Geosci. Model Dev., 9, 2755–2769, 2016
www.geosci-model-dev.net/9/2755/2016/
doi:10.5194/gmd-9-2755-2016
© Author(s) 2016. CC Attribution 3.0 License.

Geoscientific Model Development
Open Access
EGU

# YAC 1.2.0: new aspects for coupling software in Earth system modelling

**Moritz Hanke**[1], **René Redler**[2], **Teresa Holfeld**[2], **and Maxim Yastremsky**[2]

[1]Deutsches Klimarechenzentrum, Hamburg, Germany
[2]Max-Planck-Institut für Meteorologie, Hamburg, Germany

*Correspondence to:* Moritz Hanke (hanke@dkrz.de) and René Redler (rene.redler@mpimet.mpg.de)

Received: 7 December 2015 – Published in Geosci. Model Dev. Discuss.: 19 January 2016
Revised: 8 June 2016 – Accepted: 1 August 2016 – Published: 22 August 2016

**Abstract.** A lightweight software library has been developed to realise the coupling of Earth system model components. The software provides parallelised two-dimensional neighbourhood search, interpolation, and communication for the coupling between any two model components. The software offers flexible coupling of physical fields defined on regular and irregular grids on the sphere without a priori assumptions about grid structure or grid element types. All supported grids can be combined with any of the supported interpolations. We describe the new aspects of our approach and provide an overview of the implemented functionality and of some algorithms we use. Preliminary performance measurements for a set of realistic use cases are presented to demonstrate the potential performance and scalability of our approach. YAC 1.2.0 is now used for the coupling of the model components in the Icosahedral Nonhydrostatic (ICON) general circulation model.

## 1 Introduction

Within this study, we define coupling as the exchange of physical fields between model components formulated on different numerical grids. Here, we concentrate on model components which can be described, for example, as ocean, atmosphere, or sea ice and do not address the coupling of processes within these components.

In general, the tasks that have to be covered by any coupling software – named coupler hereafter – are the interpolation between source and target elements and the handling of the data exchange between them. The interpolation is usually based on a matrix–vector multiplication. The required coefficients for this multiplication can be provided by an external program or by the coupler itself. A significant part of the computation of the coefficients is the so-called neighbourhood search that determines a mapping between source and target elements. For a more detailed discussion on these aspects see Redler et al. (2010).

Neighbourhood searches between any pair of grids, source or target, with past generations of Earth system models with low spatial resolution and relatively simple block-structured grids did not consume a significant amount of CPU time. For example, in the Coupled Model Intercomparison Project Phase 5 (CMIP5) the majority of coupled models were operated at horizontal resolutions coarser than 0.5° (see Appendix 9.A in Flato et al., 2013). Furthermore, the connectivity between neighbouring cells could be deduced by the coupler directly. In addition with the very low degree of parallelism, providing efficient algorithms to perform the neighbourhood search did not pose a significant challenge. With the advancement of new numerical models formulated on irregular grids, a trend towards very high resolution of more than 0.1° in the numerical grid, and the high degree of parallelism with thousands of processes, flexible and efficient algorithms in the context of coupling are now required.

A variety of software exists in Earth system modelling for this type of coupling. A very prominent software package in this regard is the Earth System Modeling Framework (ESMF; Hill et al., 2004). The software is written in a mixture of C++ and Fortran90 and allows for a parallel exchange of data. In addition, it offers a rich functionality far beyond the pure coupling as we address in this paper. For example, with ESMF individual physics routines can be encapsulated, which allows further splitting of individ-

Published by Copernicus Publications on behalf of the European Geosciences Union.

ual model components. The software package encompasses about 800 000 lines of Fortran90 and C++ code[1].

Many coupled climate models use version 3 of the Ocean Atmosphere Sea-Ice Surface (OASIS) coupler maintained by CERFACS (Valcke, 2013). One major advantage of OASIS is the lightweight user interface whose implementation requires only very minor code changes in the user code. OASIS3 supports the most common grid types, including unstructured grids. For the data transfer, complete fields are collected by individual OASIS3 processes. In a quite recent development OASIS3 uses the Model Coupling Toolkit (MCT; Jacob et al., 2005). This allows for a direct and parallel data exchange between the participating model component processes. In its current version, OASIS3-MCT (Valcke et al., 2015) still requires the neighbourhood search to be performed and calculation of interpolation weights on a single process. Alternatively, external tools like ESMF can be used to provide the required interpolation weights a priori.

Further details about those and other important extant approaches have been assembled in a collection of articles by Valcke et al. (2012b) and in a review article by Valcke et al. (2012a). So why are we not going to use an existing software solution and adapt it to our needs? Our target is to create a software framework which allows us to easily add or replace algorithms in order to test new ideas and concepts not only in theory but also in the real life of numerical modelling. Our primary focus for our software development is on the Icosahedral Nonhydrostatic (ICON) general circulation model (Wan et al., 2013) and to a lesser degree on the model components available with the first version of the Max Planck Institute Earth System Model (MPI-ESM1) (Giorgetta et al., 2013). In addition, we contribute to the development of the climate data operator (CDO[2]) software.

ICON is designed to run on massively parallel systems. As such we require the coupling software to take full advantage of the hardware and to exchange coupling fields efficiently and in parallel. Furthermore, the handling of large grids on a single CPU may exceed the available memory; parallel algorithms working on local and thus smaller domains may offer one solution to memory exceedance. The demand to allow the model components in the future to change land–sea masks during the forward integration adds a requirement for an efficient and therefore parallel online neighbourhood search as well as the calculation of interpolation weights, rather than performing these steps offline in a pre-process step. Last but not least the software has to be sufficiently modular. This is required to, for example, allow for addition of new user-defined interpolation schemes to the existing code. These new schemes can be tailored to the specific elements in use or physical properties required. In addition, a modular design would allow an easy replacement or the addition of alternatives to already-implemented algorithms for the neighbourhood search, interpolation, communication, or other tasks. As the CDO software supports polygons beyond just triangles and quadrilaterals provided that the grid definition follows the CF (Climate and Forecast[3]) conventions, we require algorithms to work with a variety of polygons as well.

None of the existing software packages offers a combination of parallelised and efficient algorithms in particular for the neighbourhood search and interpolation, a user interface which minimises the interference with the existing model component code, all packed in a concise software package. Several of these features mentioned above have been addressed by OASIS4 (Redler et al., 2010), which is similar in functionality to OASIS3 but with improved support for the parallelism of climate model components. OASIS4 is able to perform efficient parallel online searches and handle parallel data exchange. However, in its current status, this software does not provide any support for unstructured grids and is restricted to quadrilateral elements. Furthermore, the development and support of OASIS4 has been stopped. Therefore, we have started a new approach and created a software framework that fits our specific purpose and allows us to easily use or integrate existing solutions where available. More important, our new framework allows the reuse of successfully tested algorithms and routines within other software projects like the CDO.

With this publication we present YAC with a special focus on a set of selected key aspects that we do differently compared to other coupling solutions. Our new approach is introduced in Sect. 2. We present preliminary performance results for a set of realistic use cases in Sect. 3. Further discussion of the results are in Sect. 4. We finish this paper with some major conclusions in Sect. 5 and provide an outlook on future work. In addition, we describe some key features of the user interface in Appendix A.

## 2 YAC – Yet Another Coupler

> A complete rewrite of legacy science software, with a specific focus on clarity, can prove useful. It uncovers bugs, provides a definitive statement of the underlying algorithm, lets scientists easily pose and answer new research questions, makes it easier to develop new visualisations and other interfaces, and makes the code considerably more accessible to interested third parties and the public. (Barnes and Jones, 2011)

YAC started as a small software project which we used to gain practical experience with a variety of methods, programming paradigms, and software tools which were already well known but had not found their way into the climate modelling community on a broader scale. Rather than doing this

[1] https://www.earthsystemcog.org/projects/esmf/sloc_annual

[2] https://code.zmaw.de/projects/cdo

[3] http://cfconventions.org/

in the context of an abstract software project, we chose the coupling for Earth system models as a real use case. We have taken this opportunity to generate a framework which allows us to test new concepts and algorithms. Our goal here is to be able to easily replace algorithms or add them for performance comparison purposes and to allow for an easy extension later towards a richer set of functionalities.

