D2.2 Compression Concepts

Anastasiia Novikova, Julian Kunkel, Eugen Betke, Armin Schaare

Workpackage:	WP2 Massive I/O
Responsible institution:	Thomas Dubos
Contributing institutions:	Universität Hamburg, RIKEN, IPSL
Date of submission:	2016-12-20[a]

[a] *Changes: 2017-02-07: A few typos corrected*

Contents

Listings

1 Relation to the Project

This report proposes precise definitions of noise characteristics, compression constraints, and tools for generating and characterizing compression-generated noise. The following text summarizes the project proposal for this task and deliverable:
The goal of this task is to define a set of quantities that can be used for specification of maximum permissible data degradation, and thus, defining the variable accuracy. Examples are relative and absolute error (residual), but there are more quantities that are relevant. The quantities should serve as a basis for:

- *Noise level verification, which shows, if the value is below the desired threshold or not, and if it fulfills the desired constraints.*
- *Generation of synthetic noise for given characteristics, so that users can simulate the effect of data compression and determine which noise level is acceptable for their purpose.*
- *Creation of a user-guided API for data storage with lossy compression.*

The support of (semi-)automatic computation of optimal values (T2.3) shall relieve the user of this mostly cumbersome task.

2 Related Work

Over the past years we could observe the tendency, that the climate scientists are creating increasingly large amounts of data, making the storage of the data to a challenging and expensive task. From the economical standpoint it becomes more and more reasonable to find a way to store data more efficiently. Especially suited for this are data compression methods. In an HPC environment the storage is not the only component that can benefit from compressed data, but other components like network and compute nodes can also do, improving overall application runtimes.
Before we continue our discussion about compression, we take a look how scientists store the data on HPCs. Here, a particularly important role is played by I/O libraries, or file formats, respectively.

2.1 Compression

Data compression or *bit-rate reduction* involves encoding information using fewer bits than the original representation [MMM12]. Reduced storage space consumption is only one advantage of data compression, but it can also reduce network load and speedup data transmission, especially in an HPC environment. We distinguish between three states of data: original, encoded (compressed), and decoded (decompressed). These states are described in detail later.
Compression techniques are divided in two groups: *lossless* and *lossy*. In the lossless compression no information is lost, i.e., the original data can be reconstructed completely. Lossy algorithms compress stronger by reducing the detail level, e.g., by approximating the data or by supressing unnecessary information. To measure how well a compression algorithm can compresses data we take the ratio of the number of bits required to represent the data before compression to the number of bits required after. It is called *compression ratio*.

$$\text{Compression Ratio} = \frac{\text{Data Size}_{\text{Original}}}{\text{Data Size}_{\text{Compressed}}} \tag{1}$$

For example, if the size of an image is 128x128 pixels, that requires 16384 bytes, and its compressed size is 8192 bytes. Thus, compression ratio of algorithm will be 2.
The compression tools and algorithms can be classified into the three groups.

1. Algorithms
   Compression algorithms and tools that allow to set certain accuracy or performance levels.
2. Tools,
   such as noise generation tool, which allows to add noise to the data.
3. HPC-Compression
   Compression in high-performance computing workflows.

2.1.1 Algorithms

GZIP (GNU ZIP) [Gzi] is an example of a lossless compression scheme that is widely used on personal computers. It is not the fastest, but provides a good trade-off between speed and compression ratio. GZIP compression algorithm is based on the DEFLATE algorithm, which is a variation of the lossless LZ77 (Lempel-Ziv 77) algorithm. LZ77 is the first of the Lempel-Ziv compresion algorithms. It has a dictionary-based compressor. The dictionary consists of encoded sequences of the data. The compressor examines the input data through a *sliding window*, which consists of a search sequence with a recently encoded data and a look-ahead sequence with a data to be encoded, and replaces the second occurrence of the sequence by a pointer to the previous one. The basic idea of this algorithm family is shown in the Figure 1 by decoding the triple $< 5, 3, C(r) >$. The first value of this triple is the offset (distance to the begin of match) from the end of data, the second is the lenght of the match, and the last one is the symbol of the end of match. This window is limited by a small sequences of data. The longest match is limited by the length of the look-ahead sequence. Longer matches have to be truncated at a certain arbitrary length. If compression ratio is more important than speed, deflation algorithm defers the selection of matches with a lazy evaluation mechanism and attempts a complete second search even if the first match is already long enough. For the fastest compression modes, new sequences are inserted in the hash table only when no match was found, or when the match is not too long. This saves time since there are both fewer insertions and fewer searches.

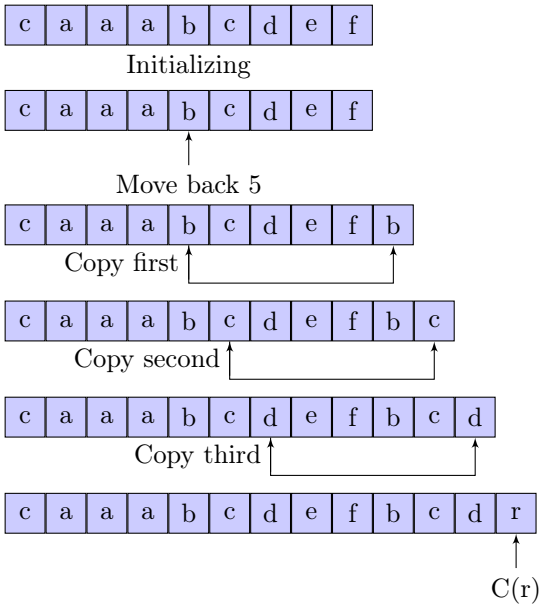
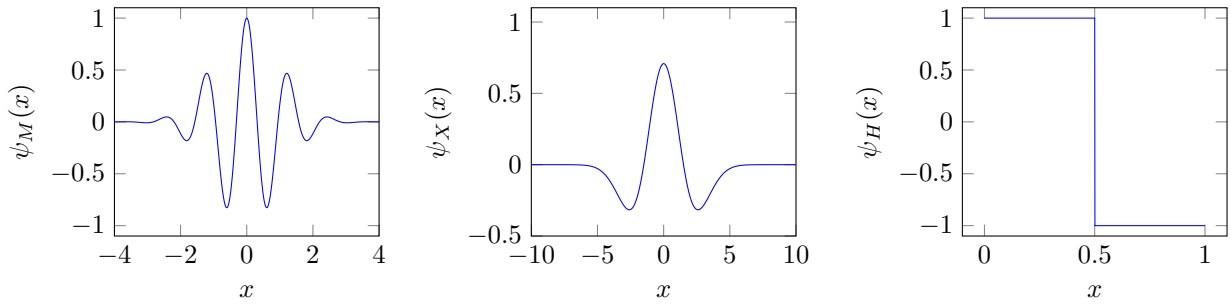


Figure 1: LZ77 encoding

ZFP [Fpc] was designed for compression of floating point data only, but the current version can be also applied to integer data. The algorithm achieves high compression ratios by lossy compression. One of its main features is error-bounded compression, where the maximum permissible relative error can be defined by some epsilon value. ZFP is similar to JPEG compression principle, because it encodes the data, that is very close to zero, and thus uses similarity of the neighboring data points. ZFP is a very fast compressor and decompressor for floating-point data that achieves upto 2 GB/s throughput. In its current release number of dimensions is limited by 3, that is mostly not enough for the climate scientific data.

FPZIP (Floating Point ZIP) [LI06] was primarily designed for lossless compression, but it also supports lossy compression. For lossy compression, ZFP compressor often outperforms FPZIP. This algorithm works as follows. Input data is traversed in some coherent order, e.g. row-by-row, and each visited data value is first predicted from a subset of the already encoded data, i.e. the data available to the decompressor. The predicted and actual values are transformed to an integer representation during which the least significant bits are optionally truncated if lossy compression is desired. Then, residuals are computed and partitioned into entropy codes and raw bits, which are transmitted by the fast entropy coder. Its compression sheme provides high compression rates whithout losts in computational efficiency and delivers high throughput.



(a) Morlet
$\psi_M(x) = e^{-\frac{x^2}{2}} \cos(5x)$

(b) Mexican hat (here $\sigma = 1.5$)
$\psi_X(x) = \frac{2}{\sqrt{3\sigma}\pi^{\frac{1}{4}}}(1 - \frac{x^2}{\sigma^2})e^{\frac{-x^2}{2\sigma^2}}$

(c) Haar wavelet
$\psi_H(x) = \begin{cases} 1 & x \in [0, \frac{1}{2}) \\ -1 & x \in [\frac{1}{2}, 0) \\ 0 & other \end{cases}$

Figure 2: Wavelets

LZ4fast [Lz4] focuses, as an adaption to the Lempel-Ziv algorithm, developed by Yann Collet (2011), on compression and decompression speed while forfeiting compression ratios.

SZ [DC15] is a novel and effective HPC data compression method with primary focus on multi-dimensional floating-point arrays compression. Given a d-dimensional floating-point array, the overall compression procedure can be split into the three steps: (1) conversion of a D-dimensional floating-point array to a 1-dimensional array, (2) compression of the 1-D array by using an array of bits to record the data points whether they can be predicted by the dynamic bestfit curve-fitting models, (3) and compression the unpredictable data by analyzing their binary representations. Experiments show that the compression ratio of a SZ-compressor ranges between 3.3 - 436, which is higher than the second-best solution ZFP by at least 2x and an order of magnitude for most cases. The compression time of SZ is comparable to other solutions, while the decompression time is less than the second best one by 50%-90%. The extreme-scale experiments show that the compression ratio of SZ exceeds that of ZFP by 80%.

WAVELET [Wava] is a C library which contains some utilities for computations involving wavelets. Wavelet is a function ψ, "waving" above and below x-axis. This function approximate with polynoms and trigonometric polynoms (Fourier analysis). Each function can be transformed and recovered with Fourier transformation. The transformed function $\hat{\psi}$ is detailed with frequency parameter ω.
The most important properties of wavelets are:

$$\int_{-\infty}^{\infty} \psi(\mathrm{x})\,\mathrm{d}x = 0 \tag{2}$$

$$\hat{\psi}(0) = 0 \tag{3}$$

$$\int_{-\infty}^{\infty} \frac{|\hat{\psi}(\mathrm{x})|^2}{|\omega|}\,\mathrm{d}\omega < \infty \tag{4}$$

There are many kinds of wavelets. The simplest one is the Haar wavelet (see Figure 2c): It is not continuous and therefore provides a very poor frequency localization.
For the compression, data are coded by wavelet coefficients. Compression using a wavelet transform can be either lossless or lossy. Data will be decomposed with filter banks. The outputs of the filter banks are downsampled, quantized and encoded. The decoder decodes the coded representations, upsamples, and recomposes the signal using a synthesis filter bank. Lossy compression can be done with thresholding by setting small values to 0. Wavelets and wavelet packets can be grown overcomplete (each overcomplete transform is invertible, etc.). The wavelet compression library is portable and can be easily modified by adding new filters.

Wavetrisk [Wavb] Wavetrisk implements a dynamically-adaptive numerical method solving the shallow-water equations (RSWE) on a hierarchy of icosahedral spherical meshes [AKD15]. It brings together a previously developed (non-adaptive) mimetic finite volume / finite difference method for the RSW and spherical wavelet transforms for scalar and vectors that preserve the mimetic properties. Adaptivity especially relies on the ability of wavelets to introduce a controlled amount of error when ignoring wavelet coefficients smaller than a certain threshold. Although Wavetrisk was not initially designed with data compression in mind, building blocks of

Wavetrisk could possibly be reused or re-implemented in order to devise lossy compression schemes tailored for data associated to icosahedral meshes.

Abstol [You10] algorithm was developed alongside SCIL at the Universität Hamburg. In the respective paper it is stated, that Abstol has a uniform scalar quantization method at its core. It is further described, that by separating the interval between the minimum and maximum value of the data into uniform regions, specified by the provided absolute error tolerance, a number of distinct bins is obtained in which each value resides. Each bin is then encoded by its index and used to recreate the original value in a lossy manner [SKri].

Sigbits [SKri] method, like Abstol, was also devised with the implementation of SCIL. In the original paper, it is explained, that Sigbits makes use of properties of the data values float representation and the provided relative error tolerance to generalize over all values. For example, if all values are positive, the sign bit of each value can be dropped and instead only be stored once, in the header of the compressed buffer. Furthermore, Sigbits quantizes the values' exponents and drops non-significant bits in the mantissa as specified by the provided relative error tolerance.

2.1.2 Tools

CNOISE [Sto11] is a C library for generating noise sequences based on $1/f^{\alpha}$ *power spectral density*, where $\alpha > 0$ and f denotes the frequency. This includes white noise ($\alpha = 0$), pink noise ($\alpha = 1$) and red or Brownian noise ($\alpha = 2$), as well as noise for values of α between 0 and 2, by Miroslav Stoyanov [MGB11]. CNOISE is partially based on an algorithm by Kasdin [Kas]. Kasdin's implementation uses a number of functions from the Numerical Recipes library [Nrl] (e.g. FOUR1, FREE_VECTOR, GASDEV, RAN1, REALFT, VECTOR) which is unfortunately a proprietary library whose components cannot be freely distributed. Therefore, the implementation of cnoise relies on the open source library "GSL" (the GNU Scientific Library) [Gnu].
Further features of CNOISE:

- Double precision arithmetic
- In the GSL Fast Fourier Transform functions the input parameter are not restricted to be a multiple of 2.

