Using and programming ICON— a first introduction

S. Rast

`sebastian.rast@mpimet.mpg.de`

October 19, 2018

Foreword

This course is a first introduction to the ICON program (ICOsahedral Nonhydrostatic model) focussing on technical aspects of the atmospheric part using a “physics” package for climate simulations (“ECHAM” physics). The first part of this document explains the compilation of ICON and performing model runs. The most important namelists will be explained for the version using ECHAM physics. In the second part, the program code of ICON will be discussed. This part contains a survey of fortran techniques used in ICON before the most important data structures and subprograms of the ICON model are introduced. You will learn to create your own input namelist, how to use 2d– and 3d–fields including geographical co–ordinates, how to read data from netcdf files, to use date and time variables, and how to define new output streams.

This is a third version based on a first draft used in the ICON course 2016. The author is very grateful to many ideas by M. Giorgetta and the participants of the other courses, in particular T. Becker, M. Esch, S. Müller, and G. Zhou who all helped to reduce the number of mistakes and to make this text more readable.

Contents

Chapter 1

Getting started with ICON

The name ICON stands for ICOsahedral Nonhydrostatic model. It is in fact a collection of models to compute the general circulation of the atmosphere and the ocean including a land surface model. All these models can be used separately or in a coupled mode. Their common ground is not only the fact that they share a common infrastructure e.g. in terms of input and output, but also the fact that they are all based on a triangular grid derived from an icosahedron. The models are all designed in a way that grid nesting can be programmed in order to achieve a higher resolution in some regions. However, this is not yet "standard" together with the climate (ECHAM) physics package and may still require some programming effort. A distortion procedure of a global grid in order to get a higher resolution in some regions (and lower resolution in others) may also be applied. Even a model geometry that differs from a sphere and may be described as a "torus" is possible. Precisely, this means a rectangle with doubly periodic boundary conditions.

The icosahedron is a convex solid and consists of 20 equilateral triangles. The earth is represented by a perfect sphere that circumscribes the icosahedron. The original triangle faces of this icosahedron projected on the circumscribed sphere will be triangulated by great circles. Five triangles meet at the original icosahedron vertices, six at any other vertices. The resulting grid is therefore called to be irregular. This irregular grid will be optimized by some so–called "spring dynamics". The ICON model entirely relies on a calculation of all gradients in "gridpoint space" meaning that all derivatives occurring in the dynamics equations are approximated directly on the "icosahedral" grid. Why do we perform this paradigm change from spectral models like the predecessor ECHAM to pure gridpoint models like ICON? There are several reasons — numerical ones and computational ones. The biggest numerical advantage of the new grid is the absence of the singularities of a longitude–latitude grid at the poles. Generally, the distance between gridpoints of the ICON grid does not depend so much on the position on the globe as for longitude–latitude grids. Computationally, the recent development of massive parallel computers favors high resolution models with many gridpoints although the approximation of derivatives is numerically difficult. On the other hand, spectral models need permanent transformation from grid point space to spectral space and vice versa that also becomes numerically demanding when the resolution becomes really high (of the order of kilometres).

ICON is mainly written in FORTRAN90 with some small parts in C and consists of a parallel code including elements of vectorization. The atmospheric part uses equations appropriate for the description of nonhydrostatic flow and "physics" equations that describe turbulent and cloud processes on various levels of detail: Equations for the "Large Eddy Model" (LEM), "Numerical

Weather Prediction" (NWP), and climate ("ECHAM") physics also referred to as "Max Planck Institute physics" which is particularly suited for climate simulations.

In this course, you will get acquainted with the technical aspects of ICON, in particular, you will learn how to perform your own atmospheric climate simulations and how you can analyse your results. In a second part of the course, we will look into the code, learn something about the code structure, the most important data structures and how to change the code.

The ICON model will be made available to the scientific community under a common license of the Max Planck Institute for Meteorology (MPI–M) and the German Weather Service (DWD) and it will be distributed over the homepage of the MPI–M.

The versioning of ICON is a bit complex and reflects the parallel development in several "flavors" like "atmosphere", "ocean", and several more. There are four important branches tagging versions that reached certain milestones: `icon-aes` (atmosphere in the earth system, mainly echam physics in the atmosphere), `icon-nwp` (numerical weather prediction, mainly dynamics and physics of the LEM and NWP configurations of the atmospheric model), `icon-oes` (ocean in the earth system), `icon-les` (land in the earth system). All tags contain all model components, but the latest tag of `icon-aes` may not contain the most recent developments of the ocean physics although these are already included into the latest `ocean-oes` tag.

We won't use a specific model tag of the `icon-aes` branch here, just a certain version of the main `icon-aes` development branch. It is not suitable for scientific research since it was not thoroughly evaluated with respect to its scientific quality. The ICON model will be provided on the supercomputer `mistral.dkrz.de` in a tar–file

Listing 1.1: Archive file of the ICON model

```
mistral.dkrz.de:∼m218036/icon_course_2018/icon-aes.tar
```

For all practical exercises, we will work on the supercomputer `mistral.dkrz.de` that has more than 100,000 CPU cores distributed over 3,300 nodes and about 54 Petabyte of disk space at the Deutsches KlimaRechenZentrum (DKRZ), Hamburg. From your terminal, you access the supercomputer by

```
ssh -X -l <course_account> mistral.dkrz.de ⏎
```

Go into the working directory of our ICON course, create a new directory named after your course account, copy the model tar–file there, and unpack it:

```
cd /work/mh1049/icon_course/ ⏎
mkdir <course_account>; cd <course_account> ⏎
cp ˜m218036/icon_course_2018/icon-aes.tar . ⏎
tar xvf icon-aes.tar ⏎
```

The folder `icon-aes` will contain the model code and example scripts. We will discuss the content of this folder next.

1.1 Source code of ICON

The directory `icon-aes` contains the following files and sub–directories (the sub–directories are marked with a (d)):

(*i*) Source code administration, compiling, running, testing. ICON uses the GNU build system (GNU autoconf, GNU automake).

`.git*`: files of source code administration tool "git" (never modify these files by hand!)

files and directories belonging to the GNU build system: `aclocal.m4`, `config`(d), `configure`, `configure.ac`, `m4`(d), `Makefile.in`, `target_confmake.ksh`, `target_database.ksh`.
The directory `config` contains configuration files for various computer platforms.

`install-with-sct-on-mistral.sh`: Works on supercomputer mistral only; intended to compile ICON with timers; creates lines to be added to the runscript.

`make_runscripts`: This script generates run scripts from run script templates. The reason for this generic procedure is that the runscripts contain platform dependent calls of executables, platform dependent information about paths, and about processor configuration. On the other hand, namelists and input files are specific for each experiment. Platform dependent commands being equal for each experiment are collected in `run/exec.iconrun`, experiment specific settings in the experiment scripts `run/exp.*`. The script `make_runscripts` creates a machine dependent, experiment specific run script from these two components.

`data`(d): contains input data for ICON that either do not depend on model resolution or are very special. The general rule is to store input data away from the code versioning system. So, this directory should be used in exceptional cases only.

`README`: Contains a short description of how to compile and run ICON.

`README.xce`: Description of compilation on CRAY

`run`: Example run scripts for important experiments like AMIP–type runs, aqua planet or radiative convective equilibrium runs, see section 1.4.1.

`schedulers`(d): documentation on performing simulations at ECMWF

`scripts`(d): scripts for special applications, like post– and preprocessing, doxygen documentation, the automatic test system buildbot and other purposes.

(*ii*) Documentation

`doc`(d): contains some preliminary model documentation. In particular there is a namelist overview `Namelist_overview.pdf` describing all input variables of ICON.

`LICENSE`: Still a placeholder for the upcoming license.

(*iii*) Source code

`blas`(d): Basic Linear Algebra Subprograms is a collection of subprograms for basic vector and matrix operations

`externals`(d): Contains source code that is used by ICON, but not really a part of ICON. Examples are the `mtime` calendar package or the `yac` coupler used for grid interpolations between ocean and atmosphere grids. But also the source code of the surface model JSBACH is located here in a folder `jsbach`.

`lapack`(d): Linear Algebra PACKage contains subprograms for numerical linear algebra.

`src`(d): Source code of icon. The source code is organized in approximately 32 subdirectories which may themselves contain subdirectories in special cases. It is often difficult to find a certain module in the correct subdirectory because the structure is rather complex. It is not an easy–to–understand categorization of the source code files in "ocean" and "atmosphere" for example. There are also many parts of the code that were not tested for a while and may not work although they are still a part of ICON. The most important subdirectories are:

  - `namelists`(**d**): containing namelist definitions and subroutines reading the namelists.
  - `drivers`(**d**): main program `icon.f90` and entry points to various model parts like non–hydrostatic atmosphere in `mo_atmo_nonhydrostatic.f90`.
  - `atm_phy_{echam,les,nwp}`(**d**): parameterized "physics".
  - `atm_phy_psrad`(**d**): "psrad" radiation transfer code
  - `hamocc`(**d**): **HAM**burg **O**cean **C**arbon **C**ycle model
  - `ocean`(**d**): oceanic circulation
  - `io`(**d**): input and output subprograms

`support`(d): Subprograms for file handling, input and output (`cdilib.c`), time measurements and other helper routines written in C.

`vertical_coord_tables`(d): List of A and B coefficients of σ–hybrid co–ordinates for certain vertical grids. These are not directly used by ICON.

1.2 Basic Compilation

Before any simulation with ICON can be started, the source code has to be translated into an executable program by compilation. The compilation translates the human readable source code into a machine–readable binary code. The binary code depends on the architecture of the machine which is used, whether it is a "parallel machine" with several processors executing calculations for the same program or a "vector machine" that pipelines input to the computing unit or a mixed architecture. We describe the compiling on `mistral.dkrz.de`. Go into the main directory of the model (`/work/mh1049/icon_course_2018/<course_account>/icon-aes`) and perform the following commands:

```
./configure --with-fortran=intel
./build_command ⏎
```

The first commando creates the Makefile that contains all commands for the compilation of the model. The `configure` script makes use of the content of the `config` directory where information about various computers is stored and uses the information of `Makefile.in`. On "supercomputers", a variety of compilers in different versions is available and can be activated by the `module_load` command. Before the compilers can be used, you have to load them. In order to avoid this step, you can call `build_command`, a program generated by the `configure` procedure. This script loads all necessary modules and launches the make process. You can use several processors to compile the program in parallel. The `-j` *n* option to the make command inside the script `build_command` tells `make` to use *n* processors in the compilation (`make -j` *n*). If no errors occur, the following executables are produced:

```
icon-aes/build/x86_64-unknown-linux-gnu/bin/icon
icon-aes/build/x86_64-unknown-linux-gnu/bin/grid_command
icon-aes/build/x86_64-unknown-linux-gnu/bin/jsb4_driver
```

The first executable is the ICON simulation program for atmosphere and ocean circulation. The second executable `grid_command` is a program to generate the various grids based on an icosahedron. The executable `jsb4_driver` is running JSBACH4 in a stand–alone mode.

1.3 Model grid

1.3.1 Horizontal grid

The horizontal grid of ICON is based on the regular, convex icosahedron that is one of the Platonic solids. It consists of 20 equilateral triangles and has 20 faces, 30 edges, and 12 vertices. All 12 vertices are located on the surface of a sphere, the circumscribed sphere. Using the icosahedron directly would lead to a grid with 20 grid cells only. But each of the faces can be triangulated further to create a finer grid of more triangles. Fig. 1.1 represents an icosahedron with one triangulated face in front.

The North and South Pole of the earth are chosen to be located at two vertices of the icoshedron that are opposite to each other and marked as N and S in Fig. 1.1.

In principle, there is the possibility to triangulate the triangular faces of the icosahedron until the desired resolution is reached, to project the result onto the circumscribed sphere (or any sphere sharing its centre with the centre of the icosahedron) in a second step, and to optimize the resulting grid by some "spring dynamics" in a last step. In fact, this seems to be the conceptually easiest method to construct a grid based on the icosahedron. Another possibility would be to triangulate the triangles of the icosahedron after they are projected onto the sphere by the use of great circles (orthodromes). If necessary, optimization steps can be used after any of these triangulation steps. This is the algorithm used by the grid generator of the ICON model. However, the result after the optimization should be the same (except of numeric inaccuracies and e.g. rotations) since the energy minimum searched by the spring dynamics should be unique if the icosahedron symmetry is preserved. Nevertheless, no proof of this hypothesis is known to the author of this script. We describe the first method here although the exact algorithm is different and the status of a proof for the uniqueness of the result of the optimization is unknown.

First, we would like to see how a triangular grid on the triangle faces of the icosahedron is brought to the circumscribed sphere. The plane containing two neighbouring vertices P, Q on an icosahedron face and the centre Z of the circumscribed sphere intersects this sphere in a great circle. It is the shortest connection (orthodrome) between the two points P, Q on the sphere. We will see that the resulting grid contains vertices that are shared by five triangles that are the vertices of the original icosahedron and vertices shared by six triangles. Consequently, not all triangles are equal.

This grid on the sphere will be optimized in a next step by so called "spring dynamics". We give the idea of the optimization only, this is no accurate description of the algorithm. Imagine that we have a collection of springs all of them of the same strength and length. We fix a mass M at each triangle vertex and fix it with glue on the circumscribed sphere. We replace each edge by one of the springs. Depending on the actual length of the edge, we have to extend some springs a bit more for the larger triangles, less for smaller ones. We now melt the glue away and let the vertices move until an equilibrium is reached provided that there is some friction of the mass points on the sphere. By this procedure, we will obtain a slightly different grid of triangles which are still slightly distorted and of unequal size, however, the vertices reached positions that reflect some "energy minimum". These triangles are the basis of the ICON horizontal grid. Such a grid has particularly advantageous numeric properties.

The resolution of such a horizontal grid is symbolized by $\mathrm{r}_n\mathrm{b}_m$ where $n \in \mathbb{N}$ and $m \in \mathbb{N}_0$. The resolution is defined in such a way that the number of triangles $N_{\mathrm{r}_n\mathrm{b}_m}$ then given by

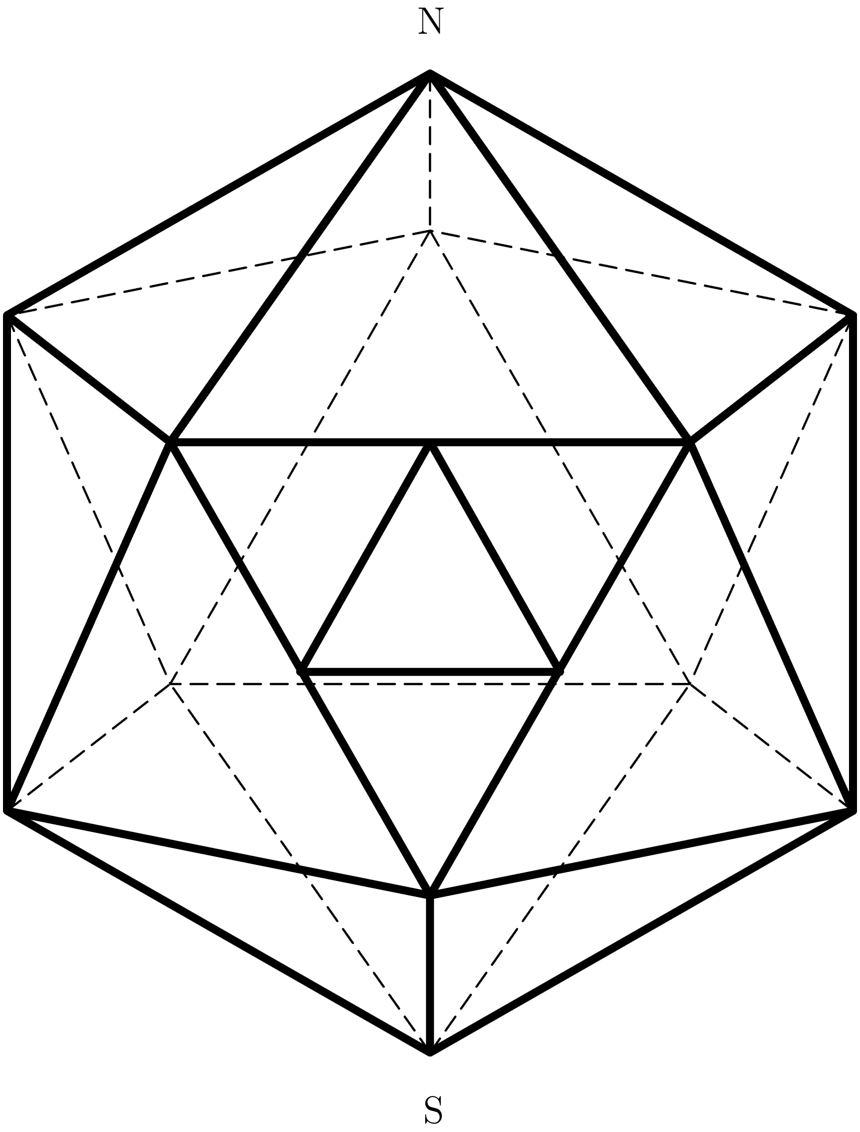


Figure 1.1: Icosahedron

$$N_{\mathrm{r}_n\mathrm{b}_m} = 20n^2 4^m \tag{1.1}$$

In the following, we describe the triangulation in detail and derive Eq. 1.1. We use plane triangles since these are easier to draw. In fact, the triangulation is performed in two steps.

First step of triangulation. Each of the 20 equilateral triangles is split into n^2 equilateral triangles (see Fig. 1.2). To this end, you first split each edge of a triangle (A, B, C) into n equal pieces by marking equidistant points $(A_1, \dots, A_{n-1})$ on edge a, $(B_1, \dots, B_{n-1})$ on edge b, and $(C_1, \dots, C_{n-1})$ on edge c going counterclockwise around the triangle. Then, you draw straight lines between corresponding pairs of points (A_i, B_{n-i}) (being parallel to edge c), (A_i, C_{n-i}) (being parallel to edge b), and (B_i, C_{n-i}) (being parallel to edge a) for $i = 1, n-1$. This fills your equilateral triangle with n^2 equilateral triangles. We call this procedure the triangulation by $n-1$ equidistant parallels. It is demonstrated for $n = 3$ in Fig. 1.2.

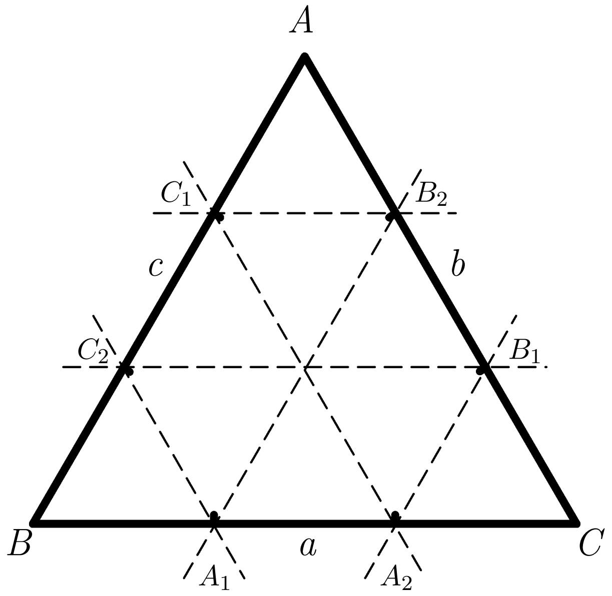


Figure 1.2: $n = 3$: Triangulation by 2 equidistant parallels

We prove that the triangulation by $n - 1$ equidistant parallels results in n^2 triangles in the case of an equilateral triangle:

Proof: Given an equilateral triangle (A, B, C) with edges (a, b, c), we complete it into a parallelogram over edge c with edges (a', b') being opposite to (a, b) (see Fig. 1.3. The lines (A_i, C_{n-i}) and (B_i, C_{n-i}) pass through A'_i and B'_i, $i = 1, \ldots, n-1$, too, since both triangles are equilateral. In such a way, we get n^2 parallelograms which are all similar to the parallelogram (A, C', B, C). Since all parallelograms have edges of the same length and two 60° and 120° angles, the lines (A_i, B_{n-i}) cut them in half forming equilateral triangles. If we also draw the lines (A'_i, B'_{n-i}), we get $2n^2$ equilateral triangles, but only dealing with triangle (A, B, C), we get n^2 equilateral triangles as stated above.

Second step of triangulation. The triangles resulting from step one are further triangulated. But in contrast to step one, we allow only a triangulation by $2 - 1 = 1$ equidistant parallels. According to the proof in step one, this results in $2^2 = 4$ triangles. This triangulation is then repeated on the resulting triangles. In total, the triangulation by 1 equidistant parallel is applied m times to each triangle resulting from the first step such giving in total 4^m smaller triangles. In total, we get $20n^2$ triangles from step one multiplied by 4^m triangles from step two. This results in Eq. (1.1) for resolution $\mathrm{r}_n\mathrm{b}_m$.

Dual hexagonal grid. The centres of the equilateral triangles contained in each triangle of the original icosahedron after triangulation are defined by the intersection of the angle bisectors (which are at the same time also altitudes) of the triangle. The centres of the triangles form a hexagonal grid that is said to be dual to the grid of triangle vertices. In the case of the ICON grid, the centres of the slightly distorted triangles form a dual grid of slightly distorted hexagons. Fig. 1.4 shows an example of such a hexagon.

