Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Masterarbeit

Enabling In-Situ Processing of Climate Simulation Data through CDO

vorgelegt von

Oliver Heidmann

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6420331
Erstgutachter:	Thomas Ludwig
Zweitgutachter:	Jannek Squar
Betreuerin:	Anna Fuchs

Hamburg, 3.11.2024

Contents

Abstract

In this work we design and introduce PyCdo. PyCdo is a prototype for a Python interface that extends the popular data processing tool CDO with in-situ capability. The interface is intended to be used with Python post processing workflows directly applied on output data at runtime of simulations. We introduce the surrounding HPC environment relevant for the design and implementation, including aspects that require the design to in incorporate the type of user base and through this the usability and maintainability. Further, we give an introduction into the architecture of for this work relevant parts of CDO and show the realization of the design with a heavy emphasis on the maintainability, the usability and a short on boarding process for the use and the development of PyCdo. We provide a working Python interface for CDO that is capable of reducing the normally required file I/O drastically as all data transfer between CDO execution within the same script can be done in-memory. Additionally the workflows for post-processing, including the use of Xarray and compatible plotting libraries, can be streamlined with PyCdo. Last, we benchmark PyCdo and show that it can outperform CDO, but also that the current version has serious flaws in terms of scalability. In the end we can provide a solid prototype and alternative to bash scripts.

1. Introduction

Climate research could not work efficiently without modern High Performance Computing (HPC). Over the years, systems and simulations have grown in size and capability as well as in complexity. With this growth, there is an ongoing struggle to improve the efficiency of the machines, the code and the employed workflows, while researchers and their projects are, and always will be, pushing the available resources. There are many projects that are working on improving the current status as well as preparing the current soft- and hardware stacks for the coming, even larger, simulations. In this work we develop a Python interface for the widely used climate simulation data post-processing tool Climate Data Operators (CDO)[35]. This interface allows in-situ processing of climate simulation data. The goal is to improve the general performance of the CDO scripts in context of I/O on HPC-Machines, all while improving the readability and adding the ability to make use of the mostly idle I/O processes at the simulations runtime. We go further into detail on why these changes and additions are beneficial as well as the design choices and libraries used. After explaining the rationale behind our implementation we show the workflows we try to improve and detail the environment in which we conduct the experiments. The second last part consists out of the performance measurements as well as résumé which design choices were ultimately implemented and which had to be discarded. The final part concludes and gives recommendations for future evaluation and additions.

1.1. Motivation

In this section we want to introduce our ideas for why and how this will benefit science and why we choose to extend CDO specifically, and also why it makes sense to enable data processing software to directly work with simulations and other programs. Further, we want to explain why we worked out a specific set requirements for developing CDO into an in-situ-enabled software. Included with the requirements is how we decided to not only provide an easy to work with software for the user but also why we set the goal to make the software as low maintenance as possible as well as why took measures so that the learning process for maintainers and users is uncomplicated and easy to approach. Since we have multiple topics that motivated us to start this endeavours the following section will be split into multiple parts. With the first part being the motivation for in-situ processing.

Exascale Problems:

Since the creation of the first atmosphere computer models, the computational power available increased exponentially. Already in 2013 were at the point where we could run parts of our models at a sub-kilometre scale[9], nowadays we are approaching entire digital twins of earth were most of Earth its systems will be at a sub-kilometre scale[31]. With this new challenges arise. Among them the problem of moving the calculated data from the compute units to storage. Shalf et al. found that moving data is also one of the fastest growing problems in term of power consumption[8]. But not only in terms of power consumption but also in terms of bandwidth we approach amounts of data where current systems are reaching their limit. And that is only in the context of exascale computing. While exascale computing (capable of 10^{18} (ops/sec)[5] is still not entirely working, researchers are already making plans for the next era: Zettascale Computing[21]. With the expected 10^{18} ops/sec with exascale we already face bottlenecks in term of HPC machine I/O bandwidth. The expected amount of data that can be written is

only 10^{12} B s^{-1}[23]. From this we can already see that writing the entire set of results to disk is infeasible and other methods need to be pursued. With this in mind, writing tools that help with reducing the amount of transferred data would help alleviating some of the problems scientific simulations face. Or in the context of this work: enabling already existing tools to work in-situ. Where in-situ processing describes data processing not after the simulation is done but directly integrated with the simulation itself. This reduces the required I/O operations drastically as the step of writing the data and then loading again for post-processing and pre-processing is removed. All while reducing the data itself by already evaluating the results and shaping/reducing the data into the required form for research performed on it later on.

Tool Choice: CDO is a prime candidate for such in-situ applications. It is a data processing tool with over 900 operators that is widely used in climate science. Our reasoning for choosing CDO is that it is used in the environment of the MPI-M where currently their climate simulation ICON[27] has been prepared to run exascale simulations, and is in the final steps for starting knowledge generating production runs. Throughout the development of ICON the need for an in-situ option has become more and more visible over the last years and multiple tools already started to adapt. Such as Yet Another Coupler (YAC) the coupling library used by ICON. This library has already been adapted to allow access to the calculated data that ICON produces at runtime for external processes. With this and that ICON employs the same reading and writing interface for file I/O as CDO, namely Climate Data Interface (CDI)[50], and that the author of this work is deeply familiar with CDO and CDI, the choice of using CDO was made. While we will not use the interface of the CDI directly to interface CDO with ICON the compatibility, through both programs using CDI, allows easier updates for a future date. In such an update, CDO could be more directly integrated, not as a post processing library but also as a repository for processing tools. While CDO already provides a Python interface and would be able to be easily integrated into the ICON/YAC workflow, in a later section, we will show that the current interface does not provide the solutions for the problems we are attempting to solve with this work. One problems is that the current interface is wrapping command-line calls of CDO with from Python started subprocesses. This means that the normal workflow of loading and writing files to disk is still mandatory when using the existing Python interface. This in the context of the stated exascale problems makes the current approach not feasible in the long term. In addition the reading and writing of large files can be, depending on the exact workflow, a burden on the runtime performance of the scripts. Which also means that the usage of CDO, even if not used in a in-situ environment, could strongly benefit from the reworking of the Python interface. Another problem is that the interface uses the command-line syntax to create CDO workflows. While this works, it has several drawbacks that we will address later in this section. A third problem is that there is no way to properly interface with e.g. Xarray. Xarray[17] is a widely used Python library for working with datasets and researchers at the MPI-M have asked for a Xarray compatible CDO version for quite some time.

When deciding on CDO we were aware that it currently is not in a form to be easily integrated into an in-situ process. As such we explored other alternatives such as Ophidia[10]. Ophidia is similar to in CDO as it provides a set of operators for processing scientific data. While Ophidia provides methods for working in-situ, it works on a server basis and its Python interface also relies on an ophidia server running in the background. Its workflow descriptions use JSON for their definition. These scripts can then be send to the backend where ophidia servers execute the workflow[16]. Ophidia has a more general workflow and data management approach where we want to develop the direct integration of CDO into the workflows.

Ultimately, we decided on CDO because of the already present integration in the MPI-M maintained codebases and the more simple approach that CDO brings, in contrast to Ophidias I/O and server stack. Additionally we would prefer a fully functional Python interface over workflow definitions with JSON, as Python allows for more flexibility and with our approach

we remove the need to maintain the layout of the JSON files. With the use of Python we can also provide features like autocompletion and other usability focused features. Further, our goal is to reduce the amount of I/O usage in the cluster network, Ophidias approach adds the communication between their servers and the clients to the already existing I/O processes in a cluster.

Another reason is that reworking the currently employed post and preprocessing scripts, that are already written with CDO, into in-situ workflows would also be more feasible than rewriting them to be able to use Ophidia. While this work is not intended to be only useful for the MPI-M and in turn ICON, certain realistic assumptions, such as that the maintenance of the provided system is in the hands of the MPI-M, and that the use of external libraries would be detrimental to the general efficiency of working on ICON/CDO and other in-house software. Due to the use of CDO we can limit the introduction of new software to the MPI-M which in turn keeps the onboarding and familiarization times low. Why these are important is discussed in a following section.

Additionally enabling the in-situ approach with CDO will pave ways to increase the repeatability and reproducibility of the employed workflows. Integrating CDO into ICON or in general into the Python workflows will allow the reuse, and with that a reduction in code duplication, of the already present post processing algorithms within CDO. Since CDO is already used for the pre- and post processing,integration reduces the need to write additional routines and data produced by ICON can easily verified with external CDO commands. With the avoided code duplication naturally comes also a reduction in maintenance. In the future one could even perceive the extraction of specific routines form YAC and ICON into CDO while keeping the workflow definitions unchanged.

Further, with a decently designed Python interface, future workflows can be written in Python instead of the currently common bash scripts. These are hard to test, hard to debug and an are hard to read. In short they are hard to share to maintain and to understand. In addition bash scripts are error prone[34] and invite unintended undiscovered behaviour. Python is in these circumstances way easier to read, maintain and use and thus easier to share.

In summary we want to help the scientific climate community with this work in several aspects. The new possibilities of using CDO in-situ will help with the exascale problems such as the amounts of data being to large to be efficiently being written to disc while using previously unused resources in form of idle processes. The use of Python will make the integration into other workflows outside the simulations easier to use while not introducing new algorithms or large codebases. In context with the earlier improvements the reuse of already existing software we will help with reproducibility and maintainability and add little untested routines. Further, we want to improve the readability, maintainability and thus shareability of scripts.

2. Background

In this section we introduce the general scientific background, CDO itself, the climate simulation software ICON, and among others the currently employed workflows as well as an overview to the used file system Lustre. In addition we will introduce Levante: the DKRZ HPC machine on which our testing is done.

2.1. Scientific Climate Modelling Environment

In the context of the Modelling environment we will discuss the aspects that the general scientific work environment entails. Some of these aspects are also applicable to work outside science while some are specific to the scientific community. All of them are important since they have consequences for the adaption rate of the provided software as well as usability and ability to maintain the software with a reasonable amount of work. Further, we will go into why it is especially important to plan for the maintainability and ease of use within the context of scientific work.

In science software is a means to an end. Researchers are not hired as software-engineers but to produce new knowledge in form of papers and conference presentations. From this follows that their craft ist not software-design and general software development skills are not the focus of their education or their day to day work[20]. Judith Segal[4] described researchers as "professional end user developers". By this she describes the researchers as working in a domain-knowledge heavy field, where they don't see themselves as programmers while spending significant amounts of time developing software. While researchers have rarely problems learning general purpose languages they lack formal training and the time to receive further formal training[4]. In combination with this the mostly highly complex field of HPC does not improve the situation. Modern simulations would not be able to be done in a reasonable time frame without HPC. Researchers have to navigate not only base level programming but also the specialized and otherwise rarely used tools of the HPC world. This includes the usage of different compilers, versioning tools, code that needs to run on multiple targets (GPU, CPU, etc.) and the often incompatible versions of the required tertiary software. In addition modern simulations are more and more required to run on different HPC machines which increases the problems software-stacks and differing hardware can cause dramatically. With all this domain specific knowledge being required to work efficiently or to work at all the fact that the researchers and also the dedicated developers are often only employed for two to three years[20]. With these large turnover numbers and the scientific culture of rather passing on knowledge verbally or working together[20] and the lack of proper systematized onboarding processes the large knowledge requirements become a problem for the dedicated developers as well as for the researchers themselves. With all this combined it seem reasonable that there is little time for maintenance or software design in general. Further, the idea of introducing stricter coding standards is often met with little enthusiasm as these standards seem to interfere with the actual goal of running experiments and tend to be perceived by the researchers as just another burden.

2.2. Climate Data Interface (CDI)

The CDI library handles the various different file formats existing within the climate research community (see fig. 2.1). Through CDI all formats can be used through a single interface since it manages access to variable, timesteps and all other components of climate related data structures. CDI serves as the main I/O component for ICON and CDO. While other I/O components are being added to ICON, CDI still handles the vast majority and even is intended to be integrated into the newer components. Within CDO all I/O is handled by CDI. CDI provides a C99 and Fortran77 interface, both strictly adhering to their standards to avoid problems with compilation. CDI was written to be as portable as possible and that it will compile on most UNIX system. It is actively being developed and in recent years updates like the capability for writing in parallel were added. CDI has been structured in a way that it is extensible and on the developers side provides the necesseray interface that can be implemented to extend the available formats.

CDO File Formats
netcdf 1 to 5
fdb
grib 1/2
SERVICE
EXTRA
IEG
nczarr

Table 2.1.: Supported file formats of CDI/CDO

2.3. Climate Data Operators

Climate Data Operators (CDO)[35] is a C++20 command-line post processing tool for climate and weather data and offers over 900 operators. CDO is widely used inside the weather and climate science community. From mathematical functions like addition subtraction and more advanced mathematical procedures to simple information collection operations to masking and regridding routines, CDO provides an ever increasing set of tools for data post- and preprocessing. CDO supports the most relevant data formats (see fig. 2.1) with the most commonly used are GRIB and netCDF.

A simple example for the usage of CDO can be seen in listing 2.1. This example would spawn three CDO-processes, each for one of the operators `max`, `monmean`, `selvar`. With this, we extract the temperature data and get the maximum value of the monthly means.

```
cdo -max -monmean -selvar,temp data_from_1990_to_2024.nc result_file.nc
```

Listing 2.1: Simple CDO call on the command-line. It computes the maximum value of the monthly means.

With this example we also introduced of a core feature of CDO: chaining operators into a single command. In the following sections we will go over the relevant features and structures, among them a more detailed description regarding operator chaining. While we will not go over all features CDO has to offer, all discussed features are relevant for the requirements and the implementation of PyCdo.

2.3.1. Operator Chaining

Chaining operators is one of CDO's main features. It is the ability to link the operators into a single workflow where the intermediate results are directly passed on to the next operator. This is done by using multiple operators in a single call where each operator is prepended with a '-'.

```
cdo -max [ -monmean -selvar,temp data_from_1990_to_2024.nc ] result_file.nc
```

Listing 2.2: Simple CDO call that computes the maximum value of the monthly means extended to show grouping feature

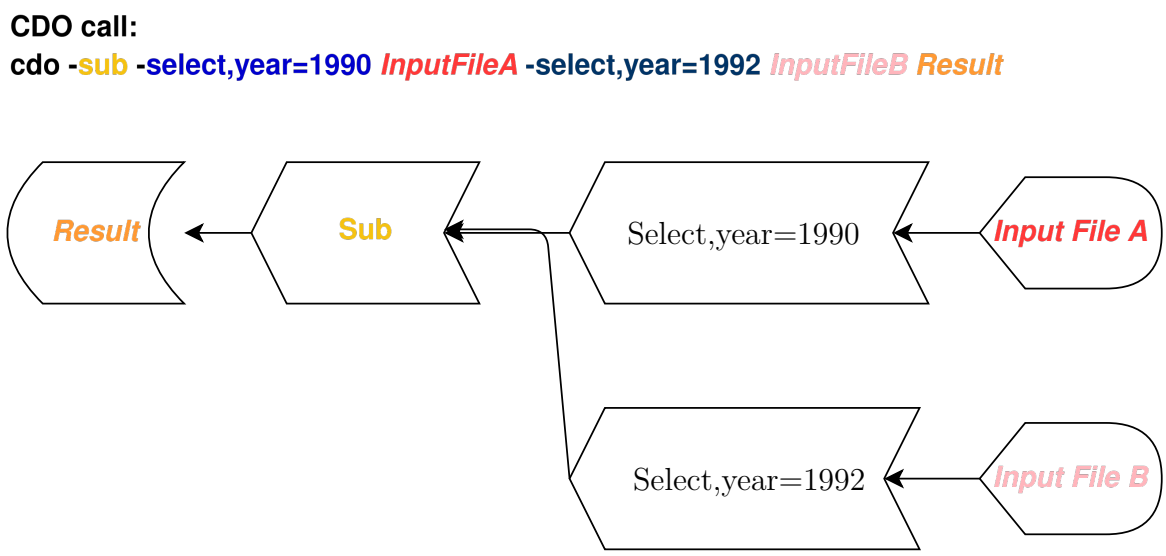

Figure 2.1.: CDO usage example showing the tree like structure that a CDO call constructs

The basic principle behind the structure of these calls is the Polish Notation [1]. While true Polish Notation is bracket-free, CDO employs additional features for operators which have at compile time unknown number of inputs. Polish notation assumes a fixed arity for each operator[1]. Some of the CDO operators can take an arbitrary number of inputs which, without the brackets, makes them impossible to nest as it is not clear where to assign the next operator. Another use of the brackets is e.g. for complex calls where organizing or clarifying which operators belong where can be benefical. For this feature CDO uses '[' and ']' to create these groups. Groups can be nested and allow previously not parseable inputs to be used. Each group its contents are then again treated as being in Polish Notation. In listing (2.2) we can see an example for the use of the grouping feature.

Each operator is processed inside its own thread while passing on already computed results to the next operator in the chain record by record. A record in this context describes a computational result. Be it a full 2D field or a single average for e.g a month or even a full 3D field. The size of the record is thus defined by the type of operator. As example we assuming a dataset with temperature values for the years from 1990 to 1995 with daily data points. Further, we assume a CDO call that computes the monthly averages for the dataset. This would result in the `monmean` operator receiving records in the form of a single 2D field of temperatures for each iteration of the operator. The operator then would have to sum the records until a full month has been received. After that it can produce the new 2D field and provide that field as its output as a single record.

In context of chaining operators CDO has multiple types of operators. These types effect how they can be chained and depending on the type operators can behave very differently when considering a CDO chain:

- **Operators that take no input**
  This are mostly operators that generate new data. For example, `-topo`, an operator that generates the topography of Earth with several options such as size and grid type.

- **Operators with a fixed number of inputs**
  These operators take a predefined number of inputs. E.g. `add` which takes 2 and `hourpctl` which has three.

- **Operators with a variable number of inputs**
  These operators can have any number of inputs more than 1. They perform the same task on each input and mostly combine the inputs into a single output.

- **Operators generating files from a single output** (Obase-Operators)
  Obase-Operators use the provided output-name-base to process and split the input into multiple files. E.g. `-splittime file experiment_timesteps_` processes a file and splits it into all the contained timesteps while writing the files experiment_timesteps_$tsid_1$ to experiment_timesteps_$tsid_n$. An example would be the `-splitdays` operator, which in combination with the previous name base example would generate multiple files (e.g. temperature_1990-22-9 , temperature_1990-23-9, temperature_1990-24-9 ...)

2.3.2. CDO Internals

In this section we introduce the relevant internal architectural structures contained within CDO. Each concept will be brought up later in the implementation and requirements sections and will contribute positively or negatively to the feasibility of the original requirements. First we will introduce the broad structure and after that we will go over the components in more detail.

