Monitoring Numerical Climate Simulations

A Tool for the EC-Earth Climate Model

Wissenschaftliche Arbeit zur Erlangung des Grades
B.Sc. Ingenieurwissenschaften
an der Studienfakultät Munich School of Engineering
der Technischen Universität München.

Geprüft von	Univ.-Prof. Dr. Hans-Joachim Bungartz
	Lehrstuhl für Wissenschaftliches Rechnen, TUM
Betreut von	Dr. Uwe Fladrich
	Rossby Centre, Sveriges meteorologiska och hydrologiska institut
Eingereicht von	Valentina Schüller
	Hohenzollernstraße 61
	80796 München
	+49 176 221 464 19
Eingereicht am	München, den 10.09.2020

Abstract

Climate model experiments are time consuming numerical simulations. Real-time monitoring of experiments allows to spot problems in the model performance early on. Conspicuous results or changes in the computational performance can be detected at runtime. Users can then interrupt the experiment, thus saving computational resources. While in-depth and model-independent analysis tools exist for finalized and post-processed output, monitoring is model-specific and uses raw output data. With changing configurations and experiment setups, monitoring tools must be extendable and should not be limited to individual model components. For the latest version of the European community Earth system model EC-Earth, a Python based monitoring application has been developed. The tool tracks the physical and computational performance of ongoing simulations based on established metrics and diagnostics in climate science. This thesis presents design considerations and decisions, as well as the status quo of its feature set. Although the concrete implementation is optimized for EC-Earth 4, the software design and choice of monitoring diagnostics is of broader relevance. The monitoring tool's capabilities are shown and discussed based on an exemplary and realistic simulation.

Kurzzusammenfassung

Numerische Simulationen zur Klimavorhersage sind zeit- und rechenintensiv. Komplikationen während eines Experiments können durch Monitoring frühzeitig erkannt werden. Rückgänge der Rechengeschwindigkeit oder unrealistische Ergebnisse werden so in Echtzeit detektierbar. Modellnutzer*innen können das Experiment bei Problemen unterbrechen und dadurch sparsamer mit Rechenressourcen umgehen. Während es modellübergreifende Anwendungen zur tiefgreifenden Analyse von nachbearbeiteten Outputdaten gibt, ist Monitoring modellspezifisch und arbeitet mit unverarbeitetem Output. Da sich die benötigten Konfigurationen und Experimenteinstellungen immer wieder ändern, müssen Monitoring-Applikationen erweiterbar sein. Darüber hinaus sollten sie sich nicht auf einzelne Komponenten spezialisieren. Für die neueste Version des europaweit entwickelten Erdsystemmodells EC-Earth wurde eine Python-basierte Monitoring-Anwendung entwickelt. Sie überwacht die physikalische und rechnerische Leistung laufender Experimente, basierend auf in den Klimawissenschaften etablierten Metriken und Diagnostiken. Diese Bachelorarbeit stellt Designüberlegungen und -entscheidungen für das Tool vor, genauso wie die momentan implementierten Funktionen. Die hier entwickelte Implementierung ist auf EC-Earth 4 ausgerichtet, doch das Software-Design und die Auswahl der Monitoring-Diagnostiken sind nicht auf dieses Klimamodell beschränkt. Anhand eines exemplarischen und realistischen Experiments wird das Potenzial der neuen Software gezeigt und diskutiert.

Contents

1. Introduction

Numerical climate simulations are one important application of high-performance computing (HPC). Large code components run in a coupled and parallel manner for a long time: days, weeks, and even months, depending on the experiment. These simulations have a lot of output that gets analyzed in-depth and compared with other climate models as well as observational data. Both of these parts–simulation and analysis–are time- and labor-intensive tasks. It is thus crucial to check the status and spot problems of a simulation early on. Mistakes in the experiment setup, unexpected outcomes of initial data or machine problems can happen every once in a while. But especially when tuning the climate model or testing a new setup, it is helpful to see general effects of changed input in real time.

Getting insight into the current model state is the goal of monitoring numerical climate simulations. This implies information about the experiment setup and progress, as well as model performance. But how does one measure the performance of a climate model? From an HPC perspective, this is usually understood as a measure of speed and efficiency in a computational sense–but this is rarely meant in climate science. Here, a distinction can be made between physical and computational performance. The physical performance of a model quantifies how well it is able to represent the actual climate on Earth. Characteristic measures for this will be explained when discussing the implementation in chapter 4. Computational performance of climate simulations is closer to the classical HPC interpretation, but not the same. Climate models are highly complex applications. The maximum sustained flops and percent of peak of a simulation are not enough to compare climate models or different experiments. Besides being insufficient, they are typically not even quantities of interest for climate model developers. Instead, combinations of experiment and machine information are often used as indicators of computational performance. Common examples are the simulated years per day (SYPD) or the core hours per simulated year (CHSY). (Balaji et al., 2017) In this thesis, such quantities of interest and relevance are referred to as *diagnostics*, no matter if they are indicators for the computational or physical performance of a simulation.

Since climate models differ in functionality, structure, and complexity, there is no single tool that can monitor multiple different models. This is especially clear for the computational performance. Depending on the runtime environment of a model, a tool would need to get the necessary information in a multitude of ways. But physical performance is not much different: While published results usually adhere to international standards, the immediate model output needs to be post-processed before this is the case. There exist model-independent analysis tools, e.g., the Earth System Model Evaluation Tool (ESMValTool, Eyring, Righi, et al., 2016) and the Program for Climate Model Diagnosis and Intercomparison (PCMDI) metrics (Gleckler et al., 2016). Their relevance is illustrated in Eyring, Bony, et al., 2016. These tools contain evaluation procedures of varying complexity to compare multiple model runs, models, and observed data. Still, they are inapplicable to get automated real-time information about a single model run. To monitor simulations, tools have to be individually developed for each climate model. Nevertheless, the

selection of monitoring diagnostics is not model-specific–and the design considerations behind such a tool are generic and thus applicable not only to climate models.

For the new version of the EC-Earth climate model, EC-Earth 4, a monitoring tool has been developed. The programming task for this was defined as follows: The tool shall monitor computational and physical performance diagnostics of the currently ongoing simulation in real time. It is to be built using the same framework as the new runtime environment of this climate model. In this context and thesis, monitoring implies: no comparison with observational data and no comparison with past model runs. It is thus clearly distinguishable from analysis. EC-Earth output can be post-processed and then analyzed with ESMValTool, there is no need to replace this workflow. The new EC-Earth monitoring tool is not limited to one component of the model (e.g., the atmosphere or ocean). Using and developing it is characterized by modularity. Extending it is possible by writing a few lines of Python code, separated from the existing parts.

This thesis gives an overview of the concepts and structure of the monitoring tool. Furthermore, it shows and discusses the current functionality, future capabilities, and limitations of the tool. The remainder of it is structured as follows: In chapter 2, a short description of EC-Earth 4, its components, and relevance is given. Chapter 3 describes the design principles and architecture of the monitoring tool. Chapter 4 contains an overview of the implemented diagnostics, chapter 5 presents examples of monitored EC-Earth 4 runs. The thesis concludes in chapter 6 with a summary, brief discussion, and outlook.

2. The EC-Earth Climate Model

EC-Earth is a climate model developed by a Europe-wide consortium of climate research groups. EC-Earth 4 is the newest version of this model and currently in development. The monitoring tool addressed in this thesis has been designed for use with EC-Earth. For this reason, a short introduction to the concepts, components, and relevance of the model is given in this chapter.

The Earth's climate is shaped by interconnected physical processes on different domains and varying spatial and temporal scales. For example, atmospheric winds drive oceanic circulation, evaporation leads to cloud coverage, changes in atmospheric chemistry influence oceanic acidity. In climate modeling, the processes are usually separated into software components that represent the physical domains. They are then coupled to provide boundary conditions for each other. EC-Earth 4 is not special in this regard. Currently, the model contains the following components:

NEMO is a model for ocean (thermo)dynamics that contains components for sea ice modeling (NEMO-SI^3), and ocean biogeochemistry (NEMO-TOP, making use of the PISCES model). NEMO-SI^3 is the successor of the Louvain-La-Neuve sea ice model LIM3 (Rousset et al., 2015) and will be referred to as SI^3 for the remainder of this thesis. In-depth explanations of these models can be found in the NEMO reference manual by Gurvan et al., 2019. Key points from it are given in the following. The primitive equations of the ocean model are the Navier-Stokes equations–simplified with various hypotheses and approximations–, as well as a nonlinear equation of state. The model uses centered, second-order finite difference approximations on a curvilinear grid for the numerical solution. It is a tripolar ORCA grid, which means it "has no singularity point inside the computational domain since two north mesh poles are introduced and placed on lands" (Gurvan et al., 2019). This removes numerical difficulties during computation. However, it has to be considered when visualizing NEMO/SI^3 output, as will be explained in section 4.4. The NEMO grid is staggered, as is often found in fluid dynamics solvers (cf. Frisch, 2014). This means that scalar values are stored in the center of the grid cell and vector-valued quantities are stored at their respective cell boundaries. When writing output, NEMO treats the center and boundary points as four different, shifted grids: the T grid for scalars, and the U, V, W grids for vectors. The ocean model writes one NetCDF file per grid in the desired frequency. SI^3 uses the scalar grid for its output which is written to separate files. A typical choice for the output frequency of an experiment is monthly output since the ocean is a slowly responding system with high seasonal variability but few large scale diurnal processes. Nevertheless, other frequencies may be chosen instead or in addition. The cell dimensions for NEMO and SI^3 output are stored in a separate NetCDF file containing the domain configuration.

OpenIFS is the physical model for the atmosphere in EC-Earth 4. It is developed by the European Centre for Medium-Range Weather Forecasts (ECMWF). OpenIFS is a portable version of the ECMWF's Integrated Forecasting System (IFS) which has been used in past versions of EC-Earth (cf. Hazeleger et al., 2010). The relevant documentation for the model can be found at ECMWF, 2019a, ECMWF, 2019b, ECMWF, 2019c. Again, key points from it are explained in the following:

OpenIFS "uses a spectral transform method to solve numerically the equations governing the spatial and temporal evolution of the atmosphere." (ECMWF, 2020) While a lot of the computation happens in spectral space, the physical parametrizations and advection are calculated in the physical, so-called grid-point space. The horizontal grid is a reduced Gaussian grid, more information about it can be found in the paper by Malardel et al., 2016. OpenIFS output is separated into spectral and grid-point space files. The developed monitoring tool only processes grid-point space data, where the relevant variables for monitoring are stored. Since OpenIFS uses a different grid than NEMO, its output has to be treated very differently than ocean and sea ice data. This is amplified by the fact that OpenIFS output is written to GRIB, not NetCDF, files at the time of writing this thesis. The atmosphere has a much lower heat capacity than the ocean, atmospheric processes have strong diurnal variability. Thus, typical OpenIFS output is of much higher frequency (e.g., 6 hours) than NEMO/SI^3, although this output is stored in one GRIB file per month.

These components are linked to each other by the OASIS coupler (Valcke, 2013). EC-Earth also contains the XML I/O server (XIOS), a library to manage input and output of climate models efficiently.

A climate model can be classified as a general circulation model (GCM) or an Earth system model (ESM). While a GCM contains components for the (thermo)dynamics of atmosphere and ocean only, an ESM is not limited to this. Currently, EC-Earth 4 is configured to be a general circulation model. Components for vegetation, air chemistry, ocean chemistry and more will be activated in the future, making EC-Earth 4 an Earth system model.

For a numerical climate simulation, all of these components have to be compiled alongside each other. During runtime, the computation input and results get shared between the individual components. A climate model experiment consists of multiple self-contained parts: Instead of simulating the whole time domain of an experiment at once, the model stops and restarts with a predetermined frequency. For example, a 50 year-long simulation can be separated into 50, consecutively executed, so-called legs. In this case one leg corresponds to one simulated year of the experiment. The components write the simulation output to a specified directory, the run directory. Here, the output files are grouped into leg folders. At the end of each leg the computation stops and EC-Earth restarts with the current internal state. This repeats until the end of the simulation is reached. The length of the legs can be adjusted by the user. An exemplary one-year leg folder of EC-Earth 4 in its current state would contain about 60 NetCDF files written by NEMO and SI^3 and 36 GRIB files written by OpenIFS. This gives an idea how data intensive climate simulations are, although the numbers vary based on the chosen setup.

EC-Earth 3 has been participating in the Climate Model Intercomparison Project (CMIP) phases 5 and 6, a global research effort to analyze results from various Earth system models. CMIP data is used extensively in the assessment and special reports by the International Panel on Climate Change (IPCC). EC-Earth 4, the latest version, is still under development but will fully replace EC-Earth 3 in the coming years. There exists a monitoring tool for the ocean in EC-Earth 3 called Barakuda (Brodeau, 2017). Barakuda is built using shell scripts and depends on external command

line tools, which makes it hard to maintain and difficult to extend. It only monitors NEMO output. The monitoring tool developed in this thesis is an extendable, component-independent Python package based on the new runtime management tool for EC-Earth 4.

3. Design Principles and Architecture

To create a monitoring tool is mostly an application of modern software design and development. Thus, an overview and discussion of the general considerations is given in this chapter from a relatively abstract perspective. It starts with a definition of the core design principles in section 3.1. Concepts and details of the existing infrastructure are laid out in section 3.2. At the end of this chapter, the structure of the monitoring tool is explained in detail.