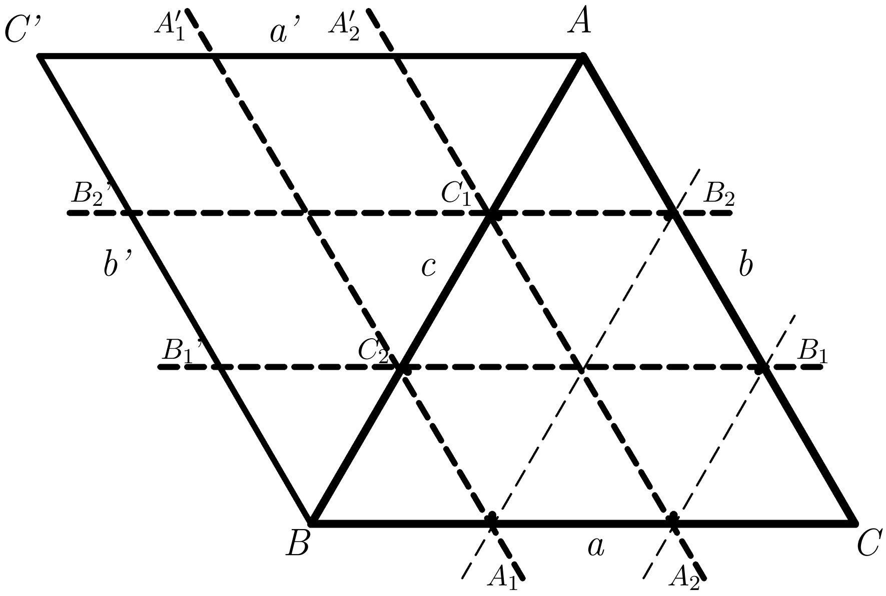


Figure 1.3: Parallelogram split into $n^2 = 9$ smaller parallelograms by a pair of 2 equidistant parallels

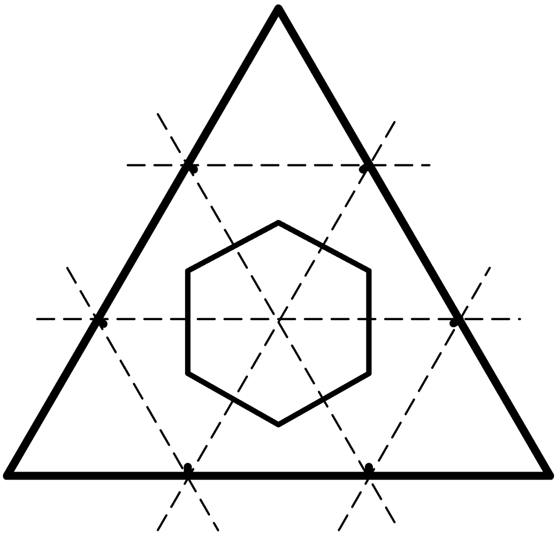


Figure 1.4: Dual hexagon to a triangle grid

Resolution of the ICON grid. There are various possibilities to describe the resolution of a grid. In the case of a grid of equal squares with side length a, the square centres form a grid with a minimum distance of neighbouring points being the length of one side a of the squares. We tend to say that the resolution Δ_{square} of this grid is $\Delta_{\text{square}} := a$. However, there are neighbouring points connected over the diagonal having a distance $\sqrt{2}a$. In a grid of equal squares, each square has 8 neighbours 4 of which are connected over sides, 4 over vertices. One could also define $\Delta'_{\text{square}} = (4a + 4\sqrt{2}a)/8 = \frac{1+\sqrt{2}}{2}a \approx 1.21a$ as resolution of the square grid. In the case of the ICON grid, there are many possibilities to define a resolution. We will present four different definitions here.

(i) One of the easiest definitions is to calculate the average triangle area A from the surface of the earth divided by the number of triangles of the grid. Then, we imagine an associated grid of squares of this area and adopt Δ_{square} of the associated grid as the resolution of the triangular grid. This results for grid $\mathrm{r}_n\mathrm{b}_m$ in:

$$\Delta_{\mathrm{r}_n\mathrm{b}_m} := \sqrt{A} = \sqrt{4R^2_{\text{earth}}\pi / \underbrace{N_{\mathrm{r}_n\mathrm{b}_m}}_{=20n^2 4^m}} = R_{\text{earth}}\sqrt{\frac{4\pi}{4 \times 5n^2 2^{2m}}} = \frac{R_{\text{earth}}}{n2^m}\sqrt{\frac{\pi}{5}}, \tag{1.2}$$

where $R_{\text{earth}} \approx 6371\,\text{km}$ is the radius of the earth. For the grid $\mathrm{r}_2\mathrm{b}_4$ with $N_{\mathrm{r}_2\mathrm{b}_4} = 20480$, we get a resolution of $\Delta_{\mathrm{r}_2\mathrm{b}_4} \approx 160\,\text{km}$. The advantage of this definition is that the T63 echam grid having 18432 grid points has a similar distance between grid points at the equator. However, the distance between grid points of a longitude–latitude grid becomes much smaller in the zonal direction near the poles.

(ii) The length a of the sides of an average equilateral triangle could also be used. Given the height h, the area A of an equilateral triangle with side a would be $A = ah/2$; we also know that $a^2/4 + h^2 = a^2$ due to the Pythagorean theoreme. This leads to $h = \sqrt{3}a/2$ and $A = \sqrt{3}a^2/4$. From this and with (i), we get:

$$\Delta'_{\mathrm{r}_n\mathrm{b}_m} := a = 2\sqrt{A/\sqrt{3}} = 2\Delta_{\mathrm{r}_n\mathrm{b}_m}/\sqrt{\sqrt{3}} = \frac{R_{\text{earth}}}{n2^{m-1}}\sqrt{\frac{\pi}{5\sqrt{3}}} \tag{1.3}$$

The vorticity is given at the triangle vertices, thus $\Delta'_{\mathrm{r}_n\mathrm{b}_m}$ gives the shortest distance between points where the vorticity is given.

(iii) A third possibility would be the shortest distance between the mid points of two triangle sides of an average surface equilateral triangle. In the resolution $\mathrm{r}_n\mathrm{b}_m$, this is exactly the length of a triangle side of a grid $\mathrm{r}_n\mathrm{b}_{m+1}$, thus we get:

$$\Delta''_{\mathrm{r}_n\mathrm{b}_m} := \Delta'_{\mathrm{r}_n\mathrm{b}_{m+1}} = \frac{R_{\text{earth}}}{n2^m}\sqrt{\frac{\pi}{5\sqrt{3}}} = \frac{1}{2}\Delta'_{\mathrm{r}_n\mathrm{b}_m} \tag{1.4}$$

The winds in the dynamics part are given at the midpoints of the triangle sides (perpendicular to the triangle sides).

(iv) As a last possibility for a definition of the resolution of the ICON grid, we would like to use the length of the side of one hexagon of the dual hexagonal grid assuming that all triangles have the same size:

$$\Delta'''_{\mathrm{r}_n\mathrm{b}_m} := \frac{R_{\text{earth}}}{n2^{m-1}}\sqrt{\frac{\pi}{15\sqrt{3}}} \tag{1.5}$$

Note that $\Delta'''_{\mathrm{r}_n\mathrm{b}_m} = \frac{2}{\sqrt{3\sqrt{3}}}\Delta_{\mathrm{r}_n\mathrm{b}_m} \approx 0.877 \times \Delta_{\mathrm{r}_n\mathrm{b}_m}$. All "physics" variables describing parametrized processes are given on the centres of the triangles.

Proof: See exercise.

In general, at each triangle vertex, six triangles meet except at the vertices of the original icosahedron where only five triangles meet. This means that the angle between the triangle sides of the spherical triangles at the original icosahedron vertices are larger than at vertices where 6 triangles meet. By the "spring relaxation procedure" this deficiency can be compensated to some degree. Note further that the sum of the angles in a spherical triangle is generally larger than 360° (this is a general property of spherical triangles).

Most of the variables are stored at the centre of the triangular grid cells, like temperature, specific humidity, tracer mass mixing ratios, pressure, geopotential, meridional and zonal velocity or the vertical velocity. We could also say that these variables are stored at the vertices of the dual hexagonal grid. However, there are variables that have to be stored on other locations according to the used discretization scheme in the dynamical core. In the dynamic part, we need to store the horizontal wind normal to the triangle egdes at the midpoints of these edges. The relative vorticity is stored at the vertices of the triangles, i.e. at the centres of the hexagons of the dual grid. Sometimes, it is said that we work on a staggered C–grid.

ICON is a model that allows for regional refinements of the grid. If we expect more structure in a certain region of the globe or we would like to have more detailed information there, we can use more triangulations in this region. The code structure of ICON is such that it allows for nested grids. So, each grid is associated with a certain region that can also be the globe and we speak of a "model domain" for each grid associated with a certain region.

1.3.2 Vertical grid

In a nonhydrostatic model, a vertical co–ordinate in terms of pressure does not make sense since it can not be taken for granted that the pressure is strictly decreasing with increasing altitude. In general, the air mass of an air column above a certain point can not be calculated from the pressure at that point anymore. Instead, a geometric altitude grid has to be used. In ICON the choice is an altitude co–ordinate system that follows the terrain and consequently, the top and bottom triangle faces are inclined with respect to the tangent plane on a sphere. Still, the top and bottom triangle faces are held parallel to each other. The exact altitude of each grid box depends on the geographical position on the globe. The top and bottom faces are called "half levels" of the vertical grid, the centre of the box is said to be at the "full level" of the vertical grid. Many variables are given at both, half and full levels. In particular radiation fluxes are given at half levels only.

With n layers, there are $n+1$ so–called half levels. The half levels $l, l+1$ enclose layers $[l, l+1[$ at the centres of which are the corresponding full levels l, for $l = 1, \ldots, n$. Layer 1 is at the top of the atmosphere and layer n at the bottom of the atmosphere. Half level $n+1$ is identical with the surface of the earth. In contrast to a pressure co–ordinate system that may start at pressure $p = 0$ and thus encompass the whole atmosphere, the z–grid does not contain the whole atmosphere.

The vertical levels are determined according to an analytical formula already used in the COSMO model. Let $\Delta z_{\mathrm{min}} > 0$ be the minimum thickness of the layers, $z_{\mathrm{max}} > \Delta z_{\mathrm{min}}$ be the altitude of half level 1, i.e. the model top and $\sigma > 1$ be a stretch factor. Then, the altitude of the half levels $z_l^{(\mathrm{h})}, l = 1, \ldots, n+1$ is defined in the following way:

$$
\begin{aligned}
\alpha &:= \ln(\frac{\Delta z_{\min}}{z_{\max}})/\ln(\frac{2}{\pi}\arccos((\frac{n-1}{n})^{\sigma})) \\
z_l^{(\mathrm{h})} &:= z_{\max}\left(\frac{2}{\pi}\arccos((\frac{l-1}{n})^{s})\right)^{\alpha}
\end{aligned}
\tag{1.6}
$$

The layer thickness can be limited with a certain algorithm in the higher troposphere or in the stratosphere. However, Eq. (1.6) does not yet provide a Smooth LEvel VErtical co–ordinate (SLEVE co–ordinate [2]) indexco–ordinate!smooth level vertical since $z_{n+1}^{(\mathrm{h})} = 0$, i.e. it is placed at 0 m altitude whether there is a mountain or not. In order to take the topography into account, the topography is first split into a "large–scale topography" and a "small–scale topography" $(\lambda, \phi) \mapsto h_{1,2}(\lambda, \phi)$, respectively, where (λ, ϕ) is the position on the globe. Then, decay functions

$$
d_i(z) := \frac{\sinh[(z_{\max}/s_i)^{\beta} - (z/s_i)^{\beta}]}{\sinh[(z_{\max}/s_i)^{\beta}]}, \quad \text{for} \quad i = 1, 2 \tag{1.7}
$$

with decay constants $s_{1,2} > 0$ and a decay exponent $\beta > 0$ are defined. The smooth terrain following co–ordinates $z^{(\mathrm{s})}$ are then:

$$
z_l^{(\mathrm{s})}(\lambda, \phi) := z_l^{(\mathrm{h})} + \sum_{i=1}^{2} h_i(\lambda, \phi) d_i(z_l^{(\mathrm{h})}), \quad \text{for} \quad l = 1, \ldots, n+1 \tag{1.8}
$$

In this course, we will work with the climate physics part of ICON that is very similar to the physics of ECHAM. A hydrostatic pressure is used in many of these equations mainly to calculate the air mass in a grid box. In order to provide a hydrostatic pressure, the hydrostatic equation is solved at each grid point and this pressure is then passed to the ECHAM physics.

1.4 Performing computer experiments with ICON

We will learn how to perform a computer experiment (simulation) with ICON in this section. Each computer experiment performing a longer simulation demands good planning in terms of the model settings and input data but also concerning output and postprocessing of output files. We have three phases: (*i*) Preparation of input data and namelist files with all the input parameters of ICON, (*ii*) performing the simulation and "baby sitting" the computer runs, and in a last phase, (*iii*) postprocessing and analysis of the results. These three phases are reflected in the following sections. All example run scripts are designed such that any ICON simulation stores the output files in the `experiments` subdirectory of the ICON model. We will not change this here. This means that your model code has to be installed on a disk with enough disk space to accomodate all output. The `work` disk of `mistral` is large enough, but only temporary storage.

1.4.1 Preparation of a computer experiment with ICON

Each simulation is started by a shell script, mostly called “run script” that has to be prepared for every computer experiment individually. These run scripts contain the links to all input–data files, the namelist settings, and the commands to execute the binary model code. The latter are common to all run scripts. In order to avoid the duplication of code, the execution commands are stored in a “basic run file” `~icon/run/exec.iconrun` where `~icon` means the icon base directory, e.g. `icon-aes`. The actual namelists and input–data files are stored in “experiment files” `~icon/run/exp.<exp_name>` according to the various experiments `<exp_name>`. The experiments that are interesting for us are `<exp_name>=atm_amip_test`, `<exp_name>=atm_amip`, `<exp_name>=atm_rce_test`, and `<exp_name>=atm_ape_test`. The basic run file and the experiment file have to be combined in order to get the corresponding run script. This is done by the following command that has to be executed in the icon base directory `~icon`:

Listing 1.2: Generation of run scripts from basic run file and experiment file

```
make_runscripts <exp_name> ↩
```

Note that you have to pass the experiment name only. Consequently, the name of each experiment file has to start with `exp.` since this is put in front of the experiment name by the `make_runscripts` script automatically. The command in Listing 1.2 creates the run script `~icon/run/exp.<exp_name>.run` . Note that this file is stored in the `run` subdirectory of `~/icon`. This is the script that has to be submitted to the queue of a (super)computer or can be executed on your workstation. On mistral, the queueing system slurm is installed. The following commands in Listing 1.3 are the most important ones for this course (`<acct>` is the account of which the computer time will be used) :

Listing 1.3: Basic SLURM commands to submit jobs

```
sbatch -A <acct> <script> # submits script to queue, accounted on
  project account <acct>
squeue -u <user> # shows status of all your jobs and <job_id>
scancel <job_id> # cancels the job with <job_id>
```

Once you created your run script, you can submit it to the queue by the `sbatch` command. We will discuss the namelists and input–data files next.

1.4.2 Namelists for ICON

Since ICON comes in different flavors and consists of a family of models that can all be coupled, there are a lot of variables determining the exact model configuration. The input of these variables are organized in various namelists that allow the user to pass a variable to the program specifying its name and value. The most important namelists are listed in Tab. 1.1.

A computer experiment with any model configuration may be a simulation over a long time period with a certain start date of the experiment and a certain end date of the experiment. These start and end dates of the experiment are characteristic of this particular experiment, e.g. for the AMIP period lasting from January 1979 to December 2008. Such a simulation could take a lot of computer time, even so much that it is not allowed to run the entire simulation by the submission of one single job even on a supercomputer. This means that this simulation has to be split up into several “chunks” consisting of shorter time periods. When the simulation over one time period is finished, a new job has to be started that continues the simulation seamlessly. In that case we say that we have to restart the model at a certain date and time. The simulation of time periods and the restarts making up a whole computer experiment have to be performed for each model component equally since all components have to be synchronized. In the file `icon_master.namelist`, there are all relevant namelists for the overall time control and information which model components are used in the particular computer experiment. All information specific to each model component, is organized in namelists contained in separate files. The names of these files can be defined in namelists of `icon_master.namelist`. However, there are "conventions" for the naming of these files as given in Tab. 1.1.

Table 1.1: Most important namelists for the atmospheric part of ICON

Namelist name	Purpose
Namelists in file `icon_master.namelist`	
`master_nml`	restart information and path to directory with input and output files
`master_model_nml`	model specific information about namelist files, parallelization
`master_time_control_nml`	calendar information, start and stop date, restarts
`time_nml`	deprecated
`jsb_control_nml`	mode of JSBACH surface/land model
`jsb_model_nml`	configuration and namelist file of JSBACH
Further namelists for other models like ocean	
Namelists in file `NAMELIST_<exp_name>_atm` describing the configuration of the atmosphere	
`run_nml`	configuration about time integration, model configuration in terms of presence of processes and tracers.
`extpar_nml`	external parameters
`initicon_nml`	mode of initialization
`grid_nml`	grid information
`sleve_nml`	vertical co–ordinate information
`nonhydrostatic_nml`	parameters for the nonhydrostatic dynamic core
`parallel_nml`	settings for parallel computing and vectorization
`transport_nml`	advective transport of tracers
`echam_rad_nml`	radiation: information about the atmosphere composition and the orbit included
`echam_phy_nml`	ECHAM physics parameterizations
`echam_cnv_nml`	ECHAM convection
`echam_cld_nml`	ECHAM cloud cover parameters
`echam_gwd_nml`	gravity wave parameterization according to Hines
`echam_vdf_nml`	vertical diffusion parameterization
`interpol_nml`	Interpolation for reconstructing variables on the grid. In particular, radial basis functions (RBF) are used.
`output_nml`	namelist specifying output
Namelists in file `NAMELIST_<exp_name>_lnd` describing the configuration of the land	
`jsb_model_nml`	overall configuration of land model JSBACH
`jsb_*_nml`	describe the configuration of JSBACH4 in more detail.

Preferably, all the namelist variables should be set in the scripts ∼`icon/run/exp.<exp_name>`

and `~icon/run/exec.iconrun`. Do not set the namelist variables inside `~icon/run/exp.<exp_name>.run` because these files are overwritten as soon as someone executes `make_runscripts` for this specific experiment by the setting of `~icon/run/exp.<exp_name>`. However, it may sometimes be hard to find or identify the namelist variables in the respective files. We go through the namelists one by one and discuss the most important variables in these respective namelists:

Namelists in icon_master.namelist

Namelist master_nml. There are only two variables in this namelist.

`lrestart`: Logical that is `.true.` if this job is a restart and continues another simulation, `.false.` if it is the initial period of a longer simulation.

`model_base_dir`: You can give a path here that may be used in other namelists by the placeholder `<path>` when a file name has to be given. E.g., you decide to write output files into a certain special directory `<my_output>`, set `model_base_dir=''<my_output>''` here and set the filename `<fname>` of the outputfiles to `output_filename=''<path>/<fname>''`. The placeholder `<path>` has to be used as written here including parenthesis.

`read_restart_namelist`: Logical that is `.true.` if the namelists are read from a file written by a previous run, `.false.` if the namelists are read from the usual namelist files generated by the run script itself. Default: `.true.`

Namelist master_model_nml. This namelist has to be given for each model component like atmosphere and ocean with the respective entries separately. The most important variables are:

`model_type`: An integer number describing the model, either 1 for the atmosphere or 2 for the ocean. Default: -1.

`model_name`: Name of the model component. Can be chosen by the user, but by convention it is either `''atmo''` for the atmosphere or `''ocean''` for the ocean model.

`model_namelist_filename`: Name of the namelist file containing the namelists describing the model `model_name`. These namelist files have to be provided in the directory ∼`icon/experiments/<exp_name>` if you do not use the `model_base_dir` variable of `master_nml`.

`model_min_rank`: An integer number that is the index of the first MPI thread simulating this model component, e.g. 0. Default: 0.

`model_max_rank`: An integer number that is the index of the last MPI thread simulating this model component, e.g. 4. Default: 65535.

`model_inc_rank`: An integer increment that describes which MPI threads are simulating this model component starting at `model_min_rank`. E.g. `model_inc_rank=2`, `model_min_rank=0`, and `model_max_rank=4` would mean that this model component is simulated by MPI threads 0, 2, and 4. Setting for another model component `model_inc_rank=2`, `model_min_rank=1`, and `model_max_rank=5`, would mean that this other component will be simulated by threads 1,3,5. Default: 1.

Namelist master_time_control_nml. This namelist contains the time information of the entire experiment. If a restart is performed, the restart date is taken from the restart file and is not inserted in this namelist. There are in principle two different types of time variables: Variables describing a certain instant and variables describing time intervals for periodic actions like writing output. The format for an instant is `<YYYY>-<MM>-<DD>T<hh>:<mm>:<ss>Z` where `<YYYY>` is the year in four digits, `<MM>` the month in two digits, `<DD>` the day in two digits, `<hh>` the hour in two digits, `<mm>` the minute in two digits, and `<ss>` the second in two digits. This format will be called DT–format in the sequel. An interval is described by `P<Y>Y<M>M<D>T<h>H<m>M<s>S`, where `<Y>` are the years followed by the unit `Y`, `<M>` are the months followed by the unit `M`, `<D>`

are the days followed by the unit `D`, `<h>` are the hours followed by the unit `H`, `<m>` are the minutes followed by the unit `M`, `<s>` are the seconds followed by the unit `S`. Note that the letter T makes the program understand that the following numbers and units refer to hour, minute, and second. If one of the quantities is zero, it can be omitted including its unit. An eight minute interval can be writte as `PT8M` instead of `P0Y0M0DT0H8M0S`. Furthermore, 60 seconds and minutes, and 24 hours should not be used but be expressed in the next larger unit. This restriction is lifted in more recent versions. This interval format will be called TI–format in the sequel. The most important variables of `master_time_control_nml` are:

`calendar`: String that describes the caledar type that is used. Possible strings are `'proleptic gregorian'` (default) for the proleptic Gregorian calendar, i.e. that the Gregorian calendar is extended backward to dates before the 15th of October 1582, `'year of 365 days'` for a calendar without leap years, and `'year of 360 days'` for a year with 12 months of 30 days each.

`experimentStartDate`: Variable of DT–format meaning the start date of the experiment (not the start date of the actual chunk of an experiment). Default: empty string.

`experimentStopDate`: Variable of DT–format meaning the end date of the experiment (not the end date of the actual chunk of an experiment). Default: empty string.

`checkpointTimeIntval`: Variable of TI–format meaning the interval at which restart files are written. The simulation is not interrupted though. Default: empty string.

`restartTimeIntval`: Variable of TI–format meaning that a restart file is written and the simulation interrupted. Default: empty string.

Namelist jsb_control_nml. The land model JSBACH plays a special role in ICON since it is not a separate model like the ocean model but an integral part of the atmosphere model. On the other hand, simulations with JSBACH alone can be performed.

`is_standalone`: Logical that is `.true.` if JSBACH is used without an explicit simulation of the atmosphere. Default: `.true.`

`restart_jsbach`: Indicates restart of JSBACH. Important for stand alone version. Default: `.false.`.

Namelist jsb_model_nml. The JSBACH land model comes in different degrees of complexity of the underlying processes. These can be described here, but the corresponding parameters have to be defined in the specific JSBACH namelists.

`model_name, model_shortname, model_description`: strings of 30, 10, and 132 characters describing the actual JSBACH configuration. Default: empty string.

`model_namelist_filename`: filename of the file containing all JSBACH specific namelists. Default: empty string.

