample of data, transforming the data into the probability distribution, which means to sort the data by its probability of appearance. So we come to the probability density function and can compare it with existing distribution functions to find the best fit.

Covariance & Correlation

The covariance of two variables X and Y or set of data is a measure of their inter-dependence.
The covariance is positive if there is a monotonic relationship between X and Y, i.e. that

is, high (resp. low) values of X go hand in hand with high (resp. low) values of Y.
The covariance, on the other hand, is negative if there is an opposing monotonic relationship between X and Y, which means high values of one random variable are associated with low values of the other random variable and vice versa.
If the result is zero, there is no monotonic relationship between X and Y. However, non-monotonous relationships are possible.

The covariance indicates the direction of a relationship between two random variables, but no statement is made about the strength of the relationship. This is due to the linearity of the covariance. In order to make a relationship comparable, the covariance must be standardized. The most common standardization - by means of the standard deviation - leads to the correlation coefficient.

Depending on whether the linear relationship between simultaneous measured values of two different characteristics or the one between measured values of a single characteristic that differ over time is considered, one speaks of either cross-correlation or auto-correlation. A correlation describes a relationship between two or more characteristics, states or functions. The relationship does not have to be a causal relationship. Some elements of a system do not influence each other but there is a stochastic relationship between them that is influenced by chance.

The correlation is particularly important when processing one-dimensional or multidimensional data, e.g. pictures. In image processing, however, the time factor is simply replaced by a position variable. The image is interpreted as a signal sequence about the location. In contrast to time functions, images do not have a time base, but pixels, the so-called spatial frequencies.
The spatial frequencies are the resolution of the image. When correlating two-dimensional images, two position variables instead of one are to be used. During image processing, it can then be determined, for example by means of auto-correlation, whether or where a certain object is located in an image. This means that object recognition is possible.

In contrast to the correlation of one-dimensional signal sequences over time, the correlation of two-dimensional signal sequences requires a disproportionately higher time calculation effort.

If two samples are correlated it means that they are not independent from each other which gives us the possibility to predict one sample from the other one. If samples are decorrelated, they will have a lower entropy.

Decorrelation

Decorrelating a signal means that it will have a lower entropy afterwards. How does this work? Decorrelation removes redundant, irrelevant and unimportant image details from the image data. It is not a compression but a transformation of the image to reduce the image data. An example is the wavelet-transformation. A signal contains the interesting information and additional noise which gives no additional information and is of no use in scientific data. So decorrelation opens the possibility to get rid of the unnecessary noise before the signal is compressed and will increase the compression rate over all.

Let us just deal with a question: What is a signal with a maximum information value? Just to say it simple: It is a signal where the value of the next sample is unpredictible from the samples already collected. And as a matter of fact white noise is a signal where you can't predict the value of the next sample taken. Therefore a good decorrelation is some kind of "whitening" the signal.

Decorrelation is normally done by linear transformation. Decorrelation is the central point for compression following normally a three step approach of:

Decorrelation - Quantisation - Encoding.

There are a lot of transformation forms in use like FFT, DFT (both are Fourier Transformation), Discrete Cosine Transformation (DCT) using sine- and cosine-functions as new bases. One of the most popular transformation forms is the wavelet transformation using wavelets as a new base (Meffert & Hochmuth 2018). In lossy compression such kind of decorrelation is always necessary.

In lossless compression some kind of prediction would be good to model the signal. Linear Predictive Coding (LPC) is an easy way for decorrelation which means just estimating signal values from linear combinations of previous samples or in case of images, samples of the spatial nearest pixels. So it predicts the value of a current pixel by its neighbours and encodes the error to this value, named the corrector.

Encoding

Coding or encoding is a process to transform a certain amount of data into another form which makes it more feasible to store or transmit so less data space is necessary than before the coding. Of course the coding should be done in a way, that the original data can be reproduced in lossless form or in a lossy form where only redundant or unimportant data shall be lost.

One of the most popular coding forms is the Morse code. It transforms the characters (which normally need 8 bit to be stored when using the ASCII notation) into much simpler forms of dots and dashes whereas characters which often occur are represented by a shorter dot/dash-combination. In the digital world the dots and dashes can be represented by bits in state 0 or 1. So instead of using 8 bits for a character this can be reduced to 5 for seldom characters and even 1 (!) for often occurring characters which will lead to a compression rate of about 3. The coding here is 100 % reversible (lossless) and this is some kind of entropy coding.

Entropy coding is a method of lossless data compression that assigns a sequence of bits of different lengths to each individual character of a text. Typical representatives are the Huffman coding and the Arithmetic coding.
So coding needs a certain transformation rule.

In the next chapter a few coding forms are described for example.

4.4. Some compression methods

Information theory and in particular algorithmic information theory for lossless compression and rate–distortion theory for lossy compression are the theoretical basis for compression. Beside those also statistical methods like probability distribution and coding theory are a necessary background to deal with compression.

For image compression a wide area of compression methods exist:
It started with Shannon-Fano coding, a form of entropy coding, in the 1940s, the Huffman coding in the 1950s, Transform coding in the 1960s and Fast Fourier Transformation (FFT) and Hadamard Transformation in the late 1960s.

JPEG of the Joint Photographic Experts Group started in the 1990s and is still one of the most popular lossy compression methods in image processing. DEFLATE, a lossless compression algorithm form of the 1990s, is used for example in the Portable Network Graphics (PNG) format.
The use of wavelets in image compression came up at the beginning of the 2000s, for example JPEG 2000 is based on wavelet compression.

Similar methods are also existing for audio and video but I don't want to go into details about those areas because for SMILE we just have to deal with image processing.

Compression methods consist of two parts: A model and a coding. The task of the model is to capture the characteristics of the original data. This can be limited to the frequency of bytes, but it can also be the occurrence of byte sequences. The coding ensures the space-saving encryption of the data. Here are some examples derived from (zur Nedden, H. 1992) based on (Bauernöppel F. 1991):

Run length coding

Basic information is derived from (zur Nedden, H. 1992) and (Kriegl A. 2003). Run-length encoding is a very simple process - not particularly efficient, but very fast. Characters that appear several times in a row are only encoded once together with a counter. Since the counter also takes up space, characters that appear twice or three times remain uncoded. There are two ways to encode the counter: the 7-bit run-length encoding and the standard run-length encoding. If the characters are bytes from the range 00h-7FH, this can be used to encode the counter by encoding bytes that occur multiple times with the counter with the seventh bit set followed by the actual counter. This method is particularly suitable for text files in which bytes with a set seventh bit rarely occur. Such special characters are to be coded with a prefixed counter 81H.

The standard run length coding encodes multiple occurring bytes in three bytes: the original byte, 90H, counter. Since the maximum count can be 255, larger occurrences must be subdivided. Furthermore, a special coding is necessary for 90H which means a counter follows. 90H is therefore coded as 90 00H.

Run length coding is a simple and useful method for images. Pixels that occur frequently one after the other are replaced by a pixel value and a counter. So this is very useful for big areas without internal structure.

Huffman coding

Basic information is derived from (zur Nedden, H. 1992) and (Lang H.W. 1997). The Huffman method encodes frequently occurring characters with fewer bits and rare characters with more bits. Original data containing only the four letters A, B, C and D may serve as an example.
Two bits are sufficient for four different characters: 00b, 01b, 10b and 11b. If A occurs ten times, B five times, C three times and D twice, the data occupy 40 bits in continuous two-bit encryption. If you code the A as 1b, the B as 01b, the C as 001b and the D as 000b, then 35 bits are sufficient.

To determine the respective bit coding, the character frequencies must be determined. There are three ways to do this:

- Static: The frequencies come from previously defined tables.
- Dynamic: The data is read once to determine real frequencies.
- Adaptive: At the beginning there are fixed start specifications for the frequencies (for example all characters appear the same number of times), which are adapted to the current code during the coding.

Once the frequencies of the characters that occur have been determined, the bit combinations can be distributed in suitable lengths, with rare characters being given a particularly large number of bits. At the beginning there are the two rarest; they get a 0b or 1b. Now you summarize both and replace them in the list of frequencies by their sum. Then the two rarest are searched again, one by 0b, the other extended by 1b and the two combined. This procedure ends when there is only one character left. If a combination of characters has a 0b or 1b appended to their code, this means that the codes of all associated characters are extended by this bit.
The determination of the codes from the frequencies of the characters leads to a tree (Huffman tree). This tree is used for coding and decoding according to Huffman:
The coding goes from the character to the root and thus results in the binary code that is supposed to stand for the character. The length of the code is the length of the path covered.

Decoding starts at the root by reading in a bit. Depending on the content, the tree branches to one (1b) or the other (0b) page, then reads the next bit and follows again according to the next branch. At the end on a leaf of the tree is the symbol you are looking for.

If the frequencies of the characters are fixed beforehand (static), the coding is only optimal if the assumed frequencies match exactly. There is no need to determine and transmit frequency distributions beforehand.
For dynamic coding, all characters must first be counted in order to determine their frequencies and use this to build the appropriate tree for the coding. The actual coding does not take place until the second pass. Furthermore, the determined frequencies or the tree created from them must be passed on to the decoder.
If starting the coding with a starting value for the frequencies and adapting it with each processed character, the coding is not optimal at the beginning, but gets better and better over time. The advantage of this method over the dynamic one is that the decoder also determines the frequencies and can build the tree itself. Consequently, with this type of Huffman coding, there is no need to pre-count the characters or to transmit the tree.
In some cases, the adaptive Huffman coding can even be better than the dynamic one. If the characteristics of the data change, for example because program and text parts alternate, the adapting Huffman adapts to the machine code and text structure.

Squeeze and Unsqueeze

Squeeze uses dynamic Huffman coding for compression (zur Nedden, H. 1992). The model of squeeze is the frequency of bytes, decorrelated while independent. As a result, the compression rate is only moderate: The assumption that all bytes occur stochastically independently of one another is mostly incorrect; for example, a 'q' is almost always followed by a 'u'.

The decompression can either read in the tree supplied by Squeeze and evaluate the corresponding value from the tree for each bit read or build the corresponding machine commands from the tree which is certainly faster. A piece of code is generated for each node of the tree, which reads in the next bit and branches accordingly, i.e. jumps to the piece of code which corresponds to the subtree or outputs the character found. This method is very quick because the tree only needs to be interpreted once.

The squeeze model works exclusively on the basis of byte frequencies. But program sources already contain many frequently repeated strings. In the case of compression based on Lempel-Ziv, the bit length of the code is fixed or at least predictable, but a large number of characters can be hidden behind such a code. There are some compression programs that use algorithms based on Lempel-Ziv.

Lempel-Ziv-Welch

Basic information is derived from (zur Nedden, H. 1992) and (Kriegl A. 2003). The Lempel-Ziv-Welch (LZW) algorithm is now implemented in two versions in compression programs: The simpler variant uses a fixed code length and no code recycling. The LZW with variable code length and reuse of codes once they have been assigned when they are not needed is more efficient. The length of the code depends on the size of the memory. Originally this was 4096 entries long, so 12-bit codes were sufficient. Code lengths of up to 16 bits are also used in newer programs.

The memory is a table of already recognized character strings. The original data is read as long as the characters are contained in the table as a character string. Reading is interrupted when a new character string occurs. This new string is one character longer than one that has already been saved and, is learned and the new character is appended to the existing character string. Reading then continues, with the last character read becoming the starting point for the next character string to be learned.

Decompression works the other way around: After a code has been decoded, it appends the first character of the character string that has just been decoded to the previous one.

At the beginning of compression and decompression the table contains the entire character set as character strings of length one. For example, if there is now the string "ABCDEFGH" to be learned, the following sequences are also "AB", "ABC", "ABCD" and so

on in the table. The entries in the table each consist of a pointer called Pred (short for Predecessor = predecessor) and a character called a suffix. In the example, the entry for the string "ABCD" contains the character "D" as a suffix and a pointer to the entry of the character string "ABC" as Pred.

Beside pointers to other table entries, there are some special Pred values: "NoPred" denotes entries that form the beginning of a character string (No Pred = no predecessor). After initialization, the table only contains strings of length one as (NoPred, character). "ImPred" blocks entries against abuse (Impossible Pred = Impossible Pred) because the associated codes have a special meaning. Finally "Free" indicates that the entry in the table is free. Free table spaces are allocated by hashing, i.e. by calculating with (pred, suffix) so that the search for a specific entry is faster.
This version of the LZW always uses 12-bit table pointers (4k table). Larger tables require a correspondingly increased code length. The table entries are made depending on the content (pred, suffix). The number of the table entry consists of the middle 12 bits of the square of the sum of Pred and suffix. If the table entry calculated in this way is occupied, an alternative entry must be used. The current entry receives a corresponding link. An additional pointer field, called a link, is used for each table entry.

The search for the alternative entry is carried out as follows: First the table entry to which the link points is examined. If it does exist, the search continues with this very one. If that didn't work either, add 101, a prime number, to the current table position and look for the next free table entry forward from there; after the last place in the table, the process continues with the first one. If it is found, its position is entered in Link and the new one is then assigned. The link values are preset as empty during the initialization.
The reason for determining the table entry for (Pred, Suffix): The search for a specific (Pred, Suffix) in the table is considerably faster than browsing through it sequentially. That is why the alternative entries are also noted; after all, the table is searched continuously during compression. The search works in such a way that the appropriate table entry is calculated from (Pred, Suffix). Then you only have to look along the link chain to determine whether the entry you are looking for is in the table or not. If the table is full, all further character strings are coded on the basis of the previous entries.

Code recycling

The extended LZW generally compresses better than the simple one because it has two special features:

- The code length is nine to 12 bits (or more if the memory is larger), as the table entries are not assigned somehow (as with a hashing algorithm), but the table is filled from the bottom. Since the first 256 places for the possible bytes are occupied with Pred = NoPred during the initialization, you need at least nine bits for a free table entry. If all of the nine-bit addressable spaces are occupied, ten bits are used, and so on.

- Table entries can become available if there is a character string that occurs only once during learning.

The variable code length can only be used if the table is filled up from below. On the other hand, the calculation of the table positions is extremely useful in order to accelerate the check if a certain pred or suffix is already in the table. This is why this LZW version uses another pointer array, called a hash. If a new entry is to be made in the table, a free space is determined in the hash array and entered there where the new entry is to be made in the table. The new table entry is simply the next free one. As part of the initialization, the pointer is preassigned the value empty. The hash array has more entries than the table to reduce the possibility of hashing collisions. In the event of a collision, the prime number 5003 is added to the current position until a free space is found.

The search for a pred or suffix works accordingly: The associated entry in hash is calculated. There the address of the one searched pred or suffix in the table can be find. Compression and decompression keep track of the number of entries in the table and always adjust the code length if an additional bit is required for addressing. In this way it is possible to start with short code lengths without having to forego calculating the address of table entries and thus being able to find them more quickly.

If a string that has been learned is never used again, it can be removed from the table and the space that has been freed up can be used for another string. The decompression procedure of the extended LZW differs from the simple one when the memory table is full. Instead of just continuing with the knowledge it has acquired, it tries to forget what is unnecessary and thus learn new things. For the decision as to whether table entries can be forgotten, they are given 'referenced´ tags and are therefore protected against deletion. During initialization, the entries are also protected with Pred = NoPred and Pred = ImPred.

When crunching, the data is compressed using the LZW algorithm, whereby the character strings are formed from bytes - depending on the version of the program with a fixed code length or with a variable code length and code reuse. Some versions of Crunch have reserved codes for special functions. To prevent problems, these codes are entered when the table is initialized with Pred = ImPred, i.e. as blocked.

Lempel-Ziv-Storer-Szymanski

Basic information is derived from (zur Nedden, H. 1992). Lempel-Ziv-Storer-Szymanski (LZSS) is a modification of the LZW: Instead of the table, it uses a ring buffer as memory in which the last characters read from the original file are stored. Before a string is compressed, a check is made to see whether it is in the buffer. If so, its length and address are transferred relative to the current position in the ring buffer. Although this algorithm seems very simple the compression rate is amazing.

When comparing the knowledge acquisition of LZW and LZSS, the following difference is noticeable: LZW builds a table from the character strings that are at the beginning of the file - afterwards this table with the knowledge is quite static. LZSS, on the other hand, searches for the character strings in the buffer that contains the last few KB of read data. If the character strings in the file change, LZSS adapts itself automatically. This is reminiscent of the difference between the dynamic and the adaptive Huffman.

There are three constants in the LZSS algorithm: The size of the ring buffer, the maximum length of a character string and the minimum length of a character string. These three constants are called buffer size, max length and min length in the following. The value MinLength-1 is generally referred to as the threshold.

Compression programs usually reserve a memory area for the buffer that is MaxLength-1 characters longer than the buffer size. The additional characters at the back of the buffer always have the same content as the first MaxLength-1 characters. They are here a second time so that the check whether a character string is in the buffer can be carried out more quickly at the end of the buffer.

The compression also generates either the original data or a length specification, followed by an address specification. So that the decompressor can understand the information received, it must be able to distinguish between a normal character and a length specification. To do this, the number of characters is simply expanded to include characters for the length specifications. As with the standard run length coding, a character for "length specification follows" could also be used. But this is actually nothing more than an expansion of the set of characters.

Usually the characters processed by the LZSS algorithm are normal bytes containing values from 00H to FFH. The specifications of the length are then represented by the next values 100H, 101H and so on. To get the corresponding character code from a length specification, add 100H minus MinLength. At first glance, this ninth bit takes some getting used to. But for a coding like Huffman the type of characters is irrelevant, the most important thing is that they can be counted.

Lempel-Ziv-Huffman

Basic information is derived from (zur Nedden, H. 1992) and (Buchanan 1999). The Lempel-Ziv-Huffman (LZH) compression codes the LZSS model with the adaptive Huffman with a small addition in the way of coding the address information. The decisive factor for Huffman is that this encoding can encode any set of characters efficiently. For byte sequences of lengths between three and 60, 256 + 58 codes (256 bytes and 58 length specifications) are required, because with the adaptive Huffman the frequency table (and thus the tree) is initialized with "all possible characters appear the same number of times" before compression begins . The odd number of 256 + 58 codes does not interfere.

An older LZH implementation uses a 4 kB long ring buffer so that the address information is 12 bits long. These are also coded: The upper 6 bits of the address are coded in three to eight bits using a table. Small values are dedicated to a few bits. Since the address details are addresses relative to the current buffer position, newer character strings are preferred. The lower 6 bits remain unchanged. The address information is therefore nine to 14 bits long.

When decoding an address, eight bits are read first. In any case, these include the code for the upper 6 bits of the address specification. This and the length of the code with which the upper 6 bits were coded are determined using a table. If this length is less than eight, there are already eight minus length bits of the lower 6 bits of the address information in the lower bits of the byte just read. Since a total of length plus 6 bits have been output and 8 bits have been read so far, the length minus 2 bits must still be read in order to obtain all the information.

Arithmetic coding

Basic information are derived from (zur Nedden, H. 1992) and (Gad A. 2022). Frequencies can be represented with real numbers between 0 and 1; where 0 means never and 1 always; For the more convenient percentages, the frequencies have to be multiplied by 100. An interval consists of the numbers that lie between two edge points (called interval boundaries), none of which, one or both of which can belong to the interval. [a, b) denotes the interval of numbers that lie between a and b, where a belongs to it (square brackets), but b does not (round brackets). [0, 1) therefore denotes all numbers that are greater than or equal to zero and less than one. In the arithmetic coding, the characters are to be represented as intervals of the form [a, b), the lengths of which correspond to the frequencies and are placed one after the other, starting with zero. Since the sum of the frequencies is 1, this chain of intervals results in the interval [0, 1). For codes that are as short as possible, one naturally takes the number with the fewest digits that lies in the interval of the character. Since all occurring numbers lie in the interval [0, 1), they are of the form zero-point-anything; the coding is therefore limited to the decimal places.

If you take the example with the letters A to D and the frequencies 0.5, 0.25, 0.15 or 0.1 for the individual characters, the intervals [0, 0.5), [0.5, 0.75), [0.75, 0.9) and [0.9, 1) are built from this and determine the appropriate binary codes. This results in the four codes 0b, 1b, 11b and 1111b corresponding to the decimal fractions 0.0, 0.5, 0.75 and 0.9375.

But this is only the first step in coding. In arithmetic coding, character strings and not individual characters are converted into a code. And this is done as follows: The first character in a sequence provides an interval. This interval, like the interval [0, 1) at the beginning, is divisible according to the frequency of the characters. Then the second character again leads to an interval that lies in the first, and so on. For example, ‘BBBB´ is converted into the number 0.10101010b which leads to the code 10101010b.
Since the interval nesting cannot go on indefinitely, the end of a coded character string

must be recognizable. An otherwise unused symbol is used for this purpose. Usually byte sequences (characters 00H to FFH) have to be coded. So you just take another character as the end identifier - of course it only occurs once in a string - and calculate the 257 required frequencies.

As with Huffman, the frequency of individual characters can be determined in three ways: static, dynamic and adaptive. A few more notes about the implementation:

- When dividing the intervals, it must be ensured that no interval shrinks to length zero due to the limited computational accuracy. To do this, the data is not encoded in a character string with a single code, but divided into several character strings. The WHILE query in the coding algorithm must be modified accordingly and both the coding and decoding algorithms must be called for as long as data is arriving.
- As soon as the first digits of the upper and lower limit of the current interval are identical, these digits can already be output and then discarded for coding. You can't change anymore. This eliminates the need to keep the complete code in memory. In addition, discarding the digits displayed reduces the length of the numbers that have to be worked with.
- When decoding the code does not have to be read in all at once. As soon as a sufficient number of digits has been read in to clearly identify an interval, the division can already begin. Similar to coding, leading digits can be discarded as soon as they are identical in the upper and lower limits of the current interval.
- Similar to LZH compression, adaptive arithmetic coding can be used for implementation in order not to be forced to read the original data twice and to transmit the frequency table.

Arithmetic versus Huffman

Basic information are derived from (zur Nedden, H. 1992). The arithmetic and the Huffman method pursue the same goal: to provide the shortest possible codes for a given frequency of an arbitrary set of characters - in the adaptive case this happens in every step. The difference is that Huffman gives codes per character, while arithmetic coding produces codes for character strings. It is mathematically provable that the arithmetic coding is more efficient than Huffman.
The compression algorithms described here are not intended for specific, but for any data. Depending on what the data really looks like, one or the other method may be better. As a rule of thumb, however, it can be said that the average compression rate from Squeeze to LZW, LZH and LZAri is getting better and better. There are also compression programs such as PKARC that first analyze the data and then decide which algorithm to use.

Another situation for the development of special algorithms arises when the speed is more important than the compression rate. The algorithms described here are generally

quite time consuming, which is not always appropriate. For example, when storing and transmitting moving images, the most important thing is the speed of compression and decompression. Sometimes even without the exact restoration of the data.