As already outlined before, we favour the OASIS-like approach due to its user-friendliness with respect to the specification of the Fortran application programming interface (API) and later use in a coupled Earth system model. In contrast to other existing software solutions this allows us to meet other boundary conditions set by the ICON project. A modular design allows for other software tools to benefit from subsets of the provided functionality. For example, a subset of internal YAC routines has already found its way into recent CDO versions. With the current version of YAC we provide different algorithms to calculate areas on the sphere which are enclosed by polygons. Tests are available to check the quality of the results for different cell types, which allows us to identify the most stable algorithm or replace the current one by a new alternative. In the same way, different search algorithms can be implemented and tested against each other with respect to performance and scalability.

## 2.1 Best practices

Best practices of code development found their way only recently into the literature of climate science (Clune and Rood, 2011; Easterbrook, 2012). Being a small group without formal management, we did not set up a formal document about a coding style, but we nevertheless defined a set of rules and principles for our development work in order to streamline and focus our effort.

### 2.1.1 Programming language

One of the most fundamental choices is the selection of an appropriate programming language. We decided to use C for the following reasons: in our particular case we are convinced that most parts of the software are much easier to program in C rather than Fortran. Specific examples are the allocation and reallocation of memory; in general C is more flexible in the handling of dynamically allocated memory. Noncommercial debugging tools for C are in general more mature compared to those available for Fortran. Debugging of Fortran programs with the GNU debugger (GDB) can be more difficult with some Fortran compilers. For example module subroutines get automatically renamed, and the inspection of data structures can sometimes cause problems. In our opinion the same argument holds for Valgrind[4], which helps during the development to detect memory leaks in a quite early stage. Writing portable code is far easier compared to Fortran as there is less diversity among C compilers than is the case with Fortran. In particular, the different levels of Fortran standards make it difficult for the programmer to write portable code. As already indicated by quoting Barnes and Jones (2011) at the beginning of this section, using C makes it much more attractive for a larger class of programmers outside the climate modelling community to join this effort and thus provide added expertise.

The CDO software in many aspects has requirements quite similar to a coupler, with the major difference being that the transfer of data happens between files rather than model components. Currently, the CDOs use in parts the same algorithm for the neighbourhood search and calculation of the interpolation weights as OASIS3. Hence, they also suffer from the same lack of performance (when data have to be processed on global high-resolution grids) and interpolation errors close to the poles. The CDOs are programmed in C, and thus the CDO software can directly benefit by using parts of YAC.

Since Earth system model codes such as ICON (and the majority of other climate model code) are written in Fortran, we also provide a Fortran–C interface.

### 2.1.2 Test suite

We provide short test programs that serve two purposes. The test programs demonstrate how to use particular subsets of the YAC functions and immediately explain the general usage (interface) and the context in which each function is intended to be used. Thus, the test programs themselves already serve as a documentation of our software. With the help of these test programs it is easier to debug the code and to detect programming bugs early in the development process, as these short test programs can be run at a high frequency. Furthermore, the tests allow for systematic checks of most special cases. Most of the test programs have the character of a unit test, which forces us to keep the different parts of the code independent with well-defined interfaces. Overall, the tests cover a large portion of the code and quickly highlight unintentional bugs due to unknown dependencies. In addition to these short test programs, we provide a set of examples which focus on the usage of the Fortran and C user API. These simple toy models demonstrate the use of the API, and the code sections can be transferred into real model code.

### 2.1.3 Documentation

Even though proper documentation of software is key to any software project, it is often neglected. It is very challenging to keep external documentation up to date with the software development unless you have a large development team with sufficient resources to dedicate some of these to documentation. In our case we rely on Doxygen[5]. Having the source

[4] http://www.valgrind.org

[5] http://www.doxygen.org

code documentation and the source code itself in the same place eases the task of keeping both parts in synchronisation with each other. Our main repository is automatically checked for new commits in regular intervals. When new commits are found, the YAC Doxygen website[6] is rebuilt, which guarantees the availability of an up-to-date documentation at any time. As the Doxygen configuration file is part of the software, users are able to generate a local copy of the HTML tree.

#### 2.1.4 Style guide

As indicated above, we have not provided a written style guide. Nevertheless we retain certain coding rules which we established while coding the first routines. We use long names for routines, data types, and variables, which makes the code longer but more readable and easier to understand. Wherever possible we avoid global variables. We restrict access to contents of internal data structures by using information hiding schemes. Access to these data structures is only available through a set of interface routines. We elaborate further on this in Sect. 2.1.7. We have kept the individual functions short to increase the readability of the code.

#### 2.1.5 Version control

We use the version control system git[7]. The main advantage for us is that git allows for smaller (local) commits during the development of a special feature which are independent of the availability of any network access. Only when a particular development cycle is finished are the changes pushed to the remote directory that is accessible to all, which then also generates a single notification by email.

#### 2.1.6 Optimisation

We first concentrate on the development of functionality without a particular focus on optimisation. Highly optimised code often comes at the cost of reduced readability, and the resulting code is often less flexible to changes of the algorithm. For some parts of the workflow, we have deviated from this paradigm, because based on our personal experience we know that high performance is key for certain aspects. Therefore, we implemented efficient algorithms for the search (see Sect. 2.3) immediately. The modular structure of the code facilitates this work as algorithms and even data structures can be replaced without any impact on other functions. In addition, we minimised the number of messages by packing data. Apart from some special support for completely regular grids in longitude and latitude we do not make use of any implicitly given connectivity as is the case for block-structured grids. Rather they are treated internally like any other unstructured grid, and we see some potential for optimisation in order to speed up the search on this particular type of source grid.

#### 2.1.7 Object-oriented programming

Even though the implementation is done in C, we tried to use an object-oriented programming paradigm. Most data are stored in objects, which are C structures that contain a number of related data fields. The data within an object are normally not accessed directly. Instead, it is read or modified using a set of interface routines.

The major advantage of this approach is that different parts of the code only depend on the interfaces to each other rather than directly on the actual data. Through use of the interfaces, all access to the data is controlled by the code that is directly associated with the data. This avoids content misuse and unwanted side effects that might happen when changing seemingly independent parts of the code. In addition, this allows alteration of data structures without modification of the original interface. Such a change might be required if an algorithm is modified or replaced.

One example of such an object within YAC is the grid. YAC supports multiple grid types. Each grid type has its own method of storing its data, such that memory consumption and access are optimal (e.g. regular grids do not store explicit connectivity information). However, there is one set of interface routines which supports all grid types. All algorithms in YAC access grid data through this interface. As a result, parts of the code that use this interface work with all grid types. For example, a new grid type would automatically work with all existing interpolation methods.

The interpolation method object is another example. In YAC an interpolation consists of a number of arbitrary interpolation methods. An interpolation starts by applying the first method to the data that are to be interpolated. Data points that cannot be handled by an interpolation method are passed to the next one. This is only possible because interpolation methods have a common interface, which is used to pass the data between them without knowledge of the other interpolation method type. The interpolation and a practical example of applying a sequence of interpolation methods are presented in Sect. 2.4.

### 2.2 Communication

YAC can be considered as a kind of abstraction which hides the complexity of the Message Passing Interface (MPI; MPI Forum, 2015) for the communication between model processes. Like MPI, YAC is programmed as a pure library which has to be linked to the model code. YAC enables the model component processes to communicate directly with each other depending on the coupling configuration.