Namelists in NAMELIST_<exp_name>_atm

Namelist run_nml. **`modelTimeStep`:** Variable of TI–format giving the model time step. Default: Emtpy string.

`num_lev`: Array of integers giving the number of model layers (full levels) for each “model domain”, i.e. for the global grid and possible local refinements. Default: 31 for each domain with a maximum of 10 domains.

`ltestcase`**:** Logical that is `.true.` if a special test case like the aqua planet or the radiative convective equilibrium has to be simulated. Default: `.true.`

`ldynamics`**:** Logical switching on (`.true.`) or off (`.false.`) the dynamical core (tendencies by adiabiatic processes). Default: `.true.`

`ltransport`**:** Logical that switches on (`.true.`) or off (`.false.`) the large scale transport of tracers (if there are any) by the dynamical core. Default: `.false.`

`iforcing`**:** Integer that describes the package according to which parameterized processes are calculated ("physics" of the model).

`iforcing`	meaning
0	no forcing
1	Held–Suarez forcing
2	ECHAM forcing/physics
3	NWP forcing
4	local diabatic forcing without "physics"
5	local diabatic forcing with "physics"

Default: 0

`ntracer`**:** Integer, number of tracers, default: 0.

`restart_filename`**:** String that specifies restart filename. You can integrate the restart date and time into the name by using the placeholder `<rsttime>`, e.g. `'restart_run1_<rsttime>.nc'`. Default: `'<gridfile>_restart_<mtype>_<rsttime>.nc'`.

`output`**:** Array of strings describing output mode for each model domain (refinement of the grid). Use `'none'` for no output at all, `'nml'` for output specified by output namelists, and `'totint'` for the output of total integrals only. Default: `'default'` for the global domain, empty strings for all other domains.

Namelist extpar_nml. This namelist is a bit an oddment since there are a lot of external parameters like aerosol data or ozone concentrations or boundary condition files like sea surface temperatures that have to be read by the model. But these are not handled here. Only two namelist variables are interesting for us:

`itopo`**:** Integer that defines whether the topography is specified by analytical functions (`itopo=1`) or read from a netcdf file `itopo=2`). Default: 0.

`extpar_filename`**:** String that gives the filename of the topography. Default: `'<path>extpar_<gridfile>`.

Namelist initicon_nml. Here, only two variables describing the initialization of ICON is important for us:

`init_mode`**:** Integer number between 1 and 7 describing the initialization mode. Default: 2 (IFS analysis data).

`ifs2icon_filename`**:** Path and name of the file containing IFS analysis data used as initial condition. Default: `<path>ifs2icon_R<nroot>B<jlev>_DOM<idom>.nc`, where `R<nroot>B<jlev>` will be replaced by the actual model resolution.

Namelist grid_nml. This namelist contains variables describing the triangular grid. In particular, the rotation of the grid can be defined and whether or not there are nested domains. The following variables are the most important ones:

`grid_angular_velocity`: Real variable describing the angular velocity of the grid, i.e. the rotation of the earth. Given in radiant per second. Default: `7.29212e-5`.

`dynamics_grid_filename`: (Array of) strings giving the filenames for the grids of each (nested) domain in arbitrary order. Default: empty string.

`dynamics_parent_grid_id`: Array of integers giving the position of the parent grid in the array `dynamics_grid_filename` by position. If the `n`th entry in this vector is `m`, this means that the `m`th grid listed in `dynamics_grid_filename` is the parent of the `n`th grid listed in `dynamics_grid_filename`. A value of "0" at the `n`th position means that the `n`th grid listed in `dynamics_grid_filename` is the global domain. Default: `0,1,2,...`.

`create_vgrid`: Logical that is `.true.` for writing the vertical grid into a file, `.false.` otherwise. Default: `.false.`

Namelist sleve_nml. This namelist defines the vertical grid as it is used in the dynamical core. The echam physics does not use this vertical grid information directly, but relies on the corresponding pressures and altitudes in kilometres.

`top_height`: Real variable giving the "model top" in metres above sea level, i.e. the height of half level 1 that is the upper limit of the highest model layer. Default: 23500.0.

`min_lay_thckn`: Real variable giving the geometric height of the lowest layer in the atmosphere in metres. If a value equal or smaller 0.01 is given, all layers will be chosen to be of the same thickness. This variable corresponds to Δz_{min} of Eq. (1.6). Default: 50.

`stretch_fac`: Real variable giving s of Eq. (1.6). Default: 1.

`decay_scale_{1,2}`: Real numbers giving the decay constants $s_{1,2}$ of Eq. (1.7). Default: 4000.; 2500.

`decay_exp`: Real number giving the decay exponent β of Eq. (1.7). Default: 1.2.

`flat_height`: Above this altitude in metres, the layers do not depend on the underlying topography anymore; real number, default: 16000.

Namelist nonhydrostatic_nml. `ndyn_substeps`: The dynamic core uses time steps that may be smaller than the physics part of the model. Divides the time step into `ndyn_substeps` integer parts. Default: 5.

`damp_height`: Above this height in metres, w–damping and the "sponge layer" is applied. It is a vector that may contain different heights for each model domain. Default: $(45000., -1., -1., \dots)$.

`rayleigh_coeff`: Rayleigh coefficient for damping in "sponge layer". Default: $(0.05, -1., -1., \dots)$.

`vwind_offctr`: The vertical wind is not calculated at the cell centre, but "off–centred" in order to stabilize the numeric procedure. Real number, default: 0.15.

`divdamp_fac`: The divergence of the wind field can be damped by this factor (applied in every dynamics substep). Default 0.0025.

Namelist parallel_nml. In terms of model parallelization, the model domain, e.g. the whole globe, is first split into various regions. To each region a processor (or thread) is assigned to perform the respective computations. On each processor, we imagine the grid–cell centres of this respective region to be stored in one long vector. The order of the grid–cell centres does not play any role for us. In fact, this vector may be too long to be effectively treated by a certain machine architecture and considerably slow down the ICON program if treated as such. In order to obtain shorter vectors, we split this long vector into several chunks of moderate length `nproma` of our choice. We can choose `nproma` freely, it is not required that it is a divisor of the number of grid cells on the processor. However, filling chunks with `nproma` grid cells successively may result in one shorter chunk at the end. These chunks are called blocks and arranged into a two–dimensional array `a(1:nbdim,nblks_c)`, where `nblks_c` is the number of the blocks and `nbdim` the maximum length of the blocks given by `nproma`. The first `nblks_c-1` blocks have values set for the full length `nproma` whereas the last one may be shorter. As we learned in Sec. 1.3.1, some of the variables are not stored at the cell centres but at the centers of the triangle edges or the vertices. In these latter two cases, the corresponding arrays are any `b(1:nbdim,nblks_e)` and `c(1:nbdim,nblks_v)`, respectively. The length of the last block is `npromz_c` for the centres, `npromz_e` for the edge centres, and `npromz_v` for the vertices, respectively.

`nproma`: Integer describing the maximum length of a block. Default: 1.

Namelist transport_nml. It is possible to define the exact numeric procedure for the tracer transport for each tracer individually. Theoretically, 44 different settings for the numeric procedure are possible. In addition, several flux limiters can be chosen. A flux limiter is set to limit the total variation of the solution in order to reduce artificial “wiggles”. To this end, the flux limiter limits the fluxes to “reasonable values”. We cannot explain all possibilities here, so we restrict ourselves to the most relevant settings. In general, the mass mixing ratio is transported. If transport is switched off, the mass mixing ratio is kept constant (not the local tracer mass).

`ihadv_tracer`: Integer vector of `ntracer` elements for each tracer describing the horizontal advection.

`ihadv_tracer`	meaning
0	no horizontal advection.
2	“Miura” scheme meaning second order with linear reconstruction.
52	mixture of “Miura” method and flux form semi–Lagrangian transport (FFSL transport).

Default: 2.

`itype_hlimit`: Integer vector of `ntracer` elements defining a flux limiter for the horizontal advection of each tracer.

`itype_hlimit`	meaning
0	no flux limiter
3	monotonous flux limiter
4	positive definite flux limiter

Default: 4.

`ivadv_tracer`: Integer vector of `ntracer` elements for each tracer describing the vertical advection.

`ivadv_tracer`	meaning
0	no vertical advection. Note that tracer mass is conserved in each grid box, not the mass mixing ratio. This is different to the horizontal advection.
3	piecewise parabolic method (ppm), works for Courant–Friedrichs–Lewy–numbers CFL > 1.

Default: 3.

`itype_vlimit`: Integer vector of `ntracer` elements defining a flux limiter for the vertical advection of each tracer.

`itype_hlimit`	meaning
0	no flux limiter
1	semi–monotone slope limiter

Default: 1.

`lvadv_tracer`: Logical that switches on (`.true.`) or off (`.false.`) the vertical tracer advection in general. Default: `.TRUE.`

Namelist echam_rad_nml. There are two radiation schemes in ICON: The older RRTM scheme and the newer PSRAD scheme. The RRTM scheme does not work together with the ECHAM physics. The `echam_rad_nml` namelist contains all variables relevant to the PSRAD radiation scheme. All variables are combined as components into one variable `echam_rad_config(1:max_dom)` of "derived type" `t_echam_rad_config`, where `max_dom` is the allowed maximum number of domains (10). For the type definition, see `mo_echam_rad_config.f90`. For each domain, the components of `echam_rad_config` have to be given separately. The following components are the most important ones:

`echam_rad_config(:)%irad_<spec>`: Integer variables that describe how the respective gas concentrations are set for the radiative transfer calculation. `<spec>` is one of `h2o`, `co2`, `ch4`, `n2o`, `o3`, `o2`, `cfc11`, `cfc12`.

`irad_<spec>`	meaning
0	the volume mixing ratio of `<spec>` is assumed to be 0
1	the volume mixing ratio of `<spec>` is taken from an interactive tracer
> 1	various profiles eventually transient in time can be chosen.
2	vertically and horizontally constant greenhouse gas concentration
4	greenhouse gas scenario
8	greenhouse gas is read from a file as 3d–field

Default: H_2O: 1, CO_2, O_2, CFC11, CFC12: 2, CH_4, N_2O: 3, O_3: 0.

`echam_rad_config(:)%vmr_<spec>`: Real variable giving the volume mixing ratios $x_{\texttt{<spec>}}$ of respective species `<spec>` if $\texttt{irad\_} < \texttt{spec} > = 2$. Default: $x_{\mathrm{CO_2}} = 348 \times 10^{-6}$, $x_{\mathrm{CH_4}} = 1650 \times 10^{-9}$, $x_{\mathrm{N_2O}} = 306 \times 10^{-9}$, $x_{\mathrm{O_2}} = 0.20946$, $x_{\mathrm{CFC11}} = 214.5 \times 10^{-12}$, $x_{\mathrm{CFC12}} = 371 \times 10^{-12}$, no default value for O_3.

`echam_rad_config(:)%frad_<spec>`: scaling factor by which the volume mixing ratio is multiplied. Default: 1.0

`echam_rad_config(:)%irad_aero`: integer number describing the aerosol mode. Many settings are possible, only "0" (meaning no aerosols) and the modes `irad_aero` > 10 work with the PSRAD radiation. `irad_aero` $= 18$: optical properties of anthropogenic aerosols are given as "simple plumes", i.e. as parametrized functions of location in the atmosphere, wave length and time. The natural background is read from files as also the optical properties of stratospheric aerosols. Default: 2.

`echam_rad_config(:)%ighg`: integer number choosig a certain greenhouse gase scenario. Default: 0 (no greenhouse gas scenario).

`echam_rad_config(:)%ldiur`: logical that switches on (`.true.`) or off (`.false.`) the diurnal cycle. Default: `.true.`

`echam_rad_config(:)%lyr_perp`: Logical to switch on (`.true.`) or off (`.false.`) the perpetual repetition of the orbit of one single year. Default: `.false.`

`echam_rad_config(:)%yr_perp`: If the orbit of a specific year has to be used in perpetual repetition, this integer gives the specific year. Default: -99999.

`echam_rad_config(:)%isolrad`: This integer gives the mode of the solar irradiation, in particular the choice of the "solar constant".

`isolrad`	meaning
0	the standard RRTM scheme solar irradiation is used.
1	transient solar irradiation "as measured" e.g. by satellites
2	pre–industrial solar irradiation
5	globally symmetric solar irradiation constant in time corresponding to an energy flux into the atmosphere like for pre–industrial solar irradiation

Default: 0.

`echam_rad_config(:)%fsolrad`: scaling factor for solar irradiation. Default: 1.0

`echam_rad_config(:)%cecc`: Real number describing the eccentricity of the orbit if a Kepler orbit is used. Default: 0.016715

`echam_rad_config(:)%cobld`: Real number describing the obliquity of the earth axis versus the plane of the orbit. Default: 23.44100

`echam_rad_config(:)%l_orbvsop87`: Logical that switches on (`.true.`) or off (`.false.`) the use of the real (observed) orbit that is slightly different from the Kepler orbit. Default: `.true.`

`echam_rad_config(:)%l_sph_symm_irr`: Logical that switches on (`.true.`) or off (`.false.`) the usage of spherically symmetric irradiation of the earth, e.g. for radiative–convective equilibrium experiments. Note that spherically symmetric irradiation needs (*i*) a scaled irradiation (e.g. `isolrad` = 5) and (*ii*) the usage of a Kepler orbit with no eccentricity since otherwise, the irradiation is scaled by a hypothetical distance sun–earth. Default: `.false.`

Namelist echam_phy_nml. Within this namelist, the various parameterized physics processes can be timed to start and end at a certain date and time and their calling frequency can be determined. There is even the option to switch them off completely. The idea behind this individual time control is to optimize efficiency since the various processes may have different individual characteristic time scales. This means that it may be sufficient to calculate some processes only every several time steps and keep their tendencies constant over these time steps adding them to the respective variables. Take the radiative transfer calculation for example. It may be scaled by the changing incoming solar radiation but the composition and temperature in a column is not changing so much that it would justify to calculate the radiative transfer in every time step when performing climate simulations. So, the fluxes are used for e.g. 12 integration time steps in a row.

For each physics process, there is a component of `echam_phy_config(1:max_dom)`, `max_dom` being the allowed maximum number of domains, that is a TI–variable describing the frequency at which this particular process is called. If the string is empty, the corresponding physics process is never called. In order to call the radiation in domain 1 every two hours, set

Listing 1.4: Example for giving an individual frequency to the radiation call

```
echam_phy_config(1)%dt_rad=''PT2H''
```

for example. The time intervals have to be integer multiples of `modelTimeStep`. In `echam_phy_config(:)%<dt_procs>`, the following processes can be triggered by giving the corresponding TI–variables for `<dt_procs>`:

`dt_rad`: TI–variable that gives the radiation time step.

`dt_vdf`: TI–variable that gives the time step for vertical diffusion.

`dt_cnv`: TI–variable that gives the time step for convection.

`dt_cld`: TI–variable that gives the time step for large scale cloud processes.

`dt_gwd`: TI–variable that gives the time step for the gravity wave drag calculation.

`dt_sso`: TI–variable that gives the time step for subgrid scale orographic effects.

`dt_mox`: TI–variable that gives the time step for methane oxidation and water vapour photolysis in the upper atmosphere (stratosphere and higher).

`dt_car`: TI–variable that gives the time step for the linearized interactive ozone model according to Cariolle and Teyssèdre.

`dt_art`: TI–variable that gives the time step for the ART aerosol and chemistry submodel (future).

By default, all TI–variables are empty strings meaning that the corresponding processes are all switched off.

The start and end dates are DT–variables that can be given in the same way as the frequencies above. Instead of `dt` these variables start with `sd` for the start date and `ed` for the end date.

Calculating radiation for only one day on the 1st of January 1979, at a frequency of two hours, would require the following variables in the namelist:

Listing 1.5: Example for giving start and end date and an individual frequency to the radiation call

```
echam_phy_config(1)%dt_rad=''PT2H''
echam_phy_config(1)%sd_rad=''19790101T00:00:00Z''
echam_phy_config(2)%ed_rad=''19790102T00:00:00Z''
```

The processes themselves may have individual namelists that may not be different on the various domains. Furthermore, surface processes can be switched on or off in a similar way. There are logicals as components of `echam_phy_config(jg)` switching on (`.TRUE.`) and off (`.FALSE.`) the corresponding process.

`ljsb`: Switch on (`.TRUE.`) or off (`.FALSE.`) the land surface model JSBACH.

`lamip`: Switch on (`.TRUE.`) or off (`.FALSE.`) the use of the AMIP sea surface temperatures.

`lice`: Switch on (`.TRUE.`) or off (`.FALSE.`) the sea ice temperature calculation.

`llake`: Switch on (`.TRUE.`) or off (`.FALSE.`) the usage of lakes in JSBACH.

`lmlo`: Switch on (`.TRUE.`) or off (`.FALSE.`) the usage of a mixed layer ocean.

Namelist output_nml. This namelist can be repeated for an arbitrary number of output files. If there are several model domains, one output file for each model domain specified in this namelist will be created under the same base name of the output files.

`output_filename`: string describing the base name of the output file. Placeholders for the path can be used, see `model_base_dir` of the `master_nml` namelist. Information about the model domain and leveltype and an extension will be included automatically. Default: empty string.

`filename_format`: string that describes the exact composition of the filename. Default: `<output_filename>_DOM<physdom>_<levtype>_<jfile>`. In this string, `<output_filename>` is a placeholder for the string as defined in `output_filename`, `<physdom>` is a placeholder for the index of the model domain, `<levtype>` is a placeholder for the level type, e.g. model levels or pressure levels, `<jfile>` is a placeholder for the index of the file in the experiment resulting from `file_interval` counted over the whole simulation period. There are other placeholders like `<levtype_l>` also for the level type, and `<datetime>`, `<datetime2>`, `<datetime3>` for the date and time at which this output file starts.

`filetype`: Integer encoding the filetype. 2: GRIB2, 4: netcdf2, 5: netcdf4. Default: 2

`file_interval`: Variable of TI–format describing the time interval at which new output files will be opened. Default: empty string.

`dom`: Vector of integers describing the model domains for which output is desired (index of model domain). Default: -1.

`output_interval`: This string is a TI–variable describing the output interval. Default: empty string.

`output_{start,end}`: These strings are DT–variables describing the start and end date and time of the output. Default: empty strings.

`{ml,pl,hl}_varlist`: Array of variable names that will be written to the output files on model, pressure, or height levels. There is a maximum of 999 model–level variables, but only 150 pressure– or height–level variables are allowed. Which variables are selectable is not easy to know, we will discuss this later. Default: empty strings.

`include_last`: Logical that indicates whether (`.true.`) or not (`.false.`) to include the last time step that has to be written into the output file.

`output_grid`: Logical that indicates whether (`.true.`) or not (`.false.`) the grid information is added to the output file. Default: `.false.`

`remap`: Integer indicating whether an interpolation to a different horizontal grid is desired.

`remap`	meaning
0	no interpolation (output on icosahedral grid)
1	output on a regular longitude–latitude grid

Default: 0

`reg_lon_def`: Array of three real numbers describing the longitudes of a regular longitude–latitude output grid. You have to give the first longitude, an increment, and the last longitude. Instead of an increment, you may give the total number of grid points in longitude direction. See the `reg_def_mode` variable for the distinction between increments and numbers of grid points. Default: none

`reg_lat_def`: Array of three real numbers describing the latitudes of a regular longitude–latitude output grid. You have to give the first latitude, an increment and the last latitude. Instead of an increment, you may give the total number of grid points in latitude direction. See the `reg_def_mode` variable for the distinction between increments and numbers of grid points. Default: none

`reg_def_mode`: This integer tells ICON whether you defined your regular grid giving increments (`reg_def_mode` = 1) or total numbers of grid points (`reg_def_model` = 2) for both longitudes and latitudes. Default: 0

`operation:` string that indicates special operations that have to be done on output, e.g. `'mean'` denotes time average over the output interval. This does not work together with remapping! Default: empty string.

1.4.3 Input data for ICON

There are three different types of input data needed to perform a simulation with ICON: (*i*) Initial conditions or restart data, (*ii*) boundary conditions, and (*iii*) parameter fields describing e.g. the composition of the atmosphere.

Most of the external data are stored somewhere in `/pool/data/ICON/` or in ∼`icon/data`. The former directory undergoes a major revision for the moment and no directory structure can be given here.

Initial conditions

The dynamics of the atmosphere is determined by functions

$$f_t^{(i)} : \begin{cases} \mathbb{S}^2 \times \mathbb{R}_+ & \to \mathbb{R} \\ (\lambda, \phi, z) & \mapsto f_t^{(i)}(\lambda, \phi, z) \end{cases}, \quad \text{for} \quad t \in \mathbb{R}_+ \tag{1.9}$$

which are the solutions of corresponding Navier–Stokes equations on a spherical shell $\mathbb{S}^2 \times \mathbb{R}_+$, i indicating the various dynamic quantities. These functions $f_t^{(i)}, t \in \mathbb{R}_+$ are called "prognostic variables". The initial state of the atmosphere must specify these prognostic variables in the atmosphere at the beginning of the simulation which we say to be at $t = 0$ without any restriction since we can shift the time in a way so that this is true. The prognostic variables of the nonhydrostatic ICON model are: vertical and horizontal winds, the air density, the Exner pressure $\Pi = (p/p_0)^{R_\mathrm{d}/c_\mathrm{p}}$, ($p_0$: reference pressure, R_d: specific gas constant of dry air, c_p: isobaric heat capacity of dry air), the virtual potential temperature $\theta = T(p_0/p)^{R_\mathrm{d}/c_\mathrm{p}}$, the specific humidity q, cloud water x_l, cloud ice x_i, possibly other tracers, and the turbulent kinetic energy k. We may imagine the set of these functions at time $t = 0$ as a point in an abstract phase space. This point will move in the phase space with time providing a trajectory that depends on the initial conditions. Weather is a chaotic system: If we start only at a slightly different point we will see us arbitrarily far from the first trajectory if we only wait long enough. In practice, we can never know the initial conditions very accurately, since there are far too few measurement stations in the atmosphere and every grid that can still be handled is too coarse. However, it is a very difficult question how accurate the knowledge of this initial condition must be to achieve a certain prediction skill.