3.1. Design Principles

The developed monitoring tool sticks to a few key principles that affect both functionality and implementation. It is designed to be helpful, robust, user-friendly, modular, and extendable:

- helpful: Monitoring a simulation should be beneficial to the user. This is a central requirement and important for the selection of diagnostics to implement.
- robust: An EC-Earth run must not be strongly affected by the monitoring tool in a negative way. In essence, this has two meanings: First of all, the tool should not slow down the model significantly. Secondly, the EC-Earth run should not interrupt because of errors in the monitoring tool. They should be logged and otherwise ignored. This principle is the main reason for the decision to keep the interaction between model and monitoring tool as small as possible.
- user-friendly: As Craig et al., 2005 mention, "like many other scientific applications, 'using' a climate model implies editing source code in many cases". EC-Earth users modify shell scripts or change parameters in Fortran namelist files to set up an experiment. User-friendliness here does therefore not imply the need for a simplistic GUI. Still, the user-software interaction should be consistent with the rest of the EC-Earth 4 runtime environment. In general, setting up a monitored experiment should not be more complex than configuring a regular simulation.
- modular: This allows for the tool to be maintainable in the long run. It should consist of independent building blocks that can be added, removed or changed as necessary.
- extendable: EC-Earth 4 is still under development. As the climate model will grow in the next months and years, the needs and possibilities of monitoring its simulations will change as well. Reacting to this must be possible with the monitoring tool.

It is the intent of this design to enable extension of the monitoring tool without overhauling the structure. The main purpose of the work behind this thesis was to develop a stable foundation which can be built upon in the future. The tool already enables the creation of meaningful diagnostics, and these will be laid out in chapter 3. But they are meant to be the groundwork for more, not the end of the line. An important consequence of this is that the monitoring tool was developed as an open-source, object-oriented Python code with automated tests and high test coverage. The

package furthermore contains a user documentation and developing guidelines (see the final chapter Code Availability and Resources for more information).

In the course of developing the monitoring tool, a lot of essential decisions had to be made. In general, these choices should be transparent and comprehensible to future users and developers alike. Whenever possible, the tool falls back on well-established climate modeling standards. This final design principle works in favor of all others and concludes this section.

3.2. Existing Infrastructure

The monitoring tool builds upon existing components: It is based on the same Python package as the new compilation and runtime environment of EC-Earth 4. Furthermore, it is controlled by this runtime environment and uses the model output as input information. Since the EC-Earth output structure has been laid out in chapter 2, this section focuses on the Python framework behind both the runtime environment and monitoring tool, ScriptEngine.

At its core, "ScriptEngine is a lightweight and extensible framework for executing scripts written in YAML." (Fladrich, 2020) [1] A user creates a YAML script and passes it to the ScriptEngine command line tool *se*. An exemplary, parsable script is given in Listing 3.1. When executed, the string *Hello, Earth!* will be shown on screen. YAML scripts for ScriptEngine always follow a similar structure: They contain a list of actions to complete, with input parameters for each element of the list.[2] For EC-Earth 4, the scripts contain procedures to store experiment parameters, create directory structures, copy files, or add jobs to the queues of computing nodes.

```
- context:
    planet: Earth
- echo:
    msg: "Hello, {{planet}}!"
```

Listing 3.1: *Exemplary YAML script for ScriptEngine.*

The words *context* and *echo* in this script are representations of an essential building block of ScriptEngine: the task. In general, a task is an entity which completes a desired action. It is both a concept and, in ScriptEngine, a Python class. To differentiate between these two facets, the Python class Task is capitalized in this thesis, whereas the concept is not. The central element of the Task class is its *run()* method. When an instantiated Task's *run()* method is called, the action gets executed. The YAML representation of the task concept consists of a name and required input parameters. *context* and *echo* are keywords mapped to the classes Context and Echo–both inherit from the base class Task.[3] When a user calls the ScriptEngine command line tool *se*, it parses the YAML file and instantiates Context and Echo with the input parameters *planet* and *msg*. Then, *se* calls their *run()* method.

[1] More information on this package, as well as the source code, can be found at https://github.com/uwefladrich/scriptengine.

[2] These can also be combined with loops and conditional clauses. Since they are not essential for the structure of the monitoring tool, these more advanced capabilities of ScriptEngine have been left out.

[3] The name of the YAML representation is very similar to the name of the Python class, although it does not have to be.

Each implemented ScriptEngine task inherits from the generic Task class. The core package contains some basic tasks (find a file, change directory, execute a command in the shell, etc.). Since ScriptEngine is implemented as a Python namespace package (Smith, 2012), other developers can add onto the existing tasks with new task sets. This is the basis for the extensions described in this thesis.

The final essential piece of ScriptEngine is the context. The context is a construct which contains all the information to execute tasks. This includes input parameters but is not limited to it. It is implemented as a single, although possibly nested, Python dictionary. The context is filled both with user-specified data from the scripts, using the context task, and ScriptEngine metadata during execution. In Listing 3.1, the key *planet* is stored with the value *"Earth"*. Other metadata stored in the context includes the working directory the command line tool was started in, *_se_ocwd*. All information in the context can be accessed by tasks. This is how the echo task determines the value of *planet*.

The monitoring tool is based on ScriptEngine and its structure. Monitoring an EC-Earth run thus consists of parsing one or multiple scripts using the *se* command line tool. These make use of newly developed tasks, information from the EC-Earth 4 output, and information from the context.

3.3. Concept and Architecture

3.3.1. Diagnostics

At the beginning of this thesis, diagnostics were defined as quantities of interest and relevance that inform about the performance of a climate model. In the monitoring case, these refer to the current model run. The monitoring tool's purpose is to compute diagnostics and visualize them in a fitting way. An important step developing the monitoring tool was thus to further define diagnostics: What type of objects can such quantities be?

In literature on climate model analysis and performance, the term *performance diagnostic* is established. It is often distinguished from another term, metrics: Whereas performance metrics for climate models are scalars (cf. Gleckler et al., 2008, Ma et al., 2013, Reichler and Kim, 2008), diagnostics are not limited to this type of data. Usually, metrics are computed by an aggregation of model output and observational data. Diagnostics, however, are not per se linked to external data–and they are not only scalars. In Gleckler et al., 2008, examples for diagnostic types are "e.g. maps, time series, power spectra". At another point they speak of "maps, time series, distributions, etc." The monitoring tool for now supports and distinguishes between four general diagnostic objects: scalars, time series, and maps that are either time-dependent or not. This categorization results from considerations regarding the dimension properties of desired quantities to monitor. An overview of the diagnostic types and their differences can be seen in Table 1, including some examples that motivate this structure. For the spatio-temporal data in time-dependent maps, the name "temporal maps" was chosen. It is used by the geographic information system ArcGIS for the same type of data (ESRI, 2012). In the future, the list of diagnostic types may be extended if it proves to be necessary.

Diagnostic Type	Dimensionality		Example
	space	time	
scalar	0	0	experiment description current size of output directory
time series	0	1	simulated years per day over time arctic sea ice volume over time
map	2	0	simulation average of the sea surface temperature
temporal map	2	1	arctic sea ice distribution over time

Table 1 Dimensionality and examples for the different types of diagnostics supported by the monitoring tool.

3.3.2. Architecture

The software architecture consists of interfaces with external components as well as the internal structure. Both of these will be outlined in this section, where the main focus lies on the internal structure.

As explained in section 3.1, the monitoring tool shall not interfere with the EC-Earth model run. For this reason, it is conceptually decoupled from every other EC-Earth component. To create diagnostics, the tool needs model output and meta information about the model run. Thus, the interface between tool and model is the output data (i.e., the run directory) and the runtime environment of the current simulation. No other input source is provided since no external data is necessary to monitor the simulation. The flow of information is one-way: The monitoring results are influenced by the model's computation but the monitoring tool does not affect the EC-Earth components in any way. This supports the design principle of robustness significantly and is in line with the programming task at hand. The monitoring tool presents the created diagnostics in a suitable form. Such a presentation format could be, for example, a collection of saved diagnostics or one document, source files for a web page, etc. They are then placed somewhere, at a presentation outlet–this is the output of the monitoring tool. At the end of chapter 4, the already supported presentation outlets will be discussed in more detail. An overview of the described I/O structure can be seen in Figure 1.

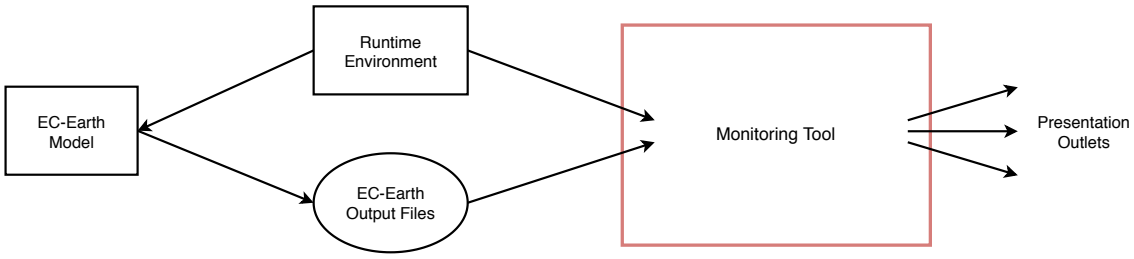


Figure 1 A schema visualizing the input and output of the monitoring tool. The flow of information is denoted by arrows. Directories and files are displayed as ellipses, software components as rectangles. Since presentation outlets can be another piece of software or file(s), no shape has been assigned.

Now that the interaction with other components is clear, the internal structure of the monitoring tool can be discussed. As it is based on the ScriptEngine framework, tasks are the main idea it builds upon. The monitoring tool has a defined workflow: At the end of each leg, using EC-Earth output, it creates relevant diagnostics. These diagnostics are then visualized in an expressive manner. Both the selection of diagnostics as well as the desired visualization might vary between experiments and are thus configurable by the user. The software consists of two different task types: **Processing tasks** process input from the model output and the runtime environment. With this, they create diagnostics and save them in files. **Presentation tasks** read these saved diagnostics and visualize them. Then, they present all diagnostics at a presentation outlet. This separation of processing and presentation allows for a very modular implementation: If the interface is well-defined, diagnostics can be added and removed without changing anything about the presentation task. In the same manner, the way diagnostics are presented can be modified without modifying the processing task itself. With this approach, a user can easily customize their monitoring setup. For example, one could decide to create diagnostics and save them on disk, without presenting them. Instead, they can use their own preferred tools but still profit from the monitoring tool. Similarly, diagnostics can be presented in multiple ways if desired. A schema of this separation can be seen in Figure 2.

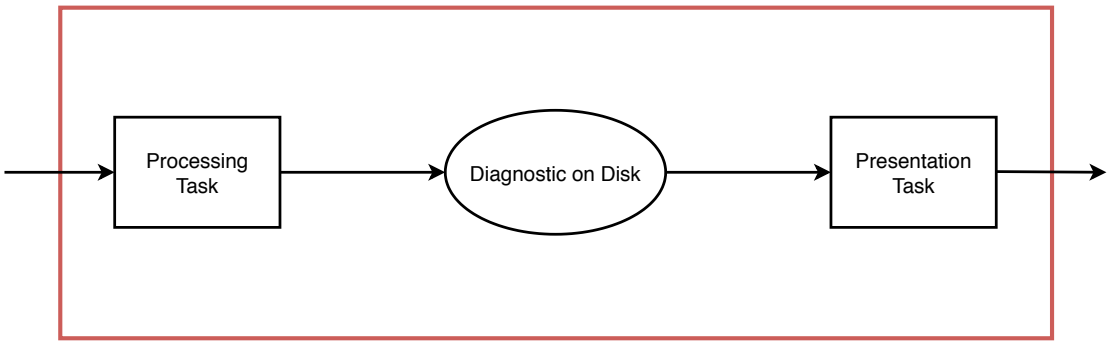


Figure 2 A schema visualizing the separation of monitoring tasks. Arrows visualize the flow of information. Directories and files are displayed as ellipses, software components as rectangles. Refer to Figure 1 for the outer structure.

3.3.3. The Structure of Processing Tasks

Processing tasks create diagnostics based on information from an EC-Earth component. These can be OpenIFS or NEMO/SI^3 output files or parameters stored in the runtime environment. Refer to chapter 2 for more information on these components. To get from this input to a final diagnostic, the task might apply some operations on spatial or temporal domains, e.g., an annual mean or a global sum. The computation results in a quantity with dimensional properties that can be linked to one of the diagnostic types in Table 1. Typically, the final diagnostic can be referred to by a variable name: Either the processing task only deals with one variable or the variable results from a user's selection.

This structure can be formulated into a naming scheme for both processing tasks (the classes, their containing modules, and YAML representations) and the file names of diagnostics on disk:

$$\text{variable\_component\_}\{\text{domain\_op...}\}\text{\_diagnostictype}$$

variable refers to the name of the resulting quantity, e.g., simulated years per day (SYPD) or sea surface temperature (SST). The EC-Earth 4 component that provides the input data for a processing task/diagnostic is abbreviated in *component*. This can be equal to, e.g., *oifs* (OpenIFS), *nemo* or *rte* (runtime environment). *domain_op* pairs can be combined consecutively to describe the spatial and temporal operations. The name finishes with the diagnostic type (cf. section 3.3.1).

This naming template is used as follows: The Python class uses the name in CamelCase notation (capitalize to separate words). YAML representation, Python module, and diagnostic on disk use the snake_case notation (lowercase, words separated by underscores). The YAML representation of processing tasks also always starts with *ece.mon.*, to separate the monitoring tasks from ScriptEngine base tasks like *context* or *find*.

A simple example is the disk usage task: DiskusageRteScalar is the name of the processing task. Here, *diskusage* is the variable name, *rte* signifies that the diagnostic results from runtime environment information. The diagnostic type *scalar* is added at the end. The YAML representation is *ece.mon.diskusage_rte_scalar* and the file name of the diagnostic on disk is *diskusage_rte_scalar.yml*.

These names get longer with physical performance tasks like NemoGlobalMeanYearMeanTimeseries: Here, NEMO output files are processed and a global and annual mean is applied to the data, resulting in a time series diagnostic. The task is not limited to one variable, a user can select one in the YAML script (the YAML representation of the task is *ece.mon.nemo_global_mean_year_mean_timeseries*). If a user chooses, e.g., *tos* (the NEMO variable name for the sea surface temperature), the resulting diagnostic on disk can be called *tos_nemo_global_mean_year_mean_timeseries.nc*.