Internally, YAC does all communication through its own communication layer. This allows us to do the communication independently of underlying low-level communication

[6]https://doc.redmine.dkrz.de/YAC/html/index.html
[7]https://git-scm.com/

routines (e.g. MPI). In addition, this layer is used to enhance the capabilities provided by the MPI library. Currently, there are two implementations of this layer. The standard one is based on MPI. For debugging purposes there is also a second implementation of the communication layer that simulates a parallel environment within a single process. This allows us to simulate a parallel application that uses asynchronous communication without the effects of non-deterministic behaviour.

In the MPI variant we avoid blocking operations in favour of an asynchronous communication scheme. For the sending we inherit a non-blocking buffered send from OASIS4. This routine works similar to a MPI_Bsend but does not need to attach and detach message buffers prior to and after the call. Instead, the buffer management is handled internally. For the receiving counterpart we provide an asynchronous receive operation. In contrast to typical asynchronous MPI receive operations, our implementation has no request argument. Instead the user needs to provide a function pointer to a callback routine. A request for the receive is set up internally by the communication layer. Once the receive request is fulfilled, the communication layer will call the callback function associated with the request. The data that were received are passed as an argument to the callback function.

We use these communication routines to split the complete workload into tasks. Each task has its own callback routine. By calling the asynchronous receive routine and by passing the respective function pointer, a task is “activated”. Once the respective data have been received by the communication layer, the task is processed. Each task can trigger other tasks by sending the respective messages. This generates dependencies between tasks. All tasks and their dependencies can be seen as a directed acyclic graph (DAG). The main advantage of this communication scheme is that independent tasks can be processed in any order, which should result in a good load-balancing. This concept was already introduced in OASIS4. However, the YAC implementation is far more general, which allows usage of this communication scheme by independent parts of the code without interference. It is used for nearly all communication in the initialisation phase of YAC.

## 2.3 Search

During the initialisation phase a so-called global search is performed. This is done once for each pair of source and target grids for which any interpolation is required. The grids can be distributed among multiple processes, and the global search does a basic intersection computation between all cells of both of these distributed grids. The result is a prerequisite for all interpolations. It greatly simplifies the interpolation-specific search described in Sect. 2.5. Furthermore, it reduces the communication between source and target processes in the interpolation search step.

In order to emphasise that we support a large variety of cells, we call these *polygons* from now on. For the global search we require that within a grid all vertices of the polygons and the polygons itself are unambiguously identifiable by IDs. For grids that are distributed among multiple processes the parts of the grid on each process need to have a halo with a width of at least one polygon. Each vertex, edge, or polygon must be owned by only one process. In the case of halos the user needs to provide the rank of the owner process of the respective data points.

Since computations of intersections between polygons can be computationally intensive, we use bounding circles as a basic estimate for possible intersections. On the sphere bounding circles have better properties than bounding boxes. A particular advantage of circles is the computation of the intersection between two of them on the sphere, because it is simple to program and fast to calculate. Furthermore, our experience with OASIS4 has shown that bounding boxes in the longitude–latitude space can have problems at the pole regions of the sphere.

If the computation of the intersection of the bounding circles yields a possible intersection between the polygons, the exact result needs to be computed. A basic operation required to do this is the computation of the intersection between two polygon edges. We currently differentiate between three different edge types: edges that are located on (1) great circles, (2) circles of longitude, and (3) circles of latitude. Intersection computation involving great circles is done in 3-D Cartesian space. In the pole regions of the sphere this is much more accurate than using trigonometric functions in the longitude–latitude space.

To reduce the number of checks that need to be done in order to determine all intersecting polygons of both grids, we have introduced an intermediate step. For this intermediate step we have implemented two different algorithms: a bucket- and a tree-based search algorithm.

### 2.3.1 Bucket-based search

Our first approach is a bucket search. It starts by mapping cells of the source and target grid to an intermediate grid that is relatively coarse compared to the source grid. We use a Gaussian grid as the intermediate grid. Due to the properties of this grid, the mapping operations have a low computational cost. Afterwards, it is sufficient to compute the intersection between source and target cells that are mapped to the same cell of the intermediate grid.

The bucket search suffers from the fact that the “cell densities” of the intermediate grid and the source and/or target grid can deviate significantly. This can result in a high number of source and/or target cells being mapped to a single cell of the intermediate grid. As a result more source and target cells might have to be compared to each other than necessary.

### 2.3.2 Tree-based search

As an alternative to the bucket search a search tree is implemented. Based on the bounding circles of all cells of the source grid, a search tree is generated. Using this search tree, we can then find all source cells whose bounding circles overlap with bounding circles of the target cells. Since the algorithm is based on the bounding circles of the grid cells, all computation is rather cheap. In our tests we have seen that the tree-based search algorithm is more than twice as fast as the bucket algorithm. Therefore, this is now the preferred algorithm in YAC.

## 2.4 Interpolation stack

In a typical coupling setup where a data field is sent from a source component to a target component, the grids of both components are not identical. This is particularly true for land–sea masks. Therefore, a target may not be completely covered by valid source polygons such that a particular interpolation may not be well suited to provide data for this target. In order to provide a solution to these problems in YAC, all interpolation methods are independent of each other but share a common interface. Multiple interpolation methods as listed in Sect. 2.5 can be combined in the so-called interpolation stack. This allows the user to freely select any sequence of interpolation methods individually for each coupling field.

In Fig. 1 we apply an example of such an interpolation stack for interpolating the World Ocean Atlas 2009 (WOA09) sea surface salinity (Antonov et al., 2010), visible as rectangles, onto the ICON R2B4 (Wan et al., 2013) grid, visible as grey triangles. As the primary interpolation we choose the first-order conservative remapping (Fig. 1a). Here only those target cells which are completely covered by valid (non-masked) source cells get a value. In a second step we try to interpolate remaining cells using the patch recovery method with a first-order polynomial (Fig. 1b). In the third step all remaining cells which still have no value are set to a fixed value, 999 in this case, and appear as dark grey areas over continents and narrow bays in Fig. 1c. We note that salinity with a value of less than 34 is coloured in light blue.

The interpolation stack can easily be modified via the Extensible Markup Language (XML) interface (see Appendix) to become, for example, a first-order conservative remapping, followed by a second-order polynomial patch recovery and then by a 4-nearest-neighbour interpolation.

We provide a small set of parameters to configure the individual interpolations. The complete list of parameters is described in the graphical user interface (GUI) user manual which is provided as a Supplement to this publication.

## 2.5 Interpolation

The results of the global search, which are independent of the actual interpolation method being used, act as the starting point for the interpolation. On the target processes, it provides for each local polygon a list of all source processes having overlapping source polygons. Based on this information, the interpolation starts by distributing the target points which need to be interpolated among the source processes. To process the received target points, the global search provides, on