Golomb coding

Basic information are derived from (Bull D. R. & Zhang F. 2018). The Golomb code is an entropy coding for all non-negative integers which, in contrast to other source coding codes, can only represent a finite range (e.g. the value range 0-255). It was developed by Solomon W. Golomb in 1966. The code uses a few bits for small numbers and more bits for larger numbers. It can be controlled via a positive, integer parameter. The larger the parameter, the slowlier the number of bits required for representation grows, but the greater the number of minimum bits required for the small numbers.

The Rice Code is a variant of the Golomb Code in which the control parameter is a power of two. This restriction is advantageous since the multiplication or division of 2 can be implemented very efficiently, especially in digital processing. The Rice Code was introduced by Robert F. Rice and J. Plaunt in 1971. It can be implemented very easily with bit shifting and logical bit operations.

The code can be used in the field of lossless data compression if the probabilities of the source data to be coded form (approximately) a geometric distribution. Typical areas of application are, as a sub-method alongside other algorithms, image compression and audio data compression.
If changes in an image are low, the code would be very effective. A compression of an image where most of the pixels are just representing the background makes a lot of sense using Golomb coding.
The Golomb code can be used when numbers of unknown size are to be stored, but the actual field of application is in data compression. If the probabilities of the numbers have a geometric distribution, the Golomb code can be as efficient as the Huffman code, but is more economical with memory, easier to implement and faster to run.

Data differencing

Data differencing uses a source and a target and produces the difference between them (Korn D. G. & Vo Kiem-Phong 2002). On base of source and differences the target can be reproduced. Normally source plus difference gives a smaller amount of data than source and target. Best results occur when one has knowledge of the data being compared and other constraints. For example "diff" is designed for line-oriented text files, particularly source code, and works best for these. Finding similar methods for images or imagettes (small parts of images) can be helpful for image processing too.

4.5. Application at SMILE

The usage of compression functions in SMILE is a derivate and extension of the implementation already used for the CHEOPS mission.

The compression software consists of several parts that can be switched one after the other, whereby each stage fulfills its own task and can still be parameterized.
In figure 4.1 it can be seen that the chain starts with the reading in from the SIB (Single Image Buffer), afterwards the data run through a preprocessing, then lossy and/or lossless compression processes will be performed and finally the results are stored in the CIB (Compression Intermediate Buffer) in the CE-format.
The CE-format is the Compression Entity format produced by the compression function and is described in the file *SdpCompressionEntityStructure.h* of the implementation.

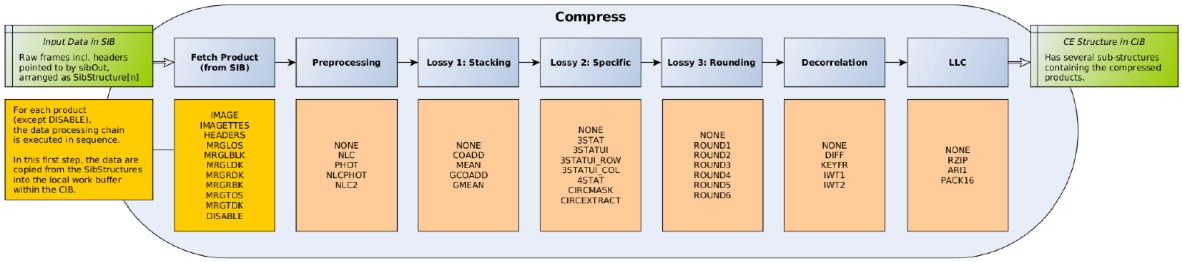


Figure 4.1.: Compression chain (Ottensamer R. 2018)

Each individual partial processing step can be specially parameterized. It is also possible to set all steps to NONE, then the data comes out unprocessed at the back of the chain. A detailed description of all the parameters can be found in (Ottensamer R. 2018), the technical notes for CHEOPS Data Processing (Ref.: CHEOPS-UVIE-INST-TN-001). The SMILE SXI IASW Requirement Specification (Reimers Ch. 2019) contains also a lot of information about the SMILE SXI IASW software.

4.5.1. How to meet SMILE-requirements

When thinking about solutions in terms of SMILE-requirements it would be necessary to know them in detail (see (Branduardi-Raymont et al. 2018), (Smit S.J.A. 2019), (Reimers Ch. 2019) and (Mecina, M. et al. 2020)). On one hand the necessary data has to be stored and transmitted in a way that it is reproduceable at the ground station. On the other hand as less as possible data should be transmitted to secure that the transport channel isn't overloaded.

Full images, event detection, housekeeping

Image data of SMILE-SXI will consist of full images to provide the astronomers at the ground station with complete images for detailed physical investigations of soft X-ray events and EUV and also to open the possibility for calibration of the scientific instruments. But still keep in mind that a complete image unbinned will have about 34 MB of data of each CCD so both CCDs together will produce 68 MB. Normally the CCDs will be operating in the 6x6 binning mode for soft X-ray events which leads to about 1 MB of data per CCD and image. If EUV detection is in place the 24x24 binning mode is used producing 63 KB images per CCD.
For the investigation of soft X-ray events or EUV it is necessary to cleary detect their intensity and spatial distribution. This could be done by a special software on board within the FEE detecting events and sorting them into event detection data packages so only these data packages need to be transmitted and not the whole image. Even when this needs additional computation time it saves compression time on the other hand and relieves the transmitting channel.
A third aspect is the transport of housekeeping data such as temperature or voltage of the instrument itself. Of course this data has to be transmitted in a lossless way and normally will not pass through the compression function.

Compression solutions for full frame images

For the transfers of complete images it would be necessary to compress the data due to the high amount of bytes per image. There are several ways to do it:

- Using lossless compression: It is of course the best way from the scientific point of view because the full image can be 100% reproduced. But it has to be investigated how big the compression rate will be and by which parameters it is influenced. For image-header data or housekeeping, lossless compression will be the only way, otherwise the data might be destroyed.

- Lossy compression: There are several ways for lossy compression of an image and the criteria will be that the necessary data (intensity and spatial distribution) will still be present after compressing and decompressing the image.
  So of course it will not be feasible to use any kind of compression where only the main characteristics of the probability density function are used like statistical properties as mean value and standard deviation and do a best fit with some existing distribution functions and transmit only those parameters. This will destroy the information about intensity and location of the single soft X-ray events or EUV.
  Also no kind of compression or preprocessing can be used where only parts of the image are used like circextract or circmask as it can be used for the CHEOPS mission.

Compression solutions for event detection data

Event detection data consists of a data header and a small image detail around the single event of 5x5 pixels. A data package has a total size of 64 bytes and is described in detail in (Smit S.J.A. 2019). For this kind of data a lossless compression will be used to ensure that the decompression will deliver the original data. An alternative could be just to pass through the data without compression depending on the data-amount and computation time for compression in the satellite. Also data differencing should be taken into account for the 5x5 pixel event imagettes.

4.5.2. SMILE compression variations

To get a clear picture of what kind of compression can be used for SMILE-SXI full frames, several compression variations were performed with test data generated by the data simulator DaSi and the most relevant parameters such as compression rate, computation time for compression and decompression. Also the residuals for each kind of compression comparing the original image with the one after the compression/decompression chain were investigated to find out if the processed images are still carrying the necessary scientific data or the disturbancies by the compression are too big for further image analysis.

Looking at the possibilities used in the CHEOPS mission there are several combinations of parametrization of the compression chain that also might be interesting and can be used in the SMILE compression (Ottensamer R. 2018).
The first one is a compression mode for science headers as described in table 4.1 which of course shall be lossless. In this example the parametrization says that the product to be processed is a header, no preprocessing or lossy compression is done and only a lossless RZIP-compression after a keyframe-decorrelation is done:

Product	**Fetch**	**Prepro-cessing**	**Lossy1: Stacking**	**Lossy2: Specific**	**Lossy 3: Rounding**	**Decorre-lation**	**LLC**
Headers	HEADERS	NONE	NONE	NONE	NONE	KEYFR	RZIP

Table 4.1.: Compression parameters for science headers

To start a lossless compression of a full frame image the parameters can be set in the following way (see table 4.2):

Product	**Fetch**	**Prepro-cessing**	**Lossy1: Stacking**	**Lossy2: Specific**	**Lossy 3: Rounding**	**Decorre-lation**	**LLC**
Full Image	STACKED	NONE	NONE	NONE	NONE	IWT1	ARI1

Table 4.2.: Lossless compression parameters for a full frame image

This parametrization means that the product full frame is not preprocessed, no lossy compression is done and after a wavelet-decorrelation a lossless compression with arithmetic coding will take place.

For event detection data beside a lossless compression even no compression can be taken into account and the data should only be collected as it is into the CIB for later handover to the board computer. The parametrization in this case would be (see table 4.3):

Product	**Fetch**	**Prepro-cessing**	**Lossy1: Stacking**	**Lossy2: Specific**	**Lossy 3: Rounding**	**Decorre-lation**	**LLC**
EventData	EVENT	NONE	NONE	NONE	NONE	NONE	NONE

Table 4.3.: Parametrization of no compression for event detection data

Another even simpler solution will be a shortcut, that means compression isn't used anyway and only the collection into the GIB takes place. In this case, some adaptations to the collected binary data have to be done to assure that a later decompression is able to distinguish between a raw event detection bytestream and a compressed data set.

4.5.3. Parameter description

A detailed description of the parameters for the various steps of the compression chain can be found in (Ottensamer R. 2018). The definitions of the single parameters for programming are set in the include-file *SdpCeDefinitions.h*.
In the test results later on the parameters used are listed (see chapter 8).

5. Data Simulation

Due to the fact that the real environment of SMILE will be available for integration of the SDP-SW in 2022 earliest and even at that time no real data from X-ray events can be expected, simulations are playing an important role in the development of the SMILE Science Data Processing.

SDP will have to deal with two different types of data sets during the SMILE mission:

- Full frame data which contains complete images of the CCD's, only one of them or both together, in binning modes 6x6 (soft X-ray) or 24x24 (EUV).
- Event detection data carrying the data of a single event in an 64 byte package.

For both types of data, simulations are necessary to test the compression and decompression in a standalone mode.

5.1. Full frame images

DaSi, the DataSimulator, is used for creating full frame images for test purposes as FITS-files. DaSi was developed at the Institute of Astrophysics by Johannes Seelig and very well described in his master's thesis (Seelig, J. 2020). So I only will give here a short overview and a description of some modifications I made for SMILE and how I used it.

5.1.1. FITS Structure

FITS stands for Flexible Image Transport System. It is the standard file format in astronomy able to store images and tables (Pence, W. D. et al. 2010).

FITS images are often represented by a 2-dimensional array of pixels containing the image of a sector of the sky, but FITS images can also contain 1-D arrays (i.e, a spectrum or light curve). Also 3-D arrays (a data cube), or even higher dimensional arrays of data are possible. An image may also have 0 dimensions, which represents an empty array. The supported datatypes for the image arrays are 8, 16, and 32-bit integers, and 32 and 64-bit floating point real numbers. Both signed and unsigned integers are supported.

FITS tables contain rows and columns of data. All the values in a particular column must have the same datatype. A cell is not restricted to a single number, and instead can contain an array or vector of numbers.

FITS tables could be ASCII or binary. ASCII tables store the data values in an ASCII representation whereas binary tables store the data values in binary format. Binary tables are generally more compact and support more features.

The simplest form of a FITS-file consists of a Header Data Unit (HDU) with an image or table. But a single FITS-file may contain multiple images or tables. Each table or image has a HDU. The first HDU in a FITS-file must be an image (but it may be of 0 dimensions). This is called the Primary Array. Additional HDUs in the file may contain either an image or a table. They are called 'extensions'.

Here you can see the header of a FITS-file generated with DaSi. There is a huge amount of library function to handle FITS-files. They are collected in FITSIO, details can be found in (Pence, W. D. et al. 2010).

```
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                   16 / array data type
NAXIS   =                    3 / number of array dimensions
NAXIS1  =                  750
NAXIS2  =                  631
NAXIS3  =                    1
EXTEND  =                    T
BSCALE  =                    1
BZERO   =                32768
DATE-OBS= '2021-07-19T11:44:22' / YYYY-MM-DDThh:mm:ss observation start, UT
EXPTIME =                  0.5 / Exposure Time in Seconds
SETTEMP =                -15.0 / CCD Temperature Setpoint in C
CCDTEMP =                -15.3 / CCD Temperature at start of Exposure Time in C
PIXDIM  = '18 x 18 '           / Native Pixel SIze in microns
XPIXSZ  =                  108 / Pixel Width in microns (after binning)
YPIXSZ  =                  108 / Pixel Height in microns (after binning)
XBINNING=                    6 / Binning factor in width
YBINNING=                    6 / Binning factor in height
READOUTM= '100 kHz '           / Readout Mode of Image
IMGTYPE = 'Light Frame'        / Type of Image
NOISE   = 'FullNoise'          / Type of Noise
IMGFRAME= 'CCD_B   '           / A, B, or A+B
FOCALLEN=                   30 / Focal length of telescope in cm
INSTR   = 'SXI     '           / Soft-Xray Imager Data Sim for SMILE Mission
OBSERVER= 'B. Dorninger'
NOTES   = '        '
END
```

Listing 5.1: FITS file header (example)

5.1.2. Data simulation with DaSi

DaSi, the Data Simulator, uses the simple form of a FITS-file, a HDU with one 2-dimensional image and in addition a form with a 3-dimensional cube. It offers the opportunities to generate Test-FITS including a certain amount of X-ray events and cosmics in the image. DaSi offers two kinds of noise simulations: "FullNoise" spreads noise over the whole image while "LocalNoise" adds noise only to small areas around events and cosmics. For SMILE only "FullNoise"-images are important because the "LocalNoise" isn't a realistic case and will not be taken into account for investigation and testing.

To meet the requirements of SMILE, some modifications were done on the original program.

5.1.3. Modifications of DaSi

The original DaSi has a configuration file config.xml but wasn't using the parameters of this file and has fixed values in the code.
The config.xml is used to configure the created FITS-file. For details see (Seelig, J. 2020).

```
<Sim_MetaInfo>

    <!-- runs -->
    <runs>1</runs>

    <!-- graphics -->
    <show_plot>0</show_plot>
    <save_plot>0</save_plot>
    <save_data>1</save_data>

    <!-- Detector info -->
    <CCD>CCD_A</CCD>
    <detector_x>          4510          </detector_x>
    <detector_y>          7582          </detector_y>
    <bias_level>           150          </bias_level>
    <read_noise>           4.5          </read_noise>
    <dark_current>         0.0          </dark_current>
    <full_noise>           0            </full_noise>
    <!-- dont touch binning_set! -->
    <binning_set>1</binning_set>
    <binning>                6          </binning>

    <single_det_x>        3791          </single_det_x>
    <single_det_y>        4510          </single_det_y>

    <exposure_time>        0.5          </exposure_time>

    <!-- Cosmics -->
    <cosmic_folder>cosmics_data/</cosmic_folder>
    <cosmic_size>            1          </cosmic_size>
    <cosmic_intensity>    5000          </cosmic_intensity>
    <cosmic_rate>           24          </cosmic_rate>
```

```
33
34      <!-- Signal -->
35      <event_amount>        1800          </event_amount>
36      <event_intensity>      100          </event_intensity>
37      <event_variability>     20          </event_variability>
38
39  </Sim_MetaInfo>
```

Listing 5.2: config.xml

The first step was to remove the hard coded values and allow the parametrization of DaSi via config.xml for noise, binning, exposuretime and the values for X-ray events and cosmics. In addition, the parametrization of the CCD via config.xml was introduced.
SMILE also needs two different binning forms, 6x6 (for soft X-ray) and 24x24 (for hard UV) and in addition also unbinned images are necessary. Therefore some additional modifications in DaSi were done to enable all 3 forms. Unbinned mode is marked in config.xml with <binning> = 1.
Just to get more Testdata with one DaSi-run, a loop over the exposuretime-values down to 0 was implemented for optional use.

The DaSi-output filenames were extended in such a way that it is visible what they are containing:
SXI_single_light_FullNoise_CCD_A_b=6_et=0.5_cr=24.0_ea=1800.fits
for example means, that the file contains a single image (no cube), a light (not a calibration) one with FullNoise of CCD-A with binning = 6, exposuretime = 0.5 seconds, containing a cosmic-rate of 24 and 1800 events.

DaSi modifications in detail

The modified DaSi-version is named V1.41 (see README in DaSi-git-repository at *smile@herschel.astro.univie.ac.at:1722/home/smile/OBSW.git* in the branch *berndt*.
In particular the following modifications were made (all marked with "BD" in the code):

- main_prog.py:
  - Comment on “print_time” to avoid printout.
  - Comments on “fullnoise”, “callib”, “repeat”, “savedat”, “ccdnr” in SMILE-simulator to keep config- and procedure-variable values.
  - Change of “exp_t” and “binningN” to config-values in MAIN program.
  - Optional loop over exposure-time values down to 0 to generate more fits in one run.
  - Optional multiple calls of procedure “SMILE_simulator” with all CCD-configurations to generate all types of fits in one run.
- utilities.py:
  - “Observer” in fits-header changed.
  - Extensions in save-filename (binning, exposure_time, cosmic_rate and event_amount).

- core_lib.py:
  - Testprints for COSMICS and EVENTS.
  - Comments on “intense” and “amount” to avoid hard-coded values.
  - Changes to ensure 24 x 24 binning and unbinned mode as well beside originally fixed 6 x 6 binning.
- global_lib.py:
  - Testprints for “config”-values.
  - Introduce configuration of CCD-choise via config.xml.

Some examples

Figure 5.1 shows the image of the above mentioned FITS-file generated with DaSi. The cosmics can be seen very easily as bright spots of different shapes, the 1800 X-ray events are the fainter spots and to the whole image a plain and even distributed flat-field is added and "FullNoise" is applied to the image.

Figure 5.1.: DaSi: Example of a generated FITS-file

DaSi opens the possibility to add flat field background to the images. The distribution of light within the flat field can be customized via a csv-file called *flat_ self.csv*. So to add an evenly distributed flat field all values within the csv-file are set to the same value.

With DaSi it is possible to create a wide range of images which can be used for some investigations about lossless and lossy compression and decompression of the images. Comparing the results with the original images and deriving the achievable compression rates will give a good overview which kind of compression algorithms are feasible for use in SMILE science data processing.
Remark: DaSi is using patterns for cosmics and events and their shape is ajusted to the usage of 6x6 binning. Nevertheless it can also be used for 24x24 binning and unbinned mode and still be close enough to a usable scenario for testing the image compression.

5.2. Event detection data

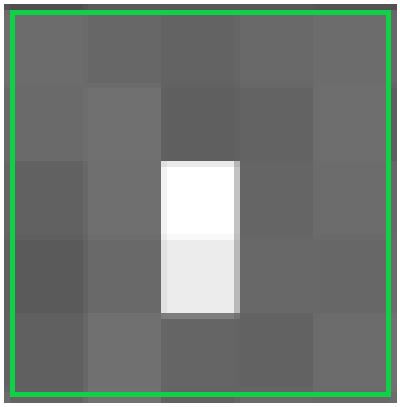
5.2.1. Data structure

Currently (April 2022) there are two different event detection data definitions in place. The first one can be found in (Smit S.J.A. 2019) and describes data as 64 byte long packages containing the 14 byte long header and 50 bytes carrying values of a 5x5 pixel area around the single event. Table 5.1 shows the header-data.

Byte	Header-data
0	Event data logical address
1	Event data protocol-id
2/3	Event data length
4/5	Event data type
6/7	Event data frame counter
8/9	Event data sequence number
10/11	Event data column
12/13	Event data row

Table 5.1.: Event detection data header

Figure 5.2 shows the image detail of a single event on the left. The related numbers of the 5x5 nearest neighbour pixels around P12 which is the “event” pixel are given on the right. The numbers are representing the order of the pixel values in the event detection data package.



P0	P1	P2	P3	P4
P5	P6	P7	P8	P9
P10	P11	P12	P13	P14
P15	P16	P17	P18	P19
P20	P21	P22	P23	P24

Figure 5.2.: Image detail of a single event. Green borders showing the 5x5 pixels provided in the event detection data package.

The second event data description can be found in (Soman M. and Randall G. 2020). Due to the fact that this description is incomplete I used the first one mentioned above for my implementation and tests.

5.2.2. Data simulation with EDSim

For the simulation of event detection data in standalone mode also files are used. In the simplest form they can be created using a hex-editor. Because this needs too much time to create a representative set of testdata, I implemented a short Event Detection Simulator (EDSim) which uses DaSi-FITS-files as an input to create appropriate event detection data files. Each file contains one or more event data packages of the form described above depending on the events found in the FITS-file.

EDSim reads the FITS-file, detects events using thresholds relative to the meanvalue of the image and collects the event detection data packages in an outputfile. Make and usage are described in the fileheader. For the program listing see EDSim.c in the git-repository. EDSim isn't working as properly as the real event detection will do and some of the events might be misinterpreted due to the used threshold-algorithm. But most of them are found and the difference is small enough to ensure relevant datasets for testing the compression/decompression of event detection data.

6. Programming of the Standalone solution

The basis for the SMILE Science Data Processing is the program from the CHEOPS mission doing the compression and decompression.
Due to the requirements of SMILE (see (Branduardi-Raymont et al. 2018), (Smit S.J.A. 2019), (Reimers Ch. 2019) and (Mecina, M. et al. 2020)) which differs from that of CHEOPS, some adaptations had to be made.

6.1. Program repository

Git is used as the program version control system. All programs and related files and documents are stored in the SMILE-git at *smile@herschel.astro.univie.ac.at:1722/home/smile/OBSW.git* in the branch *berndt*. Here the following programs are stored:

- Data simulator V.1.41: Directory *implementation/DataSim*
- Compression and CrIa-routines: Directory *implementation/CompressionEntity*
- Compare FITS-files: Directory *implementation/FitsCompare*
- Create event detection data files: Directory *implementation/EDSim*
- Compare event detection data files: Directory *implementation/EDCompare*
- Copy of this master's thesis: Directory *Master-Doc*

A schematic drawing of the directory-tree is given in appendix C.

The program user account where all the programs and testdata can be found is *berndt@a538-s-ariel.astro.univie.ac.at*.

6.2. Implementation of full frame compression and decompression

6.2.1. Design principles

The implementation for full frame compression / decompression is based on the already existing solutions for the CHEOPS mission. The CHEOPS implementation uses C and C++ as programming languages. Most of the code can be reused and only some adaptations were necessary.

