

Using and programming ICON— a first introduction

S. Rast

`sebastian.rast@pimet.mpg.de`

July 31, 2017

Foreword

This course is a first introduction to the ICON (ICOsahedra Nonhydrostatic model) program focussing on technical aspects of the atmospheric part using ECHAM physics. The first part 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 explained. This part contains a survey of fortran techniques used in ICON before the most important data structures and subprograms are explained. These are selected to get the user of ICON acquainted with the introduction of a own namelist, the usage of 2d- and 3d-fields including geographical coordinates, reading in data from netcdf files, the usage of date and time variables, and the definition of new output streams.

This is a slightly improved version of a first draft used in the ICON course 2016. The author is very grateful to many ideas by M. Giorgetta and the participants of this first course, 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

1	Getting started with ICON	1
1.1	Source code of ICON	2
1.2	Basic Compilation	4
1.3	Model grid	5
1.3.1	Horizontal grid	5
1.3.2	Vertical grid	10
1.4	Performing computer experiments with ICON	11
1.4.1	Preparation of a computer experiment with ICON	11
1.4.2	Namelists for ICON	12
1.4.3	Input data for ICON	24
2	The code of ICON	27
2.1	Flowchart of ICON	27
2.2	Survey of FORTRAN techniques	30
2.2.1	Modules	30
2.2.2	Derived types	31
2.2.3	Recursive derived types	33
2.2.4	Overloading of subprograms	34
2.2.5	Recursive subprograms	37
2.3	Modifying the ICON code	37
2.3.1	Messages and error messages in ICON	39
2.3.2	Introduction of your own namelist	40
2.3.3	Representation of 2d- and 3d-fields in ICON, geographical coordinates	43
2.3.4	Data structure containing physics and dynamics variables	47
2.3.5	Introduction of new processes into ECHAM physics	50
2.3.6	Usage of date and time variables	54
2.3.7	Reading data from netcdf input files	55
2.3.8	Defining new “streams”	59
	Bibliography	69
	List of listings	73

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 the “standard” and may require some programming effort. Distortion 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 like a “torus” (this means a rectangle with doubly periodic boundary conditions for being precise) is possible.

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 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 tricky. 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 “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 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 echem 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_2017/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_2017/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) Compiling, running, testing

aclocal.m4: calls test of data type scripts (e.g. `acinclude.mr`)

cmake(d): scripts to locate important libraries and their include files

cmake.configure: configure file using cmake (does not work)

CMakeLists.txt: needed for cmake makefiles.

config(d): Configuration files for various computer platformes

configure: Script to create a specific makefile for compilation of icon on your actual computer platform.

configure.ac: file to create configure script

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.

m4(d): contains a file testing the support of special pointers. Similar to the `acinclude.m4` file

Makefile.in: Inputfile for generation of Makefile.

make_runscripts: This script generates run scripts from run script templates. The reason for this is that the call of the ICON executable needs initial files or restart files, the actual call of the executable may depend on the platform, on the other hand, the input files and namelist for a specific model configuration are different for each experiment. The actions that are the same for every experiment are collected in `run/exec.iconrun`, the specific setting for an experiment in other files `run/exp.*`. The script `make_runscripts` creates a machine dependent, experiment specific run script from these two components that is ready for submission.

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.

schedulers(d): ICON runs on various special platforms (not interesting for us)

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

target_confmake.ksh: Script to configure and make ICON.

target_database.ksh: Contains a list of “targets”, i.e. possible arguments to the configure command.

(ii) Documentation

doc(d): contains model documentation. In particular there is a namelist overview `Namelist_overview.pdf` describing all input variables of icon and a subdirectory `userguide` that contains `icon_user_guide.pdf` that will become the standard description of ICON in terms of user relevant information. This user guide contains the namelist overview and the description of special model configuration like the aqua planet or the radiative convective equilibrium configuration. In the user guide, there is no description of potential output variables of the model configuration using ECHAM physics, data structures inside ICON, or programming concepts of the ICON code. However, there is some information about the grid generation and other useful information but it may be outdated.

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.

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

src(d): Source code of icon. The source code is organized in approximately 31 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”, “atmosphere”, “land”, 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.