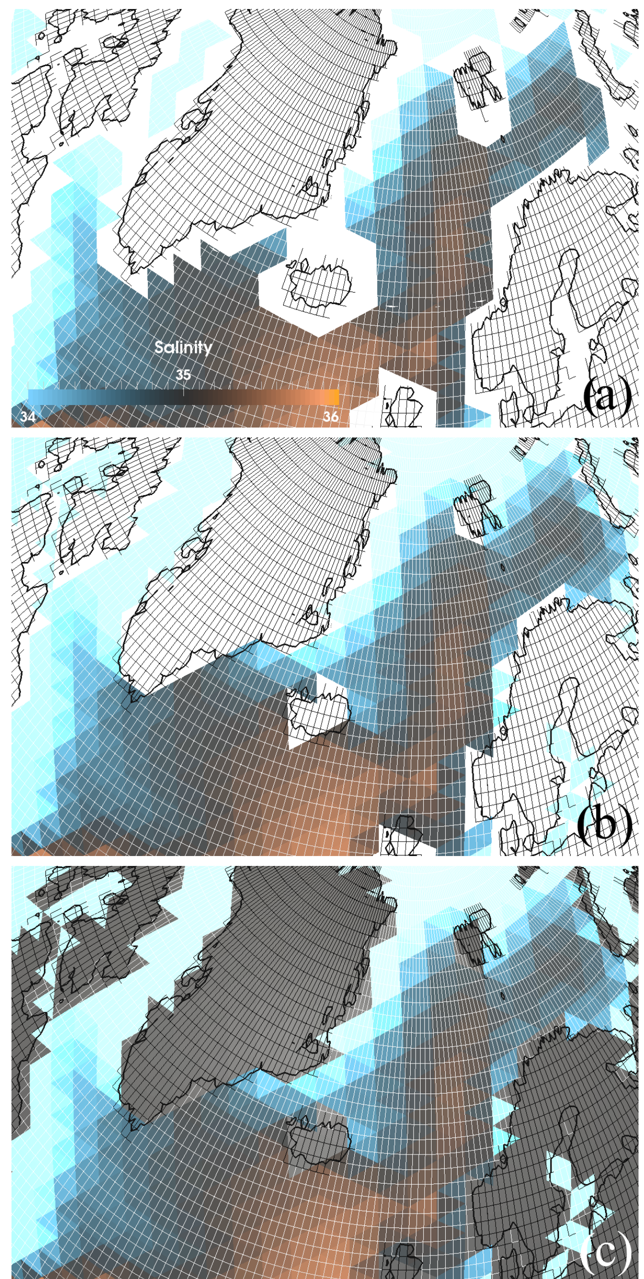


**Figure 1.** WOA09 January monthly mean sea surface salinity given on white rectangles interpolated to an ICON grid (triangles) using the YAC interpolation stack with first-order conservative remapping **(a)** plus first-order polynomial fit **(b)** and plus fixed values **(c)** (for further details see Sect. 2.4).

the source processes for each received target polygon, a list of all overlapping local polygons.

Once the interpolation is finished, each target point that can be interpolated has been assigned to a single source process. The source processes have the weights required to do the remapping of all target points assigned to them. The source processes might require data from other source processes to do the remapping of their target points. Therefore, in the actual remapping step, the respective data exchange between the source processes is performed first. Afterwards, the weights are applied to the source data in order to generate the target data, which are then sent to the target processes.

#### 2.5.1 First-order conservative remapping

Using the information provided by the global search, most of the work required for the first-order conservative remapping (Jones, 1999) is done. The main remaining task is to compute the partial overlaps between source and target cells, which are used to generate the interpolation weights. Different algorithms for the calculation of a polygon area on the sphere are implemented (see area.c and test_area.c for a comparison of these algorithms). Like in ESMF, as a default we follow L'Huilier's theorem.

#### 2.5.2 Patch recovery

Inspired by ESMF, we provide the so-called patch recovery interpolation (Zienkiewicz and Zhu, 1992). In our implementation this method is built on the very same set of preliminary information that is available for the conservative remapping. For one particular target polygon we consider all overlapping source polygons for the patch, very similar to the first-order conservative remapping. In addition we allow the user to extend this patch by one more row of source polygons surrounding the primary patch. For each source element in the patch we determine sampling points in analogy to Gauss points for a unit triangle and then apply a polynomial fit over these sampling points. The user can choose between first-, second-, and third-order polynomials, and furthermore the density or number of sampling points can also be chosen. The system of linear equations to solve for the polynomial fit can be rewritten in such a way that we are able to calculate a single value (weight) per source polygon in the patch which is only dependent on the geometry. We provide the details about the maths on our Doxygen website. In analogy to the first-order conservative remapping these weights are constant in time and can be applied to the physical fields later during the exchange.

#### 2.5.3 Nearest-neighbour

Points that need to be interpolated can be initially assigned to multiple source processes. For the nearest-neighbour search we first start by assigning each target point to only one source process. Based on the global search data, we then determine for each target point the $n$ closest source points in the local data, with $n$ being prescribed by the user through the XML configuration file. Afterwards bounding circles are constructed that have the target points at their centres. The diameter of these circles is chosen such that all points found for the respective target points are within the circle. When a halo polygon is within a bounding circle, a search request is issued to the actual owner of the halo cell containing all relevant information. It may be that a search request is forwarded multiple times until the final result is found. When the number of found source points is lower than $n$, the bounding circle is iteratively enlarged using the connectivity information until enough points have been found. The $n$ geographic inverse distances are calculated on the sphere. The inverse distances are normalised and stored as weights. As an alternative, simple weights with values $1/n$ are stored to provide the arithmetic average.

#### 2.5.4 Average

For the average interpolation we take the source polygon which contains the target point and compute the arithmetic average from all vertices of the polygon. We also provide the option to apply inverse distance weighting to the source vertices to do the interpolation.

#### 2.5.5 File

The file interpolation is able to read in interpolation weights from Network Common Data Form (NetCDF) files. Currently, it only supports a file format that is a simplified version of the weight files generated by the Spherical Coordinate Remapping and Interpolation Package (SCRIP[8]) library (Jones, 1998). However, it has the potential to be able to handle weight files generated by other programs (e.g. ESMF or CDO).

To avoid problems with memory consumption and runtime scalability, we use a parallel input scheme to read in the weight file. A subset of the source processes is selected to do the input/output (I/O). Each of these processes reads in an individual part of the data from the file. The data are then stored in a distributed directory (Pinar and Hendrickson, 2001). Afterwards, the source processes access the directory to get the data required to do the interpolation of the target points assigned to them.

The common use case for applying this interpolation method is to compute the weights once and then reuse them in order to save time in the initialisation of the model run. This approach is only feasible when reading in and distributing the weights is faster than computing them. Measurements show that computation of the weight at the start of the model run is not necessarily a significant performance factor (see Sect. 3) and that it depends on the number of processes and the number and complexity of the interpolations used.

[8]http://oceans11.lanl.gov/trac/SCRIP

Due to our concept of the interpolation stack (see Sect. 2.4) another potential use case is conceivable: if there are target points that require a special handling that is not covered by any interpolation method currently available in YAC, the user can provide a file that contains the weights for only these points. These weights could potentially be tuned by hand. The remaining points could then be interpolated using any standard interpolation or even another weight file.

#### 2.5.6 Fixed value

Last but not least we provide the option to assign a user-defined fixed value to target polygons or points, which is particularly useful when selected as part of the interpolation stack (see Sect. 2.4 and Fig. 1c).

### 2.6 Weight file generation

YAC is able to write the weights generated by an interpolation stack to a decomposition-independent weight file, which is supported by the file interpolation method (see Sect. 2.5). To activate this, the user has to specify it in the XML configuration file. Currently, this is supported for every interpolation stack except for stacks containing the fixed interpolation (see Sect. 2.5). Fixed interpolation is not supported because instead of weights it only generates target point lists that have to be assigned a certain value. This type of "interpolation" is usually not covered by typical weight file formats. However, it could be added if necessary.

### 2.7 Cell intersection computation

For first-order conservative remapping the coupler needs to compute the area of the overlap region between intersecting cells. It consists of three basic subtasks: (1) computation of the intersection point between edges, (2) clipping of two polygons, and (3) computation of the area of the overlap region.

Most couplers support edges that are represented by a section of a great circle on the sphere. If not identical, two great circles intersect twice. Usually, only one of both points is of interest to the coupler. There are three basic methods to compute this intersection point. The simplest method assumes that all edges are straight lines in latitude–longitude space, which makes intersection computation rather simple. For longitude and latitude circle edges (edges of cells of a Gaussian grid) this is accurate. However, for great-circle edges this gets more and more inaccurate the closer the edge is to a pole. A second method uses trigonometric functions to compute the intersection point of great-circle edges. Theoretically, this method should be very accurate, but due to numerical inaccuracy we observe problems when computing intersections close to the pole. The SCRIP library (Jones, 1998) provides the option to apply a coordinate transformation (e.g. Lambert equivalent azimuthal projection) for cells that are close to the pole. This is controlled with threshold latitudes. All cells that are above (for the Northern Hemisphere) or below (for the Southern Hemisphere) this latitude are transformed. The SCRIP user guide provides an example threshold value of $\pm 1.5$ radians. Even though the transformation improves accuracy, it also generates a discontinuity. A similar approach is used in OASIS4, where it causes problems in some very rare cases, because it generates "holes" in the grid; some parts of the sphere seemingly are not covered by the global grid. In YAC we apply vector operations in three-dimensional space to compute all intersections involving great-circle edges. This method is much more robust and does not require any special handling of close-to-pole cases. In addition to great-circle edges, YAC also explicitly supports latitude and longitude circle edges. Unfortunately, latitude circle edges introduce the possibility that two intersection points exist, which can be both within the bounds of the edges involved. This can occur when a great-circle edge intersects with a latitude circle edge. The computation of these points itself is not an issue, but it makes the clipping more complicated.