Noise is a general term from signal processing for some modifications that a signal may suffer during some operations. Noise can be classified by its statistical properties (sometimes called the "color" of the noise) and by how it modifies the intended signal. The color of noise is based on visible electromagnetic spectrum of signal. We are focused on the additive noise, that can be added to the data, that meets the climate scientists' requirements.
Colors of noise (see the Figure 3: sample time series of white (top), pink (middle), and brown (bottom) noise):

- white
  In discrete time, white noise is a discrete signal whose samples are regarded as a sequence of serially uncorrelated random variables with zero mean and finite variance; a single realization of white noise is a random shock. Depending on the context, one may also require that the samples be independent and have identical probability distribution. The samples of a white noise signal may be sequential in time, or arranged along one or more spatial dimensions. A random signal is considered "white noise" if it is observed to have a flat spectrum over the range of frequencies that is relevant to the context [Wikb].

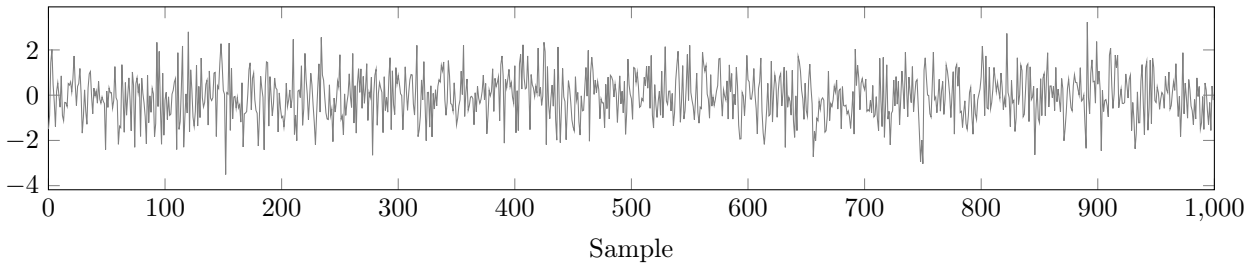
  Listing 1: Example for white noise generation.

```
1 value[time_point] = 2 * ((rand()/((double)RAND_MAX)) - 0.5)
```

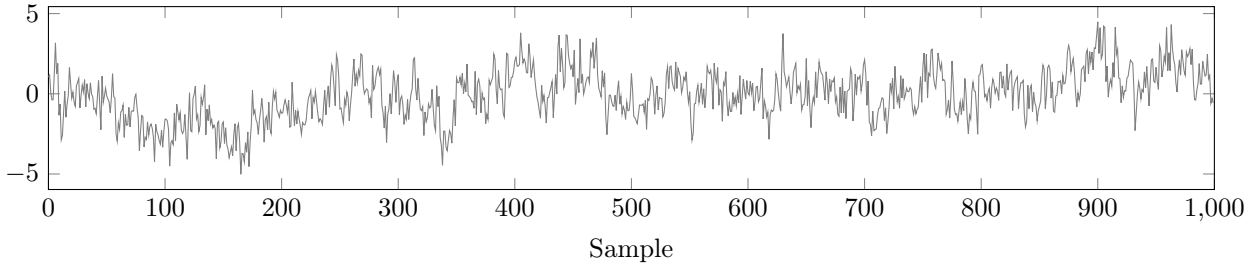
- brown
  Brown noise can be produced by integrating white noise. That is, whereas (digital) white noise can be produced by randomly choosing each sample independently, Brown noise can be produced by adding a random offset to each sample to obtain the next one [Wikb].

- pink
  One parameter of noise, the peak versus average energy contents, or *crest factor*, is important for testing purposes because the signal power is a direct function of the crest factor. Various cres factors of pink noise can be used in simulations of various levels of dynamic range compression. On some digital pink noise generators the crest factor can be specified. White noise will be stronger than pink noise (flicker noise) above some corner frequency. Corner frequency is the frequency either above or below which the

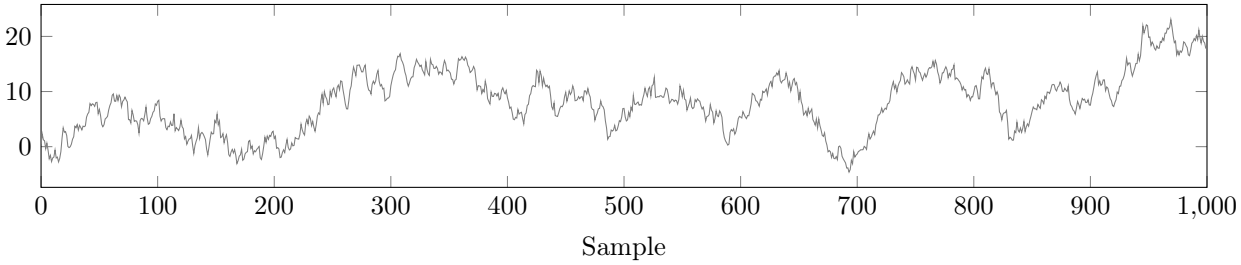
power output of a circuit, such as a line, amplifier, or electronic filter has fallen to a given proportion of the power in the passband [Wikb].



(a) White noise, $\alpha = 0$



(b) Pink noise, $\alpha = 1$



(c) Brown noise, $\alpha = 2$

Figure 3: Noise generated by CNOISE.

2.2 Scientific data

First of all we support climate scientific data. With the progress of computers and increase of observation data, numerical models were developed. A numerical climate model is a mathematical representation of the earth's climate system, that includes the atmosphere, ocean, cryosphere, etc. The model consists of a set of grids with variables such as surface pressure, winds, temperature and humidity. It is needed to predict future evolution of the earth's weather and climate. A numerical model can be encoded in a programming language resulting in an application for simulation the behavior using the model. Grids cover the area of interest.

Many models have the ability to nest finer grids within a coarse grid resulting in a nested grid with much higher resolution. The resolution of a model also depends on the area. Global climate models typically have coarse resolutions and are necessary for long-range forecasts. Regional models with limited area coverage have finer resolution and are used for short-range forecasts. They can be run closer to real time.
The first grid was developed by Lewis Fry Richardson, with idea of forecasting weather by numerical process using physically based models to represent a grid cell as the base of a vertical column of the atmosphere. Each vertical column was then divided into several layers Figure 4. A result was of short-period oscillations called gravity waves that created "noise" in the observed data set.

The effort reached by Richardson is used for future development of grids. For example, on the Figure 5 are shown three shapes of grid: rectangular, triangular and hexagonal.

2.3 File formats

Generally, parallel scientific applications are designed in such a way, they can solve complicated problems faster when running on a large number of compute nodes. This is achieved by splitting a global problem into small

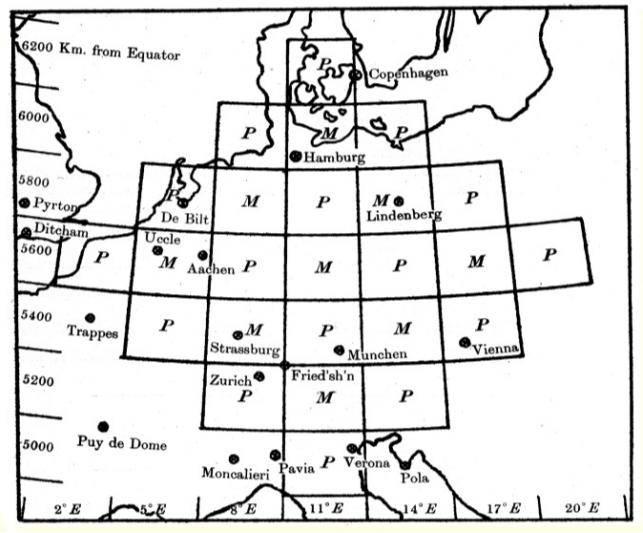


Figure 4: Grid used by Richardson

Figure 5: Types of grids

pieces and distributing it over the compute nodes; this is called domain decomposition. After each node has computed a local solution, they can be aggregated to one global solution. This approach can decrease time-to-solution considerably.

I/O makes this picture more complicated, especially when data is stored in one single file and is accessed by several processes simultaneously. In this case, problems can occur, when several processes access the same file region, e.g. two processes can overwrite the data of each other, or inconsistencies can occur when one process reads, while another writes. Portability is another issue: When transferring data from one platform to another, the contained information should still be accessible and identical. The purpose of I/O libraries is to hide the complexity from scientists, allowing them to concentrate on their research.

Some common file formats are listed in the Table 1. All of these formats are portable (machine independent) and self-describing. Self-describing means, that files can be examined and read by the appropriate software without the knowledge about the structural details of the file. The files may include additional information about the data, called “metadata”. Often, it is textual information about each variable’s contents and units (e.g.,"humidity" and "g/kg") or numerical information describing the coordinates (e.g., time, level, latitude, longitude) that apply to the variables in the file.

GRIB is a record format, NetCDF/HDF/HDF-EOS formats are file formats. It is the difference. In contrast to record format, file formats are bound to format specific rules. For example, all variable names in NetCDF must be unique. In HDF, although, variables with the same name are allowed, they must have different paths. No such rules exist for GRIB. It is just a collection of records (datasets), which can be appended to the file in any order.

GRIB-1 record (aka, ’message’) contains information about two horizontal dimensions (e.g., latitude and longitude) for one time and one level. GRIB-2 allows each record to contain multiple grids and levels for each time. However, there are no rules dictating the order of the collection of GRIB records (e.g, records can be in random chronological order) [Fil].

Name	Fullname	Version	Developer
GRIB1	GRIdded Binary	1	World Meteorological Organization
GRIB2	GRIdded Binary	2	World Meteorological Organization
NetCDF3	Network Common Data Form	3.x	Unidata (UCAR/NCAR)
NetCDF4	Network Common Data Format	4.x	Unidata (UCAR/NCAR
HDF4	Hierarchical Data Format	4.x	NCSA/NASA
HDF4-EOS2	HDF4-Earth Obseving System	2	
HDF5	Hierarchical Data Format	5.x	NCSA/NASA
HDF5-EOS5	HDF5-Earth Obseving System	5	

Table 1: Parallel data formats

Finally, a file format without parallel I/O support, but still worth to mention, is CSV (comma-separated values). It is special due to its simplicity, broad acceptance and support by a wide range of applications. The data is stored as plain text in a table. Each line of the file is a data record. Each record consists of one or more fields, that are separated by commas (hence the name). The CSV file format is not standardized. There are many implementations that support additional features, e.g., other separators and column names.

2.3.1 Example Metadata

Listing 2 gives an example for scientific metadata stored in a NetCDF file. Firstly, between Line 1 and 4, a few dimensions of the multidimensional data are defined. Here there are longitude, latitude with a fixed size and time with a variable size that allows to be extended (appending from a model). Then different variables are defined on one or multiple of the dimensions. The longitude variable provides a measure in "degreees east" and is indexed with the longitude dimension; in that case the variable longitude is an 1D array that contains values for an index between 0-479. It is allowed to define attributes on variables, this scientific metadata can define the semantics of the data and provide information about the data provenance. In our example, the unit for longitude is defined in Line 7. Multidimensional variables such as `sund` (Line 45) are defined on an 2D array of values for the longitude and latitude over various timesteps. The numeric values contain a scale factor and offset that has to be applied when accessing the data; since the data is here stored as short values, it should be converted to floating point data in the application. The `_FillValue` indicates a default value for missing data points.
Finally, global attributes such as indicated in Line 54 ff. describe that this file is written with the NetCDF-CF schema and its history describes how the data has been derived / extracted from original data.

Listing 2: Example NetCDF metadata

```
dimensions:
  longitude = 480 ;
  latitude = 241 ;
  time = UNLIMITED ; // (1096 currently)
variables:
  float longitude(longitude) ;
    longitude:units = "degrees_east" ;
    longitude:long_name = "longitude" ;
  float latitude(latitude) ;
    latitude:units = "degrees_north" ;
    latitude:long_name = "latitude" ;
  int time(time) ;
    time:units = "hours since 1900-01-01 00:00:0.0" ;
    time:long_name = "time" ;
    time:calendar = "gregorian" ;
  short sf(time, latitude, longitude) ;
    sf:scale_factor = 7.3764124573405e-07 ;
    sf:add_offset = 0.0241695530510217 ;
    sf:_FillValue = -32767s ;
    sf:missing_value = -32767s ;
    sf:units = "m of water equivalent" ;
    sf:long_name = "Snowfall" ;
    sf:standard_name = "lwe_thickness_of_snowfall_amount" ;
  short u10(time, latitude, longitude) ;
    u10:scale_factor = 0.00119632889000476 ;
    u10:add_offset = -3.29942438209637 ;
    u10:_FillValue = -32767s ;
    u10:missing_value = -32767s ;
    u10:units = "m s**-1" ;
    u10:long_name = "10 metre U wind component" ;
  short v10(time, latitude, longitude) ;
    v10:scale_factor = 0.00106014321386744 ;
    v10:add_offset = 0.829670123705566 ;
    v10:_FillValue = -32767s ;
    v10:missing_value = -32767s ;
    v10:units = "m s**-1" ;
    v10:long_name = "10 metre V wind component" ;
  short t2m(time, latitude, longitude) ;
    t2m:scale_factor = 0.00203513170666401 ;
    t2m:add_offset = 257.975148205631 ;
    t2m:_FillValue = -32767s ;
    t2m:missing_value = -32767s ;
    t2m:units = "K" ;
    t2m:long_name = "2 metre temperature" ;
  short sund(time, latitude, longitude) ;
    sund:scale_factor = 0.659209863732776 ;
    sund:add_offset = 21599.6703950681 ;
    sund:_FillValue = -32767s ;
    sund:missing_value = -32767s ;
    sund:units = "s" ;
    sund:long_name = "Sunshine duration" ;

// global attributes:
    :Conventions = "CF-1.0" ;
    :history = "2015-06-03 08:02:17 GMT by grib_to_netcdf-1.13.1:
        ↪ grib_to_netcdf /data/data04/scratch/netcdf-atls14-
        ↪ a562cefde8a29a7288fa0b8b7f9413f7-1FD4z9.target -o /data/data04/
        ↪ scratch/netcdf-atls14-a562cefde8a29a7288fa0b8b7f9413f7-CyGl1B.nc
        ↪ -utime" ;
}
```

2.4 Modern File Formats

In modern science, especially, in data intensive research areas like like climatology, meteorology and oceanography, the I/O libraries must fulfulls a number of requirements to ensure, on one hand, convienient data access for scientists, and on the other hand, performant and efficient usage of HPC. NetCDF4 with Climate Forecast

(CF) metadata and GRIB evolved to the de-facto standard formats for convienient data access for the scientists in the domain of NWP and climate.
For convenient data access, modern file formats provide a set of features. For example, metadata can be used to assign names to variables, set units of measure, label dimensions, and provide other useful information. The portability allows data movement between different possibly incompatible platforms, which simplifies the exchange of data and facilitates communication between scientists. The ability to grow and shrink datasets, add new datasets and access small data ranges within datasets simplifies the handling of data a lot. The shared file allows to keep the data in same file. Unfortunately, the last feature conflicts with performance and efficient usage of the state-of-art HPC. The files, which are accessed simultaneously by several processes, cause a lot of syncronisation overhead which slows down the I/O performance. Synchronization is necessary to keep the data consistent.
The rapid development of computational power and storage capacity, and slow development of network bandwidth and I/O performance in the last years resulted in imbalanced HPC systems. The application use the increased computational are able to process more data. More data, in turn, requires more costly storage space, higher network bandwidth and sufficient I/O performance on storage nodes. But due to imbalance, the network and I/O performance are the main bottlenecks. The idea is, to use a part of computational power for compression, improving the overall balance.
Before considering a compression method for HPC, it is a good idea to take a look at the realization of parallel I/O in modern scientific applications. Many of them use the NetCDF4 file format, which, in turn, uses HDF5 under the hood.