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

vertical_coord_tables(d): List of A and B coefficients of σ -hybrid coordinates 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 processecors 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_2017/<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 big computer platforms, 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 during 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 -n 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
```

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.

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 many triangles. Fig. 1.1 represents an icosahedron with one triangulated face in front.

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

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

$$N_{r_n b_m} = 20n^2 4^m \quad (1.1)$$

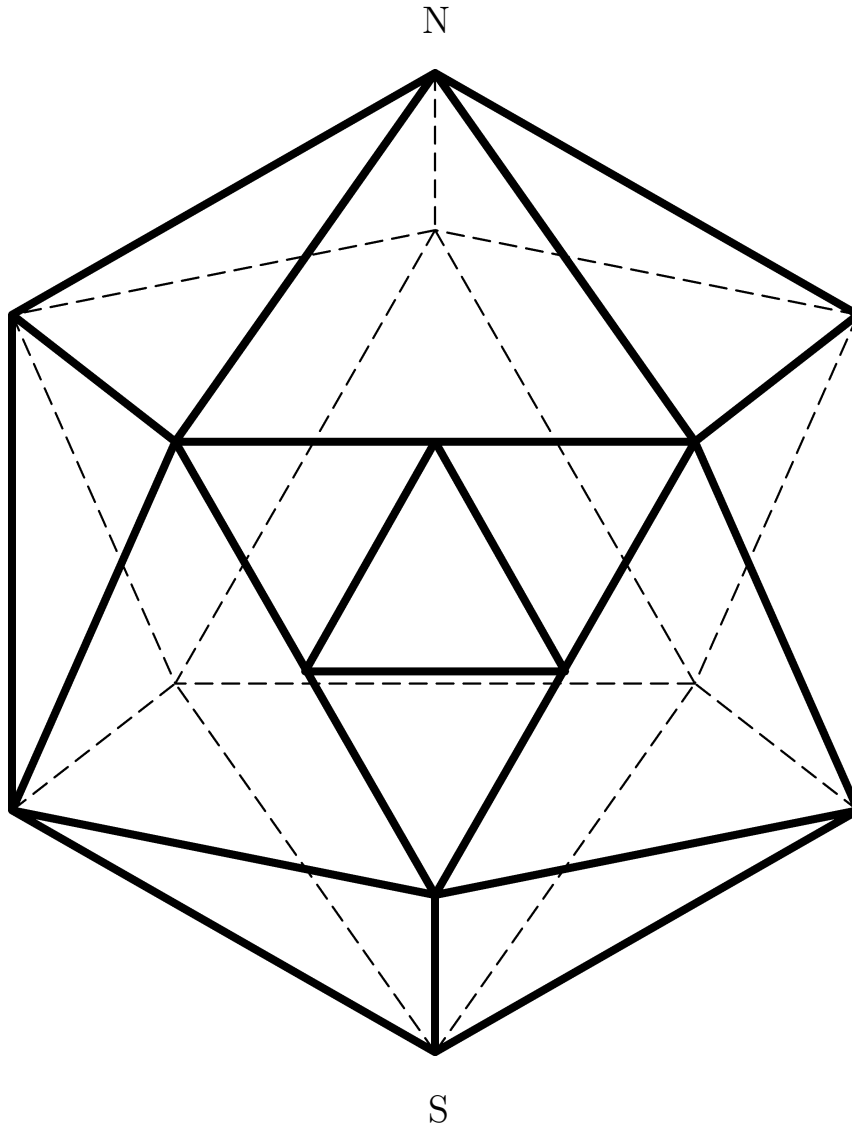


Figure 1.1: Icosahedron

We describe the triangulation in detail now. 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.

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

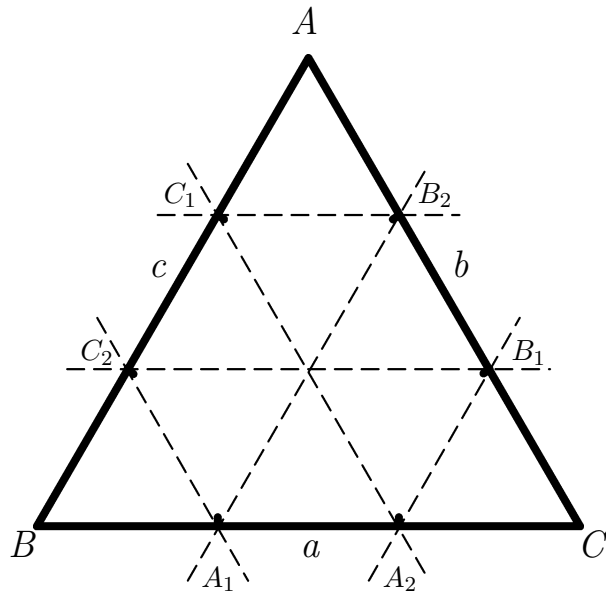


Figure 1.2: Triangulation by 2 equidistant parallels

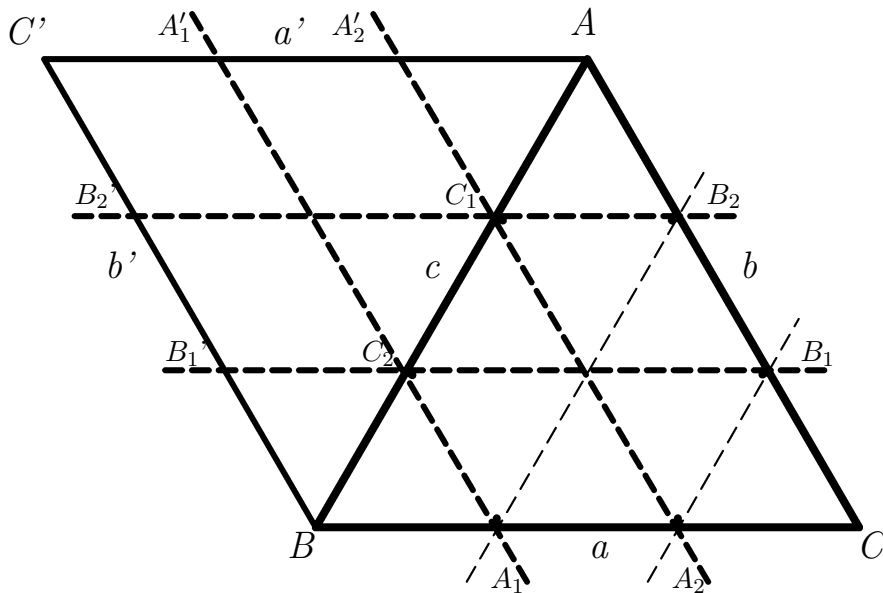


Figure 1.3: Parallelogram split into $n^2 = 9$ smaller parallelograms by a pair of 2 equidistant parallels

Proof: Given an equilateral triangle (A, B, C) with edges (a, b, c) , we complete it into a parallel-

ogram 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 , $i = 1, \dots, n-1$, 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 $r_n b_m$.

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 called to be dual to the grid of triangle vertices. On 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.

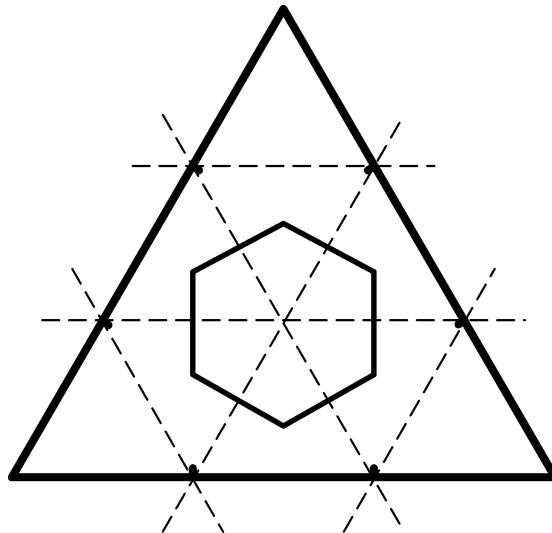


Figure 1.4: Dual hexagon to a triangle grid

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 $r_n b_m$ in:

$$\Delta_{r_n b_m} := \sqrt{A} = \sqrt{\frac{4R_{\text{earth}}^2 \pi / \underbrace{N_{r_n b_m}}_{=20n^2 4^m}}{4 \times 5n^2 2^{2m}}} = R_{\text{earth}} \sqrt{\frac{4\pi}{4 \times 5n^2 2^{2m}}} = \frac{R_{\text{earth}}}{n2^m} \sqrt{\frac{\pi}{5}}, \quad (1.2)$$

where $R_{\text{earth}} \approx 6371$ km is the radius of the earth. For the grid $r_2 b_4$ with $N_{r_2 b_4} = 20480$, we get a resolution of $\Delta_{r_2 b_4} \approx 160$ 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 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 theorem. This leads to $h = \sqrt{3}a/2$ and $A = \sqrt{3}a^2/4$. From this and with (i), we get:

$$\Delta'_{r_n b_m} := a = 2\sqrt{A/\sqrt{3}} = 2/\sqrt{\sqrt{3}}\Delta_{r_n b_m} = \frac{R_{\text{earth}}}{n2^{m-1}} \sqrt{\frac{\pi}{5\sqrt{3}}} \quad (1.3)$$

The vorticity is given at the triangle vertices, thus $\Delta'_{r_n b_m}$ 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 $r_n b_m$, this is exactly the length of a triangle side of a grid $r_n b_{m+1}$, thus we get:

$$\Delta''_{r_n b_m} := \Delta'_{r_n b_{m+1}} = \frac{R_{\text{earth}}}{n2^m} \sqrt{\frac{\pi}{5\sqrt{3}}} = \frac{1}{2}\Delta'_{r_n b_m} \quad (1.4)$$

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:

$$\Delta'''_{r_n b_m} := \frac{R_{\text{earth}}}{n2^{m-1}} \sqrt{\frac{\pi}{15\sqrt{3}}} \quad (1.5)$$

Note that $\Delta'''_{r_n b_m} = \frac{2}{\sqrt{3\sqrt{3}}}\Delta_{r_n b_m} \approx 0.877 \times \Delta_{r_n b_m}$. 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 there 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 edges 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 coordinate 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 coordinate 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 $l = 1, \dots, n$. Layer 1 is at the top of the atmosphere and level n at the bottom of the atmosphere. Half level $n + 1$ is identical with the surface of the earth. In contrast to a pressure coordinate 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 $\Delta z_{\min} > 0$ be the minimum thickness of the layers, $z_{\max} > \Delta z_{\min}$ 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 $z_l^{(h)}$, $l = 1, \dots, n + 1$ is defined in the following way:

$$\alpha := \ln\left(\frac{\Delta z_{\min}}{z_{\max}}\right) / \ln\left(\frac{2}{\pi} \arccos\left(\left(\frac{n-1}{n}\right)^\sigma\right)\right) \quad (1.6)$$