The clipping of two polygons computes the intersection between them. Typical clipping algorithms assume that all edges of the polygons are straight lines. Due to the type of edges supported by YAC, this is not the case here. We have tried to approximate the latitude circle edges using sections of great circles, but this increases computation time and makes the cell concave even though in latitude–longitude space it is convex. Currently, we use a modified version of the Sutherland–Hodgman clipping algorithm (Sutherland and Hodgman, 1974). This algorithm requires that one cell is either a convex cell that has only great-circle edges or a rectangular cell consisting of latitude and longitude circle edges. The second cell can be either convex or concave and can have any combination of edge types.

To compute the area of a spherical triangle consisting of great-circle edges, there are two basic formulas: Girard's theorem and L'Huilier's theorem. Originally, we used Girard's Theorem. While testing, we noticed an increasing error with decreasing cell size. L'Huilier's theorem is more complex but yields better results.

The output of the clipping is a polygon that is potentially concave and may contain all edge types. In a first step we assume that all edges of the polygon are great-circle sections. We split the polygon into triangles and compute their area using the above-mentioned theorem. In a second step all latitude circle edges are handled. For each of these edges the error that is made when assuming them to be a great-circle edge is computed and either added or subtracted from the overall area depending on the respective case. The sum of the partial source cell areas will always add up to the area of the respective target grid cell, up to numerical precision.

Figure 2 depicts a special clipping case near a pole. The triangle consists of great-circle edges and lies directly on the pole. The rectangle is a typical Gaussian grid cell consisting of latitude and longitude circle edges. The upper part of the

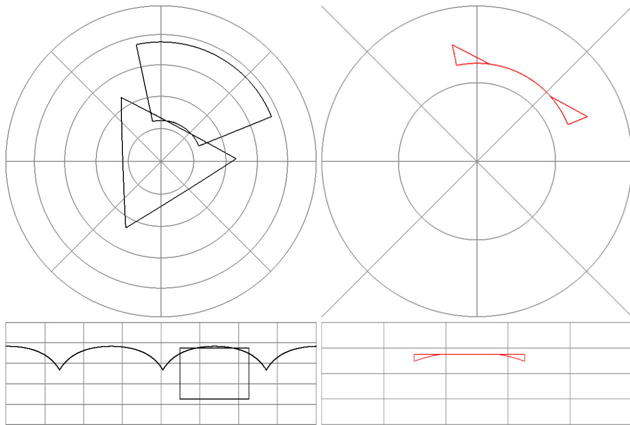

**Figure 2.** Clipping of ICON triangle and ECHAM rectangle.

figure shows a three-dimensional view directly from above the pole. The lower part depicts the same case in the two-dimensional "lon–lat space". The result of the clipping is a concave polygon with all supported edge types.

To illustrate the difference between great-circle and latitude circle edges in the area computation, we have compared the area of rectangular cells that only differ in the type of the edges. The reference cells are comprised of longitude and latitude circles' edges (typical Gaussian grid cells). The other cells have great-circle edges. For high-resolution grids (edge length of around 0.5°) the area difference is up to 1 ‰ for cells close to the pole. The area difference is higher for a low-resolution grid (edge length of around 5°) where the error goes up to 71 ‰.

## 3 Performance

In this section we present a first set of YAC performance measurements. As mentioned in Sect. 2.1.6, we do not yet concentrate on writing highly optimised code. Therefore, the measurements in this section are only preliminary. They can be seen as an estimate for the upper bound of the performance of YAC. In addition, scaling characteristics can be derived from them.

We have developed two toy models to do performance measurements: perf_toy_icon and perf_toy_cube. The toy model perf_toy_icon is based on an unstructured grid that consists of triangular cells. The other toy is based on a cubed sphere grid. Both toy models cover the whole sphere. Each toy model has one component, which defines an output field that is coupled to an input field defined on the component of the respective other model. Thus, each component acts as a source and target component, and consequently the search is performed within each (source) component.

All scaling measurements presented here were done on Mistral at the German Climate Computing Center (DKRZ), which at the time of testing was equipped with two Intel Xeon E5-2680 v3 12-core processors per node. We used the Intel compiler version 16.0.2 with Intel MPI version 5.1.2.150. The measurements were run using 24 MPI processes per node, and performance results are provided for two different grid configurations. For the first configuration the toy model perf_toy_icon uses an ICONR2B06 grid, which has 245 760 cells and around 35 km distance between the centres of neighbouring cells. In a typical atmosphere–ocean coupling setup, the ocean model component usually has a higher grid resolution than the atmosphere model component. To mimic this, we use a $405 \times 405$ cubed-sphere grid (984 150 cells; around 25 km distance between cell centres) for perf_toy_cube in this configuration, which represents the ocean in our setup. The second configuration uses grids with approximately 4 times as many cells: ICONR2B07 (983 040 cells) in perf_toy_icon and $810 \times 810$ cubed-sphere grid (3 936 600 cells) in perf_toy_cube. Four different interpolation methods are measured: first-order conservative interpolation (conserv), patch recovery with a third-order polynomial fit (patch), a file-based interpolation using the weights from the first-order conservative interpolation (file), and an interpolation that assigns a fixed value to all target points (fixed). Each component is run on its own set of nodes.

The values in Fig. 3 represent the maximum wall clock time required for the call to yac_csearch, which is responsible for nearly all of the runtime consumed by YAC in the initialisation phase of the model run. This routine does the complete search, which is comprised of the interpolation-independent global search (see Sect. 2.3) and the interpolation-specific weight computation. Due to the different resolutions, the total workload generated by this routine differs between the two components. Through adjusting the number of processes for each component individually, it would be possible to minimise the load imbalance between them. To simplify the experiment, we still use the same number of processes for each one.

A special case is the fixed interpolation, as it does not compute any weights. The measurements for yac_csearch using the fixed interpolation mainly represent the time consumed by the interpolation-independent global search. Here, the measured time represents the lower bound for the initialisation of the search in the respective grid configuration.

For up to 16 nodes per component the computation required by the interpolation-independent global search and the interpolation-specific weight computation are the two dominant runtime factors. This work scales well with the number of nodes, for up to 16 nodes or 384 processes per component. For higher node counts the computation time is negligible and the MPI communication contributes the preponderant part to the runtime.

From the measurements it is evident that the patch recovery interpolation is in our case the most time-consuming interpolation method. The file interpolation does not compute weights; instead it reads them from file and distributes them among the source processes. The time required for the read-

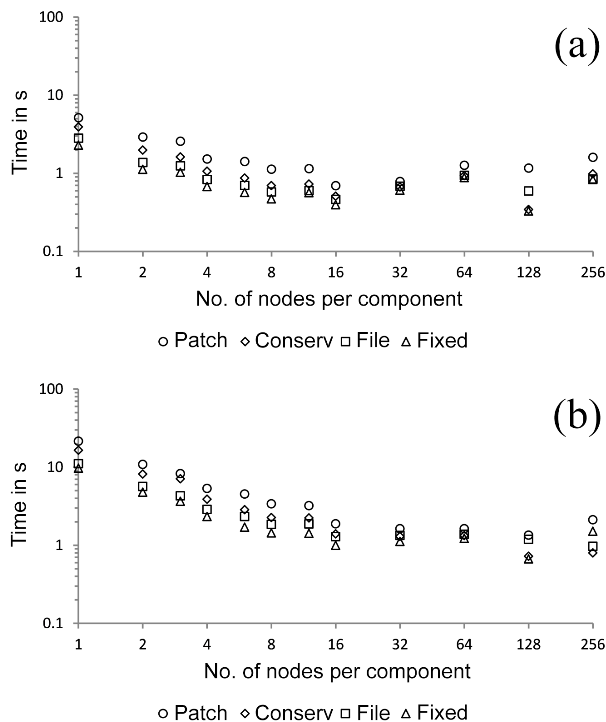


**Figure 3.** Time required for global search and calculation of weights for different interpolation methods in both directions between **(a)** an ICONR2B06 and 405 × 405 cubed-sphere grid and **(b)** an ICONR2B07 and 810 × 810 cubed-sphere grid.