General redesign

The redesign for SMILE was done in the following steps:

- Start with a clone of the CHEOPS implementation including all necessary .h, .c, .hpp and .cpp files.
- Change CheopSimFileIo.h to SmileSimFileIo.h to change parameters especially for image settings and also CheopSimFileIo.c to SmileSimFileIo.c. Implement new Read-functions in SmileSimFileIo.c according to the SMILE FITS-requirements: ReadSubImages, ReadFullFrameImages and ReadEventData.
- Redesign some parts of Compression program to be aligned with SMILE requirements. Change the following parameters in SmileSimFileIo.h and SdpCeDefinitions.h from fixed defines to global variables in SmileSimFileIo.c and CompressCcd.cpp to be flexible in image structures: FULLX, FULLY, CCD_FULL_X, CCD_FULL_Y. So both binning factors (6x6 and 24x24) can be used without any program changes.
- Implement the transfer of the binning mode to the ce-file for compression and also from the ce-file to write it to the final fits-file for decompression because SMILE uses several binning modes.
- Change EXPOS_FULLOFFX to (0 : 0) in SmileSimFileIo.h to avoid cutoffs of left and right margins.
- Change DEBUG_STACKING_NUMBER to 1 in CompressCcd.cpp because there are no stacked images.
- Include some printouts (DYNPRINT) at the beginning and at the end of the single functions to be able to do a dynamic analysis of the code. Can be switched on/off using include file *Sdpprint.h*.
- To avoid unnecessary debug-fits-files creation include *#undef DEBUGFILES* in *SdpCeDefinitions.h*.
- Optional: Reducing the amount of code by excluding unnecessary parts of the original CHEOPS-code. The dynamic analysis in appendix A for compression and appendix B for decompression will be helpful finding the relevant parts of the code. The achived code reduction is about 25%.

Enhancements for unbinned mode

In addition for the unbinned full frame mode which produces FITS-files of about 34 MByte it is necessary to extend the used buffers and size of the dynamic memory allocation:

- *CrIaDataPool.h*: Extend the buffer allocation from 1024 x 1024 to 4510 x 4510 creating *define CCD_MAX 4510* used in *CompressionEntity.cpp*. Enhance *define SDB_SIZE*.

- *CrIaDataPool.h*: There are 2 CCDs in SMILE, therefore *define CCD_NR 2*.
- *CrIaDataPool.c*: Set values for SIB-, CIB- and GIB-sizes to CCD_MAX.
- *CompressionEntity.cpp*: Set values for cibSizeWin and gibSizeWin to CCD_MAX. In addition it is necessary to adjust *cestruct.Stacked_OriginalSize* because it is truncated by 24 bit at reading from CE-file. Now 32 bits are necessary.
- *SdpCollect.c*: Adjust *cestruct.Stacked_OriginalSize* because it is truncated by 24 bit at writing to CE-file. Now 32 bits are necessary. Please note, that the structure of the CE-files has changed and data is shifted by 8 bits from here on.
- *SdpBuffers.h*: Enhance *define SRAM1_SWAP_SIZE, define SRAM1_RES_SIZE, define SRAM1_FLASH_SIZE*.
- *SdpBuffers.c*: Enhance *cestruct->CibSize*.
- *GroundSupport.h*: GROUNDBUFFERSIZE enhanced.
- *CompressCcd.cpp*: Enhance *cibSize* and *sibSize* to CCD_MAX.

The resulting programs are able to compress full frames in various forms and were used to perform several tests to get a sample of well fitting compression methods for SMILE. Further optimization of the program and in addition using an underlying new operation system UVIE FlightOS are described in chapter 11.

6.2.2. Modules

The whole program system consists of the following modules and their corresponding h-files:

- SmileSimFileIo.c: Reads FITS-input
- CompressionEntity.cpp: Class for Compression Entity.
- GroundSupport.c: Initialize Science Data Buffers Gib, Sib, Sdb.
- GsAlgorithmsImplementation.c: Contents the Decompress-function.
- GsAlgorithmsImplementationLlc.c: Mainly support decompression of Science-data.
- IfswAlgorithms.cpp: Class for decompression (of parts) of the SDP.
- SdpAlgorithmsImplementation.c: Compress function together with its many auxiliary functions and algorithms.
- SdpAlgorithmsImplementationLlc.c: Various lossless compression algorithms used in the data processing chain.
- SdpBuffers.c: Setup and allocate Compression- and Decompression-buffers.

- SdpCollect.c: A single function which packs the contents of a ComprEntStructure into a bitstream.
- SdpCompress.c: Compresses the individual parts of a FITS using the Compress-function.
- SdpNlcPhot.c: Contains the nonlinearity correction function and the quick photometry function.

Some modules from CrIa and their corresponding h-files which were copied from CrIa- to CompressionEntity-directory and partly modified:

- CrIaDataPool.c: Initializes DataPools, provide Copy and Paste functions.
- EngineeringAlgorithms.c: Contents a lot of special procedures e.g. for Centroiding.
- IfswConversions.c: Converts raw into engineering values for temperature, voltage a.s.o.
- IfswMath.c: Contents several mathematical functions like "round", "sqrt", "log" a.s.o.

and the 2 main programs for compression and decompression:

- CompressCcd.cpp: Compression module (keep parametrization of compression types)
- DecompressCcd.cpp: Decompression module

In appendix D all program modules and include-files are listed, also for the enhancement-packages of SMILE-SDP. Not every procedure of the several modules is used during runtime in the version for SMILE. It is over all a generic concept for compression and decompression of scientific data and can be customized for the specific use case.

6.2.3. Some important data structures

Compression- and decompression-program are using a lot of data structures for their work. Here I just want to mention some examples of the most important one:

- SemWindows: Is defined in file *SmileSimFileIo.h* and describes the cube-structure of images. Is used for variable *semcube*.
- ComprEntStructure: Is defined in file *SdpCompressionEntityStructure.h* and describes the structure of compressed data package. For streaming the compressed dataset to the GIB the interaction of the structure with the vector binary_data should be observed which is defined in *CompressionEntity.hpp*.

Both structures are expanded by the binning-mode for SMILE.

6.2.4. Buffersizes

Remark on buffersizes: To enable the usage of unbinned images which produces a big amount of data, the buffersizes were enhanced. It has to be checked in real environment if enough HW-memory is available to meet the now used buffersizes.

6.2.5. New (modified) functions for SMILE in SmileSimFileIo

ReadSubImages

ReadSubImages opens and reads the FITS-file which contains the image. At first it is reading the header-data necessary for the dimensions of the image. Then the global variables FULLX, FULLY and FULLFRAME are initialized. In the next step the image is read to *fdata* and if everything was working allright, *fdata* will be returned. Details can be seen in SmileSimFileIo.c in the git-repository.

ReadFullFrameImages

ReadFullFrameImages uses the ReadSubImages routine to read the image of a FITS-file. It also initializes the semcube-structure and does the copy from tframe to semcube (see SmileSimFileIo.c).
Note: Semcube is the most important structure containing the data of the input-image.

6.2.6. Dynamic Program Sequence

For dynamic analysis of the program I inserted "ENTER"- and "LEAVE"-statements via DYNPRINT (printf) in all procedures. They shall be removed before the program will go live, because printf-statements shall not be included in space-SW.
Beside DYNPRINT also other sorts of printf definitions can be found in *Sdpprint.h* and can be switched off or on by assigning them to "printf" or the "noop" function also defined in *Sdpprint.h* just doing nothing and avoiding any printouts.
The idea behind is to be able to trace the dynamic behavior of the program. In the first step this was necessary to analyse all the functions and in which chronological order they are carried out because when "reading" a program written by someone else, it is necessary to get a dynamic view on the program beside the static analysis of the code.
In addition it is helpful for tracing the program during runtime when changing parts of the code or adding new functions to it.
The dynamic charts in appendix A for compression and appendix B for decompression show how the program is running during compression and decompression of full frame images. Some of the functions are run several times in a row like the CrIa_Copy and CrIa_Paste functions or bits_used so they are shown only once in the runtime for such occurrence.

6.2.7. Parameterization of the compression chain

The compression chain includes the steps from reading the product (Header, image a.s.o.) from the SIB, over preprocessing, lossy-compression steps, decorrelation and lossless compression to finally writing the compressed data to the CIB. In the offline-modus the data is read from a FITS-file and the final compression is written to a CE-file.

Which kind of lossy and lossless compression is used can be parametrized by setting the values of *WinHeader_CeKey* or *FullHeader_CeKey* for the FITS-file-header and *Stacked_CeKey* or *FullImage_CeKey* for the image in file *CompressCcd.cpp* depending on the decision if a window-image or a full frame is in use.
The values for the parameterization are defined in file *SdpCeDefinitions.h*.

An example for such kind of parameterization can be:
unsigned int Stacked_CeKey = PRODUCT_STACKED | PREPROC_NLCPHOT | LOSSY1_MEAN | LOSSY2_CIRCEXTRACT | LOSSY3_NONE | DECORR_IWT1 | LLC_ARI1;

The values will be used to compress the data and will also be written to the compressed CE-file via *ce->cestruct* to save them for further decompression.
In the decompression cycle the values are read from the CE-file and used to parametrize the decompression chain.

6.2.8. A short program description

Compression

CompressCcd starts with the Initialization of the DataPool in *InitDataPool* which can be found in *CrIaDataPool.c* and is just preparing the structures for the application, the flight-SW and Cordet.
After reading the commandline-option (-f) in the now called function *ReadFullFrameImages* the Fits-file is read and the content is copied to an area called *dest* with the help of function *cparea*. In structure *semcube* the relevant data like dimension of the frame and also the pointer to the frame-content *fullframe* is stored.
The next step is setting the configuration for the compression in *SetCeConfiguration* using the data from *semcube*. Here the parametrization of the compression according to the "product" like HEADER, FULLFRAME a.s.o. is done.

After reading the stackorder (which is 1 in our case), a new instance of the *CompressionEntity* called *ce* is created which does a *SetupCeBuffers* and then via function *SetCeSib* the science- compression- and ground-buffers are initialized. *SetCeSib* does this using the functions *ConfigSdb* and *InitSdbFull* of module *Groundsupport.c*.
The next step is to initialize the compression entity structure *ce* of the datapool using function *InitComprEntStructureFromDataPool* of module *Sdpbuffers.c*. Here the time, ce-counter, version, exposuretime and all CE-Keys are set.

Then *DummyHeaders* could be set but this isn't really necessary.

Afterwards the image input data is copied to the SIB. The SIB is the input buffer for the compression. This is done by the function *FillSib* of module *GroundSupport.c*.
The input image is not just a 2-dimensional array of pixel values. In addition, header data, such as temperatures and voltages, counters, settings such as exposure time etc. are also stored within a SIB. In order to generate the internal layout of the SIB, the CrIaSib structure is set up for an incoming image whenever the first packet of such a set is processed. For full-frame mode function *CrIaSetupSibFull* from *SdpBuffers.c* is used to fill the sib-structure.

Now in function *Compress* of the created instance of *CompressionEntity* compress of all the data is done and the bitstream of the CE is created. This is a complex process which will not be described here in detail.
Incrementing the sib by stackorder is unnecessary in our case, because there are no stacked images. So function *Collect* of the created instance of *CompressionEntity* is the next step. It just packs the contents of the ComprEntStructure ce into a bitstream using function *Collect* of module *SdpCollect.c*. The output can also be picked up in the ce->binary_data vector.
This output is also written to a file named "CE_Full_00000.ce" for the decompression later on.
Allocated memory space for the groundbuffer is released using *ce->FreeArrays* in the last step.

For more details please refer to the inline documentation of the modules mentioned above.

Decompression

DecompressCe also starts with the Initialization of the DataPool in *InitDataPool* which can be found in *CrIaDataPool.c* and is just preparing the structures for the application, the flight-SW and Cordet.

After reading the compression ce-file into structure *idata*, a new instance of the *CompressionEntity* called *newce* is created which does a *SetupCeBuffers* and the science-compression- and ground-buffers are initialized and a print of the buffer-adresses) is done. Structure *idata* is provided in *newce->binary_data* and allocated memory of *idata* is released.
newce->Decompress is started and is doing the decompression.
Finally a FITS-file containing the decompressed image data is created and the instance *newce* is deleted which also releases the allocated memory.

For more details please refer to the inline documentation of the modules mentioned above.

6.2.9. Make and run programs

The Makefile is helpful for building the programs for compression and decompression. After a successful "make", the programs *CompressCcd* and *DecompressCe* can be found in the directory *build*.

To compress a FITS-file just start:
./build/CompressCcd -f FITS-filename > SM_Cmp_Full_print.txt
in the commandline where the *-f*-option means to work with full frames, *FITS-filename* is the name of the FITS-file to compress and *SM_Cmp_Full_print.txt* is the name of the print-redirection of the program for later analysis of the program runtime behavior. You will get a file *CE_Full_00000.ce* with the compressed content of the FITS-file.

To decompress the file just start:
./build/DecompressCe CE_Full_00000.ce FITS-output > SM_DeCmp_Full_print.txt
in the commandline where *CE_Full_00000.ce* is the name of the compressed file, *FITS-output* is the name of the decompressed FITS-file and *SM_DeCmp_Full_print.txt* is the name of the print-redirection of the program for later analysis of the program runtime behavior. *FITS-output* will contain the result of the decompression as FITS-file and can be compared with the original one.

6.3. Event Detection Data: Using CHEOPS Compression methods

Beside full frame images also event detection data from single events have to be transmitted to the ground station. Event detection is done by the EDU of the FEE and the event detection data is stored in a 64 byte long package as described in chapter 5.

6.3.1. Design principles

At the moment (April 2022) the interface for event detection data isn't specified finally but a draft version is available (Smit S.J.A. 2019) which is used as the base to start with a preliminary version of data compression:
The whole data amount for a single event is 64 byte. The ce-structure information necessary for decompression is already 37 byte long. The ce-structure or relevant parts of it also have to be transmitted to the ground station, otherwise decompression will not work.
This overhead for a single event might be too big. Assuming that the compression rate of a lossless compression of the event detection data is about 2, a short appraisal of the data to be transmitted gives 64 / 2 + 37 = 69 byte which is bigger than the original 64 bytes.

The assumption is now, that all event detection data packages of an image frame are stored in one data set. This data set can be treated as whole and compressed in one step. Also a preprocessing is possible, because some data are the same for all event detection data packages (e.g. frame number).

To reuse as many as possible already existing program procedures and data-structures for compression and collection, the data of a single event is treated like an image of dimension 32 x 1 x 1 of shorts and all packages are forming a 32 x 1 x 1 x event-amount structure which is treated like a full frame image for the following program steps. A difference is that there is no image header and therefore also no header compression and decompression will be in place.

6.3.2. Compression

Implementation for compression was done in the following way:

- Split the compression in *CompressCcd.cpp* in a part for full frame and a second part for event detection data where function *ReadEventData* is called.
- Add a new option *-e* for event detection data to *CompressCcd.cpp*.
- Define *EVENTDATABUFFER* with a length of 32 in *SmileSimFileIo.h*. The length of 32 determines an array of 32 short int corresponding to the 64 byte of the event detection data package length.
- Insert an additional define *SDPEVENTDATACEKEY_ID* in *CrIaDataPoolId.h*.

- Extend all datapool-variables with a new entry for the SdpEventDataCeKey. That's *dpIasw*, *dataPoolSize*, *dataPoolAddr* and *dataPoolMult* in module *CrIaDataPool.c*. Also extend the initialization procedure *initDpAddresses* in the same module.
- Add *#define ACQ_TYPE_EVENT 6* and *#define PRODUCT_EVENTDATA 0xB0000000* to *SdpCeDefinitions.h*.
- Implement a new function *ReadEventData* in module *SmileSimFileIo.c* to read event detection data.
- Add the definition for *unsigned short * ReadEventData (char *infilename, struct SemWindows *semcube, unsigned int mode);* to *SmileSimFileIo.h*.
- To distinguish between the image processing and the event processing the *cestruct.Stacked_CeKey* is set to *SDPEVENTDATACEKEY_ID* in *CompressCcd.cpp* before starting *ce->Compress()*. This leads to a new branch later on in the Compress-function which can be found in *SdpAlgorithmsImplementation.c*. It should be noted here, that not the *sdbstate* will be taken to separate between the different products but the *cestruct.Stacked_CeKey*. The *sdbstate* will still be *CrIaSdb_CONFIG_FULL* to treat the Event Detection Data like a full frame at all but at those positions in the code where a special processing of Event Detection Data is necessary, the differentiation will be done via *cestruct.Stacked_CeKey*.
- In function *SdpCompress* of module *SdpCompress.c* it is necessary to avoid the header compression.
- Implement the new Compress-branch for event detection data in the Compress-function of *SdpAlgorithmsImplementation.c*.

6.3.3. ReadEventData

The function ReadEventData is reading the 64 byte long event detection data packages stored in a hex-file. For more details see the procedure in SmileSimFileIo.c.

In module *CompressCcd.cpp* an additional branch for the event detection data is necessary.

6.3.4. Decompression

Implementation for decompression was done in the following way:

- In function *Decompose* in module *CompressionEntity.cpp* a new case *PRODUCT_EVENTDATA* was introduced working similar as *PRODUCT_STACKED*.
- In the main-module *DecompressCe.cpp* the usage of function *newce->GetImagettesCropCoordinates* and the returnvalue *crops* is not allowed for sdp-product *PRODUCT_EVENTDATA*.

- In main-module *DecompressCe.cpp* the processing of imagettes is excluded for sdp-product *PRODUCT_EVENTDATA*.
- In main-module *DecompressCe.cpp* it is necessary to distinguish between images which are written into a FITS-file and event detection data packages written to a hex file. So a short code-structure for the second case was needed.

6.4. Using a new Compression-method for Event Detection Data

Another possibility is to compress the single event detection data sets one by one. That means the header-data must be compressed (or not) separately and the 5x5 event image also for each event.
I tried out a short implementation of such kind of compression using the ARI-code with IWT2-decorrelation and as an alternative the Golomb-code. Both trials show no satisfying results because the "compression" needs more space than the original data.
For this reason this kind of data processing wasn't taken into account for further investigations and I developed the following compression method using the differences between the single pixels of a 5x5 event image. When more than one event is transferred in the event list, also the header data of the following events can be reduced.

6.4.1. Reducing header data

As described in (Smit S.J.A. 2019) and listed in chapter 5 out of the 14 bytes of an event header 8 (Event data type, Event data sequence number, Event data column and Event data row) are individual for each event in a list generated by the same frame.
Therefore only the header of the first event must be stored completely, for all other events in the list only type, sequence number, column and row are necessary which reduces the amount of data from 14 bytes to 8 bytes.

6.4.2. Reducing 5x5 event image data

The assumption is, that in the surrounding of an event-pixel most of the other pixels have more or less a background-value. These values should be very similar to each other and the difference between them should be small. Therefore most of the differences should be lower than 256 and can be stored within one byte instead of the two necessary for a complete pixel-value. Depending on the content of a 5x5 event image this could lead to a datareduction from 50 bytes to 32 bytes in the optimal case which will be the normal one for most of the events. It is a derivate of the data differencing method (Korn D. G. & Vo Kiem-Phong 2002).

6.4.3. Design principles

- Store only the header of the first event in total. For the following events only type, sequence number, column and row are stored. Logical address, protocol-Id, length and frame counter can be derived from the first header.

- Reduce data for 5x5 event images by deriving the difference between each 2 pixels. Store the first pixel in total, for the others just store the differences. If a difference is lower than 256 it will be stored in one byte, otherwise two bytes are necessary. To know if one or two bytes are carrying the difference, a bitmask represented by a

4-byte integer will show 1 in bits for pixels with 2-byte difference values and 0 in bits for pixels with 1-byte difference values.

- Using only positive difference values makes the implementation easier. Therefore the minimum value is stored and for the other pixels only the differences.
- The complete reduced data set for an event will look like:

Byte	Data
0 - 7	Reduced header data (type, seq. number, column, row)
8 - 11	Bitmask indicating small or large differences
12 - 13	Minimum value pixel
14 - var.	Difference-dataset: Difference values of pixels (variable length)

Table 6.1.: Event detection difference data set

- The variable length of the difference-dataset does not need to be transmitted because it can be derived from the bitmask during decompression.
- Include a new define for CC_LLC_EVENT_DIFF into *CrIaDataPool.h*.
- Include a new define for LLC_EVENT_DIFF into *SdpCeDefinitions.h*.
- Add an additional initial value for EventData_CeKey including LLC_EVENT_DIFF to *CompressCcd.cpp* and *UVCompressCcd.cpp* for test purposes.
- Add a new case to the compression branch for lossless compression in module *SdpAlgorithmsImplementation.c* function *Compress* similar to that one of Arithmetic compression.
- Add a new case to the decompression branch for lossless decompression in module *GsAlgorithmsImplementation.c* function *Decompress* similar to that one of Arithmetic decompression.
- Add two new modules *SdpEventDiffCompression.c* and *SdpEventDiffDecompression.c* using the data structures of the SMILE Compression as described above. For details see the module code below.

Event compression and event decompression are listed in SdpEventDiffCompression.c and SdpEventDiffDecompression.c in the git-repository.

Data structure

The compression entity of SMILE delivers the CE-structure including the CE-information for later decompression and the CE-data containing the compressed data. This will stay the same for the new event compression type. The only difference is, that the compressed data needs an internal structure because the data-info and length of the compressed

data differs from event to event (see table 6.1). Therefore the data-content isn't flat but structured in the following way:

[event-counter]
[compressed event data (variable length)]
[compressed event data (variable length)]
⋮ ⋮
[compressed event data (variable length)]

6.4.4. Make and run programs

Make of the programs is the same as described for full frame images.
To process an event detection data file just start:
./build/CompressCcd -e filename
The *-e*-option means to work with event detection data stored in *filename*. You will get a file *CE_Event_00000.ce* with the compressed content of the Event detection data file.
To decompress the file just start:
./build/DecompressCe CE_Event_00000.ce Eventdata-output
in the commandline where *CE_Event_00000.ce* is the name of the compressed file, *Eventdata-output* is the name of the decompressed Event detection data file which can be compared with the original one.

7. Test Strategy for Standalone compression

A number of tests are necessary to ensure that the program runs as error-free as possible. Attention must be paid to the greatest possible test coverage with regard to the possible cases that may arise. Test coverage will include on one hand to ensure that the software is running as error-free as possible under all frame conditions and is able to handle exceptions in datainput and interfaces.

Therefore, a test suite was created with as much test data as possible, which covers a wide range of cosmics, events, exposure times and both binning modes. Also it is ensured that data from one CCD and also data-packages from both CCD's and all kind of readout areas can be handled.

A lot of test strategies for testing of software are in place and hundreds of articles and books are dealing with this theme.
The challenge is to find an appropriate strategy for testing the Science Data Processing to ensure and show that it is working end to end which means the compressed and afterwards decompressed data really is the same as the original one in case of lossless compression and similar enough to meet the scientific requirements in case of lossy compression methods on one hand and also not too much extend the effort for a master's thesis at all.