$$z_l^{(h)} := z_{\max} \left(\frac{2}{\pi} \arccos\left(\left(\frac{l-1}{n}\right)^s\right) \right)^\alpha$$

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 coordinate (SLEVE coordinate [2]) since $z_{n+1}^{(h)} = 0$, i.e. it is placed at 0m 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

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

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

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

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: Preparation of input data and namelist files with all the input parameters of ICON, actually performing the simulation and “baby sitting” the computer runs, and in a last phase, postprocessing and analysis of the results. These three phases are reflected in the following sections. All example run scripts store 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 accommodate 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
    <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. This simulation of time periods and the restarts making up a whole computer experiment has 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 <code>icon_master.namelist</code>	
<code>master_nml</code>	restart information and path to directory with input and output files
<code>master_model_nml</code>	model specific information about namelist files, parallelization
<code>master_time_control_nml</code>	calendar information, start and stop date, restarts
<code>time_nml</code>	deprecated
<code>jsb_control_nml</code>	mode of JSBACH surface/land model
<code>jsb_model_nml</code>	configuration and namelist file of JSBACH
Further namelists for other models like ocean	
Namelists in file <code>NAMELIST_<exp_name>.atm</code> describing the configuration of the atmosphere	
<code>run_nml</code>	configuration about time integration, model configuration in terms of presence of processes and tracers.
<code>extpar_nml</code>	external parameters
<code>initicon_nml</code>	mode of initialization
<code>grid_nml</code>	grid information
<code>sleve_nml</code>	vertical coordinate information
<code>nonhydrostatic_nml</code>	parameters for the nonhydrostatic dynamic core
<code>parallel_nml</code>	settings for parallel computing and vectorization
<code>transport_nml</code>	advective transport of tracers
<code>radiation_nml</code>	general radiative processes, information about the atmosphere composition included
<code>psrad_nml</code>	parameters describing the new PSRAD scheme alone
<code>psrad_orbit_nml</code>	orbit of the earth, can be used with PSRAD only
<code>mpi_phy_nml</code>	ECHAM physics parameterizations
<code>echam_conv_nml</code>	ECHAM convection
<code>echam_cloud_nml</code>	ECHAM cloud cover parameters
<code>gw_hines_nml</code>	gravity wave parameterization according to Hines
<code>interpol_nml</code>	Interpolation for reconstructing variables on the grid. In particular, radial basis functions (RBF) are used.
<code>output_nml</code>	namelist specifying output
Namelists in file <code>NAMELIST_<exp_name>.lnd</code> describing the configuration of the land	
<code>jsb_model_nml</code>	overall configuration of land model JSBACH
<code>jsb_{srf,soil,veg}_nml</code>	configuration of surface, soil, and vegetation calculations

Preferably, all the namelist variables should be set in the scripts `~icon/run/exp.<exp_name>` and `~icon/run/exec.iconrun`. However, it may sometimes be hard to find or identify these variables therein. We go through the namelists one by one and discuss the most important variables in these respective namelists:

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.

Namelist `master_model_nml.` This namelist has to be given for each model component like atmosphere and ocean with the respective entries. 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 number that describes which MPI threads are simulating this model component by an increment 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 the unit. An eight minute interval can be written as `PT8M` instead of `POYOMODTOH8MOS`. 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 calendar 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 month 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 there. 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_shortcode, 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.

Namelist `run_nml`.