In ICON, there are various initialization procedures. For some test cases, the initial conditions are set in certain subprograms, all collected in ∼`icon/src/testcases`. For the aqua–planet, initialization is performed by the subroutine `init_nh_state_prog_APE` in file `mo_nh_ape_exp.f90`, for the radiative convective equilibrium experiment (RCE), it is done by `init_nh_state_rce_glb` in file `mo_nh_rce_exp.f90`. For the AMIP experiments, initial files are used instead. The initial files have a naming convention `<descriptor>_R<rr>B<bb>_DOM<dd>.nc` where `<descriptor>` is either `ifs2icon` or `dwdFG`, or `dwdana` for the various data sources, the resolution $\mathrm{r}_n\mathrm{b}_m$ is encoded by two digits $<$ `rr` $> = n$ and two digits $<$ `bb` $> = m$, the domain index is encoded by two digits $<$ `dd` $>$. For the "standard" AMIP–experiment, the initial file for IFS analysis data is therefore `ifs2icon_R2B04_DOM01.nc`. The initial data for JSBACH are stored in a file `ic_land_soil.nc`. In principle, an arbitrary name of the initial file can be given by the variables `<descriptor>_filename` of the namelist `initicon_nml`. You can find various examples for initial files in `/pool/data/ICON/setup/ifs_iconremap_amip`. These have to be linked to the corresponding standard names as described above. The land data are in a complicated path you can find in the run script examples.

Boundary conditions

Boundary conditions give the values of prognostic variables at the boundaries of the atmosphere. Apart from the prognostic variables in the dynamics part, there is the radiation transfer that also needs boundary conditions: The solar irradiation at the top of the atmosphere and the reflectivity at the surface of the earth (bottom of the atmosphere). The boundary conditions can be either constant in time or periodic with a daily or yearly cycle that is repeated each day or year, or "transient" meaning that they are time dependent but without periodicity. The sea surface temperature and sea ice data have to be in one file, `bc_sst.nc` and `bc_sic.nc` for all simulated years, respectively. The solar irradiance has to be stored in a file `bc_solar_irradiance_sw_b14.nc` for all simulated years, if not one of the standard irradiances is used which are constant in time. The orbit influences the solar irradiance since the irradiance will be scaled by the sun–earth distance.

Parameter data sets

As for boundary conditions, these data sets can be constant, periodic or transient in time. There are many parameter sets, depending on the exact configuration of the models. We give a list of parameter files here without being exhaustive.

The implementation of aerosol optical properties, ozone concentrations, and solar irradiation is preliminary and may be erroneous. They will all be revised. The volcanic aerosols of Stenchikov are outdated and should be replaced by the CMIP6 volcanic aerosols.

Table 1.2: Parameter data for ICON

standard filename	description
Data related to the composition of the atmosphere	
`bc_ozone_<yyyy>.nc`	3d–ozone volume mixing ratio for year `<yyyy>` (four digits).
`bc_aeropt_kinne_sw_b14_fin_<yyyy>.nc`	fine mode aerosol optical properties after S. Kinne, solar radiation for year `<yyyy>` (four digits)
`bc_aeropt_kinne_sw_b14_coa_<yyyy>.nc`	coarse mode aerosol optical properties after S. Kinne, solar radiation for year `<yyyy>` (four digits)
`bc_aeropt_kinne_lw_b16_coa_<yyyy>.nc`	coarse mode aerosol optical properties after S. Kinne, thermal radiation for year `<yyyy>` (four digits)
`bc_aeropt_stenchikov_lw_b16_sw_b14_<yyyy>.nc`	stratospheric aerosol optical properties for solar and thermal radiation for year `<yyyy>` (four digits). These volcanic aerosols are outdated, should not be used anymore, and will soon be replaced by CMIP6 volcanic aerosols.
`MACv2.0-SPv1.nc`	optical properties of anthropogenic aerosols in parametrized functions
`bc_greenhouse_gases.nc`	transient greenhouse gas concentrations of greenhouse gases with uniform global distribution
`ECHAM6_CldOptProps.nc`	cloud optical properties
Parameter file for radiation	
`lsdata.nc`	“light sponge”: absorption coefficients and other parameters for the PSRAD radiation.
Land parameter files	
`bc_land_frac.nc`	land fraction at each grid cell
`bc_land_hd.nc`	land hydrological discharge model
`bc_land_phys.nc`	physical properties of land
`bc_land_soil.nc`	soil properties
`bc_land_sso.nc`	subgrid scale orography (statistical parameters for the description of unresolved orography in a grid cell).

Chapter 2

The code of ICON

2.1 Flowchart of ICON

We present a very much simplified flowchart of the atmosphere version of ICON in Fig. 2.1.

We give a short description of the location and the purpose of the various subprograms. We list them in the order as they appear in the flowchart Fig. 2.1. Since all code files are collected in subdirectories of the ∼`icon/src` directory with only very few exceptions, we will abbreviate this by the symbol `SRC` .

`program icon:` (`SRC/drivers/icon.f90`) main program of ICON sets global attributes including a version number, initializes MPI.

`init_master_control:` (`SRC/drivers/mo_master_control.f90`) assigns MPI ranks (processor) to each model part, e.g. atmosphere, ocean.

`read_master_namelist:` (`SRC/namelists/mo_master_nml.f90`) reads `icon_master.namelist`.

`atmo_model:` (`SRC/drivers/mo_atmo_model.f90`) calls all relevant subprograms for the atmosphere. At this point, the ocean model or some special test modes call `ocean_model` and `icon_testbed` instead of `atmo_model`, respectively. In that way, the ocean is well separated from the atmospheric part and the calling sequences in `ocean_model` are rather distinct from the `atmo_model`, although the organization of the code in parts of an initialization phase, time integration loop, cleanup phase is similar.

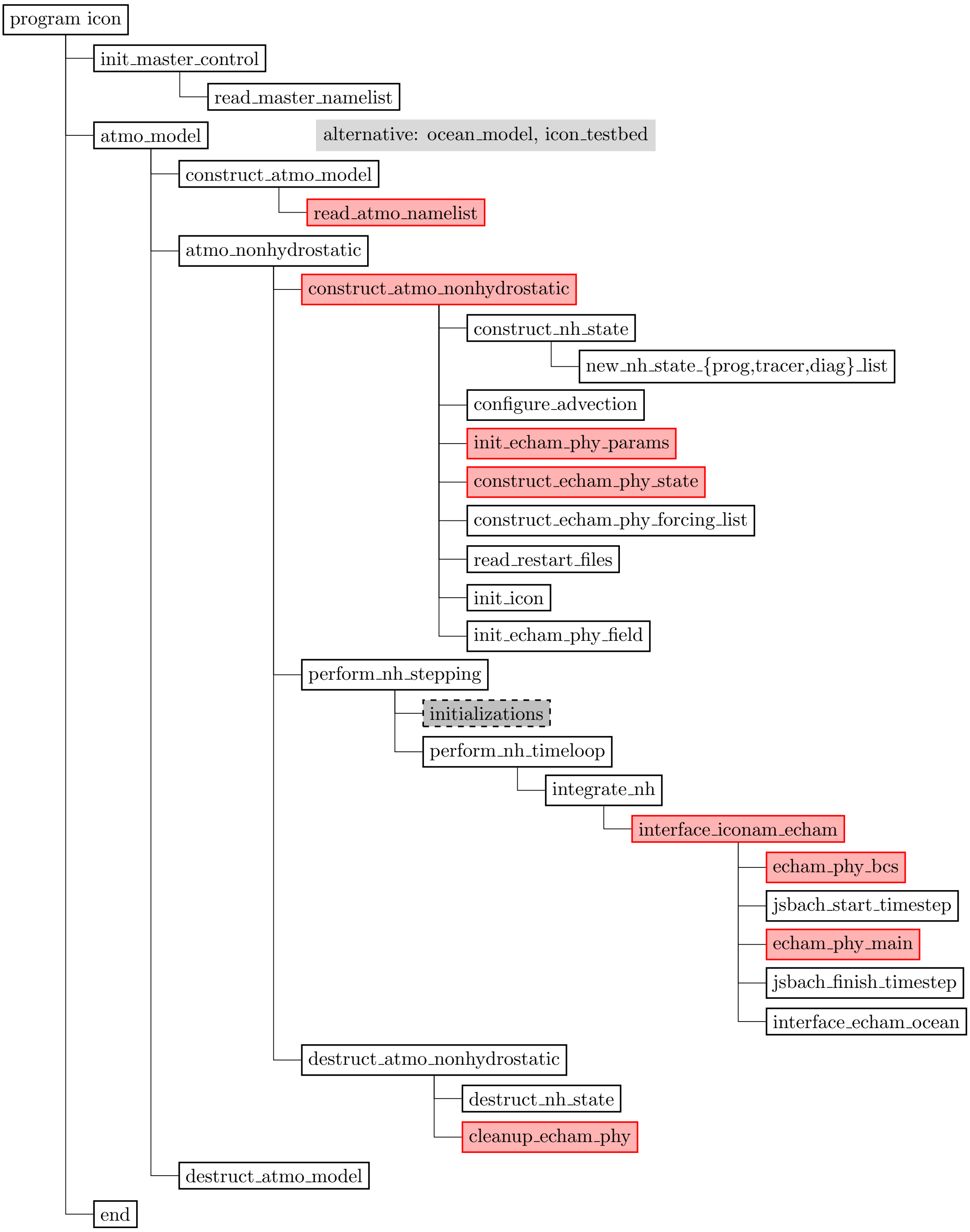
`construct_atmo_model:` (`SRC/drivers/mo_atmo_model.f90`) calls the reading of namelists and all important initializations outside time integration loop.

`read_atmo_namelists:` (`SRC/namelists/mo_read_namelists.f90`) reads namelists concerning the atmospheric model from `NAMELIST_<exp_name>_atm`.

`atmo_nonhydrostatic:` (`SRC/drivers/mo_atmo_nonhydrostatic.f90`) calls all relevant subprograms to initialize (also memory allocation) and simulate a nonhydrostatic atmosphere, and to free memory at the end.

`construct_atmo_nonhydrostatic:` (`SRC/drivers/mo_atmo_nonhydrostatic.f90`) call all subprograms establishing the derived types needed for the nonhydrostatic model and assigning all relevant variables concerning the choice of dynamics equations.

Figure 2.1: Flowchart of ICON



`construct_nh_state:` (`SRC/atm_dyn_iconam/mo_nonhydro_state.f90`) calls subprograms to establish a derived type describing the state of the non–hydrostatic model.

`new_nh_state_{prog,tracer,diag}_list:` (`SRC/atm_dyn_iconam/mo_nonhydro_state.f90`) allocating the derived types containing all prognostic variables, tracer variables (transported quantities), and diagnostic variables.

`configure_advection:` (`configure_model/mo_advection_config.f90`) set all variables relevant for the choice of the numerical method to calculate advection.

`init_echam_phy_params:` (`SRC/atm_phy_echam/mo_echam_phy_init.f90`) Parameters of echam physics are set here. A consistency test of parameter settings is performed. This comprises the setup of convection, diffusion, gravity waves, methane oxidation in the stratosphere, interactive Cariolle ozone, radiation. Data structures are provided by `construct_echam_phy_state`.

`construct_echam_phy_state:` (`SRC/atm_phy_echam/mo_echam_phy_memory.f90`) allocates memory and the derived types for all quantities that are needed in the parameterized physics calculation (equations of echam physics).

`construct_echam_phy_forcing_list:` (`SRC/atm_phy_psrad/mo_psrad_forcing_memory.f90`) Creates new data structure (stream) for radiative fluxes.

`read_restart_files:` (`SRC/io/restart/mo_load_restart.f90`) Reads state of prognostic variables of a previous run.

`init_icon:` (`SRC/atm_dyn_iconam/mo_initicon.f90`) Read initial state of atmosphere.

`init_echam_phy_field:` (`SRC/atm_phy_echam/mo_echam_phy_init.f90`) Set initial conditions for echam physics.

`perform_nh_stepping:` (`SRC/atm_dyn_iconam/mo_nh_stepping.f90`) initialization of time integration loop and call of subroutines performing the time integration.

`perform_nh_timeloop:` (`SRC/atm_dyn_iconam/mo_nh_stepping.f90`) compute some diagnostics and initializations.

`integrate_nh:` (`SRC/atm_dyn_iconam/mo_nh_stepping.f90`) time integration loop with calls of time dependent boundary conditions or parameter sets and the dynamics and physics (parameterized equations) inside; recursive subroutine.

`interface_iconam_echam:` (`SRC/atm_phy_echam/mo_interface_iconam_echam.f90`) Interface routine to the ECHAM physics parameterization and the dynamical core. This subroutine has to call the land surface model and the interaction between ocean and atmosphere.

`echam_phy_bcs:` (`SRC/atm_phy_echam/mo_echam_phy_bcs.f90`) sets the boundary conditions and parameters (composition of the atmosphere) depending on time for the ECHAM physics.

`jsbach_start_timestep:` (`SRC/lnd_phy_jsbach/interfaces/mo_jsb_interface.f90`) interface to the land surface model JSBACH.

`echam_phy_main:` (`SRC/atm_phy_echam/mo_echam_phy_main.f90`) this subroutine corresponds to `physc.f90` in ECHAM and calls radiation, vertical diffusion, large scale cloud processes, and convection by the use of interface subroutines.

`jsbach_finish_timestep:` (`SRC/lnd_phy_jsbach/interfaces/mo_jsb_interface.f90`) interface to the land surface model JSBACH.

`interface_echam_ocean:` (`SRC/atm_phy_echam/mo_interface_echam_ocean.f90`) interface to the ocean model for atmosphere–ocean interactions.

`destruct_atmo_nonhydrostatic:` (`SRC/drivers/mo_atmo_nonhydrostatic.f90`) subroutine for cleanup of memory.

`destruct_nh_state:` (`SRC/atm_dyn_iconam/mo_nonhydro_state.f90`) deallocates memory of derived types for the dynamics of the non–hydrostatic model.

`cleanup_echam_phy:` (`SRC/atm_phy_echam/mo_echam_phy_cleanup.f90`) deallocates memory of derived types used in the ECHAM physics part.

`destruct_atmo_model:` (`SRC/drivers/mo_atmo_model.f90`) deallocation of general memory for the atmospheric model.

2.2 Survey of FORTRAN techniques used in ICON

The code of ICON uses user–defined derived types and modules extensively. It may be good to recapitulate these FORTRAN features in order to better understand the code. In addition, this section offers the opportunity to discuss some of the FORTRAN conventions used in ICON, although we will not be exhaustive in that respect. There is a detailed "style guide" `~icon/doc/style/icon_standard.pdf` [1].

General remark on real variables: They are all typed according to

Listing 2.1: Declaration of real variables

```
USE mo_kind, ONLY: wp
REAL(wp) :: <varlist>
```

The module `SRC/shared/mo_kind.f90` contains all available kinds of variables.

2.2.1 Modules

The ICON code has a main program and many subprograms which are all organized in modules. However, the modules contain much more than only subprograms: All important data types, mostly of derived type, and many constants are also declared in modules and can be used in other modules. A module in ICON has the following syntax:

Listing 2.2: Modules in ICON

```
MODULE <module_name>

USE <any_module_1>, ONLY: <ent11>, <ent12>, ...,<ent1m_1>
...
USE <any_module_n>, ONLY: <entn1>, <entn2>, ...,<entnm_n>

IMPLICIT NONE
PRIVATE
```

```
PUBLIC :: <names of public entities>

<declaration of entities used in the whole module>

CONTAINS

<subprograms>

END MODULE <module_name>
```

Note that the module name is used in the `MODULE` and `END MODULE` statement.

In the `USE` statements, we let the module know that it has to use entities of other modules. It is the convention that all `USE` statements are collected at the beginning of a module and no `USE statements` are given in any of the subprograms of ICON. The advantage is to see immediately all entities that are used from other modules in a particular module. It is easy to know where a certain entity comes from since all “`USE`”s are collected at the beginning. Consequently, it is not possible to use the same entity `a` in one subprogram from module `A` and in another subprogram of the same module from module `B`. The `USE` statements are always applied together with the `ONLY` statement. This means that only the explicitly stated entities `ent11`, . . . of a module can be used. This makes your module much more readable because you immediately know from which module a certain quantity in any of your subprograms comes from without that you have to search in all the modules appearing in a `USE` statement. Possible conflicts are also immediately visible.

The `IMPLICIT NONE` statement has the effect that all entities of the module have to be declared explicitly. This prevents you from being a victim of your own typo errors since such variables are then of no known type and the compilation will fail with a respective error message.

The general `PRIVATE` statement has the effect that other modules cannot use any entity that is not explicitly stated `PUBLIC`. This is useful in particular if somebody does not state the `ONLY` in his `USE` statements for protecting your entities.

Do not forget the `CONTAINS` statement before you define any subprogram.

2.2.2 Derived types

In Fortran 90, it is possible to combine variables of different types under one name. So, you can access a collection of integer, character and real variables with one single name and pass them under this name into subprograms like an array. Such a complex data structure has first to be declared in a type statement. Then, you can declare variables of this new data type.

The syntax of the declaration of a new data type by a type statement is as follows:

Listing 2.3: type statement

```
  type <typename>
    declaration of var1 (e.g. integer   :: iv1)
    ! ...
    declaration of varn
  end type [<typename>]
```

Then, you can declare variables of type `<typename>` by

Listing 2.4: type

```
type (<typename>) [,attributes] :: <varlist>
```

A variable *var* of $< varlist >$ has then n “components”

Listing 2.5: components

```
var%var1
var%var2
! ...
var%varn
```

Each component can be handled separately or *var* can be treated as an integral entity by using its name (e.g. for passing it into subprograms, in write–statements if there are no pointers in the data type involved).

A good example are vectorfields which represent functions $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$, meaning that we will represent each component $f_i : \mathbb{R}^m \rightarrow \mathbb{R}$, $i = 1, \ldots, n$ by an m–dimensional field in FORTRAN 90. For the full description of f, we need a set of n such m–dimensional arrays which we can store in a data structure containing each component $f_i, i = 1, \ldots, n$ as the components of the data structure. The horizontal wind field is an example with $m = 3$ and $n = 2$ (the u and v components depend both on the longitudes, latitudes, and levels).

Listing 2.6: Derived type of a vectorfield

```
TYPE t_windfield
   ! nlon: number of longitudes
   ! nlev: levels
   ! nlat: latitudes
   real, dimension (nlon, nlev, nlat) :: u, v
END TYPE t_windfield

TYPE (t_windfield) :: hwind
```

In this example, `hwind%u, hwind%v` contain the `u` and `v` components of the horizontal windfields. The components `hwind%u, hwind%v` are themselves three–dimensional arrays (vectors). The zonal wind u at a certain longitude index `ilon`, latitude index `ilat` and level index `ilev` is then given by `hwind%u(ilon,ilev,ilat)`.

Derived types can be “nested”. If we use a regional refinement, we need the horizontal wind for each “domain”. Given the type `t_windfield` as in Listing 2.6, we can define dynamic prognostic variables for every domain in the following way:

Listing 2.7: Usage of “nested” derived types

```
INTEGER, PARAMETER         :: max_dom=5

TYPE t_dynvars
  TYPE(t_windfield) :: hwind ! [m/s] horizontal winds
  REAL(wp)          :: temp  ! [K] temperature
END TYPE t_dynvars

TYPE(t_dynvars)     :: dom_dynvars(max_dom)
```

If we would like to access the zonal wind at longitude and latitude index `ilon`, `ilat`, and level index `ilev` of domain $1 \leq$ `jdom` $\leq$ `max_dom`, we can do that by

```
dom_dynvars(jdom)%hwind%u(ilon,ilev,ilat)
```

We can pass `dom_dynvars` to a subprogram `<sub1>`, but we can also pass only a certain domain `jdom` to a subprogram `<sub2>` or even only a certain wind compontent of that domain to a subprogram `<sub3>` as demonstrated in Listing 2.8.

Listing 2.8: Passing derived types into subprograms: calls of subroutines

```
CALL <sub1>(dom_dynvars,max_dom,...)
CALL <sub2>(dom_dynvars(jdom),...)
CALL <sub3>(dom_dynvars(jdom)%hwind%u,nlon,nlev,nlat,...)
```

The corresponding definitions of the subprograms which we assume to be subroutines would be as in Listing 2.9.

Listing 2.9: Passing derived types into subprograms

```
SUBROUTINE <sub1>(dom_dynvars,max_dom)
  INTEGER, INTENT(IN)               :: max_dom
  TYPE(t_dynvars),INTENT(INOUT) :: &
                           & dom_dynvars(max_dom)
  dynvars(1:max_dom)=<expr>(dom_dynvars(1:max_dom))
END SUBROUTINE <sub1>

SUBROUTINE <sub2>(dynvars, nlon, nlev, nlat...)
  INTEGER, INTENT(IN) :: nlon, nlev, nlat
  TYPE(t_dynvars), INTENT(INOUT) :: dynvars
  dynvars%hwind%u=<expr>(dynvars)
  dynvars%hwind%v=<expr>(dynvars)
END SUBROUTINE <sub2>

SUBROUTINE <sub3>(u, nlon, nlev, nlat,...)
  INTEGER, INTENT(IN) :: nlon, nlev, nlat
  REAL(wp), INTENT(INOUT) :: u(nlon,nlev,nlat)
  u(1:nlon,1:nlev,1:nlat)=<expr>(u(1:nlon,1:nlev,1:nlat))
END SUBROUTINE <sub3>
```

2.2.3 Recursive derived types

In Fortran 90, it is possible to construct recursively defined types. In this way, it is possible to create very complex data structures. In particular, it is possible to concatenate derived data structures in a “list of infinite length”. This means that we do not need to know how long this list is (how many elements it will contain) at compile time of the program because the list will be constructed during run time and can be of arbitrary length. One possible application is the definition of a list which contains the information about the tracers in each list element. This can be very complex information for each list element like the mass mixing ratio, the chemical properties, the name, and chemical formula of the tracer. When we need to define a new tracer during runtime of a program, we just append it to the end of the existing list. In the following example, we will present a recursive list each element of which contains the mass mixing ratio of the tracer only. So, we define a data type that contains a 3d–field for the mass mixing ratio of the tracer but in addition a pointer that can point to the next element of the list:

Listing 2.10: Recursive data types

```
TYPE tracer
  REAL(wp), ALLOCATABLE :: xtfield (:,:,:)
  TYPE(tracer), pointer :: next
END TYPE tracer
```

Let us now define three variables of type *tracer*.