The general structure of CDO consists of three structural sections (see fig. 2.2). One is the command-line related systems like the setting and declaring of *options* and the *parser* which translates command-line inputs into actual CDO node structures. The second is the *process manager* and the *CDO-processes* as well as the required structures for *inter process communication*, the *factory* and all related building blocks and the *base class* for *modules* as well as the *nodes* which model the communications structure for later process execution. The third is its *operators* which are implemented and grouped as CDO-*Modules*. The CDO-*Modules* are defined in their respective files and, using the *registration* component from the second block, are automatically registered with a *factory* at runtime where each operator gets its own entry inside the operator table. This is done statically which means that the registration is done at runtime before the main function is called. When registered, *operators* are available to the *parser* or other top level systems via said *factory*. Once available, the entries are used by CDO *nodes*. These nodes make up the *CDO-node tree* and are used for handling errors and mismatches or missing sections like missing in- or outputs. Once the tree is verified the CDO-processes assumes that everything is correct. The tree is then passed to the *process manager*. Which in turn starts and manages each *operators* own *CDO-process* which is in this context synonym to a thread. Inside CDO these *CDO-processes* are contained in a class called *Process*. We found that the naming conventions inside CDO were extremely confusing and as such decided that for this work we would call this structure ModuleBase instead of Process as this clears up most of the confusion that we encountered. We also are using *ModuleDefinition* for the in CDO used name *Module* to better distinct between a *Module* (implemented) a *ModuleDef* (Defining Core Attributes) and the *ModuleBase*. In the following explanations of the components we will list the name describing the component followed in brackets by their actual name in CDO, items without brackets kept their name for this work.

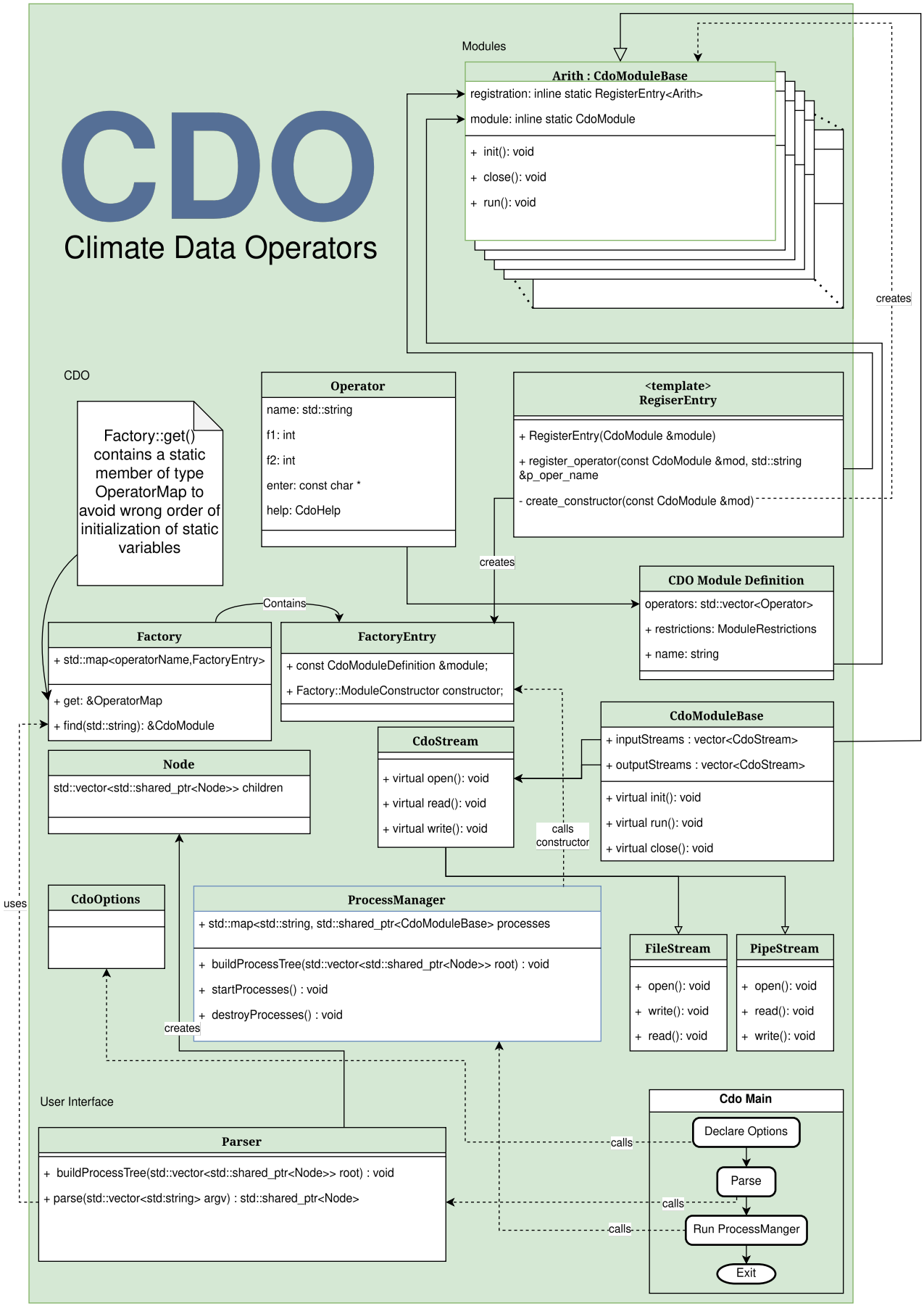

Figure 2.2.: CDOs internal structure

- **ModuleDefinition(CdoModule)**
  Modules define the shared properties between a set of operators. These properties include the restrictions that some operators have, such as that they must be the first operator in the chain or that they cannot be chained and must be a leaf in the node tree. The details of these components can be seen in listing 2.3. In this figure we can see the vector `operators` which is filled with the operators at time of instantiation of a module. *Aliases* can be seen as well and each *alias* is registered as if an operator has been declared twice just with another name. The member `mode` declares whether the module operators should be listed in the documentation and other listings of operators. Internal modules are mostly in development and not ready for release or modules that are only intended to be used by developers. The `number` variable declares if the module can handle complex numbers while the `constraints` member declares restrictions such as that the operator can only be first in a chain or cannot take in or output.

```
std::string name; // Module name                                    1
std::vector<oper_t> operators; // List of contained operators       2
std::vector<Alias> aliases; // Aliases                              3
short mode;    // Module mode: 0:intern 1:extern                    4
short number;  // Allowed number type REAL/COMPLEX                  5
module_constraints constraints; // only first, only files as input ... 6
```

Listing 2.3: The components of CDO-Modules

- **Help Definitions** The CDO repository contains a single file where all help strings are stored. Each help is stored in a `std::vector<std::string_view>` where each line is an entry in the vector. The help text itself is organized into five categories:
  - NAME: Contains the names of the included operators and a brief one line description
  - SYNOPSIS: Lists in and out files and parameters and their positions and usage.
  - DESCRIPTION: A description of the Module and its general purpose.
  - OPERATORS: The list of operators with an extra description what the specific operator does.
  - PARAMETERS: The parameters, the expected format/type as well as a short description.

  These help entries are organized on a Module base where multiple operators are described at the same time. An example can be seen in listing (2.4), showing the different sections and their contents, as well as the structure that they are contained in.

- **Operators(oper_t)**
  CDOs operators are mostly defined through the module in which they are declared. The *oper_t* structure that represents a single operator is thus rather small and only contains the name of the operator itself, as well as two variables used for various purposes, such as function pointers, options, or specific values. In total *oper_t* contains only five member variables and the last two are used for displaying a string on user input (*enter*) and a reference to the help string.

```
const CdoHelp ArithcHelp = {
    "NAME",
    "    addc, subc, mulc, divc, minc, maxc - Arithmetic with a constant",
    "",
    "SYNOPSIS",
    "    <operator>,c  infile outfile",
    "",
    "DESCRIPTION",
    "    This module performs simple arithmetic with all field elements of a dataset",
    "    and a constant. The fields in outfile inherit the meta data from infile.",
    "",
    "OPERATORS",
    "    addc  Add a constant",
    "          o(t,x) = i(t,x) + c",
    "    subc  Subtract a constant",
    "          o(t,x) = i(t,x) - c",
    "    mulc  Multiply with a constant",
    "",
    "PARAMETER",
    "    c  FLOAT  Constant",
};
```

Listing 2.4: The help string defined within CDO for the operators of Arithc. The Module has multiple operators and the documentation is done on module level so that each help would be listed in the default help.

```
class oper_t
{
  public:
    std::string name;
    int f1 = 0;
    int f2 = 0;
    const char *enter = nullptr;
    const CdoHelp &help = default_help;
}
```

Listing 2.5: Structure defining CDO operators

- **CdoStream, FileStream, PipeStream**
  The streams of CDO manage the data exchange between operators as well as the loading and writing of files. For file I/O the `FileStream` class exists, which delegates all operations to CDI. For inter operator communication the `PipeStream`s employ a CDO class that handles the data movement between threads. Both classes implement the interface defined through the `CdoStream` class which serves as the abstract interface for streams within CDO.

- **ModuleBase(Process)**
  With the model base CDO manages the inter operator communication as well as the execution structure. The model base provides storage and access to CdoStream, which are the constructs that manage data transfer between operators. More importantly the base defines the virtual *init*, *run* and *close* methods that are later implemented by each specialization of this Type. Instantiation of these is done by passing a processed *node* as input, which then informs about required connections to other operators. At this point it is expected that the processed node is correctly parsed and constructed. Each instantiation of a base represents a process/thread at execution time.

```
class Arith : public ModuleBase                                                    1
{                                                                                  2
public:                                                                            3
  using ModuleBase::ModuleBase; //using defined constructor of ModuleBase         4
  inline static ModuleDefinition module = {                                        5
    .name = "Arith",                                                               6
    .operators = { { "add", FieldFunc_Add, 1, ArithHelp },                         7
                   { "setmiss", FieldFunc_Setmiss, 0, ArithHelp } },               8
    .aliases = {},                                                                 9
    .mode = EXPOSED,     // Module mode: 0:intern 1:extern                         10
    .number = CDI_BOTH,  // Allowed number type                                    11
    .constraints = { 2, 1, NoRestriction }, // 2 inputs, 1 output, No Other       12
↪ Restrictions
  };                                                                               13
  inline static RegisterEntry<Arith> registration = RegisterEntry<Arith>(module);  14
                                                                                   15
private:                                                                           16
  // other members required for the behaviour are declared here                    17
                                                                                   18
public:                                                                            19
  void                                                                             20
  init() override                                                                  21
  { /* impl details */ }                                                          22
  void                                                                             23
  run() override                                                                   24
  { /* impl details */ }                                                          25
  void                                                                             26
  close() override                                                                 27
  { /* impl details */ }                                                          28
}                                                                                  29
```

Listing 2.6: Shortended version of the Arithc module.

- **Module Implementation (Uppercase Files)**
  While Modules are declared via the ModuleDef type their behaviour is defined in the module implementation file. These files are within the CDO-repository and by internal convention marked through being the only files with a name that has the first letter capitalized. Through inheriting the ModuleBase the Module is forced to implement the virtual *init*, *run* and *close* methods. Inside the three functions the different behaviours for the operators are distinguished through if statements. Operators define specific names by which the specific behaviour of a module can be selected. Modules self register with the *factory* through the inline declaration of a static `RegisterEntry` (see: lst 2.6, line 14).

- **FactoryEntry**
  FactoryEntries only use is to provide access to the module of the entry as well as to store a factory function for later use.

- **Factory**
  CDOs factory is a namespace in which the std::map which at runtime contains all registered operators. This map is statically declared inside the `Factory::get` function (see lst:2.7).

  With this CDO circumvents problems in context of initialization order when using static variables. Whenever the get function is called the first time the static member *factory* will be instantiated which in turn means, that access is only possible after instantiation as the get method is the only access point to the map.

```
typedef std::map<std::string, FactoryEntry> OperatorMap;
OperatorMap &
get()
{
  static OperatorMap factory;
  return factory;
}
```

Listing 2.7: CDOs factory declaration and the get function that contains it.

- **Nodes**
  CDO *nodes* are the intermediate step between the user, defining the intended workflow, and CDO building the actual *process tree*. Nodes represent the connections between the *operators* without modelling the actual communication. They serve as a separation between the errors that can happen while constructing the workflow from the internal errors. For this they can store an iterator to the, from the command-line, received vector of operators. This is done to properly indicate where in the input the error occurred. When constructing nodes, they receive the information about the in and output requirements and limitations about the chosen operator from the factory.

- **Parser**
  The exact functionality of the parser is, the context of this work, less relevant as we will show that we do not need it to create the required node tree. In general in takes the command-line interface inputs as a string and converts it into a process tree. Important to note for our later implementation is that the parser handles almost all errors in regard to unknown operators, wrongly applied operators, missing arguments and other for this work less relevant errors.

- **Options**
  CDO has its own version of a getopt like command-line options Parser. It allows to declare short and long form options, their effect and what happens on missing arguments or as default as well as optional arguments. In listing (2.8) we can see an example.

```
CLIOptions::option("num_threads", "P")
    ->describe_argument("nthreads")
    ->add_effect([&](const std::string& argument) {
        Threading::numThreads = parameter_to_int(argument);
        })
    ->set_category("Multi Threading")
    ->add_help("Set number of OpenMP threads.");
```

Listing 2.8: CDOs system for declaring options and settings

  This example shows how an options named *num_threads* is declared. Which sets the number of openMP threads to be used. With `describe_argument` a short description for the argument of the option is defined, with `add_effect` the behaiviour of the options is set and with `add_help` a more help test is defined.

2.3.3. Existing CDO Python Interface

In this thesis we will provide a Python interface for CDO. In turn we have to address the already present interface and its capability and implementation. The current implementation already provides a method for returning Xarray data structures. A problem with this method is that is

relies on the creation of temporary files. As we discussed in the motivation this is not feasible in the long term. While all operators are available only a single one can be called at a time via a Python object. For chaining a string has to be used to declare the rest of the chain which is then used as input for the Python object (see lst. 2.9).

```
cdo.setname('random',
    input =  "-mul -random,r10x10 -enlarge,r10x10 -setyear,2000 -for,1,4",
    output =  ofile,
    options =  '-f nc')
```

Listing 2.9: Old Python interface example with chaining.

While all operators are available they are only available through manually querying the CDO executable for the list of operators. In this list the number of outputs and inputs need to be specified and with the list being printed on the standard output it needs to be captured and parsed. Which in turn adds additional code to the interface implementation and additional risks for compatibility with changes inside CDO. Since CDO is used through subprocesses and the path to the executable is a variable this kind of interface offers the option to change the underlying CDO which can be beneficial when older or newer versions are required for the task at hand.

2.4. Yet Another Coupler (YAC)

To realise the coupling of Earth system model components within ICON the DKRZ and MPI-M developed YAC[14] in a joint initiative. The library allows to exchange 2 dimensional fields between a source and destination grid. With this it allows the coupling of model components while the coupling is agnostic to grid structure or element types which in turn also means that exchanges between regular and irregular grids is possible with YAC. In addition YAC provides Interpolation, neighbourhood search and interpolation routines. These can be used to e.g. interpolate the lowest layer of the atmosphere simulation with a size of 4000x2000 down to a size of 2000x1000. In this example this would be the size of the ocean grid, thus providing the tools required for the intended coupling between the two components.

Figure 2.3.: YAC logo from the YAC Github repository[39]

2.4.1. YAC-Output

Recently, with the preparation for handling the enormous requirements of exascale computing, YAC has been extended to not only be able to couple the different models but also to serve as an access point to computed data. This system, in the form of a python library, allows to request fields with their grid from a specific ICON component. E.g. `atm_output,icon_atmos_grid,temp` will request the temperature of the atmosphere module with the specified grid. Since YAC-output uses Python as its user interface this enables new workflows to be used within an ICON run. For example the direct processing of data in-situ via other Python libraries such as Xarray, Pandas[7] or other data processing tools. Further, YAC-output provides its data in a format suitable for direct conversion into netCDF datasets which creates a convienent interface for followup workflows and this work.

2.5. ICON

ICON is a modelling system for weather forecast and climate prediction. The German Weather Service (Deutscher Wetter Dienst DWD) and the the Max Planck Institute for Meteorology (MPI-M) initiated the development of ICON in 2001. The name ICON is from from the usage of spherical grids. Where ICO stands for the base form of the grid, an ICOsahedron and the N from the the non-hydrostatic dynamics[27]. Since the inception the number of participant has grown, including the German Climate Computing Centre (Deutsches Klima Rechen Zentrum DKRZ), the Swiss Federal Institute of Technology (ETH) in Zurich as well as the Karlsruhe Institute of Technology (KIT). ICON consists of multiple modules[31]:

Figure 2.4.: ICON logo from the MPI-M webiste[43]

- Numerical weather prediction model ICON-NWP
- Climate atmosphere model ICON-A
- Land model JSBACH (Nabel et al., 2020)
- Ocean model ICON-O (Korn et al., 2022)
- Atmosphere aerosol and chemistry model ICON-ART
- Marine biogeochemistry model HAMOCC

The ICON Earth system model (ICON-ESM) consists of[31]:

- ICON-A
- JSBACH
- ICON-O
- HAMOCC

The different modules of ICON do not necessarily use the same timesteps, which means that the simulation step in each module can process a different Δ_t. The different timesteps are caused by the different physical mediums each module simulates, e.g. water moves a lot slower than air on average inside the simulations. For this the simulation times need to be different for efficiency reason. Simulating water with the same Δ_t as air would result in huge amount of unnecessary compute steps as the extra calculated timesteps have no real effect on the accuracy of the simulation and could have been skipped. The modules are generally not connected and for coupling of these modules, meaning the exchange of data between them, YAC is used. Where YAC takes care of the interpolation between timesteps and grid sizes. The general I/O of ICON is done via CDI. Parallel output is done via CDO-PIO which is a extension of CDI that allows the parallel writing and reading of files. Additionally YAC-output is being used to access specific variables and grids for more specific post processing. Yaco is another library which enables I/O for ICON. Yaco was written to provide the possibility to write data directly into FDB[19].

ICON can use multiple dedicated I/O processes for aggregating and writing files. The experiment setup of ICON is dictated by user defined parameter files[32]. These contain, among other configurations, the so called namelists. These can be used to define for example outputs, and to define which variables are calculated and how. Namelists that are used to define the outputs are called *output_nml* within the setup and each *output_nml* creates a new output file if no other options are chosen. In 2024 ICON has been made open source and while the full history of ICON-repository is not yet available, which means that working on ICON itself in a collaborative fashion is problematic, the code of ICON is fully available under the permissive open source licence (BSD-3C).

```cpp
#include <iostream>

#include <pybind11/numpy.h>
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
#include <pybind11/stl_bind.h>

class UserDefinedClass{
    public:
        void hello_world() {
            std::cout << "hello world" << std::endl;
            }
};
PYBIND11_MODULE(ExampleModule, "ExampleModule")
{
  // pybind11 also handles exceptions thrown from C++
  pybind11::register_exception<UserDefinedException>(pybind11Module,
      "UserDefinedException");

  // Defining the class and getting a reference to that class representation
  auto class_rep = pybind11::class_<UserDefinedClass>(ExampleModule,
↪  "UserDefinedClass");
  // adding a method
  class_rep.def("hello_world", &UserDefinedClass::hello_world,"prints hello world);
  ExampleModule.def("example_module_func", [&](UserDefinedClass &ref) {
↪  ref.hello_world(); }

}
```

Listing 2.10: Example for PyBind11 Usage

```python
#import ExampleModule as example

example_instance = UserDefinedClass()
example.example_module_func(example_instance)
```

Listing 2.11: Resulting Python code from the ExamplePyModule

2.6. Pybind11

Pybind11[18] is a lightweight, header only library that allows to map C++ routines and structures to Python and vice versa. It is very similar to Boost.Python[36] but removes the long list of dependencies that Boost contains. PyBind11 has integrated support for numpy data structure which makes it in context with scientific data very easy to use. Generic C++ types are supported as well, including converting from C++ vectors or maps to Python lists and dictionaries. This makes passing arguments straight forward in most cases. PyBind11 also allows, due to being header only, direct integration into the code base instead of being linked later on[33]. An example of a with Pybind11 defined module can be seen in listing (2.10). The example creates a Python module named *ExampleModule* containing the class definition for the *UserDefinedClass*. After compilation we can then use the defined structures in a Python script (see: fig (2.11)).