7.1. End-to-end testing strategy

Choosing an end to end test strategy seems to be the most effective way to check if the software is working properly and to ensure that the results are meeting the above mentioned requirements.

To my long lasting experience in designing and testing software this seems to be the optimal strategy. Nevertheless more than 300 testcases were performed to get a necessary coverage of the different possibilities for images and also event detection data.

There is also a lot of literature about end to end testing. For a good and quick introduction please refer to (testim 2021) and (Borg B. 2020).

7.2. Full frame testing

Full frame standalone tests means, that the Compress- and Decompress-function is tested with image FITS-files generated with DaSi and is running on a normal PC or a computer of the Institute for Astrophysics in Vienna.

As already mentioned, DaSi offers a wide range of opportunities and so various test-data was generated varying event-amount, cosmics, exposuretime. All 3 kinds of binning-types were used, i.e. unbinned mode, 6x6 binning and 24x24 binning. Also the combinations of only one CCD (A or B) or both together (A+B) were taken into account.

Also FullNoise and LocalNoise were both tried whereas LocalNoise isn't a working mode of the real SMILE-environment and only an artificial mode to see how it influences the compression conditions. Of course LocalNoise increases the compression rate because a certain part of the image contains no additional information but the results of such tests are not relevant for SMILE and will not be shown here.

7.2.1. lossless compression

In the first step the lossless compression was tested because this will be of course one of the most important use cases in practise to compress and transmit images or parts of images. Tests had always the same sequence:

- Create a test-FITS with DaSi using the parametrization via config.xml: Result will be a set of FITS-files for CCD-A, -B and A+B and several exposuretimes down to 0 from chosen maximum value in steps of 1 second.
- All these files are used to test the compression by compressing and decompressing them.
- An end to end comparison of original FITS with compressed & decompressed one will give the residuals after compression&decompression cyclus.

End to end comparison

For the above mentioned end to end comparison, I implemented a short program **F_comp** comparing 2 FITS-files pixel per pixel and calculating the difference per pixel, print it if any and creates a Compare_Residual.fits file where the differences per pixel can be seen in detail.

Comparison of FITS-files can also be done using tools like DS9 or ImageJ but **F_comp** offers some additional opportunities to quickly determine the pixel-differences between the image files.

7.2.2. lossy compression

Beside the lossless compression also various methods of lossy compression are tested to find out the most suitable one for usage in SMILE.
The overall strategy (creating test-FITS, testing compression / decompression, end-to-end comparison) is the same as used for lossless compression. It should be taken into account that for SMILE the whole image is used and necessary and therefore several types of lossy compression of CHEOPS, which are using only a part of the image, can't be applied.

7.3. Event detection data testing

A similar approach as for the full frame images is used for event detection data testing. Testdata are generated with EDSim. Only lossless compression is tested because no data should be lost of the event data packages otherwise a decompression will lead to distorted data.

End to end comparison

Files normally can be compared in a simple manner with commands like "diff" or "cmp" but to detect all byte-errors within the lossless compressed/decompressed files I implemented a short program **ED_comp** comparing 2 files on HEX-basis, printing all differences and counting the errors over all. It's an analog approach as it was done with **F_comp** for images.

Remark: The decompression often puts the output data into short integer instead of byte values. A conversion leaving out the MSB of the shorts is necessary. Therefore 2 different types of compare algorithms are implemented in *ED_comp*.

For event detection data only lossless compression will be used and tested because no data shall be lost or irreversibly changed during compress/decompress cycle.

7.4. Meeting the program requirements

Of course the requirements of SMILE about full frame images and event detection data have to be taken into account (see (Branduardi-Raymont et al. 2018), (Smit S.J.A. 2019)).

The scientific data process in SMILE will produce full frame images unbinned and in the binning modes of 6x6 for soft X-ray respectively 24x24 for EUV. Beside the full frame images also Event Detection Data is produced by the Event Detection Unit of the Front End Electronic (FEE) and sent to the DPU which carries the IASW and the SDP as a part of the IASW.

One of the main points in testing and teststrategy is to have a close look at the goals of the program. In this case the compression rate and the necessary time for compression and decompression are the main criteria because the whole procedure shall be able to transmit the scientific data from the satellite to the ground station within a timeframe short enough to meet the transmit channel requirements and also use as less computation resources in the satellite as possible.

That means the faster the compression&decompression cycle and the higher the compression rate are, the better the programs are meeting the requirements and in the best case every readout cycle of the CCD's can be supported. Readout cycles of the FEE are described in (Smit S.J.A. 2019) and related visio graphics both delivered by University College London. The theoretical basis is the rate distortion theory. It describes the minimum data transmission rate required to transmit information with a certain quality.

8. Compression Test Results

8.1. Compression time and compression rate

The goal of the tests was to find suitable compression concepts for the SMILE SXI-images which can be used as a base for the final implementation of the SMILE SDP.

Due to the fact that compression time, compression rate and decompression time are very important for the usage of compression methods it would be good to know how these parameters are depending on exposure time, cosmics, event amount, local or full noise and binning.

While the values for compression rates are absolute, compression- and decompression-times can only be treated as relative to each other because they are depending on the runtime environment and will be different on the target-DPU of SMILE.

Test samples

The following images were used for testing:

SXI_single_light_FullNoise_CCD_A_b=6_et=1.0_cr=24.0_ea=500.fits
for 6x6 binning.

SXI_single_light_FullNoise_CCD_A_b=24_et=1.0_cr=24.0_ea=500.fits
for 24x24 binning.

SXI_single_light_FullNoise_CCD_A_b=1_et=1.0_cr=24.0_ea=500.fits
for unbinned mode.

Test data

- The size-relation between original data and compressed data gives the compression rate.
- For compression- and decompression-time the overall runtimes of the programs are taken.

Important note:
Please note that the runtimes are only relative values between the several tests running in the identical environment. Running on a DPU programs will show a different runtime behaviour and therefore runtime-figures should be treated only for a first comparison between the tests but can't be taken a serious measurements for the behaviour on a DPU.

Test arrangements

Some minor changes were necessary in modules *CompressCcd.cpp* and *DecompressCe.cpp*:

- Include clock_gettime at the beginning and the end of the main-procedure to get the runtime.
- Change parameters for *FullImage_CeKey* to change the compression style.

8.2. Full frames

Standalone tests were performed using the simulation data from DaSi and the program *F_comp* to compare end-to-end results of compression and decompression with the original images. In all tests both binning factors (6x6 and 24x24) were used to assure that soft X-ray images and EUV images can be transmitted and also a test set for the unbinned mode was necessary. A full frame image FITS-file created with DaSi has a size of 929 KB with 6x6 binning and 62 KB using 24x24 binning. For unbinned mode the file-size is about 34 MB.

8.2.1. Lossless compression

In the first step the lossless compression was performed because it will deliver 100% reproducible images. Using the lossless compression no differences between the end-to-end results and the original images were found with *F_comp* (no residuals).
The compression parameters mentioned in chapter 4 are set for this test as follows for the image (see table 8.1 and for the header set table 8.2):

Product	**Fetch**	**Prepro-cessing**	**Lossy1: Stacking**	**Lossy2: Specific**	**Lossy 3: Rounding**	**Decorre-lation**	**LLC**
Full Image	STACKED	NONE	MEAN	NONE	NONE	IWT1	ARI1

Table 8.1.: Lossless compression parameters for frame images.

Product	**Fetch**	**Prepro-cessing**	**Lossy1: Stacking**	**Lossy2: Specific**	**Lossy 3: Rounding**	**Decorre-lation**	**LLC**
Headers	HEADERS	NONE	NONE	NONE	NONE	KEYFR	RZIP

Table 8.2.: Compression parameters for Science Headers.

Lossless compression: 6x6-binning

Tables 8.3 to 8.5 show an overview of the test results for 6x6 binning:

Binning = 6, cosmics = 12, events = 500, light-frame			
exposuretime	compression-time	compression rate	decompression-time
0	0.166	2.215	0.269
1	0.164	2.231	0.276
2	0.167	2.218	0.271
3	0.165	2.219	0.273
4	0.164	2.236	0.271

Table 8.3.: Execution times (seconds) and compression rate depending on exposuretime.

Binning = 6, exposuretime = 1, events = 500, light-frame			
cosmics	compression-time	compression rate	decompression-time
0	0.166	2.225	0.271
6	0.165	2.228	0.272
12	0.164	2.231	0.276
18	0.166	2.219	0.273
24	0.167	2.218	0.272

Table 8.4.: Execution times (seconds) and compression rate depending on cosmic rate.

Binning = 6, exposuretime = 1, cosmics = 24, light-frame			
events	compression-time	compression rate	decompression-time
0	0.166	2.227	0.271
250	0.166	2.223	0.271
500	0.167	2.213	0.272
750	0.168	2.224	0.273
1000	0.167	2.221	0.272
2000	0.168	2.218	0.272
4000	0.165	2.186	0.275

Table 8.5.: Execution times (seconds) and compression rate depending on event amount

Lossless compression: 24x24-binning

Changing the binning to 24 x 24 shows a similar picture just with the exception that the runtimes are lower overall and the compression rates are higher which corresponds with the smaller images in comparison to the 6 x 6 binning (see tables 8.6 to 8.8).

Binning = 24, cosmics = 12, events = 500, light-frame			
exposuretime	compression-time	compression rate	decompression-time
0	0.023	3.665	0.032
1	0.023	3.458	0.032
2	0.023	3.391	0.032
3	0.023	3.636	0.032
4	0.023	3.428	0.032

Table 8.6.: Execution times (seconds) and compression rate depending on exposuretime.

Binning = 24, exposuretime = 1, events = 500, light-frame			
cosmics	compression-time	compression rate	decompression-time
0	0.023	3.482	0.031
6	0.024	3.332	0.031
12	0.023	3.458	0.032
18	0.023	3.366	0.031
24	0.024	3.446	0.031

Table 8.7.: Execution times (seconds) and compression rate depending on cosmic rate.

Binning = 24, exposuretime = 1, cosmics = 24, light-frame			
events	compression-time	compression rate	decompression-time
0	0.023	3.518	0.031
250	0.024	3.139	0.031
500	0.024	3.446	0.031
750	0.023	3.481	0.031
1000	0.024	3.119	0.031
2000	0.024	3.124	0.031
4000	0.023	3.430	0.031

Table 8.8.: Execution times (seconds) and compression rate depending on event amount.

Lossless compression: Unbinned mode

For several purposes it is also necessary to transmit unbinned frames beside the 2 operation modes 6x6 binning and 24x24 binning. This kind of full frames are much larger than the other 2, we are speaking of about 34 MByte of data in comparison to the 950 kByte resp. 60 kByte for the two operation modes. The necessary program-adaptations to handle such amount of data are described in chapter 6.

The following table 8.9 shows the results for the unbinned mode. Only the case with exposuretime = 1 and cosmics = 24 is shown because the other ones are similar.

Unbinned, exposuretime = 1, cosmics = 24, light-frame			
events	compression-time	compression rate	decompression-time
0	3.961	4.434	7.693
250	3.941	4.398	7.698
500	3.922	4.657	7.675
750	3.943	4.379	7.720
1000	3.965	4.412	7.710
2000	3.938	4.502	7.706

Table 8.9.: Execution times (seconds) and compression rate depending on event amount for unbinned mode.

Lossless compression dependencies overview

- There are no dependencies on exposuretime or cosmic rate of compress- or decompress-runtime or compression rate.
- There is only a very small dependency on event amount for full noise images of compress- or decompress-runtime or compression rate.
- There is a dependency on binning. 24 x 24 binning shows lower runtimes over all and the compression rates are higher than for 6x6 binning.
  For unbinned images compression rates are higher than for 6x6 binning. Runtimes are also higher due to the higher amount of data.

Some details about noise

Noise is an important factor in lossless compression. Pure noise is "incompressible" as already stated, so it would be good to get rid of the noise. Looking at the two alternatives FullNoise and LocalNoise of the DaSi-FITS, it can be clearly seen that the LocalNoise is much more compressible. I did some tests with LocalNoise just to get an idea of compressibility.
On the other hand full frames with local noise just around the events are not a real scenario processing SMILE images. Therefore this kind of full frames with local noise will no longer be taken into consideration for the following implementation- and test-strategy.

Various alternatives of lossless compression

In the previous chapter it can be seen that there were no dependencies on exposuretime, cosmic rates or event amount for the compression of full noise images. So the next step is to investigate several alternatives of lossless compression. Therefore a certain FITS-file *SXI_single_light_FullNoise_CCD_A_b=6_et=1.0_cr=24.0_ea=500.fits* (see figure 8.1) was chosen to find out if there are any differences:

Figure 8.1.: Full Noise with 24 cosmics and 500 X-ray events used as compression example.

Lossless compressions as described in *SdpCeDefinitions.h* is also combined with various decorrelation settings. The settings are done as described in chapter 6.

Which kind of lossy and lossless compression are used can be parametrized by setting the values of *FullImage_CeKey* for the image in file *CompressCcd.cpp*.

The values for the parameterization are defined in file *SdpCeDefinitions.h*. An example for such kind of parameterization might look like:
unsigned int FullImage_CeKey = PRODUCT_STACKED | PREPROC_NONE | LOSSY1_NONE | LOSSY2_NONE | LOSSY3_NONE | DECORR_DIFF | LLC_ARI1;
as it is used for the tests of lossless compression.

Now varying DECORR_... for decorrelation and LLC_... for lossless compression show the following results (see table 8.10):

Decorrelation	Lossless	compression-time	compression rate	decompression-time
NONE	RZIP	0.125	0.625	0.206
	ARI1	0.149	1.141	0.128
DIFF	RZIP	0.125	0.609	0.206
	ARI1	0.124	2.219	0.258
KEYFR	RZIP	0.125	0.625	0.206
	ARI1	0.126	1.064	0.328
IWT1	RZIP	0.126	0.638	0.211
	ARI1	0.126	2.218	0.266
IWT2	RZIP	0.126	0.645	0.216
	ARI1	0.126	2.244	0.261

Table 8.10.: Comparison of various lossless compression in combination with decorrelation.

Lossless compression alternatives show no residuals which means there are no losses in the image content.
Combinations with LLC_PACK16 were not taken into account because it is just a re-packaging and not a compression algorithm.

Only some combinations show a compression rate better than 2 where LLC_ARI1 in combination with DECORR_IWT2 seems to be the best one closely to the combinations with DECORR_IWT2 and DECORR_DIFF. Compression-times are very similar for all variations and decompression-times are also very similar for the best three alternatives.

For further investigations of lossy compression combinations the pair of Decorrelation = DECORR_IWT2 and lossless compression = LLC_ARI1 was used because it shows the best results for lossless compression only.

DECORR_IWT2 and LLC_ARI1

To get a good lossless compression of a noisy image it is necessary to decorrelate it first and then do the compression.
IWT2 means a basic 3-tap integer wavelet transform applied to the data set in 2D. (Ottensamer R. 2018)
ARI1 is an arithmetic coding using chunks of 8 kwords as input and encodes them using the probability distribution of the previous chunk. For the first chunk of a dataset, an initial model is used. (Ottensamer R. 2018)

8.2.2. Combination of lossy compressions with lossless compression

We take the best result of lossless compression from the previous chapter as a starting point and combine this with several lossy compression alternatives and look for the best one overall in compression rate.

Options of the various steps

First let's have a short look at the options for the various steps. Not all available combinations will make sense for compressing images so here I used only some of them which gave a meaningful result. The others are excluded after an evaluation I made and the reasons are listed here:

- **Preprocessing:** Here I used NONE. Photometry isn't used because aperture photometry makes no sense for the soft X-ray images. The detector is different, such that NLC2 does not apply.
- **Lossy1: Stacking:** Using just single frames stacking-options makes no sense and therefore LOSSY1 is set to NONE.
- **Lossy2: Specific:** I just set it to NONE because specific parameterization on statistic properties and also circular masks or circular extraction makes no sense when the details of the whole images are necessary and no areas can be cut off.
- **Lossy3: Rounding:** Rounding will give a coarser resolution but shall decrease the amount of compressed data. I tried ROUND2 up to ROUND5 in this case with good result. Soft X-ray events or EUV are still present after compression and decompression in their original spatial distribution as well as relative intensity of the single events but the statistic error increases with ROUNDx.

The parameterization for the image will look like:

unsigned int FullImage_CeKey = PRODUCT_STACKED | PREPROC_NONE | LOSSY1_NONE | LOSSY2_NONE | LOSSY3_ROUND2 | DECORR_IWT2 | LLC_ARI1;

For this parameters I got a compression rate of 3.5, a compression time of 0.121 seconds and a decompression-time of 0.231 seconds. That means compression rate increases while the times are very similar as for pure lossless compression.
With LOSSY3_ROUND3 the compression rate even increases to 4.4 while the times stay the same. Looking at figure 8.2 in comparison to figure 8.1 the loss in resolution seems to be within acceptable limits because cosmics and X-ray events are still detectable in their spatial distribution and intensity. A direct pixel by pixel comparison makes no sense in this case. Compression rates increase with ROUNDx and ROUND5 gives even a rate of 7.2.

Figure 8.2.: Image after compress/decompress with lossy ROUND3.
Compression rate = 4.4

Figure 8.3 shows the differences between the original image, figure 8.1, and the image after compression and decompression, figure 8.2, pixel per pixel.

This difference shows just some rounding-noise spread over the whole image with values of 0 to 3 ADU per pixel while soft X-ray events show values of about 2.000 to 3.500 ADU and cosmics above 50.000 ADU per pixel. The rounding noise is much lower than the X-ray event values and can be neglected for most of the use cases which offers the opportunity for a higher compression rate combining lossless with lossy rounding compression.

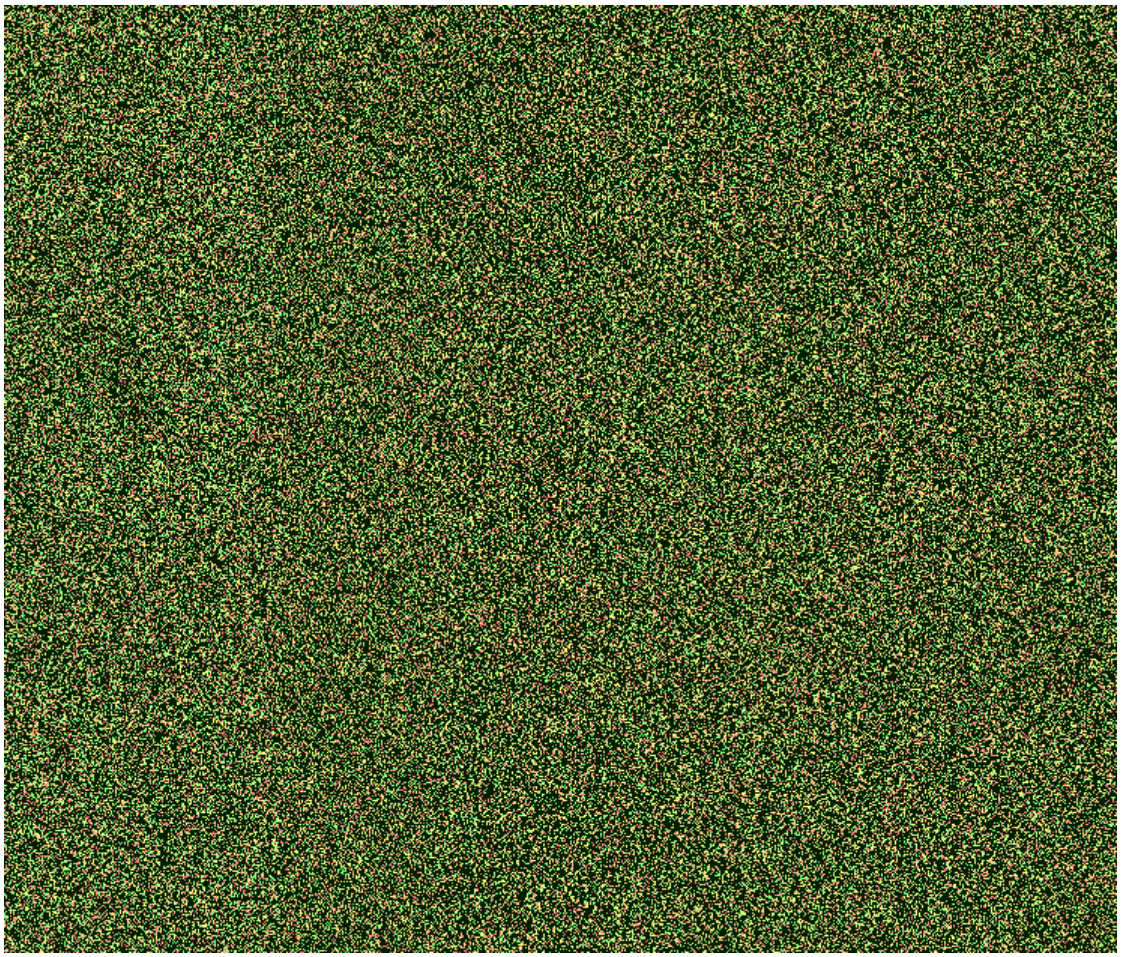


Figure 8.3.: Image comparison of original image compressed/decompressed with lossy ROUND3 by subtracting ROUND3 from original one.
The image is colored for better visibility of the 3 levels of resulting rounding noise: Black = no rounding difference, green shows the lowest, yellow shows the medium and red shows the highest rounding difference.

8.2.3. Compression summary for full frames

With lossless compression alone a compression rate of about 2.4 in best can be achieved using the combination of decorrelation = DECORR_IWT2 and lossless compression = LLC_ARI1.
In addition using a lossy compression of ROUND2 or ROUND3 prior to the lossless one will bring the compression rate to 3.5 or even 4.4 without losing significant scientific image data. Other lossless compression methods beside the ROUNDing will make no sense for full images in SMILE.

Using these combination of compression methods will enable the SDP to compress and transmit a 6x6-binning full image of one CCD every 12 seconds (including some spare time). This can be calculated as follows:
The 6x6-binning full image of one CCD has 950.400 byte. The communication channel to the ground station has a maximum bandwidth of 160 kbit/sec. A compression rate of 4.3 gives about 222.000 byte or 1.776.000 bit. To transmit this amount of data to the ground station will need 11,1 seconds.
So in the normal case about 5 images per minute can be compressed and transmitted for one CCD or 2,5 images per minute for the two CCDs together.

8.3. Event detection data

8.3.1. Using CHEOPS Compression methods

For event detection data only lossless compression makes sense because data have to be restored after compression/decompression without any loss or changes.

Lossless compression as described in *SdpCeDefinitions.h* is also combined with various decorrelation settings. The settings are done as described in chapter 6.

Which kind of lossless compression is used can be parametrizied by setting the values of *EventData_CeKey* for the event detection data in file *CompressCcd.cpp*.