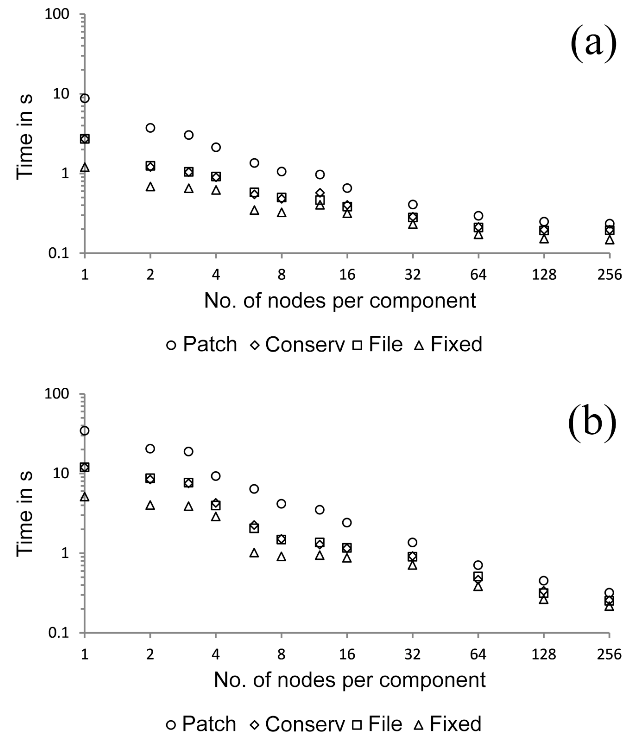


**Figure 4.** Time required for ping-pong exchange between **(a)** an ICONR2B06 and 405 × 405 cubed-sphere grid and **(b)** an ICONR2B07 and 810 × 810 cubed-sphere grid.

ing and distribution of the weights is the difference between the file and the fixed measurements.

The measurements for the ping-pong exchange depicted in Fig. 4 represent the time required by perf_toy_icon to execute MPI_Barrier followed by a put and get call. Perf_toy_cube executes the required matching calls. To improve accuracy, we have measured the total time it took to execute a significant number of the ping-pong exchanges and divided the result by the number of exchanges.

A put consists of three main parts. At first, the source processes exchange halo data among themselves, if required by the interpolation. Afterwards, each source process computes the target points assigned to it using the interpolation weights. The results are then sent directly to the targets.

As for the yac_csearch, the fixed interpolation is a special case. It does not require halo exchanges between source processes or the computation of target points, but it does send target points containing the fixed values from the source to the target processes. Again, the time shown here for the fixed interpolation can be considered as the lower bound for an exchange.

The patch recovery interpolation, with the parameters used in this setup, has the most weights per target point, which is why it is the slowest interpolation method for the ping-pong exchange. The conservative and file interpolations use identical weights and nearly identical communication patterns for the source-to-source and source-to-target communication, resulting in identical interpolation results on the target processes. This is why the time measurements for these two methods are very similar.

Both tested grid configurations exhibit similar behaviour for the measurements of yac_csearch and the ping-pong. Obviously, the second configuration takes longer due to the higher resolution. A higher resolution enables better scaling, which is most visible for high node counts in Fig. 4b.

## 4 Discussion and outlook

Here, we provide some insight into YAC search algorithms, supported interpolations, and parallelism, and we elaborate on some design aspects and best practices. For the first development step, we focus on solving existing problems and on providing basic functionality. At the same time, we take potential future problems into consideration and design the software such that later integration of new programming modules is not hampered or blocked. This flexibility was exercised during addition of the first-order conservative remapping and the patch-recovery interpolation (Sect. 2.5). In a second step we focus on optimisation, such as providing an

alternative to the bucket search (Sect. 2.3) and introducing asynchronous collective communication routines.

YAC comprises roughly 38 000 lines of source code for the library plus another 32 000 lines of code for unit tests. On our development platform, the YAC source code compiles in less than a minute, and the unit tests take seconds to run. These tests prove to be very useful during the development and also in the exploration phase. However, it remains a challenge to achieve full test coverage and to test all possible use cases and all internal functionality. Designing reasonable tests and the implementation of those tests takes some time. Initially, this tends to slow down the generation of library source code. But by strictly following the rule of writing tests along with the development of library code we are able to detect programming errors at a very early stage and improve the robustness of existing code; new development is much less likely to break existing code. We are thus convinced that the unit tests help to significantly speed up the development process.

Overall, we consider YAC as being relatively efficient in terms of the number of lines of code, compile time, and development cycle turnover. This efficiency allows us to test various methods and to experiment within a concise software framework. In addition, the code development for YAC also provides direct contributions to the development of CDOs, since most parts of YAC are programmed in C. As the CDO software is used worldwide in climate research, the parts of YAC which are already included in CDOs will probably be used more extensively than YAC itself.

In Sect. 3 we provide some preliminary performance measurements for typical model or grid configurations. The initialisation phase of YAC scales reasonably well with the number of MPI processes, provided that the local problem size remains sufficiently large. With an increasing number of MPI processes and a thus reduced local problem size, the communication increasingly dominates the time required to perform the neighbourhood search in the initialisation phase but still remains within a few seconds. As already mentioned in Sect. 3, the time required for the interpolation-independent global search corresponds with the measurements for the fixed interpolation. From this we surmise that the previous statements regarding scaling behaviour applies to the global search as well as the interpolation-specific weight computation, which is the difference between fixed and the other interpolation methods.

In our test case the file interpolation uses precomputed conservative interpolation weights and can thus be compared with the (online) conservative interpolation. For small node counts (up to 16 nodes) online weight calculation is slower than reading and distributing of the precomputed weights, but not by a large margin. For 32 to 64 nodes there is hardly any difference between the two interpolation methods. Depending on the chosen interpolation, the grid configuration, and the number of MPI processes involved, there is the potential for gaining a few seconds for lower node numbers, but the advantage diminishes to fractions of a second for higher node numbers (greater than 16 in our case).

For a low to medium number of nodes the exchange step scales well. Even for higher node counts the runtime still decreases with an increasing number of nodes per component. As explained in Sect. 3, the results for the fixed interpolation mainly consist of the time required to send the interpolation results from the source to the target processes. In addition to this, the patch recovery interpolation also requires a data exchange between the source processes and the remapping. The time required to perform these two steps accounts for the difference between the measurements of the two methods. The remapping should scale nearly perfectly with the number of nodes. This is probably the reason why the patch recovery interpolation scales better than the fixed interpolation for a low to medium number of nodes, depending on the problem size. For a higher number of nodes the time required for the remapping should be negligible. The source-to-source and the source-to-target communication should have similar scaling behaviour. This can be seen especially well in Fig. 4a; there both methods show nearly identical scaling behaviour for node counts higher than 16.