Figure 2.5.: PyBind11 logo gitlab repository[39]

With Pybind11 writing extensions is simple and convenient, keeps the boilerplate code to a minimum and allows to enable large sets of library functionalities to be available in Python, all while being lean and easy to maintain. The automatic conversion of C++ types to Python data structures further reduces the amount of work, test and maintenance developers have to do.

2.7. NetCDF

The Network Common Data Form (NetCDF)[2] is a set of data formats and tools that was created to facilitate the creation and sharing of scientific N-dimensional data. The data format of NetCDF is self describing, machine-independent and in binary form. Each data set is accompanied by metadata that describes the content. Popularized through the geoscience community it enjoys wide acceptance in the scientific community and netCDF is supported by libraries in many programming languages, including C, Python, C++, Java and Fortran. The core data structure is the data set. Each of the datasets is structured into dimensions, variables and attributes each given an unique name. These names, as well as name the content descriptions for units, history and other metadata are backed by standardized convention such as the Climate and Forecast (CF) Conventions[3]. With these conventions mappings from coordinates to variables are provided.

2.8. Xarray

Xarray[17] was created to provide scientists with labelled N-dimensional data structures. Xarray keeps and maintains the consistency of labels with each operation. While providing a multitude of features to the users:

Figure 2.6.: Xarray logo from the Xarray website[52]

- **Label-based indexing**: Xarray objects support both integer- and label- based lookups along each dimension. Xarray objects additionally have named dimensions, for access without relying on positional ordering
- **Arithmetic**: Arithmetic between Xarray objects vectorizes based on dimension names, automatically looping over each distinct dimension.
- **Aggregation**: Calculation of statistics
- **Plotting**: Xarray employs the usage of Matplotlib where possible making plotting easy and combination with the often used Matplotlib unproblematic.
- **Missing Data**: Smooth handling of missing data Interactivity with pandas[7]: Xarray objects convert to and from pandas objects for interaction with the PyData ecosystem.
- **Serialization and I/O**: Xarray supports direct serialization and I/O to several file formats:
  1. netCDF
  2. pickle
  3. Grib1/1(read only)
- **Out-of-core computation**: Dask[12] can be used as backend, making the processing of larger than memory data sets possible.

2.8.1. Data Structures

Xarray uses NetCDF as a basis for its data model and provides a portable serialization format through two main data structures.

The Data Array is the first of those two structures. In the form of a dictionary it provides multiple attributes to describe the characteristics of the data it contains.

- data: is a N-dimensional array holding the values
- coords: a dict-like container of arrays of coordinates that label each point
- dims: names for the dimensons for each axis
- attrs: a dictionary (ordered) holding user defined metadata
- name: an user defined name for the array

The *attrs* and *name* attributes build on the functionality of pandas[7] and are used to further describe the contents of the DataArray.

The Dataset is the second main data structure of Xarray, which is Xarray's equivalent of a pandas DataFrame. A Xarray Dataset is a dict-like container with Data Arrays as its content and it allows to store the data in-memory. Datasests have four main properties.

- data_vars: a dictionary of DataArray objects for variables
- coords: a dictionary of DataArray objects for labelling points from *data_vars*
- dims: dictionary for mappings from dimension names to the fixed length of each dimension
- attrs: a dictionary holding arbitrary metadata

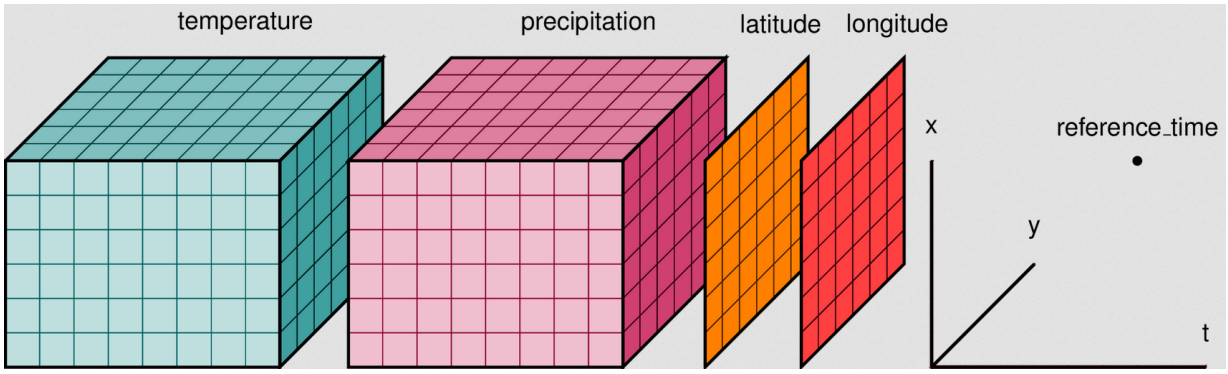

Figure 2.7.: An example structure of a netCDF or Xarray dataset
1D coordinate Dimensions: time, y,x
3D data variables: temperature, precipitation
2D coordinates: longitude and latitude
Zero-dimensional reference time.
Image source:[51]

With all these attributes and features Xarray provides a way of easily handling complex and large datasets. With the arithmetic functionalities and it's plotting capabilities Xarray is a powerful tool in the hands of each scientist.

2.9. Lustre

Lustre[15] is a client-server, parallel, distributed, network file system commonly used in the HPC environment due to it providing a modular and scalable storage framework. Lustre provides Posix *standard-compliant Unix files system interface. The core systems of Lustre

Figure 2.8.: Lustre logo from the Lustre website[42]

are Clients, Object Storage Targets(OST) and Meta-
Data servers[22]. The goal of Lustre is to efficiently
connect the compute nodes of a cluster (clients) to the storage system the cluster employs. Most compute nodes do not have large amounts of storage and thus the generated data is required to be moved fast to avoid problems with memory or an unnecessary waiting period for the data to be stored somewhere. Files in lustre are made up of two parts. A metadata inode and a set of data objects.

- **Lustre File/Inode/Data Object**
  A file in Lustre is comprised of a metadata inode object and one or more data objects. Where an inode is a data structure in Unix-style files systems that are used to describe file system objects like files or directories and a data object is a simple array that stores buld data associated with a files content.

- **Management Server (MGS)**
  The Management Servers serves as the central information source for all Lustre file systems present in a cluster. It stores the configuration of the system and provides it to all other Lustre components. Lustre Targets provide information via the MGS and the clients retrieve said information from the MGS[15],[26].

- **Metadata Servers (MDS)**
  MDS make metadata available to the Lustre clients. MDS handle the names and directories in the corresponding Lustre file system while providing the network request handling for on or multiple Metadata Targets[15].

- **Metadata Targets (MDT)** The MDTs, of which there is at least one in each file system, store the metadata on storage attached to the MDS. This includes among others filenames, directories and permissions as well as file layouts. A Target on shared storage targets can be available to multiple MDSs. Multiple MDS can not access the MDT simultaneously. MDS failover is a mechanism where if an active MDS fails a second node can replace it and maintain availability to the clients[15].

- **Object Storage Target (OST)**
  Objects can store user file data. Where one file can be an object or divided into multiple objects. Each object is then stored on an separate OST[15].

- **Object storage servers (OSS)**
  The OSS provides the file I/O service and network request handling while storing and providing the file contents to Lustre clients[26].

- **File Layouts**
  Files are stored in chunks an then distributed over the storage targets. The stripe count is directly translatable to the number of OSTs used to store the file[26]. This allows files that are larger than the capacity of a single storage target. More importantly this allows to read from the chunks simultaneously and thus using the bandwidth capacity of multiple storage mediums[26]. There are two different file layouts that can be used[26]:

  - Standard File contents are striped over multiple OSTs and *distributed equally*
  - Composit
    More involved striping patterns where a *multiple different patterns* can be applied on a single file.

Read and write operations first go through the Linux Virtual File System (VFS) and form there into the Lustre LLITE layer which implements the VFS interface. There are two possibilities for the request to proceed, the first is a request to meta data. This will be routed to the MDCs. Data request are routed to the OSC. Finally, the requests are sent to the Lustre servers and then over the wire via the Lustre Networking (LNet) subsystem. When received by the lustre server the requests return back up until they either reach the OSS component in case of a data request or in case of a metadata request until they reach the MDS. Both the OSS and MDS are multi-threaded and are capable of handling requests for multiple storage targets (OSTs or MDTs) on the same server. Data requests are then send from the OSS to the OBD Filter Device and then to the OSD which then handles access to the backend file system.

2.10. Levante

Levante is the HPC machine of DKRZ. Since all our testing and experiments as well as the resulting program and its additions are manly, but not only, intended for said system we will show the specific configuration as well as the nodes, gpus and the of course the Lustre setup that is employed. Levante is a BullSequana XH2000 supercomputer using the 3rd generation of AMD EPYC CPUs (Milan), NVIDIA A100 GPUs, and a 130 Petabyte DDN file system. An overview of the different nodes can be seen in table (2.2). In the table we can see a total of 3042 nodes, of which 2982 are CPU nodes with varying memory sizes and 60 are GPU nodes with different GPUs. When working the user can select a node depending on the required features. For the CPU node that means the maximum available memory while for GPU nodes the feature denotes the GPU-type the selected node provides. Each node is equipped with AMD EPYC processors with 64 processor cores each. In total Levante has a total memory of more than 800 terabytes[41].

node type	configuration	count	feature
CPU(standard memory)	2x AMD 7763 CPU; 128 cores in total, 256 GB main memory	2670	256G Memory
CPU(more memory)	2x AMD 7763 CPU; 128 cores in total, 512 GB main memory	294	512G Memory
CPU(fat memory)	2x AMD 7763 CPU; 128 cores in total, 1024 GB main memory	18	1025G Memory
GPU	2x AMD 7713 CPU; 128 cores in total, 512 GB (or 1024 GB) main memory; 4x NVIDIA A100 (either 40 or 80 GB of memory)	60	a100_40 / a100_80

Table 2.2.: Overview of Levantes nodes and their features in terms of memory/GPU-type

All nodes are connected to a Nvidia Mellanox Infiniband HDR100 fabric, building a pruned fat tree. The connection between compute nodes and storage components has a data transfer rate of up to $200\mathrm{GB\,s^{-1}}$[40]. In total levante provides 130TB of disk storage. Additionally the machine is connected to a tape storage for long duration storage of datasets. The Peak performance of Levantes compute power is listed as 2.8Pflops/sec[41]. Levante employs Lustre as its file system which manages the 130PB of storage distributed over at 40 OSTs, where each OST has approximately 90 HDDs, assigned to it. These 90 disks are then again separated into 9 RAID units where 8 disks are used for data and 2 for parity.

2.11. Workflows

Within the context of CDO an ICON there exist multiple workflows that are relevant for our work. The general workflow of ICON, assuming we have a correctly compiled ICON version with all the features we require, consists of the preprocessing of the input data. Several data files are required to perform experiments with the ICON model. There are, depending on the choosen type of run, up to four categories of necessary data[32]:

- Horizontal grid files
- External parameters
- Initial Conditions (DWD analysis or ECMWF IFS data)
- Limited area mode requires accurate boundary conditions sampled at regular time intervals.

Together with the general configuration of ICON this data is provided via so called run-scripts. ICON contains a templating engine which provides basic configurations which create the run scripts, including required I/O setup, for different HPC systems. With the created run script and slight modifications, that add the setup for YAC-output, ICON is ready to be executed, assuming all required input files are in place. The ICON I/O workflow can be seen in figure (2.9). We included possible preprocessing to the workflow. In the figure we show the different steps like pre- and post processing and can see when the file system is used. The red arrows can represent multiple files and as such the amount of file system requests can be quite large. Ignoring the input, since we focus on the in-situ post processing, we can see the different ways ICON can produce output as well as CDI which is the main library for reading and writing, YAC-Output writing directly to disk, and possible direct post processing done within the YAC-output script. Commonly the output is first saved to disk by CDI and then reloaded for post processing. We also show the other possibilities of writing to disk with smaller red connections but most common is the post processing with CDI, Python and CDO, indicated by thicker lines.

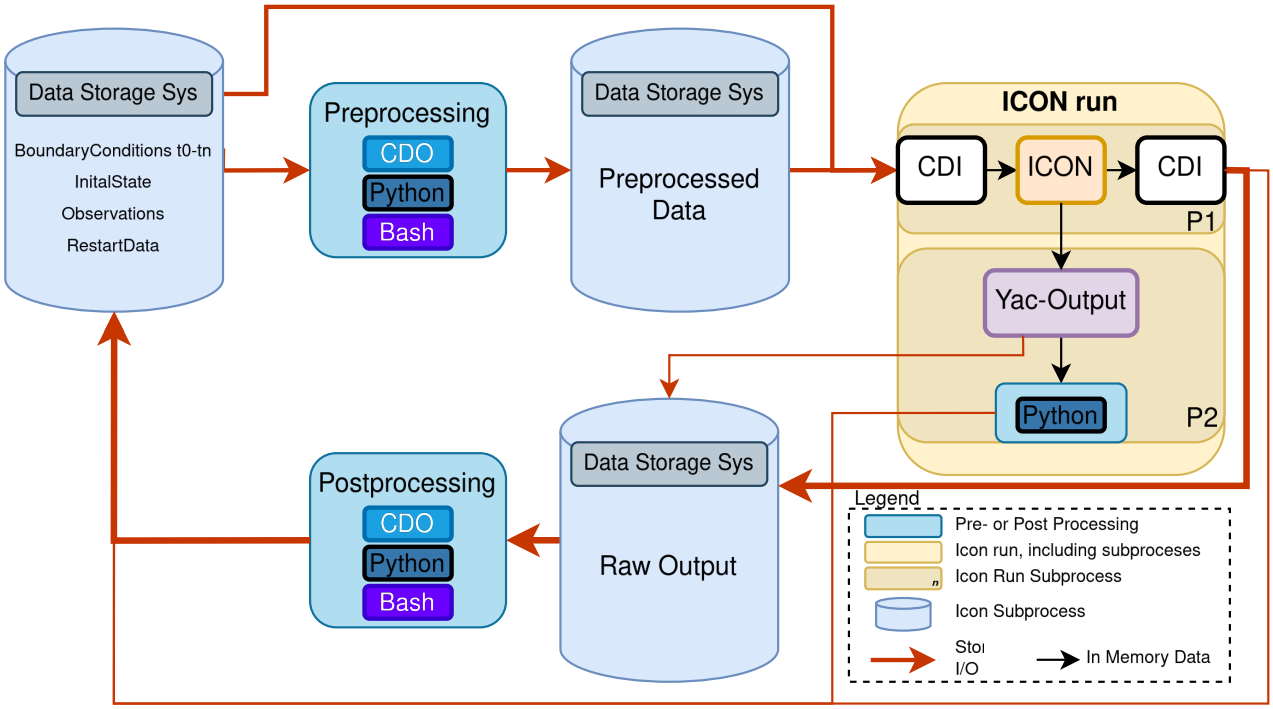

Figure 2.9.: The I/O related workflow with ICON, red arrows denote read and write operations.P1 and P2 within the ICON processes denote the separated but connected processes started within the run scripts

CDO is used in different contexts but, be it Windows, GNU/Linux, MAC or the specific setups of HPC machines the different workflows have no real distinction. The distinction that

```
cdo -subc,16.0 -selname,t ${allData} ${t_Data}
cdo -subc,35.0 -selname,s ${allData} ${s_Data}

out_t=masked_${t_Data}
out_s=masked_${s_Data}

#nts = number of timesteps
cdo div -seltimestep,${nts} ${t_Data} -selname,${maskName} -seltimestep,1 ${allData} ${out_t}
cdo div -seltimestep,${nts} ${s_Data} -selname,${maskName} -seltimestep,1 ${allData} ${out_s}
```

Listing 2.12: CDO process with multiple CDO calls. Each call has to write and load the data from disk.

```
cdo -f nc -sellonlatbox,-55,-20,-45,45 -remapbil,r450x225 -topo tst.nc;
cdo -P 16 gencon,tst.nc $f AtlBoxTOregularBox.nc;
cdo -remap,tst.nc,AtlBoxTOregularBox.nc $f remapnn\_r450x225\_$f;
```

Listing 2.13: CDO example that shows the generation and use of the grid file tst.nc.

can be made is the difference between direct use via command-line, scripts or the older Python interface. In all three cases the workflow is still the same. In figure (2.12) we can see that `allData`' is loaded 4 times, which means that the file system has to load and as such move the file over the system 4 times. This does not even include `t_Data` and `s_Data` wich also need to be first written and then loaded again.

With the script in figure (2.12) the files are written to disk. For plotting, for example through python, the files have to be reloaded again. We can see the resulting workflow in figure (2.10).

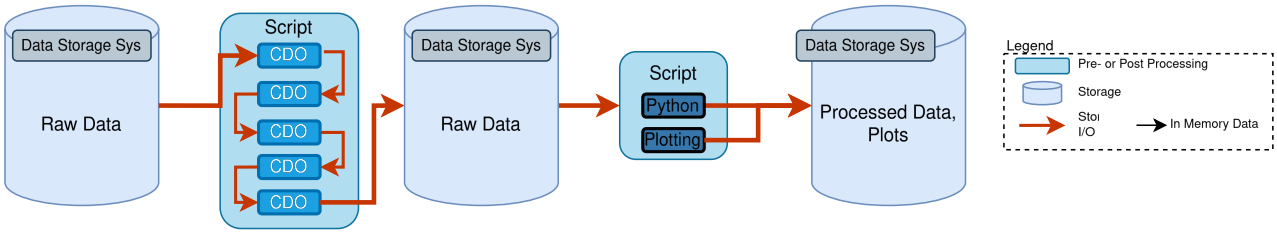

Figure 2.10.: The CDO workflow when calling CDO multiple time causing the loading and writing of files multiple times. Additionally we show another script for plotting since CDOs plotting capabilities are limited and for plotting the files have to be loaded again.

In our evaluation of the current state of Icon and its CDO usage as well as the general usage of CDO, we could find several instances where CDO was used to generate a grid file for later use to specify the type of grid for example in an interpolation or regridding operation as can be seen in figure (2.13). This gridfile is given as a parameter for an operator and is only loaded when the operator itself is executed and not when the general infrastructure of CDO is building the process tree with the I/O CDO-nodes. Instead of the CDO infrastructure these operators use CDI directly and bypass and control via CDO.

3. Related Work

In this section we will introduce similar works related to in-situ data processing, management of scientific workflows and attempts to organize the huge amounts of data that today's simulations generate. Most of the time these three are combined into a single system.

3.1. Ascent

Ascent[30] is a library designed for in-situ processing. It was designed to be lightweight in context of simulation code. This means that while it should run on the same resources as the simulation it must not drain the simulation of its resources. Another focus of the library is for the API to be simple to learn and easy to integrate for novice users. Its primary usage is intended to provide direct access to the memory of the simulation. It avoids making copies of the data and aims to directly use the simulations in-memory data. With Ascent researchers can analyse the data of the simulation while it is running and can use all of the five following mechanics to extract information about the state of the simulation and work with the data.

- Pipelines represent the data transformation and processing.
- Scenes render images.
- Extracts are used for file I/O.
- Queries provide quantitative summarization of the data.
- Triggers are conditions on which other actions are executed.