HDF (Hierarchical Data Format) is a technology suite that includes several specifications and tools for management of extremely large and complex data collections. In particular, it describes a data model for representation of complex data objects and metadata, specifies the interfaces for C, C++, Fortran 90 and Java, and defines a portable file format. From the beginning, the HDF5 was designed with high scalability in mind. It should be supported by a wide range of devices, from home computer to large-scale HPC. One particulary pleasant characteristic for the latter is that the filesize is not restricted by the standard, but by the current HDF5 implementation (32 ExaByte). If required, the maximum value can be easily extended. HDF5 has a relatively long list of elaborated and powerful features, but it is not our intention to discuss them all in detail. Our object of interest is the compression support and related features.
Note that large parts of the following description has been taken from the official HDF5 documentation[1].

Compression in HDF5 Compression in HDF5 requires chunking. Chunking is the way to store a dataset, wich is characterised as size-fixed and N-dimensional, partitioned into chunks. The dataset is represented as an array upto 32 dimensions and consists of a set of data elements and its description (metadata). It can be stored in the HDF5-file only as 1-dimensional array. A dataset is stored in a file contains header and data array. The header is needed for interpretation of the data portion and metadata, and consists of name of the object, number of its dimensions, data type, information about data is stored and other provided information. On the Signature 1 is shown a function to create a dataset.

Signature 1: HDF5 create an array

```
hid_t H5Dcreate(
  hid_t loc_id,
  const char
  *name,
  hid_t type_id,
  hid_t space_id,
  hid_t dcpl_id )
```

This function creates an empty dataset at the specified location and returns a dataset identifier. It creates a data set with a name `name`, in the file or in the group specified by the identifier `loc_id`. The length of a dataset name is not limited. `name` can be a relative path based at `loc_id` or an absolute path from the root of the file. The dataset's datatype and dataspace are specified by `type_id` and `space_id`. These are the datatype and dataspace of the dataset as it will exist in the file, which may differ from the datatype and dataspace in application memory. H5Dcreate will return an error if a link with the name specified in name already exists at the location specified in `loc_id`. `dcpl_id` is an `H5P_DATASET_CREATE` property list created with H5Pcreate and initialized with various property list functions described in HDF5 manual [Hdfa].

[1] https://support.hdfgroup.org/HDF5/doc/

Chunked dataset can be splitted into multiple chunks, which are separetely stored in the file, and then can be also separately read Signature 2 and written Signature 3.

Signature 2: HDF5 read an array

```
herr_t H5Dread(
  hid_t dataset_id,
  hid_t mem_type_id,
  hid_t mem_space_id,
  hid_t file_space_id,
  hid_t xfer_plist_id,
  void * buf )
```

H5Dread reads a (partial) dataset, specified by its identifier dataset_id, from the file into an application memory buffer buf. Data transfer properties are defined by the argument xfer_plist_id. The memory datatype of the (partial) dataset is identified by the identifier mem_type_id. The part of the dataset to read is defined by mem_space_id and file_space_id. file_space_id is used to specify only the selection within the file dataset's dataspace. Any dataspace specified in file_space_id is ignored by the library and the dataset's dataspace is always used. mem_space_id is used to specify both the memory dataspace and the selection within that dataspace. If raw data storage space has not been allocated for the dataset and a fill value has been defined, the returned buffer buf is filled with the fill value.

Signature 3: HDF5 write an array

```
herr_t H5Dwrite(
   hid_t dataset_id,
   hid_t mem_type_id,
   hid_t mem_space_id,
   hid_t file_space_id,
   hid_t xfer_plist_id,
   const void * buf )
```

This function writes raw data from a buffer to a dataset.

To use chunking a single call have to be done before the dataset is created. In this way we can switch between using chunked and contiguous datasets (see Listing 3).

Listing 3: Sample code for chunking in HDF5.

```
int main(void) {
    hid_t   file_id, dset_id, space_id, dcpl_id;
    hsize_t chunk_dims[2] = {2, 2};
    hsize_t dset_dims[2] = {8, 8};
    int     buffer[8][8];

    //Create the file
    file_id = H5Fcreate(file.h5, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    // Create a property list
    dcpl_id = H5Pcreate(H5P_DATASET_CREATE);

    // Set the property list to use chunking
    H5Pset_chunk(dcpl_id, 2, chunk_dims);

    // Create the dataspace
    pace_id = H5Screate_simple(2, dset_dims, NULL);

    // Create the chunked dataset
    dset_id = H5Dcreate(file, dataset, H5T_NATIVE_INT, space_id, dcpl_id,
    ↪ H5P_DEFAULT);

    // Write to the dataset
    buffer = H5Dwrite(dset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
    ↪ H5P_DEFAULT, buffer);

    // Close processes
    H5Dclose(dset_id);
    H5Sclose(space_id);
    H5Pclose(dcpl_id);
    H5Fclose(file_id);
    return 0;
}
```

Data chunks can pass through user-defined filters on the way to or from disk. The filters operate on chunks of an `H5D_CHUNKED` dataset can be arranged in a pipeline so output of one filter becomes the input of the next filter.
Each filter has a two-byte identification number (type `H5Z_filter_t`) allocated by NCSA and can also be passed application-defined integer resources to control its behavior. Each filter also has an optional ASCII comment string. Two types of filters can be applied to raw data I/O: permanent filters and transient filters. The permanent filter pipeline is defned when the dataset is created while the transient pipeline is defined for each I/O operation. During an `H5Dwrite()` the transient filters are applied first in the order defined and then the permanent filters are applied in the order defined. For an `H5Dread()` the opposite order is used: permanent filters in reverse order, then transient filters in reverse order. An `H5Dread()` must result in the same amount of data for a chunk as the original `H5Dwrite()`.
The permanent filter pipeline is defined by calling `H5Pset_filter()` for a dataset creation property list while the transient filter pipeline is defined by calling that function for a dataset transfer property list.

Signature 4: HDF5 set filter

```
herr_t H5Pset_filter (
  hid_t plist,
  H5Z_filter_t filter,
  unsigned int flags,
  size_t cd_nelmts,
  const unsigned int cd_values[]);
```

This function adds the specified filter and corresponding properties to the end of the transient or permanent output filter pipeline (depending on whether `plist` is a dataset creation or dataset transfer property list). The `flags` argument specifies certain general properties of the filter and is documented below. The `cd_values` is an array of `cd_nelmts` integers which are auxiliary data for the filter. The integer values will be stored in the dataset object header as part of the filter information.

Signature 5: HDF5 get filters

```
int H5Pget_nfilters (hid_t plist);
```

This function returns the number of filters defined in the permanent or transient filter pipeline depending on whether plist is a dataset creation or dataset transfer property list. In each pipeline the filters are numbered from 0 through N-1 where N is the value returned by this function. During output to the file the filters of a pipeline are applied in increasing order (the inverse is true for input). Zero is returned if there are no filters in the pipeline and a negative value is returned for errors.

Signature 6: HDF5 get filter

```
H5Z_filter_t H5Pget_filter (
  hid_t plist,
  int filter_number,
  unsigned int *flags,
  size_t *cd_nelmts,
  unsigned int *cd_values,
  size_t namelen,
  char name[]);
```

This is the query counterpart of H5Pset_filter() and returns information about a particular filter number in a permanent or transient pipeline depending on whether plist is a dataset creation or dataset transfer property list. On input, cd_nelmts indicates the number of entries in the cd_values array allocated by the caller while on exit it contains the number of values defined by the filter. The filter_number should be a value between zero and N-1 as described for H5Pgetn_filters() and the function will return failure (a negative value) if the filter number is out of range. If name is a pointer to an array of at least namelen bytes then the filter name will be copied into that array. The name will be null terminated if the namelen is large enough. The filter name returned will be the name appearing in the file or else the name registered for the filter or else an empty string.

Each filter is bidirectional, handling both input and output to the file, and a flag is passed to the filter to indicate the direction. In either case the filter reads a chunk of data from a buffer, usually performs some sort of transformation on the data, places the result in the same or new buffer, and returns the buffer pointer and size to the caller. If something goes wrong the filter should return zero to indicate a failure. [Hdfb]

NetCDF (Network Common Data Form) is a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data. NetCDF developement started in the year 1988 with the 32-bit file format, called "NetCDF classic", which restricts the variable size to 2 GiB. In 2004 NetCDF classic was extended to NetCDF 64-bit version, which allows variable sizes upto 4 GiB. Although, both legacy file formats are outdated, they are still in use by a number of legacy applications and therefore still supported by the NetCDF Group. To keep up with technological progress, the next upgrade came very soon, in the year 2008. It was the starting point of the general NetCDF4 and the NetCDF4 Classic Model formats. The general NetCDF4 version is a feature rich format with a more complex interface. The NetCDF Classic Model benefits of the most important HDF5 features only (like compression), but provides a simple interface.
NetCDF4 inherited a large set of features from HDF5. They were used to implement the interface, provide extended functionality, and achieve high performance. For example NetCDF variables are represented by HDF5 datasets, and dimensions are HDF5 datasets with a special attribute. Dimensions are attached to variables by HDF5 references. NetCDF benefits by a number of HDF5 features like chunking, compression, and datatypes. Many powerful features are not used like virtual datasets. There is also a lot auf functionaly that is used indirectly, e.g., caching, metadata.
A comparisson of features related to compression is shown in the Table 2.

2.4.0.1 Compression in NetCDF4

With the NetCDF functions nc_def_var_deflate and nc_inq_var_deflate ↪ the compression parameters can be set and retrieved, respectively. They are shown and explained in the Signature 7.

[2] The NetCDF filesize is not limited by the standard, but by HDF5 library implementation.
[3] The HDF5 filesize is not limited by the standard, but by the current HDF5 implementation (max. 32 EiB).
[4] Variables in NetCDF.
[5] In NetCDF Classic last variable can exceed the 2 GiB size limitation.

Feature	NetCDF			HDF5
	Classic	64-bit	NetCDF4	
compresion	no	no	yes	yes
max. file size	8 EiB	8 EiB	unlimited[2]	unlimited[3]
max. dataset[4] size	2 GiB[5]	4 GiB	unlimited	unlimited
signed integer	8-bit 16-bit 32-bit	8-bit 16-bit 32-bit	8-bit 16-bit 32-bit 64-bit	8-bit 16-bit 32-bit 64-bit
unsigned integer			8-bit 16-bit 32-bit 64-bit	8-bit 16-bit 32-bit 64-bit
floating point	32-bit 64-bit	32-bit 64-bit	32-bit 64-bit	32-bit 64-bit + prog. lang. specific + hardware specific
compound	no	no	yes	yes
chunking			yes	yes
other data types				time references
string	no	no	yes	yes
tree depth	1	1	unlimited	unlimited
unlimited dimensions	yes (only 1)	yes (only 1)	yes	yes

Table 2: Comparisson of NetCDF and HDF5

Signature 7: NetCDF compression and decompression function.

```
EXTERNL int nc_def_var_deflate(
  int ncid,
  int varid,
  int shuffle,
  int deflate,
  int deflate_level);

EXTERNL int nc_inq_var_deflate(
  int ncid,
  int varid,
  int *shufflep,
  int *deflatep,
  int *deflate_levelp);
```

`ncid`	Identifier of NetCDF file, returned by `nc_create` or `nc_open`, or of a NetCDF group, returned by `nc_def_grp`.
`varid`	Identifier of a NetCDF variable, returned by `nc_def_var`.
`shuffle`	Turns on the shuffle filter, if non-zero. The shuffle filter can assist with the compression of integer data by changing the byte order in the data stream.
`deflate`	Turns on compression, if non-zero. Set this argument to a non-zero value and set the `deflate_level` argument to the desired compression level.
`deflate_level`	Numeric value between 0 and 9 specifying the amount of compression, where 0 is no compression and 9 is the most compression.

2.4.0.2 NetCDF4 Example The code (Listing 4) creates a 2-dimensional variable and writes the data to the "data.nc" file using compression level 9. First, in Line 11 we declare a 2-D integer array and fill it with values in Line 15 using the scheme `d[x][y] = x + y`. This should create many equal data sequences and enable the deflate algorithm to compress the data with a high compression ratio. Then, in Line 21 and Line 22 we define a NetCDF variable and enable compression. Finally, in Line 24 NetCDF writes the data to the file, compressing the data on the fly.