These processing tasks will be discussed in the next chapter in more detail. Categorizing and naming tasks this way makes their functionality visible to users. It also helps developers to choose names for new processing tasks.

3.3.4. Diagnostics on Disk

To achieve the structure in Figure 2, the interface between processing and presentation tasks has to be very concise. For this reason, it was important to define clear criteria how a diagnostic can be saved in a file. This representation of diagnostics is called **diagnostic on disk**. The file type, structure, and requirements for diagnostics on disk are fundamentally important for a working monitoring tool. In this section, their properties are therefore explained in detail.

Since the diagnostic on disk contains all the information for the presentation task, it must be self-describing. The file therefore needs to contain some metadata next to the actual value(s) of the quantity. The contents of a diagnostic on disk can be separated into three blocks:

- The actual data. Depending on the diagnostic type, this can be very simple or complex. For scalar diagnostics, this will be a single number or piece of text. For the other diagnostic types, this can be thought of as a multi-dimensional array. In the latter case, this will have to be

structured by coordinates and/or with labels.

- Metadata the user needs to interpret the quantity at hand. This includes the title or name of the diagnostic, as well as a description or comment how it was obtained.
- Metadata that is necessary for the presentation task. Multi-dimensional data arrays will need to be visualized in a very different way than a single number or some text. To account for these fundamental differences, information regarding the desired visualization needs to be saved alongside the data.

The file type of a diagnostic on disk should support this underlying structure. On the one hand, diagnostic files should be simple enough to contain scalars and their metadata. But besides that, there needs to be a way to save multi-dimensional data arrays in an expressive and compact format. Furthermore, they should be easy to work with in the Python context. In climate and weather prediction, NetCDF files are a well-established, expressive, and regularly updated format for storing multi-dimensional, labeled data. This file type was therefore chosen for time series, maps, and temporal maps. Since scalars are zero-dimensional in space and time, they are simpler to handle than the other types: No coordinate labeling or boundaries and few metadata has to be stored. For this reason, YAML files proved to be a suitable file representation.

In the Design Principles section, the advantage of standardized approaches for this tool was illustrated. This principle becomes very apparent in the structure of diagnostics on disk. In fields that depend on data analysis as much as climate science, unified data storage guidelines are nothing new. For the sharing and processing of NetCDF files, the climate and forecast (CF) conventions (Eaton et al., 2020) are a metadata standard in meteorology and climate science. They provide guidelines for structuring coordinates, naming variables, assigning units, and much more. Since EC-Earth output has been and will be used to participate in the Climate Model Intercomparison Project, this is another source for conventions to save diagnostics. CMIP 6 data has to be in a format conforming to the CMIP data request. This is a standard stricter and more limited than the CF conventions. The CMIP data request introduces controlled vocabulary for more variable metadata. Besides that, it prescribes the governing structure for CMIP 6 data: The CMIP data request expects exactly one variable per NetCDF file. This was adopted for the diagnostics on disk as well: A file written by a processing task contains one variable (i.e., data for one diagnostic).

The CF conventions and CMIP 6 data request are built around physical quantities. The controlled vocabulary for variables and units does therefore not cover computational performance diagnostics like the simulated years per day. Whenever possible, a diagnostic file fully conforms with CMIP 6 and CF requirements. But even when a diagnostic does not represent a physical quantity or if it is not saved as a NetCDF file, it meets as much of the requirements as possible.

All diagnostics on disk (no matter if YAML or NetCDF file) contain the following information: one variable/diagnostic per file, as prescribed by the CMIP Data Requirements; an attribute *diagnostic type*; the two attributes *title* and *comment*, as suggested by the CF conventions. For NetCDF diagnostics on disk, the attributes are stored as global attributes. In the YAML files, they are

stored as key-value pairs. *title* is mandatory and "a succinct description of what is in the dataset", *comment* is optional and contains "miscellaneous information about the data or methods used to produce it" (Eaton et al., 2020). The data in YAML files is stored under the key *value*. In NetCDF files, the data is stored in variables and coordinates.

All NetCDF diagnostics on disk (i.e., saved time series, maps, and temporal maps) contain some more information to make them compliant with the CF conventions. Processing tasks add an attribute *long_name* to a variable, as the conventions and the NetCDF user's guide (Unidata, 2020) recommend. This attribute is "a long descriptive name which may, for example, be used for labeling plots" (Eaton et al., 2020). A variable also contains the attribute *cell_methods*. Cell methods got introduced by the CF conventions and have the form *name: method*. They indicate which operations were applied to the contained data, e.g., summation or mean over one or multiple dimensions. The file also contains the global attribute *source*: As explained in the CF conventions section 2.6.2, this refers to the "method of production of the original data. If it was model-generated, source should name the model and its version, as specifically as could be useful." (Eaton et al., 2020) This is then equal to *EC-Earth 4*. Finally, all NetCDF diagnostics on disk have a global attribute *Conventions* with the value CF-1.8, to show the CF compliance of the dataset. For all implemented processing and presentation tasks, the Python package Iris (Met Office, 2010b) is used when working with NetCDF files, which ensures CF conformity.

If a diagnostic is a physical quantity that gets saved as a NetCDF file, it is even more standardized: First of all, the variable has another attribute, the *standard_name*. This is a string from the CF Standard Name Table to describe the variable's physical properties (Eaton et al., 2020). Furthermore, variable name, standard name, long name, and units adhere to the CMIP 6 data request where possible. Time series and temporal maps have a dimensional time coordinate. It must be monotonically increasing, as required by the CF conventions. Since time progresses forward in EC-Earth simulations, time cannot decrease during the monitoring.

Next to the diagnostic type, NetCDF diagnostics on disk contain a custom attribute to control their visualization: Maps and temporal maps on disk have the global attribute *map_type*. Depending on the EC-Earth component, the diagnostic, and the dimensions (latitude-longitude maps, latitude-pressure level maps,...) the desired visualization will differ significantly. For now, supported map types are *global ocean*, *global atmosphere*, and *polar ocean*. But the list can be easily extended in the future. This will be explained further in section 4.4.

To summarize: A set of ScriptEngine tasks is the basis for the monitoring tool. There are two sets of tasks for the creation and visualization of diagnostics. Between processing and presentation, diagnostics are saved as YAML or NetCDF files. This file is as close to established standards as possible. They affect the structure but also the naming of variables and units. The tool can be extended by creating new processing and presentation tasks. These must inherit from the ScriptEngine Task class and use the structure of diagnostics on disk defined here. An overview of the diagnostic on disk requirements was given. Diagnostics can be scalars, time series, maps, and temporal maps–but if necessary, this list can be extended in the future.

4. Implemented Monitoring Tasks

This chapter provides an overview of the implemented processing and presentation tasks, to demonstrate the capabilities of the monitoring tool in its current state. Motivations and usage examples are given for each task. For more complex tasks the assumptions, implemented procedures, and limitations are explained in detail.

The chapter is structured in the following manner: It starts with the implemented processing tasks (4.1-4.3) and ends with the presentation tasks (4.4). The processing tasks are separated not by their diagnostic type (cf. section 3.3.1) but the EC-Earth 4 component they are linked to. In section 4.1, an overview of computational performance tasks is given. These get their input from the EC-Earth runtime environment. Section 4.2 contains all tasks that process NEMO and SI^3 output, which results in monitoring diagnostics for oceanic and sea ice variables. OpenIFS processing tasks are summarized in section 4.3, they create relevant atmosphere diagnostics.

This structure allows to summarize assumptions about and challenges with the output generated by the EC-Earth components. However, it does not reflect how the processing tasks are categorized in the code. A more effective classification for this is the diagnostic type: As was explained in detail in the Diagnostics on Disk section, the structure of the saved netCDF and YAML files depends on the diagnostic type. The procedure for saving on disk is identical for all diagnostics of the same type.[1] For each of the four diagnostic types, a base class has been implemented: Scalar, Timeseries, Map, Temporalmap. All processing tasks inherit from their respective base class. There is a very important difference between the former and latter task classes: Map and Temporalmap are not visible to the user, i.e., they can not be instantiated using a YAML script. Scalar and Timeseries can be called directly from the YAML script. This is because there was a clear demand for a generalized, user-customizable scalar and time series diagnostic, but not for generalized (temporal) maps. Since this chapter illustrates the possibilities when *using* the monitoring tool, only the first two tasks will be covered in detail in this thesis, in section 4.1. Presentation tasks can vary a lot based on the presentation outlet. Thus, no generalized version of this task type has been implemented. An overview of this implementation structure is shown in Figure 3.

Motivation for the implemented diagnostics and presentation tasks were not only group discussions, but also existing tools with similar goals. These include the Earth System Model Evaluation Tool (Eyring, Righi, et al., 2016), the monitoring tool Barakuda (Brodeau, 2017) for the ocean in EC-Earth 3, and the post-processing tools for EC-Earth 3 (Le Sager et al., 2019).

4.1. Processing Tasks for Computational Performance

The processing tasks described in this section result in diagnostics which provide information about the general model progress and computational performance. Such diagnostics usually show

[1] Besides that, there are more properties and procedures that can be implemented on the level of the diagnostic type and not each individual processing task. This includes checking file extensions and more.

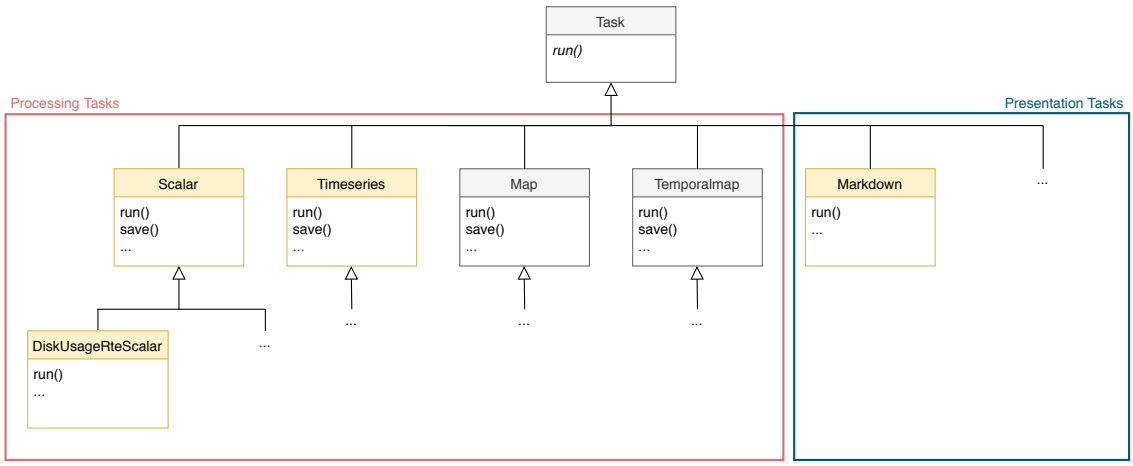


Figure 3 A class diagram showing the inheritance structure of the implemented monitoring tasks. While all implemented presentation tasks inherit from the ScriptEngine Task directly, the processing tasks contain a two-layered substructure. The first layer consists of generalized diagnostic type tasks. Tasks shaded in gray are implemented but not accessible to the user, while those in yellow have YAML representations.

the value of a scalar quantity at one moment or over the time of a simulation. The implemented processing tasks for computational performance are hence scalars or time series.

4.1.1. Scalar

This is the least complex processing task, and also the base class for all implemented scalar tasks. For an experiment, a user might want to display some general, non-dimensional information: The ID of the experiment (a four-digit code), a description why it was set up or when it was started. The Scalar processing task enables this: A user can write custom output to a YAML file which fulfills the diagnostic on disk requirements from section 3.3.4. The Scalar class is mapped to the YAML representation *ece.mon.scalar*:

```
- ece.mon.scalar:
    title: Some Scalar
    value: value
    comment: Optional description.
    dst: ./somescalar_scalar.yml
```

Scalar expects a title and data value for the diagnostic, as well as the path to a file with a valid YAML extension (.yml or .yaml). This is where the diagnostic will be saved. Optionally, a user can provide a description using the parameter *comment*. To save the experiment ID as a diagnostic on disk, the task can be called like this:

```
- context:
    exp_id: MON1
- ece.mon.scalar:
    title: Experiment ID
    value: "{{exp_id}}"
    dst: "./expid_scalar.yml"
```

The contents of *expid_scalar.yml* are shown below.

```
title: Experiment ID
value: MON1
diagnostic_type: scalar
```

All other tasks for scalar diagnostics inherit from Scalar and reuse its *save()* method (cf. Figure 3).

4.1.2. DiskusageRteScalar

The scalar diagnostic resulting from this processing task informs the user how large a specified folder is. Although this can be any directory, it is intended and particularly useful for the run directory of the current simulation. Research proposals in climate science usually contain storage estimates and requests for projects. The disk usage is thus a critical quantity for experiments. The processing task DiskusageRteScalar computes the size of the directory at the provided path and saves the diagnostic on disk, using the *save()* method of the Scalar task. The Python class is linked to *ece.mon.diskusage_rte_scalar* for YAML parsing. A usage example can be seen below:

```
- ece.mon.diskusage_rte_scalar:
    src: "{{run_dir}}"
    dst: ./diskusage_rte_scalar.yml
```

4.1.3. SimulatedyearsRteScalar

The simulated years scalar diagnostic is a measure for the simulation progress. Climate simulations in EC-Earth always have a start and end date, multiple years apart. A displayed number of simulated years indicates the current state of the experiment. The processing task is called SimulatedyearsRteScalar, it can be instantiated from the YAML script as *ece.mon.simulatedyears_rte_scalar*.

```
- ece.mon.simulatedyears_rte_scalar:
    start: "{{schedule.start}}"
    end: "{{schedule.leg.end}}"
    dst: ./simulatedyears_rte_scalar.yml
```

In the runtime environment of EC-Earth, the simulation start date is stored in the context, as well as the end date of the current leg. The *run()* method of SimulatedyearsRteScalar gets these two dates–they get parsed as Python *datetime.datetime* objects–and computes the difference in years. As this is again a scalar diagnostic, it gets saved in a YAML file.