modelTimeStep: Variable of TI-format giving the model time step. Default: Empty 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 adiabatic 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 the dynamics and transport by parameterized processes is 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 one variable 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).

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 echem physics does not use this vertical grid information directly, but relies on the corresponding pressures.

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.

dca_y_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., ...).

rayleigh_coeff: Rayleigh coefficient for damping in “sponge layer”. Default: (0.05, -1., -1., ...).

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). For a more detailed description of the tracer transport properties, see [3].

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.

<code>itype_hlimit</code>	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.

<code>ivadv_tracer</code>	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.

<code>itype_hlimit</code>	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.

Namelist `radiation_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. Since the `radiation_nml` namelist already contained a lot of variables which are relevant for the PSRAD scheme also, it was intended to avoid duplication of these variables in a `psrad_nml` namelist. On the other hand, there are variables for the PSRAD scheme alone which are in the `psrad_nml` namelist. Furthermore, the `radiation_nml` namelist contains a lot of variables that do not really describe the radiation itself but the composition of the atmosphere for example. In order to start a separation of these groups of variables, a new namelist `psrad_orbit_nml` was introduced that describes the orbit of the earth.

`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`.

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

Default: H₂O: 1, CO₂, O₂, CFC11, CFC12: 2, CH₄, N₂O: 3, O₃: 0.

`vmr_<spec>`: Real variable giving the volume mixing ratios $x_{<spec>}$ of respective species `<spec>` if `irad_<spec> = 2`. Default: $x_{\text{CO}_2} = 348 \times 10^{-6}$, $x_{\text{CH}_4} = 1650 \times 10^{-9}$, $x_{\text{N}_2\text{O}} = 306 \times 10^{-9}$, $x_{\text{O}_2} = 0.20946$, $x_{\text{CFC11}} = 214.5 \times 10^{-12}$, $x_{\text{CFC12}} = 371 \times 10^{-12}$, no default value for O₃.

`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.

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

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

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

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

`isolrad`: This integer gives the mode of the solar irradiation, in particular the choice of the “solar constant”.

<code>isolrad</code>	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.

Namelist `psrad_nml`. This namelist describes settings that concern the PSRAD radiation alone. They do not have any effect on the RRTM radiation.

lradforcing: 2d-logical that switches on (.true.) or off (.false.) the computation of instantaneous aerosol forcing versus reference aerosols for short and longwave radiation, respectively. Default: (/false., false./). **Disabled for the moment.**

irad_aero_forcing: Integer that describes the mode of reference aerosols. The same numbers as for irad_aero can be chosen. Default: 0. **Disabled for the moment.**

fco2: real factor by which the CO₂ volume mixing ratio is multiplied in the PSRAD radiation calculation for easy definition of scenarios. Default: 1.0

Namelist psrad_orbit_nml. This namelist contains parameters that influence the orbit if PSRAD radiation is used.

cecc: Real number describing the eccentricity of the orbit if a Kepler orbit is used. Default: 0.016715

cobld: Real number describing the obliquity of the earth axis versus the plane of the orbit. Default: 23.44100

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.

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 mpi_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 very 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 `mpi_phy_config(jg)`, `jg` being the domain index, 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