With these interoperable actions Ascent allows to describe and execute workflows.

3.2. Ophidia

Ophidia[10],[16] is a framework designed for data intensive analytics. The framework is configurable for different infrastructures such as multi core environments, HPC systems and cloud based architectures. Ophidia employs a client server architecture for distributing data analytics over several targets. This approach was chosen to circumvent e.g. memory restrictions when working with extremely large data sets. For defining workflows Ophidia provides a predefined JSON Schema in which workflows defined and then send to an Ophidia server. The Ophidia Framework is based on a relational data store to manage multidimensional scientific data. This means that it relies on the concept of datacube, a big data analytics solution that gives the user the necessary abstraction to easier deal with high dimensional data. Data is accessed via the I/O nodes exploiting a set of I/O servers that provide the parallel I/O features. Datacubes managed by the I/O servers are organized in a hierarchical structure of subsections, which are distributed

Figure 3.1.: Ophidia Logo from their website[46]

over multiple I/O servers running on different hosts. Ophidia allows operator submissions that then dispatch the job and that have managed queues and priorities.

Ophidia was introduced in 2014[10] and since then has been further developed and upgraded. In 2016 Donatello Elia et al. added in memory capabilities for their Ophidia I/O servers[13]. In 2017 Alessandro D'Anca et al enhanced the workflow description capabilities of Ophidia by e.g. adding support for loops inside workflow descriptions. In 2021 Donatello Elia et al. added server side computation for analytics[25]. The Ophidia architecture offers a set of APIs providing interoperability with different types of I/O servers. Among the supported backends are classical relational database management systems (i.e. MySQL) and the native in-memory system from the Ophidia I/O server. Through this, ophidia supports both persistent and volatile back-ends to store and manage data.

3.3. The SENSEI Generic In Situ Interface

With the SENSEI interface E. Wes Bethel et. al.[28] tackle the problem of non compatibility for different in-situ solutions. To solve this they implemented the SENSEI generic in-situ interface. It allows, once a software has been instrumented, to use it for taking advantage of a number of different potential in situ or in transit methods or tools. In their paper they can show that SENSEI allows the use of a diverse set of tools such as Libsim[6], Catalyst[11] and Ascent[30] as well as user written Python code. Especially the compatibility with Python allows for a wide range of extra methods compatibility for data analysis and visualization. In addition SENSEIs API promotes code portability and reusability. This is done through their adaptor model implementation which handles the coupling between the implementation specific code of the simulations and the native data producers and consumers. With generic data model and through the use of user-definable bidirectional mappings they allow zero-copy data access for many common use cases. More over SENSEI allows to simplify simulation codes by providing a common interface for a multitude in-situ tools.

3.4. Colza

With Colza Matheiu Dorier et al.[29] are attempting to allow in-situ visualization to be elastic in terms of used resources. With the commonly for inter-process communication used MPI library it is often not possible to change the number of nodes/cores once a task has started. Colza provides both a C++ and a Python API with which simulations can be extended with a Colza client. This client then connects to a server on which as Colza provider is running. Running processes can then request the addition of new Colza processes on new nodes. Processes can also inform the Colza system when they need less resources which then causes the removal of Colza nodes. In their work they claim to their best knowledge that they are the first to provide a tool with the elasticity they provide.

4. Design

In this section we will first define the goal for the implementation part of this work. The result, namely PyCdo, a Python interface for CDO, then has to fulfil a set set of requirements which partly are extracted from the background and partly from general software quality standards. This includes the changes to CDO and CDI and of course the design for the Python interface. From the introduced scientific context (see: section2.1) we will derive several goals for the implementation especially in the context of *usability* and *maintainability*. We will list the requirements and explain the importance of them briefly.

The goal of this work is to provide an in-situ enabled CDO Python interface named PyCdo. This Python interface should be easy to learn, easy to maintain and easy to install for the users. For the developers the changes to CDO and CDI should be improvements and not add bloat or new unnecessary maintenance problems. The interface itself should be as lean and simple as possible and should also be easy to maintain while providing the full capabilities of CDO.

From the previously introduced exascale and general HPC context we can expect a substantial benefit from in-situ processing of climate data. As shown, CDO currently does not provide its full capabilities via its Python interface and relies on writing and reading files from disk instead of using in memory methods. With the goal defined we will elaborate on the decisions that lead to it. We start with the choice of programming language in which we want the interface to be And then we will go over the Usability and Maintainability aspects of our design. After that we will introduce the feature list for PyCdo, that enables the interface to be a viable replacement for the current usage of CDO.

4.1. Enabling In-Situ

With this work we want to enable the use of CDO in an In-Situ context. For this we planned the PyCdo Python interaface as this extra layer for CDO would allow the use with Yac-output. We described Yac-output in section (2.4.1) and its capability to provide data at runtime. With the planned interface the use of CDO inside of these yac output scripts would be possible and as such an in-situ method for CDO would be available.

4.2. Language Choice

Python was chosen for the interface not only because Yac provides its dedicated output extension via Pyton but also because it is widely known and used in the scientific environments. It provides convenient access to a multitude of libraries specifically designed for the analysis and plotting of large amounts of data. Most libraries offer acceptable performance as many libraries have their cores functionalities written in c or C++. Among these libraries are also several packages that allow the extension of c or C++ packages with Python interfaces. Other languages, such as R, also provide such extension packages but are in our context not as widely used as Python. With

the wide distribution we assure that most of the developers that will work with on on PyCdo do not need to learn a new language as Python finds application in most, in the scientific context, relevant university courses[24]. Many tools already provide interfaces for Python, among them YAC, netCDF and many others used in science. Python is still an interpreted language and these types of languages bring certain drawbacks with them. The interpreter has to parse the code on each use which in turn leads to slow run times if not reduced to a minimum by using Python only for the top level while the routines that are performance critical are written in rust, c or C++. With CDO getting its interface many resource intensive calculations can now be done via CDO instead of using Python directly.

4.3. Usability

Usability was chosen as the most important design aspect for PyCdo. With the intended audience for our software being researchers all in the background mentioned aspects are relevant here. Especially the aspects about the priorities of the researchers and the non IT background most researchers come from. Our software is intended to allow the work of researchers to be easier and to contribute to the shareability of workflows. For this work we split usability into two distinct parts as each of them requires special attention for them to be fulfilled.

1. Ease of use
   PyCdo should be easy to use in terms of Documentation, Tutorials and installation. For documentation we intend to provide code completion mechanism as well as the in CDO already present documentation for all operators. The documentation will include tests which will be used as examples and material for explanations in the documentation as well as this work.

2. Installability
   With the complex field of HPC computing many tasks require multi step setups for software to work. We want PyCdo to be easily installable via the Python Package Manager(PIP)[38].

4.4. Maintainability

Maintainability is the second most important aspect for this work. Here we also took our reasoning from the background. Many developers that will later be working on our software will have a background in research. Even if they are trained software developers the maintainability of PyCdo must be as low as possible due to the use of project work and the high throughput of developers.

1. Clearly separated responsibility of the libraries/Code-Artefacts
   To ensure a long lasting maintainability, discouraging the mixing of responsibilities by splitting them properly from the very beginning will be very beneficial. Further, this will turn each module and library into a, from each other separated, organizational structure and thus making it easier to test, maintain and develop. This will in the future also help with extensibility and reusability.

2. Developer documentation
   With the developer documentation the PyCdo specific aspects will be listed. These aspects include implementation examples, howtos and general knowledge required for maintaining and developing PyCdo

3. Tests as a form of Documentation
   Tests are of course required to verify the correct behaviour of PyCdo. While the correct implementation of CDOs algorithms will still be tested by CDO itself, the availability and functionality of PyCdo will be documented through the tests. This will also serve as a source for examples on how to use PyCdo and as such will not only be part of the developer documentation but also of the user documentation. Further, the tests should document the requirements of CDO as it contains several methods and variables that need to be specified before CDO is fully functional.

4. Low maintenance cost
   For this low lines of code are our top metric. That will not mean that we will needlessly obfuscate the code but that to be added systems will be carefully evaluated if they are necessary or are already present in some of the sublibraries. Adding operators to CDO should automatically add them to PyCdo, extra work for each additional operators must be avoided. The same is true for the documentation of operators. These should be directly taken from the CDO operators modules without any extra work. Using the tests of PyCdo as a form of documentation will also reduce the amount of work that needs to be put into keeping the documentation up to date.

4.5. Functionality

Functionality wise we are aiming for a fully usable implementation. That means that all to the user visible features from the command-line version of CDO need to be available. We aim for the user to be able to rewrite their scripts without having to make compromises in functionality and even to improve the experience with the addition of the features that Python itself provides all while being easier to share and read. With the functionality we already list points that are derived from the maintainability and ease of use requirements.

1. **PyCdo**
   a) *Operator Availability*
      For PyCdo to enable the users to rewrite their bash scripts all possible operators are required as a single missing one could lead to scripts staying in bash or new scripts having to be a bash script as one required operator is not provided by PyCdo.
   b) *Loading and writing files*
      PyCdo has to support the reading and writing of files from and to disk. Without this no relevant workflow is possible. For this prototype we aim to provide all the I/O possibilities CDO provides. This includes the loading of GRIB, netCDF and all other formats CDI can handle.
   c) *Handling in memory data*
      Two of the main goals for the interface is the seamless integration with Python and the possibility to use in-memory data to circumvent using the file system unnecessarily. For this we need to provide methods to use already loaded files or created datasets. In addition the workflows we intend to provide would not work without this capability. Loading a file with netCDF,Pandas or Xarray and processing it with said tools and then processing it with PyCdo should not require any data being written to disk.
   d) *Returning in-memory data*
      In line with the avoidance of file I/O the Python interface must be able to return the results in an in-memory fashion. For the formats we intend to provide the output in the form of netCDF byte arrays. Other formats like GRIB, have been requested but

will not be part of the original goal. The same reasoning as with accepting in-memory data applies here: we want to provide the possibility to work with the by PyCdo calculated results without the need to write them to disk first.

e) *Operator chaining*
As we introduced CDO we explained that the chaining of operators is a major feature of CDO. This allows to conveniently describe entire workflows in a coherent and efficient manner. As such we see this capability to be retained in the Python interface as a must. Together with our reasoning about translating older bash scripts to Python the chaining feature is one of the most important factors for a successful implementation of our goals. Allowing the chaining to be used naturally means that all previously introduced operator types have to be supported by PyCdo:

i. Operators that take no input

ii. Operators with a fixed number of inputs

iii. Operators with a variable number of inputs

iv. Operators with the obase feature

f) *Handling operators arguments*
Many of the operators of CDO have their own set arguments. Without a mechanism to set these arguments the operators cannot be used. From this follows the implementation requirement for such a mechanism as otherwise not all operators can be supported.

g) *Options and Settings*
CDO provides a long list of options, many of them are also relevant to the usage of our system. For a fully featured Python version of CDO naturally most of these options need to be included. Some can be left out as they are not valid for in script use. Some examples of left out options are the setting for the colour of the output the help option or options related to grouping features as they all are either completely unnecessary or are replaced by the syntax of the newly written library.

h) *Xarray compatibility*
Xarray employs numpy's data structures and is an accepted and well known tool within science. Among the features of Xarray is the compatibility with many scientific data formats as well as Dask, a Python library for parallel and distributed computing already used and requested as a I/O feature for CDO. In section (2.8) we listed the features of Xarray many of these are extremely useful and with compatibility to Xarray not only Dask but also netCDF and Pandas, Xarray is well suited to be the I/O of PyCdo. In addition Xarray can already handle the metadata used within climate-science and as we already mentioned in section (2.8) it was designed with the intention to maintain the consistency of the metadata.

i) *Autocompletion in IDEs*
In terms of usability and ease of use as well as understandability, providing the capability for IDEs to show autocompletion together with the function and data type documentation is a fundamental feature for any application such as ours.

j) *Error Handling*
For our first implementation we intend to handle all errors that can happen during the construction of the node tree and the parsing of operator arguments. All further errors that CDO can handle need further reworks of CDO and we accept that these might not be handled graciously and can cause trouble in terms of user experience.

2. **CDO**

   a) *Splitting CDO into three parts*
   For the proper distribution of roles and making it easier to maintain these roles we will split CDO into three parts.

   i. CLI interface(CDO) The CLI interface will be the parser, all current options and the main function currently used in CDO. It will be on the same level as PyCdo and will use the newly created CDO libraries.

   ii. Operator library(libcdo-operators) This library will include all operators and module declarations and all module specific arithmetical routines.

   iii. Corelib(libcdo) Inside this library we will put all core mechanism of CDO. It will be the library that is used by CLI CDO and PyCdo.

   - Process Management, creation of the process tree, nodes and node creation.
   - Interprocess communication through pipes, including the threading.
   - The factory and the functions required for the automated registration of operators.
   - Systems for defining options and CDO- and CLI- arguments.
   - Any core kernels and the definition and grid specific functions that are not in CDI.

   b) *Interface to access and create factory entries and node structures*
   For CDO to be a library capable of being used inside CLI CDO and PyCdo we need to define and implement the interface to be used by other software.

   c) *Handling in memory streams*
   CDO currently provides two types of streams, one for inter process communication and one for file I/O. We will add a third type that will handle the netCDF memory stream capabilities.

3. **Libcdi**

   a) *Addition of in memory handling (netcdf)*
   for CDI the only changes included in the design are the addition of extending the current use of netCDF inside CDI to be able to open and handle in memory netCDF files.

5. Design Realization

In this section we will show the, for the Realization of our goals, required modifications to CDO and CDI and the newly added PyCdo and how we fulfilled each of our stated requirements. The section is split into the 2 main parts where the first section regards the changes to CDO and CDI and the later regard the implementation of PyCdo and the surrounding infrastructure like installability via pip and documentation related topics.

5.1. CDO and CDI

To realize the in memory handling we extended the FileStream class of CDO. The FileStream class implements the CdoStream abstract class and provides all file I/O for CDO through CDI which in turn provides a netCDF interface. NetCDF's interfaces for in memory and normal file I/O do not differ, except when opening and closing files. For this we used CDO's FileStream as a basis. From there we implemented the required open,write and close functions inside CDI. Where the `streamOpenWriteNCMem` and `streamOpenReadNCMem` functions both use `streamOpenNCMem`.

```
int streamOpenNCMem(int ncidp, char mode);
int streamOpenReadNCMem(int ncid);
int streamOpenWriteNCMem(int ncid);
```

Due to the similarities between in memory and regular file I/O all other functions from FileStream could be reused since netCDF does not distinguish between these two modes once the netCDF data structure is created. With CDI providing the interface for in-memory handling we could then implement the use of these capabilities within CDO. For this we first split CDO into the in section (2.3.2) mentioned 3 already logically existing blocks (see 5.2), the CLI related structures, the operators and for the third part the infrastructure and arithmetic structures. The split was done via the implementation of a new CMake build system for each of those categories. The build system handles the creation of all 3 blocks and creates the CDO executable, a libcdo shared object, as well as a liboperators shared object.

The split also ensured that interfaces must be provided which existence further enforce the separation of concerns as it is now harder to implement or use library features where they do not belong. Only one needed to be added since the factory system of CDO already provided the necessary interface for access to the operators for both the CLI CDO and PyCdo. For libcdo we

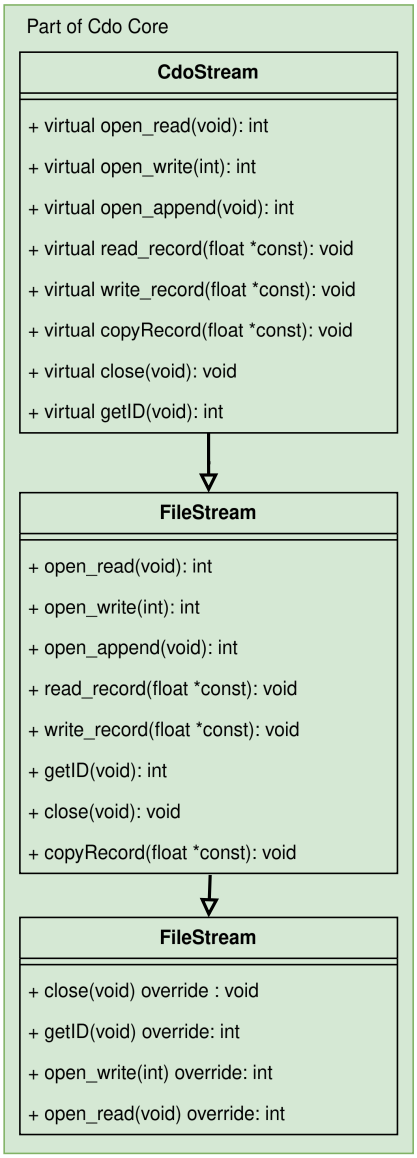

Figure 5.1.: Hierarchy of CDO streams with the newly added MemoryStream

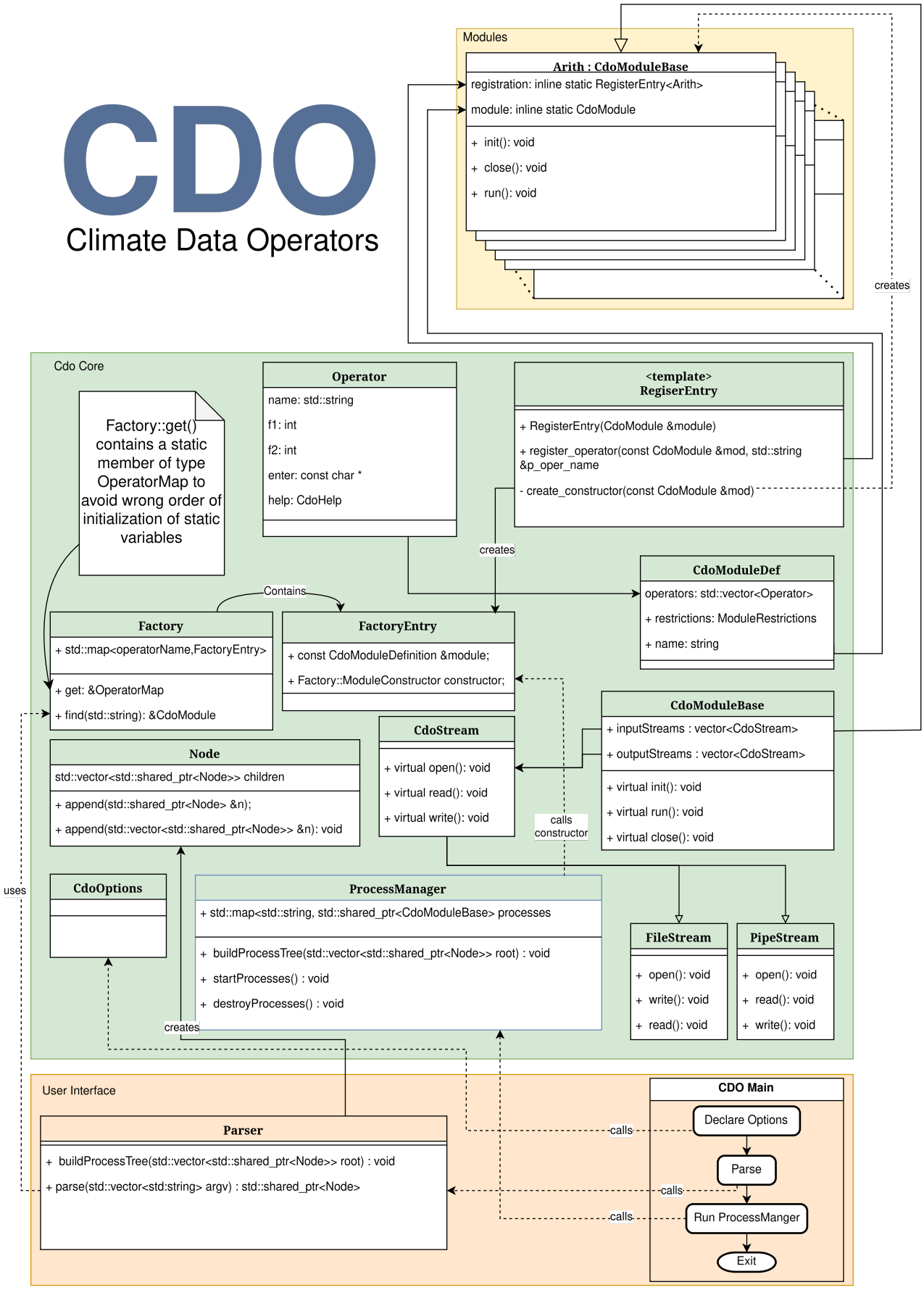

Figure 5.2.: The separation of the CDO components into distinct shared libraries. Shown are the three components: A library that contains all modules with the corresponding arithmetical functionalities, another that contains all CDO's internal infrastructure. Such as the factory and the base class for modules and process and at last the CLI interface of CDO

created a user-interface containing the routines and data structures required to make use of the options, modules/operators and nodes as well as the processManager. Additionally we had to extend the ModuleBase class. This class handles the creation of its in-and output streams. To handle memoryStreams the corresponding functions that add these as I/O were added.