4.1.4. Timeseries

This processing task allows a user to create a custom time series diagnostic, illustrating the progression of a scalar quantity over the duration of the current experiment. In its most simple form, it can be called like this:

```
- ece.mon.timeseries:
    title: Some Diagnostic
```

```
    data_value: "{{some_value}}"
    coord_value: "{{current_time}}"
    dst: ./somediagnostic_timeseries.nc
```

In this example, only the required parameters are given. To save the diagnostic on disk, the task requires a title, a new value for the time coordinate and the data array, and a destination for the diagnostic on disk. It either creates a new NetCDF file or appends new values to the existing one. The key assumption for a time series diagnostic is that the coordinate values are monotonically increasing (cf. 3.3.4). Besides that, a user can freely choose their input. The script and diagnostic can be customized further using the optional parameters:

```
- ece.mon.timeseries:
    title: An Interesting Title
    data_value: "{{some_value}}"
    coord_value: "{{current_time_in_seconds}}"
    comment: Diagnostic Description.
    coord_name: x-axis label
    coord_units: s
    data_name: y-axis label
    data_units: m
    dst: ./somediagnostic_timeseries.nc
```

When these are not given, the task assumes values: The comment does not get set, the names of the coordinate and data variable are set to "time" and the value of *title*, respectively. The units are set to "1" if a user does not provide any.

While the task is very generic, it allows for two specific diagnostics that are very relevant for the computational performance: Using information from the context, a user can create time series for the simulated years per day (SYPD) or the core hours per simulated year (CHSY). These diagnostics are common in climate science (Balaji et al., 2017) and do not require more specific task implementations. For example, an SYPD time series can be created like this:

```
- ece.mon.timeseries:
    title: Simulated Years per Day
    coord_value: "{{leg_num}}"
    coord_name: Year
    comment: SYPD development during this simulation.
    data_value: "{{((schedule.leg.end - schedule.leg.start)/
       script_elapsed_time/365)}}"
    dst: ./sypd_timeseries.nc
```

To ensure CF compliance of the diagnostic on disk, the task uses the Python package Iris (Met Office, 2010b) when saving the diagnostic. Since computational performance diagnostics are not covered by the CMIP data request, diagnostics on disk created by Timeseries do not adhere to

these standards. The algorithm for saving this generic time series on disk is reused by all other tasks with the same diagnostic type.

4.2. Processing Tasks for NEMO and SI^3

For these processing tasks, certain assumptions were made about the input data. If they are not met, the resulting diagnostics will either not be created or they might not show the expected results.

All processing tasks for ocean and sea ice assume that the input data are output files from NEMO or SI^3, i.e., NetCDF files on a global curvilinear grid. Currently, only 2D variables can be treated–this will be improved on in future releases. Furthermore, it is assumed that data for land cells is flagged as invalid. The task NemoMonthMeanTemporalmap in section 4.2.3 as well as the sea ice processing tasks (4.2.4 and 4.2.5) will fail if the output frequency is not monthly (e.g., daily or annual output). As explained in chapter 2, this is the typical frequency for NEMO output and thus not considered problematic. Finally, a leg length of one year is expected. Longer or shorter lengths will not lead to failure but some of the file descriptions might be inaccurate (e.g., the *comment* attribute might say "annual mean" despite being a half-year mean).

In all processing tasks, the Iris package (Met Office, 2010b) is used to open and modify the NetCDF files. Loading and saving NetCDF files with Iris ensures CF compliance. Variable names, metadata, and units are changed in the tasks to meet some of the CMIP data request standards.

4.2.1. NemoGlobalMeanYearMeanTimeseries

The sea surface is the interface between ocean and atmosphere. Climatic changes are thus reflected in sea surface variables such as the sea surface temperature (SST), sea surface salinity, sea surface height, etc. Temporal developments of these and other 2D quantities are very common ocean diagnostics.

By taking a global (i.e., spatial) and annual mean, this processing task creates time series diagnostics that emphasize large-scale trends on Earth. The task can produce such diagnostics for various 2D ocean quantities and inherits from the generic Timeseries task. In the future, it can be extended to support 3D variables, such as the oceanic heat content, as well.

The Python class is mapped to the YAML representation *ece.mon.nemo_global_mean_year_-mean_timeseries*. A usage example can be seen beneath this paragraph, where "tos" is the NEMO variable name for the SST.

```
- ece.mon.nemo_global_mean_year_mean_timeseries:
    src: "{{t_files}}"
    varname: tos
    domain: "{{rundir}}/domain_cfg.nc"
    grid: T
    dst: ./tos_nemo_global_mean_year_mean_timeseries.nc
```

The task extracts the variable *varname* from the list of NEMO output files in *src*. Using the domain configuration file in *domain*, it computes the area-weighted mean. The optional parameter *grid* can be equal to T, U, V, and W. It specifies which of the four NEMO grids should be used for the weighted mean. If no value is given for *grid*, the task assumes to use the T grid for scalar quantities. It also averages over the time span from the first to the last input file (e.g., one year if all files of a year-long leg are provided). Month lengths are taken into account for the temporal mean. The diagnostic gets saved at *dst*. The structure of the corresponding NetCDF file can be seen in Figure 4.

4.2.2. NemoAllMeanMap

When monitoring a simulation, it is helpful to see the spatial distribution of ocean surface variables and not only the global average. This shows if the model reproduces regional phenomena. The use of such diagnostics can be seen in, e.g., Figure 3 of Sterl et al., 2012 and Figure 2.2 of IPCC, 2014. The processing task NemoAllMeanMap creates a map containing the mean of an ocean surface variable over the so-far simulated, "all", time. This processing task is mapped to the YAML representation *ece.mon.nemo_all_mean_map*:

```
- ece.mon.nemo_all_mean_map:
    src: "{{t_files}}"
    varname: tos
    dst: ./tos_nemo_all_mean_map.nc
```

The task extracts the variable *varname* from the list of NEMO output files in *src*, in this case the SST. It averages over the time span from the first to the last input file (e.g., one year if all files of a year-long leg are provided). Month lengths are taken into account for this first temporal mean. After the first leg, this map gets saved at *dst*. If the file already exists, the processing task computes the weighted temporal mean of the existing map and the newly created leg average. This simulation average is then saved at *dst* again.

To illustrate this further, here is a more concrete example: For a simulation over three years, with yearly legs, the processing task would first compute the average of the first year. Since one year of simulated time has passed, this is the same as the current simulation average. The task then saves this diagnostic at *dst*. After the second leg, it computes the time mean of the second year. Then, it aggregates the two yearly averages into one, two years long simulation average. The same procedure repeats for the third year. As the saved simulation average now covers two years, it will have twice the weight as the new annual mean when aggregating the averages. This saving procedure is implemented in the *save()* method of the generic Map task (cf. Figure 3).

As was mentioned in section 3.3.4, maps and temporal maps contain the global attribute *map_type* for visualization. This is set to *global ocean* for this processing task and the next one.

4.2.3. NemoTimeMeanTemporalmap

Temporal map diagnostics allow a user to see developments during a simulation as well as spatial patterns. They combine time series with map diagnostics. This processing task computes either the

```
netcdf tos_nemo_global_mean_year_mean_timeseries {
dimensions:
    time_counter = 25 ;
    bnds = 2 ;
variables:
    double tos(time_counter) ;
        tos:standard_name = "sea_surface_temperature" ;
        tos:long_name = "sea surface temperature" ;
        tos:units = "degC" ;
        tos:cell_methods = "time_counter: mean
                            (interval: 1 month)
                           area: mean" ;
        tos:coordinates = "nav_lat nav_lon" ;
    double time_counter(time_counter) ;
        time_counter:axis = "T" ;
        time_counter:bounds = "time_counter_bnds" ;
        time_counter:units = "seconds since 1900-01-01
           00:00:00" ;
        time_counter:standard_name = "time" ;
        time_counter:long_name = "Time axis" ;
        ...
    double time_counter_bnds(time_counter, bnds) ;
    ...

// global attributes:
    :comment = "Global average time series of **tos**.
    Each data point represents the (spatial and temporal)
    average over one leg." ;
    :source = "EC-Earth 4" ;
    :title = "sea surface temperature (Annual Mean)" ;
    :type = "time series" ;
    :Conventions = "CF-1.8" ;
}
```

Figure 4 Example excerpt of a *ncdump -h* call on an SST time series diagnostic. As was laid out in section 3.3.4, the diagnostic on disk contains one variable only, *tos*. The time coordinate *time_counter* has continuous bounds *time_counter_bnds*. Clearly visible are the global and variable attributes. They are in accordance with the CF conventions.

annual (if a leg is one year long) or monthly development of ocean surface variables. The former works analogously to the time series in section 4.2.1, without taking the spatial mean. The latter requires almost no data manipulation: As the input is assumed to be a list of monthly files, the data is appended and stored in one file instead of twelve. These two variants are implemented as two tasks, NemoYearMeanTemporalmap and NemoMonthMeanTemporalmap. Both inherit from NemoTimeMeanTemporalmap, which bundles some of their functionality but cannot be called by a user. They are mapped to *ece.mon.nemo_year/month_mean_temporalmap*:

```
- ece.mon.nemo_month_mean_temporalmap:
    src: "{{t_files}}"
    varname: tos
    dst: ./tos_nemo_month_mean_temporalmap.nc
- ece.mon.nemo_year_mean_temporalmap:
    src: "{{t_files}}"
    varname: tos
    dst: ./tos_nemo_year_mean_temporalmap.nc
```

The processing task opens the input files at *src*, specifically only the data of the variable *varname*. Depending on the task, a leg mean is either created or not. If a file already exists at *dst*, the processing task appends to the existing diagnostic. NemoTimeMeanTemporalmap inherits from the generic Temporalmap task and reuses its *save()* method (see Figure 3).

4.2.4. Si3HemisSumMonthMeanTimeseries

"As an interface between ocean and atmosphere, sea ice controls most of the heat, momentum and fresh water transfers in sea ice covered regions" (Sterl et al., 2012). It is thus an important component of the climate and relevant to monitor. Changes in Earth's sea ice content are driven by seasonal variability on the hemispheres. EC-Earth 4 should reproduce this. The processing task Si3HemisSumMonthMeanTimeseries allows to create time series for the hemispheric sea ice volume and area with an emphasis on their seasonal cycle.

Conceptually, it differs from the NemoGlobalMeanYearMeanTimeseries. Since sea ice volume and area are extensive quantities, their mean is less informative than the accumulated value. Instead of a spatial mean, a sum over latitude and longitude is thus necessary. The variables in NEMO and SI^3 are always stored as "per grid cell area", which is why the sum (and spatial mean) need to be area-weighted. The interesting ocean quantities, such as the SST, are intensive quantities. Hence, multiplying or dividing by area does not lead to changes in unit or physical meaning. But dividing sea ice volume and area by grid cell area leads to a different variable:

$$\frac{[m^3]}{[m^2]} = [m], \quad \frac{[m^2]}{[m^2]} = [1].$$

Sea ice volume per area becomes the mean sea ice thickness in a grid cell. Sea ice area per grid cell area is referred to as the sea ice concentration. When computing the desired sea ice diagnostics from NEMO output and multiplying with the grid cell areas, the units and standard names therefore need to be adjusted.

Instead of a global operation, the hemispheres are separated during computation, splitting Arctic from Antarctic sea ice. This is motivated by the fact that seasons on the northern hemisphere are opposite to those in the southern hemisphere. Global instead of hemispheric sums would partially hide the seasonal variability. An annual mean as in the NemoGlobalMeanYearMeanTimeseries is also not logical here. To highlight the seasonal cycle, a different operation is necessary. Possible options include: a mean over seasons; a semi-annual mean; monthly developments; the maximum and minimum total sea ice volume or area. Typical in the literature is an analysis of maximum and minimum values. Instead of computing these values manually, the March and September means are usually displayed (e.g., Sterl et al., 2012) since these months represent the end of the hemispheric summer or winter. It is visible from observational data (cf. NSIDC, 2020) that the Antarctic minimum occurs a bit earlier than the Arctic maximum. Hence, another possible choice is to use the February and September means for the Antarctic and March (or a later month) and September for the Arctic. Although it would have been conceptually easier to use maximum and minimum values[2], this processing task sticks to the more established diagnostic. A user can simply provide a selection of monthly means that best represents the development they want to see.

The Python class is mapped to *ece.mon.si3_hemis_sum_month_mean_timeseries*:

```
- ece.mon.si3_hemis_sum_month_mean_timeseries:
    src:
        - "{{mar_file}}"
        - "{{sep_file}}"
    domain: "{{rundir}}/domain_cfg.nc"
    varname: sivolu
    hemisphere: north
    dst: ./sivol_si3_north_sum_mar+sep_mean_timeseries.nc
```

The task extracts the variable *varname* from the file(s) in *src*. Using the domain configuration file in *domain*, it computes the area-weighted sum over the *hemisphere* specified by the user. Metadata, units, and variable names get changed as explained above. The diagnostic is saved at *dst*, using the *save()* method from Timeseries.

4.2.5. Si3HemisPointMonthMeanAllMeanMap

Similarly to ocean variables, the spatial distribution of maximum and minimum sea ice thickness (in this case, volume per area) and sea ice concentration are established and helpful diagnostics. The processing task Si3HemisPointMonthMeanAllMeanMap can create a simulation average of a sea ice variable in one month and on one hemisphere, for example: a simulation average of the sea ice concentration in February on the southern hemisphere. The task separates the hemispheres again since Arctic and Antarctic sea ice have opposing seasonal cycles. Besides this, the model user is only interested in the grid cells with sea ice content. This processing task thus masks all cells that are either on the other hemisphere or have the value 0. When displaying sea ice maps, only a part of the globe is shown usually to focus on the regional phenomenon (e.g., Figure 12 in

[2] Iris directly supports operations such as taking a minimum, maximum, mean or sum, so this would not have been difficult to implement.