The values for the parameterization are defined in file *SdpCeDefinitions.h*. An example for the parameterization might look like:
unsigned int EventData_CeKey = PRODUCT_EVENTDATA | PREPROC_NONE | LOSSY1_NONE | LOSSY2_NONE | LOSSY3_NONE | DECORR_NONE | LLC_RZIP;

Now varying DECORR_... for decorrelation and LLC_... for lossless compression shows the following results (see table 8.11).

Note that a compression rate lower 1 means an increase of data amount!

Decorrelation	Lossless	compression-time	compression rate	decompression-time
NONE	RZIP	0.0065	1.012	0.0059
	ARI1	0.0082	0.960	0.0087
KEYFR	RZIP	0.0067	1.012	0.0061
	ARI1	0.0078	0.960	0.0088
IWT1	RZIP	0.0071	0.608	0.0060
	ARI1	0.0089	0.831	0.0092
IWT2	RZIP	0.0084	0.608	0.0075
	ARI1	0.0098	0.831	0.0110

Table 8.11.: Comparison of various lossless compressions for event detection data.

For the tests I used an event detection data file *303events.cdx* derived with *EDSim* from file *SXI_single_light_FullNoise_CCD_A_b=6_et=1.0_cr=24.0_ea=500.fits* including 303 detected events which gave an amount of 19392 byte data.

In (Soman M. and Randall G. 2020) about 320 events per second and CCD are estimated for a strong solar wind. So the 303 are meeting these estimations fair enough.

With *EDCompare* I compared the original file with the result of the compression/decompression cycle and found no byte errors for the examples in table 8.11. I sorted out

all kind of the lossless compression alternatives which show any byte errors after the compression/decompression cycle. So the DIFF option for decorrelation shall not be used anyway and also the combinations of IWT1 or IWT2 for decorrelation with PACK16 for compression. They are causing byte-errors after decompression.

For further use I chose the combination of DECORR_NONE for decorrelation and LLC_RZIP for compression which seems to be the best in runtime and compression rate over all.

With this combination I investigated event detection data sets with 186 events showing a compression rate of 0.91, 80 events with 0.85 as compression rate, and 4 events showing a compression rate of 1.11.

So even the best combination gives a compression rate of about 1 or even lower.

Compression of single event detection data

As mentioned in chapter 6 this kind of compression shows no satisfying results.

8.3.2. Compress Event Detection Data using event pixel differencing

The method described in 6.4 shows the best results for compressing event detection data. Compression rates of 1.49 up to 1.58 were achievable for event detection data lists during the tests while most of the testcases (about 90%) show results in an interval of 1.56 to 1.58.

The main advantage of this method is, to compress the header and the pixel-data of an event separately. This opens the possibillity to use separate compression methods costumized to the datastructure of the header and the 5x5-pixel-image.

For a single event the maximum achievable compression rate will be 1.6 in theory (40 bytes in comparison to 64 bytes). Compression depends mostly on the content of the 5x5 pixel imagettes. If it contains only one pixel with a high value (just the event itself), the compression is at its best.

Testmethods are the same as described in capter 8.3.1 using again *EDCompare* for end-to-end result-comparison.

For the operation in the SMILE-SXI this kind of compression should be selected.

8.3.3. Compression summary for event detection data sets

As a conclusion it must be stated, that compression of event detection data sets seems only to make sense using the event pixel differencing scheme introduced in this master's thesis. For all other methods inherited from CHEOPS the compression rates are poor and in most combinations even show an increase of the data amount.

9. Usage of UVIE FlightOS proc_chain

In addition the compression SW should be able to run in the new UVIE FlightOS environment using its features to accelerate the compression as well.

9.1. UVIE FlightOS proc_chain basics

Applications require an operating system to be able to talk to the hardware, other applications and other modules and also for data and task management.
The operating system contains architecture-dependent basic functions such as all the drivers that talk to the hardware and the kernel code, which includes tasks such as memory and task management, timing or communication. A system call interface is provided for the applications in order e.g. to be able to carry out memory or communication commands. The applications sits on top of the OS.

UVIE FlightOS is a new operating system designed and implemented by the Institute of Astronomy of the University of Vienna. It was designed to provide a better OS as the base for SDP offering the opportunity to distribute the SW-tasks over all available cores of the underlying DPU hardware.

UVIE OS has a few extras to make processes for the applications more performant and flexible using a program package called proc_chain which provides a powerful programming interface. You can pack individual functions in your own processing tasks, these are provided with input and output buffers and can then be assembled into a new program sequence by connecting the individual tasks to a processing network.

The advantage is, that tasks can be distributed to different processor cores if some are faster and others need more CPU time. So it is best suited for multi-core processors, but also runs on single-core systems. SMILE SXI has already 2 cores and in the future there certainly will be processors with more cores in space missions.

Figure 9.1 shows the fundamental architecture. At the top is the user space where user applications are set up and executed. The user applications will use the C library functions and also the System Call Interface to interact with the kernel code of the OS. The System Call Interface, the Kernel Code and the Architecture-Dependent Code are forming the Kernel Space.
A detailed description can be found in (Luntzer A. 2017a).

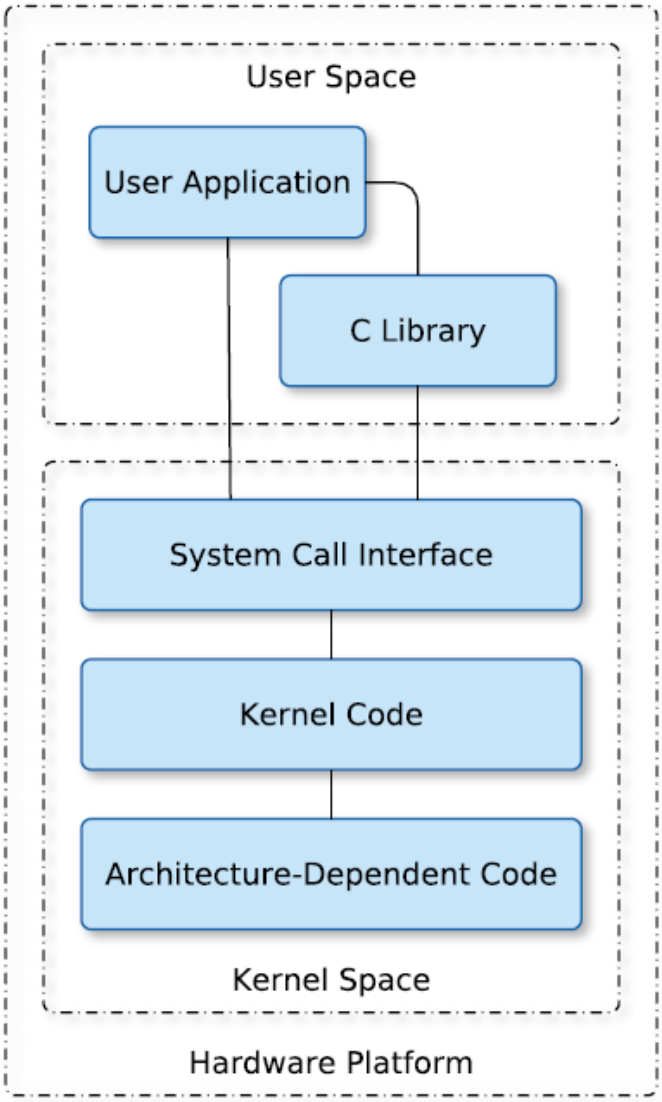


Figure 9.1.: Fundamental architecture of UVIE FlightOS (Luntzer A. 2017a)

The basic idea of UVIE FlightOS is to split up a monolithic program which just represents a fixed series of procedures being executed one after the other into smaller pieces of code which can be run separately controlled by the OS.

In the first step a network is set up which will contain the separated code fragments wrapped up into independent processing tasks (figure 9.2).
A processing task is the fundamental element of the processing network containing a certain functionality represented by a set of program procedures (OP Code).

They are processing the data contained in the Data Buffer. The OS is responsible for the task scheduling and also setting the status of the different OPs from TODO to DONE. The fill status of the Data Buffer is a trigger for the OS performing task-scheduling.

The single processing tasks can now be linked to a processing network. In order to process the tasks fed into the network, nodes must be created, which execute the desired processing steps listed in the task.

These nodes are the processing trackers. Each processing tracker is initialised with an op-code and a function to execute the particular operation (Luntzer A. 2017b). The output buffer of one task can be linked as input buffer for the next task.

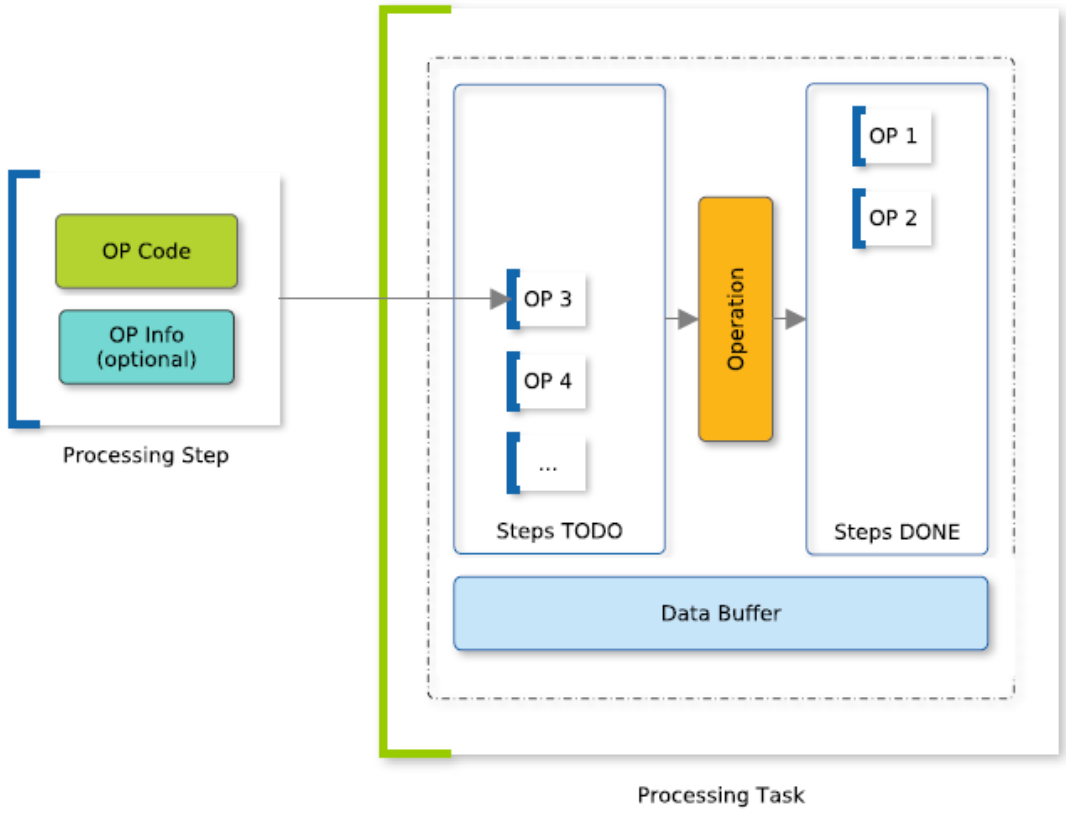


Figure 9.2.: Processing Task (Luntzer A. 2017b)

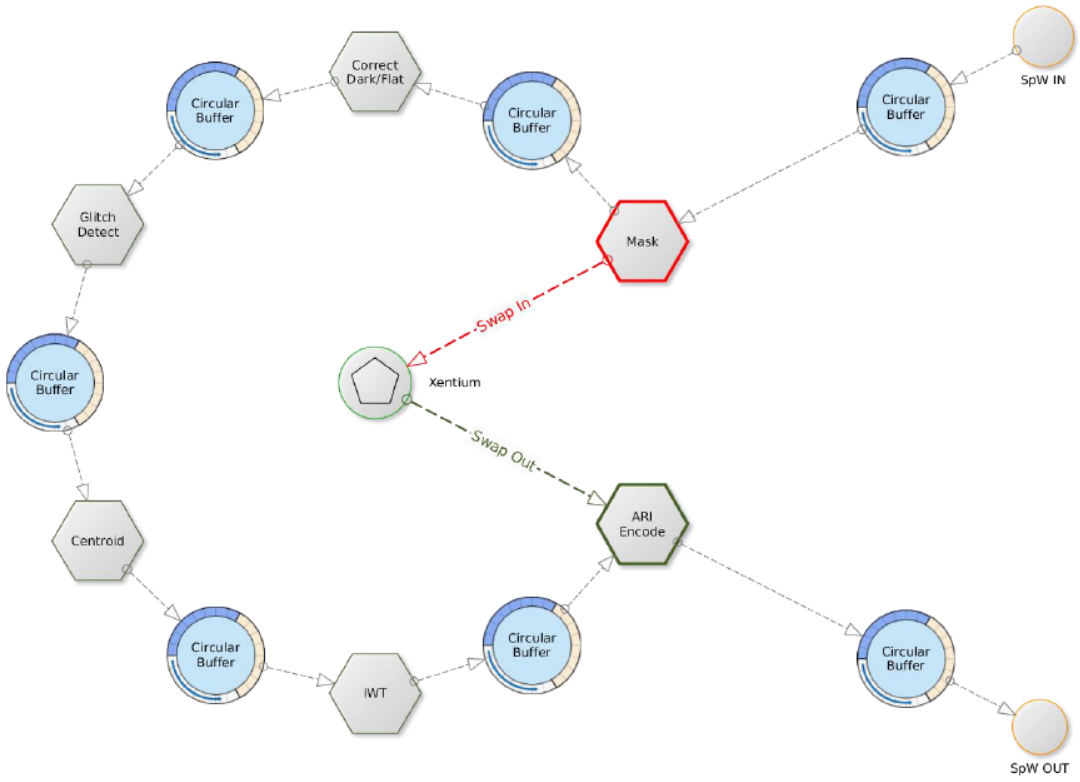


Figure 9.3.: Processing Network (Luntzer A. 2017b)

An example of a network is shown in figure 9.3. In addition a separate Input-task at the starting point of the network and an output task at the end have to be defined.

9.2. A generic network concept

To create and perform the functionality of such kind of data processing networks with UVIE FlightOS the following steps are necessary (for details see (Luntzer A. 2017b)):

- At first a processing network is created by `struct proc_net *pn = pn_create();`.
- In the next step functions are defined which carry the desired features like compression. It is necessary to equip them with the right parameters:
  `int op_Comp (unsigned long op_code, struct proc_task *tsk);` defines for example a function for compression were `op_code` is the operation code of the function and `*tsk` a pointer on the task-structure of the function.
- Within the function it is possible to access the related data buffer by calling:
  `p = (unsigned int *) pt_get_data(t);`

Remark: The data can be of any kind: Integer as in this example, but also char or even complex structures. The pointer p should be declared in the way, that it is a pointer to the chosen data structure.

- Next it is necessary to register the function `op_Comp` as a new tracker by calling:
  `pt = pt_track_create(op_Comp , 0x12345678 , 5);`
  were 0x12345678 is the op-code and 5 is the critical level and add it to the network:
  `pn_add_node(pn, pt);`

- At last it is necessary to create an output node for the network like:
  `int op_output(unsigned long op_code , struct proc_task *t)`
  This output node is added to the network by calling:
  `pn_create_output_node(pn, op_output);`
  and shall forward the processed data to other programs (e. g. via an output buffer or a file) and delocate the used data buffer with `kfree(p);`
  and destroy the now no longer needed task by calling `pt_destroy(t);`

- So far the processing network is defined and it is possible to add some tasks:
  `struct proc_task *t = pt_create (data , datalen, steps , type , seq );`
  were `steps` is the amount of processing steps.
  These steps can be assigned to the task by calling:
  `pt_add_step(t, OP_INC , NULL);`
  were `OP_INC` is the operation code of the step.

- Then the necessary data initialization shall take place and the data shall be set and introduced to the network by:
  `pt_set_data(t, data, sizeof(data));`
  `pt_set_nmemb(t, n);`

- The task is added to the input node of the network:
  `pn_input_task(pn, t);`

- The processing of the network is started with:
  `pn_process_inputs(pn);`
  and the dynamically processing of the tasks is done by repeatedly calling:
  `pn_process_next(pn);`

- At the end all input tasks which have reached the output node are processed by:
  `pn_process_outputs (pn);`
  and the network is destroyed and cleaned up:
  `pn_destroy (pn);`

In (Schmidt K. 2021) a good example for a short program in C can be found showing all the details for a complete processing network.

9.3. SDP improvements for SMILE with proc_chain

The SMILE mission was chosen as the qualification program for UVIE FlightOS so we have here a new OS offering interesting new opportunities for designing and running the DSP.

This new operating system now makes it possible to break up the previously rigid compress chain and to pack the individual steps into separate processing tasks. The OS then takes over the monitoring of how much computing time is allocated to the individual tasks and how they are best distributed over the processor cores.

After the steps have been done as described in the design principles in chapter 6, the program will be redesigned as follows:

- Splitting up the program into new partitions which will fit into the task-concept of UVIE FlightOS.
- Create a processing network as described above.
- Fill the split up components of the SMILE-SDP into the processing network.

9.3.1. Design principles

The idea as already stated is, to split up the compression program into several steps and link them together via a processing network. A working assumption for the single steps could be as follows:

- Input task: Reading the data.
- Set the Compress Entity configuration.
- Prepare and initialize the necessary buffer structures (SIB, CIB, GIB).
- Data Compression.
- Collect data to bit-stream.
- Output task: Write data to CE file and fill GIB.

9.3.2. Analyse runtime length of the single steps

In the first step it is necessary to analyse the runtimes of the single parts of the compression program. Therefore several time intervals are measured by taking timestamps.
It turns out, that for full frames most of the time (about 92 %) is used for the data-compression itself and the rest of about only 8% is used to read the input data, prepare the buffer-structure, collect and write the output data and clean the whole data structures.

To find a useful task splitting, it is necessary to dig deeper into the compression function and find some possibilities for a splitting within this function. Again the dynamic analysis of the program is helpful (see appendix A).

Findings for the constellation of LOSSY3 = ROUND3, DECORR = IWT2, LLC = ARI1:

- Time used for function CompressionEntity::Compress: 91%.
- Time used for CompressStacked (SdpCompress.c) compress full frames: 91%. Therein used times for:
  - Lossy compression: 3%.
  - Decorrelation: 7%.
  - Lossless compression (LLC): 81%.
    * Time used for function fmari_compress within LLC: 15%.
    * Time used for function Checksum_CRC16_32 within LLC: 65%.
- Header-compression needs less than 1% of the time.

The numbers are representing the percentage of the total time amount.
The most time consuming function is *Checksum_CRC16_32* embedded in the lossless compression branch. It needs more than 65% of the total compression time.

To get a complete overview of the time needed for the single steps, table 9.1 shows all relevant numbers. The measurements where taken under the following conditions:

Compressing the full frames:
SXI_single_light_FullNoise_CCD_A_b=6_et=1.0_cr=24.0_ea=500.fits
for 6x6 binning and
SXI_single_light_FullNoise_CCD_A_b=24_et=1.0_cr=24.0_ea=500.fits
for 24x24 binning with LOSSY3 = ROUND3, DECORR = IWT2, LLC = ARI1.

Compressing an event detection date file with 186 events using
LOSSY3 = NONE, DECORR = NONE, LLC = RZIP.

Caution: It should be mentioned that the times can be different in the real DPU-environment because the processor is different and may have a different timing for the several procedure-parts. A final design of the task-splitting should be done after realtime measurements in the DPU.

	6x6 binning	24x24 binning	event detection (186)
Read input data	2%	7%	2%
Compress	91%	85%	78%
Compress data	91%	85%	78%
Lossy3	3%	2%	–
Decorrelation	7%	5%	–
Lossless	81%	77%	75%
Checksum_CRC16_32	65%	62%	47%
fmari_compress / RZip32	15%	15%	28%
Compress header	1%	1%	–
Collect	3%	4%	10%

Table 9.1.: Overview of runtime of single compression steps

Avoiding *Checksum_CRC16_32* leads to the following time distribution (see table 9.2):

	6x6 binning	24x24 binning	event detection (186)
Read input data	7%	13%	3%
Compress	74%	65%	56%
Compress data	73%	65%	56%
Lossy3	7%	4%	–
Decorrelation	18%	10%	–
Lossless	44%	48%	51%
fmari_compress / RZip32	44%	47%	51%
Compress header	1%	1%	–
Collect	8%	10%	18%

Table 9.2.: Runtime of compression steps without *Checksum_CRC16_32*

Function *Checksum_CRC16_32* is the most time consuming one, needed just for testing and can be removed from the program logic in these cases.

It should be mentioned, that printouts during runtime are canceled because they are disturbing the time measurement. See the definitions of DYNPRINT and others in *Sdpprint.h*.

9.3.3. Implementation

A package of C-sources and include-files of proc_chain has to be copied to the src-directory of the ComressionEntity. This includes:

- *data_proc_net.c* and *data_proc_net.h.*
- *data_proc_task.c* and *data_proc_task.h.*
- *data_proc_tracker.c* and *data_proc_tracker.h.*
- *list.h.*

In addition, 2 steps are necessary to combine the compression code with the proc_chain code of UVIE FlightOS:

- Minor changes in proc_chain code: Parts of the compression program is written in C++. On the other hand the proc_chain implementation is written in C-code and uses variable names like "new" which is a keyword in C++. Therefore "new" has to be replaced by "p_new" in *list.h.* Also because the C++ compiler is more restrictive than GCC some additional minor changes like variable conversion for malloc-returns, restrictive type-match or avoiding of unused variables is necessary in *data_proc_tracker.c.*
- A complete redesign of *CompressCcd.cpp* is necessary splitting up the program in several procedures and creating the proc_chain network (see listing which shows the relevant parts of the code).
- In addition the functions *UV_Cmp_...* are collected in a new module *SmileCompressionControl.cpp* for reuse to be able to provide compression in the old style as program CompressCcd and in the new style using the proc_chain as program UVCompressCcd.

The program listings of *SmileCompressionControl.cpp* and *UVCompressCcd.cpp* with its include-file can be found in the git-repository.
Remark: The listing shows only the relevant part for the introduction of proc_chain to the Compression program and not the whole listing of of UVCompressCcd.cpp. The program CompressCcd.cpp isn't listed here separately because it only consists of some lines of code calling the *UV_Cmp_*-functions in the order: Input, Init, Compress, Collect and Output.
Beside the definition of the op-codes and the necessary procedures the include-file contains a small structure including a pointer to the CompressionEntity in use and an identifier for the action to be performed if the timestep granularity should be improved within the task-calls.

The usage of proc_chain is intended only for the Compression part not for the Decompression because compression takes place in the satellite in the UVIE FlightOS environment while decompression is done on ground.