Listing 2.11: Variables of type tracer to generate a linked list

```
type (tracer), pointer :: xtracer, firsttracer, lattertracer
```

Using the pointer structure of these variables, we can construct a “chain” of tracers of arbitrary length by the following pointer construct:

Listing 2.12: Linked list of tracers

```
ALLOCATE (firsttracer)
ALLOCATE (firsttracer%xtfield (nlon, nlev, nlat))
!this allocates a 3--dim field for the mass mixing ratio

lattertracer => firsttracer

DO i=2, ntrac ! ntrac is the number of tracers
  ALLOCATE (xtracer)
  ALLOCATE (xtracer%xtfield (nlon, nlev, nlat))
  lattertracer%next => xtracer
  lattertracer => xtracer
END DO
```

We go step by step through the lines of code of Listing 2.12: `firsttracer` will be associated with some memory, the second statement tells FORTRAN90 to allocate memory for `firsttracer%xtfield` that contains the 3d–field of the mass mixing ratio of the first tracer, so tracer number 1.

Furthermore, `lattertracer` points to `firsttracer`, so that we save the information about where the data of `firsttracer` are stored in the variable `lattertracer`.

For `i=2`:
The first two allocate statements reserve memory for another `xtracer` component and its associated 3d–field of mass mixing ratio. The third statement in the loop now connects the `%next` component of `firsttracer` which was intermediately stored in `lattertracer` to the actual (second) tracer. This assures that we can get access to the actual second tracer by `firsttracer%next`. In a last step, we link `lattertracer` to the actual tracer so that we will be able to associate the `%next` component of the actual (second) tracer in a subsequent step to the new (third) tracer.

If we continue our recursive chain over further steps, we see that `firsttracer%next%next%...%next%xtfield` (containing `%next` $(n-1)$–times) is the 3d–field of the n'th tracer of our recursive pointer structure.

In ICON, such linked lists are used for sets of variables describing the state of the atmosphere.

2.2.4 Overloading of subprograms

Each subprogram of FORTRAN90 can be interpreted as a mathematical function (mfunction hereafter) with a set of permissable inputs and outputs. The subprograms become mfunctions because each element of the input set is connected to exactly one element of the output set. There may be several elements of the input set connected to the same element of the output set, but the output element is always unique to each input element. The input and output sets may be very intricate, but they are all finite since each computer operates on finite sets only. The subprograms can be `subroutine`s or `functions`. There is a third category, the `operators`, that are mfunctions also since they take values of an input set and relate them to exactly one element of the output set, like the operator “`+`” or `.gt.`. We are accustomed to the fact that these operators can take real or integer values, but the internal implementation may be different for different FORTRAN types of input. Generally, there are type specific implementations of all these mfunctions. We consider as an example the implementation of

$$+_{\mathrm{irr}} : \begin{cases} \mathbb{Z}_{\mathrm{c}} \times \mathbb{R}_{\mathrm{c}} & \to \mathbb{R}_{\mathrm{c}} \\ (i, x) & \mapsto y \end{cases} \tag{2.1}$$

The symbols $\mathbb{Z}_{\mathrm{c}}$ and $\mathbb{R}_{\mathrm{c}}$ denote the set of integer numbers and real numbers that can be represented in the computer and are finite sets in both cases. The operation “$+_{\mathrm{irr}}$” comprises the conversion of i into a real type, then the addition of two real numbers and the storage of the result, a real number. If we would like to add two integer numbers and get an integer number as result, no type conversion is necessary and the algorithm of adding two integer numbers for an mfunction $+_{\mathrm{iii}}$ is used. In the case of the operator “$+$”, FORTRAN uses the correct respective implementation according to the input types. This is called “overloading”: In fact, the symbol “$+$” stands for the implementation of many mfunctions, e.g. $+_{\mathrm{irr}} : \mathbb{Z}_{\mathrm{c}} \times \mathbb{R}_{\mathrm{c}} \to \mathbb{R}_{\mathrm{c}}$ or $+_{\mathrm{iii}} : \mathbb{Z}_{\mathrm{c}} \times \mathbb{Z}_{\mathrm{c}} \to \mathbb{Z}_{\mathrm{c}}$. For the cosine function, it is similar: According to the input and output type, an implementation giving the result in the desired accuracy is chosen by FORTRAN without that the programmer has to think of the exact implementation.

In FORTRAN90, the user can himself define overloaded subroutines, functions or operators. The user can even extend existing operators. Here are some examples.

Printing a value of a variable

In `SRC/shared/mo_exception.f90`, there is a subroutine `print_value` for printing either logical or integer, or real values. This subroutine is connected to the respective implementations by

```
INTERFACE print_value            ! report on a parameter value
  MODULE PROCEDURE print_lvalue  ! logical
  MODULE PROCEDURE print_ivalue  ! integer
  MODULE PROCEDURE print_rvalue  ! real
END INTERFACE
```

The respective interfaces of the subroutine are:

```
SUBROUTINE print_lvalue (mstring, lvalue)
  CHARACTER(len=*), INTENT(IN)   :: mstring
  LOGICAL, INTENT(IN)            :: lvalue
```

```
SUBROUTINE print_ivalue (mstring, ivalue)
  CHARACTER(len=*), INTENT(IN)   :: mstring
  INTEGER, INTENT(IN)            :: ivalue

SUBROUTINE print_rvalue (mstring, rvalue)
  CHARACTER(len=*), intent(in)   :: mstring
  REAL(wp), INTENT(IN)           :: rvalue
```

The program chooses one of these three implementations according to the type of the second argument when `print_value` is called.

Reading data from files

For various purposes, arrays of data have to be read into ICON. E.g. 3d–ozone concentrations, 2d–sea–surface temperatures, or arrays of parameters describing the aerosol distribution or parameters for the radiative transfer calculation. The reading from netcdf–files is performed by subroutines collected in ∼`icon/io/shared/mo_read_interface.f90`. Although there exists only one implementation of most of the subroutines, they are often defined using the interface technique as explained above. The most important subroutines to read data from netcdf files are those to read a time slice of a 2d– or 3d–array (`read_2D_time` and `read_3D_time`, respectively) and the reading of a 1d–, 2d–, 3d–array (`read_1D`, `read_bcast_REAL_[23]_D`). In order to get the exact interfaces of these subroutines, we have to search for the names under which their implementation can be found. Of the last three routines, only `read_1D` has an interface:

```
INTERFACE read_1D
  MODULE PROCEDURE read_bcast_REAL_1D
END INTERFACE read_1D
```

The definition is as follows:

```
SUBROUTINE read_bcast_REAL_1D(file_id, variable_name, &
         & fill_array, return_pointer)

  INTEGER, INTENT(IN)          :: file_id
  CHARACTER(LEN=*), INTENT(IN) :: variable_name
  REAL(wp), TARGET, OPTIONAL   :: fill_array(:)
  REAL(wp), POINTER, OPTIONAL  :: return_pointer(:)
```

There are also examples of “overloading” e.g. `read_2D_extdim`, `read_2D`, and others. A more detailed description of reading data is given in section 2.3.7.

Comparison of dates and times, example of an extended operator

DT– and TI–variables are handled by the external library `mtime` that is written in C. The source code is stored in ∼`icon/externals/mtime/src`. A basic documentation via doxygen is available in ∼`/icon/externals/mtime/doc`. The link to fortran90 is given through the modules in ∼`/icon/externals/mtime/src/libmtime.f90`. All entities that can be used in fortran90 code are collected in several modules that are finally all integrated into one single module `mtime` that serves as “central module”. The operators `+`, `-`, `>`, `<`, `<=`, `>=`, `==`, `/=` can be used to add, subtract or compare two variables of DT/TI–type indexDT–variable!compare and are first collected in module `mtime_timedelta`. Since the latter is included in module `mtime`, the use statement has to be as follows:

Listing 2.13: Use statement for the extensions of various operators for DT–variables

```
USE mtime, ONLY: datetime, operator(+), operator(-),            &
               & operator(>), operator(<), operator(>=),        &
               & operator(<=), operator(==), operator(/=)
```

All DT–variables are of type `datetime` in icon, e.g.

Listing 2.14: Usage of type `datetime`

```
TYPE(datetime) :: emission_start, actual_datetime
```

You can now use an operator to check whether the actual date and time `actual_datetime` is before or after the date `emission_start` when the emissions should start:

Listing 2.15: Usage of extended operators for DT–variables

```
IF (emission_start <= actual_date) THEN
  perform emissions
ELSE
  do nothing
END IF
```

How DT–variables can be set from namelist entries will be explained later (see Sec. 2.3.6).

There are other extension to some of the above operators programmed in other modules, e.g. in `SRC/shr_horizontal/mo_delaunay_types.f90`. It is even possible to use several of these extensions in one module by the inclusion of the respective operator in several use statements.

2.2.5 Recursive subprograms

In any subprogram `subroutine` or `function`, it is possible to call the same subprogram again if it is declared to be a recursive subprogram. Such a recursive subroutine is used in the nonhydrostatic dynamical core for the time integration. The syntax is the following (`SRC/atm_dyn_iconam/mo_nh_stepping.f90`):

```
  RECURSIVE SUBROUTINE integrate_nh (datetime_local, jg,  &
         & nstep_global, iau_iter, dt_loc, mtime_dt_loc, &
         & num_steps, lat_bc                            )
  ...
  CALL integrate_nh( datetime_current, 1, jstep-step_shift, &
           & iau_iter, dtime, model_time_step, 1, latbc    )
  ...
  END SUBROUTINE integrate_nh
```

Be aware that it is very easy to program infinite loops in this way, if there is not a proper exit condition.

2.3 Modifying the ICON code

In most of the cases, you will not modify the core routines of ICON, but have a rather well defined "add–on" project to realize inside the ICON code. Such an "add–on" project can be the implementation of new diagnostic variables, new parametrizations for the composition of the atmosphere, the implementation of (hypothetical) trace gases ("tracers") from the transport of which you will learn something about the "physics" processes, or the modification of existing physics parametrizations. All these tasks have in common that you will not change the structure of ICON fundamentally but use the existing structure to read new variables or data fields, modify existing processes and to add new output.

The modification of existing processes is a very special task and cannot be the subject of this course since it concerns the modification of the parametrized equations that have to be understood also in terms of their physics content. On the other hand, this course is an attempt to provide knowledge that helps you to perform some standard tasks and use some of the important data structures.

We will discuss the following tasks in some detail:

(*i*) Writing (error) messages

(*ii*) Introduction of your own namelist

(*iii*) Representation of 2d– and 3d–fields in ICON, usage of geographical co–ordinates, and data structures in the dynamics and physics part of ICON

(*iv*) Introduction of new processes into the physics part of ICON

(*v*) Usage of time variables

(*vi*) Reading data from netcdf files

(*vii*) Implementation of a new output stream

Except of (*vi*), we will test our new skills in the implementation of a new passive tracer, emitted at a user defined location. The explanations in the course will treat the more general case and be rather abstract. It is your task to put them alive during the accompanying practical work on the computer and apply them in an example.

In general, we would like to avoid any unnecessary modification of the original ICON code and therefore collect all necessary subprograms for a new feature in a few separate modules. The original ICON code will then be modified by some added `USE` statements and calls of those subroutines. The advantage of this method is that it allows easy updating of the original ICON code and makes a clear separation between your new feature and ICON. All these developments should be done on a personal workstation and intensive tests of the code are necessary before it can be sent to any supercomputer, although mistral can also be used for tests and code development if you use appropriate compiler options. Technical tests must comprise at least tests of vectorization, parallelization, and the restart facility. They can be performed on the standard `atm_amip_test` experiment by the use of the `icon_dev.checksuite` script in `~icon/run/checksuite.icon_dev`:

Listing 2.16: Testing of the ICON code — `exp.atm_amip_test`

```
icon_dev.checksuite -c -m rnmo
```

The option `-c` switches on colour output, `-m` defines the test mode, `rnmo` standing for restart, nproma, mpi, and openmp test. You can omit one of these tests and perform a subset of tests only. The `-h` option shows the usage of the test script.

Here are a few words about generalities for the implementation of new features in ICON. Our goal is to write computationally efficient and easily readable code. In some cases, these two goals may be mutually exclusive, but we have to find a good compromise. A good code documentation is therefore very important although it is often neglected. The consequence is that some code will be abandoned and written again (without documentation) because nobody understands the original code.

A good documentation consists of several "parts":

(*i*) Comments in the code that help the reader to understand the code. Personally, I prefer to have a minimum of comments in the code itself because I still like to "see" the code and not just the comments. Good code should also be self–explanatory up to a certain degree. However, it is important to comment the meaning of the dummy parameters of subprograms and to give some summary of what the code actually does.

(*ii*) It is particularly important to write a "scientific documentation" describing the respective equations and numerical methods you used in your new feature. This must include the description of tests you performed on the code. This is the documentation of your work and the basis of any discussion with your supervisor. A more condensed version of this documentation should be included into the scientific documentation of ICON as soon as your feature becomes an official part of ICON. There is still no comprehensive version of a scientific description of the ICON code.

(*iii*) A "technical documentation" is the description of the subprograms and their connection with the ICON code. It is particularly important to document the dummy parameters of subprograms and all namelist parameters if others are expected to use them. The technical documentation should be as concise as possible. Try to organise it such that it is easy to find the description of the various subprograms, variables and namelists.

Considering many features being implemented into ICON, we notice that most of the features have a program part that does not depend on (model) time and can be performed outside the time integration loop, as for instance reading namelists and files, or certain preliminary computations. Other computations depend on time and must be performed inside the time integration loop. In terms of the overall performance of ICON, it is important to separate these tasks. Typically, you will have the following three steps for new features in the atmospheric part with ECHAM physics:

(*i*) Reading of input namelists: Call your subprogram in `read_atmo_namelists` of `SRC/namelists/mo_read_namelists.f90`

(*ii*) Time independent calculations: Call your subprogram in `init_echam_phy_params` of `SRC/atm_phy_echam/mo_echam_phy_init.f90`

(*iii*) Calculations inside the time loop have to be inserted into `echam_phy_main` of `SRC/atm_phy_echam/mo_echam_phy_main.f90` at the appropriate place.

2.3.1 Messages and error messages in ICON

Sometimes, messages and error messages have to be printed, when ICON produces an error. This is not trivial in a highly parallelized code since you do not want that your (error) message is printed several hundred times. There are several subroutines available to perform this task, all defined in `SRC/shared/mo_exception.f90`. We present `message` here, that writes a message text but continues the execution of the program. The syntax is:

Listing 2.17: The `message` subroutine to output messages and continue the execution of the ICON code

```
SUBROUTINE message (name, text, out, level, all_print,
   adjust_right)
  CHARACTER (len=*), INTENT(in) :: name
  CHARACTER (len=*), INTENT(in) :: text
  INTEGER,           INTENT(in), OPTIONAL :: out
  INTEGER,           INTENT(in), OPTIONAL :: level
  LOGICAL,           INTENT(in), OPTIONAL :: all_print
  LOGICAL,           INTENT(in), OPTIONAL :: adjust_right
```

The formal parameter `name` should be the name of the subprogram calling `message`, `text` being the message you like to print. All other parameters are optional. You can determine another output device than the standard error output with `out`. The `level` variables allows you to choose among prefixes to the message like "INFO", "WARNING" etc.. You can find the possible values in the definition of `message`. The variable `all_print` has to be set to `.true.` if all processors have to print your message.

Your text may be adjusted to the right by setting `adjust_right`.

If a severe error occurs, the ICON program should stop its execution. If this is not done properly, some processors may wait in vain for results from other processors and waist computer time. In order to print a message and stop the ICON program, use `finish`:

Listing 2.18: The `finish` subroutine to print a message and stop the ICON program

```
SUBROUTINE finish (name, text, exit_no)
  CHARACTER(len=*), INTENT(in)           :: name
  CHARACTER(len=*), INTENT(in), OPTIONAL :: text
  INTEGER,          INTENT(in), OPTIONAL :: exit_no
```

Similar to the `message` subroutine, `name` should be the name of the calling subprogram and `text` an error message. The variable `exit_no` prints a prefix `FATAL ERROR` if it is equal to one, no prefix otherwise.

2.3.2 Set up the configuration of your feature – Introduction of your own namelist

We imagine that your new feature consists of a process that you intend to call inside the climate physics part. This new process may need its own namelist, reading external data from netcdf files, and some new diagnostic variables besides its own computation of tendencies used in the time integration of the model. As said at the beginning of this chapter, we would like to keep modifications of the original ICON code at a necessary minimum. At the same time, we will closely follow the method used for other processes implemented already in the echam physics part. In the exercises to this course, you will yourself implement such a new process that simulates the emission and transport of a chemically inert tracer. This is the best way to learn how physics processes are implemented into ICON. The nwp–part of the ICON model uses similar techniques, but has (slightly) different data structures, variables, and names for the physics quantities.

Our implementation is designed such that it is adapted to be used with multiple domains (grid refinements), although grid refinements (nesting) are still not possible together with the climate (echam) physics of ICON in the version used in this course.

There is a certain naming convention applied to all physics processes in the climate physics part: Each process gets a unique acronym consisting of three letters and represented by `<prc>` hereafter. Most of the modules, subroutines, derived types, and variables connected with this process `<prc>` and used or called in the climate (echam) physics part contain `echam_<prc>` in their names.

Thinking of parameters describing `<prc>` and being read from a namelist, the majority of them has to be read for each domain. Our first task is to provide a data structure that can hold all parameters for all domains. The parameters themselves may be of all sorts of types. It is therefore convenient to use a configuration variable of derived type that is an array of the length of all possible domains. We create a module `mo_echam_<prc>_config.f90` in `SRC/atm_phy_echam/`. Inside this module, we declare a type

Listing 2.19: Derived type hosting the namelist parameters of new process `<prc>`

```
TYPE t_echam_<prc>_config
  TYPE(<...>) :: <var1>
  TYPE(<...>) :: <var2>
     .............
END TYPE t_echam_<prc>_config
```

and an array

Listing 2.20: Declaration of `echam_<prc>_config`

```
USE mo_impl_constants, ONLY         : max_dom
type(t_echam_<prc>_config), TARGET :: echam_<prc>_config(max_dom)
```

where `max_dom` is the possible maximum number of domains defined in `mo_impl_constants`.

If we need parameters that are equal for all domains, we can declare a type `t_echam_<prc>_config_global` and a variable

Listing 2.21: Declaration of `echam_<prc>_config_global`

```
type(t_echam_<prc>_config_global), TARGET :: &
    & echam_<prc>_config_global
```

that does not depend on the domains. An alternative is to declare these variables one by one if there are only a few. All configuration variables `echam_<prc>_config[_global]` and other namelist variables have to be public in `mo_echam_<prc>_config.f90`.

There are now four steps to set up the configuration of process `<prc>`:

(*i*) set initial values for all domains and variables

(*ii*) read in values from namelist

(*iii*) check these values in terms of consistency, plausibility

(*iv*) print the values as they will be used inside the program.

We go through these four steps:

(step *i*) Write a subroutine `init_echam_<prc>_config` in `mo_echam_<prc>_config.f90` that sets initial values of all components of `echam_<prc>_config[_global]` and all other configuration variables. This has to be done for all domains, i.e. `echam_<prc>_config(:)%<var1>=<val1>`, ... The subroutine `init_echam_<prc>_config` has to be public and will be used in step (*ii*).

(step *ii*) Before the namelist is read, we initialize the namelist parameters here. Reading namelists is a subtle task in ICON because many namelists are in one file and they have to be stored for restarts of the program. All these tasks are done by the subroutine `process_nml` from module `mo_process_nml`. However, namelists cannot be passed as arguments in parameter lists. The method is therefore the following. Define a module `SRC/namelists/mo_echam_<prc>_nml.f90`. Therein, you use all variables from `mo_echam_<prc>_config` that have to go into the namelist and `process_nml` from `mo_process_nml`. Declare your namelist

Listing 2.22: Declaration of namelist `echam_<prc>_nml`

```
NAMELIST/echam_<prc>_nml/echam_<prc>_config, &
  & echam_<prc>_config_global,...
```

Note that the whole array `echam_<prc>_config` is a member of the namelist. Therefore, the values can be set for every domain by

Listing 2.23: Set values of variables in namelist `echam_<prc>_nml`

```
echam_<prc>_config(1)%<var1>=<val1>
echam_<prc>_config(1)%<var2>=<val2>
          ...
echam_<prc>_config(2)%<var1>=<val1>
echam_<prc>_config(2)%<var2>=<val2>
          ...
```

in the namelist input file. The subroutine `process_echam_<prc>_nml` contains the following

Listing 2.24: Subroutine `process_echam_<prc>_nml`

```
  SUBROUTINE process_echam_<prc>_nml( filename )
    CHARACTER(LEN=*), INTENT(IN)   :: filename
    CALL init_echam_<prc>_config
    CALL process_nml(filename, 'echam_<prc>_nml', nml_read, &
                   & nml_write)
  CONTAINS
    SUBROUTINE nml_read(funit)
      INTEGER, INTENT(in) :: funit
```

```
      READ(funit, NML=echam_<prc>_nml)
    END SUBROUTINE nml_read
    SUBROUTINE nml_write(funit)
      INTEGER, INTENT(in) :: funit
      WRITE(funit, NML=echam_<prc>_nml)
    END SUBROUTINE nml_write
  END SUBROUTINE process_echam_<prc>_nml
```

The mechanism to make `process_nml` to read and write an arbitrary namelist consists in passing the names of the subroutines `nml_read` and `nml_write` as actual parameters into `process_nml`. These two subroutines are defined inside `process_echam_<prc>_nml` and "know" the namelist definition that is given in `mo_echam_<prc>_nml`. The subroutine `process_echam_<prc>_nml` has to be called in `read_atmo_namelists` of `mo_read_namelists` with `atm_namelist_filename` as argument being `NAMELIST_<exp>_atm` in our case.

(step *iii***)** The subroutine `eval_echam_<prc>_config` (`mo_echam_<prc>_config.f90`) has as purpose to check the admissibility of the values of all input variables for all domains $1, \ldots,$ `n_dom` that are actually used, `n_dom` being used from `mo_grid_config`. If values are not admissible, the ICON simulation should be terminated by a call to `finish` of `mo_exception`.

(step *iv***)** The subroutine `print_echam_<prc>_config` should print all input parameters for all domains in use ($1, \ldots,$ `n_dom`). The subroutines of steps (*iii*) and (*iv*) have to be called in `init_echam_phy_params` of `SRC/atm_phy_echam/mo_echam_phy_init.f90`. In general, the evaluation and printing may be called if and only if the process `<prc>` is switched on. How this is done, we will see later. Here, we just call the two routines.

2.3.3 Representation of 2d– and 3d–fields in ICON, geographical co–ordinates

We first consider a 2d–field that depends on the geographical position. i.e. on longitudes and latitudes, only. There is no obvious order of the cells of a grid derived from the icosahedron like indexing them according to longitudes and latitudes. Instead, we just order the cells in a deliberate way and index them in this order with ascending integer numbers. This means that our 2d–field becomes a 1d-array depending on the cell indices. Such arrays are associated with the centres of the triangular grid cells. We do that in a similar way for the edges and vertices of the triangles. If we need a vertical dimension for a 3d–field we get 2d–arrays, the first index being the cell (or edge or vertex) index, the second index being the height level.