We note here that the time required to perform the search depends on the actual grid configuration, its masks, and the selected interpolation stack. When the distance between a target point and the nearest non-masked source points is large, the nearest-neighbourhood search can become very time-consuming. In this case the required search radius can be very large and encompass the compute domains of several MPI processes. This increases communication and computation time. Likewise, the calculation for the patch recovery is very costly when the third-order polynomial fit is selected together with the maximum number of fix points per source triangle as performed in YAC. Reducing the number of fix points, selecting a lower order for the fit, or reducing the local problem size by employing more MPI processes can significantly reduce the CPU time. We cannot provide a simple recipe to select the best interpolation configuration. Similar to tuning model physics for new model configurations, the interpolation results need to be checked and interpolation parameters need to be adjusted when necessary.

The communication scheme described in Sect. 2.2 works much better than that in OASIS4 due to the general interface which allows different parts of the code to successfully use the same mechanism without any knowledge of the other parts. For example, implementing the communication for new interpolation methods can be done without any impact on existing communication-related code. However, a disadvantage of this communication scheme is that it makes the code in some parts hard to read. For some sections of the code we include communication diagrams in the Doxygen documentation for clearer illustration of the interdependency between functions.

In its current status YAC provides all functionality required to couple the Earth system model components we

have in our focus while at the same time leaving ample room for extensions and improvements. From the very beginning, we have implemented algorithms from which we expect a reasonable weak and strong scaling behaviour. More sophisticated interpolation schemes like a second-order conservative remapping or a bicubic spline interpolation can help to improve the quality of interpolation results. Built-in support for the coupling of vector fields will greatly simplify the handling of these fields. A longer-term future development will be support for changing land–sea masks at runtime. In addition, the abstraction of the communication layer might enable a thread-level parallel functional decomposition of the work performed by YAC in the initialisation phase.

## 5 Conclusions

In support of constructing coupled Earth system models, we have redesigned and developed a coupler from scratch: YAC 1.2.0. Our development focus lies on the neighbourhood search, the calculation of interpolation weights, and the data exchange between model components. The efficient and fully parallelised algorithms directly support unstructured and block-structured numerical grids. This efficiency allows for the online calculation of interpolation weights during the initialisation phase of each model run. Furthermore, we offer an alternative to read in interpolation weights which are generated offline by the climate data operator software. This significantly widens the scope of numerical model configurations for which our coupling software can be applied. The software is available to the climate modelling community and is used in the next-generation Max Plank Institute for Meteorology Earth System Model. As an added value, the intentional choice of the programming language – in this case C – allows us to directly transfer parts of our software into the CDO and thus contribute to its improvement.

## 6 Code and data availability

Information about access to our software is provided on our YAC Doxygen website (https://doc.redmine.dkrz.de/YAC/html/index.html) under section code availability.

## Appendix A: User interface

As is the case with OASIS, our user interface is split into two parts. First we provide a set of library function calls that are to be called from the user code. Using these interfaces, the user makes static information, which is "known" by the model component at runtime, available to YAC. This is required to perform the search and data exchange. To allow for some flexibility to explore the functionality of the coupler, a set of free parameters has to be provided at runtime like the activation and deactivation of coupling fields or a particular choice of an interpolation or the definition of an interpolation stack. In OASIS3 this information is provided via the namcouple file, an American Standard Code for Information Interchange (ASCII)-formatted file, while in OASIS4 this kind of information is provided via XML-formatted files.

### A1 XML

In the recent literature (Ford et al., 2012, and articles therein) it has been pointed out that XML is the preferred language to describe metadata. Similar to OASIS4 we use XML files for the description of the coupling. Compared to OASIS4 the XML structure in YAC is simpler, more readable, and smaller in size. In the coupling XML file the user can specify the fields that have to be coupled. Other metadata are provided in the same file as well, like the interpolation sequence and the coupling frequency. In order to facilitate the generation of the coupling XML file, a Java-based graphical user interface is available.

For defining the structure of XML elements, we provide an XML Schema Definition (XSD). It describes the elements, attributes, types, keys, and key references (Fallside and Walmsley, 2004). The XSD allows validating the coupling XML instances. Furthermore, it helps to adjust programs that access the XML files, since it ensures that the XML elements satisfy a predefined structure.

We provide a component XSD and a coupling XSD. An XML instance of the component XSD defines the element structure for basic information about components: model name, transients, grids, and timing information. It is used as the input for our GUI. The GUI simplifies the generation of an XML instance of the coupling XSD, which provides detailed information on the actual coupling, like which transients are to be coupled; the configuration of the interpolations; and the role of the transients: whether they act as a source or target.

### A2 Graphical user interface – the XML GUI

With the design of a minimised XML Schema, the complexity of XML instance files is reduced in a way that it becomes easily readable for humans. Still, it is a tedious task to generate XML files by hand and to get all references correct. To ease the editing process of the coupling configuration, we provide a Java GUI that allows the user to create and manipulate the coupling XML configurations with a few clicks. The user can load component XML instances, couple them together, manipulate all coupling configuration settings, and produce a valid coupling XML file. For the implementation of the GUI we chose Java Swing because it provides a simple means for the creation of lightweight, platform-independent, portable GUIs (Eckstein et al., 1998), and because Java provides libraries for easy XML parsing and file I/O.

The initial GUI is showing a pane that is split into two parts, left and right. The user can load a component XML to each side and then couple the transients together. An arrow indicates the coupling direction. With a click on a couple, the GUI presents a detail window where the user can configure the interpolation settings, the time intervals, time lags, and some debug settings. Figure A1 shows a screenshot of the GUI in action. We provide a more detailed description of the GUI in our manual, which is available as the Supplement to this publication.

### A3 Application programming interface

With our API we closely follow the philosophy of OASIS4. For a more detailed discussion about the main principle the reader is referred to Redler et al. (2010); for a detailed description of the YAC API the reader is referred to our Doxygen website for YAC. As is the case for the internal library routines, the API is programmed in C. As ICON and most other climate codes are programmed in Fortran, we provide a Fortran wrapper for the API. In order to get rid of the problem with underscores in symbol names which are handled differently by individual Fortran compilers, we rely on the ISO_C_BINDING module which was introduced with the advent of the Fortran 2003 standard. The Fortran interface uses overloading with respect to data types and certain data structures. The Fortran interface accepts geographical grid information (longitudes and latitudes coordinates) as well as coupling fields in `REAL` or `DOUBLE PRECISION`. As the C API only accepts `double`, Fortran `REAL` is internally converted to `DOUBLE PRECISION` before being passed to the C API. In Fortran grid coordinates for regular grids in longitude and latitude are passed through the same interface routines (due to overloading) as the grid data for unstructured grids, while in C different interfaces are provided for the different grid types. We do not provide any direct access to our internal data structure. Instead, we hand back opaque handles (integer values) for components, grids, masks, and fields.