CPU	Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz
RAM	256GB
Network	InfiniBand FDR
File system	Lustre 2.7.14
Operation System	Red Hat Enterprise Linux Server release 6.8 (Santiago)
Kernel	2.6.32-642.3.1.el6.x86_64

Table 3: Benchmark environment

Listing 4: Sample code for compression in NetCDF.

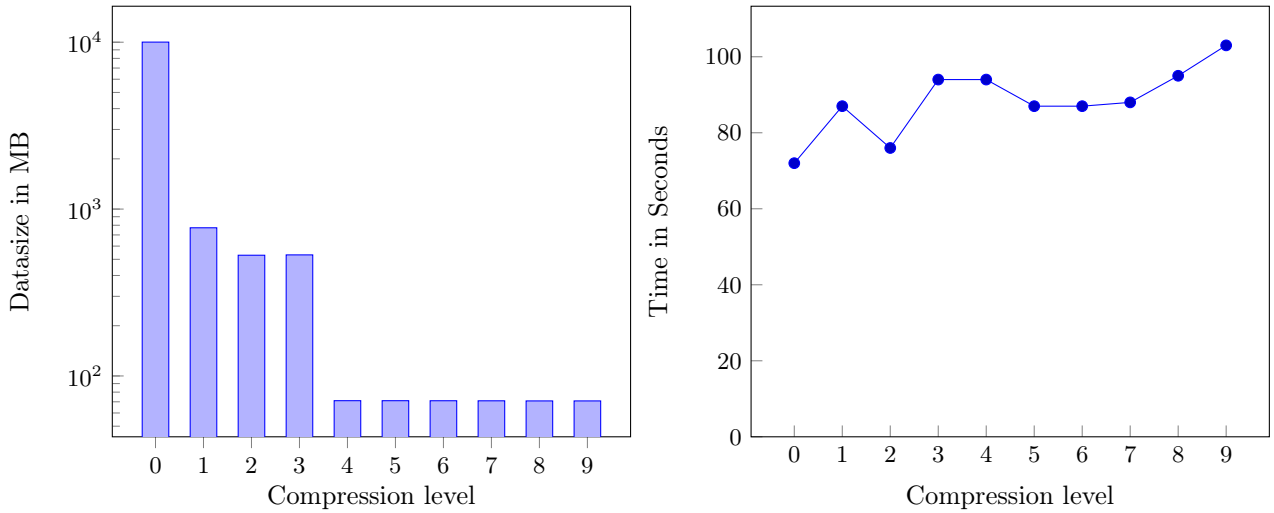
```
#include <stdlib.h>
#include <netcdf.h>

#define NDIMS 2
#define NX 100
#define NY 100

int main(int argc, char** argv) {
   int ncid, x_dimid, y_dimid, varid;
   int dimids[NDIMS];
   int d[NX][NY];
   int x, y, retval;
   for (x = 0; x < NX; x++)
      for (y = 0; y < NY; y++)
         d[x][y] = x + y;
   nc_create("data.nc", NC_NETCDF4 | NC_CLOBBER, &ncid);
   nc_def_dim(ncid, "x", NX, &x_dimid);
   nc_def_dim(ncid, "y", NY, &y_dimid);
   dimids[0] = x_dimid;
   dimids[1] = y_dimid;
   nc_def_var(ncid, "data", NC_INT, NDIMS, dimids, &varid);
   nc_def_var_deflate(ncid, varid, 0, 1, 9);
   nc_enddef(ncid);
   nc_put_var_int(ncid, varid, &d[0][0]);

   nc_close(ncid);
   return 0;
}
```

For the benchmark, we change the code slightly, especially, to be able to allocate large memory chunks and run the benchmark with different parameters, to see how compression ratio and application runtime change, when increasing comression level. The experiment was conducted on Mistral [Mis], the HPC of DKRZ. The relevant characteristics of the benchmark environment are listed in the Table 3 and the results are show in the Figure 6. Obviously, the data generated by the benchmark is particulary suitable for compression. With the first compression level we achieve a compression ratio of 13, and with compression level 4 the compression ratio was 143. After compression level 4 there are no significant improvements. Compared to compression ratio, the benchmark runtime increases slightly with increasing compression level. From these results, following can be deduced: by using several processes we expect to decrease the overall application runtime until the network bandwidth or I/O performance of the storage nodes is saturated.
The file, created by NetCDF4 can be examined by the NetCDF tools, as well as by the HDF5 tools. In Listings 5 to 8 we compared both the output of HDF5 and NetCDF, and compressed and non-compressed files. The complete output of HDF5 was quite verbose, too large to be shown in this report. Therefore, we look only at the dataset and skip dimensions description. In the `Storage_LAYOUT` we can, that chunking feature was automatically enabled.
[raster columns=2,enhanced,equal height group=C]



(a) Compressed datasize *(b)* Runtime

Figure 6: NetCDF compression of a 50000x50000-array of integer values (9.3 GiB) with one single process.

Listing 5: HDF5 dump: Compression disabled

```
$ h5dump -p -A -d data data.nc
HDF5 "data.nc" {
DATASET "data" {
  DATATYPE  H5T_STD_I32LE
  DATASPACE  SIMPLE { ( 100, 100 ) / ( 100, 100 ) }
  STORAGE_LAYOUT {
     CONTIGUOUS
     SIZE 40000
     OFFSET 6192
  }
  FILTERS {
     NONE
  }
  FILLVALUE {
     FILL_TIME H5D_FILL_TIME_IFSET
     VALUE  -2147483647
  }
  ALLOCATION_TIME {
     H5D_ALLOC_TIME_LATE
  }
  ATTRIBUTE "DIMENSION_LIST" {
     DATATYPE  H5T_VLEN { H5T_REFERENCE { H5T_STD_REF_OBJECT }}
     DATASPACE  SIMPLE { ( 2 ) / ( 2 ) }
     DATA {
     (0): (DATASET 239 /x ), (DATASET 514 /y )
     }
  }
}
}
```

Listing 6: HDF5 dump: Compression enabled

```
$ h5dump -p -A -d data data.nc
HDF5 "data.nc" {
DATASET "data" {
  DATATYPE  H5T_STD_I32LE
  DATASPACE  SIMPLE { ( 100, 100 ) / ( 100, 100 ) }
  STORAGE_LAYOUT {
     CHUNKED ( 100, 100 )
     SIZE 638 (62.696:1 COMPRESSION)
  }
  FILTERS {
     COMPRESSION DEFLATE { LEVEL 9 }
  }
  FILLVALUE {
     FILL_TIME H5D_FILL_TIME_IFSET
     VALUE  -2147483647
  }
  ALLOCATION_TIME {
     H5D_ALLOC_TIME_INCR
  }
  ATTRIBUTE "DIMENSION_LIST" {
     DATATYPE  H5T_VLEN { H5T_REFERENCE { H5T_STD_REF_OBJECT }}
     DATASPACE  SIMPLE { ( 2 ) / ( 2 ) }
     DATA {
     (0): (DATASET 239 /x ), (DATASET 514 /y )
     }
  }
}
}
```

[raster columns=2,enhanced,equal height group=D]

Listing 7: NetCDF dump: Compression disabled

```
$ ncdump -s -h data.nc
netcdf data {
dimensions:
  x = 100 ;
  y = 100 ;
variables:
  int data(x, y) ;
    data:_Storage = "contiguous" ;
    data:_Endianness = "little" ;

// global attributes:
    :_Format = "netCDF-4" ;
}
```

Listing 8: NetCDF dump: Compression enabled

```
$ ncdump -s -h data.nc
netcdf data {
dimensions:
  x = 100 ;
  y = 100 ;
variables:
  int data(x, y) ;
    data:_Storage = "chunked" ;
    data:_ChunkSizes = 100, 100 ;
    data:_DeflateLevel = 9 ;
    data:_Endianness = "little" ;

// global attributes:
    :_Format = "netCDF-4" ;
}
```

3 Quantities to Control Accuracy and Realizability

One feature characteristic for lossy compression is that after compression and decompression steps the data can lose precision. The difference between the data states is called *distortion*. Another important term related to precision loss is *accuracy*. It describes the degree to which a given value is correct and free from the error, or how close is the value to the actual value. Another important term is realizability. Realizability refers to physical constraints of the data, e.g., kinetic energy or water vapor mixing ratio being strictly positive. Data that fails realizability conditions may be unsuitable for further analysis and may break post-processing chains. Throughout this section we assume, that data can be of any basic data type, i.e., integer or floating point data of various bit widths. Note that, in contrary to floating point, integer is an exact data type, i.e., as long as the domain of the variable fits, the exact value can be stored. The result of a floating-point calculations often doesn't fit in the variable and must be rounded.

This section is organized as follows. Firstly, we describe the quantities and then, we classify them in two groups, in those that can be applied on individual points and those that depend on their neighbours or some

other values. In addition, we identify critical constraints for the realizability which may reasonably be enforced by lossy compression schemes. Finally, we identify quantities that are especially useful for scientists and/or match our requirements.

3.1 Accuracy of independent points

For a precise mathematical description of quantities, we introduce a number of terms. Suppose, the position of a point is indexed by x (this can be n-dimentional), $d(x)$ is the data on position x, $\hat{d}(x)$ is the value of the compressed data on the point x, then its value be $\hat{d}(x) = f(d(x))$. Note, that some functions such as standard deviation may actually need the full data set to define the compression bounds. Such methods need to operate in two passes on the data: in the first pass the parameters p are defined by a function P, then it is applied: $p = P(d)$, then $\hat{d}(x) = f(d(x), p)$. Result of this function is a vector of parameters. The decompressed value on the point x we can define as $\tilde{d}(x) = f(\hat{d}(x))$.
The following quantities determine compression of each individual point. The example in Figure 12 illustrates how data can be changed, when quantities are applied.

Absolute error tolerance is the maximum amount of the residual error in the calculations, which is defined as $\varepsilon = |\tilde{d}(x) - d(x)|$, then $|\tilde{d}(x) - d(x)| \leq \varepsilon$.

Relative error tolerance is a measure of absolute error compared to the size of the calculations, which is defined as $\eta = \frac{\varepsilon}{|d(x)|}$. Alternatively, it can be written as $\delta = 100\% \cdot \eta$.

Relative error finest absolute tolerance With a relative tolerance, small numbers around 0 are problematic for compressors, e.g. 1% relative error for the data value 0.01 results in the compressed accuracy of 0.01 ± 0.0001. The finest absolute tolerance limits the smallest relative error. In our example, setting a relative error finest absolute tolerance of 0.01 would result in an error of ± 0.01 for small numbers, while for large numbers their relative error is considered. Thus, it is the lower bound and guaranteed error for relative error bounds, where as the absolute tolerance is the guaranteed resolution for all data points.

Precision bits and precision digits indicates how much bits or decimal digits are required to represent the array values.

Excurse: The largest integer value that can be represented by 24 bits is 16777215. It consists of 8 decimal digits, but how can we compute this? This can be done by solving the equation $\lceil \ln(2)/\ln(10) \cdot 24 \rceil = 7.225 \approx 8$. The answer is, all 24-bit binary numbers can be represented by 8-digit decimal numbers.

The largest floating-point value is $(2^1 - 2^{-23}) * 2^{127} \approx 3.402823 * 10^{38}$ and consists of 1 sign bit, 8 bits of exponent and 23 bits of significand precision. It gives from 6 to 9 significant decimal digits precision.

Mean squared error (MSE) is the arithmetic mean of squared errors between uncompressed and original values.

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (\tilde{d}(x_i) - d(x_i))^2 \tag{5}$$

Standard deviation is the squared root of the mean squared error.

$$RMSE = \sqrt{MSE} \tag{6}$$

Average absolute deviation summarises the statistical dispersion or variability.

$$\overline{d} = \frac{1}{N} \sum_{i=1}^{N} \tilde{d}(x) \tag{7}$$

$$\sigma = \frac{1}{N} \sum_{i=1}^{N} |\tilde{d}(x) - \overline{d}| \tag{8}$$

In the general form [Wika] the arithmetic mean $\overline{d}$ can be replaced by some other point, e.g., median or the result of another measure of central tendency.

Peak signal-to-noise ratio (PSNR) is the ratio between the maximum possible power of a signal ($max(d)^2$) and the power of corrupting noise that affects the fidelity of its representation.

$$PSNR = 10 \log_{10} \left(\frac{max(d)^2}{MSE} \right) \tag{9}$$

Preserved values This list contains values that must be preserved literally, i.e., they cannot be changed and must be preserved, i.e., only lossless compression can be applied to those values.

3.2 Accuracy of fields

The accuracy of fields considers the values in the neighborhood of data, e.g., we assume that relevant patterns are preserved after the compression. A field is represented as a grid. Multi-layer grid (multigrid) methods are multiresolution. Multigrid refinement depends on number of layers. Multigrid methods lead to regional different resolutions where on finer resolutions neighooring data points are naturally smoother than on coarser grids. For later analysis it is important to keep relevant features of the field, for example, outstanding high/low values or those that lead to a high gradient.

Interesting quantities are:

- Maximum absolute step change (a_{max}): We assume, the user tolerates a maximum step change between two neighboring data points in the original data, i.e., $\forall$ x in the grid $\forall$ y neighbor of x : $|d(x) - d(y)| \leq a_{max}$. The reason for such a limit might be physical constraints (e.g., this is actually part of realizability) or simply a technical issue that higher differences break subsequent processing steps.