Sterl et al., 2012 or Figure 4.14 in Vaughan et al., 2013). Although the underlying grid is the same, a presentation task will have to treat sea ice maps differently from global ocean maps. Thus, this processing task has the map type *polar ice sheet*.

Si3HemisPointMonthMeanAllMeanMap is mapped to the YAML representation *ece.mon.si3_hemis_point_month_mean_all_mean_map*. A usage example can be seen below.

```
- ece.mon.si3_hemis_point_month_mean_all_mean_map:
    src: "{{ice_file_sep}}"
    varname: sivolu
    hemisphere: south
    dst: ./sivolu_si3_north_point_sep_mean_map.nc
```

The task loads the data for the variable *varname* from the provided input file in *src*. This allows the user to decide which month they want to resemble the maximum/minimum sea ice content, as in the other SI^3 processing task. A simulation average is computed and saved at *dst*. See the explanation in section 4.2.2 for more details.

4.2.6. Si3HemisPointMonthMeanTemporalmap

This processing task is almost equivalent to the map task in the last section. But as opposed averaging over the month means, Si3HemisPointMonthMeanTemporalmap appends them. This results in a temporal map diagnostic on disk, showing the annual development of a sea ice variable on one hemisphere, for example: the sea ice concentration in February on the southern hemisphere in the years 1990 until 2000. The map type is *polar ice sheet* and the YAML representation is *ece.mon.si3_hemis_point_month_mean_temporalmap*.

```
- ece.mon.si3_hemis_point_month_mean_temporalmap:
    src: "{{ice_file_sep}}"
    varname: siconc
    hemisphere: south
    dst: ./siconc_si3_south_point_mar_mean_temporalmap.nc
```

The required parameters are the same as in section 4.2.5. Joining the monthly data can be compared to the procedure in the task NemoTimeMeanTemporalmap.

4.3. Processing Tasks for OpenIFS

The OpenIFS tasks process 2D variables in grid-point space, saved in GRIB files. Some more assumptions were made about the input data for these tasks. Again: If they are not met, the resulting diagnostics will either not be created or they might not show the expected results.

An important assumption concerns the output frequency of OpenIFS: All tasks assume a constant output frequency of the atmosphere model. In particular, monthly output is not considered constant since month lengths differ throughout a year. The processing tasks compute time bounds based on

the input and use the first time interval for this calculation. Additionally, temporal means do not get weighted in these tasks (as opposed to the NEMO/SI[3] tasks in section 4.2). As mentioned in chapter 2, diurnal or higher output frequencies are typical for the atmosphere. Thus, this requirement is fulfilled in almost all experiment setups. Finally, a leg length of one year is again expected. Longer or shorter lengths will not lead to failure but some of the file descriptions might be inaccurate, as with the ocean and sea ice diagnostics (e.g., the comment attribute might say "annual mean" despite being a half-year mean).

If the Python package iris-grib (Met Office, 2010a) is installed alongside Iris, GRIB files can be loaded into Iris cubes. They can then be modified and saved as CF compliant NetCDF files. When loading the files, Iris replaces the GRIB codes with names from the CF Standard Name Table, this mapping is stored in iris-grib. Since the package did not recognize all variables from the OpenIFS output by default, the monitoring tool extends it with custom keys relevant for atmosphere monitoring. As for the ocean diagnostics, variable names, metadata, and units are changed in the processing tasks to meet some of the CMIP data request standards.

The physical space in OpenIFS is discretized using a reduced Gaussian grid, as was explained in chapter 2. To keep the resolution more uniform, the amount of cells per latitude ring decreases towards the poles in this type of grid (Malardel et al., 2016). Reduced Gaussian grids are not directly supported by Iris: Latitude, longitude, and data points are all loaded as separate one-dimensional arrays. Typical operations such as area-weighted means had to be implemented manually. A standard way of handling reduced Gaussian grids would be to apply a regridding procedure, either manually or using tools like the Climate Data Operators suite (Schulzweida, 2019). This makes data operations and visualization a lot easier and similar. However, it would alter the output data significantly. Additionally, choosing an interpolation method is variable-dependent and not trivial (Shea, 2014). Monitoring an experiment should show the current model status in a form close to the saved output. Thus, the OpenIFS output does not get regridded by the processing tasks.[3]

4.3.1. OifsGlobalMeanYearMeanTimeseries

This processing task is the atmospheric equivalent of the NemoGlobalMeanYearMeanTimeseries for the ocean. It enables EC-Earth users to monitor the global annual mean of a 2D atmosphere variable, for example the two meter air temperature or humidity. These are then time series diagnostics, produced by the processing task OifsGlobalMeanYearMeanTimeseries. It is mapped to the YAML representation *ece.mon.oifs_global_mean_year_mean_timeseries*:

```
- ece.mon.oifs_global_mean_year_mean_timeseries:
    src: "{{gg_files}}"
    grib_code: 167
    dst: ./167_oifs_global_mean_year_mean_timeseries.nc
```

The task extracts the data for GRIB code *grib_code* from the list of GRIB files in *src*. 167

[3] For the same reasons, the presentation tasks visualize atmosphere diagnostics without interpolating to a regular 2D mesh.

(parameter name: 2t) is the ECMWF GRIB code for the two meter air temperature. The code is mapped to a CF compliant variable and the metadata gets changed accordingly. Before the contained data is modified, time bounds are added to the time coordinate. Since GRIB files do not contain this information, they are computed based on the difference between the first two time values. As mentioned before, this is possible since a constant OpenIFS output frequency is assumed. The temporal mean (spanning from the earliest until the last input file) is computed without weights for the same reason. The task takes a spatial mean using a manually implemented area-weighted mean.[4] If the input files do not span one year, the result will be the mean over the time spanned by the files. It is assumed that this is equivalent to one year. The task will still create a valid diagnostic on disk if this is not the case, but the metadata will be less accurate. The diagnostic gets saved at *dst* using the *save()* method from the Timeseries processing task.

4.3.2. OifsAllMeanMap and OifsYearMeanTemporalmap

The processing tasks OifsAllMeanMap and OifsYearMeanTemporalmap are the OpenIFS equivalents of the tasks described in 4.2.2 and 4.2.3. They are mapped to *ece.mon.oifs_all_mean_map* and *ece.mon.oifs_year_mean_temporalmap*, respectively.

```
- ece.mon.oifs_all_mean_map:
    src: "{{gg_files}}"
    grib_code: 167
    dst: ./167oifs_all_mean_map.nc
- ece.mon.oifs_year_mean_temporalmap:
    src: "{{gg_files}}"
    grib_code: 167
    dst: ./167_oifs_year_mean_temporalmap.nc
```

OifsAllMeanMap saves a simulation average map of the parameter *grib_code*, created from the files at *src*. OifsYearMeanTemporalmap saves a temporal map of the annual/leg (or monthly) mean. The saving procedure is the same as for the ocean and sea ice diagnostics of the respective type. An important difference is the *map_type* attribute: The map type of the created diagnostics is *global atmosphere*, since the reduced Gaussian grid cannot be visualized in the same way as the curvilinear NEMO grid. The metadata is again adapted to make the files CF and partly CMIP data request compliant. The diagnostics are saved at *dst*.

4.4. Presentation Tasks

The diagnostics on disk are visualized and presented using the tasks described in this section. Currently, two presentation tasks are implemented. Naming presentation tasks is not as standardized as for processing tasks (cf. section 3.3.3). The task/class/module name should be the presentation outlet, for example Markdown. Their YAML representation is preceded by *ece.mon.presentation* to make them distinguishable from processing tasks.

[4] The reduced Gaussian grid has equally spaced grid cells along one latitude ring. The area weights can thus be computed by calculating the area of this latitude ring and dividing by the amount of grid cells in it.

4.4.1. Markdown

The Markdown presentation task visualizes all input diagnostics on disk and creates a markdown document containing text, images, and GIFs. It can be instantiated using *ece.mon.presentation.markdown* in a YAML script. To illustrate what it does, a usage example is given:

```
- ece.mon.presentation.markdown:
    src:
        - ./expid_scalar.yml
        - ./description_scalar.yml
        - ./simulatedyears_rte_scalar.yml
        ...
        - ./2t_oifs_all_mean_map.nc
    template: "{{rundir}}/markdown_template.md.j2"
    dst: ./markdown
```

The task iterates over all paths contained in *src*. For each file, it first determines the diagnostic type. Based on this, a sequence of visualization steps is initiated. While scalar diagnostics are best presented as plain text, time series and (temporal) maps should be presented as images. This assignment is shown in Table 2. At the end of these steps, a Python dictionary called *presentation object* is created for each diagnostic on disk. It contains keys *title*, *presentation_type*, *data* (for text) or *path* (for media), and optionally *comment*. The presentation objects and the Markdown template file *template* are parsed using the Jinja2 template engine.[5] This results in a markdown file *summary.md* at *dst*. The monitoring tool provides an exemplary template but a user can customize it or create their own without changing the implemented presentation task.

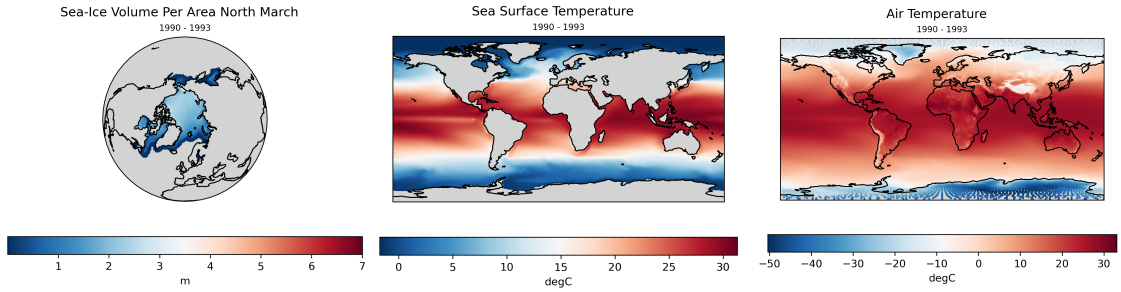
Diagnostic Type	Presentation Type
scalar	text
time series	image
map	image
temporal map	image

Table 2 A table showing the presentation types that the Markdown task assigns to diagnostics based on their type.

For scalar diagnostics, the visualization steps are straightforward: The YAML file is loaded using the PyYAML library. The structure of a scalar diagnostic on disk already resembles a dictionary close to the structure required by the template (cf. the exemplary YAML file in 4.1.1). Therefore, only the key *presentation_type* has to be added with the value *text*.

Time series diagnostics are best visualized by plotting the values on a graph. The task calls a routine that creates such a plot. A plot title, axes, and units are added (as can be seen in Figure 6). This is then saved as a PNG file in the *dst* directory. Then, the Python dictionary for the presentation is created: *title* and *comment* are equal to the global attributes as saved in the NetCDF file (see section 3.3.4 or Figure 4). *pres_type* is equal to *image* and *path* contains the relative path to the

[5] Jinja2 processes input with a Python-like syntax to create custom output files. Since ScriptEngine uses Jinja2 templating in various places, it was consistent to use it here instead of other libraries. More information on Jinja can be found at https://jinja.palletsprojects.com.



(a) The sea ice volume per area averaged over three years. The diagnostic was created with the task from section 4.2.5.

(b) A visualization of the SST simulation average, created with the NemoAllMeanMap processing task.

(c) A visualization of the two-meter air temperature simulation average, created with the processing task from section 4.3.2.

Figure 5 These three plots illustrate the plotting routines for different map types as they are used in the Markdown presentation task. They are visualizations of *all_mean_map* diagnostics for the sea ice, ocean and atmosphere.

PNG file.

Maps and temporal maps are visualized with two-dimensional plots. For temporal maps, multiple of these are created and saved as GIFs. Maps are saved as PNG files. Based on the diagnostic type, a different creation and saving routine is called, leading to different file extensions in the *path* key. The *title* and *comment* values are equivalent to the global attributes in the diagnostics on disk. Both diagnostic types have another property in common: The *map type* attribute. The relevance of it is discussed in the next paragraph.

The map type attributes in (time) map diagnostics on disk determine how the presentation task visualizes it. This is independent of the final file type (e.g., PNG, GIF, MP4) and the presentation type (image or video). Instead, it affects how the plots themselves are created. Three different map types are currently implemented in the monitoring tool: *global ocean*, *global atmosphere*, and *polar ocean*. Figure 5 illustrates the relevant differences between these types. The general design of the plots does not differ: color bar, title, and subtitle design are the same. Figure 5a and 5b both use the visualization technique for the tripolar ORCA grid as it is recommended in the Iris user guide[6]. They differ in the chosen map projection: 5a uses an orthographic projection for less polar distortion. 5b uses the PlateCarree map projection to give a global view. Figure 5c uses the same map projection but a different plotting routine: Since OpenIFS output is defined on a reduced Gaussian grid, the data is stored in a 1D array (cf. section 4.3). Because of this different format, the standard 2D plotting procedure *pcolormesh()* can not be used directly. The currently used alternative creates a colored scatter plot, with dots at every point of the reduced Gaussian grid.

Users can also define some custom visualization options for individual diagnostics. Currently, the feature allows different colormaps for the (temporal) maps and a value range for all types of plots.[7] To use this functionality, the syntax in the *src* key is a bit different:

[6] https://scitools.org.uk/iris/docs/v2.2/examples/General/orca_projection.html

[7] The possible colormaps are listed in the Matplotlib documentation: https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html.

```
src:
    - ./expid_scalar.yml
    ...
    - path: ./tos_year_mean_temporalmap.nc
      value_range: [-2, 30]
      colormap: viridis
    ...
```

The created dictionaries are finally passed to the template engine. Based on the presentation type, a different syntax is selected to embed the image, video, or piece of text in the Markdown report. An excerpt of such a file is shown in Figure 6.