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

for example. The time intervals have to be integer multiples of `modelTimeStep`. The following processes can be triggered by this method:

`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 Teyss  re.

`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, you would have to give the following variables in the namelist:

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

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

Furthermore, surface processes can be switched on or off in a similar way. There are logicals as components of `mpi_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_1>` 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 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.

<code>remap</code>	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_mode** = 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) parameters describing e.g. the composition of the atmosphere.

Most of the external data are stored either in `/pool/data/ICON/` or in `~icon/data`.

Initial conditions

The dynamics of the atmosphere is determined by functions

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

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 are vertical and horizontal winds, the air density, the Exner pressure $\Pi = (p/p_0)^{R_d/c_p}$, p_0 : reference pressure, R_d : specific gas constant of dry air, c_p : isobaric heat capacity of dry air, the virtual potential temperature $\theta = T(p_0/p)^{R_d/c_p}$, 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 the data handling is a nightmare already now. 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<r>B<bb>.DOM<dd>.nc` where `<descriptor>` is either `ifs2icon` or `dwdFG`, or `dwdana` for the various data sources, the resolution $r_n b_m$ is encoded by one digit `<r> = 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 the 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.

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)
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 files for radiation	
rrtmg_sw.nc	parameters for radiative transfer calculation used in PSRAD for the underlying RRTMG algorithm, solar radiation
rrtmg_lw.nc	parameters for radiative transfer calculation used in PSRAD for the underlying RRTMG algorithm, thermal 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_namelist.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.

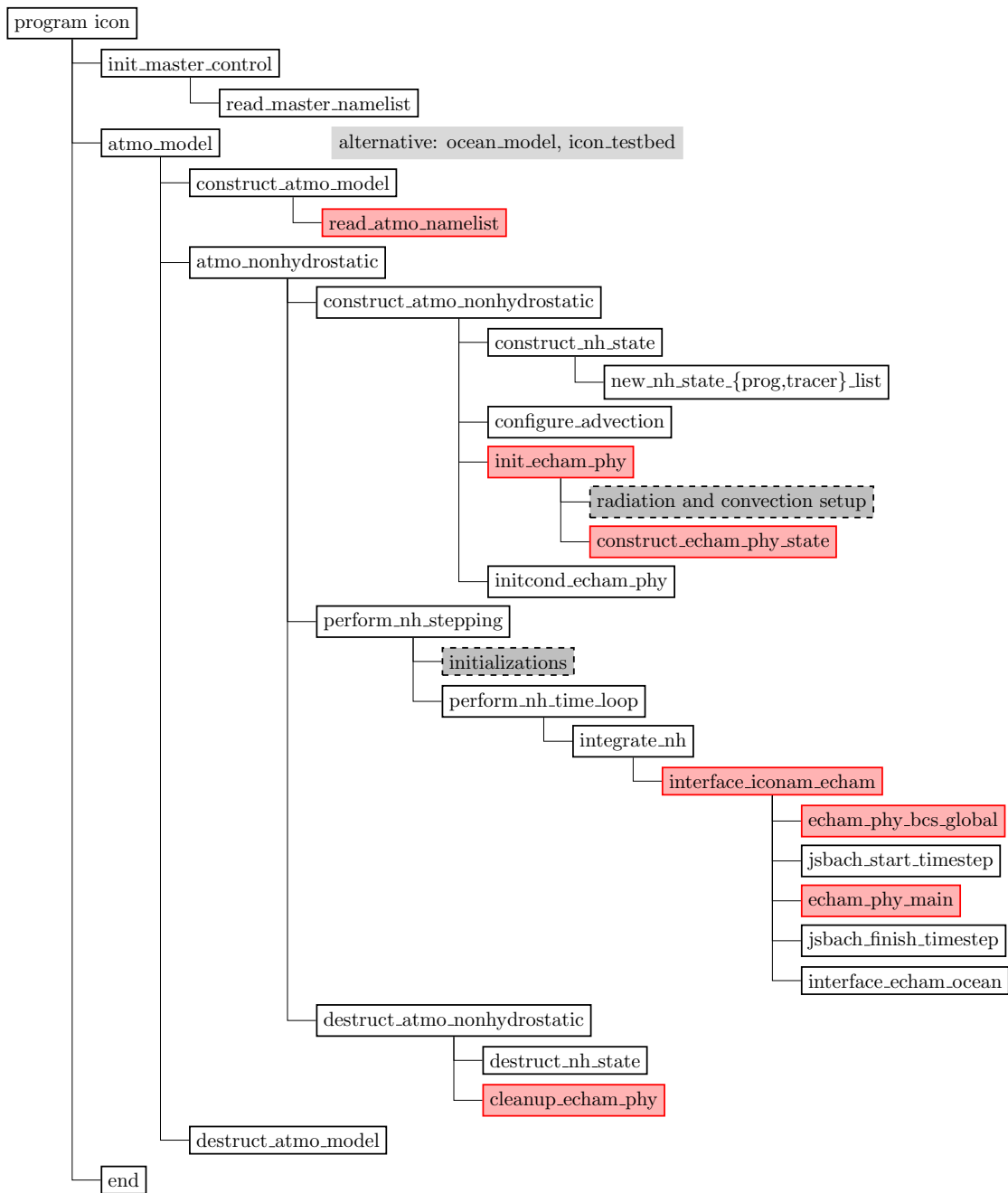
`construct_atmo_model:` (`SRC/drivers/mo_atmo_model.f90`) calls the reading of namelists and all important initializations outside time integration loop.

`read_atmo_namelist:` (`SRC/namelists/mo_read_namelists.f90`) reads namelists concerning the atmospheric model.

`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}_list:** (SRC/atm_dyn_iconam/mo_nonhydro_state.f90) allocating the derived types containing all prognostic variables and the tracer variables (transported quantities).
- 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:** (SRC/atm_phy_echam/mo_echam_phy_init.f90) initializations of echam physics that have to be done outside the time loop, e.g. radiation and convection setup. The 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).
- initcond_echam_phy:** (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_time_loop:** (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.
- 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_global:** (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.
- 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 defined 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>, ..., <ent1m1>
...
USE <any_module_n>, ONLY: <entn1>, <entn2>, ..., <entnmn>

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 another 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 $f_i : \mathbb{R}^m \rightarrow \mathbb{R}$, $i = 1, \dots, n$ by an m -dimensional field in FORTRAN 90. For the full description of f , we need a set of n such m -dimensional vectors which we can store in a data structure containing each component $f_i, i = 1, \dots, n$ 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 a 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      :: NDOMAIN_MAX=5

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

TYPE(t_global_dynvars)  :: global_dynvars(NDOMAIN_MAX)