  Thus, this property should be preserved after the compression, i.e., $\forall$ x in the grid $\forall$ y neighbor of x : $|\hat{d}(x) - \hat{d}(y)| \leq a_{max}$

- Conservation of the sum (σ_{con}): This value defines the maximum tolerable value for the sum of the values for the compressed data and the sum of the values for the original data ($|\sum_x d(x) - \sum_x \hat{d}(x)| \leq \sigma_{con}$). For some fields, analysis may require very small errors on the domain-integral of the field. This typically happens when this field represents the local amount of a quantity (water, carbon dioxide) whose global amount is close to equilibrium and therefore changes very slowly. Errors in that total amount translate into errors of its time derivative which must be small compared to global sources and sinks involved in its budget, themselves possibly small.

3.3 Realizability of independent points

Physical data obeys in principle many realizability conditions. Post-processing chains may implicitly rely on such conditions and fail when they are not met. Realizability conditions may be quite complex, e.g. the pressure, density and temperature of air stored in a file should be related through an equation of state, or the determinant of a self-correlation matrix should be positive. However only the simplest conditions may reasonably be enforced by a lossy compression method. Especially since compression is typically a single-field operation, realizability conditions involving several fields are out of scope. Furthermore it does not seem necessary to consider realizability conditions that many numerical schemes violate. Conversely if many numerical schemes are designed to satisfy a certain condition, this points to the importance of that condition and suggests that enforcing it during lossy compression is desirable.
At this time we consider only one realizability condition:

- Positivity. A primary reason to consider it is its simplicity. A second compelling reason is that most transport schemes in current use guarantee the positivity of the transported fields (mixing ratios of water species, chemical species or aerosols).

4 Data Generated by Simulations

Inside an application, a grid is used to describe the covered surfaces of the model, which often is the globe. Traditionally, the globe has been divided based on the longitude and latitude into rectangular boxes. Since this produced unevenly sized boxes and singularities closer to the poles, modern climate applications use hexagonal and triangular meshes. Particularly triangular meshes have an additional advantage, that one can refine regions and, thus, can decide on the granularity that is needed locally – this leads to numeric approaches of the multi-grid methods. Grids that follow a regular pattern such as rectangular boxes or simple hexagonal grids are called

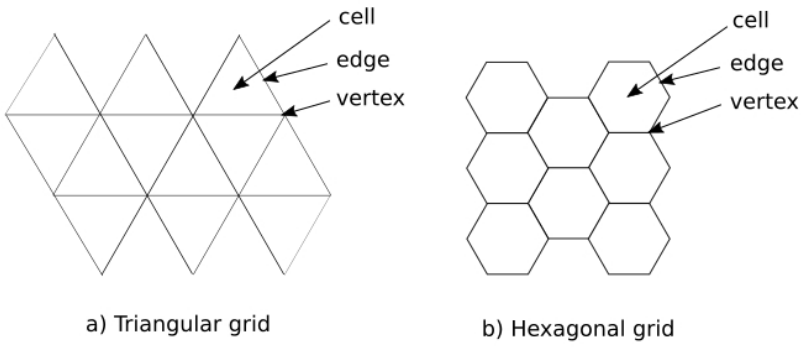


Figure 7: Scope of variables inside the grids

structured grids. With partially refined grids or when covering complex shapes instead of the globe, the grids become unstructured as they form an irregular pattern.
To create an hexagonal or triangular grid from the surface of the earth, the earth can be considered to be an icosaheder where each side can be refined. Variables contain data that can either describes a single value for each cell, the edges of the cells, or the vertexes of the cells.
Figure 7 shows this localization – the scope of data – for the triangular and hexagonal grids. In some applications, it is useful to use dummy (so called fill-) values to encode invalid data regions. An example are water temperature on the globe – data on land cells could be encoded with 30,000.
Larger grids are shown in Figure 9 and in (Figure 8). There are figures provided that illustrate the neighborhood between data points and for different data localization.

4.0.0.1 Hexagonal grid consists of cells shaped as a flat topped hexagon (Figure 8a). Two ways can be used to map data to the grid: vertical or horizontal. Values can be located at the centers of the primal grid (hexagons Figure 8b), and if we connect it to each other, we would see the grid of triangles Figure 8c. If values are located at the edges Figure 8d and they are connected with their neighbours, then the grid will be seen Figure 8e. If the values are located at the vertices and they are connected with their neighbours, then the grid will be seen (Figure 8f).

4.0.0.2 Triangular grid consists of cells shaped as a triangle (Figure 9a). It's structure is similar to hexagonal grid. Values can be located at the centers of the primal grid hexagons Figure 9b, and if we connect it to each other, we would see the grid of triangles Figure 9c. If values are located at the edes (Figure 9d) and they are connected with its neighbours, then the grid will be seen Figure 9e. If the values are located at the vertices and they are connected with its neighbours, then the grid will be seen in Figure 9f.

4.1 Addressing Data

In a programming language, regular grids can usally be addressed by n-dimensional arrays. Thus, a 2D array can be used to store the data of a regular 2D longitude/latitude-based grid.
However, storing irregular grids is not so trivial. For example, a 1D array can be used to hold the data but then the index has to be determined. Staying by our 2D example, to map a 2D coordinate onto the 1D array, a mapping between the 2D coordinate and the 1D index has to be found. One strategy to provide the mapping are space-filing curves. They have the advantage that the indexes of points that close together in the coordinates are also close together – thus is beneficial as often operations are conducted on neighboring data (stencil operations, for example). A hilbert curve is an example for one possible enumeration of a multi-dimensional space.

4.1.0.1 Hilbert curve is a continuous space-filing curve, that helps to represent a grid as n-dimensional-array of values. To visualize its behavior, a 2D grid is shown in Figure 10. In 2D, the basic element of the Hilbert curve is a square with one open side. Every such square has two end-points, and each of these can be the entry-point or the exit-point. So, there are four possible varieties of open side. A first order Hilbert curve consists of one basic element. It is a 2x2 grid. The second order Hilbert curve replaces this element by four (smaller) basic elements, which are linked together by three joins (4x4 grid). Every next order repeats the process by replacing each element by four smaller elements and three joins (8x8 grid).
On the Figure 11 is represented 5th level Hilbert curve for the 256x256 data, that is mapped to 32x32 grid.

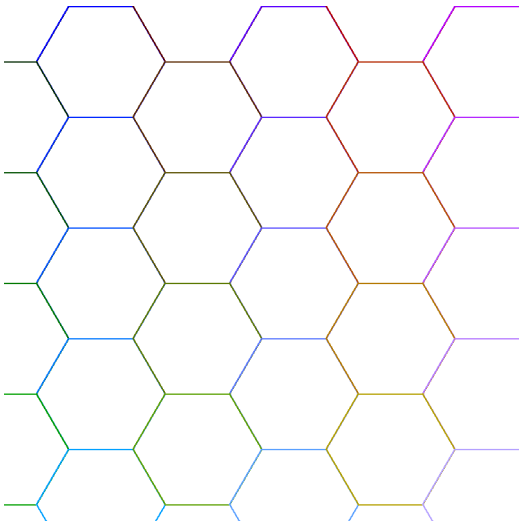
The characteristics of Hilbert curve can be extended to more than two dimensions. The first step figure can be wrapped up in so many dimensions as it is needed and the points neighbours will be always saved.

4.2 Quantities not related to data quality

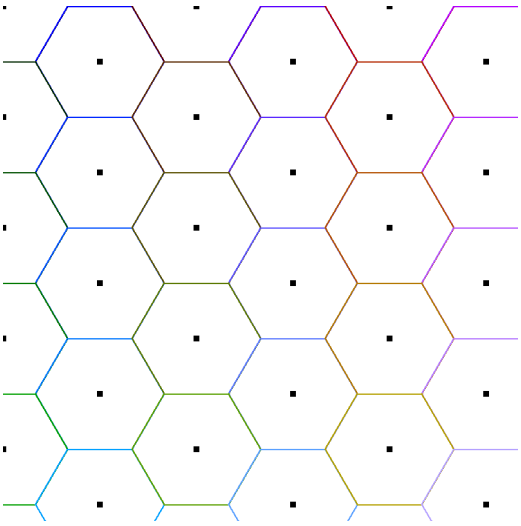
An other important quantity is the compression/decompression speed. When set, the compression/decompression throughput will be limited to this value, otherwise a default will be used, to achieve maximum error tolerance.

4.3 Selection of Quantities

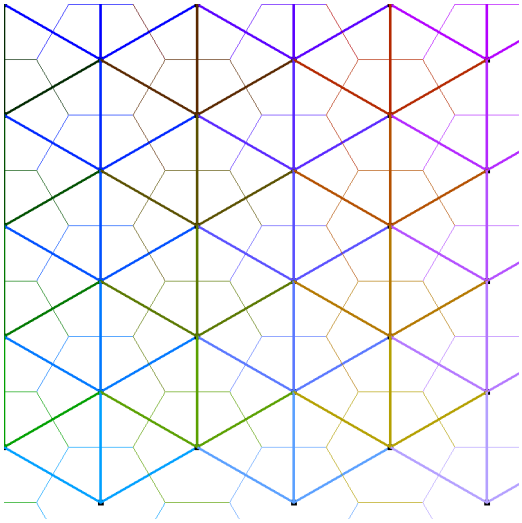
Using quantities we will be able to develop a compression tool with auto-selection of compression algorithm, considering the noise level, throughput, compression ratio, as well as other user specifications. We will also create a tool for generate of synthetic noise with given characteristics. Users will be able to simulate the effect of data compression and determine an acceptable noise level.



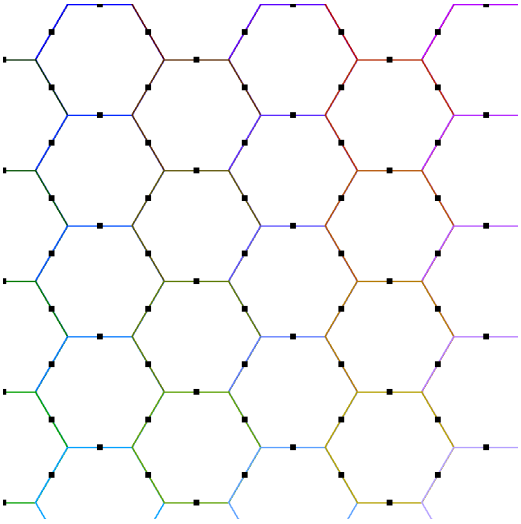
(a) Empty hexagonal grid



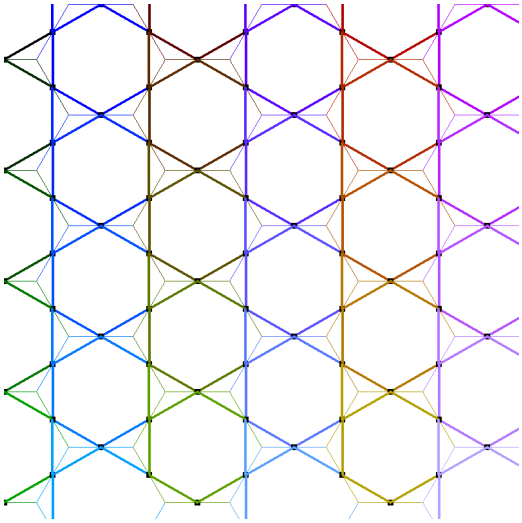
(b) Hexagonal grid with data at the cell centers



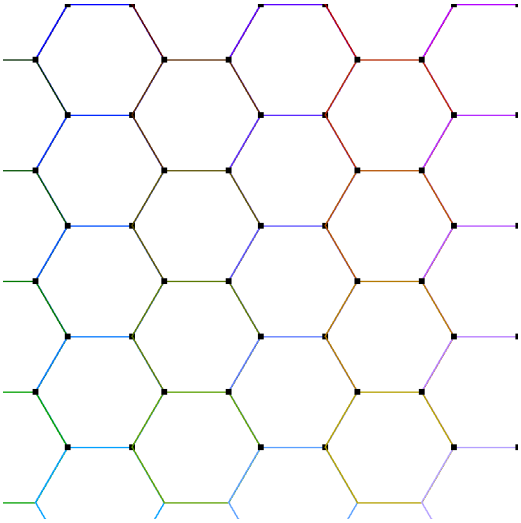
(c) Hexagonal grid with data at the cell's centers, connected neighbours