We may have a global grid and a region with a certain refinement and an even finer grid inside the first refinement. In that case, we would say that we have three domains: The global domain and the two domains of the refinement. On each model domain, we need the same collection of 2d– and 3d–fields in 1d– or 2d–arrays `<var1>`, `<var2>`, ... in order to describe the state of the atmosphere. They may be prognostic variables or also diagnostic variables, this does not matter at this point. The idea is to collect all these 2d– and 3d–fields `<var1>`, `<var2>`, ... in one data structure `field` of type `t_field` and create a vector `field(1:n_dom)` such that each field element `field(i)` for domain $i = 1,$ `n_dom` is of type `t_field`. Then, each component `field(i)%<var1>` is a 1d– or 2d–array hosting a 2d– or 3d–field depending on the cell (or edge or vertex) index and in the second dimension on the vertical levels if there is a vertical dimension.

If we would do it exactly like this and if we would have each domain on one processor, we would have very long arrays at high resolution. The number of cells would be $N_{\mathrm{r}_2\mathrm{b}_9} \approx 2.097 \times 10^7$ for example. Since it may be difficult to store such long vectors on one processor, we must think about a reduction of the vector length on the processors involved in the computation. The first remedy for this problem is to distribute each model domain onto several processors. This means that we have only certain regions of a domain on each processor. However, even using $1,000$ processors, would reduce the number of cells to $\approx 2.097 \times 10^4$ only. That is still about $21,000$ cells in a 1–d array on each processor. Such high vector lengths can considerably slow down the computations on certain computer architectures. We therefore need another method to reduce the vector length further. We will split the long vector into several chunks of a much smaller length `nproma` and store the long vector in a 2d–array, the first index counting the elements in a block, the second index counting the blocks. The last block may be shorter since `nproma` is not necessarily a divisor of the number of cells. Fig. 2.2 shows an example with 5 grid cells and a maximum vector length `nproma` = 2.

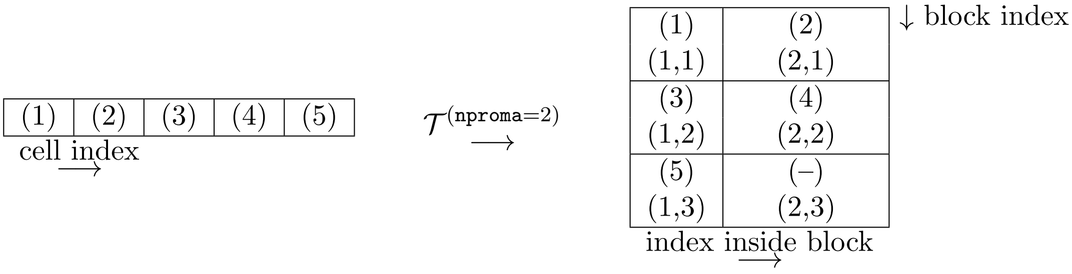


Figure 2.2: Vectorization in ICON

The 3d–fields that were stored in 2d–arrays with the cell index as the first dimension and the second being the vertical co–ordinate, will be stored in 3d–arrays with the first index counting the elements in a block, the second index counting the levels and the third index counting the blocks. The reason for this particular ordering is that we would like to pass blocks of columns to some subprograms which are called inside a loop over blocks. In that case, we can omit the block index inside these subprograms thus reducing the dimensions of the 2d– and 3d–arrays by one.

We will now discuss some important data structures of ICON. It is important to note that all these data structures have a domain index, such that each vector element contains all information for a certain model domain. The first variable `p_patch` is an array of length `n_dom` declared in `SRC/shr_horizontal/mo_model_domain.f90` and contains all important information about the grid.

Listing 2.25: Grid information as stored in `p_patch`

```
TYPE(t_patch), PUBLIC, TARGET, ALLOCATABLE :: p_patch(:)

TYPE t_patch
...
TYPE(t_grid_geometry_info) :: geometry_info
...
INTEGER          :: parent_id
INTEGER          :: parent_child_index
...
INTEGER          :: n_patch_cells
```

```
INTEGER          :: n_patch_edges
INTEGER          :: n_patch_verts
...
INTEGER          :: nblks_c, nblks_e, nblks_v
INTEGER          :: npromz_c, npromz_e, npromz_v
INTEGER          :: nlev, nlevp1
...
TYPE(t_grid_cells)    :: cells
TYPE(t_grid_edges)    :: edges
TYPE(t_grid_vertices) :: verts
...
END TYPE t_patch
```

We explain the “simple” variables first, then the other variables of derived types.

`parent_id, parent_child_index`: gives the index of the parent domain and the index of this child to this parent, respectively.

`n_patch_{cells,edges,verts}`: contains the number of cells, edges, and vertices of the domain on the respective processor.

`nblks_{c,e,v}`: number of blocks of cells, edges, vertices on the respective processor.

`npromz_{c,e,v}`: number of elements in the last block for cells, edges, and vertices on the respective processor.

`nlev, nlevp1`: Number of “full levels”, this means layers, and half levels, so layer interfaces, respectively.

The type `t_grid_geometry_info` is defined in `SRC/shr_horizontal/mo_grid_geometry_info.f90` and contains geometric information:

Listing 2.26: Type for geometric information `t_grid_geometry_info`

```
TYPE t_grid_geometry_info
  INTEGER :: cell_type
  INTEGER :: geometry_type
  ...
  REAL(wp) :: mean_edge_length        ! (meters)
  REAL(wp) :: mean_dual_edge_length   ! (meters)
  REAL(wp) :: mean_cell_area          ! (meters^2)
  REAL(wp) :: mean_dual_cell_area     ! (meters^2)
  REAL(wp) :: domain_length           ! (meters)
  REAL(wp) :: domain_height           ! (meters)
  REAL(wp) :: sphere_radius           ! (meters)
  REAL(wp) :: mean_characteristic_length ! sqrt(mean_cell_area)
END TYPE t_grid_geometry_info
```

`cell_type`: In principle, triangular or hexagonal cells are used, the hexagons forming the dual grid of the triangular grid. This integer variable has either the value of the parameters `triangular_cell` or `hexagonal_cell` that are defined in the same module.

`geometry_type`: Variable describing the overall geometry of the global domain. The most important ones are `sphere_geometry` for a sphere, so the sphere of the earth, and `planar_torus_geometry` that is a plane rectangle with doubly periodic boundary conditions.

`mean_characteristic_length`: This is equal to $\Delta_{\mathrm{r}_n \mathrm{b}_m}$.

The type `t_grid_cells` of `SRC/shr_horizontal/mo_model_domain.f90` gives us information about the grid cells themselves, in particular about their geographical co–ordinates and Coriolis parameter:

Listing 2.27: Information about grid cells provided by the type `t_grid_cells`

```
TYPE t_grid_cells
  ...
  TYPE(t_geographical_coordinates), ALLOCATABLE :: center(:,:)
  REAL(wp), POINTER :: area(:,:)
  REAL(wp), ALLOCATABLE :: f_c(:,:)
  ...
END TYPE t_grid_cells
```

Each component of this data structure is a 2d–array, the first index counting the elements in a block of cells and the second index counting the blocks.

`center`: The type `t_geographical_coordinates` contains as components only the real variables `lon` and `lat` that give the longitude and latitude of the cell center in radiant (not in degree!). The type is declared in `SRC/shared/mo_math_types.f90`

`area`: Gives the surface (area) of each grid cell.

`f_c`: Coriolis parameter at the cell centres.

The types `t_grid_edges` and `t_grid_vertices` contain similar information as the type `t_grid_cells` but for edges and vertices, respectively.

Finally, the geographical co–ordinates of the cell centres can be used by

Listing 2.28: Geographical co–ordinates of cell centres

```
USE mo_model_domain, ONLY: p_patch
...
! jg: domain index
DO jb=1,nblks_c
  DO jc=1,kproma ! actual block length
    lon(jc,jb)=p_patch(jg)%cells%center(jc,jb)%lon
    lat(jc,jb)=p_patch(jg)%cells%center(jc,jb)%lat
  END DO
END DO
```

The maximum block length `nproma` has to be used from `mo_parallel_config` (`SRC/configure_model_mo_parallel_config.f90`):

Listing 2.29: Usage of maximum block length `nproma`

```
USE mo_parallel_config, ONLY: nproma
```

2.3.4 Data structures containing physics and dynamics variables

The state of the atmosphere is described by a set of 2d– and 3d–fields that are either prognostic (integrated over time) or diagnostic (determined by all prognostic variables, boundary conditions, and the composition of the atmosphere) variables. These variables are all collected in one big data structure. The advantage is that "the state" can be easily passed to any subprogram with one argument. Furthermore, this data structure contains all information about "output properties" of these fields, i.e. the names of these variables can be used in any output file.

The structure of this derived type is similar to the variable `p_patch`: We have a vector of `n_dom` elements of the derived type `t_echam_phy_field` for the variables at the previous or prognostic time step, and of derived type `t_echam_phy_tend` for the tendencies, so the derivative with respect to time for the prognostic variables defined in `SRC/atm_phy_echam/mo_echam_phy_memory.f90`:

Listing 2.30: Variables describing the state of (ECHAM) physics in ICON

```
  TYPE(t_echam_phy_field),ALLOCATABLE,TARGET :: prm_field(:)
  TYPE(t_echam_phy_tend ),ALLOCATABLE,TARGET :: prm_tend (:)
```

Here is a list of the more important fields being components of these data structures:

Listing 2.31: Components of `prm_field` and `prm_tend` all at t if not stated differently

```
TYPE t_echam_phy_field
  REAL(wp),POINTER ::      &
  & clon       (:,:),     &!< [rad]   longitude at cell center
  & clat       (:,:),     &!< [rad]   longitude at cell center
  & areacella  (:,:),     &!< [m2]    atmosphere grid-cell area
  & zh         (:,:,:),   &!< [m]     geometric height at half
     levels
  & zf         (:,:,:),   &!< [m]     geometric height at full
     levels
  & dz         (:,:,:)     !< [m]     geometric height of layer
  & ua         (:,:,:),   &! [m/s]   zonal wind
  & va         (:,:,:),   &! [m/s]   meridional wind
  & vor        (:,:,:),   &! [1/s]   relative vorticity
  & ta         (:,:,:),   &! [K]     temperature at
  & tv         (:,:,:),   &! [K]     virtual temperature
  & qtrc       (:,:,:,:),&! [kg/kg] tracer mass mixing ratio
  & omega      (:,:,:),   &! [Pa/s]  vertical velocity
  & geoi       (:,:,:),   &! [m2/s2] geopotential at half levels
  & geom       (:,:,:),   &! [m2/s2] geopotential at full levels
  & presi_old  (:,:,:),   &! [Pa]    pressure at half levels
  & presm_old  (:,:,:),   &! [Pa]    pressure at full levels
  & presi_new  (:,:,:),   &! [Pa]    pressure at half levels t+Δt
  & presm_new  (:,:,:),   &! [Pa]    pressure at full levels t+Δt

  ...
  & aclc       (:,:,:),   &!< [m2/m2] cloud area fractional
  & aclcov     (:,  :),   &!< [m2/m2] total cloud cover
  ...
END TYPE t_echam_phy_field
```

```
TYPE t_echam_phy_tend
  REAL(wp), POINTER ::   &
  ...
  & ta     (:,:,:)   , & ! temperature tendency
  & ta_dyn (:,:,:)   , & ! due to resolved dynamics
  & ta_phy (:,:,:)   , & ! due to parameterized processes
  & ta_cld (:,:,:)   , & ! due to large scale cloud processes
  & ta_cnv (:,:,:)   , & ! due to convective cloud processes
  & ta_vdf (:,:,:)   , & ! due to vertical diffusion
  & ta_sso (:,:,:)   , & ! due to sub grid scale orography
  & ta_gwd (:,:,:)   , & ! due to non-orographic grav. waves
  & ta_rsw (:,:,:)   , & ! due to shortwave radiation
  & ta_rlw (:,:,:)   , & ! due to longwave radiation
  ...
  &   qtrc (:,:,:,:), & ! tracer tendency
  ...
END TYPE t_echam_phy_tend
```

The tracer fields `prm_field(:)%qtrc(:,:,:,:)` and `prm_tend(:)%qtrc(:,:,:,:)` have as dimensions the index describing the position of a cell in a block, the levels, the number of the block and as fourth dimension the tracer index. In the standard ECHAM physics, only 3 tracers are used for water vapour, cloud water, and cloud ice. The tendencies are given in the units of the respective quantity per second, e.g. the temperature tendency would be in K/s. There are various tendencies: the tendency over all processes and tendencies stemming from single processes like dynamics `<var>_dyn`, overall physics `<var>_phy`, i.e. the tendency of `<var>` accumulated over all parameterized processes, the tendency of `<var>` due to large scale cloud processes `<var>_cld`, convective cloud processes `<var>_cnv`, vertical diffusion `<var>_vdf`, subgrid scale orographic effects `<var>_sso` and non–orographic `<var>_gwd` gravity waves. For temperature, there are also the tendencies due to solar (shortwave) and thermal (longwave) radiation `ta_rsw` and `ta_rlw`, respectively. The variables `<var>` are temperature `ta`, the zonal and meridional winds `ua` and `va`, the mass mixing ratio of tracers `qtrc` containing at least the tracers water vapour, cloud water and ice, respectively.

All these variables can be written to output files by giving their names as they appear in subsequent calls of the subroutine `add_var` (search for `field%<var>` in `mo_memory_echam_phy.f90`. For the tendencies, you search for `tend%<var>` and use the name in the `add_var` subroutine preceeded by `prefix`. The prefix is `tend_` in that case, so that the overall temperature tendency being under `tend%ta` in an `add_var` call can be written to the output under the name `tend_ta`.

In a similar way, there is an array `p_nh_state(1:n_dom)` containing the dynamic state. In that case, this means all variables that have to be integrated over time. Its type `t_nh_state` is declared in `SRC/atm_dyn_iconam/mo_nonhydro_types.f90`:

Listing 2.32: Type `t_nh_state` for the description of the state of the nonhydrostatic atmosphere

```
TYPE t_nh_state
  TYPE(t_nh_prog),  ALLOCATABLE :: prog(:) !dimension: time
     levels
  TYPE(t_nh_diag)               :: diag
  ...
  TYPE(t_nh_metrics)            :: metrics

END TYPE t_nh_state
```

The array `prog` contains elements of type `t_nh_prog` for each time slice that is needed for the time integration. For the nonhydrostatic standard time integration, the number of time slices is two, time t for the current time and $t + \Delta t$ for the prediction. In contrast to the prognostic variables, the diagnostic variables must be known at current time only, therefore `diag` is not an array.

The type `t_nh_prog` contains the following components:

Listing 2.33: Type `t_nh_prog` that hosts the prognostic variables

```
TYPE t_nh_prog
  REAL(wp), POINTER ::&
    w(:,:,:),         & ![m/s]    orthogonal vertical wind
    vn(:,:,:),        & ![m/s]    orthogonal normal wind
    rho(:,:,:),       & ![kg/m^3] density
    exner(:,:,:),     & ![-]      Exner pressure
    theta_v(:,:,:),   & ![K]      virtual potential temperature
    tracer(:,:,:,:),  & ![kg/kg]  tracer concentration
    tke   (:,:,:)       ![m^2/s^2]turbulent kinetic energy
    ...
END TYPE t_nh_prog
```

The orthogonal normal wind is given at the midpoints of the triangle edges and is pointing outward and orthogonal to the edges. All other quantities are given at the centres of the triangles. The density is therefore given as

Listing 2.34: Density of the atmosphere as state variable of the nonhydrostatic dynamic core

```
p_nh_state(1:n_dom)%prog(1:2)%rho(1:nproma,1:nlev,1:nblks_c)
```

Here, the index of `p_nh_state` represents the model domain, the index of `prog` the time slice (either t or $t+\Delta t$). The first index of `rho` is the number of the cell in the block, the second index represents levels and the third index the number of the block. Note that the number of blocks (`nblks_c` here) differs for variables given at the centre or edges or vertices of the triangles, even if the length of the block `nproma` is chosen to be the same for all these variables.

The type `t_nh_diag` is more interesting for us since it contains a lot of diagnostic variables from the dynamics:

Listing 2.35: Type `t_nh_diag` containing diagnostic variables from the dynamics

```
TYPE t_nh_diag
  REAL(wp), POINTER:: &
  &  u(:,:,:),            & ! [m/s] zonal wind
  &  v(:,:,:),            & ! [m/s] meridional wind
  ...
  &  omega_z(:,:,:),      & ! [1/s] relative vertical vorticity
                            ! at dual grid (at vertices)
  &  vor(:,:,:),          & ! [1/s] relative vertical vorticity
                            ! interpolated to cells
  ! some tendencies
  ...
  &  temp(:,:,:),         & ! [K] temperature
  &  temp_ifc(:,:,:),     & ! [K] temperature at half levels
```

```
  ...
  &  dpres_mc(:,:,:),     & ! [Pa] ''pressure thickness''
  ...
  &  airmass_now(:,:,:),  & ! [kg/m^2] air mass actual time step
  &  airmass_new(:,:,:),  & ! [kg/m^2] air mass new time step
  ...
  ! variables needed for grid nesting
END TYPE t_nh_diag
```

The winds are interpolated to the cell centres in that case. A "pressure thickness" is given that can be used to calculate the approximate mass of a grid cell, but it is better to use the air mass variables directly.

2.3.5 Introduction of new processes into ECHAM physics

All ECHAM physics processes are called in `SCR/atm_phy_echam/mo_echam_phy_main.f90`. However, as we already saw in the namelist `echam_phy_nml`, each process can have its individual calling frequency and start and end date and time. Furthermore, there are two different ways how the so–called operator splitting is handled. With operator splitting we mean that many processes are calculated separately neglecting the coupling between these processes. Let us consider radiation and (large scale) cloud processes. They influence each other, but the model neglects this influence on the short time interval of one time step and calculates the radiative fluxes with constant clouds and the cloud formation with constant radiation, i.e. constant temperature for one time step. If we have to handle a sequence of processes, we may either base their calculation on the same state of the atmosphere or update all variables before entering the next process by all processes computed so far. The latter has the advantage that negative tracer concentrations cannot occur, if all single processes do not produce negative tracer concentrations. On the other hand, the result depends on the order of the processes in contrast to the former method.

In addition to the various operator–splitting techniques, it may be useful to calculate a certain process but not use its tendencies. In that case, the process is only diagnostic. The parameters determining how the processes are handled, are all collected in the variable `echam_phy_config(max_dom)`, `max_dom` being the allowed maximum number of domains (see `SRC/shared/mo_impl_constants.f90`). Each array element is of type `t_echam_phy_config` which is declared in `SRC/configure_model/mo_echam_phy_config.f90`. We discuss this type first:

Listing 2.36: Type `t_echam_phy_config` and its components for a process `<prc>`

```
LOGICAL  :: lparamcpl
CHARACTER(len=max_timedelta_str_len) :: dt_<prc>
CHARACTER(len=max_datetime_str_len ) :: sd_<prc>
CHARACTER(len=max_datetime_str_len ) :: ed_<prc>
INTEGER                              :: fc_<prc>
```

The variable `echam_phy_config` is the only member of the `echam_phy_nml` namelist. All its components can be set for each domain by reading them from the `echam_phy_nml` namelist. The component `lparamcpl` is `.FALSE.` if the variables are not updated before going into another physics process, `.TRUE.` if the physics state is updated before the calculation of the next process by all previously calculated processes. The default is `lparamcpl=.TRUE.`.

The component `dt_<prc>` stands for the time interval at which the process `<prc>` will be called, `sd_<prc>` and `ed_<prc>` stand for the start and end date between which the process `<prc>` will be called. Their default values are empty strings. The component `fc_<prc>` encodes whether the process is diagnostic (`fc_<prc>=0`) or used to update the physics state (`fc_<prc>=1`). The latter is the default.

The components of `echam_phy_config` describing TI and DT variables cannot be handled by the `mtime` library that performs all calendar and time computations in ICON. They have to be converted into another format and will be collected in `echam_phy_tc(max_dom)`, the array elements of which are of type `t_echam_phy_tc`:

Listing 2.37: Specific components of `t_echam_phy_tc` for a process `<prc>`

```
TYPE(timedelta), POINTER :: dt_<prc>
TYPE(datetime ), POINTER :: sd_<prc>
TYPE(datetime ), POINTER :: ed_<prc>
TYPE(event    ), POINTER :: ev_<prc>
```

where `dt_<prc>` is again representing the time intervall at which process `<prc>` is called, `sd_<prc>` and `ed_<prc>` again represent the start and end date, and `ev_<prc>` is an "event variable", telling ICON whether the process has to be calculated in a certain time step.

The components of `echam_phy_config` have to be initialized with empty strings for all domains for each specific process `<prc>` in the subroutine `init_echam_phy_config` of `mo_echam_phy_config.f90`:

Listing 2.38: Initialization of `echam_phy_config`

```
echam_phy_config(:)%dt_<prc>=''
echam_phy_config(:)%sd_<prc>=''
echam_phy_config(:)%ed_<prc>=''
echam_phy_config(:)%fc_<prc>=1
```

These values will be overwritten if they are specified in the `echam_phy_nml` namelist for any domain. After that, these values are checked whether the given variables are valid TI– and DT–variables, respectively. For this purpose, call `eval_echam_phy_config_details` in the subroutine `eval_echam_phy_config` of `mo_echam_phy_config.f90` for every domain and every process `<prc>`:

Listing 2.39: Check TI– and DT–variables given by namelist `echam_phy_nml`

```
CALL eval_echam_phy_config_details(TRIM(cg),          '<prc>', &
   &                             echam_phy_config(jg)%dt_<prc>, &
   &                             echam_phy_config(jg)%sd_<prc>, &
   &                             echam_phy_config(jg)%ed_<prc>, &
   &                             echam_phy_config(jg)%fc_<prc>  )
```

In this call, `cg` is the domain index transformed into a character of length 2. The loop runs over `jg=1,n_dom`.