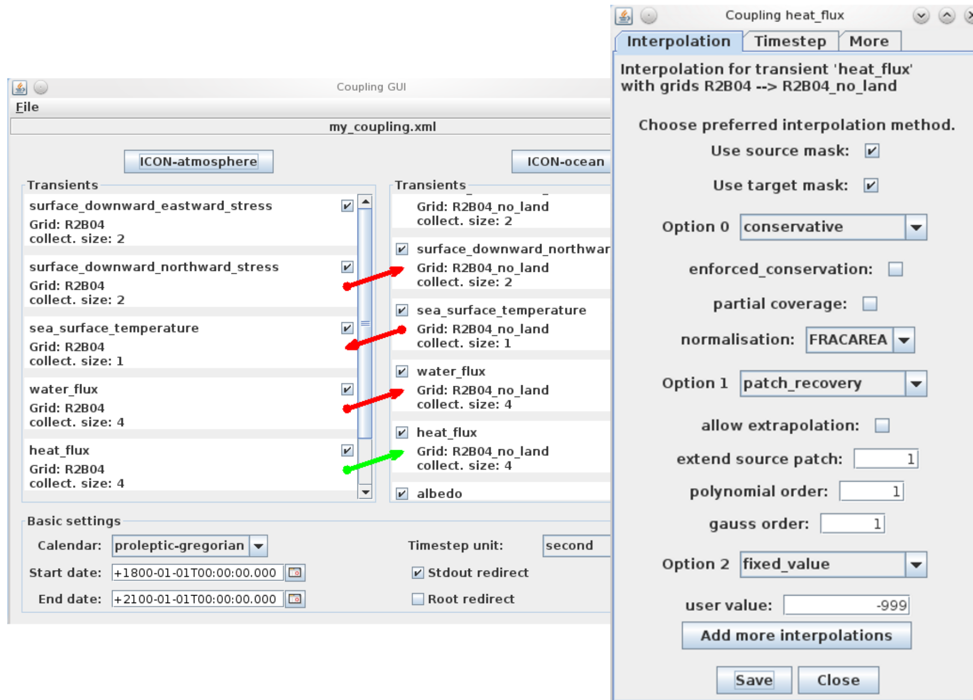


**Figure A1.** Coupling GUI with detail window.

**The Supplement related to this article is available online at doi:10.5194/gmd-9-2755-2016-supplement.**

*Acknowledgements.* We are grateful to our colleagues Jörg Behrens, Hendrik Bockelmann, and Thomas Jahns at DKRZ for very fruitful discussions during the development and adding the configure mechanism (TJ). Uwe Schulzweida took the burden and transferred several of our internal routines into the CDO package and was thus able to do far more testing of our algorithms than anticipated and thus helped us to make our software more robust for production use. We appreciate the luxury of having the freedom by our institutes, DKRZ and MPI-M, to do some development outside of any project requirements and without knowing beforehand where this would lead us.

Edited by: J. Fyke
Reviewed by: two anonymous referees

## References

Antonov, J., Seidov, D., Boyer, T., Locarnini, R. A., Mishonov, A., Garcia, H., Baranova, O., Zweng, M., and Johnson, D.: World Ocean Atlas 2009, Volume 2: Salinity, edited by: Levitus, S., NOAA Atlas NESDIS 69, U.S. Government Printing Office, Washington, D.C., USA, 184 pages, 2010.

Barnes, N. and Jones, D.: Clear Climate Code: Rewriting Legacy Science Software for Clarity, IEEE Software, 28, 36–42, 2011.

Clune, T. L. and Rood, R. B.: Software Testing and Verification in Climate Model Development, IEEE Software, 28, 49–55, 2011.

Easterbrook, S.: Code Design and Quality Control, in: Earth System Modelling – Volume 2: Algorithms, Code Infrastructure and Optimisation, edited by: Bonaventura, L., Redler, R., and Budich, R., Springer Briefs in Earth System Sciences, 51–65, Springer Heidelberg, Germany, 2012.

Eckstein, R., Loy, M., and Wood, D.: Java Swing, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

Fallside, D. C. and Walmsley, P.: XML Schema Part 0: Primer Second Edition, W3C recommendation, available at: http://www.w3.org/TR/xmlschema-0/ (last access: 18 August 2016), 2004.

Flato, G., Marotzke, J., Abiodun, B., Braconnot, P., Chou, S., Collins, W., Cox, P., Driouech, F., Emori, S., Eyring, V., Forest, C., Gleckler, P., Guilyardi, E., Jakob, C., Kattsov, V., Reason, C., and Rummukainen, M.: Evaluation of Climate Models, book section 9, 741–866, Cambridge University Press, Cambridge, UK and New York, NY, USA, doi:10.1017/CBO9781107415324.020, 2013.

Ford, R., Riley, G., and Redler, R. (Eds.): Earth System Modelling – Volume 5: Tools for Configuring, Building and Running Mode, Springer Briefs in Earth System Sciences, Springer Heidelberg, Germany, 2012.

Giorgetta, M. A., Jungclaus, J. H., Reick, C. H., Legutke, S., Bader, J., Böttinger, M., Brovkin, V., Crueger, T., Esch, M., Fieg, K., Glushak, K., Gayler, V., Haak, H., Hollweg, H.-D., Ilyina, T., Kinne, S., Kornblueh, L., Matei, D., Mauritsen, T., Mikolajewicz, U., Mueller, W. A., Notz, D., Pithan, F., Raddatz, T., Rast, S., Redler, R., Roeckner, E., Schmidt, H., Schnur, R., Segschneider, J., Six, K., Stockhause, M., Timmreck, C., Wegner, J., Widmann, H., Wieners, K.-H., Claussen, M., Marotzke, J., and Stevens, B.: Climate and carbon cycle changes from 1850 to 2100 in MPI-ESM simulations for the coupled model intercomparison project phase 5, J. Adv. Model. Earth Syst., 5, 572–597, doi:10.1002/jame.20038, 2013.

Hill, C., DeLuca, C., Balaji, V., Suarez, M., and da Silva, A.: Architecture of the Earth System Modeling Framework, Computing in Science and Engineering, 6, 18–28, doi:10.1109/MCISE.2004.1255817, 2004.

Jacob, R., Larson, J., and Ong, E.: MxN Communication and Parallel Interpolation in CCSM3 Using the Model Coupling Toolkit, Int. J. High Perform. C., 19, 293–307, 2005.

Jones, P. W.: A User's Guide for SCRIP: A Spherical Coordinate Remapping and Interpolation Package, http://oceans11.lanl.gov/svn/SCRIP/trunk/SCRIP/doc/SCRIPusers.pdf (last access: 18 August 2016), 1998.

Jones, P. W.: First- and Second-Order Conservative Remapping Schemes for Grids in Spherical Coordinates, Monthly Weather Review, 127, 2204–2210, doi:10.1175/1520-0493(1999)127<2204:FASOCR>2.0.CO;2, 1999.

MPI Forum: MPI: A Message-Passing Interface Standard Version 3.1, Tech. rep., Knoxville, TN, USA, 2015.

Pinar, A. and Hendrickson, B.: Communication Support for Adaptive Computation, Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia, USA, 12–14 March 2001, SIAM, 2001, available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.7896 (last access: 18 August 2016), 2001.

Redler, R., Valcke, S., and Ritzdorf, H.: OASIS4 – a coupling software for next generation earth system modelling, Geosci. Model Dev., 3, 87–104, doi:10.5194/gmd-3-87-2010, 2010.

Sutherland, I. E. and Hodgman, G. W.: Reentrant Polygon Clipping, Commun. ACM, 17, 32–42, doi:10.1145/360767.360802, 1974.

Valcke, S.: The OASIS3 coupler: a European climate modelling community software, Geosci. Model Dev., 6, 373–388, doi:10.5194/gmd-6-373-2013, 2013.

Valcke, S., Balaji, V., Craig, A., DeLuca, C., Dunlap, R., Ford, R. W., Jacob, R., Larson, J., O'Kuinghttons, R., Riley, G. D., and Vertenstein, M.: Coupling technologies for Earth System Modelling, Geosci. Model Dev., 5, 1589–1596, doi:10.5194/gmd-5-1589-2012, 2012a.

Valcke, S., Redler, R., and Budich, R. (Eds.): Earth System Modelling – Volume 3: Coupling Software and Strategies, Springer, Berlin, Germany, 96 pp., 2012b.

Valcke, S., Craig, T., and Coquart, L.: OASIS3-MCT User Guide, OASIS3-MCT 3.0, Tech. Rep. 1875, CERFACS/CNRS SUC URA, Toulouse, France, 2015.

Wan, H., Giorgetta, M. A., Zängl, G., Restelli, M., Majewski, D., Bonaventura, L., Fröhlich, K., Reinert, D., Rípodas, P., Kornblueh, L., and Förstner, J.: The ICON-1.2 hydrostatic atmospheric dynamical core on triangular grids – Part 1: Formulation and performance of the baseline version, Geosci. Model Dev., 6, 735–763, doi:10.5194/gmd-6-735-2013, 2013.

Zienkiewicz, O. and Zhu, J.: The Superconvergent Patch Recovery and a Posteriori Error Estimates. Part 1: The Recovery Technique, Int. J. Numer. Meth. Eng., 33, 1331–1364, 1992.