REMARK: Further improvement could be done by enhancing the granularity of the processing steps if necessary due to the runtime split of the single tasks. Most of the time is necessary for the compression itself. The working assumption could be revised in the following way:

- Input task: Reading the data.
- Set the Compress Entity configuration and prepare and initialize the necessary buffer structures (SIB, CIB, GIB).
- Data Compression: All tasks except Lossless Compression and Decorrelation.
- Data Compression: Decorrelation.
- Data Compression: Lossless Compression with fmari_compress or RZip32.
- Collect data to bit-stream.
- Output task: Write data to CE-file and fill GIB.

Even a separate task for Checksum_CRC16_32 can be created because this function is very time consuming.

Additional task-splitup as described above wasn't implemented so far to keep the structure of the single procedures of the Compression modules as they are. Only the main-program of CompressCcd was completely redesigned. For further possible steps see the description of Outlooks in chapter Conclusion.

9.3.4. Make and run programs

The Makefile is the same as before but now includes also the sources for the proc_chain automatically building the programs for compression and decompression. After a successful "make" the programs *UVCompressCcd* and *DecompressCe* can be found in the directory *build*.
Beside the program *UVCompressCcd* in addition a program *CompressCcd* is still created which works the old style not using the proc_chain mechanisms.

10. Additional compression type - Golomb coding

10.1. Intention

In the PLATO project an alternative data compression algorithm using Golomb Rice Coding was implemented and it is of vital interest to evaluate its usage also for SMILE. A detailed description about the usage of the algorithm can be found in (Loidolt D. 2021). The evaluation and usage of the PLATO Golomb Rice Coding will be done in 3 steps:

- Do a first evaluation based on the file interface of the existing PLATO code.
- Implement a standalone fits-file compression and decompression to process the existing SMILE fits-images.
- Incorporate the PLATO Golomb Rice Coding in the SMILE implementation if step 2 shows promising results which should be as good or better than the existing one.

10.2. First evaluation

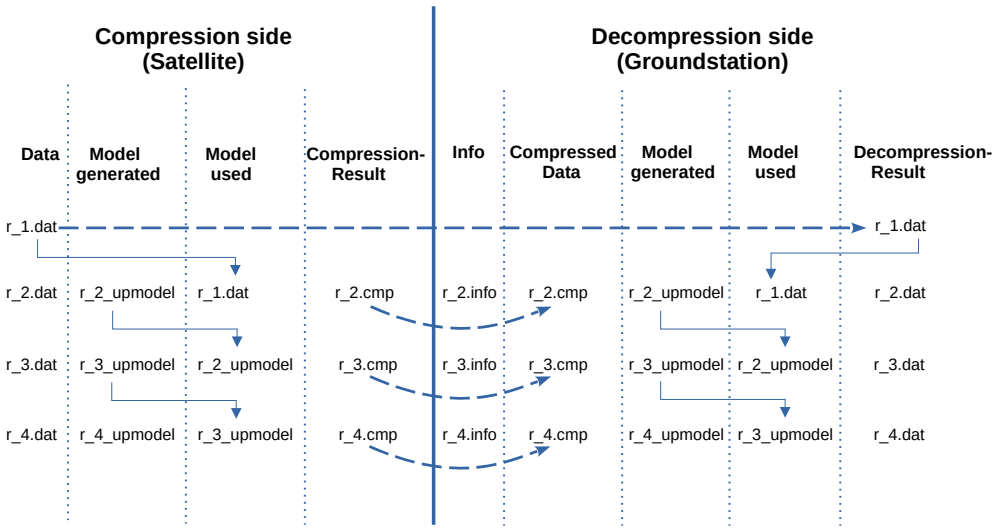


Figure 10.1.: Compression and Decompression methods using PLATO Golomb coding

The basic idea is to compress a fits image row by row using the data of the previous row as model for the next one as shown in figure 10.1.
In a first step the compression rate of the PLATO implementation was evaluated using the already known test-fits-files from chapter 8. These are:

SXI_single_light_FullNoise_CCD_A_b=6_et=1.0_cr=24.0_ea=500.fits
for 6x6 binning and

SXI_single_light_FullNoise_CCD_A_b=24_et=1.0_cr=24.0_ea=500.fits
for 24x24 binning.

SXI_single_light_FullNoise_CCD_A_b=1_et=1.0_cr=24.0_ea=500.fits
for unbinned mode.

The PLATO-implementation uses ASCII-files describing the Hex-content in ASCII-representation separated by blanks like: "AB CD EF 01 23 45 .....".
With a short Python program I derived the image content of the fits-files and brought it into this form. For every single row a separate file is created.

The result is a set of files named XXX_nr.dat from the original fits-file named XXX.fits with "nr" indicating the row-number within the fits-file.
The conversion leads to a bigger amount of data of course which is now the starting point of the evaluation.

10.2.1. Evaluation method

The way is to start with a fits-file, convert it into a set of dat-files which contains the image-row-data as ASCII-HEX, then do the compression and get compressed files and finally decompress them and compare them with the dat-files.

Due to the fact that this is a lossless compression there should be no differences between the original dat-file and the one after the compress/decompress-cycle.
I used the following commands (description see (Loidolt D. 2021)):

./cmp_tool –model_cfg > cfg/default_config_model.cfg
to create a configuration file. Afterwards ensuring that "samples" and "buffer_length" is set to 0 which allowes variable length of input data.

Compression:
For the first row no model is available. Therefore the compression will be done using the 1d-configuration:
./cmp_tool -c cfg/default_config_1d.cfg -d test_data/test24_1.dat -o compressed/test24_1

For the 2^{nd} row the first row test24_1.dat can be used as a model:
./cmp_tool -c cfg/default_config_model.cfg -d test_data/test24_2.dat -m test_data/test24_1.dat -o compressed/test24_2

For compressing rows form the 3^{rd} row onwards the updated model mode can be used:
./cmp_tool -c cfg/default_config_model.cfg -d test_data/test24_3.dat -m compressed/test24_2_upmodel.dat -o compressed/test24_3

Decompression:
For the first row again no model is available:
./cmp_tool -i compressed/test24_1.info -d compressed/test24_1.cmp -o decompressed/test24_1

To decompress the 2^{nd} row the first row is used as a model:
./cmp_tool -i compressed/test24_2.info -d compressed/test24_2.cmp -m decompressed/test24_1.dat -o decompressed/test24_2

Decompressing the rows from the 3^{rd} row onwards the updated model can be used:
./cmp_tool -i compressed/test24_3.info -d compressed/test24_3.cmp -m decompressed/test24_2_upmodel.dat -o decompressed/test24_3

The compression-info is transmitted in a separate file, but will be stored in an info-data-header in a later version.
To compare the files end-to end just a diff command is necessary:
diff test_data/test24_2.dat decompressed/test24_2.dat

10.2.2. Evaluation results

The parameters used in of the evaluation were to use mode 3 (model mode with multi escape symbol mechanism), Golomb parameter = 4 and no rounding.
According to the compression and decompression cycle described above the following results can be reported:

datatype	file-size [byte]		Compression rate	
	original	compressed	real	guessed
examples	301	73	4.12	4.32
unbinned	27025	6330	4.27	4.35
6x6 binning	4501	2720	1.65	2.41
24x24 binning	1123	1290	0.87	1.67

Table 10.1.: Compression results using Golomb coding

As datatypes the example-dataset provided by the author of (Loidolt D. 2021) and files of single pixelrows of full frame images in 6x6 binning mode, 24x24 binning mode and

unbinned were used. For each measure the original file size, the compressed file size, the resulting compression rate and the guessed compression rate are recorded.
Compressed file size and real compression rates are (rounded) average values over a sample of 10 rows for the pixelrow-files.
Using the guess-function of *cmp_tool* it is possible to get a first guess of the compression rate.

10.2.3. Comparison with the Compression methods of CHEOPS

The evaluation of the Golomb code implementation of PLATO in comparison with CHEOPS-arithmetic compression shows the following results (table 10.2):

datatype	Compression rate	
	Golomb/Diff	Arithmetic/Wavelet
unbinned	4.27	4.65
6x6 binning	1.65	2.24
24x24 binning	0.87	3.44

Table 10.2.: Comparison of Golomb code with CHEOPS arithmetic code

The PLATO implementation uses Differentiation as Decorrelation and Golomb code for lossless compression. In the CHEOPS implementation which is also used for SMILE, Decorrelation is done via a Wavelet transformation and for lossless compression an Arithmetic coding is used.
Overall the CHEOPS/SMILE implementation shows better results.

The combination of Wavelet transformation for Decorrelation and Golomb coding for lossless compression is still a field for investigations even when Arithmetic coding shows no difference in compression rates using Differentiation or Wavelet transformation as Decorrelation (see table 8.10 in chapter 8).

Remark: It should be noted that the compression rate highly depends on the content of the data and its entropy. This is also an explanation of the influence of the binning mode. In higher binning modes the content of more pixels is summed up which leads to a bigger difference of the values of the single entries between the image rows and so to a higher entropy of the data. That means higher binning rates are increasing the entropy.
I also "normalized" the intensity of binned images by their binning mode just doing a division. The averaging lowers the entropy. This leads to a bigger compression rate of about 5.1 for 6x6- and 24x24-binned images but such kind of normalization will also lead to a loss of data because the intensity is truncated and can't be reproduced exactly.

10.3. Evaluation of compression and decompression of complete images

In the second step to evaluate the usage for SMILE a standalone testprogram was implemented to verify the efficiency of the Golomb implementation of PLATO for complete SMILE fits-images of all 3 sorts of binning modes.

Program description:

The standalone program works as follows:

- Read a fitsfile.
- Decompose the image into separated rows.
- Compress the image row by row using the concept described in figure 10.1.
- Create a CompressEntity-file containing the compressed image rows and additional info-headers per row.
- Read the CompressEntity-file and create single data structures for every compressed image row.
- Decompress the data row by row.
- Compose the image collecting all the single rows.
- Create an output-fitsfile.

The procedures *compression* and *decompression* are similar to those of the original cmp_tool from PLATO but customized to the necessities of SMILE which also means that no HW-compression is used. From the PLATO-implementation the modules *cmp_icu.c*, *decmp.c*, *cmp_data_types.c* and *cmp_support.c* are used in their original version with slight adaptations to meet c++ compiler requirements like correct type conversions.
In contradiction to the original PLATO implementation the config-data is not read from a separate file but will be defined in the program itself. It should be noted that the first row is treated in a separate way as shown in figure 10.1.
Also the info-data for the single rows are not stored in info-files but within the data-info structures connected to the single rows (see chapter 10.3.1 below).

10.3.1. Data structure

The compression entity of SMILE delivers the CE-structure including the CE-information for later decompression and the CE-data containing the compressed data. This will stay the same for the new Golomb compression type. The only difference is, that the compressed data needs an internal structure because the data-info and length of the compressed data differs from image-row to image-row. Therefore the data-content isn't

flat but structured in the following way:

[image-dimensions from fits-file]
[data-info] [compressed data (variable length)]
[data-info] [compressed data (variable length)]
⋮ ⋮
⋮ ⋮
[data-info] [compressed data (variable length)]

Each row forms a block [data-info][compressed data (variable length)]. The value of the data-length is included in the data-info. The image-dimensions are not necessary for the real implementation in the last step because they can be derived from the decompressed image-header.

10.3.2. Make and run programs

Use the Makefile for building the program. A successful "make" delivers the program *SM_ GC* the SMILE-Golomb-Coding.

To compress and decompress a FITS-file just start:
./SM_ GC FITS-original-file FITS-result-file
FITS-original-file is the name of the FITS-file to be compressed and *FITS-result-file* is the name of recomposed FITS-file at the end of the compress/decompress-cycle.
In addition a file *CE_ Full_ GOLOMB.ce* containing the compressed fits-image is produced which size gives a measure of the compression in relation to the original FITS-file.

With the fits-compare program *F_ comp* (see chapter 7) the original-file and the result-file can be compared.

10.3.3. Comparison with the Compression methods of CHEOPS

Similar to the results collected in table 10.2 the results of the 3 different binning methods for complete fits-files are listed in 10.3:

datatype	Compression rate	
	Golomb/Diff	Arithmetic/Wavelet
unbinned	4.20	4.65
6x6 binning	1.66	2.24
24x24 binning	0.88	3.44

Table 10.3.: Comparison of Golomb code with CHEOPS arithmetic code for fits-files

The figures again confirm that the CHEOPS/SMILE implementation shows better results.

10.4. Adaptation of PLATO Data Compression to SMILE

Looking at the results of the previous chapters 10.2.3 and 10.3.3 it is questionable if an implementation of the Golomb coding in SMILE makes sense. It should be taken into consideration, that the implementation will need additional program code which means additional memory-space in the DPU of the satellite which shall be avoided without additional benefit.
But on the other hand, Golomb coding might be faster in the DPU environment than the Arithmetic coding and it is in any case an interesting add-on to the overall compression portfolio.
For these reasons an additional branch for Golomb coding is inserted into the compression programs.

10.4.1. Design principles

The overall approach for an implementation is as follows:

- Start with a subset of the PLATO implementation including all necessary .h and .c files which was already used for step 2. Only the software compression modules are necessary. These are *cmp_data_types.c*, *cmp_icu.c*, *cmp_support.c*, *decmp.c* and their corresponding .h-files.
- Integrate this subset into SMILE Compression as a new branch.
- While integrating the Golomb coding the data structures of SMILE are used for source and destination of data for the Compression and the Decompression.

General redesign

The integration of Golomb coding derived from the PLATO implementation was done in the following steps (for details see program-listings in the git):

- Include a new define for CC_LLC_GOLOMB into *CrIaDataPool.h*.
- Include a new define for LLC_GOLOMB into *SdpCeDefinitions.h*.
- Add an additional initial value for FullImage_CeKey including LLC_GOLOMB to *CompressCcd.cpp* and *UVCompressCcd.cpp* for test purposes.
- Add a new case to the compression branch for lossless compression in module *SdpAlgorithmsImplementation.c* function *Compress* similar to that one of Arithmetic compression.
- Add a new case to the decompression branch for lossless decompression in module *GsAlgorithmsImplementation.c* function *Decompress* similar to that one of Arithmetic decompression.

- Add two new modules *SdpGolombCompression.c* and *SdpGolombDecompression.c* containing the interface procedures for compression and decompression functions for Golomb coding to use the PLATO Golomb coding implementation like it was implemented in SM_GC but using now the data structures of the SMILE Compression as described above. For details see the module code below.

10.4.2. Comparison with the Compression methods of CHEOPS

Similar to the results collected in table 10.3, the results of the 3 different binning methods for complete fits-files running within the SMILE-compression-program are listed in table 10.4:

datatype	Golomb coding			Arithmetic coding		
	c-rt	c-r	dec-rt	c-rt	c-r	dec-rt
unbinned	3.542	4.176	6.569	4.060	4.364	7.396
6x6 binning	0.115	1.654	0.205	0.127	2.510	0.228
24 x 24 binning	0.012	1.027	0.019	0.012	2.164	0.022

Table 10.4.: Comparison of Golomb code with Arithmetic code running within the SMILE-compression-program
(c-rt: Compression Runtime, c-r: Compression Ratio, dec-rt: Decompression Runtime)

Again the already known test-fits-files from chapter 8 are used. These are:
SXI_single_light_FullNoise_CCD_A_b=1_et=1.0_cr=24.0_ea=500.fits
SXI_single_light_FullNoise_CCD_A_b=6_et=1.0_cr=24.0_ea=500.fits
SXI_single_light_FullNoise_CCD_A_b=24_et=1.0_cr=24.0_ea=500.fits

The Arithmetic coding shows better compression ratios especially for the binned modes while the Golomb coding is a little bit faster.
All in all Arithmetic coding seems to be the benchmark for compression of images.

11. From Standalone Simulations to Integrated Flight Software

So far fullframe fits-images created with DaSi and Event Detection Data created with EDSim processed with the Compression-Entity of SMILE were investigated in this master's thesis to get a base for a well fitting compression algorithm for SMILE. Programs are running standalone and are not embedded in the SMILE real environment.
The whole process environment looks like as shown in figure 11.1.

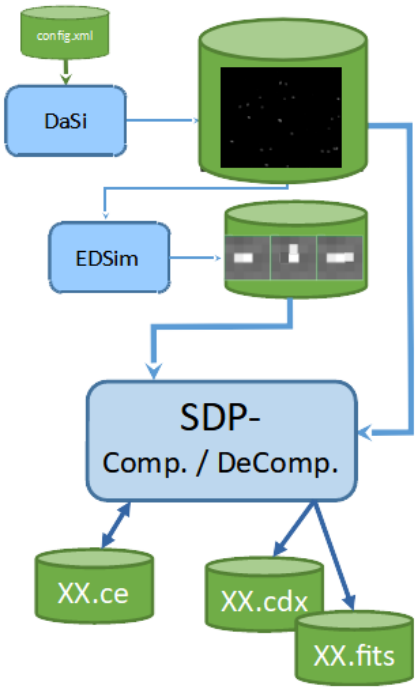


Figure 11.1.: Standalone environment for testing SDP

Tests described in the previous chapters were performed using this environment.
But at the end of the day we need an integrated version of the Science Data Processing embedded in the real environment of SMILE. The SMILE real environment will be available at a very late stage in the project. Therefore it is necessary to test the SMILE Compression by also using an emulation testing environment.

11.1. Integration into FEE-DPU emulation environment

Remembering the SXI structure shown in figure 3.10 in chapter 3 and its SW-layers (see figure 3.11), the SDP-compression is part of the DPU application software, the so called IASW. It has to work together with other parts of the application software and be able to handle the data provided by the FEE for full frame images (binned or unbinned) and events identified by the EDU (Event Detection Unit) of the FEE.

Figure 11.2 shows the emulation environment which has to be implemented to test the SDP-SW. The colors of the several modules are corresponding to the ones in figure 3.10 of chapter 3.

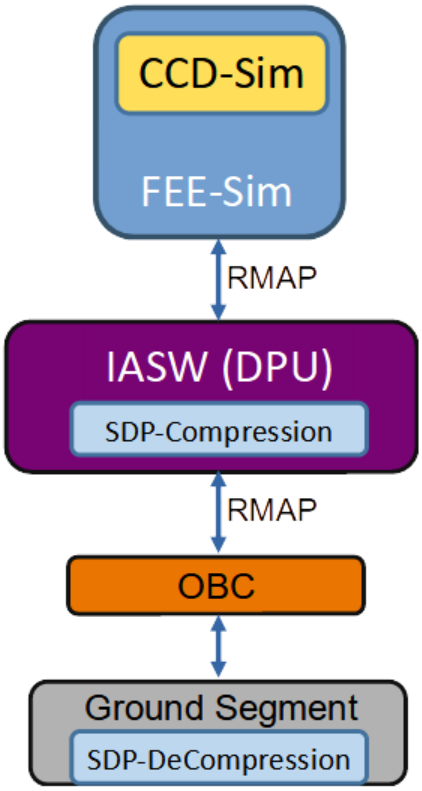


Figure 11.2.: Emulation environment for testing SDP

The emulation environment consists of the following parts:

- FEE-Sim: The FEE-Simulator has an embedded CCD-Simulator which delivers physically more realistic images than the so far used DataSimulator DaSi. FEE-Sim is also able to deal with several Binning modes and Event Detection. Beside the data generation necessary for the Science Data Processing, FEE-Sim can also handle

the requests for Housekeeping Data and Register settings as described in (Smit S.J.A. 2019). RMAP is used for Data communication with the IASW.

- IASW: The Instrument Application Software is described in (Mecina, M. et al. 2020) and (Reimers Ch. 2019). Science Data Processing is a part of it and the data transfer is handled via an internal data interface. The IO-module of the SDP has to be adapted to this interface to replace the fits- and compression-file-IO of the standalone version. The SDP is doing the compression and the compressed data is forwarded to the OBC (On Board Computer) which simulates the Spacecraft computer.

- OBC (On Board Computer): Simulates the Spacecraft computer.

- Ground Segment: Simulates the ground station and contains the Decompression of the SDP.

The emulation environment is not part of this master's thesis and is provided by colleagues from the Institute of Astrophysics (IfA) Vienna. Only the IO-adaptations of the SDP are done be myself. Also emulation tests are not covered by this master's thesis but are part of the overall SMILE project.
Both sides, the SDP-Compression as a part of the IASW and the SDP-Decompression as a part of the Ground Segment SW needed to be tested and therefore both parts also need an interface to be linked to the corresponding environment. Figure 11.3 shows the SW-stacks for both sides:

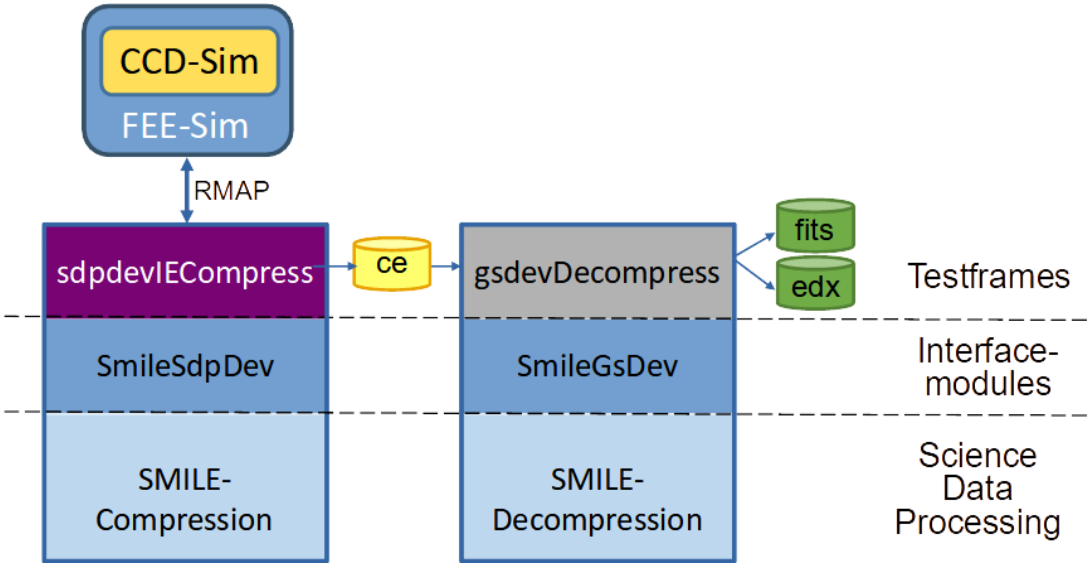


Figure 11.3.: Interfaces and testframes of SDP

Compression

The FEE_Sim delivers images via an RMAP-Interface and the testframe (sdpdev.....) calls the interfacemodule *SmileSdpDev* which links testframes (and later on the IASW) to the SMILE-SDP. Imagedata is forwarded from the testframe to the interface and from the interface to the SDP-Compression and after the compression is done, a datastructure containing the compressed data is handed back to the testframe for further processing.