5.2. PyCdo

With the changes to CDI and CDO we were able to use these new interfaces in combination with Pybind11 to create a Python class that extends CDO's nodes. We implemented the feature for adding input to operators through the functions `add_data` and `add_file`, where `add_data` is used for adding data that is already in memory to the workflow and `add_file` for adding files on disk. Files on disk are directly handled via CDI and CDO and the method is just a wrapper to distinguish the two types of input and sets the necessary flags. With access to the factory provided and all operators being registered beforehand (see: 2.3.2) we were then able to iterate over all entries in the factory (see: 5.1). We then defined a function for each operator and named them after the operator. Each of these functions will, when called at runtime, return a predefined PyCdo-node for later use and from this we will be able to build the process tree. The extension of the CDO-Node can be seen in figure (5.3).

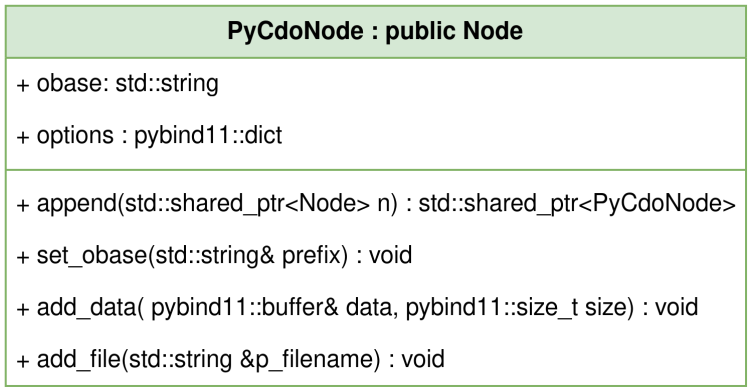

Figure 5.3.: PyCdo extension of CDO Node

```
for (auto& f : Factory::get()) { //access to the factory and iteration over content
  std::string name = f.first; // get operator name
  CdoModule mod = f.second.module; // get module struct
  m.def( // Pybind11 module method to add definitions to a class
      name.c_str(), // name of the function we will create
      // the follwing is a C++ lambda which is the function that creates
      // the CDO-node wrapper class and returns it
      [mod, name](std::string args) {
        auto new_node = PyCdoNode(name, args, mod.constraints);
        return new_node;
      },
      pybind11::doc(help.c_str()), // setting the documentation
      pybind11::kw_only(), // only keyword arguments are allowed
      pybind11::arg("args") = ""); // add keyword for operator arguments
}
```

Listing 5.1: An example version of accessing CDO's registered modules. This turns them into Python functions which return a PyCdoNode. These nodes can be put together to form a workflow tree.

Creating CDO workflows is implemented through the `append(PyCdoNode other)` function. By appending the nodes to each other a by the *ProcessManager* parsable node-tree is created. Error handling is also mostly done through the append function. Adding to many nodes to another node that would go over the required inputs results in an error. The same goes for attempting to chain operators that have no output or operators without input. This is realized by the CDO-nodes which form the basis of the PyCdoNodes. Inside the base these checks are

```
def addc(*, in: numpy.ndarray[numpy.float32] = ..., args: str = '') -> PyCdoNode:
    """
        addc  Add a constant
              o(t,x) = i(t,x) + c
    Keyword arguments:
    in   -- list of nc data (as bytes)
    args -- operator arguments as string (see PARAMETERS)
    DESCRIPTION:
            This module performs simple arithmetic with all field elements of a dataset
            and a constant. The fields in outfile inherit the meta data from infile.
    SYNOPSIS:
        addc ,c  infile outfile
    PARAMETER:
            c  FLOAT  Constant
    """
```

Listing 5.2: The help string defined within PyCdo for the operators addc. The Module has more than one operator and the documentation is done on operator level so that only the relevant information is shown while parts containing information for other operators are removed.

already done so that for PyCdo no extra checks are required. In context of the output of a run we have already explained that if there is missing output, in memory outputs are created.

Operator Documentation is done similarly to the way we register the operators. As we were also able to extract the documentation for each operator and add it to the created methods. For this we were required to implement the functionality of splitting the operator documentation into parts. This was necessary since most operator documentation is used for more than one operator and we wanted to reduce the documentation to only the necessary parts. In listing (5.2) we can see this for a Module with multiple operators. Since CDO contains operators with more than 10 operators, the doc string would be bloated and contain irrelevant information making working with it less efficient.

Our doc strings are reduced to the relevant information. This means that we removed the *NAME* section completely and reduced the *OPERATORS* section to only the currently viewed one. The difference can easily be seen when comparing listing (2.4) and listing (5.2).

Handling In-Memory Data has been made possible through the addition of the MemoryStream within CDO. The way netCDF data is handled by, CDI through the use of the netCDF-c[45] library and through it passing of byte arrays is sufficient for CDO and CDI. For Pybind11 and the interface between C++ and Python it was required to define a suitable `pybind11::capsule`. Such a capsule handles the memory management when passing data between C++ and Python. For the capsule a `pybind11::buffer_info` info is required which is created when we extract the results from a process. The capsule also handles the destruction of the data on the C++ side so that on relinquishing the data inside the Python context the C++ resources are also freed. For parsing the netCDF data structures back and forth we make use of of the *ncid*. Inside PyCdo we create a new in-memory file for output streams when no other output is given. The *ncid* of the created object is then passed on to CDO via the MemoryStream and filled and managed by CDO until it is finished.

Executing a PyCdo workflow is done through calling `run()` function on the top node of a node-tree. This function passes the created tree to the process manager of CDO. When passing the PyCdo node on which `run()` was called is used as the root tree and all CDO-output streams are created automatically and added to the fitting processes. Obase output is also done here and if encountered the process get the appropriate flags assigned to internally handle the creation of the required outputs. When the outputs are created the processing is done with no differences to the CLI-output. After the processing is done the root process is queried for its

```
for (auto& [option_name,opt] : CLIOptions::get_options()) {
    if (option_name.size() > 2) {
      auto& argument = opt->argument;
      std::string description =
          std::accumulate(
              opt->description.begin(),
              opt->description.end(),
              std::string(),
              [](const std::string& a, const std::string& b)
                  { return a + b + "\n"; }
                  );
      if (argument.default_value.size() > 0) {
        argument.value = argument.default_value;
      }
      if (opt->hasArgument) {
        option_name = "set_" + option_name.substr(2);
        m.def(option_name.c_str(),
            [&](const std::string& value) {
                  argument.value = value;
                  opt->execute(); },
          pybind11::doc(description.c_str()),
          pybind11::arg("value") = argument.default_value);
      } else {
        option_name = "enable_" + option_name.substr(2);
        m.def(option_name.c_str(),
                [&]() { opt->execute(); },
        pybind11::doc(description.c_str()));
      }
    }
  }
```

Listing 5.3: The PyCdo mechanism to create the CDO settings. It can be seen that two distinct method types are created. One for options that have no arguments and one for options with arguments.

outputs streams and the in the streams stored netCDF in-memory data structures are gathered and returned as a list of byte arrays. Contained in these byte arrays are the values together with their meta data. Important to note is that with every run call a new process-tree is created from the root node downward. This makes each run call independent from each other in terms of execution, in files and out memory buffers.

PyCdo Options and settings were added by duplicating the use of the CDO command-line arguments system (see: 2.3.2) and removing the command-line only related options leaving only the CDO core related ones. With the options declared we can process the registered options like we already processed the operators and their documentation. We were able to iterate over each option, check if they have arguments and deepening on the argument count could create `set_<option_name>` for options with arguments or `enable_<option_name>` for options without arguments. These functions could then be documented automatically with the already existing help text from the options.

Options are done for the entirety of the CDO library. That means set options are in force for each execution of a node-tree call.

Installability is provided through the *setup.py* file contained in the root of the PyCdo project. With this *pip*, a package manager officially recommended by the Python Packaging Authority[48], can be used to install PyCdo. The setup.py describes and defines the package including the build process. For this to work pybind11 contains an example[37] how to create the setup.py in a way so that it builds the C++ code via CMake and then integrates the shared object into the Python package.

```
class CMakeExtension(Extension):
    def __init__(self, name: str, sourcedir: str = "") -> None:
        super().__init__(name, sources=[])
        self.sourcedir = os.fspath(Path(sourcedir).resolve())

class CMakeBuild(build_ext):
    def build_extension(self, ext: CMakeExtension) -> None:
        # [...]
        # 48 lines ommited: cmake execution

setup(
    packages=['PyCdo']
    # [...]
    # 6 lines ommited
    packages=['PyCdo'],
    setup_requires=['numpy'],
    install_requires=["numpy"],
    ext_modules=[CMakeExtension("PyCdo")],
    cmdclass={"build_ext": CMakeBuild},
    include_package_data=True,
    package_dir= {"PyCdo": "."},
    package_data = { 'PyCdo': ['__init__.py','*.so', 'py.typed', '*.pyi'] },

    classifiers=[
        'Development Status :: 2 - Beta',
        'Intended Audience :: Science/Research',
        'License :: OSI Approved :: BSD License',
        'Operating System :: POSIX :: Linux',
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.4',
        'Programming Language :: Python :: 3.5',
    ],
)
```

Listing 5.4: Code snipped from the setup.py contained within the root of the PyCdo repository. CMake execution is handled in the class definition of CMakeExtension and CMakeBuild.

IDE Support was done with the help of pybind11-stubgen[47]. This tool creates Python stub files that contain the public interface of a Python module. These include classes, variables and functions and their types[49]. The generated file can then be used by IDEs to provide autocompletion. In our research for methods to generate these files we found that there apparently is no easy way to include the generation within the build process. That means that we had to manually execute pybind11-stubgen and then add the file to the repository. The reason for this is, that the tool requires an already installed Python package to generate the stubs. The setup.py provides no clean methods to integrate the generation of the required stubs after all files are installed.

6. Evaluation

In this section we will evaluate our work and show which of our goals we achieved and which are partially missing or could not be fulfilled. We will go over the workflow changes our new interface brings. After that we will list our design choices and iterate which could be realized and which had to be scraped.

6.1. Workflows changes

For ICON, with the by us provided new Python module, the workflow described in section (2.11) can now be simplified. Now in-situ post processing is possible with CDO through PyCdo. Comparing figure (6.1) with figure (2.9) from section (2.11) we can see the changes than the in-situ capability brings. With the new workflow the post-processing can be done in-situ which removes the need to write to disk within the simulation. With this the writes after the post-processing outside of ICON are also removed. Further, we can see that bash scripts can now be eliminated from the post-processing. In addition we can see that the transfer from ICON to the post-processing with PyCdo is now done in memory instead of with file I/O.

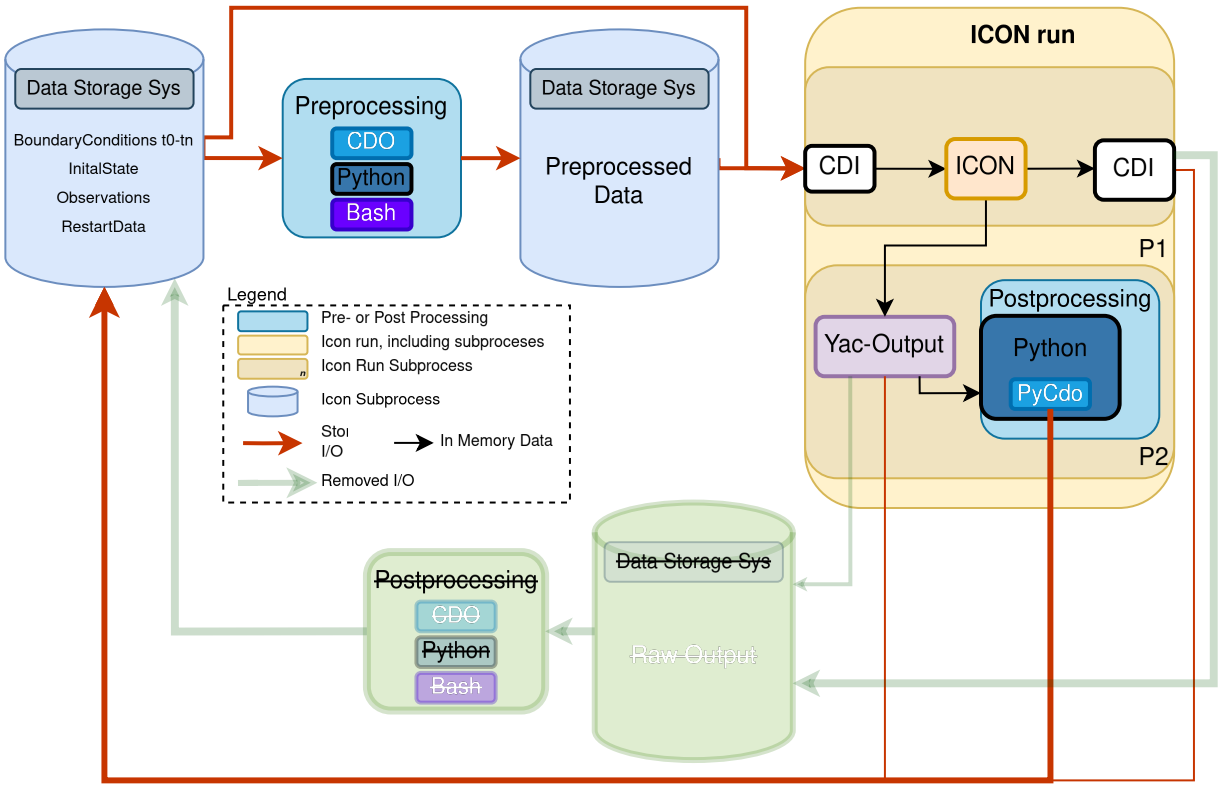

Figure 6.1.: The, though PyCdo, newly possible I/O related workflow with ICON, red arrows denote read and write operations.P1 and P2 within the ICON processes denote the separated but connected processes started within the run scripts

For CDOs scripting the changes are are similar, where it was required to save the processed data to disk in between CDO calls, it is now possible to hold the data, assuming enough is present, in memory. In the best case this even reduces the number of required files and scripts.

In figure (6.2) we can see the changes and also that with the new workflow the need for the first script that contains the CDO calls is eliminated. We have to mention that the removal of that script is already possible with the old CDO Python interface but it would use file I/O where our system uses its in-memory capabilities. The main improvement can be seen within the Python script. Where all data movement is now in-memory. I/O.

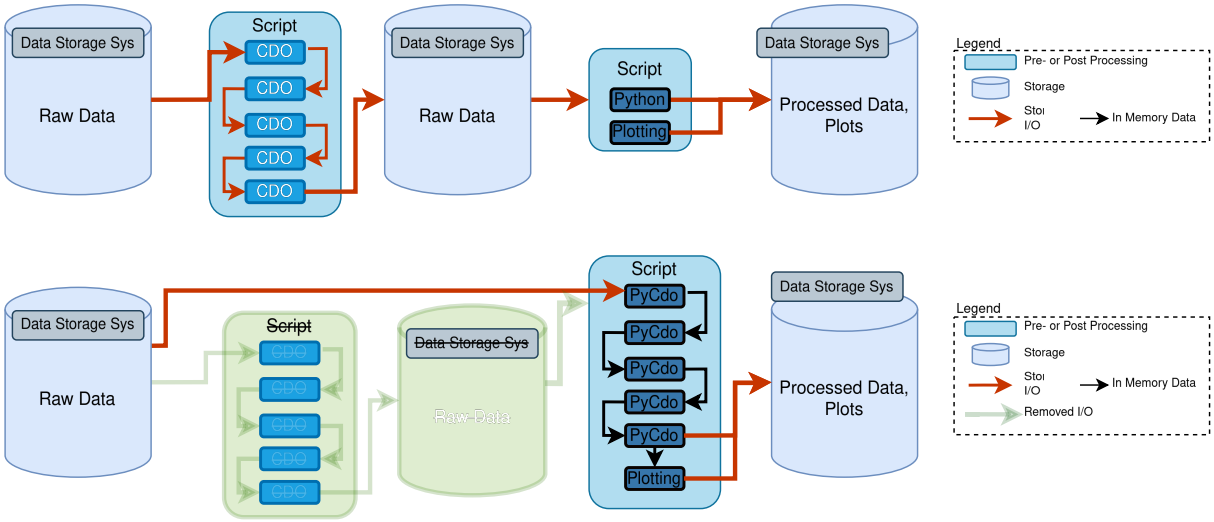

Figure 6.2.: Comparison of workflows between the CLI-CDO and PyCdo showing removed participants and the now possible use of in-memory processing.

The workflow within the scripts also changes with the new interface. Chains can now be created independently from their execution. The `append` `add_file` and `add_data` function together with the general structure and behaviour of the nodes allow the construction without execution.

6.2. Features

In section (4.5) we introduced all features that are required for PyCdo to be considered fully featured. In the following paragraphs we will evaluate if these requirements could be met.

- **Operator Chaining**
  We were able to implement the Operator Chaining feature of CDO. In section (5.2) we could show that we extended the CDO *node* class within PyCdo and could use them as the tools for constructing the workflow tree. Once the tree is constructed CDO can use its internal structures to execute the given task.

- **Operator Availability**
  We were able to include every single operator from CDO in PyCdo in section (5.1) we show how we were able to make all operators available and that the factory of CDO made this task relatively simple and straight forward. Together with Operator Chaining and our PyCdoNode structure we were able to include every type of operator mentioned: obase, multiple inputs, not output. Due to the use of the CDO's errors and the module restrictions we could also provide operators that have special restrictions and/or limitations.