The next step is to convert the string variables into `mtime` compatible format by a call of `eval_echam_phy_tc_details` in `eval_echam_phy_tc` of `mo_echam_phy_config.f90`. Also this subroutine has to be called for all domains and processes `<prc>`:

Listing 2.40: Conversion of TI–and DT–variables into `mtime` compatible format for process `<prc>`

```
CALL eval_echam_phy_tc_details(TRIM(cg),     '<prc>',  &
   &                  echam_phy_config(jg)% dt_<prc>,  &
   &                  echam_phy_config(jg)% sd_<prc>,  &
   &                  echam_phy_config(jg)% ed_<prc>,  &
   &                  echam_phy_tc    (jg)% dt_<prc>,  &
   &                  echam_phy_tc    (jg)% sd_<prc>,  &
   &                  echam_phy_tc    (jg)% ed_<prc>,  &
   &                  echam_phy_tc    (jg)% ev_<prc>   )
```

Here, `cg` is again a character of length 2 giving the domain index and this call has to be performed in a loop for `jg=1,n_dom`.

In principle, this should be sufficient for the functionality of the program. However, writing the configuration of your program into an output file makes debugging easier. You have two print routines, one for the components of `echam_phy_config` and and another for the components of `echam_phy_tc`, both called in `print_echam_phy_config` in a loop over all domains `jg=1,n_dom`:

Listing 2.41: Printing the physics configuration variables

```
CALL print_echam_phy_config_details(TRIM(cg), '<prc>', &
   &                   echam_phy_config(jg)% dt_<prc>, &
   &                   echam_phy_config(jg)% sd_<prc>, &
   &                   echam_phy_config(jg)% ed_<prc>, &
   &                   echam_phy_config(jg)% fc_<prc>  )
CALL print_echam_phy_tc_details(TRIM(cg),      '<prc>', &
   &                       echam_phy_tc(jg)% dt_<prc>, &
   &                       echam_phy_tc(jg)% sd_<prc>, &
   &                       echam_phy_tc(jg)% ed_<prc>  )
```

Again, `cg` is a character of length 2 containing the domain index as a string.

Since `echam_phy_config(:)%dt_<prc>` is the time step at which `<prc>` is called, and a time step of zero means that it is never called, we can modify our call of the process specific parameter evaluation and printing such that this is done only if `<prc>` is switched on (see Sec. 2.3.2).

Listing 2.42: Evaluation and printing of process specific parameters

```
lany=.FALSE.
DO jg = 1,n_dom
  lany = lany .OR. (echam_phy_tc(jg)%dt_<prc> > dt_zero)
END DO
IF (lany) THEN
  CALL eval_echam_<prc>_config
  CALL print_echam_<prc>_config
END IF
```

All the above code lines serve to define an additional process only. We now describe how such a process `<prc>` is introduced into `mo_echam_phy_main`. There are two tasks to perform: (*i*) it has to be decided whether this process `<prc>` has to be called at all in a particular time step according to the DT– and TI–variables `dt_<prc>`, `sd_<prc>`, `ed_<prc>` and, if it has to be called (`ii`) the corresponding blocks of columns have to be passed one by one to the subprogram computing the process `<prc>`. Step (*i*) is an evaluation of if clauses that is done in `echam_phy_main` directly. Each process is then wrapped by an interface routine that performs the block related computations and calls the process according to the result of the if clauses in step (*i*). The reason for this complicated separation of tasks is that the tendencies resulting from process `<prc>` are added inside the interface routine depending on the result of the if clauses. We start with the if clauses for step (*i*) that is performed in `echam_phy_main`:

Listing 2.43: If clauses to evaluate whether a process has to be called or not

```
IF ( echam_phy_tc(jg)%dt_<prc> > dt_zero ) THEN
  is_in_sd_ed_interval = &
 & (echam_phy_tc(jg)%sd_<prc> <= datetime_old) .AND. &
 & (echam_phy_tc(jg)%ed_<prc> >  datetime_old)
  is_active = &
 & isCurrentEventActive(echam_phy_tc(jg)%ev_<prc>,datetime_old)
  CALL message_forcing_action(’process␣<prc>’, &
       &                      is_in_sd_ed_interval,is_active)
  CALL omp_loop_cell_tc( patch, interface_echam_<prc>,    &
                       & is_in_sd_ed_interval, is_active, &
                       & datetime_old, pdtime             )
END IF
```

The subroutine `omp_loop_cell_tc` is a general subroutine performing all computations concerning the start and end indices `jcs`, `jce`, of all blocks in an OpenMP loop and calling our physics process `<prc>` for us. The subroutine `interface_echam_<prc>` has to have the following parameter list:

Listing 2.44: Parameter list of interface subroutine for process `<prc>`

```
SUBROUTINE interface_echam_<prc>(                      &
                     & jg, jb, jcs, jce, nproma, nlev, &
                     & is_in_sd_ed_interval, is_active,&
                     & datetime_old, pdtime            )
INTEGER, INTENT(in) :: jg, jb, jcs, jce, nproma, nlev
LOGICAL, INTENT(in) :: is_in_sd_ed_interval, is_active
TYPE(datetime), POINTER :: datetime_old
REAL(wp), INTENT(in):: pdtime
```

The meaning of the parameters is the following:

formal parameter	meaning
`jg`	index of the domain
`jb`	block index
`jcs`, `jce`	start and end index of the cells inside block `jb` for which the calculations have to be performed (`jcs` can be larger than 1)
`is_in_sd_ed_interval`	`.TRUE.` if actual time step is inside time interval for which the process has to be calculated, `.FALSE.` otherwise
`is_active`	`.TRUE.` if the process has to be recalculated in this time step (determined by `echam_phy_config(jg)%dt_<prc>`) `.FALSE.` otherwise
`datetime_old`	is the DT variable giving date and time of the previous time step
`pdtime`	integration time step in seconds

Inside the interface subroutine, your physics process has to be called for each block of the actual domain. Here is an example of such an interface subroutine:

Listing 2.45: Interface routine for calling a physics process

```
MODULE mo_interface_echam_<prc>
  USE mo_kind,             ONLY: wp
  USE mo_echam_phy_memory, ONLY: t_echam_phy_field, prm_field, &
    &                            t_echam_phy_tend, prm_tend
  USE mtime,               ONLY: datetime
  USE mo_run_config,       ONLY: iqt
  USE mo_echam_<prc>,      ONLY: echam_<prc>

  IMPLICIT NONE

  PRIVATE

  PUBLIC :: interface_echam_<prc>

CONTAINS

  SUBROUTINE interface_echam_<prc>(jg, jb, jcs, jce,      &
       &                         nproma, nlev,            &
       &                         is_in_sd_ed_interval,    &
       &                         is_active,               &
       &                         datetime_old,            &
       &                         pdtime                   )
    INTEGER,                   INTENT(in) :: jg,jb,jcs,jce
    INTEGER,                   INTENT(in) :: nproma,nlev
    LOGICAL,                   INTENT(in) :: is_in_sd_ed_interval
    LOGICAL,                   INTENT(in) :: is_active
    TYPE(datetime),            POINTER    :: datetime_old
    REAL(wp),                  INTENT(in) :: pdtime

    TYPE(t_echam_phy_field),   POINTER  :: field
    TYPE(t_echam_phy_tend) ,   POINTER  :: tend
    LOGICAL,                   POINTER  :: lparamcpl
    REAL(wp)                            :: z<prc>dt(nproma)

    field     => prm_field(jg)
    tend      => prm_tend(jg)

    IF ( is_in_sd_ed_interval ) THEN
      IF ( is_active ) THEN
        z<prc>dt=0._wp
        CALL echam_<prc>( jg,                 jcs,          &
                        & jce,                nproma,       &
                        & field%mair(:,nlev,jb),           &
                        & field%clat(:,jb),                &
                        & z<prc>dt                          )
      END IF
```

```
      ! in this problem, the tendency is always added,
      ! echam_phy_config(jg)%fc_<prc> is not yet used.
      tend%qtrc_phy(jcs:jce,nlev,jb,iqt)= &
       & tend%qtrc_phy(jcs:jce,nlev,jb,iqt)+ &
       & z<prc>dt(jcs:jce)
    END IF
    ! dissassociate pointers
    NULLIFY(field,tend)
  END SUBROUTINE interface_echam_<prc>
END MODULE mo_interface_echam_<prc>
```

The if–clause `IF(is_in_se_ed_interval)` assures that `echam_<prc>` is calculated only if the actual date and time is in the desired date–time interval, the if–clause `IF (is_active)` determines whether the process has to be recalculated in this time step.

The programming of the specific operator splitting can be done in the following way by completing the `interface_echam_<prc>` subroutine and set the respective tendency variables. As an example, we assume that we calculated the tracer tendency of a tracer with index `iqt`, i.e. `tend(:)%qtrc_<prc>(:,:,:,iqt)` by a call to our new process `echam_<prc>` as shown in Listing 2.45. We recall that `echam_phy_config(:)%fc_<prc>` was equal to 0 if this tendency was diagnostic only and 1 if the model had to use this tendency to update its model state. Furthermore, `echam_phy_config(:)%lparamcpl=.TRUE.` indicates that the model state should be updated by all previous processes before it enters the next process. The following code contains the respective assignments:

Listing 2.46: Assignement of tendencies according to `fc_<prc>` and `lparamcpl`

```
USE mo_echam_phy_config,    ONLY: echam_phy_config
               ...
LOGICAL,                   POINTER  :: lparamcpl
INTEGER,                   POINTER  :: fc_<prc>
TYPE(t_echam_phy_field),   POINTER  :: field
TYPE(t_echam_phy_tend) ,   POINTER  :: tend

lparamcpl => echam_phy_config(jg)%lparamcpl
fc_<prc>  => echam_phy_config(jg)%fc_<prc>
field     => prm_field(jg)
tend      => prm_tend(jg)

SELECT CASE(fc_<prc>)
CASE(0)
  ! tendency is computed for diagnostic reasons only
CASE(1)
  ! use tendency to update model state
  tend%qtrc_phy(jcs:jce,nlev,jb,iqt) = &
  & tend%qtrc_phy(jcs:jce,nlev,jb,iqt)+ &
  & tend%qtrc_<prc>(jcs:jce,nlev,jb,iqt)
  echam_<prc>(jg)%tracer_emi(jcs:jce,jb) = &
  & tend%qtrc_<prc>(jcs:jce,nlev,jb,iqt)
END SELECT
  ! compute an intermediate physics state
  ! updated by this process for
```

```
  ! input to the next physics process
  IF (lparamcpl) THEN
    field%qtrc(jcs:jce,nlev,jb,iqt) = &
    & field%qtrc(jcs:jce,nlev,jb,iqt) + &
    & tend%qtrc_<prc>(jcs:jce,nlev,jb,iqt)*pdtime
  END IF
! disassociate pointers
  NULLIFY(lparamcpl,fc_<prc>,field,tend)
```

2.3.6 Usage of date and time variables

In Section 2.3.5, we already had to handle DT– and TI–variables: They were read as strings from a namelist, converted to an `mtime` library compatible format, and used later to decide whether a physics process `<prc>` has to be called or not. Another important task is to interpolate external data representing boundary conditions or parameter sets to a certain date and time. To this end, the `mtime` library has to be used directly.

First, DT– and TI–variables are given as strings in namelists for example. These strings have maximum lengths as given by the following two integer parameters of `mtime`

Listing 2.47: Maximum string lengths of DT– and TI–variables

```
USE mtime, ONLY:  max_datetime_str_len, max_timedelta_str_len
```

The corresponding `mtime` library compatible formats are the following types

Listing 2.48: `mtime` library compatible format of DT– and TI–variables

```
USE mtime, ONLY: datetime, timedelta
```

Strings containing DT– and TI–variables can be converted to the `mtime` compatible format by the `newDatetime` and `newTimedelta` functions of `mtime`:

Listing 2.49: Conversion of DT– and TI–variables from stings into `mtime` library compatible format

```
USE mtime, ONLY: max_datetime_str_len, max_timedelta_str_len
USE mtime, ONLY: datetime, timedelta
USE mtime, ONLY: newDatetime, newTimedelta
CHARACTER(len=max_datetime_str_len) :: my_date_time
CHARACTER(len=max_timedelta_str_len):: my_delta_time
TYPE(datetime ), POINTER :: my_date_time_mt
TYPE(timedelta), POINTER :: my_delta_time_mt
my_date_time_mt => newDatetime(my_date_time)
my_delta_time_mt => newDeltatime(my_delta_time)
```

You can transform variables in `mtime` compatible format back into strings by the subroutines `datetimeToString` and `timedeltaToString` of `mtime`:

Listing 2.50: Conversion of `mtime` library compatible variables into strings

```
USE mtime, ONLY: max_datetime_str_len, max_timedelta_str_len
USE mtime, ONLY: datetime, timedelta
USE mtime, ONLY: datetimeToString, timedeltaToString
```

```
CHARACTER(len=max_datetime_str_len) :: my_date_time
CHARACTER(len=max_timedelta_str_len):: my_delta_time
TYPE(datetime ), POINTER :: my_date_time_mt
TYPE(timedelta), POINTER :: my_delta_time_mt
CALL datetimeToString(my_date_time_mt , my_date_time)
CALL timedeltaToString(my_delta_time_mt , my_delta_time)
```

These strings can then be printed e.g. by the `message` subprogram.

An often occuring task is the interpolation of external data sets to a certain date and time. Currently, there is only one method available to determine time interpolation weights corresponding to a linear interpolation in time with respect to monthly given external data. These data are expected to be stored in an array with indices related to the months ranging from 0 to 13, associating the data of December of the predecessor year with index 0 and associating the data of January of the subsequent year with index 13 (ECHAM physics part). The corresponding data type is defined in SCR/shared/mo_bcs_time_interpolation.f90:

Listing 2.51: Data type for time interpolation weights

```
TYPE t_time_interpolation_weights
  TYPE(datetime) :: reference_date
  REAL(wp)       :: weight1, weight2
  ...
  ! MPIM style 0-13 for month indexing
  INTEGER        :: month1_index, month2_index
  LOGICAL        :: initialized = .FALSE.
END TYPE t_time_interpolation_weights
```

The interpolation weights and corresponding indices can be calculated using the function `calculate_time_interpolation_weights` of the same module:

Listing 2.52: Calculation of time interpolation weights

```
FUNCTION calculate_time_interpolation_weights(current_date) &
& RESULT(time_interpolation_weight)
  TYPE(t_time_interpolation_weights) :: time_interpolation_weight
  TYPE(datetime), POINTER, INTENT(in) :: current_date
```

In that case, the result of `calculate_time_interpolation_weights` is of type `t_time_interpolation_weights` as defined in Listing 2.51. Thus, the respective components of the function give the indices and time interpolation weights.

2.3.7 Reading data from netcdf input files

There are different data sets that may be read by ICON: Parameter data sets that have to be known at every grid point, 2d– or 3d–data sets of surface properties or atmospheric composition, or zonal mean values that depend on latitude only. Their vertical co–ordinate may be a pressure co–ordinate or the data may be given on geometric altitudes. All these data sets may depend on time being “transient” data sets or just following a certain seasonal cycle or they may be completely time independent. There is no kind of a “boundary condition tool” handling all these cases in a comprehensive way in ICON. We first discuss various aspects of external data in general.

Grid independent parameter sets: Grid independent parameter sets have to be known to each processor and typically contain much less array elements than the grid itself. Even if they are time dependent, they are the least problematic data sets because the total fraction of memory needed for these data decreases with increasing resolution.

2d–data sets: At high horizontal resolutions, the data sets can be rather large and if they change rapidly with time, the reading may be rather time consuming. The problem may be that the actual ICON resolution is larger than the spatial resolution of the data itself. If the frequency to read such data sets is high, but their actual resolution much lower than the ICON resolution, it is worth to think about an "online–interpolation" of the data.

3d–data sets: The problems are similar to the ones of 2d–data sets. In addition, the vertical co–ordinate demands interpolation. If these data are to be used in the ECHAM–physics part, a vertical pressure co–ordinate can be used since there is a diagnosed hydrostatic pressure available in this part of the program. But also the geometric height can be used.

zonal mean values: The simplest way to handle zonal mean values seems to be an extension and interpolation of these data onto the ICON grid. However, zonal mean values are often very rough estimates of a certain quantity (e.g. optical properties of stratospheric volcanic aerosols). This indicates that the horizontal resolution of such data sets is much lower than the actual ICON resolution. An extension of such a data set means to blow up the data set without gaining any information. For a one–hundred year data set, the final size may then be of the order of several TB. Reading several TB into a parallel program turns out to be very slow and should be avoided. In that case, interpolation inside ICON is certainly the better option.

We consider two basically different read–routines of the module `SRC/io/shared/mo_read_interface.f90`, here. There is a first category of subroutines that allows the reading of data being on the ICON grid only. These routines use a "distributed" read and each processor gets its respective part of grid points. There is a second category of routines reading data on more or less arbitrary grids. For those, it is assumed that the whole data array has to be known to every processor and typical examples are parameter fields. In that case, they are read by one i/o processor and distributed to all other processors in a second step.

Distributed reading of a field on the ICON grid

The subroutine `read_3D_time` belongs to the first category and is good for the reading of 3d–data sets on the ICON grid that depend on time in a fourth dimension. This subroutine has an interface and is implemented under the name `read_dist_REAL_3D_time` (`SRC/io/shared/mo_read_interface.f90`). The "`dist`" means "distributed reading". Here is the definition:

Listing 2.53: Reading a time–dependent 3d–data field on the ICON grid

```
SUBROUTINE read_dist_REAL_3D_time(
    & stream_id,          location,        &
    & variable_name,      fill_array,      &
    & return_pointer,     start_timestep,  &
    & end_timestep,       levelsDimName,   &
    & has_missValue,      missValue        )
```

```
TYPE(t_stream_id), INTENT(INOUT)   :: stream_id
INTEGER, INTENT(IN)                    :: location
CHARACTER(LEN=*), INTENT(IN)           :: variable_name
REAL(wp), TARGET, OPTIONAL             :: fill_array(:,:,:,:)
REAL(wp), POINTER, OPTIONAL            :: return_pointer(:,:,:,:)
INTEGER, INTENT(in), OPTIONAL          :: start_timestep, &
                                        & end_timestep
CHARACTER(LEN=*), INTENT(IN), OPTIONAL :: levelsDimName
LOGICAL, OPTIONAL                      :: has_missValue
REAL(wp), OPTIONAL                     :: missValue
```

Parameters:

`stream_id`: Variable of type `t_stream_id` as returned by the function `openInputFile` (see Listing 2.54) describing the netcdf–file.

`location`: This integer number describes whether your data are associated with cell centres, vertices, or edge midpoints. It is better to use predefined constants here than to fill in explicit numbers since these may change. In the module `mo_impl_constants`, you find the integer variables `on_cells`, `on_vertices`, `on_edges` describing whether the data describe a quantity on the cell centres, the triangle vertices, or the midpoints of the edges, respectively.

`variable_name`: is the name of the variable in the netcdf–file. In the netcdf–file, this variable must have the shape `(time, nlev, ncells)` when looked at it with `ncdump` with C–style output.

`fill_array`: If `fill_array` is present, the data will be stored in this field, `fill_array` must have the right shape to accomodate a 3d–field and the time dimension. It is expected to have the shape `fill_array(nbdim,nlev,nblks,time)`.

`return_pointer`: If this pointer is present, it will be shaped according to `(nbdim,nlev,nblks,time)` and contain the data read from file. If `fill_array` is present at the same time, `return_pointer` is associated with `fill_array`.

`start_timestep, end_timestep`: Index of the first and last time step to be read from the netcdf–file.

`levelsDimName`: Name of the vertical levels dimension in the netcdf–file. This is for checking the name only. If no name is given, it will not be checked. It is anyhow assumed that the second dimension of the variable is the vertical dimension in the necdf–file.

`has_missValue`: If this logical is present, it will be `.TRUE.` on output if the netcdf–file contains a global attribute `missing_value`, otherwise it is set to `.FALSE.`

`missValue`: If this variable is present, it will contain the missing value if the netcdf–file contains a global attribute `missing_value`.

The function `openInputFile` is an overloaded function and has the following parameter list for reading one variable in distributed mode from a netcdf–file:

Listing 2.54: Function `openInputFile` for opening a netcdf file for distributed read

```
TYPE(t_stream_id) FUNCTION openInputFile_dist(filename, patch, &
                                                & input_method)
 CHARACTER(LEN=*), INTENT(IN) :: filename
 TYPE(t_patch), TARGET, INTENT(IN) :: patch
 INTEGER, OPTIONAL, INTENT(IN) :: input_method
```

`filename`: Name of the netcdf input–file.

`patch`: This variable of type `t_patch` contains all information of the distribution of a global variable to the various processors. The type `t_patch` and the variable `p_patch` containing this distribution information of all model domains (refinements) are present in `mo_model_domain`. Actually, `p_patch(1:n_dom)` is a vector the elements of which are all of type `t_patch` (see Sec. 2.3.3).

`input_method`: Integer variable describing whether the files are read by a single i/o processor and distributed then or whether they are read in a distributed way. Use the parameters `read_netcdf_broadcast_method` and `read_netcdf_distribute_method` of `mo_io_config` to assign `input_method` for the two methods, respectively. The standard should be the distributed input, in particular at high resolutions.

Reading of a general field sent to all processors after reading

An example for a subroutine to read a general field that is not defined on the ICON grid and must be sent as a whole to all processors is the subroutine `read_1D_extdim_extdim_time` (`SRC/io/shared/mo_read_interface.f90`). An interface leads us to `read_bcast_REAL_1D_extdim_extdim_time`. The parameter list is as follows:

Listing 2.55: Read a general field and send it to all processors

```
  SUBROUTINE read_bcast_REAL_1D_extdim_extdim_time( &
    & file_id,              variable_name,          &
    & fill_array,           return_pointer,         &
    & dim_names,            start_timestep,         &
    & end_timestep                                  )

    INTEGER, INTENT(IN)           :: file_id
    CHARACTER(LEN=*), INTENT(IN) :: variable_name
    REAL(wp), TARGET, OPTIONAL    :: fill_array(:,:,:,:)
    REAL(wp), POINTER, OPTIONAL   :: return_pointer(:,:,:,:)
    CHARACTER(LEN=*), INTENT(IN), OPTIONAL :: dim_names(:)
    INTEGER, INTENT(IN), OPTIONAL:: start_timestep, end_timestep
```

`file_id`: File unit number of input netcdf file as given by the function `openInputFile` of module `SRC/io/shared/mo_read_interface.f90`, see Listing 2.56.

`variable_name`: Name of the variable in the netcdf file that has to be read.

`fill_array`: 4d–array that will accomodate the field of the netcdf–file. The shape of this array will be determined by the shape of the variable in the netcdf file or has to have the corresponding shape.

`return_pointer`: If `fill_array` is present, it will be associated with `fill_array` or will just contain the input data.

`dim_names`: Must contain the names of the dimensions except the last dimension which has to be `time`.