Decompression

Due to the fact that at the moment there is no Ground-SW or transmit-channel in place, data for and connection to the SMILE-Decompression must be handled in a different way. At first, the sdpdev-testframes will create a compression-file (CE) instead of the transmit channel. Afterwards a gsdev-testframe takes the compression-file, reads the data into an array and then hands over the data to the interfacemodule *SmileGsDev* which links testframe (and later on the Ground-SW) to the SMILE-SDP. Compressed data is forwarded from the testframe to the interface and from the interface to the SDP-Decompression and after the decompression is done, a datastructure containing the decompressed data is handed back to the testframe for further processing. For testing purposes the data is written to a fits-file for images or a cdx-file for event detection data.

11.2. Integrate SMILE-SDP-Compression

As long as the CCD and FEE are not available for the integration tests, the modules of the emulation environment have to replace the real SMILE-SXI modules for testing purposes. A FEE-simulator with an integrated CCD-simulator is available to simulate the FEE, which is called FEE_Sim.
To use FEE_Sim and the related test-framework SDPDEV in combination with the compression program, the following steps are necessary:

- Start the FEE_Simulator in the directory smile_fee-master/SIM with command *./fee_sim* in the background.
- Start the Science Data Processing Device SDPDEV in the directory smile_fee-master/SDPDEV using command *./sdpdev*. This is a testframe communicating with the FEE_Simulator via RMAP and delivers fits-files of the 4 readout-areas of the 2 CCDs.
  The fits-files can be compressed with the SMILE-SDP-Compressions CompressCcd or UVCompressCcd as it was done in the previous chapters. For testing purposes this was done successfully.
- Instead of starting *sdpdev* also a SMILE testframe *sdpdevCompress* can be used which consists of the testframework sdpdev calling the interfacemodule for SDP-compression of SMILE (description see below).
  That means no intermediate fits-files are necessary, data handover goes via an internal

procedure call and the compression of the data is done directly in the program. A pointer to the compressed data will be returned by function *SDP_start_compression* to calling programs like the testframe or also the IASW.

- There are 3 different testframes for SDP-Compression available:
  - sdpdevCompression: The "classical" compression of images.
  - sdpdevUVCompression: The compression of images using UVOS proc_chain.
  - sdpdevIECompression: The compression of images and in addition an event detection simulation and compression of the event data.

Design principles

The necessary data isn't read from a fits-file as it was in the standalone version, but is delivered via a procedure call. Beside that it also means, that CompressCcd.cpp or UVCompressCcd.cpp are no longer the main-programs because the "main"-procedure is part of *sdpdev.c* or a similar program and the start of the compression runs via the mentioned procedure call *SDP_start_compression*. Therefore the following steps are necessary:

- Replace CompressCcd.cpp / UVCompressCcd.cpp and move the *UV_Cmp_...*-functions contained there to a new module *SmileCompressControl.cpp*. In addition create an interface-module *SmileSdpDev.cpp* which connects the main-procedure of *sdpdev* with the Compression functions via *SDP_start_compression*.

- Replace the now unnecessary fits-read-functions by a copy of the relevant data handed over by *sdpdev.c*.

- Run compression as usual for the image.

- *UV_Cmp_Output* gets an additional parameter *fileout* to control the creation of an output-compression-file. This parameter is set only for the standalone-version and not set here for the integrated version.

- The necessary data for compressing an image are now transferred to the compression unit copying it to the relevant SDP-compressiondata:
  - The image data or event detection data to the data-array *fdata* (which was filled with the image data in procedure *ReadSubImages* in the standalone version).
  - The dimensions of the image are copied to *ax* (similar as in *ReadSubImages* in the standalone version). For images the columns are indicating the x-dimension and the rows the y-dimension. For Event Detection Data arrays the eventcounter is relevant.

Program description

Function *SDP_start_compression* is the interface-procedure and is called by the program sdpdev which emulates the IASW-environment.
SDP_start_compression is the "main"-procedure of the compression and calls the *UV_Cmp_...*-functions like the main-program in CompressCcd did.
Only the function *UV_Cmp_Input* is replaced by *SDP_Cmp_Input* because no commandline input is necessary and data isn't read from a fits-file any more. Therefore also procedures *ReadFullFrameImage* and *ReadEventData* are replaced by *ReadDevImage* and *ReadDevEvent* which copies the inputdata to the relevant data-structures of the compression.
The rest of the program runs as in the standalone case.
The compressed data is delivered by *SDP_start_compression* returning a pointer to the structure of the compressed data which can be used by *sdpdev* or later on by the IASW but also must be released (free) by these programs because the memoryspace is allocated by *SDP_start_compression* but not released. To release the allocated memory space, *SmileSdpDev.cpp* offers a function *SDP_destroy_ce_data*. The structure for the compressed data is defined in the include-file *SmileSdpDev.h* containing the datalength and the data itself.

Interfacemodule for Compression

The whole interface procedures for combining the SDP-SW with sdpdev or IASW are collected in a new module *SmileSdpDev.cpp*.

11.3. Integrate SMILE-SDP-Decompression

Program description

In this case only the testframe *gsdevDecompress*, the interfacemodule *SmileGsDev* and the SMILE-Decompression need to be linked together to one program.
Function *GS_start_compression* is the interface-procedure and is called by the program gsdev which emulates the GroundSegment-environment.
GS_start_compression is the "main"-procedure of the decompression returning a pointer to the decompressed datastructure.

For both, the compression- and the decompression-interface and also for the corresponding testframes (and later on for the IASW and Ground-SW) an includefile describing the data structures and interface-procedures is necessary.

11.3.1. Make and run programs

All c-, h-, cpp- and hpp-files from CompressionEntity/src are necessary and in addition the interfacemodules *SmileSdpDev.cpp*, *SmileGsDev.cpp* and the h-file which are also located in this directory. Also module *SmileCompressControl.cpp* which contains the

UV_Cmp_...-functions instead of *CompressCcd.cpp* or UVCompressCcd.cpp is necessary. The example-programs *sdpdevCompress.c*, *sdpdevUVCompress.c* and *sdpdevIECompress.c* for compression and *gsdevDecompress.c* for decompression, located in the directory "examples" of "CompressionEntity" are new.
With the necessary modules of the FEE-Simulator the whole package is compiled and linked to the programs *sdpdevCompress*, *sdpdevUVCompress* which uses UVOS proc_chain as described in chapter 9 and *sdpdevIECompress* which uses procedures from *EDSim* for event data detection (see chapter 5) for the compression-part. For the decompression-part the program *gsdevDecompress* is created.
The Makefile has now extensions for the four new examples-programs.

To run the program system first start the FEE_Simulator in the directory smile_fee-master/SIM with command *./fee_sim* in the background. Then start *sdpdevCompress* or *sdpdevUVCompress*.
Output will be compress-files CE_Full_frame.ce created by *sdpdevCompress* or *sdpdevUVCompress* (not by the compress-functions itself!) for the time being until IASW is available to process the compressed output via the data interface. *sdpdevIECompress* will provide compress-files CE_Event_frame.ce with the event detection data in addition.

Beside the Compression also a prototype for the integration of the Decompression into the Ground-Segment-Environment is available. It uses the Interfacemodule *SmileGsDev.cpp*, the reverse partner of *SmileSdpDev.cpp*.
Also a testframe *gsdevDecompress* is available which uses that Interfacemodule. This testframe simulates the data-input by a Ground-Segment and starts with a compressed-data-file as inputparameter. It delivers a fits-file or an event-detection-data-file depending on the content of the compressed-data-file.

NOTE:
So far this master's thesis provides a complete set of Software for Compression and Decompression for the SMILE mission including also the full framework for testing and emulation integration beside the necessary scientific background.
As an addon to the work done in this master's thesis the following steps in the SMILE project are possible as soon as the necessary prerequisites are in place:

- As soon as the event detection data sets are available by sdpdev or IASW also the compression-interface for this kind of data can be tested.
  At the moment a solution for these tests is implemented using the image-data of sdpdev, create event detection data with procedures derived from EDSim and take that data for compression processing of event detection data (see testprogram *sdpdevIECompress*).

- As soon as the IASW environment is available, also the Compression can be integrated into this program set.

- At the moment no emulation environment for the Decompression is available (Ground-Segment). An integration would be done in a similar way as for the Compression-part.
  At the moment an environment for these kind of tests is available using compress-files as input which are read by a testframe *gsdevDecompress*. This testframe has a data-interface to the *SmileGsDev*-interfacemodule which handles the decompression similar as the compression is done by *SmileSdpDev*.

12. Conclusion

12.1. Results

The SMILE satellite provides a transmission channel to the ground station with a capacity of about 160 kbit/sec. The CCDs have a total amount of 4510 to 4510 pixels each where 4510 to 3791 pixels are used to produce the images and the other rows are just for temporary storage of pixel-data. There are 2 CCDs producing images simultaneously. In the unbinned mode every CCD is creating a full frame image of 4510 x 3791 x 16 bit = 34.195 MByte of data. Both CCDs together give 68.39 MByte.
These 68.39 MByte will need about 3420 seconds or nearly an hour to be transmitted! So additional steps in image processing are necessary:

- First of all, we can assume that unbinned images are transmitted only for calibration purposes which means they are produced and transmitted very seldomly.
- The normal operation mode of the CCDs is 6x6 binned which decreases the data amount per fullframe image down to 1.9 MByte and 95 seconds transmitting time for one image which is still far too long.
- Lossless compression and additional lossy compression reduces data-amount and transmission time by a factor of 2.4 resp. 4.6 resulting in a transmission timeframe of about 22 seconds for a full frame image in 6x6 binning mode.
  The type of data in a SMILE full frame unbinned allowes to achieve a compression rate of about 4.3 with the Golomb coding. This is just a lossless compression in addition and seems to be an improvement for data compression for the unbinned frames but not so for the binned ones, because here the compression rates are poorer than for the arithmetic compression.

But even full frame images are not the normal way SMILE is operating. They will be transmitted from time to time but usually the Event Detection Unit is just providing a set of event detection data per frame where each event needs 64 byte of data.

Beside the necessary adaptations of the compression & decompression used in CHEOPS to fit to the SMILE requirements an additional compression type, the Golomb coding, was implemented and also a special compression type for event detection data, the pixel differentiation, was developed.

The discussion how many events will be detected within one full frame is ongoing at the moment. For a rough estimation 100 events could be taken into account as a mean value, 300 for a strong solar storm (Soman M. and Randall G. 2020).

100 events will give 6400 bytes data for a full frame. To transmit 6.4 kbyte only 0.32 seconds are necessary! Even about 300 events per second for a strong solar wind will result in 19.2 kbyte which can be transmitted within a second. So they might be transmitted without any compression at all. Nevertheless the pixel differencing shows promising compression rates of about 1.56. Only such kind of lossless compression shall be used for event detection data because data shall be recomposed without any losses.

Additional compression methods can be inserted to the compression chain as it was done in this masters thesis with the Golomb coding originally used for the PLATO mission or the method using pixel differences newly introduced in this master's thesis.
So far the Arithmetic coding is the benchmark for images showing the best Compression Rate and for Event Detection Data the pixel-differences-method is the best one.

The usage of UVIE FlightOS proc_chain opens a possibility for accelerating the compression processes in the satellite particularly for HW-processors with more than one core. Therefore proc_chain was successfully introduced to the Compression SW.

To ensure that the compression & decompression of the several data sets will meet the SMILE-requirements a simulator-set was developed including the already in use DaSi and the new Event Detection Simulator EDSim.

Testframes for standalone-tests and also integrated emulation-tests using the FEE-simulator together with new interface-modules are providing a wide range of tests before integrating the compression & decompression into the SMILE real environment later on. As the most efficient method an end-to-end teststrategy was chosen.

Science Data Processing as it was implemented during the work on this master's thesis is a generic approach which can be customized to several space missions and covers the requirements of the SMILE mission at the same time.

12.2. Outlook

For end to end testing without a real satellite SXI/DPU the standalone and also the emulation environment described in chapter 11 can be used.
Last but not least the whole SW-package shall be integrated and tested in the real satellite environment using the already implemented interface-modules which are also used for the emulation tests.

In the areas of compression rate, compression runtime and amount of code further **optimization and customizing** could be done:

- Evaluation of other compression algorithms would be interesting as it was done in this master's thesis for the Golomb code (see chapter 10) or the pixel differencing (see chapter 6.4).
  So far the benchmark for the compression rate for SMILE-images is the Arithmetic coding and for event data the pixel differencing.
- If more than one processor core is available a more granular task design for the Compression SW can further optimize the runtime as described in chapter 9 as a generic approach.
- A first step of code optimization was tried out during the work on this master's thesis as described in chapter 6 clearing the unnecessary functions of the CHEOPS implementation. But within some procedures additional code optimization (e.g. by cleaning the Imagette parts) is still possible.
  The program package is delivered without any optimization to keep all possible compression procedures. Optimization shall be done during the final implementation for the specific mission.

Enhancements by adding other compression algorithms to the already implemented spectrum are possible like it was done with Golomb coding or pixel differencing.

The concept is open to **reuse** in future space missions where image- or data-processing is an issue which is valid for most of or nearly all space missions. Lossless and lossy compression features are already implemented and can easily be adapted to new requirements.

13. List of Acronyms

ASCII	American Standard Code for Information Interchange
BMP	Windows Bitmap
CAS	Chinese Academy of Sciences
CCD	Charge Coupled Device
CCS	Central Checkout System
CHEOPS	CHaracterising ExOPlanet Satellite
CIB	Compression Intermediate Buffer
DaSi	Data Simulator for SMILE
DPU	Data Processing Unit
EDSim	Event Detection Simulator
EDU	Event Detection Unit
ESA	European Space Agency
EUV	Extreme Ultra Violett
FEE	Front End Electronics
FFT	Fast Fourier Transformation
FITS	Flexible Image Transport System
FITSIO	FITS Input Output
GIB	Ground Image Buffer
GIF	Graphics Interchange Format
HDU	Header Data Unit
IASW	Instrument Application Software
IfA	Institute for Astrophysics (Vienna)
IMF	Interplanetary magnetic field
IWF	Institut für Weltraumforschung (Graz)
JPEG	Joint Photographic Experts Group
LIA	Light Ion Analyser
LPC	Linear Predictive Coding
LZAri	Lempel-Ziv Arithmetic
LZH	Lempel-Ziv-Huffman compression
LZMA	Lempel-Ziv-Markow-algorithm
LZO	Lempel-Ziv-Oberhumeralgorithm
LZSS	Lempel-Ziv-Storer-Szymanski
LZW	Lempel-Ziv-Welch algorithm
LZ77	Lempel-Ziv 77

MAG	Magnetometer
MPEG	Moving Picture Experts Group
MPO	Micro Pore Optics
MP3	MPEG audio Layer-3
NASA	National Aeronautics and Space Administration
OBC	On Board Computer
PDF	Probability Density Function
PF	Platform
PKARC	Phil Katz ARC
PLM	Payload Module
PM	Propulsion Module
PNG	Portable Network Graphics
RAW	Raw data format for digital cameras
RMAP	Remote Memory Access Protocol
ROSAT	ROentgen SATellite
SDP	Science Data Processing
SIB	Single Image Buffer
SLR	Single Lens Reflex
SMILE	Solar Wind Magnetosphere Ionosphere Link Explorer
SVM	Service Module
SW	Software
SWCX	Solar Wind Charge Exchange
SXI	Soft X-ray Imager
TGA	Targa Image File Format
UVI	Ultra Violett Imager
UVIE FlightOS	University of Vienna Flight Operating System

Bibliography

Bauernöppel F. 1991, Verfahren und Techniken zur Datenkompression

Borg B. 2020, End-to-end vs integration tests: what's the difference?, https://www.onpathtesting.com/blog/end-to-end-vs-integration-testing

Branduardi-Raymont, G., Agnolon, D., Li, J., & Colangeli, L. 2018, SMILE - Definition Study Report. ESA, 1(1)., https://doi.org/10.5270/esa.smile.definition_study_report-2018-12

Buchanan, B. 1999, Huffman/Lempel-Ziv Compression Methods (Boston, MA: Springer US), 24–31

Bull D. R. & Zhang F. 2018, Golomb Code, https://www.sciencedirect.com/topics/engineering/golomb-code

Carter, J. & Sembay, S. 2008, Identifying XMM-Newton observations affected by solar wind charge exchange - Part I

Dennerl, K. 2010, Space Science Reviews, 157, 57

Dennerl, K. 2013, Unser Sonnensystem im Röntgenlicht, https://www.mpg.de/7787121/mpe_jb_2013

Dennerl, K., Lisse, C. M., Bhardwaj, A., et al. 2012, Solar system X-rays from charge exchange processes

Fei H., Xiao-Xin, Z., Xue-Yi, W., & Bo, C. 2014, EUV emissions from solar wind charge exchange in the Earth's magnetosheath: Three-dimensional global hybrid simulation

Gad A. 2022, Arithmetic Coding, https://neptune.ai/blog/lossless-data-compression-using-arithmetic-encoding-in-python-and-its-applications-in-deep-learning

Gallagher, D. 2002, Structure of the Magnetosphere, https://en.wikipedia.org/wiki/Magnetosphere

Ginzel R. 2010, Wechselwirkung niederenergetischer hochgeladener Ionen mit Materie

Korn D. G. & Vo Kiem-Phong. 2002, Engineering a Differencing and Compression Data Format, https://www.usenix.org/legacy/events/usenix02/full_papers/korn/korn_html/

Kriegl A. . 2003, Lempel-Ziv-Welch, https://www.mat.univie.ac.at/~kriegl/Skripten/CG/node46.html

Kriegl A. 2003, Run-Length Encoding (RLE), https://www.mat.univie.ac.at/~kriegl/Skripten/CG/node44.html

Kuntz, K. D. 2018, Solar wind charge exchange: an astrophysical nuisance

Lang H.W. 1997, Huffman-Code, https://www.inf.hs-flensburg.de/lang/algorithmen/code/huffman/huffman.htm

Lelewer. D. A. & Hirschberg S. 2007, Data Compression, https://www.ics.uci.edu/~dan/pubs/DataCompression.html

Loidolt D. 2021, PLATO Data Compression User Manual

Luntzer A. 2017a, LEANOS Architectural Design Document, https://gitlab.phaidra.org/luntzea4/flightos/-/blob/master/Documentation/OS/architecture/LEANOS-UVIE-ADD-001_Issue_1_0.pdf

Luntzer A. 2017b, LEANOS User Manual, https://gitlab.phaidra.org/luntzea4/flightos/-/blob/master/Documentation/OS/usermanual/LEANOS-UVIE-UM-001_Issue_1_0.pdf

Mecina, M., Ottensamer, R., Reimers, C., et al. 2020, Implementation of the SMILE/SXI Instrument Application Software

Meffert, B. & Hochmuth, O. 2018, Werkzeuge der Signalverarbeitung, 2nd edn. (Humboldt-Univerität zu Berlin)

Ottensamer, R. 2009, Intelligent Detectors: Data Processing of n-Dimensional Detector Arrays"

Ottensamer R. 2018, CHEOPS On-Board Data Processing Steps

Pence, W. D., Chiappetti, L., Page, C. G., Shaw, R. A., & Stobie, E. 2010, Definition of the Flexible Image Transport System (FITS), version 3.0, https://doi.org/10.1051/0004-6361/201015362

Phamdo N. 2000, Lossy compression, https://faculty.uml.edu//jweitzen/16.548/classnotes/Theory%20of%20Data%20Compression.htm

Pu I. 2004, Data compression, https://london.ac.uk/sites/default/files/study-guides/data-compression.pdf

Raymond, J. C., Downs, C., Knight, M. M., et al. 2018, The Astrophysical Journal, 858, 19

Reimers Ch. 2019, SMILE SXI IASW Software Requirement Specification

Russ, J. 2006, The Image Processing Handbook (CRC Press)

Schmidt K. 2021, Python Bindings for the UVIE FlightOS Process Chain

Seelig, J. 2020, A Data's Tale. A look into uVie-IfA's data processing.