- **Handling In-Memory data**
  PyCdo is capable of handling its own output as input as well as already loaded netCDF data. Through making use of the Pybind11 feature to seamlessly transferee numpy data from and between C++ and Python together with the netCDF-c library and further

```python
import PyCdo as c
import netCDF4 as nc4
import xarray as xr
ds = xr.open_dataset("test.nc")
input = ds.to_netcdf()
addc = c.addc()
addc.add_data(data=input,size=len(input))
result = addc.run() #PyCdo returns an in-memory result

info = PyCdo.info()
info.add_data(result[0],result[0].nbytes) # and the result can immediately be reused
```

Listing 6.1: PyCdo add_data function allowing in-memory data to be added to the workflow.

Pybind11 features that make transfer of data possible (see: 5.2) we were able to implement most I/O tasks of CDO to be able to use in-memory data. In the listing (6.1) we show how this feature can be used. Within these figures we can see how to add, already with Xarray/NetCDF loaded, data as well as how data returned by PyCdo can directly be reused for other PyCdo routines.

- **Xarray Compatibility and File I/O**
  Through the use of pybind11 inbuilt exchange protocol for numpy arrays between Python and C++ PyCdo is capable of receiving and returning netCDF data in form of byte arrays. Xarray provides the `to_netcdf()` function which return a for PyCdo suitable array, which then can directly passed to PyCdo as inputs. To use the convenient dict like syntax of Xarray/NetCDF the user is required to create a new `Dataset` and fill it with the returned memory. Since Xarray is based on netCDF the compatibility between PyCdo, netCDF and Xarray is automatically given.

```python
import PyCdo

topo = PyCdo.topo()

topo_results = topo.run()

addc = PyCdo.addc(args="1")
sub = PyCdo.sub()

sub.append(addc)
sub.append(PyCdo.topo())
sub_results = sub.run()

with open("result.nc",'wb') as file:
    file.write(sub_results[0])
```

Listing 6.2: How to write PyCdo results to disk.

```python
import PyCdo
import netCDF4 as nc4

topo = PyCdo.topo()
topo_results = topo.run()
res = topo_results[0]
ncRes =
↪  nc4.Dataset('file.nc',memory=res)
print(ncRes["topo"])
```

Listing 6.3: PyCdo results can be used with netCDF4 datasets which allows convenient access to the contents

In (6.3) we can see that the contents of the in the in listing (6.2) written data are correctly recognized when querying the metadata of CDO.

Loading files from is available through the `add_file` function and we could show in section (5.2) that the extension of the CDO nodes allows for CDO to handle the files through CDI.

- **Operator Types** All operator types listed in section (1e) are supported by PyCdo. With the use of the Node class from CDO we could directly access CDO capabilities for all

```
cdo -sinfo result.nc
   File format : NetCDF
    -1 : Institut Source   T Steptype Levels Num    Points Num Dtype : Parameter ID
     1 : unknown  unknown  c instant       1   1    259200   1  F32  : -1
   Grid coordinates :
     1 : lonlat                  : points=259200 (720x360)
                              lon : -179.75 to 179.75 by 0.5 degrees_east  circular
                              lat : -89.75 to 89.75 by 0.5 degrees_north
   Vertical coordinates :
     1 : surface                 : levels=1
cdo    sinfo: Processed 1 variable [0.00s 46MB]
```

Figure 6.3.: File content of the results from listing (6.2). We can see that the returned data keeps its metadata and has the expected contents.

types. Only for the obase-operators we had to introduce an extra function and an extra member to the PyCdo nodes to properly distinguish the different types.

6.3. Usability

- Installability
  We added the required files to allow installation via pip (see: 5.2) in the PyCdo repository. We were able to install PyCdo through `pip install .` called from within the repository. Upload to the Python package manager is still outstanding but download is possible from the MPI-M gitlab server with:
  `git clone --recursive  git@gitlab.dkrz.de:m300433/PyCdo.git`.

- Autocompletion
  Due to the in section (5.2) shown steps we were able to provide autocompletion for any Python IDE that can use files with a pyi suffix. In figure (6.4) we can see parts of the list of operators that are provided. All functions and data structures are, through the use of pybind11-stubgen, included. What we could not provide is automatic generation of the required pyi files. These have to be generated by the developer. A Python module has to be installed before these files can be generated via pybind11-stubgen and because of this we were not able to implement the pip-setup in such a way that this file is automatically generated at compile/install time.

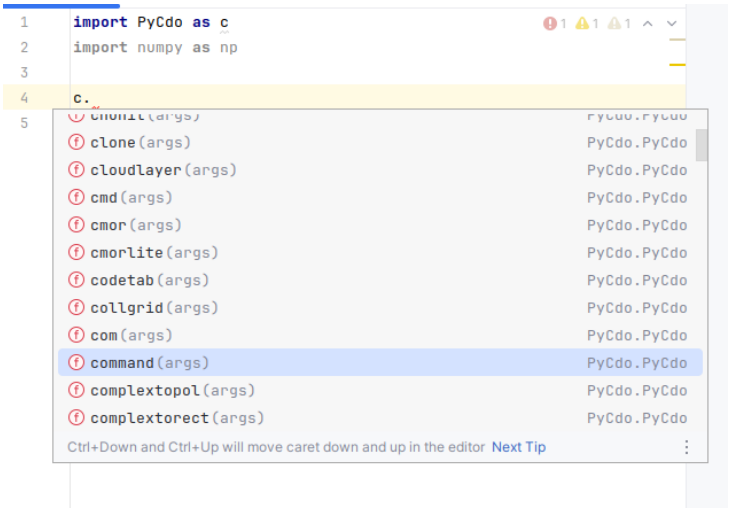

Figure 6.4.: Working autocompletion with PyCdo, shown inside PyCharm, and editor for Python scripts

- Operator Documentation

Together with the autocompletion we were able to include descriptions for each operator that is already documented in CDO. In section (5.2) we have shown that we were able to format, split and add the descriptions that are stored within the CDO factory. Through this most operators have all their relevant information contained inside the Python doc string. In figure (6.6) we can see this feature working. The figure shows the documentation for the operator add. While the documentation is not fully translatable from CLI-Interface to Python-Interface, the current state of the documentations increases the searchability of operators as well as all relevant information. We can also see the amount of in- and out-puts that this operator requires. A description what the operator does as well as a list of possible extra operator-arguments can be seen in figure (6.5).

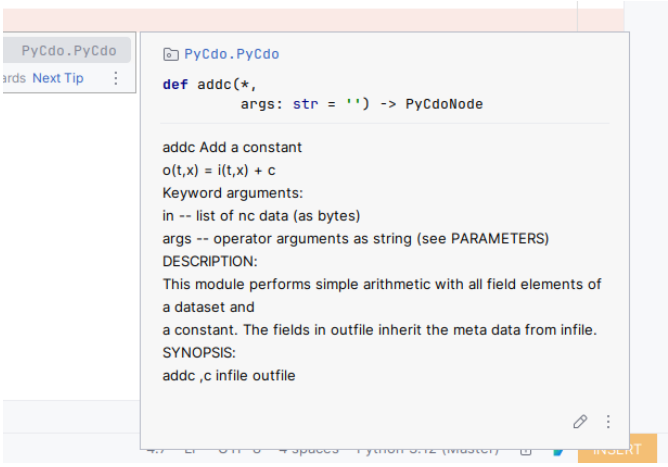

Figure 6.5.: Working documentation details with PyCdo, shown inside PyCharm, and editor for Python scripts

Figure 6.6.: Working documentation with PyCdo, shown inside PyCharm, and editor for Python scripts

The implementation of this documentation feature is the most bloated and convoluted of PyCdo. This was necessary since the documentation is done on module level and contains all information about the module. Since CDO defines these inside of a `std::vector<string_view>` and no subdivisions are made, the extraction of the relevant information caused the required new functions within CDO and PyCdo to be more convoluted than maintainable. While the extraction of the relevant information is work intensive the access to the documentation data was a huge boon for this work. Since CDO

provides the help inside the repository and as C++ strings we were able to use these to easily access the documentation strings within PyCdo. We highlight the ease of access due to our requirements in terms of maintainability. The ease of extraction and the fact that we are able to reuse all documentation from CDO are a huge benefit and reduce the work required for updating the documentation negligible. This also means that future maintainers of CDO do not need to update PyCdo manually when adding new operators or new documentation. This maintains the documentation inside CDO as the single source of truth.

- User Documentation
  In the readme as well as the test folder within the PyCdo repository we give examples and explain the usage of PyCdo. Among the explanations we show how to load files with netCDF or Xarray and then pass the data on to PyCdo. The creation of note trees and the passing of operator arguments are shown and explained. Due to the implementation of the autocompletion feature all of the public function within PyCdo have at least a documentation string and are discoverable by using the editors exploratory features. In contrast to CDO which does not have out of the box autocompletion, PyCdo is already a huge improvement in context of discoverability of operators and options. PyCdo only contains a single additional data structure. The nodes are simple and only have five public function relevant for the user. Since we create the Documentation while creating the bindings between the C++ node and the Python representation, the documentation of these operator is shown inside Python editors.

- Tests
  Rudimentary tests have been added to the PyCdo repository. These are not all encompassing and should be extended. Functions, the creation of node trees and the passing of data to and from PyCdo are tested. The effect of setting options of CDO or the behaviour of repeated execution of node trees are not tested. While the tests give a rough overview of how PyCdo is intended to work they do not fully express the capabilities, pitfalls or intentions behind PyCdo. As a form of documentation they are lacking and have to be extended in the future.

- Error Handling
  Pybind11 allowed us to register the exceptions thrown from CDO. With these the process manager and the nodes most errors relevant for this work are handled. In section (5.2) we showed that the extension of the CDO-nodes already covered the errors and that outputs are generated automatically. For operators with variable inputs we added additional checks within PyCdo. Output that has multiple files, which are only obase operators, the same principle of auto generated outputs is applicable. Since PyCdo does never write to file itself, for these operators there is no error source in terms of wrongly created node trees.

6.4. Maintainability

PyCdo Interface LOC			
lines	words	bytes	file name
241	792	8568	PyCdo.cpp
38	83	1049	pycdo_node.cc
25	52	552	pycdo_node.h
450	1393	18607	pycdo_def_options.cc
5	9	114	pycdo_def_options.h
759	2329	28890	total
304	927	10169	total - options

Table 6.1.: Lines of code for Pycdo.

Old Python Interface LOC			
lines	words	bytes	file name
874	3090	32935	cdo.py

Table 6.2.: Lines of code previous Python interface

Starting with the number of lines of code our implementation, although it contains more features such as automatic collection of functions, options and doc strings, is shorter than the old interface. The largest file of PyCdo is the one containing all option declarations. This file contains almost no logic in is generally low in maintenance. Excluding this file we can see an even heavier difference. In the section (5.2) we could show that no extra maintenance is required when adding new operators to CDO. We have shown the implementation details that allow developers to implement new operators, options or to add documentation without needing to change PyCdo and the only action that needs to be taken is to update the, with git registered, commit to the newest version of CDO and recompile and distribute PyCdo with a new version tag. This is due to the mechanism used to automatically extract the information from CDO. As long as the formats within CDO do not change there is no extra work to be done. In addition to this the provided but limited documentation and test cases allow make the introduction to PyCdo easier. Combining this with the low amount of lines of code we reached our goal of an easy to maintain extension to CDO.

6.4.1. Splitting CDO

We were able to split CDO into parts by implementing a new build system using CMake. CDO was split into two shared libraries (liboperators, libcdocore) and the CLI interface of CDO. While we were not able to reorganize the folder structure to match this pattern we were able to provide distinct libraries which will cement the different blocks and serve as deterrent to mix the responsibilities of each section for now. Further, improvements, especially the folder structure changes will have to be part of the work of turning this prototype into a fully realized software.

6.4.2. Developer Documentation

This work describes all intentions, reasons and the implementation. As well as the benchmarks and its results. At release of this work a shortened version of this works will be added to the PyCdo repository and will serve as the developer documentation of PyCdo until everything is moved into a proper Wiki.
In the repository are tests that document the features and capabilities of PyCdo they are also intended to be examples for not only the users but also the developers where they can familiarize themselves with the usage of PyCdo.

6.4.3. Additions to CDI and CDO

We were able to keep the additions to CDO and CDI almost minimal. With CDI we fully succeeded in the additions being unintrusive and parallel to the regular execution to CDI while adding the required capabilities for PyCdo. The addition of the in-memory handling in CDI could also prove beneficial for future developments. For CDO we mostly succeeded, the addition of the MemoryStream also runs parallel to the usual processes in CDO. We also had to add new functions to the CDO ModuleBase for the creation for MemoryStreams. A small flaw is that we

had to add the function that splits the module help strings into digestible chunks this function is rather complex which is why we declare it as a flaw. Although the possibility of splitting the chunks a worthwhile additions as CDO can benefit from this the implementation is haphazard and a proper rework of the whole help definitions, e.g. moving them into the Module files, would have been better. Otherwise we made no additions or changes to CDO which is in line with our intentions to keep the newly added maintenance requirements of CDO and CDI low. While we mentioned the flaws we found, we did not cross a border where the changes were outside of the margin that we set ourselves. Especially noted should be that we were not required to make any changes whatsoever to the Modules themselves, which means that their implementation are completely independent from the type of top-level user (e.g. PyCdo, CDO-CLI).

6.5. Test Enviroment

6.5.1. ICON Run Setup

For testing we use a pre-existing setup from the ICON repository. With help from staff from the DRKZ we were able to complete the compilation of ICON including YAC and setting up the required run-scripts. The Run we are using is a Climate Limited-area mode run used for developing and testing. The setup start a simulation over 10 days with roughly 200000 grid points. In production this setup produces seasonal weather forecasts. In the development setup it produces data over 10 days with 27 variables being written out. The variables are distributed over 10 namelists and of these namelists 3 write data for each hour. One name lists writes every 3 hours and 2 lists write every 6. The last 2 write their data only once a simulated day.

Namelist and Variable configs			
Nml Nr	var type	output	variable
1	ml	3-hourly	alb_si, fr_seaice, t_g, hpbl, aod_550nm
2	ml	daily	w_so, t_s, resid_wso
3	ml	hourly	rain_con, tot_prec, t_2m, snow_melt
4	ml	daily	tmax_2m, dursun
5	ml	3-hourly	tqc, cape_ml
6	pl	3-hourly	temp p_levels =20000, 50000, 85000
8	ml	hourly	shfl_s, sob_t, sob_s
9	ml	hourly	sob_s
10	ml	at start	z_ifc, topography_c, fr_land, depth_lk, fr_lake, soiltyp

Table 6.3.: Namelists of the ICON CLM experiment with the writing schedule and the contained variables.

With a simulation over 10 days and the given write timings and number of variables the run produces 795 files. We want to highlight that this setup is a development version and even if we extended the run from three to ten days, a production run, or a simulation over tens of decades would produce magnitudes more files.

6.5.2. Benchmarks

To benchmark PyCdo and compare it to CDO we did multiple different benchmarks. In all of them we measured the time and for all that were executed on Levante we also measured the Read and Write data from Lustre.
With the help of staff of the DRKZ, we gained access to the normally inaccessible lustre logs. We

parsed and evaluated 5 metrics from these logs where from each we extracted their respective read/write sizes in MB:

- llite_read: read data requests in MB
- llite_write: write data requests in MB
- osc_read: actually read data from the system, excluding cached data in MB
- osc_write: actually written data, excluding cached data in MB

With these metric we can directly see the effects on the read and writes of PyCDo in comparison to CDO.

B1: The first benchmark is to check for a semi realistic workflow and its behaviour. For this we took the in section (6.5.1) described ICON simulation with its 200000 grid points and created an arbitrary workflow that processes the output of said simulation. In figure (6.7) we can see the benchmark which loads all of the 795 files and processes them. The simulation writes each time step of each namelist into a single file so that these files contain multiple variables. Our benchmarks first merges all time steps of a given namelist and then adds an arbitrary constant to stop Lustre from caching results in between. For adding computation time for CDO itself we also set the missing values to -9999. This is often done in post processing and plotting for convince reasons and as such we included it with this semi realistic example. In the same CDO call we then split the now merged data into the contained variables and write these with all their time steps as a single file to disk. Each file that was written through this is then loaded again by the next CDO call. This call adds another arbitrary constant to the values in the files an then calculates the monthly mean. This is then written out to disk again. This benchmarks totals in 822 files being read in and 54 files being written in the whole process. Of these only the 27 writes after the merging and splitting and the 27 reads of these files are relevant for the comparison. In this benchmark we measured the Lustre behaviour and the runtime of the scripts. This script has been used in two different benchmarks. One where we executed the script independently 20 times to measure the time of each execution. We also did measurements on a Laptop to see differences in behaviour.

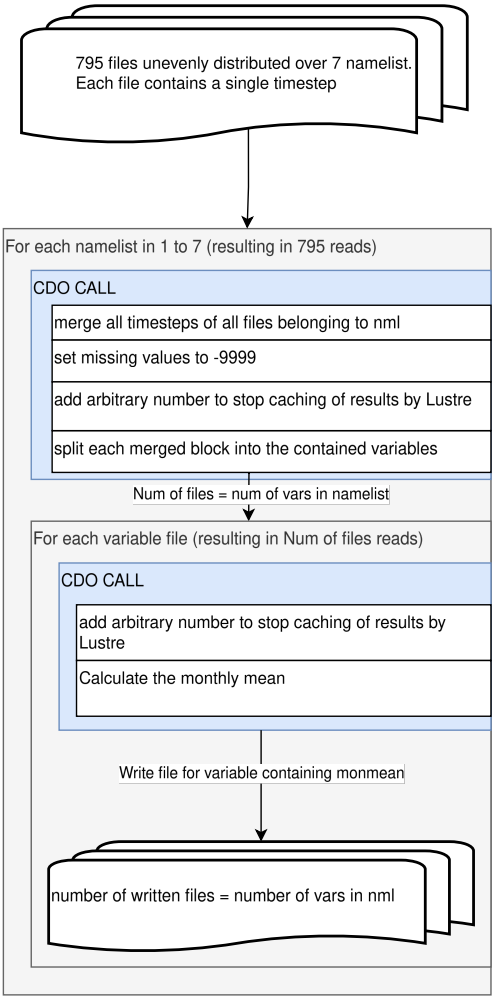

Figure 6.7.: Benchmark B1 which tests a semi realistic workflow using 795 input files, combines the time steps and splits the result back into distinct files. Each file at the end contains a monthly mean of a single variable.

B2: The second benchmark is generates and consumes its own files. With this benchmark we can highlight the difference between CDO and PyCdo as there exist no files before the execution or after that have to be written or read from disk. In the execution of CDO there will be file I/O in between CDO calls but with PyCdo we should see no file I/O at all or at least a minuscule amount. In figure (6.8) we show the benchmark. Within we create two files each with a size of 123 MB. These are written to disk and reread by the next CDO call which creates a sum over the two files. At last we read the sum and display the contents. While this benchmark is quite unrealistic it allows us to reduce noise and focus on the actual read and write operations. In these scripts we create files through the use of `topo`. The `topo` command for CDO creates a topography for Earth. With a grid size of 8000x4000

grid cells the resulting files each have the 123M in size. The `info` command for CDO scans the file and prints information about it.

B3: Our third benchmark is intended to check for the scalability of PyCdo. In this we take the Benchmark *B2* and add the possibility to set the created gridsize and the number of files that should be summed up. Since the add operator of CDO can only add two files we exchanged with `enssum`. This operator takes variable number of files and sums them. We then again load the sum and print it. As *B2* does, this script does not require input files nor does it create final output files and all created files are intermediate results that are read back in with the next CDO call. All our benchmarks were done on Levante. Further, all were done on the work partition which uses Hard Disk Drives. Due to the limited access to the Lustre logs we were forced to use a single compute (l10547) node and could not distribute our testing over multiple nodes. In the appendix we listed the compilers, versions, Python environment and the spack packages we used as well as CDO branch.