(d) Hexagonal grid with data on the edges

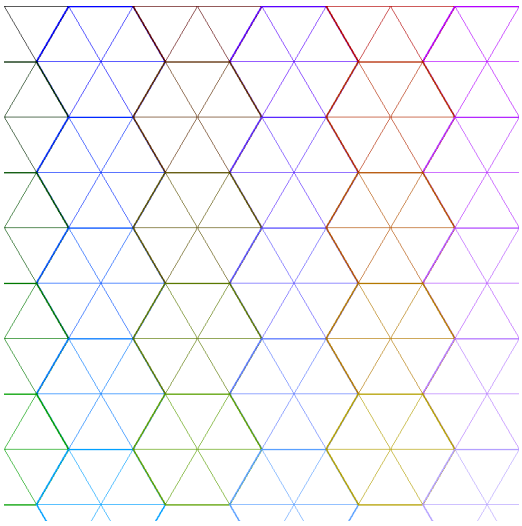


(e) Hexagonal grid with data on the edges, connected neighbours

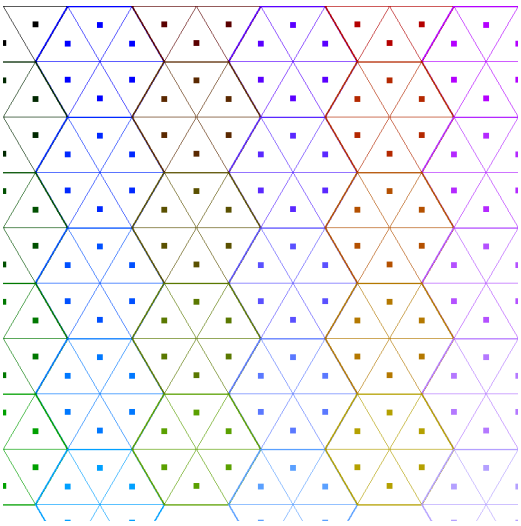


(f) Hexagonal grid with data at the vertices / connected neighbours

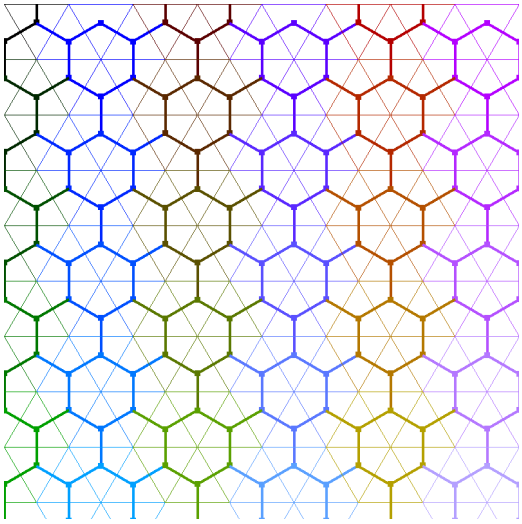
Figure 8: Hexagonal grid



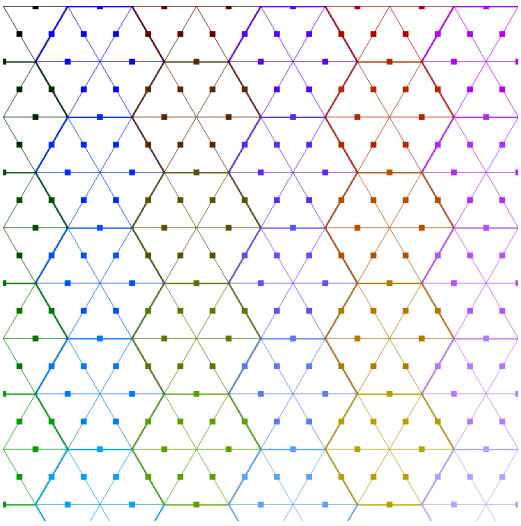
(a) Empty triangular grid



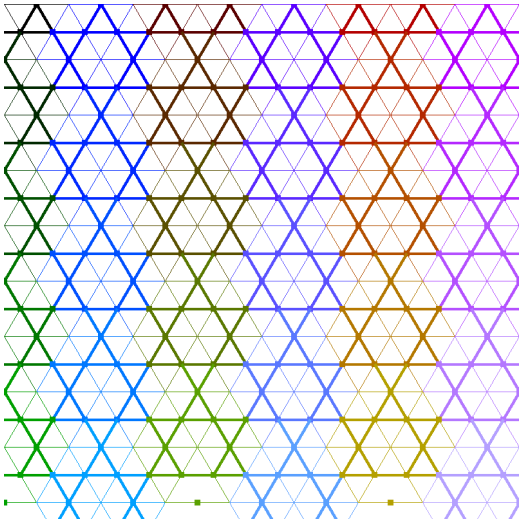
(b) Triangular grid with data at the cell centers



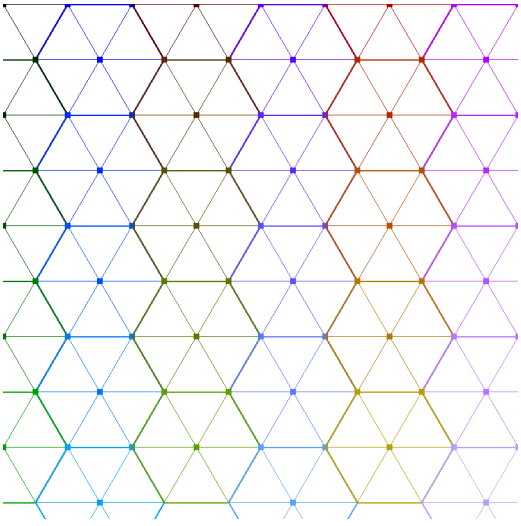
(c) Triangular grid with data at the cell centers, connected neighbours



(d) Triangular grid with data on the edges



(e) Triangular grid with data on the edges, connected neighbours



(f) Triangular grid with data on the vertices / connected neighbours

Figure 9: Triangular grid

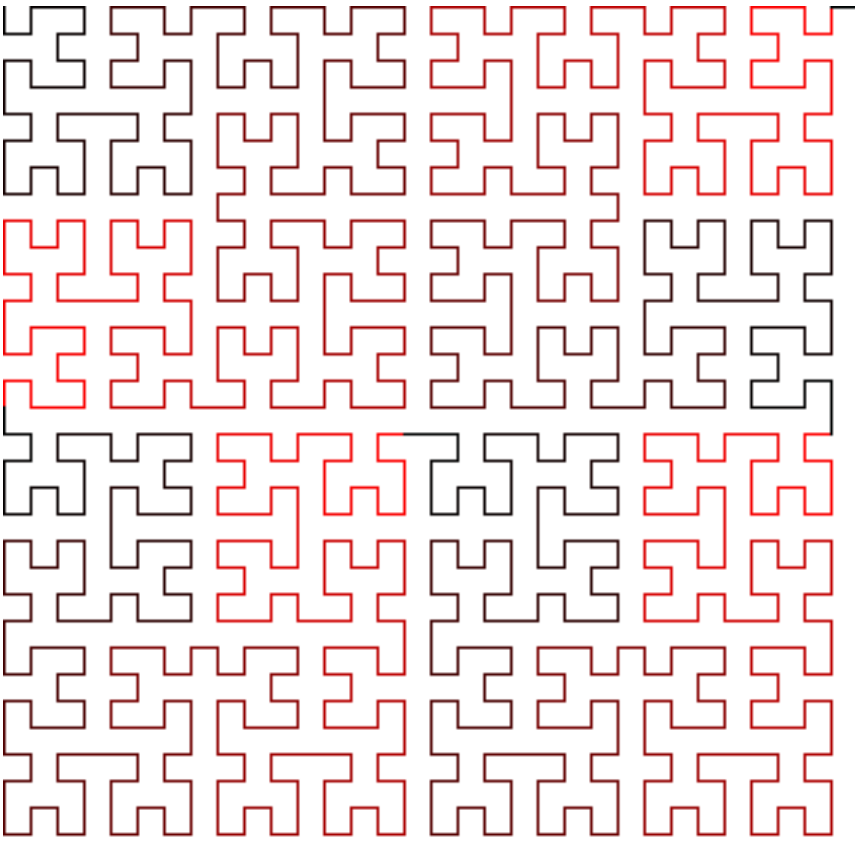


Figure 10: Hilbert fitting curve

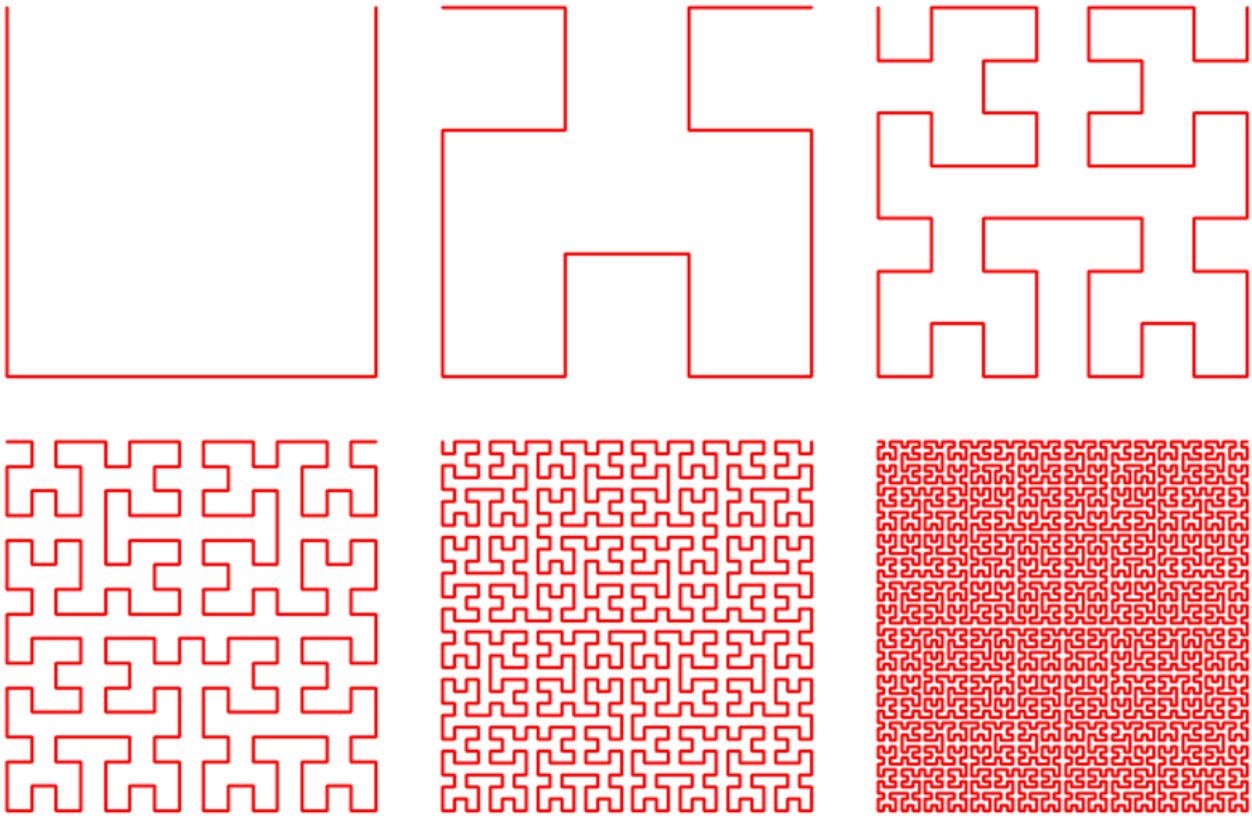


Figure 11: Hilbert fitting curve

1.0000	2.0000	3.0000
5.0000	4.0000	1.0000
7.0000	3.0000	5.0000

(a) Array before compression.

1.0018	2.0036	3.0054
5.0071	4.0065	1.0018
7.0092	3.0054	5.0071

(b) Decompressed array.

$$\eta \leq 0.002$$
$$\varepsilon \leq 0.01$$
$$MSE = 0.00003388$$
$$RMSE = 0.005820739$$
$$\sigma = 1.667988889$$
$$PSNR \approx 60$$

(c) Quantities.

Figure 12: Lossy compression of a 3x3-array.

5 Design

Within AIMES, we design and implement the **Scientific Compression Interface Library (SCIL)**. The main purpose of SCIL is compression/decompression of scientific data, especially, of climate modeling data. It uses different third party compression libraries as well as specifically developed lossy and lossless compression methods. The advantage of the library is, that users don't need to be familiar with all the compression algorithms and its characteristics. Based on user specified quantities, the library selects automatically the best method.
This section describes the design of our prototype library and how the accuracy quantities can be exploited. First, we will start with compression path and explain how quantities can be used to compress/decompress data. Then, we will show how other SCIL capabilities can be used. That are (1) creation of various random data patterns, (2) addition of noise with certain characteristics to existing data (3) and validation of the correctness of the compression in respect to the selected accuracy. Then, we describe the additional tools delivered with our prototype that allow us to use these features on existing data. And finally, we will describe the high-level internals of the library.

5.1 Interfacing I/O Middleware

In our work we primarily target the NetCDF4 C-API. Our intention is to integrate the clever compression functionality of SCIL into NetCDF4. This can be done directly in NetCDF4, but due the dependency, the functionality can also be added indirectly into HDF5 library. Even more, the architecture of HDF5 allows a loosely integration of SCIL, i.e. it can be developed as an independent library. The advantages of this approach are good testability without HDF5 and reusage of the library in other projects. Later, the interface can also be ported to any programming language with C-bindings support, or used by other libraries.
In order to make the compression path work NetCDF and HDF5 require additional functionality. Firstly, a quantity passing mechanism is required, to pass quantities from application to the SCIL library. Secondly, an HDF5 filter is required for communication with SCIL.
The prototypes of the quantity passing mechanism and the HDF5 filter are already implemented (Figure 13). Using them, the application can send the data together with quantities to NetCDF4, NetCDF4 can pass them to HDF5, and HDF5 in turn can pass them to SCIL, yielding compressed data as result. After that, the data can be saved in a file. The decompression step works similar, but in reverse direction.

5.2 Tools

We provide three categories of tools. Thus, like shown on the Figure 14, we will create a compression tool, a noise adder and a 3D plotter. Each of them is connected with supported by SCIL file formats: NetCDF and CSV. Compression tool and noise adder will support user setted file format, one plotter we want make for each file format.

5.3 C-API

In this section we present a subset of the SCIL C-API, of which we think can outline the compression concept. We will take a look at data structures, (de-)compression function and some auxiliary functions. SCIL is still under development and the API can change. Please, visit our GIT-Repository [6] for the most recent information.