```

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

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

We can pass `global_dynvars` to a subprogram `<sub1>`, but we can also pass only a certain domain `jdom` (subprogram `<sub2>`) or even only a certain wind component of that domain (subprogram `<sub3>`) into any subprogram as demonstrated in Listing 2.8.

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

```

CALL <sub1>(global_dynvars, NDOMAIN_MAX, ...)
CALL <sub2>(global_dynvars(jdom)%hwind, ...)
CALL <sub3>(global_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>(global_dynvars , NDOMAIN_MAX)
  INTEGER , INTENT(IN)                :: NDOMAIN_MAX
  TYPE(t_global_dynvars), INTENT(INOUT) :: &
                                     & global_dynvars(NDOMAIN_MAX)
  dynvars(1:NDOMAIN_MAX)=<expr>(global_dynvars(1:NDOMAIN_MAX))
END SUBROUTINE <sub1>

SUBROUTINE <sub2>(dynvars , nlon , nlev , nlat...)
  INTEGER , INTENT(IN) :: nlon , nlev , nlat
  TYPE(t_dynvars) , INTENT(INOUT) :: dynvars
  dynvars%domain_hwind%u=<expr>(dynvars)
  dynvars%domain_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 the runtime of the 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 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 the description of a variable describing the state of the atmosphere containing different sets of variables depending on the exact model configuration.

2.2.4 Overloading of subprograms

Each subprogram of FORTRAN90 can be interpreted as a mathematical function (mfunction hereafter) with a set of permissible 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 complex, but they are all finite since each computer operates on finite sets only. The subprograms can be `subroutines` 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

$$+_{\text{irr}} : \begin{cases} \mathbb{Z}_c \times \mathbb{R}_c & \rightarrow \mathbb{R}_c \\ (i, x) & \mapsto y \end{cases} \quad (2.1)$$

The symbols \mathbb{Z}_c and \mathbb{R}_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 “ $+_{\text{irr}}$ ” 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 $+_{\text{iii}}$ 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. $+_{\text{irr}} : \mathbb{Z}_c \times \mathbb{R}_c \rightarrow \mathbb{R}_c$ or $+_{\text{iii}} : \mathbb{Z}_c \times \mathbb{Z}_c \rightarrow \mathbb{Z}_c$. 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 the subroutines collected in `~icon/io/shared/mo_read_interface.f90`. Although there exists only one implementation for 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(:)
```

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 there 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-type 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 extension 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_current, jg, &
    & nstep_global, dt_loc, num_steps )
    ...
CALL integrate_nh( datetime_current, jgc, &
    & nstep_global, dt_sub, nsteps_nest )
    ...
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 coordinates, 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 use 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 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` is for switching 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. A comprehensive documentation can also be found in [3].

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 and 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. This documentation should be included into the user guide [3] as soon as your new feature becomes an official part of ICON. 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.

Most of the new features have a program part that does not depend on 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` 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 or even 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 (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 parameters `name` should name 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 previous prefixes to the message like “INFO”, “WARNING” etc.. You can find the possible values in the definition of `message`. The variable `all_print` triggers which processors 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 it’s 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 `test` 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 assume that you already created a module for your feature `SRC/*/mo_<nfeature>.f90`. In this module, you should provide a derived type `t_<nfeature>` the components of which comprise all namelist variables. We declare a variable of this new type and make it public by

```
TYPE t_<nfeature>
  type(<...>) :: <var1>
  type(<...>) :: <var2>
  ...
END TYPE t_<nfeature>

PUBLIC          :: <nfeature>_config

TYPE(t_<nfeature>) :: <nfeature>_config
```

Our next task will be to assign values to all of these components, either by setting default values or by replacing them by values read from the input namelist.

All modules reading namelists are collected in the directory `SRC/namelists`. For the introduction of your own namelist `<yrrnlist>_nml`, create a new module `SRC/namelists/mo_<yrrnlist>_nml.f90`. In this module, declare all namelist variables as module variables and the namelist itself. You declare a namelist by the `NAMELIST` statement as shown in Listing 2.19. Furthermore, we `USE` the configuration variable `<nfeature>_config`.

Listing 2.19: Namelist module – Declaration of namelist

```
USE mo_<nfeature>, ONLY: <nfeature>_config
NAMELIST /<yrrnlist>_nml/ <var1>, <var2>, ...
```

```
<type1>    :: <var1>
<type2>    :: <var2>
...
```

Our next task is to assign default values, read the namelist from an input file, make sure that these values are available to all processors and documented in a file `nml_atmo.log`, and write the namelist to some special file for eventual restarts. The strategy in ICON is, that namelists are read by all processors such that it is not necessary to send the values from an i/o processor to all other processors. We go through all these steps now that have to be performed in a subroutine `read_<yrrnlist>_namelist`:

Assignment of default values: The first step is to assign default values to `<var1>`, `<var2>`, ... These values will be copied to the corresponding components of `<nfeature>_config` at the end of this subroutine.