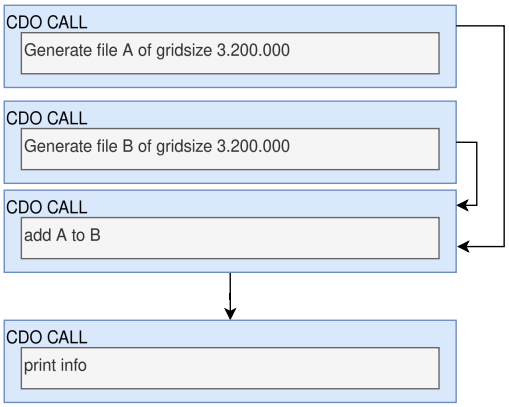

Figure 6.8.: Benchmark B2, A simpler benchmark that creates and consumes all generated data sets, resulting in a script with no I/O before or after execution of PyCDO.

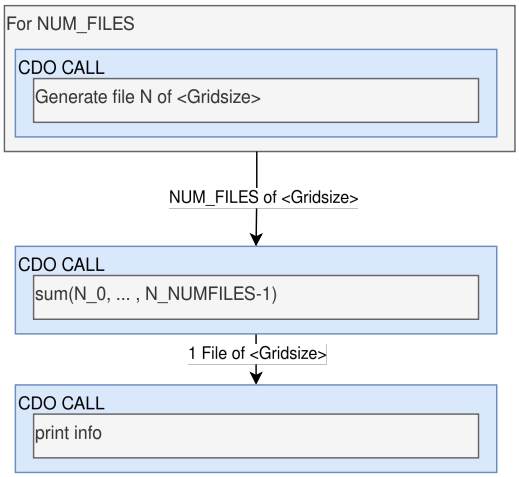

Figure 6.9.: Benchmark B3, a variation of B2 where the number of files and the grid size are scaled up incrementally

6.6. Performance

When executing our *B1* benchmark we could not find any relevant differences in runtime. Both PyCdo and CDO performed the same in context of plain run time. This is surprising since we expected PyCdo to perform better than CDO due to the removal of multiple reads and writes and thus requests to the file I/O. By evaluating the Lustre logs for our script we even found that, for the CDO scripts, the output of a CDO call is cached by the lustre system so that the access times to the file system are drastically reduced when reloading it for the next CDO call. This could not be avoided since on Levante the command to drop the caches is hidden behind root access rights. This does still

type	w/r	cdo	pycdo
llite	write	285.3759 MB	24.8223 MB
llite	read	3574.1281 MB	2203.5815 MB
osc	write	138.036 MB	17.224 MB
osc	read	1033.210 MB	1084.570 MB

Table 6.4.: CDO and PyCdo lustre log evaluation for Benchmark B1.

not explain the similar run-times as the access to memory within PyCdo should be faster than accessing cached data via lustre.

While we evaluated the lustre data we also found that the reads of CDO were three times as high as expected. We expected reads in the size of 1 GB but as we can see in table (6.6) there were read requests for 3GB of data. All while only the expected 1GB were actually read. PyCdo showed 2GB reads, so double the expected amount. We found that multiple operators of CDO read data twice. The `mergetime` operator for example is currently required to scan the files first. This scan is used to find out how large the file is in terms of number of time steps. Only when this is known the operator can successfully merge the given files. When looking this up within the code of CDO we were able to find that the operator has a section where every input file is opened again. At the time of implementing PyCdo we were not aware of this behaviour. Accidentally PyCdo supports this behaviour since the second time the file is read in is also done via CdoStreams and not through CDI. Since we implemented our MemoryStream on the basis of the CdoStreams, the behaviour of reloading files is covered by the capabilities of PyCdo and the re-reading of the files is also done from the in-memory data.

	PyCdo	CDO
	7.18s	8.62s
	10.29s	13.59s
	13.40s	7.93s
	15.88s	13.18s
	10.90s	11.31s
	9.73s	10.76s
	8.91s	8.10s
	8.51s	9.36s
	9.77s	10.29s
$\bar{x}$	10.4	10.35

Table 6.5.: CDO and PyCdo runtime evaluation for benchmark B1.

type	w/r	cdo	pycdo
llite	write	399.449 MB	0.96196 MB
llite	read	398.612 MB	2.31483 MB
osc	write	349.97 MB	0.429698 MB
osc	read	65.2083 MB	23.2009 MB
llite	write	399.449 MB	0.961963 MB
llite	read	398.612 MB	2.31483 MB
osc	write	351.916 MB	0.429699 MB
osc	read	65.2083 MB	23.2009 MB
llite	write	399.449 MB	0.961995 MB
llite	read	398.612 MB	2.31484 MB
osc	write	351.9 MB	2.9e-05 MB
osc	read	65.2083 MB	23.2009 MB

Table 6.6.: Cdo and PyCdo benchmark results. The data extracted directly from Lustre logs.

With benchmark *B2* we used operators that do not have this behaviour of reading files multiple times to confirm that the seen behaviour is not related to any errors within our scripts or PyCdo. With the use of these two operators we build a script that has no external input files that need to be read in nor do we have any final output files. The only files that are written and read are inside the script and used to pass information between CDO calls. This means that if we implement this benchmark with PyCdo, with its capability to hold all intermediate files in memory, we should see almost no reads and writes when comparing CDO to PyCdo. In table (6.6) we can see multiple results of these tests. With the generated files being of size 123MB and the reloading in each step we expect 3 instances where a 123 MB file is written. Two for the generation and one for the summation resulting in expected writes of 369 MB. The same is true for the writing as we write the same amount of data 3 times.

While we expected a read values of 0 in the table we can see reads of 23 MB. These reads are Python related including our generated shared library with 16MB and other imports. When testing this with an empty Python script that only contained the import statements we got almost exactly the same values as with the PyCdo tests run. This shows that the only file I/O done by PyCdo dann be attributed to the imports and the loading of the shared object containing all CDO routines and our interface.

llite	write 0.961873 MB
llite	read 2.31373 MB
osc	write 2.9e-05 MB
osc	read 23.1927 MB

Table 6.7.: Result of the import statements only script showing the accessed file sized by our Python imports.

To get a general sense how PyCdo compares to CDO in terms of differently sized workflows we implemented scaling benchmarks (*B3*). In these we took the self contained benchmark and changed two variables: the number of

files and the size of the files. In total we ran 40 distinct benchmarks for this. Each with a different file number and size. For each run we doubled the size and file count resulting in 1,2,4,8,16 for the counts and grid sizes, which dictate the size, of 2000x1000, 4000x2000, 8000x4000 and 16000x8000 which results in file sizes of 7.7M,31M,123M,489M respectively. The results of these benchmarks can be seen in figure (6.10) which shows the used grid size together with the number of generated grids. The runtime is shown via the coloured tiles and the time is given within these tiles in seconds. It can be seen that PyCdo seems to perform better than CDO when a medium file count with a medium file size is used. In the smallest of our tests PyCdo always performed worse than CDO. From then on PyCdo leads by a large margin until we reach 123MB files. Above these file sizes the performance of PyCdo dropped drastically with the largest tests running five times slower than CDO.

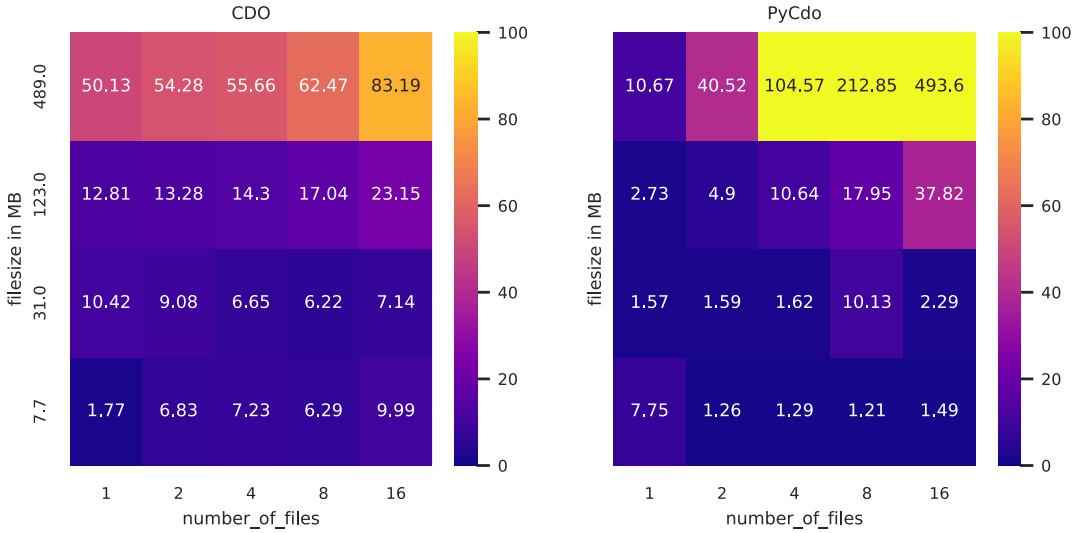

Figure 6.10.: Scaling behaviour of CDO and PyCdo. Runtime is shown in seconds. Example (1/2)

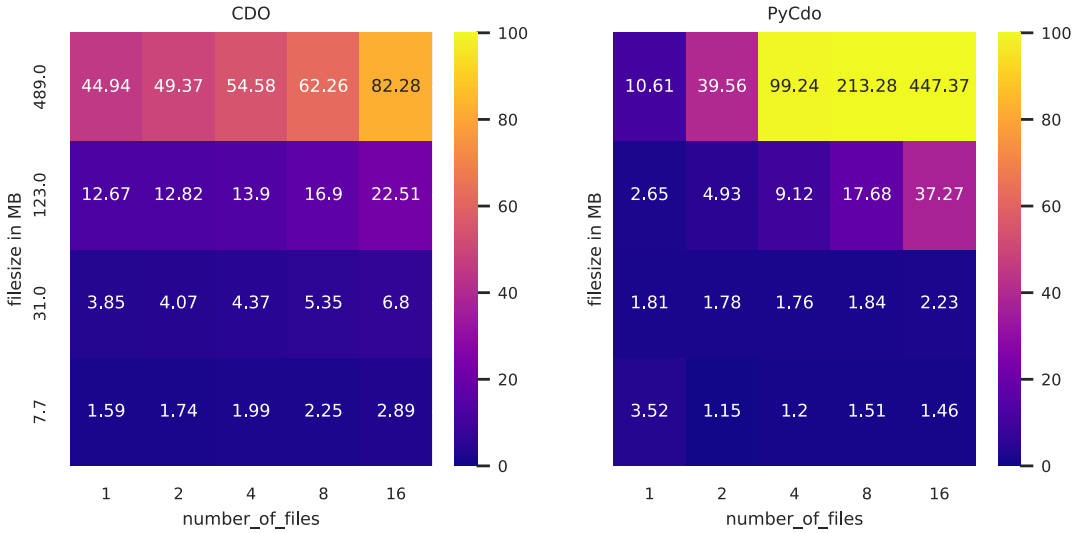

Figure 6.11.: Scaling behaviour of CDO and PyCdo. Runtime is shown in seconds. Example (2/2)

Similar behaviour can be seen in programs that exceed the memory capacity of a system, where then virtual, very slow, memory is used to provide enough space.

Since this scaling behaviour would mean that PyCdo is not usable inside of an exascale context and because this behaviour was neither expected nor did CDO show the same behaviour we re-examined PyCdo and our benchmarks. Using perf, a profiling tool, we analysed the largest instances of our scaling benchmarks. The results of these can be seen in figure (6.12)

```
Performance counter stats for 'python -X perf pycdo_scaling_bm.py  16 r16000x8000':

        474,392.34 msec task-clock:u              #    1.030 CPUs utilized
                 0      context-switches:u        #    0.000 /sec
                 0      cpu-migrations:u          #    0.000 /sec
         8,512,669      page-faults:u             #   17.944 K/sec
    90,048,894,739      cycles:u                  #    0.190 GHz
       497,164,499      stalled-cycles-frontend:u #    0.55% frontend cycles idle
       200,906,252      stalled-cycles-backend:u  #    0.22% backend cycles idle
   135,421,504,928      instructions:u            #    1.50  insn per cycle
                                                  #    0.00  stalled cycles per insn
    23,847,682,935      branches:u                #   50.270 M/sec
        27,005,310      branch-misses:u           #    0.11% of all branches

     460.399067966 seconds time elapsed

      23.280561000 seconds user
     447.873823000 seconds sys
```

Figure 6.12.: Perf results for a B3 benchmark with 16 files and a gridsize of 1280000. Compiled with optimization option O3

```
Performance counter stats for 'python -X perf pycdo_scaling_bm.py  16 r16000x8000':

         80,680.85 msec task-clock:u              #    1.236 CPUs utilized
                 0      context-switches:u        #    0.000 /sec
                 0      cpu-migrations:u          #    0.000 /sec
         8,512,520      page-faults:u             #  105.509 K/sec
    51,632,737,868      cycles:u                  #    0.640 GHz
       393,957,818      stalled-cycles-frontend:u #    0.76% frontend cycles idle
     3,221,667,786      stalled-cycles-backend:u  #    6.24% backend cycles idle
    95,601,876,767      instructions:u            #    1.85  insn per cycle
                                                  #    0.03  stalled cycles per insn
    17,497,656,564      branches:u                #  216.875 M/sec
        17,177,464      branch-misses:u           #    0.10% of all branches

      65.298905182 seconds time elapsed

      24.776073000 seconds user
      55.238504000 seconds sys
```

Figure 6.13.: Perf results for a B3 benchmark with 16 files and a gridsize of 1280000. Compiled with optimization option O2

While we could not find a direct cause or the reason for the scaling issues, we found that compiling CDO and PyCdo with a lesser optimization setting improved performance drastically. This can be seen in figure (6.13) where the reported runtime is almost 8 times lower than the report for the, with O3 compiled version in figure (6.12) Since the performance was already

improved just by changing the optimization level we added a new file size to the benchmarks while dropping the smallest benchmarks with file sizes of 7.7 MB. The new file size corresponds to a ICON run which has a grid size of 335544320 grid points. These files have a size of 1.2 GB. In figure (6.14) we can see that the performance is drastically improved and PyCdo is comparable to CDO in context the previous maximum tests of our scaling benchmarks. Where the earlier benchmarks showed time of 400 seconds for 16 files of 489 MB size, now the result is in line with CDO where CDO has a runtime of 73.8 seconds and our interface is only 9 seconds slower with a runtime of 82.35 seconds. The new, larger sized, benchmarks still show a similarly or worse drastic degradation of performance as the previously largest benchmarks. Where the newest largest test is 8 times slower than CDO.

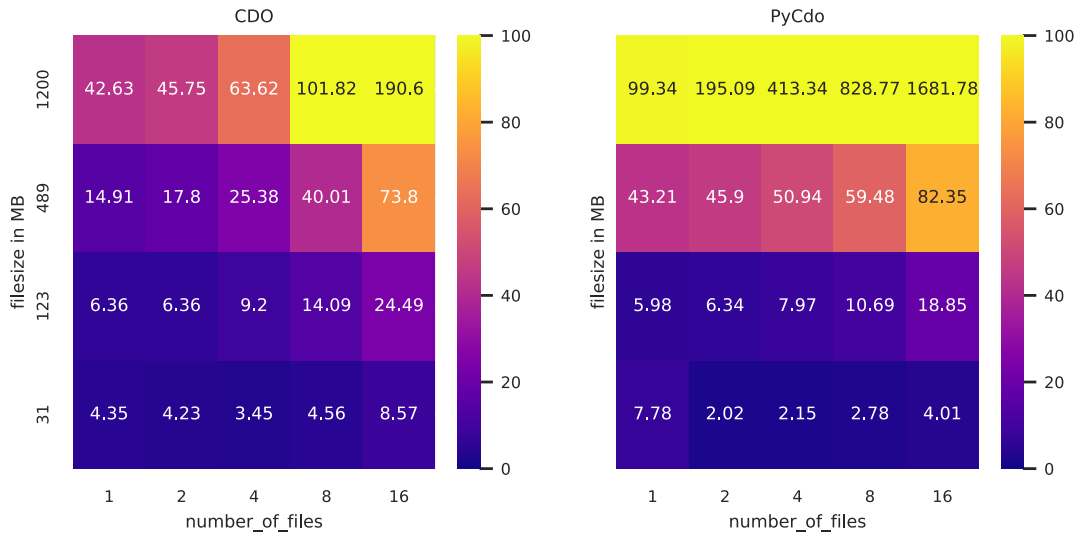

Figure 6.14.: Scaling behaviour of CDO and PyCdo compiled with O2 instead of O3. Runtime is shown in seconds.

Further, investigation could not find any reason for this behaviour. Only when using mprof[44], another profiling tool, we found that in these runs where profiling with mprof was enabled the runtime drastically improved. Nothing else was changed but the use of mprof. And the run time went down from 1600 seconds to 250s and thus getting nearer to the runtime of CDO of 190 seconds. PyCdo is still a full minute slower than CDO. We were unable to narrow down the problem further. For PyCdo to be a viable alternative for exascale data processing these issues must be fixed. The measurements we did indicate an error which mprof circumvents somehow. One theory is that the garbage collection of Python is being overwhelmed when checking for ready to release memory.

6.7. Limitations

- Building and installing PyCdo works via pip, but we encountered errors, specifically improper linking of libstdC++. These errors occur only at runtime. We could find that these errors stem from not properly configured Python paths. From our side we could not add features to find and remedy the possibility of this since it is in context of the module system of levante and spack packages. When these paths are correct the installation of PyCdo works as intended.
- Documentation of the operators sometimes contains CLI specific sections or notes. Since our focus was to add the documentation and descriptions without additional maintenance requirements we accepted this trade of. In later iterations of CDO this could be alleviate

by restructuring the way operators are declared and generalize the option handling of operators.

- Conda would have been another choice for the distribution of PyCdo, out of familiarity with the usage of pip by the author, it was decided to use pip. Conda would provide methods for managing packages, dependencies and environments. Anaconda a Python/R data science distribution perhaps would have been the better choice, but due to late discovery and the authors own knowledge of pip we could not dive into developing another package through conda/Anaconda.
- Pybind11 does not provide automatic generation of the files needed for IDE and autocompletion features. We were not able to implement the generation of such files automatically as the by us uses pybind11-stubgen requires an already installed package. This will be changed in future versions of PyCdo. We manually provided files and committed them to the package. But this means extra steps when packaging PyCdo and another possible error source: forgetting to regenerate the documentation when releasing a new version.
- In section (6.5.2) we can see that for the same result PyCdo needs more lines of code than bash. We see this as a design flaw. While bash scripts are notoriously for their low maintainability, PyCdo with the extra needed lines is at best a slight improvement. This could be remedied by implementing the '+' operator and using it instead of the `append` function of the PyCdoNodes.
- In section (2.11) we introduced an use case where operators have filename arguments. These operators do not have proper support with PyCdo. That means that these still load their files via CDI and do not benefit from in-memory capabilities.