[6] `https://github.com/JulianKunkel/scil.git`

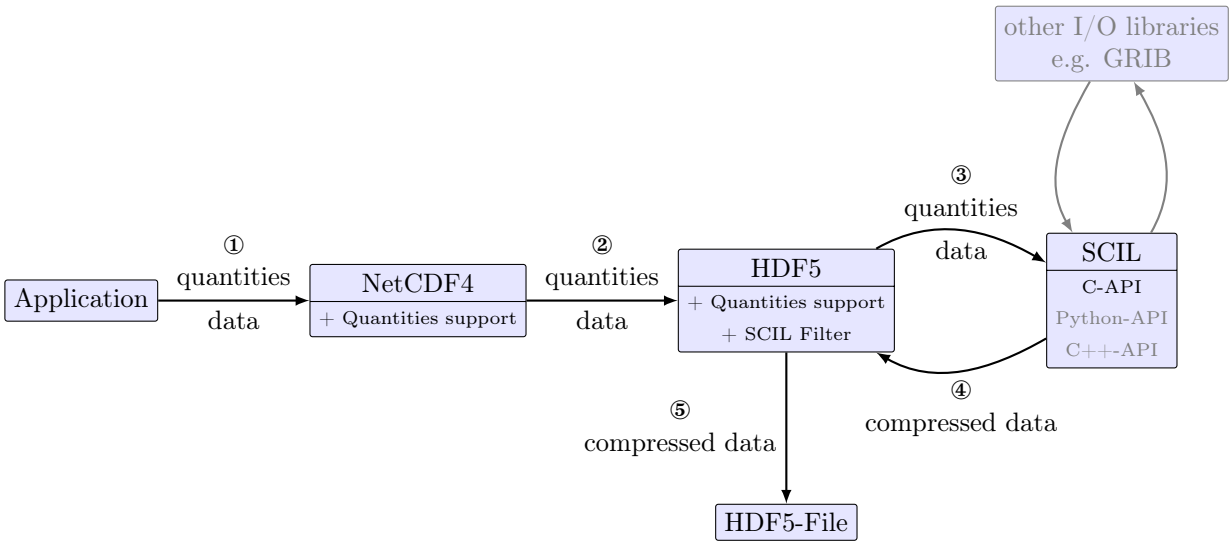


Figure 13: Compression path. Decompression works in inverse direction. (Gray-out elements are out of scope of the project.)

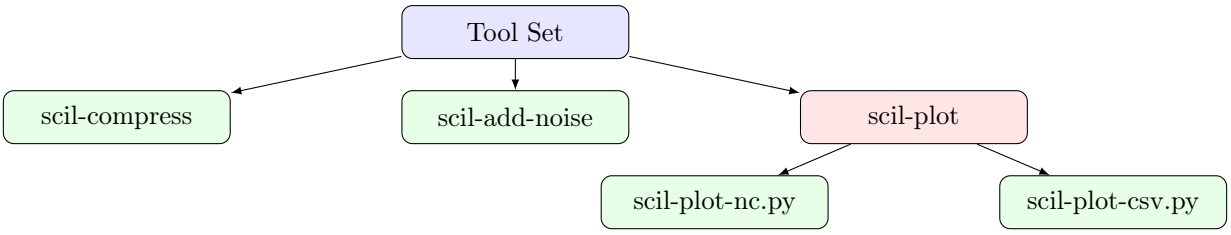


Figure 14: Tool set

5.3.1 Initialization of Hints, Context and Dimensions

User hints are stored in the object of type `scil_user_hints_t`, that must must be initialize by the function `scil_initialize_user_hints`, before it can be used. This object contains a number of members, that can be directly accessed and modified. In the example in Listing 9 we use `force_compression_methods`, `absolute_tolerance` and `significant_bits`.

Signature 8: SCIL user hints initialization.

```
void scil_initialize_user_hints(
  scil_user_hints_t* hints);
```

OUT	`hints`	User hints.

The context is a data structure of type `scil_context_t`. It contains information that is needed to choose the optimal compression method. It is also possible to pass a list of special values that for some reason shall not be touched by lossy compression algorithms. The function `scil_create_context` is responsible for creation of valid contexts.

Signature 9: SCIL context initialization.

```
int scil_create_context(
  scil_context_t** out_ctx,
  SCIL_Datatype_t datatype,
  int special_values_count,
  void* special_values,
  const scil_user_hints_t* hints);
```

OUT	out_ctx	Pointer to the context.
IN	datatype	Datatype of the data (e.g. SCIL_TYPE_FLOAT, ...)
IN	special_values_count	Number of special values.
IN	special_values	List of values that must be preserved.
IN	hints	User hints.
RETURN		Error type. If no error returns 0.

The dimensions are represented through the object of type scil_dims_t. Currently, it supports upto 4 dimensions and can be initalized by one of the functions listed in Signature 10. There are also preparations for dynamic initialization of arbitrary number of dimensions, but no implementation at the moment. For climate data 4 dimensions seems to be sufficient.

Signature 10: SCIL dimension initialization.

```
void scil_initialize_dims_array(
  scil_dims_t* dims,
  uint8_t dimensions_count,
  const size_t* dimensions_length);
```

OUT	dims	Pointer to the dimension object.
IN	dimesions_count	Number of dimensions.
IN	dimensions_length	Size of dimensions.

5.3.2 Compression and Decompression

The SCIL compresson method (Signature 11) uses the data from the context ctx to select the best compression algorithm. ctx is a data structure where user specified quantities are located.

Signature 11: SCIL compression function.

```
int scil_compress(
  byte* restrict dest,
  size_t dest_size,
  void* restrict source,
  scil_dims_t* dims,
  size_t* restrict out_size,
  scil_context_t* ctx);
```

OUT	dest	Pointer to the encoded data.
IN	dest_size	Maximal size of encoded data in bytes.
IN	source	Pointer to the uncompressed data.
IN	dims	Dimensions of the uncompressed data.
IN	ctx	Compression context.
OUT	out_size	Current size of encoded data in bytes.
RETURN		Error type. If no error returns 0.

The definitions of accuracy and dimensions are required before the compression can take place. In the next step, data the dimensions can be initialized an assigned to the data. After that the data can be compression. The dimensions must be defined before the decompression. After that the decompression function (Signature 12) can be called.

Signature 12: SCIL decompression function.

```
int scil_decompress(
  enum SCIL_Datatype datatype,
  void* restrict dest,
  scil_dims_t* dims,
  byte* restrict source,
  const size_t source_size);
```

IN	datatype	Type of the decompressed data.
OUT	dest	Pointer to the uncompressed data.
IN	dims	Dimensions of the decoded data.
IN	source	Pointer to the encoded data.
IN	source_size	Size of encoded data in bytes.
RETURN		Error type. If no error returns 0.

5.3.3 Noise Generation

Noise follows SCIL's hints limitations. With user hints we can choose parameters of noise and it's level. This function (will add the noise of a given type to the data array.

Signature 13: SCIL noise function.

```
int scil_add_noise(
  void* data,
  scil_dims_t* dims,
  scil_user_hints_t hints);
```

IN/OUT	data	Pointer to data.
IN	dims	Dimensions of data.
IN	hints	User hints.
RETURN		error type. If no error returns 0.

The task of the scil_add_noise_xxx tool family is to add noise to the users data. Two different ways are provided by our tools. (1) Creating a copy of the file with noise. (2) Adding noise on-the-fly, while reading data. This can be implemented by a HDF5 filter.
Our tools shall support the following noise modification features. (1) Apply a filter to reduce or amplify certain characteristics. (2) Add multiple noise functions together, typically with a weighted sum so that we can control how much each noise function contributes to the total. (3) Interpolation of generated noise, to produce smooth noise values.

5.3.4 Validation

SCIL provides a validation function (Signature 14) to check compressed data for accuracy. It compares compressed and decompressed data, and checks if conditions provided by ctx parameter are met. Validation fails if the computed relative error is larger than the value in ctx.

Signature 14: SCIL validation function.

```
int scil_validate_compression(
  enum SCIL_Datatype datatype,
  const void* restrict data_uncompressed,
  scil_dims_t* dims,
  byte* restrict data_compressed,
  const size_t compressed_size,
  const scil_context_t* ctx,
  scil_user_hints_t* out_accuracy);
```

IN	datatype	Data type of the input data.
IN	data_uncompressed	Pointer to the uncompressed data.
IN	dims	Dimensions of the uncompressed data.
IN	data_compressed	Pointer to the compressed data.
IN	compressed_size	Size of compressed data in bytes.
IN	ctx	Compression context.
OUT	out_accuracy	Output accuracy contains a set of hints with the observed finest resolution/required precision to accept the data.
RETURN		Error type. If no error returns 0.

5.3.5 Example

Listing 9: SCIL in action.

```c
#include <scil.h>
#include <scil-util.h>

int main(){
  /* HINTS */
  scil_user_hints_t hints;
  scil_initialize_user_hints(&hints);
  hints.force_compression_methods = "[]";
  hints.absolute_tolerance = 0.5;
  hints.significant_bits = 5;

  /* CONTEXT */
  scil_context_t* ctx;
  scil_create_context(&ctx, SCIL_TYPE_DOUBLE, 0, NULL, &hints);

  /* DIMENSIONS */
  const size_t count = 100;
  scil_dims_t dims;
  scil_initialize_dims_1d(&dims, count);

  /* COMPRESSION */
  size_t u_buf_size = count * sizeof(double);
  double* u_buf = (double *)SAFE_MALLOC(u_buf_size);
  for(size_t i = 0; i < count; ++i) {
    u_buf[i] = (double)(i % 10-5.1);
  }
  size_t c_buf_size = u_buf_size + SCIL_BLOCK_HEADER_MAX_SIZE;
  byte* c_buf = (byte*)SAFE_MALLOC(c_buf_size*4);
  scil_compress(c_buf, c_buf_size, u_buf, &dims, &c_buf_size, ctx);

  /* DECOMRESSION */
  double* data_out = (double*)SAFE_MALLOC(u_buf_size);
  scil_decompress(SCIL_TYPE_DOUBLE, data_out, &dims, c_buf, c_buf_size,
    &c_buf[c_buf_size*2]);

  /* VALIDATION */
  scil_user_hints_t accuracy;
  scil_validate_compression(SCIL_TYPE_DOUBLE, u_buf, &dims, c_buf,
    c_buf_size, ctx, &accuracy);

  free(c_buf);
  free(data_out);
  free(u_buf);
  free(ctx);
  return 0;
}
```

5.4 Compression chains

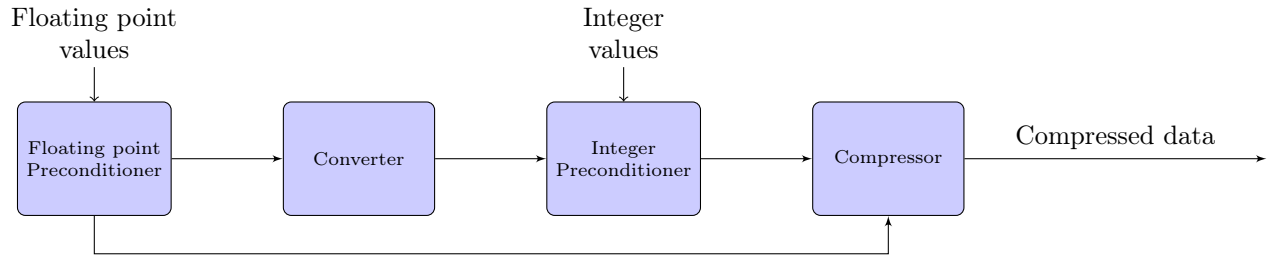
SCIL allows some degree of customization of the compression process. The data can be preconditioned and converted before it is passed to the compression algorithm. Optionaly, after that, a second compression algorithm can be applied on compressed data.. This section discusses the usage of SCIL in our work.

5.4.1 Strategy

We can imagine three strategies to implement compression chain:

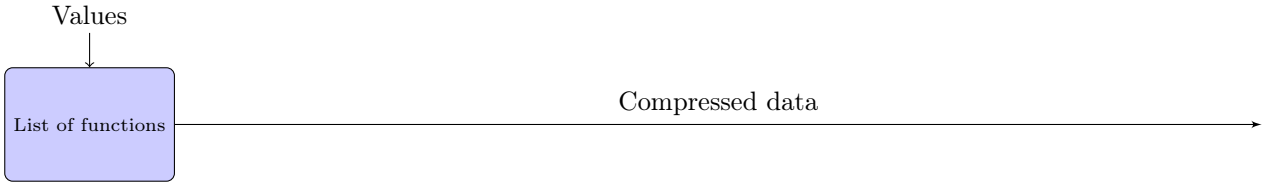
1. With conversion (Figure 15a). This strategy uses a floating point values to integer converter. This leads to accuracy reduction. The task of the preconditioners is to increase compression rates of implemented algorithms. One is placed before quantizer and one after. In the final step, a compressor compresses the byte array.

2. Without conversion (Figure 15b). This strategy operates on all data types without quantization.



(a) First strategy: compression chain with quantization.

(b) Second strategy: compression chain without quantization.



(c) Third strategy: list of function.

Figure 15: Sequence diagrams of compression chains.

	Strategy		
	1	2	3
Debuggability	+	+	-
Ease-of-implementation	-	+	-
Ease-of-use	-	-	+?

Table 4: Comparison of chain strategies.

3. As list of functions (Figure 15c). This strategy uses several preconditioner functions only.

The advantages and disadvantages of the strategies are summaries in the Table 4.
The basic idea is to split the compression into different steps and compression process. Using this approach our library will support different compression scenarios.
Then we define three types of building block:

1. *Preconditioner* uses data transformation techniques to present the data in a different form, but without changing the "solution". The purpose of a preconditioner is to produce optimal data representation for the next chain build block, so that the next step can be done more efficiently. There are two types of preconditioners.
   - Floating point preconditioner operates on floating point data type.
   - Integer precoditioner operates on integer data type.
2. *Converter* transforms the data. For example it can be a *qunatizer* which transformst floating point values type to integer values.
3. *Compressor* applies compression algorithm on any type of input data. The output of compressor is byte data.

Not all compression algorithms support all data types. This may be problematic for some compression chains. Therefore, we must adapt input (uncompressed) data to the given by a compression algorithm data types.