Reading in case of a restart: If the particular run is a restart, so a continuation of a simulation, we have to open the file with the respective namelist and read the namelist. This is done by the following commands:

Listing 2.20: Read namelist in restart

```
USE mo_master_config,      ONLY: isRestart
USE mo_restart_namelist,  ONLY: open_and_restore_namelist, &
                           & close_tmpfile
...
SUBROUTINE read_<yrrnlist>_namelist (filename)
CHARACTER(LEN=*), INTENT(IN)  :: filename
INTEGER                      :: funit
...
IF (isRestart()) THEN
  funit = open_and_restore_namelist('<yrrnlist>_nml')
  READ(funit,NML=<yrrnlist>_nml)
  CALL close_tmpfile(funit)
END IF
...
END SUBROUTINE read_<yrrnlist>_namelist
```

It seems to the author that you read the actual namelist even in a restart. Thus, you have the possibility to change your namelist values when a restart is performed.

Reading the namelist in case of an initial run: We have to open the file containing our namelist, find it there and read it. In addition, we have to write the default values of the namelist and the actual settings to a temporary file from which the documentation file `nml_atmo.log` will be created. The following commands are used for this purpose:

Listing 2.21: Read namelist in initial run

```
USE mo_namelist,          ONLY: position_nml, POSITIONED, &
                           & open_nml, close_nml
USE mo_mpi,              ONLY: my_process_is_stdio
USE mo_nml_annotate,    ONLY: temp_defaults, temp_settings
USE mo_io_units,        ONLY: nnml
...
```

```

INTEGER                :: iunit, ist
...
IF (my_process_is_stdio()) THEN
  iunit = temp_defaults()
  ! write defaults to temporary text file
  WRITE(iunit, <yrnlist>_nml)
END IF
CALL open_nml(TRIM(filename))
CALL position_nml('<yrnlist>_nml',STATUS=ist)
SELECT CASE (ist)
CASE (POSITIONED)
  ! read actual values from file
  READ (nnml, <yrnlist>_nml)
  IF (my_process_is_stdio()) THEN
    iunit = temp_settings()
    ! write actual values to temporary text file
    WRITE(iunit, <yrnlist>_nml)
  END IF
END SELECT
CALL close_nml

```

Write actual values for restart: The namelist is stored in a file NAMELIST-<exname>-restart-*{atm,land}* for the atmosphere and the land by the following commands:

Listing 2.22: Store namelist for restart

```

USE mo_mpi,                ONLY: my_process_is_stdio
USE mo_restart_namelist,   ONLY: open_tmpfile, &
                           & store_and_close_namelist
...
INTEGER                    :: funit
...
IF(my_process_is_stdio()) THEN
  funit = open_tmpfile()
  WRITE(funit,NML=<yrnlist>_nml)
  CALL store_and_close_namelist(funit, '<yrnlist>_nml')
ENDIF

```

Fill configuration data structure: The last step is to fill in the configuration data structure <nfeature>_config. This means that the values of all namelist variables <var1>, <var2>, ... have to be assigned one by one to the components of the variable <nfeature>_config of derived type t_<nfeature>. Note that the variables <var1>, <var2>, ... contain either the default values or the values read from the namelist. When you do this assignment, it is a very good idea to test for the validity of the values which may come from the namelist file and be as such erroneous. If erroneous values appear, we have to terminate the program with an appropriate error message. Writing the values of the variables (integer, real, or logical values) to the standard output can be done by the `print_value` subroutine discussed in Sec. 2.2.4. If you like to include values of variables into a longer error message, you have to convert them into a string first. Here is the recipe:

Listing 2.23: Conversion of non-character variables (single values) into a string


```

USE mo_impl_constants ,          ONLY: MAX_CHAR_LENGTH
...
CHARACTER(LEN=MAX_CHAR_LENGTH):: c<var1>
...
WRITE(c<var1>,* ) <var1>

```

The actual error message can be written by the use of the subroutine `finish` that also terminates the ICON run:

```

USE mo_exception ,              ONLY: finish
...
CALL finish('<subprogram>:<module_name>', 'error_message')

```

As the very last step, we have to introduce our new subroutine `read_<yrrlist>_namelist` into the ICON code so that it is executed at a certain point when all the namelists are read. The right place is the subroutine `read_atmo_namelists` of module `SRC/namelists/mo_read_namelists.f90` or an analogue for the ocean or land model. You have to add only two lines to `mo_read_namelist.f90`:

Listing 2.24: Call of reading namelist in ICON

```

USE mo_<yrrlist>_nml ,  ONLY: read_<yrrlist>_namelist
...
CALL read_<yrrlist>_namelist (atm_namelist_filename)
...

```

This minimizes modifications of the original ICON code and helps to keep updates to new ICON versions fairly easy. Where exactly you have to call the reading of your namelist inside `mo_read_namelists.f90` may depend on its possible interference with other namelist.

2.3.3 Representation of 2d- and 3d-fields in ICON, geographical coordinates

We first consider a 2d-field that depends on the geographical position only. There is no obvious order of the cells in a 2d-array 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 datastructure `field` of type `t_field` and create a vector `field(1:ndomain)` such that each field element `field(i)` for domain $i = 1, ndomain$ 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, we would have each domain on one processor and we would have very long arrays at high resolution. The number of cells would be $N_{r_2b_9} \approx 2.097 \times 10^7$ 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 each processor 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's still about 21,000 cells 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`.