EC-Earth 4: Monitoring

Experiment ID: SI07

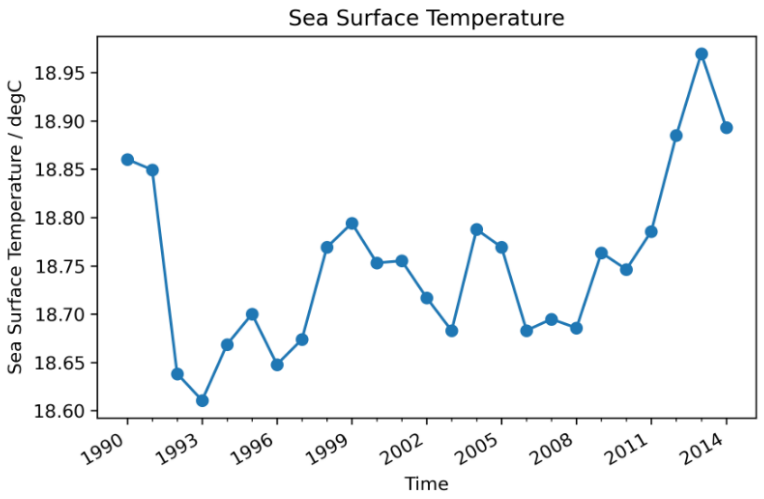
Experiment Description: Reimplement ICE_ALB_SPEC_WEIGHT with allocatable array. The array is set up in SURWN_MOD.

Simulated Legs: 25
Current amount of folders in output directory.

Simulated Years: 25
Current number of simulated years.

Disk Usage in GB: 425.6
Current size of output directory.

Sea Surface Temperature (Global Average Time Series)



Global average time series of **tos**. Each data point represents the (spatial and temporal) average over one leg.

Figure 6 An excerpt of a file created by the Markdown presentation task. It contains both a text and an image. Formatting differences can be clearly seen: While the value of *title* of the scalar diagnostics is printed in bold text, it is printed as a heading for the time series.

4.4.2. Redmine

While the Markdown report already supplies a solution for a structured visual presentation of monitoring diagnostics, it requires the user to connect to the supercomputer and retrieve the file manually. There is a demand for a more automated process: the diagnostics should get visualized and then automatically put on the web, with updates every time they are updated.

EC-Earth users and developers use Redmine–a project management web application–to track issues and the model development progress. The Redmine presentation task visualizes monitoring

diagnostics and uploads the visualization to the EC-Earth development portal as a new issue. This issue gets updated over time and allows users to directly view monitoring results on the web, while being able to discuss problems or ideas in-place.

In general, the task functions similarly to the Markdown task:

```
- ece.mon.presentation.redmine:
    src:
        - ./expid_scalar.yml
        - ./description_scalar.yml
        - ./simulatedyears_rte_scalar.yml
        ...
        - ./2t_oifs_all_mean_map.nc
    template: "{{rundir}}/redmine-template.txt.j2"
    subject: Issue Title
    api_key: "{{api_key}}"
    local_dst: ./redmine
```

First, the task creates visualizations and presentation objects of the input files in *src*, as described for the Markdown task. The images and GIFs are saved locally at *local_dst*. The presentation objects are then inserted into a document template, provided by *template*. Again, a user can modify this template to their liking, although the monitoring tool provides a suggested structure. This template uses the Redmine Wiki syntax, which is similar to Markdown, to create a formatted issue description. Using the *api_key* to log in, the task creates or updates an issue on the development portal, with the subject provided by the user. It uploads all necessary attachments and the description, providing live updates about the monitored experiment.

5. Exemplary Monitoring Results

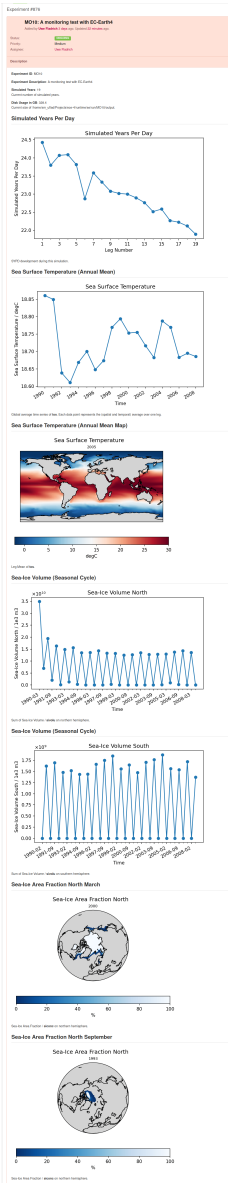


Figure 7 Screenshot from the EC-Earth development portal.

In these early stages of EC-Earth 4 development, the model is tested in the simpler general circulation model (GCM) configuration, as opposed to a fully coupled Earth system model. During the development of the monitoring tasks, the coupling with the sea ice component SI^3 has started and some early tests suggested considerable problems in the physical performance. This proved to be a good testing ground for the new monitoring tool. A 25 year-long monitored simulation was set up, with OpenIFS, NEMO and SI^3 activated. The test was performed on Tetralith, a high-performance computing cluster that is frequently used for large EC-Earth experiments. Tetralith is operated by the National Supercomputing Centre in Sweden (NSC) and incorporates 1908 computing nodes. (NSC, 2020) For this experiment 14 nodes, with 32 CPU cores each, were used, resulting in 448 cores.

To get a full overview of the monitoring tool's performance, almost all processing tasks were used. A selection of diagnostics was presented in real-time on the EC-Earth development portal with the Redmine task. Figure 7 contains an excerpt of the corresponding page. After the experiment a Markdown summary was created for each component, presenting all of the diagnostics.

In this chapter, an overview of the monitoring results from this exemplary EC-Earth 4 simulation is given. Section 5.1 discusses some conclusions about the experiment which can be drawn from the monitoring diagnostics. The computational performance of the monitoring tasks themselves is analyzed in section 5.2.

5.1. Performance Results of EC-Earth 4

This section will examine three illustrative monitoring results from this EC-Earth 4 experiment. The full Markdown summaries, as well as the YAML script which produced these diagnostics, are available on GitHub as additional resources for this thesis: https://github.com/valentinaschueller/ece-4-monitoring-resources.

The simulated years per day development was computed and visualized using the Timeseries processing task, the plot can be seen in Figure 8. In general, the speed of EC-Earth 4 looks promising. Typical major experiments like the CMIP 6 historical simulations span around 150 years. (Eyring, Bony, et al., 2016) An average of 22.7 SYPD, as measured here, means that these could be completed in about one week of computation time. The result also shows a decrease in model speed over time. While EC-Earth 4 took about 58 minutes for the first leg, the last one needed almost 67 minutes–a 14% increase. Past tests of EC-Earth 4 already indicated that

the model gets slower over time. The monitoring tool can track the speed automatically and in real-time. When developing solutions for this issue, the Timeseries task can thus be used to check if code changes affect the computational performance.

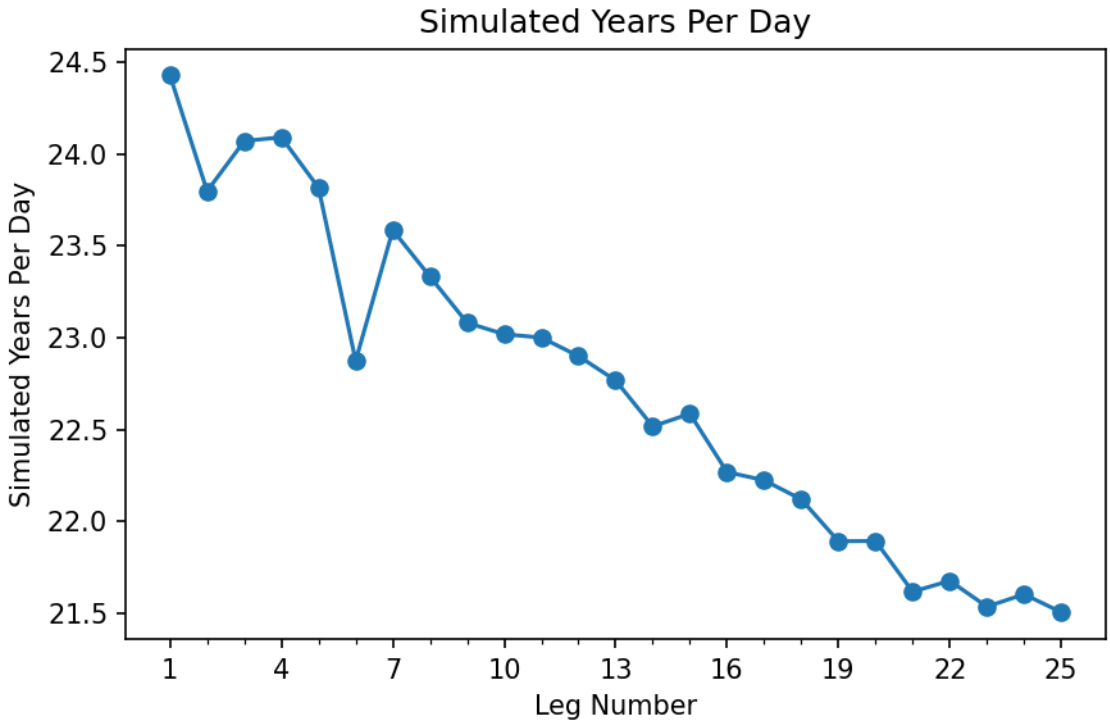


Figure 8 The SYPD development over the duration of the experiment. A decrease in this number indicates that EC-Earth 4 slows down over time.

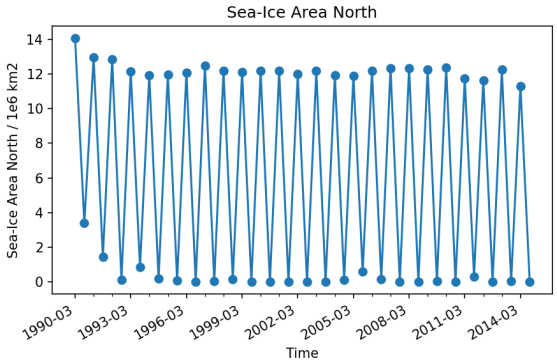
As was mentioned in the beginning of this chapter, EC-Earth 4 is currently not producing a realistic amount of sea ice. This can have various reasons and it is not the goal of this thesis to solve the underlying problem. Since the implemented tasks cover SI^3 output, this phenomenon should be visible in the monitored sea ice variables as well. The task Si3HemisSumMonthMeanTimeseries was used to create time series for the seasonal variability in sea ice area on the northern and southern hemisphere. The final plots can be seen in Figure 9.

"Sea ice typically covers about 14 to 16 million square kilometers in late winter in the Arctic and 17 to 20 million square kilometers in the Antarctic Southern Ocean." (NSIDC, 2019) The time series plots in Figure 9a and 9b show that the sea ice in EC-Earth 4 stays well below these numbers, especially on the southern hemisphere. Sea ice also almost vanishes at the end of summer on both hemispheres, which does not fit observed values (cf. NSIDC, 2020). EC-Earth 4 users can activate the necessary monitoring tasks to automatically get these results at runtime and thus detect problems in the physical performance.

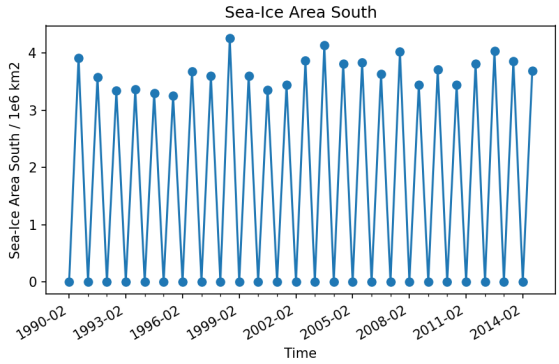
A more detailed view of sea ice development in this experiment can be seen in the additional resources on GitHub. The GIFs of temporal maps visualize the regional differences in sea ice over time.

Exemplary parameters to influence the sea ice development are the scale factors for the sea ice

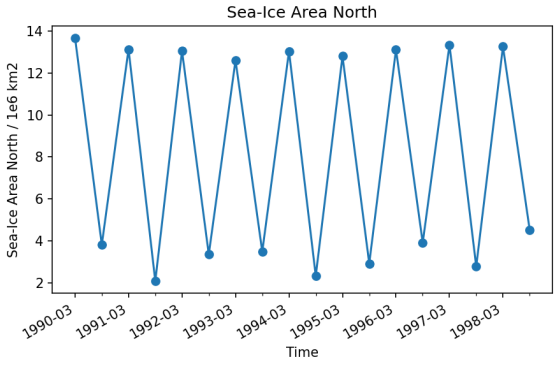
albedo mapping[1]: SI[3] sends one value for the sea ice albedo over all wavelengths to the atmosphere model, via the coupler. This value is mapped to six spectral intervals as required by OpenIFS using scale factors. To confirm the significance of this mapping, a short test experiment with higher scale factors was set up as well. In Figure 9c and 9d, the results can be seen. The effect of changing the scale factors is especially prominent on the northern hemisphere. As with the SYPD time series, the monitoring tool can be used in this example to see how adjustments in the setup change model results.



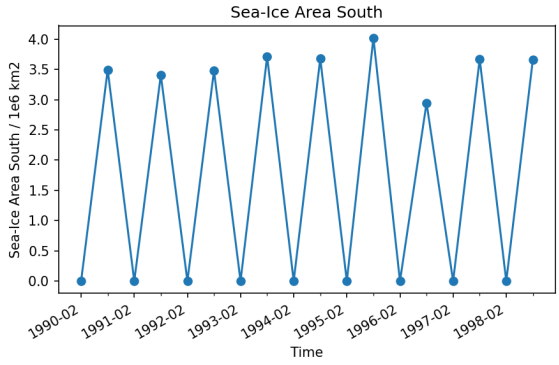
(a) The total area of arctic sea ice in March and September. The two months resemble the end of winter and summer on the northern hemisphere.