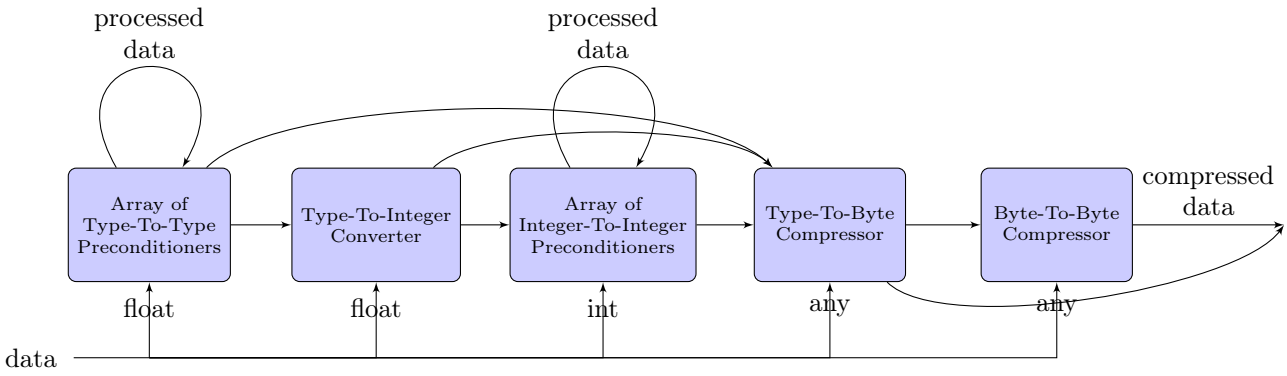


Figure 16: Complete compression chain.

6 Implementation

In this section we will describe the SCIL compression chain and SCIL context in more detail and tell you how the components will be integrated in HDF5 and NetCDF4.

6.1 Implementation of compression chain.

Our approach can benefit from lossy and lossless compression methods in the following way. The first compression algorithm is thought to be specialized on number compression. Typically, it will be a lossy compression algorithm and will reduce the precision of data. The second algorithm will interpret data as a byte stream, and will compress it with lossless method. Actually, we distiguish between two types of compressors:

- Data type specific compressor can be lossy or lossless, takes distinct data and produces byte data.
- Byte compressor is always lossless, takes data of any data type and produces byte data.

In our exprience converters and specialized compressors work best on special data. Therefore, before we place before preconditioners before the modules. In some cases we need to apply several functions to the data, before the optimal state is reached, therefore it is possible to set several preconditioners.
To be flexible the component must support a broad range of types.
The output from previous building block is used as input in the next building block. The preconditioner blocks are optional and can also contain more than one preconditioner.
One of the most important objects of study is the automatic chain builder for optimal compression. At the moment we do not have a clear idea how we are going to implement this. We think we can do it by a set of empirical rules, which has to be created first. At this point we are very optimistic. We hope, that we get enough rules to train a decision tree. In that case after pruning we will probably be able to transform the decision tree into a new set of short and meaningful rules, which will be also valid for missing information. In this way we hope to learn new information.

6.2 Integration in HDF5/NetCDF

The two main question related to HDF5 and NetCDF4 are how to integrated SCIL and how to pass user quantities.
HDF5 provides a general and well-documented filter concept, based on a set of callback functions. The implementation these functions and creation of a compatible filter in general will probably be a straightforward task. The integration and usage of third party filters is also well-supported in HDF5. We will use the following interfaces for that. The filter and compression interface (H5Z) can be used for (un-)registering a filter at runtime. The dataset interface (H5D) can be used to attach a filter to a dataset. The property list interface (H5P) can be used to pass configuration to datasets and filters (it is also a general concept).
H5P is a powerful configuration mechanism in HDF5. There are no restriction on how a property list has to look like. HDF5 simply forwards the property list from application directly to the filter. The interpretation of the list is done by the filter. It allows to put set of quantities into a property list and forward it to the filter. Unfortunately, there is no such mechanism in NetCDF. NetCDF uses a completely different and incompatible approach. Instead of passing a configuration object to the opened file object it uses specialized functions to change its internal state. Therefore, in our view the extension of NetCDF interface is the best solution to this

problem. At the moment we don't have a clear picture, how the interface will be look like, but we can image it work like in Listing 10. This will be a subject of discussion with NetCDF community.

6.3 Extenstion of NetCDF4 Interface

The code in the Listing 10 shows two alternatives of how SCIL can be integrated in NetCDF4.
The first alternative in Lines 26 to 32 is motivated by the rule: *"Good interfaces are easy to use correctly and hard to use incorrectly"*. Besides, it follows the basic NetCDF4 interface concept, i.e, no direct access to the context is allowed. In this implementation a context would be allocated by `nc_def_ctx` and managed internally by NetCDF. Modifications of the context would only be possible by special functions like `nc_add_ctx_hint`. The function `nc_add_ctx_hint` is a bit different. Each call to the function takes a key-value-pair and adds a hint to the context, allowing to define any number of quantities. Actually, to be fully compliant with NetCDF4, this job must be done by arrays of keys and values, but we decided not go this way, since this is error prone. This design is generic enough to be used with other libraries. Even the already available deflate algorithm could be easily ported.
The second alternative in Lines 34 to 38 provides on one side a set of powerful features, and on the other side it has serious drawbacks. Here `hints` are created and defined using the SCIL interface. Then they are passed as a pointer to `nc_def_var_scil_compression` function. The function would create a deep copy of `hints` to prevent a category of invalid memory accesses, that can occur when the object is destroyed, after it was passed to NetCDF. The main advantages of this concept are probably flexibility and maintainability. `hints` could be passed from application to NetCDF, from NetCDF to HDF5, and from HDF5 to SCIL without any conversion and would simplify the implementation a lot. Besides, it would make SCIL completely independent from NetCDF, and new SCIL features would be immediately available without any adaption of NetCDF interface. The work in NetCDF and HDF5 has to be done once and, if done properly, never again. The saved time could be used for development of SCIL. The drawbacks are the usage of two different interfaces is non-intuitive and is an additional hurdle for the user. Furthermore, this approach allows a context to exist outside the NetCDF4, which violates the NetCDF4 interface concept and is less secure. It can be a source of memory errors, like uninitialized or invalid memory access. Uninitialized memory access can occur, when user forgets to call `scil_initialize_user_hints`. Invalid memory access can occur when no memory was allocated to the pointer. Both types of error cause undefined behavior and could be difficult to debug.

Listing 10: Sample code for compression in NetCDF using SCIL.

```c
#include <stdlib.h>
#include <netcdf.h>

#define NDIMS 2
#define NX 100
#define NY 100

int main(int argc, char** argv) {
   int ncid, x_dimid, y_dimid, varid;
   int dimids[NDIMS];
   int d[NX][NY];
   int x, y, retval;
   for (x = 0; x < NX; x++)
      for (y = 0; y < NY; y++){
        d[x][y] = x + y;
      }
   nc_create("data.nc", NC_NETCDF4 | NC_CLOBBER, &ncid);
   nc_def_dim(ncid, "x", NX, &x_dimid);
   nc_def_dim(ncid, "y", NY, &y_dimid);
   dimids[0] = x_dimid;
   dimids[1] = y_dimid;
   nc_def_var(ncid, "data", NC_INT, NDIMS, dimids, &varid);

   /* nc_def_var_deflate(ncid, varid, 0, 1, 9); */

   /* Alternative 1: */
   int ctxid;
   nc_def_ctx(NC_SCIL_COMPRESSION, &ctxid);
   nc_add_ctx_hint(NC_SCIL_SIGNIFICANT_BITS, 5, ctxid);
   nc_add_ctx_hint(NC_SCIL_ABSOLUTE_TOLERANCE, 0.5, ctxid);
   nc_add_ctx_hint(NC_SCIL_FORCE_COMPRESSION_METHODS, 1, ctxid);
   nc_def_var_compression(ncid, varid, ctxid);

   /* Alternative 2: */
   scil_user_hints_t hints;
   scil_initialize_user_hints(&hints);
   hints.absolute_tolerance = 0.5;
   nc_def_var_scil_compression(ncid, varid, &hints);

   nc_enddef(ncid);
   nc_put_var_int(ncid, varid, &d[0][0]);

   nc_close(ncid);
   return 0;
}
```

7 Summary and Conclusions

The main purpose of compression methods is to shrink data size and to save storage space, but they also possess a huge potential to reduce the gap between computational power and I/O performance, because often after compression less data has to be moved. For these reasons, many modern file formats, in particular HDF5 and NetCDF4, provide native support for compression. This is especially beneficial in climate science, where data amounts are huge and are growing constantly. Unfortunately, the compression algorithms used in HDF5 and NetCDF4 are lossless and doesn't meet the requirements of climate science to full extent. In climate sciene, there is often nothing against reducing the precision of data by lossy compression algorithms, when the impact on the simulation results is negligible. After that, the data can still be compressed with a lossless compression algorithm. Application of both kinds of compression methods can result in a higher compression ratio, compared to compression ratio of only one algorithm.
Theoretically, it's feasible to do this work manually, but due to high effort it's unacceptable. All compression methods have their strenghts and weaknesses. Many of them are even specialized on a particular kind data. Therefore good knowledge of compression algorithms as well as of data is required to do the complex task right. Scientific Compression Interface Library (SCIL) facilitates this task. It supports a number of lossless and lossy compression algorithms and can choose automatically the best one based on pre-defined user hints. As the name suggests, user hints are a set of metrics, that describe required precision, maximum error, consider noise and take some other metrics into account.
The primary objective of our project is the integration of SCIL into NetCDF4/HDF5 libraries. This includes improvement of the SCIL library, development of an HDF5 filter and implementation of a NetCDF4 interface extension. The secondary objective is the development of stand-alone auxiliary tools, e.g., for noise generation or visualization.
We have already a functional prototype of the SCIL library. Its compression chain allows to construct different compression processes for different compression scenarios on the fly. It works pretty well with user hints, which can be set at runtime. The extensible architecture of HDF5 allows to attach easily third party filters and plugins. For that purpose, it provides a number of well documented interfaces. Based on these interfaces we build a partially functional HDF5-SCIL-Filter. Unfortunately, the NetCDF architecture doesn't support integration of any kind of third party functionality. It make the integration of SCIL more complicated. One possible solution to this problem (and also our approach) is the extension of the NetCDF4 interface. We have different proposals for how this can be done, but the final decision will probably be made in the course of the project.

Acknowledgement

This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 „Software for Exascale Computing“ (SPPEXA).

References

[AKD15] M. Aechtner, Kevlahan, and T. Dubos. “A conservative adaptive wavelet method for the shallow-water equations on the sphere”. In: *Q.J.R. Meteorol. Soc.* 141.690 (July 2015), pp. 1712–1726.

[DC15] Sheng Di and Franck Cappello. “Fast Error-bounded Lossy HPC Data Compression with SZ”. In: (2015).

[Fil] *File formats for climate data.* `https://climatedataguide.ucar.edu/climate-data-tools-and-analysis/common-climate-data-formats-overview`. [Online; accessed 04-10-2016].

[Fpc] *Floating point compression.* `http://computation.llnl.gov/projects/floating-point-compression`. [Online; accessed 04-10-2016]. 2016.

[Gnu] *GSL - GNU Scientific Library.* `https://www.gnu.org/software/gsl/`. [Online; accessed 04-10-2016].

[Gzi] *GZIP algorithm.* `http://www.gzip.org/algorithm.txt`. [Online; accessed 04-10-2016].

[Hdfa] *HDF5.* `https://support.hdfgroup.org/HDF5/`. [Online; accessed 04-10-2016]. 2016.

[Hdfb] *HDF5 Filters*. https://support.hdfgroup.org/HDF5/doc/H5.user/Filters.html. [Online; accessed 04-10-2016].

[Kas] Jeremy Kasdin. "Discrete Simulation of Colored Noise and Stochastic Processes and 1/f power a Power Law Noise Generation," in: *Proceedings of the IEEE, V. 83, Number 5* ().

[LI06] Peter Lindstrom and Martin Isenburg. "Fast and Efficient Compression of Floating-Point Data". In: *IEEE Transactions on Visualization and Computer Graphics, 12(5):1245-1250* (2006).

[Lz4] *Yann Collet. LZ4 explained.* http://fastcompression.blogspot.de/2011/05/lz4-explained.html. [Online; accessed 04-10-2016]. May 2011.

[MGB11] M.Stoyanov, M. Gunzburger, and J. Burkardt. "Pink Noise, 1/f power alpha Noise, and Their Effect on Solutions of Differential Equations," in: *International Journal for Uncertainty Quantification, V. 1, Number 3* (2011).

[Mis] *HLRE-3 Mistral*. https://www.dkrz.de/Klimarechner/hpc. [Online; accessed 04-10-2016]. 2016.

[MMM12] O.A. Mahdi, M.A. Mohammed, and A.J. Mohamed. "Implementing a Novel Approach an Convert Audio Compression to Text Coding via Hybrid Technique". In: *International Journal of Computer Science Issues. 9 (6, No. 3): 53–59.* (2012).

[Nrl] *Numerical recipes*. http://numerical.recipes/. [Online; accessed 04-10-2016].

[SKri] Armin Schaare and Julian Kunkel. *SCIL - Scientific Compression Interface Library*. Software Lab Report. April 2006.

[Sto11] Miroslav Stoyanov. *CNOISE* $1/F^{\alpha}$ *Power Law Noise Generation*. https://people.sc.fsu.edu/~jburkardt/c_src/cnoise/cnoise.html. [Online; accessed 04-10-2016]. 2011.

[Wava] *Source code for 2D wavelets, wavelet packets (complete or overcomplete), complex wavelets, and complex wavelet packets*. http://eeweb.poly.edu/~onur/source.html. [Online; accessed 04-10-2016]. 2012.

[Wavb] *Source code wavelet-based adaptive numerics on icosahedral spherical meshes*. https://bitbucket.org/kevlahan/wavetrisk. [Online; accessed 03-02-2017]. 2016.

[Wika] *Average absolute deviation*. https://en.wikipedia.org/wiki/Average_absolute_deviation. [Online; accessed 04-10-2016]. 2016.

[Wikb] *Colors of noise*. https://en.wikipedia.org/wiki/Colors_of_noise. [Online; accessed 04-10-2016].

[You10] Yuli You. "Audio Coding: Theory and Applications". In: *Springer* (2010).