7. Conclusion

In our work we discussed the relevance of maintainable and easy to use software in the context of scientific sofware. We were able to provide a prototype of a CDO Python interface with in-situ capability. CDO is can now be used within Python and in combination with YAC-output can be directly used within the runtime of simulations like ICON. Further, PyCdo eliminates the need for in between file I/O within workflows using PyCdo. Where CDO is required to write to and from disc between calls, PyCdo is capable of handling these transfers in-memory. Additionally PyCdo provides netCDF, Xarray and Numpy support for its I/O which enables most commonly used scientific Python libraries to be used with PyCdo. In our work we have shown the structure of CDO and could show that extending CDO with PyCdo could be achieved without invasive changes to CDI and CDO. With this we could fulfil our goal of not adding additional heavy loads on the maintainers. Additionally PyCdo is completely written in C++ and has lower lines of code than the older Python interface for CDO while adding new features like automatic documentation of the operators inside PyCdo or IDE-support. Through these quality of life features we did not only cut done on the maintains cost of PyCdo but also increased the discoverability of operators and modules. When it comes to the readiness of PyCdo we showed that while it performs adequate with smaller workloads it is not yet a competitor to CDO when the workloads reach the size of the current exascale era. We could provide hints to where the problems could be and our best theory is that Python in context of its garbage collector has problems with how we handle large memory blocks either directly in PyCdo or even in our tests scripts. In general we could show that PyCdo has the potential to be used in the intended context and that it would provide the benefits we expected. We could show this for smaller workflows and we see good indicators that the problems do not stem from a general design flaw within PyCdo.
Aside from the problems of PyCdo, our work, as a by-product, moved CDO away from being a command line tool and toward being a self contained library. With additional work on it, CDO could be turned into an in-situ library or a general tool-box of in simulation routines required algorithms and data processing options.

7.1. Future Work

With our work we paved the way for multiple possible paths for enhancing CDO and its workflows as well as the workflows of ICON. But before further enhancements can be done, the scaling issue we found needs to be researched thoroughly. For this we propose further experimentation with PyCDo, Python and CDO to determine the causes and possible solutions. If and only if these problems are solved further investment into PyCdo is feasible. Making PyCdo a useful tool for exascale simulations and following even larger simulations is the natural follow up of our work.

With the by us stated limitations that PyCdo has, there is enough work left for a full project focusing on improving PyCdo from its prototype state into a fully realized software. We see huge potential in the possibilities PyCdo offers for streamlining the workflows of data evaluation, pre- and post-processing as well as the removal of hard to read bash scripts.

Another required follow up work would be in regards to full simulation runs using PyCdo. In these experiments the actual benefit of the in-situ capabilities could be shown. For this we

propose the instrumentation of a production ICON run with PyCdo. The instrumentation would consist of setting up ICONs namelist to send relevant amounts of data via YAC-output to the IO-nodes where then PyCdo is used to process and write the data. Within the proposed work a benchmark suit would have to be developed that can handle the in different languages implemented processes. Additionally methods how to measure the CPU-use on a core by core basis would be required to show that PyCdo can use previously unused resources and that the simulation itself does not need to wait for the post processing to done. Should the post processing be to slow the simulation could stall and thus make the benefits from the direct processing obsolete as it would replace one bottleneck with another. Depending on the work required for the just proposed topic it could be necessary to split the creation of the benchmarking system and the benchmarking of PyCdo into two distinct works.

Assuming these issues can be fixed the by us started transition of CDO into a fully fledged library would bring substantial benefits for general data processing as well as in-situ processing. The full librification of CDO would create a tool that provides a multitude of tried and tested, efficient, and multithreaded routines that can be directly used by simulations. Not as a post processing tool but as a literal library of routines. Having a central repository for often required routines would reduce the amount of code, maintenance and thus would increase the reliability and reusability. Thus we could see dedicated work on the advancement of CDO in this regard.

Lastly we generally see huge potential in CDO itself. Already a widely used tool for climate science we see the creation of Physics Data Operators (PDO), Chemistry Data Operators (ChDO), or any discipline that requires large amounts of simulation or experiment data to be processed, as a possibility to give the research community a powerful tool based on the CDO core library. This would require multiple years of work researching the differences between the disciplines, the commonalities and the feasibility of such a tool to provide a substantial benefit. In addition we expect that each discipline will already have their own tools, so that the need for such a general tool must be researched first.

Books/Chapters

[22] Peter Braam. *The Lustre Storage Architecture*. Vol. abs/1903.01955. 2019. arXiv: 1903.01955. URL: http://arxiv.org/abs/1903.01955.

[28] E. Wes Bethel et al. *The SENSEI Generic In Situ Interface: Tool and Processing Portability at Scale*. Ed. by Hank Childs, Janine C. Bennett, and Christoph Garth. Cham: Springer International Publishing, 2022, pp. 281–306. ISBN: 978-3-030-81627-8.

[30] Matthew Larsen et al. *Ascent: A Flyweight In Situ Library for Exascale Simulations*. Ed. by Hank Childs, Janine C. Bennett, and Christoph Garth. Cham: Springer International Publishing, 2022, pp. 255–279. ISBN: 978-3-030-81627-8.

Articles

[1] W Marciszewski. "Polish Notation". In: *Dictionary of Logic as Applied in the Study of Language: Concepts/Methods/Theories* (1981), pp. 252–254.

[2] R. Rew and G. Davis. "NetCDF: an interface for scientific data access". In: *IEEE Computer Graphics and Applications* 10.4 (1990), pp. 76–82. DOI: 10.1109/38.56302.

[3] Brian Eaton et al. "NetCDF Climate and Forecast (CF) metadata conventions". In: *URL: http://cfconventions. org/Data/cf-conventions/cf-conventions-1.8/cf-conventions. pdf* (2003).

[4] Judith Segal. "Some Problems of Professional End User Developers". In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. 2007, pp. 111–118. DOI: 10.1109/VLHCC.2007.17.

[5] Saman Amarasinghe et al. "Exascale software study: Software challenges in extreme scale systems". In: *DARPA IPTO, Air Force Research Labs, Tech. Rep* (2009), pp. 1–153.

[6] T Kuhlen, R Pajarola, and K Zhou. "Parallel in situ coupling of simulation with a fully featured visualization system". In: *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization (EGPGV)*. Vol. 10. Eurographics Association Aire-la-Ville, Switzerland. 2011, pp. 101–109.

[7] Wes McKinney et al. "pandas: a foundational Python library for data analysis and statistics". In: *Python for high performance and scientific computing* 14.9 (2011), pp. 1–9.

[8] John Shalf, Sudip Dosanjh, and John Morrison. "Exascale Computing Technology Challenges". In: *High Performance Computing for Computational Science – VECPAR 2010*. Ed. by José M. Laginha M. Palma et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–25. ISBN: 978-3-642-19328-6.

[9] Yoshiaki Miyamoto et al. "Deep moist atmospheric convection in a subkilometer global simulation". In: *Geophysical Research Letters* 40.18 (2013), pp. 4922–4926.

[10] Sandro Fiore et al. "Ophidia: A full software stack for scientific data analytics". In: *2014 International Conference on High Performance Computing & Simulation (HPCS)*. 2014, pp. 343–350. DOI: 10.1109/HPCSim.2014.6903706.

[13] Donatello Elia et al. "An in-memory based framework for scientific data analytics". In: *Proceedings of the ACM International Conference on Computing Frontiers*. CF '16. Como, Italy: Association for Computing Machinery, 2016, pp. 424–429. ISBN: 9781450341288. DOI: 10.1145/2903150.2911719. URL: https://doi.org/10.1145/2903150.2911719.

[14] M. Hanke et al. “YAC 1.2.0: new aspects for coupling software in Earth system modelling”. In: *Geoscientific Model Development* 9.8 (2016), pp. 2755–2769. DOI: 10.5194/gmd-9-2755-2016. URL: https://gmd.copernicus.org/articles/9/2755/2016/.

[16] Alessandro D’Anca et al. “On the use of in-memory analytics workflows to compute eScience indicators from large climate datasets”. In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2017, pp. 1035–1043.

[17] S. Hoyer and J. Hamman. “xarray: N-D labeled arrays and datasets in Python”. In: *Journal of Open Research Software* 5.1 (2017). DOI: 10.5334/jors.148. URL: https://doi.org/10.5334/jors.148.

[19] Simon D. Smart, Tiago Quintino, and Baudouin Raoult. “A Scalable Object Store for Meteorological and Climate Data”. In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC ’17. Lugano, Switzerland: Association for Computing Machinery, 2017. ISBN: 9781450350624. DOI: 10.1145/3093172.3093238. URL: https://doi.org/10.1145/3093172.3093238.

[20] Arne Johanson and Wilhelm Hasselbring. “Software Engineering for Computational Science: Past, Present, Future”. In: *Computing in Science & Engineering* 20.2 (2018), pp. 90–109. DOI: 10.1109/MCSE.2018.021651343.

[21] Xiang-ke Liao et al. “Moving from exascale to zettascale computing: challenges and techniques”. In: *Frontiers of Information Technology & Electronic Engineering* 19 (2018), pp. 1236–1244.

[23] Francis Alexander et al. “Exascale applications: skin in the game”. In: *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190056.

[24] Shunye Wang et al. “Design and Implementation of an Online Python Teaching Case Library for the Training of Application-Oriented Talents”. In: *International Journal of Emerging Technologies in Learning (iJET)* 15.21 (Nov. 2020), pp. 217–230. ISSN: 1863-0383. URL: https://www.learntechlib.org/p/218362.

[25] Donatello Elia, Sandro Fiore, and Giovanni Aloisio. “Towards HPC and Big Data Analytics Convergence: Design and Experimental Evaluation of a HPDA Framework for eScience at Scale”. In: *IEEE Access* 9 (2021), pp. 73307–73326. DOI: 10.1109/ACCESS.2021.3079139.

[26] Anjus George et al. “Understanding Lustre Internals. Second Edition”. In: (Sept. 2021). DOI: 10.2172/1824954. URL: https://www.osti.gov/biblio/1824954.

[27] T. V. Pham et al. “ICON in Climate Limited-area Mode (ICON release version 2.6.1): a new regional climate model”. In: *Geoscientific Model Development* 14.2 (2021), pp. 985–1005. DOI: 10.5194/gmd-14-985-2021. URL: https://gmd.copernicus.org/articles/14/985/2021/.

[29] Matthieu Dorier et al. “Colza: Enabling Elastic In Situ Visualization for High-performance Computing Simulations”. In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2022, pp. 538–548. DOI: 10.1109/IPDPS53621.2022.00059.

[31] L. Linardakis et al. “Improving scalability of Earth system models through coarse-grained component concurrency – a case study with the ICON v2.6.5 modelling system”. In: *Geoscientific Model Development* 15.24 (2022), pp. 9157–9176. DOI: 10.5194/gmd-15-9157-2022. URL: https://gmd.copernicus.org/articles/15/9157/2022/.

[32] F Prill et al. “ICON tutorial”. In: *Working with the ICON Model. Available online: https://www. dwd. de/DE/leistungen/nwv_icon_tutorial/pdf_einzelbaende/icon_tutorial2020. pdf* (2022).

[33] Dilawar Singh and Steven S Andrews. "Python interfaces for the Smoldyn simulator". In: *Bioinformatics* 38.1 (2022), pp. 291–293.

[34] Yiwen Dong et al. "Bash in the Wild: Language Usage, Code Smells, and Bugs". In: *ACM Trans. Softw. Eng. Methodol.* 32.1 (Feb. 2023). ISSN: 1049-331X. DOI: 10.1145/3517193. URL: https://doi.org/10.1145/3517193.

[40] *Levante Description from the DKRZ website*. Accessed: 2024-09-27. URL: https://www.dkrz.de/en/projects-and-partners/projects/focus/levante-spotlight.

[41] *Levante Technical Description from the DKRZ website*. Accessed: 2024-09-27. URL: https://www.dkrz.de/en/systems/hpc/hlre-4-levante.

Software/Documentation/Data

[11] Andrew C Bauer, Berk Geveci, and Will Schroeder. *The ParaView Catalyst User's Guide v2. 0. Kitware*. 2015.

[12] Matthew Rocklin et al. "Dask: Parallel computation with blocked algorithms and task scheduling." In: *SciPy*. 2015, pp. 126–132.

[15] Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs. "The Lustre V6 reference manual". In: *Verimag, Grenoble, Dec* (2016).

[18] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. 2017.

[35] Uwe Schulzweida. *CDO User Guide*. Version 2.3.0. Oct. 2023. DOI: 10.5281/zenodo.10020800. URL: https://doi.org/10.5281/zenodo.10020800.

[36] *Boost.Python documentation*. Accessed: 2024-02-12. URL: https://www.boost.org/doc/libs/1_84_0/libs/python/doc/html/index.html.

[37] *CMake example for setup.py. From the pybind11 repository.* Accessed: 2024-10-14. URL: https://github.com/pybind/cmake_example/blob/master/setup.py.

[38] The pip developers. *pip 24.2*. Accessed: 2024-10-10. URL: https://pypi.org/project/pip/.

[44] *memory profiler*. Accessed: 2024-10-28.

[45] *netcdf-c Repository*. Accessed: 2024-10-14.

[47] *PyBind11-stubgen Repository*. Accessed: 2024-10-14.

[48] *Python Packaging User Guide - tool recommendations*. Accessed: 2024-10-14.

[49] *Python Stubs Documentation*. Accessed: 2024-10-14. URL: https://mypy.readthedocs.io/en/stable/stubs.html.

[50] Uwe Schulzweida. *Climate Data Interface*. Accessed: 2024-09-2. URL: https://gitlab.dkrz.de/mpim-sw/libcdi.

Images

[39] Hanke, M. and Redler, R. and Holfeld, T. and Yastremsky, M. *YAC Logo*. [Online; accessed August, 2024]. URL: https://gitlab.dkrz.de/dkrz-sw/yac/-/raw/master/doc/Figures/yak_small.jpg?ref_type=heads.

[42] *Lustre Logo*. [Online; accessed August, 2024]. URL: https://www.lustre.org/wp-content/uploads/lustre-logo-RGB-1024x216.png.

[43] Max-Planck-Institute for Meteorology. *Icon Logo*. [Online; accessed August, 2024]. URL: https://mpimet.mpg.de/fileadmin/_processed_/1/f/csm_Icon_logo3_eed7626ac0.jpg.

[46] *Ophidia Logo*. [Online; accessed October, 2024]. URL: https://ophidia.cmcc.it/wp-content/themes/ophidia/images/oph_logo_symbol_big.png.

[51] xarray Developers. *Xarray Example Data Layout*. [Online; accessed September, 2024]. URL: https://docs.xarray.dev/en/v0.7.2/_images/dataset-diagram.png.

[52] xarray Developers. *Xarray Logo*. [Online; accessed September, 2024]. URL: https://docs.xarray.dev/en/v2024.09.0/index.html#.

A. Software Environment used for Benchmarking

For reproducibility we list the environment we used to compile, test and benchmark CDO and PyCdo.

```
Climate Data Operators version 2.4.0 (https://mpimet.mpg.de/cdo)
Features: 501GB 256threads c++20 pthreads sse2
Libraries: yac/3.1.0 NetCDF
CDI data types: SizeType=size_t
CDI file types: srv ext ieg grb1 nc1 nc5
     CDI library version : 2.4.0
 cgribex library version : 2.2.0
  NetCDF library version : 4.9.2 of Aug  8 2024 12:50:27 $
    exse library version : 1.5.0
    FILE library version : 1.9.1

CDO:
Branch name: python_interface
Commit hash: 1fcd45a63a80c8b3d4e95558ab470ab894d455f0
CDI:
Branch name: m300433/netcdf_memory_read_write,
COMMIT hash: 984143a0f70d08945a4ac0288d56fb84cf80c7f5

- Compiler: gcc (Spack GCC) 13.3.0
- Python Version: Python 3.10.10
- cmake version 3.30.20240808-gc0cd10c
- pybind11 version: py-pybind11-2.10.1-77opk
```

Figure 1.: CDO version and sub-library version used for Benchmarking

Python Module	version	Python Module	version
asciitree	0.3.3	Packaging	24.1
cdo	1.6.1	Pandas	2.2.2
certifi	2024.7.4	Pillow	10.4.0
cftime	1.6.4	Platformdirs	4.3.2
charset-normalizer	3.3.2	Pooch	1.8.2
contourpy	1.2.1	Pybind11	2.13.1
cycler	0.12.1	PyCdo	0.1.0
Cython	3.0.11	Pyparsing	3.1.2
fasteners	0.19	Python-dateutil	2.9.0.post0
fonttools	4.53.1	Pytz	2024.1
fsspec	2024.9.0	PyYAML	6.0.2
h5netcdf	1.3.0	Requests	2.32.3
h5py	3.11.0	Scipy	1.14.1
idna	3.8	Six	1.16.0
isodate	0.6.1	Typing	3.7.4.3
kiwisolver	1.4.5	Typing_extensions	4.12.2
matplotlib	3.9.1.post1	Tzdata	2024.1
mpi4py	4.0.0	Urllib3	2.2.2
netCDF4	1.7.1.post1	Xarray	2024.7.0
numcodecs	0.13.0	Zarr	2.18.3

Table 1.: Python environment used for Benchmarking PyCdo.

```
-- linux-rhel8-zen3 / gcc@11.2.0 ---------------------------------
bzip2@1.0.8                libaec@1.0.5    numactl@2.0.14
c-blosc@1.21.5             libiconv@1.17   openssl@1.1.1g
cuda@12.2.0                libxml2@2.9.7   pkg-config@1.4.2
diffutils@3.9              lz4@1.9.4       snappy@1.1.8
gmake@4.4.1                m4@1.4.18       zlib@1.2.11
hdf5@1.12.1                ncurses@6.2     zstd@1.5.2
intel-oneapi-mpi@2021.5.0  netcdf-c@4.9.2

-- linux-rhel8-zen3 / gcc@13.3.0 ---------------------------------
autoconf@2.69                         mpc@1.3.1
autoconf-archive@2023.02.20           mpfr@4.2.0
automake@1.16.5                       ncurses@6.4
berkeley-db@18.1.40                   ncurses@6.4
binutils@2.30                         ninja@1.11.1
bzip2@1.0.8                           openssl@1.1.1g
ca-certificates-mozilla@2023-01-10    perl@5.36.0
cmake@master                          perl@5.36.0
diffutils@3.9                         pigz@2.7
expat@2.5.0                           pigz@2.7
gawk@5.2.1                            pkg-config@1.4.2
gcc@13.3.0                            pkgconf@1.9.5
gdbm@1.23                             py-pybind11@master
gdbm@1.23                             py-wheel@0.37.1
gettext@0.21.1                        python@3.10.10
gettext@0.21.1                        re2c@2.2
gmake@4.4.1                           readline@8.2
gmp@6.2.1                             readline@8.2
libbsd@0.11.7                         sqlite@3.40.1
libffi@3.4.4                          tar@1.34
libiconv@1.17                         tar@1.34
libmd@1.0.4                           texinfo@7.0.3
libsigsegv@2.14                       util-linux-uuid@2.38.1
libtool@2.4.6                         xz@5.4.1
libxcrypt@4.4.33                      zlib@1.2.11
libxml2@2.9.7                         zlib@1.2.13
m4@1.4.18                             zstd@1.5.5
==> 74 loaded packages
```

Listing .1: Compile environment for PyCdo on levante showing all loaded modules via spack

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

4. 11. 2024 — Ort, Datum

Oliver Hüdmann — Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

4. 11. 2024 — Ort, Datum

Oliver Hüdmann — Unterschrift