(b) The antarctic sea ice area in February and September, commonly used as the minimum and maximum values on the southern hemisphere.



(c) Arctic sea ice area development in a shorter test experiment with higher scale factors for the sea ice albedo mapping.

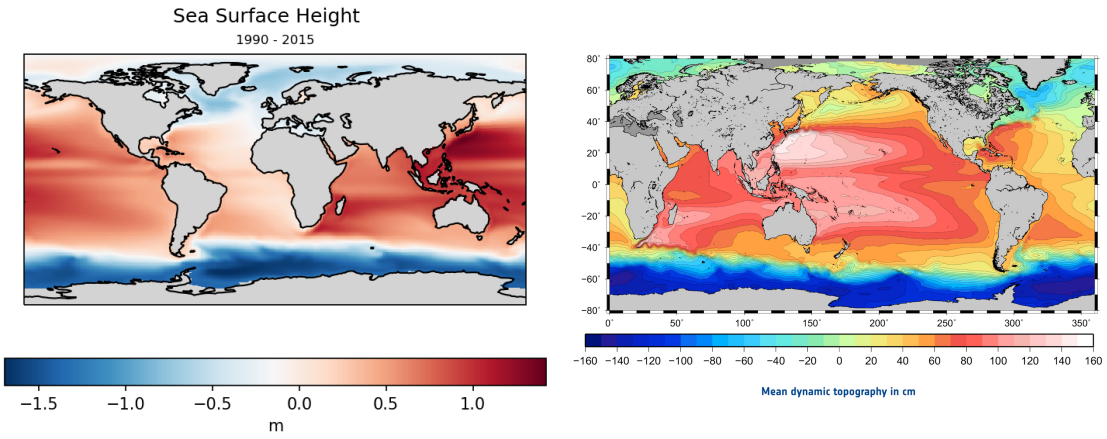


(d) Antarctic sea ice area development in a test experiment with higher scale factors for the sea ice albedo mapping.

Figure 9 These four plots show the development of the sea ice area on the northern and southern hemisphere. (a) and (b) display results from the test experiment. Higher scale factors for the sea ice albedo mapping led to the plots in (c) and (d).

The implemented map processing tasks can indicate if EC-Earth reproduces regional phenomena. One example for this is the mean dynamic (ocean) topography: the time-averaged sea surface minus the geoid ("the level surface that corresponds to the surface of an ocean at rest", Stewart, 2008). It is formed by the large ocean currents which are left when taking multi-year means of the sea surface. The associated variable to obtain the mean dynamic topography from NEMO output is the SSH. A comparison of EC-Earth 4 results and observational data can be seen in Figure 10. It shows that NEMO reproduces the regional differences inherent to the mean dynamic topography: SSH slopes in the different oceanic basins are clearly visible and the value range is similar to observations. The task NemoAllMeanMap creates diagnostics to confirm such properties quickly.

[1] Albedo is "the ratio of incident to reflected sunlight" (Stewart, 2008), a measure for the diffuse reflection of solar radiation.



(a) (b)

Figure 10 These two plots compare the 25-year mean of the SSH as computed by EC-Earth 4 (a) with the mean dynamic ocean topography as computed using observational data from the European Space Agency's GOCE satellite (b). Figure (b) was obtained from ESA, 2014, Copyright: ESA/CNES/CLS.

In general, the monitoring tasks enable users to see various properties of an EC-Earth simulation: OpenIFS, NEMO, SI^3, and even computational performance diagnostics can be created at runtime and allow to get a first idea of an experiment's characteristics. Serious problems in physical performance can thus be spotted during an experiment. A user can then decide to interrupt the EC-Earth simulation and enter a new testing cycle, saving computational resources.

The currently implemented tasks cover a broad range of variables but only a limited number of typical climate model diagnostics. For example, common atmosphere measures like the surface precipitation minus evaporation can not be computed yet, as with means of 3D oceanic variables. These missing diagnostics are discussed in more detail in the final chapter.

5.2. Computational Performance of the Monitoring Tasks

ScriptEngine can measure the time that individual Task instances take to complete. This feature was activated during the test simulation. It makes an analysis of the monitoring tool's computational performance possible. Table 3 shows the time per monitoring task for one exemplary leg. The task OifsYearMeanTemporalmap was not used to monitor this experiment, therefore it is missing in the list–in other tests, it took about as long as OifsAllMeanMap. The last line, the ScriptEngine task Command, signifies the amount of time needed for the model computation.

In the YAML script used for monitoring the EC-Earth experiment, more ScriptEngine tasks than the ones presented in chapter 4 were used. For example, output files are collected using the *Find* task and directories are created using the *MakeDir* task. The purpose of this section is the analysis of the newly developed tasks in comparison to an average EC-Earth leg. Other instantiated tasks were therefore left away.

In the exemplary leg in Table 3, EC-Earth 4 took about 65 minutes to simulate one year while the monitoring script's execution took 11.3 minutes. The leg length varied over time (see Figure 8),

Task	Total Time in s
Scalar	0.008
SimulatedYearsRteScalar	0.074
DiskusageRteScalar	0.98
Timeseries	3.649
Si3HemisPointMonthMeanAllMeanMap	5.964
Si3HemisPointMonthMeanTemporalmap	6.052
Si3HemisSumMonthMeanTimeseries	11.863
NemoGlobalMeanYearMeanTimeseries	15.761
NemoAllMeanMap	15.817
NemoYearMeanTemporalmap	16.662
Redmine	18.768
OifsAllMeanMap	153.917
OifsGlobalMeanYearMeanTimeseries	161.876
Command	3879.904

Table 3 Total execution times for the different monitoring tasks during an exemplary leg of the test experiment. The Command task starts the actual simulation.

while the time for monitoring tasks stays roughly constant. Monitoring one leg increased the computation time by 17.0% in this leg and between 16.9% and 19.5% in this experiment. In general, these values can be higher or lower, depending on the amount of diagnostics a user wants to create.

Table 3 reveals discrepancies between the monitoring tasks. Loading files and data manipulation takes time. As a consequence, physical performance tasks take longer than computational performance tasks. Due to the additional spatial operation, the time series tasks for physical performance often take longer than their map or temporal map counterparts. Creating OpenIFS diagnostics takes about ten times as long as the NEMO equivalents. The NEMO tasks on the other hand take noticably longer than their SI^3 counterparts. These differences arise from the time it takes to load input files: The SI^3 tasks load one to two files when they are called, whereas NEMO diagnostics are usually based on twelve input files. These have to be loaded and appended, resulting in a more time-consuming operation. The much slower OpenIFS tasks use GRIB files as their input, as opposed to the NetCDF output from NEMO and SI^3. The Python library used to load input and save diagnostics, Iris, takes longer to load GRIB files than anticipated during development.

To improve the monitoring tasks' performance, the OpenIFS diagnostics should get worked on first. Ideas for this will be discussed in detail in the final chapter, Discussion and Outlook. To reduce the impact of long execution times, an EC-Earth 4 user can run the monitoring tasks on a dedicated computing node. Then the simulation continues while the diagnostics are created and presented.

6. Discussion and Outlook

With the software described in this thesis, the programming task defined in the introduction is completed: A component-independent monitoring tool for the computational and physical performance of EC-Earth 4, based on the ScriptEngine framework, has been developed. The design principles from section 3.1 provide the basis for a more in-depth discussion of the implementation's strengths and limitations: The developed tool should be helpful, robust, user-friendly, modular, and extendable. It should furthermore stick to established standards where possible.

As was motivated in detail in chapter 4, the implemented processing tasks create meaningful, relevant, and established diagnostics to monitor climate model performance. The chapter Exemplary Monitoring Results gives a general overview of the used computational resources and the physical features of the test simulation. The diagnostics reveal problems with the sea ice coupling while also showing realistic values for oceanic variables. The live updates from the presentation tasks enable an accessible user experience of monitoring the current simulation. By the requirements from section 3.1, the tool can thus be considered helpful. Nevertheless, some very common and more complex diagnostics are missing: Diagnostics such as the AMOC index or the Barotropic Stream Function are typical ocean diagnostics that the tool does not monitor yet. For the atmosphere, measures like the surface precipitation minus evaporation or net heat fluxes should be added in the future. Besides that, the tool so far cannot analyze 3D variables or create output for custom regions of the ocean or land surface. Such extensions will make the tool more helpful to all EC-Earth 4 users. A list of useful diagnostics that cannot be created yet has therefore been assembled and can be worked on in the coming months.

Robustness as defined in section 3.1 consists of two aspects: The monitoring tool should neither interrupt the experiment nor slow down EC-Earth significantly. The implemented processing and presentation tasks catch the most typical user input errors with expressive log statements. In that case, the task will return without output and the next one is instantiated. Unexpected problems will however lead to failure of the task and, more importantly, a ScriptEngine error. Since the same instance of ScriptEngine is used for running EC-Earth and its monitoring, the experiment will then be interrupted by an error in a monitoring task. When testing a new setup or the monitoring tool specifically, this might be less of a problem, or even a positive effect. On the other hand, this contradicts the design principle of robustness as it was defined in section 3.1. The problem will not be solved by simply changing the monitoring tasks, as can be showed by a simple example: Suppose a processing task catches all exceptions that can happen during the *run()* method using a generic try-except statement:

```
class ExampleProcessingTask(Task):
    def __init__(self, parameters):
        # define required parameters here
        # e.g., 'src', 'dst',...
```

```
def run(self, context):
    try:
        # load input data
        # process data
        # save diagnostic on disk
    except:
        self.log_warning("Unexpected exception! Returning"
            )
        return
```

Even if one chose this wildcard approach[1], the EC-Earth experiment might stop because of a monitoring error: If a user forgets a required argument in the YAML script, the ScriptEngine base class Task will stop the initialization with a RuntimeError. A behavior consistent with the design principle would be the following: For all monitoring tasks, the ScriptEngine framework ignores exceptions and continues on with the next task. As soon as the monitoring is complete, it returns to its regular "runtime environment" mode where Task errors lead to experiment failure. Such a feature will be implemented in the ScriptEngine framework in a future release. To prepare for the change, a list of possible user input errors has been created, complemented by strategies of dealing with them.

By the results in chapter 5, monitoring an EC-Earth experiment increases the computing time by 16.9-19.5%. This is a significant slowdown affecting the EC-Earth experiment, unless one allocates a dedicated computing node just for the monitoring tool. The amount of time spend on monitoring diagnostics varies depending on the extent of the user's YAML script, but is also influenced by the chosen processing tasks: 70.3% of these 11 minutes is spent on processing the OpenIFS diagnostics (3 out of 33 instantiated tasks). Considering the fact that the goal was a lightweight tool, the performance needs to improve by a lot in the future, and working on the OpenIFS tasks should be the beginning of this effort. The root of the large time consumption is the loading of GRIB files. There are different possibilities to deal with this like, e.g., reusing loaded GRIB data across processing tasks. Depending on how this is carried out, this might however lead to storing very large amounts of data in memory. A more promising approach seems to be that the latest release of OpenIFS (43r3) supports writing NetCDF output using XIOS. Once EC-Earth 4 upgrades to this newer version of OpenIFS, the respective tasks will need to be rewritten to load NetCDF instead of GRIB files. But after these changes, monitoring OpenIFS output should be a lot faster and comparable to the processing tasks for NEMO and SI^3. Assuming that such new tasks take about 30s per instantiation–a conservative estimate based on the values in Table 3–the monitoring script used for the experiment in chapter 5 could be executed in about 4.5 instead of 11.3 minutes.

In general, the monitoring tool is not robust by the definition in 3.1, and the goal is not easily achievable without updates to EC-Earth 4 or the ScriptEngine framework. The design principle is

[1] This is not a recommended way of handling exceptions, as the Python documentation mentions: "Use this with extreme caution, since it is easy to mask a real programming error in this way!" (https://docs.python.org/3/tutorial/errors.html)

nevertheless very important to the monitoring tool's further development. The expected changes to EC-Earth 4 are promising in this regard.

Since the tool is based on ScriptEngine, EC-Earth 4 users will not have to adjust to a new or complex interface. With the naming scheme described in chapter 3, a consistent user experience has been established. A user guide has been added to the package for additional help. During development, a focus was set on processing and presentation tasks that are flexible and less specialized on single diagnostics. This creates a rich set of selectable monitoring diagnostics, even in this first release, and gives the user more control. It makes it more likely that users use this tool instead of their own scripts. On the other hand, the more abstract tasks might be harder to grasp. Usage examples were thus key in the accompanying resources. In general, the monitoring tool can be considered user-friendly by the requirements in chapter 3.

The modularity of the developed software mostly results from the modularity of ScriptEngine. Customizing the monitoring setup is unproblematic since few tasks depend on each other. A very important part is played by the task separation described in section 3.3. This allows new processing tasks to be developed without changing presentation tasks and vice versa[2]. The substructure of processing tasks shown in Figure 3 adds dependencies between them. This decision leads to less redundant code and was deemed preferable to completely independent processing tasks.

The monitoring tool is available as an open-source Python package. Automated tests have been added for each module, resulting in 95% code coverage and a basis for future development without unknowingly breaking existing functionality. Its underlying concepts are independent of EC-Earth components, as opposed to past tools like Barakuda, which was built to monitor NEMO output (Brodeau, 2017). No concept described in this thesis is closed-off: Processing and presentation tasks, as well as the defined diagnostic types and diagnostics on disk can be added onto in the future. The naming scheme in section 3.3.3 allows for a diverse set of diagnostics and processing tasks with consistent titles. Developer guidelines in addition to the user documentation summarize requirements and suggestions for future development, such as a logging policy and code structure recommendations. All of these aspects work towards the same goal: The monitoring tool is built to be extendable, thus fulfilling the fifth design principle.