`start_timestep, end_timestep`: Index of the first and last time step to be read.

The `file_id` is again provided by `openInputFile`, but this time, the parameter list must contain the filename only in contrast to the previous case, and the specific function `openInputFile_bcast` is called by this interface.

Listing 2.56: Function `openInputFile` for opening a netcdf file for reading on the i/o processor

```
INTEGER FUNCTION openInputFile_bcast(filename)
  CHARACTER(LEN=*), INTENT(IN) :: filename
```

All files should be closed after the call to the read subprograms by a call to `closeFile` (`SRC/io/shared/mo_read_interface.f90`). There are again two different subroutines behind `closeFile`:

```
  SUBROUTINE closeFile_dist(stream_id)
  TYPE (t_stream_id), INTENT(in) :: stream_id
  ...
  SUBROUTINE closeFile_bcast(file_id,return_status)
  INTEGER, INTENT(in) :: file_id, return_status
```

2.3.8 Defining new “streams”

We already know the data structures `prm_field` and `prm_tend`. We also know that they are of derived type and that their components can be written into output files. For this output mechanism, we need to provide further information to ICON. In fact, the array `prm_field(1:n_dom)` of type `t_echam_phy_field` has a “cousin” `prm_field_list` of type `t_var_list` that contains the information for the data handling. Any variable of type `t_var_list` maybe called a stream. The information contained in `prm_field_list` can be filled in by calls to certain subprograms. Let us discuss the type `t_var_list` and its “child” `t_var_list_intrinsic`, first (see Listing 2.57). It is declared in `SRC/shared/mo_linked_list.f90`. You will see that there is a recursive pointer structure hidden in this type.

Listing 2.57: Type `t_var_list` for “stream” information

```
TYPE t_var_list
  TYPE(t_var_list_intrinsic), POINTER :: p
END type t_var_list

TYPE t_var_list_intrinsic
  ...
  CHARACTER(len=128) :: name          ! stream name
  TYPE(t_list_element), POINTER :: first_list_element
                        ! reference to first list element
  ...
  INTEGER             :: list_elements ! allocated elements
```

```
  LOGICAL            :: loutput        ! output stream
  LOGICAL            :: lrestart       ! restart stream
  LOGICAL            :: linitial       ! initial stream
  CHARACTER(len=256) :: filename       ! name of file
  CHARACTER(len=8)   :: post_suf       ! suffix of output  file
  CHARACTER(len=8)   :: rest_suf       ! suffix of restart file
  CHARACTER(len=8)   :: init_suf       ! suffix of initial file
  ...
  INTEGER            :: patch_id
  INTEGER            :: vlevel_type
  ...
  LOGICAL            :: lmiss          ! flag: true, if
                ! variables should be initialized with missval
  ...
END TYPE t_var_list_intrinsic
```

The stream has a name being a component of the type `t_var_list_intrinsic`. The type `t_var_list_intrinsic` contains a "linked list" in form of a recursive pointer structure, the first element of which is `first_list_element`. There is further general information that concerns all variables in this list (stream). It is e.g. indicated the number of allocated list elements at a certain instant in the program (`list_elements`). Note that this number can change at any time by just adding a new list element. There are also logical variables indicating whether this stream is written to output files (`loutput`), restart files (`lrestart`) or is a stream the variables of which have to be used for initialization (`linitial`). The `patch_id` indicates the model domain (so the refinement step of the grid) to which it belongs. The variable `vlevel_type` indicates whether the stream contains variables on model levels or pressure levels.

We will inspect the type `t_list_element` further. This type is also declared in `SRC/shared/mo_linked_list.f90` (see Listing 2.58).

Listing 2.58: Types `t_list_element` and `t_var_list_element` for linked lists

```
TYPE t_list_element
  TYPE(t_var_list_element)      :: field
  TYPE(t_list_element), POINTER :: next_list_element
END TYPE t_list_element
```

This type is recursive by the fact that it contains `next_list_element` of the same type `t_list_element`. The component `field` is of type `t_var_list_element` that is declared in `SRC/shared/mo_var_list_element.f90`:

Listing 2.59: Type `t_var_list_element` containing information about individual list elements

```
TYPE t_var_list_element
  REAL(dp), POINTER            :: r_ptr(:,:,:,:,:)
  INTEGER,  POINTER            :: i_ptr(:,:,:,:,:)
  LOGICAL,  POINTER            :: l_ptr(:,:,:,:,:)
  ...
  TYPE(t_var_metadata)         :: info
  TYPE(t_var_metadata_dynamic) :: info_dyn
END type t_var_list_element
```

The components `r_ptr`, `i_ptr`, and `l_ptr` accomodate the corresponding fields. The redundant dimensions will be of length one and get the index 1. The "meta data" information is particularly interesting for us. It contains the following (see `SRC/shared/mo_var_metadata_types.f90`):

```
TYPE t_var_metadata
  ...
  CHARACTER(len=VARNAME_LEN) :: name
  ...
  TYPE(t_cf_var)             :: cf
  TYPE(t_grib2_var)          :: grib2
  ...
  LOGICAL                    :: lrestart
  LOGICAL                    :: loutput
  LOGICAL                    :: lrestart_cont
  ...
  TYPE(t_union_vals)         :: initval
  ...
  TYPE(t_vert_interp_meta)   :: vert_interp
  TYPE(t_hor_interp_meta)    :: hor_interp
  ...
  LOGICAL                    :: lmiss    ! flag: true, if variable
                             ! should be initialized with missval
  TYPE(t_union_vals)         :: missval ! missing value
  ...
  END TYPE t_var_metadata
```

`name`: Name of the "stream element". A stream element in that case is a 2d– or 3d–variable in the simplest form, but can be a whole group of 3d–variables also. E.g. all tracers can be stored as one variable "tracer". It is possible to reference them separately by another data structure.

`cf`: In this derived type, the metadata for netcdf–format output according to the CF (Climate and Forecast) conventions can be stored.

`grib2`: This derived type contains the metadata for output in the GRIB2 (GRIdded Binary data, version 2) format.

`lrestart`: is `.TRUE.` if the stream element has to be written to the restart files, `.FALSE.` otherwise.

`loutput`: is `.TRUE.` if this stream element has to be written to the output file, `.FALSE.` otherwise.

`lrestart_cont`: There are variables that must be read from the restart file. On the other hand, if a kind of a new submodel is switched on while starting from a restart file, the variables belonging to this submodel are not in the first restart file. Nevertheless, the model should continue even if these variables do not exist in the (very first) restart file. In that case, the `lrestart_cont` variable can be set to `.TRUE.` so that ICON continues although it does not find this stream element in the restart file. If it is mandatory that the stream element is in the restart file, you must set `lrestart_cont` = `.FALSE.`.

`initval`: This variable can be used to assign an initial value to the field of the stream element. There are three components `initval%rval`, `initval%ival`, and `initval%lval` that will host the real, integer or logical value according to the types of the fields associated with this stream element.

{`vert,hor`}`_interp`: Derived types containing information about the vertical and horizontal interpolation method. E.g. if the variable is interpolated to pressure levels when it is written to an output file.

`lmiss`: When this variable is set to `.TRUE.` the field of the corresponding stream element is set to a missing value specified in `missval`.

`missval`: This variable has components `missval%rval`, `missval%ival`, and `missval%lval` for the real, integer and logical fields associated with this stream element. If `lmiss` is set to `.TRUE.`, this variable contains the corresponding missing values.

Before we proceed to the recipe how to define a new stream, let us summarize this survey of data structures associated with streams. We discuss again the example of `prm_field` and `prm_field_list` that are both defined in `SRC/atm_phy_echam/mo_echam_phy_memory.f90`:

```
TYPE(t_echam_phy_field),ALLOCATABLE,TARGET :: prm_field(:)
TYPE(t_var_list),ALLOCATABLE :: prm_field_list(:)
```

We first note that the stream `prm_field_list` will be allocated as a vector of length `n_dom` such that each element is of type `t_var_list`. Similarly, `prm_field` has `n_dom` elements of type `t_echam_phy_field`. We will see that it would be sufficient to work with `prm_field_list` alone, and `prm_field` is just defined for convenience. Considering `prm_field_list` first, we see that it has one single component `p` given for each domain `jg`:

```
prm_field_list(jg)%p
```

This component `p` contains information about the name of the stream, or whether it is written to an output file or how many list elements are allocated in domain `jg` at that position in the program as shown in the following examples, respectively:

```
prm_field_list(jg)%p%name
prm_field_list(jg)%p%loutput
prm_field_list(jg)%p%list_elements
```

On the other hand, the component `p` is the anchor of the linked list:

```
prm_field_list(jg)%p%first_list_element
```

This component `first_list_element` being of derived type `t_list_element` contains two components the first of which points to the next list element, the second of which contains all information of the first list element, respectively:

```
prm_field_list(jg)%p%first_list_element%next_list_element
prm_field_list(jg)%p%first_list_element%field
```

Inspecting the component `field` further, we see that its type `t_var_list_element` contains multidimensional arrays for hosting the values of the field and other components for metadata for this stream element, respectively:

```
prm_field_list(jg)%p%first_list_element%field%r_ptr(:,:,:,:,:)
prm_field_list(jg)%p%first_list_element%field%i_ptr(:,:,:,:,:)
prm_field_list(jg)%p%first_list_element%field%l_ptr(:,:,:,:,:)
!metadata:
prm_field_list(jg)%p%first_list_element%field%info
```

In the component `info`, we can find the name of the stream element, a "flag" whether it is written to the output file or information about the stream element according to the CF conventions or the GRIB2 data format:

```
prm_field_list(jg)%p%first_list_element%field%info%name
prm_field_list(jg)%p%first_list_element%field%info%loutput
prm_field_list(jg)%p%first_list_element%field%info%cf
prm_field_list(jg)%p%first_list_element%field%info%grib2
```

Until now, we were examining the first list element only. If we would like to access the fields of the i'th list element, this would involve $i-1$ times a `next_list_element`. Thus, accessing the array of the i'th list element, we would have to write:

Listing 2.60: Array of $i-1$st stream element

$$\texttt{prm\_field\_list(jg)\%p\%first\_list\_element\%}\overbrace{\texttt{next\_list\_element\%...\%next\_list\_element}}^{i-1\text{ times}}\texttt{\%}$$
$$\texttt{\%field\%r\_ptr(:,:,:,:,:)}$$

As you notice, the access to the arrays of this linked list hosting 2d– or 3d– variables becomes very lengthy, although this structure is very convenient for the output. This is the moment when the cousin `prm_field` comes into play. In this case, the access to the arrays is rather simple. For the zonal wind in domain `jg`, we have for example:

Listing 2.61: Zonal wind as member of `prm_field`

```
prm_field(jg)%ua(:,:,:)
```

However, we have to ensure that this array "contains the same data" as the corresponding array of Listing 2.60 assuming that the zonal wind `ua` is the i'th stream element in the linked list. "Contains the same data" in this case would mean that we wish that ICON uses the same place in the memory under two different access names, the one of Listing 2.60 and the one of Listing 2.61. In fact, the array of Listing 2.60 is a pointer associated with `prm_field(jg)%ua(:,:,:)`.

Our next task will be to explain how we can build such a linked list and how we may connect it to a "simple data structure" like the `prm_field`. The "simple data structure" will be referred to as Fast Access Data structure (FAD). This data structure must be accessible from many modules and may contain a considerable number of 2d– and 3d–arrays. It should be defined in a separate module in order to avoid circular dependencies in the sequel. A good naming example would be `mo_<prc>_memory.f90`. In the same module, the stream or linked list must be defined and the connection must be established between the linked list and the FAD. We proceed in the following steps:

(i) Definition of a derived type `t_<fad>` containing all 2d– and 3d–arrays of variables that will be integrated into the linked list later in a file `mo_<prc>_memory.f90`. The variables must all have the pointer attribute:

```
TYPE t_<fad>
  {REAL,INTEGER,LOGICAL}, POINTER :: <var1>(:,:[,:[,:[,:]]])
  ...
  {REAL,INTEGER,LOGICAL}, POINTER :: <varn>(:,:[,:[,:[,:]]])
END TYPE t_<fad>
```

(*ii*) Declare a 1d–array `<fad>(:)` the elements of which are of type `t_<fad>` in order to host the data for each model domain. Furthermore, we need a 1d–array of type `t_var_list` for our linked list. Both variables have to be allocatable since the number of domains is not known at compile time:

```
USE mo_linked_list, ONLY: t_var_list
...
TYPE(t_<fad>),ALLOCATABLE,TARGET :: <fad>(:)
TYPE(t_var_list),ALLOCATABLE     :: <fad>_list(:)
```

(*iii*) We will use several subprograms to construct the linked list and therefore create a subroutine `construct_<prc>_list`. We need to know the actual number of domains and later some other variables which we will get from `patch_array` of type `t_patch`. The subroutine will be:

```
USE mo_model_domain,   ONLY: t_patch
USE mo_impl_constants, ONLY: SUCCESS
USE mo_exception,      ONLY: finish
...
SUBROUTINE construct_<prc>_list (patch_array)
TYPE(t_patch),INTENT(IN) :: patch_array(:)
INTEGER :: n_dom,ist
n_dom=SIZE(patch_array)
ALLOCATE(<fad>(n_dom), STAT=ist)
IF (ist/=SUCCESS) THEN
 CALL finish('construct_<prc>_list␣of␣mo_<prc>_memory.f90',&
             &'allocation␣of␣<fad>␣array␣failed')
END IF
ALLOCATE(<fad>_list(n_dom), STAT=ist)
IF (ist/=SUCCESS) THEN
 CALL finish('construct_<prc>_list␣of␣mo_<prc>_memory.f90',&
             &'allocation␣of␣<fad>_list␣array␣failed')
END IF
END SUBROUTINE construct_<prc>_list
```

(*iv*) We have to build the linked list and allocate the arrays for each domain separately. Since we will do the same for each domain, but we have multiple tasks for each domain, the best is to define a new subroutine `new_<prc>_list` that will be called in a loop over the domains. This loop will be in `construct_<prc>_list`

```
USE mo_impl_constants,      ONLY: MAX_CHAR_LENGTH
...
SUBROUTINE construct_<prc>_list (patch_array)
...
INTEGER :: jg
CHARACTER(len=MAX_CHAR_LENGTH) :: listname
```

```
DO jg = 1, n_dom
!the listname should contain the model domain index
WRITE(listname,'(a,i2.2)') '<fad>_D',jg
CALL new_<prc>_list(jg,             patch_array(jg), &
                    &TRIM(listname),                  &
                    &<fad>_list(jg), <fad>(jg)        )
END DO
END SUBROUTINE construct_<prc>_list
```

(*v*) The new subroutine `new_<prc>_list` for each model domain must first create a new linked list and then add elements to this list. When the elements are added to this list, we can get pointers to the 2d– and 3d– (or up to 5d–arrays) and associate them with our elements in the data structure `<fad>`. The subroutine must fill in all the information about the elements of the linked list. We will first present such a subroutine and then explain the data structures and subprograms in more detail.

```
USE mo_cf_convention,  ONLY: t_cf_var
USE mo_var_list,       ONLY: new_var_list, &
    &                         default_var_list_settings
USE mo_grib2,          ONLY: t_grib2_var, grib2_var
USE mo_cdi,  ONLY: DATATYPE_FLT32,  DATATYPE_FLT64,   &
    &              GRID_UNSTRUCTURED, GRID_CELL
USE mo_cdi_constants, ONLY: GRID_UNSTRUCTURED_CELL
USE mo_zaxis_type,    ONLY: ZA_REFERENCE
USE mo_io_config,      ONLY: lnetcdf_flt64_output
USE mo_parallel_config,ONLY: nproma
...
SUBROUTINE new_<prc>_list (jg,         p_patch, &
                           &listname,           &
                           &<fad>_list, <fad>   )
INTEGER,INTENT(IN)                  :: jg
TYPE(t_patch),INTENT(IN)            :: p_patch
CHARACTER(len=*),INTENT(IN)         :: listname
TYPE(t_var_list),INTENT(INOUT)      :: <fad>_list
TYPE(t_<fad>),INTENT(INOUT)         :: <fad>

TYPE(t_cf_var)                      :: cf_desc
TYPE(t_grib2_var)                   :: grib2_desc
INTEGER                             :: shape3d(3)
INTEGER                             :: datatype_flt

CALL new_var_list(<fad>_list, TRIM(listname), patch_id=jg)
!default settings for each list element
CALL default_var_list_settings(<fad>_list,  lrestart=.TRUE.)
!Each variable (list element) has to be added to the list
!We give one example here
!Create data structures first that have to be passed
IF ( lnetcdf_flt64_output ) THEN
  datatype_flt = DATATYPE_FLT64
ELSE
```

```
  datatype_flt = DATATYPE_FLT32
ENDIF
cf_desc = t_cf_var('<var>','<units>',&
                 &'<description>',datatype_flt)
grib2_desc = grib2_var(                           &
             &<discipline>,<category>,<parameter>, &
             &<ibits>, GRID_UNSTRUCTURED, GRID_CELL)
shape3d = (/nproma,p_patch%nlev,p_patch%nblks_c/)
CALL add_var(<fad>_list,'<varname>',<fad>%<var>,  &
            &GRID_UNSTRUCTURED_CELL, ZA_REFERENCE,&
            &cf_desc, grib2_desc,                 &
            &ldims=shape3d,                       &
            &vert_interp =                        &
            &create_vert_interp_metadata(         &
            &vert_intp_type=vintp_types("P","Z","I") )&
            &)
END SUBROUTINE new_<prc>_list
```

The subroutine `new_var_list` takes as arguments the variable `<fad>_list` of type `t_var_list`, a name of the list and the index of the patch (domain). With the subroutine `default_var_list_settings`, it is possible to assign default values to all arguments that are passed to the subroutine `add_var`. Once a default value is assigned, it cannot be changed by a second call of `default_var_list_settings`! You can just override the value in a call of `add_var`. The output in netcdf format allows either 64 or 32 bits. You can choose this in the `io_nml` namelist in the variable `lnetcdf_flt64_output` (`.TRUE.` or `.FALSE.`). In order to make this choice effective, we have to pass it to the `cf_desc` argument of `add_var`. Otherwise, `cf_desc` contains the variable name as it will appear in the netcdf output file, the units and a longer description of the variable. In the GRIB2 format, the variables are categorized into a “discipline”, a “category” in each discipline and the “parameter” unique to each variable itself. They are all integer numbers and can be found in the WMO GRIB2 documentation. It is worth to lookup the respective numbers when adding new variables in order to avoid conflicts with other variables. The parameter `<ibits>` tells ICON the number of bits to be used for packing the variables in the GRIB2 files. The constants `DATATYPE_PACK16` and `DATATYPE_PACK24` are provided by the module `mo_cdi` for this purpose.

The shape of the variable is just an array giving the length of each dimension of the respective variable.

The call of the subroutine `add_var` has as first argument the linked list `<fad>_list` of type `t_var_list`, the second argument is the name of the list element, the third argument a component of the `<fad>` data structure. Exactly this allows that the arrays of the FAD are connected to the linked list (stream) later. The following two arguments describe the horizontal grid and the vertical grid. We always use variables on the “unstructured” icosahedral grid, the vertical co–ordinate is always `ZA_REFERENCE` or `ZA_REFERENCE_HALF` for the layer interfaces, and `ZA_SURFACE` for 2d– (surface) fields. These constants are defined in module `mo_zaxis_type` The last argument tells ICON how to vertically interpolate the data if necessary.

(*vi*) It is good practice to deallocate the memory at the end of a program. This includes the deallocation of the linked list and the corresponding `<fad>`:

```
USE mo_var_list, ONLY: delete_var_list
```

```
SUBROUTINE destruct_<prc>_list
INTEGER :: n_dom, jg, ist
n_dom = SIZE(<fad>)
DO jg = 1, n_dom
  CALL delete_var_list ( <fad>_list(jg) )
END DO
DEALLOCATE(<fad>_list,STAT=ist)
IF (ist/=SUCCESS) THEN
CALL finish('destruct_<prc>_list␣of␣mo_<prc>_memory.f90', &
           &'deallocation␣of␣<fad>_list␣array␣failed')
END IF
DEALLOCATE(<fad>,STAT=ist)
IF (ist/=SUCCESS) THEN
CALL finish('destruct_<prc>_list␣of␣mo_<prc>_memory.f90', &
           &'deallocation␣of␣<fad>␣array␣failed')
END IF
END SUBROUTINE destruct_<prc>_list
```

(*vii*) The last step is to introduce the subroutines `construct_<prc>_list` and `desctruct_<prc>_list` into ICON. Clearly, these routines have to be called outside the time loop during the initialization and the “clean–up” phase, respectively. If we are using the “ECHAM physics”, the right place to call `construct_<prc>_list` is `construct_atmo_nonhydrostatic` of `SRC/drivers/mo_atmo_nonhydrostatic.f90`. The clean–up would then be in `cleanup_echam_phy` of `SRC/atm_phy_echam/mo_echam_phy_cleanup.f90`.

Bibliography

[1] L. Bonaventura, M. Esch, H. Frank, M. Giorgetta, T. Heinze, P. Korn, L. Kornblueh, D. Majewksi, A. Rhodin, P. Rípodas, B. Ritter, D. Reinert, and U. Schulzweida. ICON programming standard. Report, Deutscher Wetterdienst, Max Planck Institute for Meteorology, Hamburg, 2012.

[2] D. Leuenberger, M. Koller, O. Fuhrer, and C. Schär. A generalization of the SLEVE vertical coordinate. *Monthly Weather Review*, 138:3683–3689, 2010.

Listings

List of Symbols

$\mathrm{r}_n\mathrm{b}_m$	Horizontal resolution of icosahedral grid
$N_{\mathrm{r}_n\mathrm{b}_m}$	Number of grid cells of icosahedral grid
$\Delta_{\mathrm{r}_n\mathrm{b}_m}$	Resolution: square root of surface of average triangle of icosahedral grid
$\Delta'_{\mathrm{r}_n\mathrm{b}_m}$	Resolution: Side length of an average triangle of icosahedral grid
$\Delta''_{\mathrm{r}_n\mathrm{b}_m}$	Resolution: Shortest distance between the midpoints of triangle sides of an average triangle of icosahedral grid
$\Delta'''_{\mathrm{r}_n\mathrm{b}_m}$	Resolution: Side length of an average hexagon of the dual hexagonal grid
`SRC`	subdirectory of source code in main model directory
$+_{\mathrm{irr}}$	sum of integer and real number resulting in a real number
$+_{\mathrm{iii}}$	sum of two integer numbers resulting in an integer number

Index