Shannon, C. E. 1948, The Bell System Technical Journal, 27, 379

Sibeck, D. G., Allen, R., & Aryan, H. 2018, Imaging Plasma Density Structures in the Soft X-Rays Generated by Solar Wind Charge Exchange with Neutrals

Sibeck, D. 2018, Imaging Plasma Density Structures in the Soft X-Rays Generated by SolarWind Charge Exchange with Neutrals

Sidoriv K. & Marshall D. 2006, Introduction to Compression, https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/PDF/09_Basic_Compression.pdf

SMILE-team. 2018, SMILE Mission Status (ESA)

Smit S.J.A. 2019, SMILE FEE to DPU Interface Requirements Document (IRD)

Soman M. and Randall G. 2020, SMILE SXI CCD Testing and Calibration Event Detection Methodology

Sun, T. R., Wang, C., Sembay, S. F., et al. 2019, Soft X-ray Imaging of the Magnetosheath and Cusps Under Different Solar Wind Conditions: MHD Simulations, https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2018JA026093

testim. 2021, What Is End To End Testing? A Helpful Introductory Guide, https://www.testim.io/blog/end-to-end-testing-guide/

TI. 1997, An Introduction to Fractal Image Compression, https://www.ti.com/lit/an/bpra065/bpra065.pdf

Trümper, J. 1999, ROSAT und seine Nachfolger, https://onlinelibrary.wiley.com/doi/pdf/10.1002/phbl.19990550910

Voigt, H. H. 2012, Abriss der Astronomie, 6th edn. (WILEY-VCH Verlag GmbH)

Wallace, K., Bavdaz, M., Collon, M., et al. 2007, Development of micro-pore optics for x-ray applications, https://sci.esa.int/documents/34490/36224/1567255130848-SPIE-HEO-manuscript-DRAFT.pdf

Weigert, A., Wendker, H., & Wisotzki, L. 2016, Astronomie und Astrophysik, v/4 edn. (WILEY-VCH Verlag GmbH), 177–180

Wiki-GS. 2001, Geomagnetic storm, //https://en.wikipedia.org/wiki/Geomagnetic_storm

zur Nedden, H. 1992, Squeeze, LZH & Co., https://www.schoenbuchsoft.de/Grundlagen/basics/compression/art.htm

A. Compression program calls

	A	B
1	**Module**	**Functions**
2		
3	CompressCcd.cpp	main
4	CrIaDataPool.c	↳InitDataPool
5		↵
6	SmileSimFileIo.c	↳ReadFullFrameImages
7	SmileSimFileIo.c	↳ReadSubImages
8		↵
9	SmileSimFileIo.c	↳cparea
10		↵
11		↵
12	CompressCcd.cpp	↳SetCeConfiguration
13	CrIaDataPool.c	↳CrIa_Paste_PC (multiple calls)
14		↵
15		↵
16	CrIaDataPool.c	↳CrIa_Copy_PC
17		↵
18	CompressionEntity.cpp	↳CompressionEntity
19	SdpBuffers.c	↳SetupCeBuffers
20	SdpBuffers.c	↳swapalloc
21		↵
22	CrIaDataPool.c	↳CrIa_Copy_PC
23		↵
24		↵
25		↵
26	CompressCcd.cpp	↳SetCeSib
27	SdpBuffers.c	↳sdballoc
28		↵
29	GroundSupport.c	↳ConfigSdb
30	CrIaDataPool.c	↳CrIa_Copy_PC (multiple calls)
31		↵
32	SdpBuffers.c	↳sdballoc (multiple calls)
33		↵
34	SdpBuffers.c	↳resalloc (multiple calls)
35		↵
36		↵
37	CrIaDataPool.c	↳CrIa_Copy_PC
38		↵
39	GroundSupport.c	↳InitSdbFull
40	CrIaDataPool.c	↳CrIa_Paste_PC (multiple calls)
41		↵
42		↵
43	GroundSupport.c	↳PrintSdbConfiguration
44	SdpBuffers.c	↳resalloc
45		↵
46	SdpBuffers.c	↳sdballoc
47		↵
48		↵
49		↵
50	SdpBuffers.c	↳InitComprEntStructureFromDataPool
51	CrIaDataPool.c	↳CrIa_Copy_PC
52		↵
53	CrIaDataPool.c	↳CrIa_Paste_PC
54		↵
55		↵
56	IfswAlgorithms.cpp	↳DummyHeaders
57	IfswAlgorithms.cpp	↳DummyCentroid
58	CrIaDataPool.c	↳CrIa_Copy_PC

Figure A.1.: CE-dyn-Comp-1

	A	B	C	D	E	F	G	H	I	J	K	L	M
59	**Module**	**Functions**											
60													
61						↵							
62				↵									
63		↵											
64	CrIaDataPool.c	└CrIa_Copy_PC											
65		↵											
66	IfswAlgorithms.cpp	└IfswAlgorithms::RawHeadersToSib											
67	CrIaDataPool.c		└CrIa_Copy_PC										
68			↵										
69	SdpBuffers.c		└CrIaSetupSibFull										
70	CrIaDataPool.c			└CrIa_Copy_PC									
71				↵									
72			↵										
73	SdpBuffers.c		└updateXib										
74			↵										
75		↵											
76	CrIaDataPool.c	└CrIa_Copy_PC											
77		↵											
78	GroundSupport.c	└FillSib											
79	CrIaDataPool.c		└CrIa_Copy_PC										
80			↵										
81	SdpBuffers.c		└CrIaSetupSibFull										
82	CrIaDataPool.c			└CrIa_Copy_PC									
83				↵									
84			↵										
85	SdpAlgorithmsImplementation.c		└sram2memcpy										
86			↵										
87	EngineeringAlgorithms.c		└Centroiding										
88	CrIaDataPool.c			└CrIa_Copy_PC									
89				↵									
90	EngineeringAlgorithms.c			└getStartCoord									
91				↵									
92	CrIaDataPool.c			└GetFpmSide									
93	CrIaDataPool.c				└CrIaCopyArrayItem								
94					↵								
95				↵									
96	SdpAlgorithmsImplementation.c			└CropCopyFromSibAndUnpack									
97				↵									
98	EngineeringAlgorithms.c			└CheckRoiForStar									
99	CrIaDataPool.c				└CrIa_Copy_PC								
100					↵								
101	SdpAlgorithmsImplementation.c				└MeanSigma								
102					↵								
103	SdpAlgorithmsImplementation.c				└Median								
104					↵								
105	EngineeringAlgorithms.c				└MinMaxU32								
106					↵								
107				↵									
108	SdpAlgorithmsImplementation.c			└saveFitsImage									
109				↵									
110	EngineeringAlgorithms.c			└IntensityWeightedCenterOfGravity2D									
111	EngineeringAlgorithms.c				└WeightedCenterOfGravity2D								
112	CrIaDataPool.c					└CrIa_Copy_PC							
113						↵							
114	SdpAlgorithmsImplementationLlc.c					└bits_used							
115						↵							
116	EngineeringAlgorithms.c					└CenterOfGravity2D							

Figure A.2.: CE-dyn-Comp-2

	A	Functions
117	**Module**	
118		
119		↵
120		↵
121		↵
122	EngineeringAlgorithms.c	↳PrepareSibCentroid
123	CrIaDataPool.c	↳CrIa_Copy_PC
124		↵
125		↵
126		↵
127	SdpBuffers.c	↳updateXib
128		↵
129	CrIaDataPool.c	↳CrIa_Paste_PC
130		↵
131		↵
132	SdpBuffers.c	↳CrIaSetupSibFull
133	CrIaDataPool.c	↳CrIa_Copy_PC
134		↵
135		↵
136	CompressionEntity.cpp	↳CompressionEntity::Compress
137	SdpCompress.c	↳SdpCompress
138	CrIaDataPool.c	↳CrIa_Copy_PC
139		↵
140	SdpBuffers.c	↳SetupCeBuffers
141	SdpBuffers.c	↳swapalloc
142		↵
143	CrIaDataPool.c	↳CrIa_Copy_PC
144		↵
145		↵
146	SdpBuffers.c	↳ForwardProcBuf
147		↵
148	SdpBuffers.c	↳ProcBufBorgMode
149		↵
150	SdpCompress.c	↳CompressStacked
151	CrIaDataPool.c	↳CrIa_Paste_PC
152		↵
153	SdpBuffers.c	↳PrepareSwap
154		↵
155	SdpAlgorithmsImplementation.c	↳Compress
156	CrIaDataPool.c	↳CrIa_Copy_PC
157		↵
158	CrIaDataPool.c	↳CrIa_Paste_PC
159		↵
160	SdpBuffers.c	↳CrIaSetupSibFull
161	CrIaDataPool.c	↳CrIa_Copy_PC
162		↵
163		↵
164	SdpBuffers.c	↳UnpackU16ToU32
165		↵
166	CrIaDataPool.c	↳CrIa_Paste_PC
167		↵
168	SdpAlgorithmsImplementation.c	↳BitRounding32u
169		↵
170	CrIaDataPool.c	↳CrIa_Copy_PC
171		↵
172	CrIaDataPool.c	↳CrIa_Paste_PC
173		↵
174	SdpAlgorithmsImplementation.c	↳IntegerWaveletTransform_Decompose (mult. calls)

Figure A.3.: CE-dyn-Comp-3

	A	Functions
175	**Module**	
176		
177	SdpAlgorithmsImplementation.c	↳IntegerWaveletTransform_Core (mult. calls)
178		↵
179		↵
180	SdpAlgorithmsImplementation.c	↳Transpose2Di
181		↵
182	SdpAlgorithmsImplementation.c	↳Map2Pos32
183		↵
184	CrIaDataPool.c	↳CrIa_Paste_PC
185		↵
186		↵
187	SdpAlgorithmsImplementation.c	↳Checksum_CRC16_32
188		↵
189	SdpAlgorithmsImplementationLlc.c	↳fmari_compress
190	SdpAlgorithmsImplementationLlc.c	↳init_arimodel
191		↵
192	SdpAlgorithmsImplementationLlc.c	↳init_Aritable
193		↵
194	SdpAlgorithmsImplementationLlc.c	↳makeCumulative
195		↵
196	SdpAlgorithmsImplementationLlc.c	↳fmari_encode_chunk
197	SdpAlgorithmsImplementationLlc.c	↳fmari_encodeSym8k (m. c.) LOOP
198		↵ LOOP
199		↳fmari_flushEncoder LOOP
200		↵ LOOP
201		↵
202	SdpAlgorithmsImplementationLlc.c	↳update_fm_ari_model
203		↵
204	CrIaDataPool.c	↳CrIa_Paste_PC
205		↵
206	SdpAlgorithmsImplementationLlc.c	↳vbwl_midsize
207	SdpAlgorithmsImplementationLlc.c	↳bits_used (multiple calls)
208		↵
209		↵
210	SdpAlgorithmsImplementation.c	↳sram2memcpy
211		↵
212		↵
213		↵
214		↵
215	SdpCompress.c	↳CompressHeaders
216	CrIaDataPool.c	↳CrIa_Paste_PC
217		↵
218	SdpBuffers.c	↳PrepareSwap
219		↵
220	SdpAlgorithmsImplementation.c	↳Compress
221	CrIaDataPool.c	↳CrIa_Copy_PC
222		↵
223	CrIaDataPool.c	↳CrIa_Paste_PC
224		↵
225	CrIaDataPool.c	↳CrIa_Copy_PC
226		↵
227	SdpBuffers.c	↳SibToFlatHeaders
228	CrIaDataPool.c	↳CrIa_Copy_PC
229		↵
230		↳CrIaSetupSibFull
231		↵
232		↵
233	CrIaDataPool.c	↳CrIa_Copy_PC

Figure A.4.: CE-dyn-Comp-4

	A	B
234	**Module**	**Functions**
235		
236		↵
237	CrIaDataPool.c	↳CrIa_Paste_PC
238		↵
239	SdpAlgorithmsImplementation.c	↳FrameSubtract
240		↵
241	SdpAlgorithmsImplementation.c	↳Map2Pos32
242		↵
243	CrIaDataPool.c	↳CrIa_Paste_PC
244		↵
245	SdpAlgorithmsImplementation.c	↳Checksum_CRC16_32
246		↵
247	SdpAlgorithmsImplementationLlc.c	↳RZip32
248		↵
249	SdpAlgorithmsImplementation.c	↳sram2memcpy
250		↵
251		↵
252		↵
253		↵
254		↵
255	CrIaDataPool.c	↳CrIa_Copy_PC
256		↵
257	SdpBuffers.c	↳updateXibNTimesFwd
258	SdpBuffers.c	↳updateXibFwd
259		↵
260		↵
261	CrIaDataPool.c	↳CrIa_Paste_PC
262		↵
263	CompressionEntity.cpp	↳CompressionEntity::Collect
264	CrIaDataPool.c	↳CrIa_Copy_PC
265		↵
266	SdpCollect.c	↳SdpCollect
267	CrIaDataPool.c	↳CrIa_Copy_PC
268		↵
269		↵
270	SdpBuffers.c	↳updateXib
271		↵
272	CrIaDataPool.c	↳CrIa_Paste_PC
273		↵
274		↵
275	CrIaDataPool.c	↳CrIa_Copy_PC
276		↵
277	SdpBuffers.c	↳updateXibFwd
278		↵
279	CrIaDataPool.c	↳CrIa_Paste_PC
280		↵
281	CompressionEntity.cpp	↳CompressionEntity::FreeArrays
282		↵
283	CrIaDataPool.c	↳CrIa_Copy_PC
284		↵
285	CrIaDataPool.c	↳CrIa_Paste_PC
286		↵
287	CompressionEntity.cpp	↳CompressionEntity::~CompressionEntity
288	CompressionEntity.cpp	↳CompressionEntity::FreeArrays
289		↵
290		↵
291	SmileSimFileIo.c	↳FreeSemCube
292		↵

Figure A.5.: CE-dyn-Comp-5

B. Decompression program calls

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	**Module**	**Functions**											
2													
3	DecompressCe.cpp	main											
4	CrIaDataPool.c	↳InitDataPool											
5		↵											
6	CompressionEntity.cpp	↳CompressionEntity											
7	SdpBuffers.c		↳SetupCeBuffers										
8	SdpBuffers.c			↳swapalloc									
9				↵									
10	CrIaDataPool.c			↳CrIa_Copy_PC (multiple calls)									
11				↵									
12			↵										
13		↵											
14	CrIaDataPool.c	↳CrIa_Paste_PC (multiple calls)											
15		↵											
16	SdpBuffers.c	↳sdballoc											
17		↵											
18	GroundSupport.c	↳ConfigSdb											
19	CrIaDataPool.c		↳CrIa_Copy_PC (multiple calls)										
20			↵										
21	SdpBuffers.c		↳sdballoc (multiple calls)										
22			↵										
23	SdpBuffers.c		↳resalloc (multiple calls)										
24			↵										
25		↵											
26	CrIaDataPool.c	↳CrIa_Paste_PC											
27		↵											
28	GroundSupport.c	↳InitSdbWin											
29	CrIaDataPool.c		↳CrIa_Paste_PC (multiple calls)										
30			↵										
31		↵											
32	GroundSupport.c	↳PrintSdbConfiguration											
33	SdpBuffers.c		↳resalloc										
34			↵										
35	SdpBuffers.c		↳sdballoc										
36			↵										
37		↵											
38	CompressionEntity.cpp	↳CompressionEntity::Identify											
39	CompressionEntity.cpp		↳CompressionEntity::CheckForCe										
40			↵										
41		↵											
42	CompressionEntity.cpp	↳CompressionEntity::Decompress											
43	CompressionEntity.cpp		↳CompressionEntity::Decompose										
44	SdpBuffers.c			↳SetupDecomprBuffers									
45	CrIaDataPool.c				↳CrIa_Copy_PC (multiple calls)								
46					↵								
47	SdpBuffers.c				↳AllocFromProcBuf (multiple calls)								
48					↵								
49	SdpBuffers.c				↳swapalloc								
50					↵								
51	SdpBuffers.c				↳ForwardProcBuf								
52					↵								
53	SdpBuffers.c			↳ProcBufBorgMode									
54				↵									
55			↵										
56	IfswAlgorithms.cpp		↳IfswAlgorithms::DecompressHeaders										
57	SdpBuffers.c			↳PrepareSwap									
58				↵									

Figure B.1.: CE-dyn-DeComp-1

	A	B	C	D	E	F	G	H	I	J	K	L	M
59	**Module**	**Functions**											
60													
61	SdpAlgorithmsImplementation.c				└sram2memcpy								
62					↵								
63	GsAlgorithmsImplementation.c				└Decompress								
64	CrIaDataPool.c					└CrIa_Copy_PC							
65						↵							
66	GsAlgorithmsImpelmentationLlc.c					└UnRZip32							
67						↵							
68	SdpAlgorithmsImplementation.c					└Checksum_CRC16_32							
69						↵							
70	GsAlgorithmsImplementation.c					└UnMap2Pos32							
71						↵							
72	GsAlgorithmsImplementation.c					└UndoFrameSubtract							
73						↵							
74					↵								
75	SdpAlgorithmsImplementation.c				└Checksum_CRC16_32								
76					↵								
77	SdpAlgorithmsImplementation.c				└sram2memcpy								
78					↵								
79			↵										
80	CompressionEntity.cpp		└CompressionEntity::FlatHeadersToMeterology										
81	CrIaDataPool.c				└CrIa_Copy_PC								
82					↵								
83	IFswConversions.c				└get…. / convert…..							L	
84					↵							O	
85	CrIaDataPool.c				└CrIa_Paste_PC							O	
86					↵							P	
87			↵										
88	IfswAlgorithms.cpp		└IfswAlgorithms::DecompressStacked										
89	SdpBuffers.c				└PrepareSwap								
90					↵								
91	SdpAlgorithmsImplementation.c				└sram2memcpy								
92					↵								
93	GsAlgorithmsImplementation.c				└Decompress								
94	CrIaDataPool.c					└CrIa_Copy_PC							
95						↵							
96	GsAlgorithmsImpelmentationLlc.c					└fmari_decompress							
97	SdpAlgorithmsImplementationLlc.c						└init_arimodel						
98							↵						
99	GsAlgorithmsImpelmentationLlc.c						└unvbwl_midsize						
100							↵						
101	SdpAlgorithmsImplementationLlc.c						└initAriTable						
102							↵						
103	SdpAlgorithmsImplementationLlc.c						└makeCumulative						
104							↵						
105	GsAlgorithmsImpelmentationLlc.c						└fmari_decode_chunk					L	
106	GsAlgorithmsImpelmentationLlc.c							└fmari_decodeSym8k (multiple calls)				O	
107								↵				O	
108							↵					P	
109	SdpAlgorithmsImplementationLlc.c						└update_fm_ari_model						
110							↵						
111						↵							
112	SdpAlgorithmsImplementation.c					└Checksum_CRC16_32							
113						↵							
114	GsAlgorithmsImplementation.c					└UnMap2Pos32							
115						↵							
116	SdpAlgorithmsImplementation.c					└Transpose2Di							

Figure B.2.: CE-dyn-DeComp-2

	A	B	C	D	E	F	G	H	I	J	K	L	M
117	**Module**	**Functions**											
118													
119						↵							
120	GsAlgorithmsImplementation.c					└IntegerWaveletTransform_Inverse_Decompose (mult. calls)							
121	GsAlgorithmsImplementation.c						└IntegerWaveletTransform_Inverse_Core (mult. calls)						
122							↵						
123						↵							
124	GsAlgorithmsImplementation.c					└UnBitRounding32u							
125						↵							
126					↵								
127	SdpAlgorithmsImplementation.c				└Checksum_CRC16_32								
128					↵								
129	SdpAlgorithmsImplementation.c				└sram2memcpy								
130					↵								
131				↵									
132		↵											
133	CompressionEntity.cpp	└CompressionEntity::GetImagettesCropCoordinates											
134	EngineeringAlgorithms.c		└getStartCoord										
135			↵										
136	CrIaDataPool.c		└CrIa_Copy_PC										
137			↵										
138		↵											
139	DecompressCe.cpp	└enterHeaders											
140		↵											
141	DecompressCe.cpp	└enterComprHeaders											
142		↵											
143	CompressionEntity.cpp	└CompressionEntity::~CompressionEntity											
144	CompressionEntity.cpp		└CompressionEntity::FreeArrays										
145			↵										
146		↵											

Figure B.3.: CE-dyn-DeComp-3

C. Directory tree of the implementation

```
SMILE/OBSW
        |
        |-- implementation
        |       |-- DataSim /* Data Simulator to create fits-images
        |       |       |-- frames /* fits images
        |       |
        |       |-- FitsCompare /* Compare FITS-files
        |       |
        |       |-- EDSim /* Data Simulator to create Event Detection Data
        |       |       |-- ev_data /* event detection data sets
        |       |
        |       |-- EDCompare /* Compare event detection data files
        |       |
        |       |-- CompressionEntity /* SMILE Compression
        |       |       |-- src       /* source- and include-files
        |       |       |-- examples/* main-sources for compression and decompression
        |       |       |-- build     /* created run-files
        |       |
        |       |-- UVOS /* Examples for usage of UVIE FlightOS proc_chain
        |       |
        |       |-- PLATO /* Evaluation of Golomb code used in PLATO
        |               |-- cmp_tool-master /* PLATO original code
        |               |-- SM_GC           /* SMILE evaluation
        |
        |-- Master-Doc /* Master thesis
```

D. List of program modules and corresponding include-files

Modules for Science Data Compression and include-files (w/o Golomb and Event difference):

SmileSimFileIo.c	SmileSimFileIo.h, Sdpprint.h
SmileCompressControl.cpp	UV_Compress_data.h, Sdpprint.h
CompressionEntity.cpp	CompressionEntity.hpp, SdpCompressionEntityStructure.h
GroundSupport.c	GroundSupport.h
GsAlgorithmsImplementation.c	GsAlgorithmsImplementation.h
GsAlgorithmsImplementationLlc.c	GsAlgorithmsImplementationLlc.h
IfswAlgorithms.cpp	IfswAlgorithms.hpp
SdpAlgorithmsImplementation.c	SdpAlgorithmsImplementation.h
SdpAlgorithmsImplementationLlc.c	SdpAlgorithmsImplementationLlc.h
SdpBuffers.c	SdpBuffers.h
SdpCollect.c	SdpCollect.h
SdpCompress.c	SdpCompress.h
SdpNlcPhot.c	SdpNlcPhot.h
CrIaDataPool.c	CrIaDataPool.h, CrIaDataPoolId.h
EngineeringAlgorithms.c	EngineeringAlgorithms.h
IfswConversions.c	IfswConversions.h
IfswMath.c	IfswMath.h

Modules for Event difference compression and include-files:

SdpEventDiffCompression.c	SdpEventDiffCompression.h
SdpEventDiffDecompression.c	

Modules for Golomb compression and include-files:

cmp_data_types.c	cmp_data_types.h
cmp_icu.c	cmp_icu.h
cmp_support.c	cmp_support.h, cmp_debug.h, byteorder.h
decmp.c	decmp.h
SdpGolombCompression.c	SdpGolombCompression.h
SdpGolombDecompression.c	SdpGolombDecompression.h

Modules for proc_chain and include-files:

data_proc_net.c	data_proc_net.h, list.h
data_proc_task.c	data_proc_task.h
data_proc_tracker.c	data_proc_tracker.h

D. List of program modules and corresponding include-files

Main modules for standalone versions:

CompressCcd.cpp
DecompressCe.cpp
UVCompressCcd.cpp

Modules of the FEE_Sim-testenvironment and include-files:

gresb.c	gresb.h
smile_fee.c	smile_fee.h
smile_fee_cmd.c	smile_fee_cmd.h, smile_fee_cfg.h
smile_fee_ctrl.c	smile_fee_ctrl.h
smile_fee_rmap.c	smile_fee_rmap.h
rmap.c	rmap.h

Interface-Modules and emulation programs:

SmileSdpDev.cpp SmileSdpDev.h
SmileGsDev.cpp
sdpdevCompress.c, sdpdevUVCompress.c, sdpdevIECompress.c
gsdevDecompress.c

Software Releases:

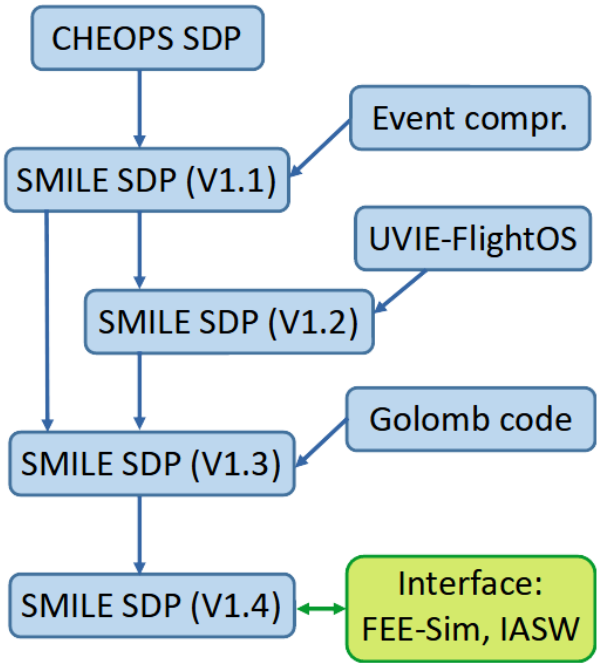


Figure D.1.: Science Data Processing: Software development path