The final principle mentioned in section 3.1 was sticking to established standards. The names of supported diagnostic types are influenced by literature on climate model performance, as well as geographic information systems. By trying to be CF compliant whenever possible, as well as using design choices and controlled vocabulary from the CMIP data request, common conventions from climate science were taken into account. Implemented processing tasks are always motivated by well-known diagnostics. New guidelines were only developed where none existed so far: For example, the naming scheme was developed as it became necessary and no other comparable analysis tool had a consistent way of naming diagnostics.

The design principles from chapter 3 have played a large role in developing the monitoring tool. Most of the goals described in that section have been achieved and the others are achievable in the

[2] as long as the diagnostic on disk interface does not have to be extended because of a new diagnostic or map type

near future.

While the developed tool is limited to usage with EC-Earth 4, the key design decisions in this thesis are not. In general, monitoring tools for coupled climate models profit from a component-agnostic design. This approach encourages long-term considerations that are less dependent on the current version of individual software components. It enforces code driven by standardization and extensibility, rather than being hard to maintain but tailored for one specific application. Various aspects of a simulation can now be monitored in one and the same place, giving EC-Earth 4 users a real-time overview of their experiment. The separation of data creation and visualization is a central element of this concept and can be applied to tools with similar purposes. It requires a concise interface, defined here as diagnostics on disk, which gives code maintainers clear guidelines for future developments. This makes the tool more robust and well-defined. Finally, the creation of extensive supplementary material (documentation, examples, and tutorials) and a high test coverage will be beneficial to the future usability of the monitoring tool.

The development of EC-Earth 4 will be ongoing in the next couple of years. Accordingly, the monitoring tool will require modification because of new or discontinued features. This long-term development can demonstrate which concepts and decisions enable or limit the extensibility and modularity. In the next months, more complex diagnostics like the AMOC index and regional maps should be added. The computational performance of the monitoring tool must not be neglected: Switching to NetCDF output for OpenIFS will reduce the problems found in chapter 5 significantly–but until that is the case, other solution strategies might have to be explored.

7. Code Availability and Resources

The monitoring tasks for EC-Earth 4 are available on GitHub at https://github.com/uwefladrich/scriptengine-tasks-ecearth/. The descriptions in this thesis are based on the code version at commit https://github.com/valentinaschueller/scriptengine-tasks-ecearth/commit/ad43d2c. The documentation is on Read the Docs: https://scriptengine-tasks-ecearth.readthedocs.io/. Additional material, such as exemplary monitoring results and scripts as well as presentations are in the EC-Earth 4 monitoring resources: https://github.com/valentinaschueller/ece-4-monitoring-resources/.

Acknowledgements

This thesis was written as part of an internship at the Swedish Meteorological and Hydrological Institute in Norrköping. To everyone who made this work out during these extraordinary times: I would like to express my sincerest gratitude to you. First and foremost, thank you to Uwe Fladrich for his supervision and support. These months have been an invaluable learning opportunity and they would not have been possible without him. Thank you to Univ.-Prof. Dr. Hans-Joachim Bungartz at the Technical University of Munich for examining, and thus enabling, this external thesis. Pablo, thank you so much for proofreading this thesis and coming along. To David Docquier, Torben Königk, Klaus Zimmermann, Pasha Karami, Klaus Wyser, Tim Kruschke, and everyone else at the Rossby Centre: Thank you for all of the feedback and discussions along the way.

Finally, I would like to thank all developers who enable projects like this with their useful packages and comprehensive documentation: The monitoring tasks make use of the Python packages Iris, Matplotlib, PyYAML, Numpy, Cartopy, Imageio, Python-Redmine, Jinja2, and Pytest.

Acronyms

AMOC Atlantic Meridional Overturning Circulation. 37, 40
CF climate and forecast. 14, 15, 19, 20, 22, 26, 27, 39
CHSY core hours per simulated year. 3, 19
CMIP Climate Model Intercomparison Project. 6, 14, 15, 19, 20, 26, 27, 32, 39
ECMWF European Centre for Medium-Range Weather Forecasts. 5, 27
ESM Earth system model. 1, 6, 32
ESMValTool Earth System Model Evaluation Tool. 3, 4, 16
GCM general circulation model. 6, 32
HPC high-performance computing. 3
SSH sea surface height. 20, 34, 35
SST sea surface temperature. 11, 13, 20–23, 29
SYPD simulated years per day. 3, 13, 14, 19, 32–34
XIOS XML I/O server. 6, 38

Bibliography

Balaji, V., Maisonnave, E., Zadeh, N., Lawrence, B. N., Biercamp, J., Fladrich, U., Aloisio, G., Benson, R., Caubel, A., Durachta, J., Foujols, M.-A., Lister, G., Mocavero, S., Underwood, S., & Wright, G. (2017). CPMIP: Measurements of real computational performance of Earth system models in CMIP6. *Geoscientific Model Development*, *10*(1), 19–34. https://doi.org/10.5194/gmd-10-19-2017

Brodeau, L. (2017). Barakuda. Retrieved August 11, 2020, from https://github.com/brodeau/barakuda

Craig, A. P., Jacob, R., Kauffman, B., Bettge, T., Larson, J., Ong, E., Ding, C., & He, Y. (2005). CPL6: The New Extensible, High Performance Parallel Coupler for the Community Climate System Model. *The International Journal of High Performance Computing Applications*, *19*(3), 309–327. https://doi.org/10.1177/1094342005056117

Eaton, B., Gregory, J., Drach, B., Taylor, K., Hankin, S., Blower, J., Caron, J., Signell, R., Bentley, P., Rappa, G., Höck, H., Pamment, A., Juckes, M., Raspaud, M., Horne, R., Whiteaker, T., Blodgett, D., Zender, C., & Lee, D. (2020). NetCDF Climate and Forecast (CF) Metadata Conventions, 183.

ECMWF. (2019a). Part III: Dynamics and Numerical Procedures. In *IFS Documentation CY46R1*. ECMWF. https://www.ecmwf.int/node/19307

ECMWF. (2019b). Part IV: Physical Processes. In *IFS Documentation CY46R1*. ECMWF. https://www.ecmwf.int/node/19308

ECMWF. (2019c). Part VII: ECMWF Wave Model. In *IFS Documentation CY46R1*. ECMWF. https://www.ecmwf.int/node/19311

ECMWF. (2020). Spectral representation of the IFS. Retrieved August 12, 2020, from https://confluence.ecmwf.int/display/FCST/Spectral+representation+of+the+IFS

ESA. (2014). Understanding the 'OC' in GOCE. Retrieved July 9, 2020, from http://www.esa.int/Applications/Observing_the_Earth/GOCE/Understanding_the_OC_in_GOCE

ESRI. (2012). ArcGIS Pro. *Redlands, CA*.

Eyring, V., Righi, M., Lauer, A., Evaldsson, M., Wenzel, S., Jones, C., Anav, A., Andrews, O., Cionni, I., Davin, E. L., Deser, C., Ehbrecht, C., Friedlingstein, P., Gleckler, P. J., Gottschaldt, K.-D., Hagemann, S., Juckes, M., Kindermann, S., Krasting, J., … Williams, K. D. (2016). ESMValTool (v1.0) – a community diagnostic and performance metrics tool for routine evaluation of Earth system models in CMIP. *Geoscientific Model Development*, *9*(5), 1747–1802. https://doi.org/10.5194/gmd-9-1747-2016

Eyring, V., Bony, S., Meehl, G. A., Senior, C. A., Stevens, B., Stouffer, R. J., & Taylor, K. E. (2016). Overview of the Coupled Model Intercomparison Project Phase 6 (CMIP6) experimental design and organization. *Geoscientific Model Development*, *9*(5), 1937–1958. https://doi.org/10.5194/gmd-9-1937-2016

Fladrich, U. (2020). ScriptEngine. Retrieved June 25, 2020, from https://github.com/uwefladrich/scriptengine

Frisch, J. (2014). *Towards Massive Parallel Fluid Flow Simulations in Computational Engineering* (Dissertation). Technische Universität München. https://mediatum.ub.tum.de/doc/1222749/1222749.pdf

Gleckler, P. J., Doutriaux, C., Durack, P. J., Taylor, K. E., Zhang, Y., Williams, D. N., Mason, E., & Servonnat, J. (2016). A more powerful reality test for climate models. *EOS*. https://doi.org/10.1029/2016EO051663

Gleckler, P. J., Taylor, K. E., & Doutriaux, C. (2008). Performance metrics for climate models. *Journal of Geophysical Research: Atmospheres*, *113*(D6). https://doi.org/10.1029/2007JD008972

Gurvan, M., Bourdallé-Badie, R., Chanut, J., Clementi, E., Coward, A., Ethé, C., Iovino, D., Lea, D., Lévy, C., Lovato, T., Martin, N., Masson, S., Mocavero, S., Rousset, C., Storkey, D., Vancoppenolle, M., Müeller, S., Nurser, G., Bell, M., & Samson, G. (2019). *NEMO ocean engine (v4.0)*. Zenodo. https://doi.org/10.5281/zenodo.3878122

Hazeleger, W., Severijns, C., Semmler, T., Briceag, S., Yang, S., Wang, X., Wyser, K., Dutra, E., Baldasano, J., Bintanja, R., Bougeault, P., Caballero, R., Ekman, A., Christensen, J., Hurk, B., Jimenez-Guerrero, P., Jones, C., Kallberg, P., Koenigk, T., & Willén, U. (2010). EC-Earth: A Seamless Earth-System Prediction Approach in Action. *Bulletin of the American Meteorological Society*, *91*, 1357–1363. https://doi.org/10.1175/2010bams2877.1

IPCC. (2014). *Climate Change 2014: Synthesis Report. Contribution of Working Groups I, II and III to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change*. Geneva, Switzerland, IPCC.

Le Sager, P., Tourigny, E., & Davini, P. (2019). EC-Earth 3 Post-Processing Tools. Retrieved September 9, 2020, from https://github.com/plesager/ece3-postproc

Ma, H.-Y., Xie, S., Boyle, J. S., Klein, S. A., & Zhang, Y. (2013). Metrics and Diagnostics for Precipitation-Related Processes in Climate Model Short-Range Hindcasts. *Journal of Climate*, *26*(5), 1516–1534. https://doi.org/10.1175/JCLI-D-12-00235.1

Malardel, S., Wedi, N., Deconinck, W., Diamantakis, M., Kuehnlein, C., Mozdzynski, G., Hamrud, M., & Smolarkiewicz, P. (2016). A new grid for the IFS. *ECMWF*. https://doi.org/10.21957/ZWDU9U5I

Met Office. (2010a). *GRIB interface for Iris* (v1.2). Exeter, Devon. http://scitools.org.uk/

Met Office. (2010b). *Iris: A Python library for analysing and visualising meteorological and oceanographic data sets* (v2.4). Exeter, Devon. http://scitools.org.uk/

NSC. (2020). Tetralith. Retrieved September 9, 2020, from https://www.nsc.liu.se/systems/tetralith/

NSIDC. (2019). State of the Cryosphere: Sea Ice. Retrieved March 9, 2020, from https://nsidc.org/cryosphere/sotc/sea_ice.html

NSIDC. (2020). Charctic Interactive Sea Ice Graph. Retrieved August 20, 2020, from https://nsidc.org/arcticseaicenews/charctic-interactive-sea-ice-graph/

Reichler, T., & Kim, J. (2008). How Well Do Coupled Models Simulate Today's Climate? *Bulletin of the American Meteorological Society*, *89*(3), 303–312. https://doi.org/10.1175/BAMS-89-3-303

Rousset, C., Vancoppenolle, M., Madec, G., Fichefet, T., Flavoni, S., Barthélemy, A., Benshila, R., Chanut, J., Levy, C., Masson, S., & Vivier, F. (2015). The Louvain-La-Neuve sea ice

model LIM3.6: Global and regional capabilities. *Geoscientific Model Development*, *8*(10), 2991–3005. https://doi.org/10.5194/gmd-8-2991-2015

Schulzweida, U. (2019). *CDO User Guide*. https://doi.org/10.5281/zenodo.3539275

Shea, D. (2014). The Climate Data Guide: Regridding Overview. (National Center for Atmospheric Research Staff, Ed.). Retrieved August 20, 2020, from https://climatedataguide.ucar.edu/climate-data-tools-and-analysis/regridding-overview

Smith, E. V. (2012). *Implicit Namespace Packages* (PEP No. 420). Retrieved June 25, 2020, from https://www.python.org/dev/peps/pep-0420/

Sterl, A., Bintanja, R., Brodeau, L., Gleeson, E., Koenigk, T., Schmith, T., Semmler, T., Severijns, C., Wyser, K., & Yang, S. (2012). A look at the ocean in the EC-Earth climate model. *Climate Dynamics*, *39*(11), 2631–2657. https://doi.org/10.1007/s00382-011-1239-2

Stewart, R. H. (2008). *Introduction to Physical Oceanography*. College Station, Texas. https://hdl.handle.net/1969.1/160216

Unidata. (2020). The NetCDF User's Guide (v4.7.4). Retrieved August 11, 2020, from https://www.unidata.ucar.edu/software/netcdf/docs/user_guide.html

Valcke, S. (2013). The OASIS3 coupler: A European climate modelling community software. *Geoscientific Model Development*, *6*(2), 373–388. https://doi.org/10.5194/gmd-6-373-2013

Vaughan, D., Comiso, J., Allison, I., Carrasco, J., Kaser, G., Kwok, R., Mote, P., Murray, T., Paul, F., Ren, J., Rignot, E., Solomina, O., Steffen, K., & Zhang, T. (2013). Observations: Cryosphere. In *Climate Change 2013: The Physical Science Basis. Contribution of Working Group I to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change* (pp. 317–382). Cambridge, United Kingdom, New York, NY, USA, Cambridge University Press. https://doi.org/10.1017/CBO9781107415324.012