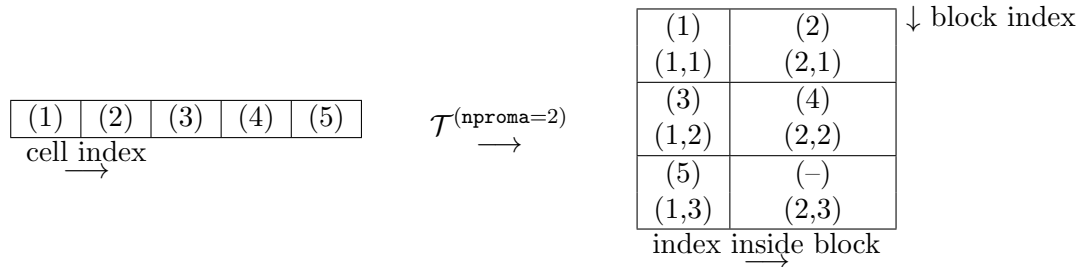


Figure 2.2: Vectorization in ICON

The 3d-fields that were stored in 2d-arrays with the cell index as the first dimension and the second being the vertical coordinate, 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 is that we would like to pass the blocks one by one to some subprograms which are called in a loop over the blocks. In that case, we can cut the block index in these subprograms without any reshape of the arrays.

We will now discuss some important data structures in 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 `ndomain` 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 your processor, respectively.

nblks_{c,e,v}: number of blocks of cells, edges, vertices on this processor, respectively.

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

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 $\Delta_{r_n b_m}$.

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 coordinates 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 datastructure 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 coordinates of the cell centres can be used by

Listing 2.28: Geographical coordinates 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`:

Listing 2.29: Usage of maximum block length `nproma`

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

2.3.4 Data structure 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_domain` elements of the derived type `t_echam_phy_field` for the variables at various time “levels” t , $t + \Delta t$, 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 datastructures:

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
  & qx        (:,:,), &! [kg/kg] total concentration of
    ! hydrometeors
  & 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  $t + \Delta t$ 
  & presm_new (:,:,), &! [Pa]   pressure at full levels  $t + \Delta t$ 
  ...
  & 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_gwh (:,:,) , & ! 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(:,:,,:)` 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, i.e. all parameterized processes `<var>_phy`, 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>_gwh` 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 either giving their names as they appear in subsequent calls to 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 preceded 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 a vector `p_nh_state(1:n_domain)` 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 the 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 measured 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_domain)%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 (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) is different 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 `mpi_phy_nml`, each process can have its individual calling frequency and start and end date and time. We discuss first how to introduce these process specific variables into `SRC/configure_model/mo_mpi_phy_config.f90`. There are two relevant variables (*i*) `mpi_phy_config` of type `t_mpi_phy_config` containing the entries read from the namelist `mpi_phy_nml` and (*ii*) `mpi_phy_tc` of type `t_mpi_phy_tc` containing the same information as `mpi_phy_config` but in a form that can be handled by the `mtime` library.

For each process `<prc>` (all processes have to be identified by three letters), the following three variables must be added as components in the declaration of `t_mpi_phy_config`:

Listing 2.36: Specific components of `t_mpi_phy_config` for a process `<prc>`

```

CHARACTER(len=max_timedelta_str_len) :: dt_<prc>
CHARACTER(len=max_datetime_str_len ) :: sd_<prc>
CHARACTER(len=max_datetime_str_len ) :: ed_<prc>

```

where the component `dt_<prc>` stands for the time interval at which the process will be called, `sd_<prc>` and `ed_<prc>` stand for the start and end date between which the process `<prc>` will be called.

The type `t_mpi_phy_tc` has four corresponding components for each process `<prc>`:

Listing 2.37: Specific components of `t_mpi_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 interval 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 `mpi_phy_config` have to be initialized with empty strings for all domains for each specific process `<prc>` in the subroutine `init_mpi_phy_config` of `mo_mpi_phy_config.f90`:

Listing 2.38: Initialization of `mpi_phy_config`

```
mpi_phy_config(:)%dt_<prc>=' '
mpi_phy_config(:)%sd_<prc>=' '
mpi_phy_config(:)%ed_<prc>=' '
```

These values will be overwritten if they are specified in the `mpi_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_mpi_phy_config_dt_sd_ed` in the subroutine `eval_mpi_phy_config` of `mo_mpi_phy_config.f90` for every domain and every process `<prc>`:

Listing 2.39: Check TI- and DT-variables given by namelist `mpi_phy_nml`

```
CALL eval_mpi_phy_config_dt_sd_ed(TRIM(CG),          '<prc>', &
&                                mpi_phy_config(jg)%dt_<prc>, &
&                                mpi_phy_config(jg)%sd_<prc>, &
&                                mpi_phy_config(jg)%ed_<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_mpi_phy_tc_dt_sd_ed_ev` in `eval_mpi_phy_tc` of `mo_mpi_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_mpi_phy_tc_dt_sd_ed_ev(CG,          '<prc>', &
&                                mpi_phy_config(jg)% dt_<prc>, &
&                                mpi_phy_config(jg)% sd_<prc>, &
&                                mpi_phy_config(jg)% ed_<prc>, &
&                                mpi_phy_tc      (jg)% dt_<prc>, &
&                                mpi_phy_tc      (jg)% sd_<prc>, &
&                                mpi_phy_tc      (jg)% ed_<prc>, &
&                                mpi_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 makes debugging easier. You have two print routines, for the components of `mpi_phy_config` and `mpi_phy_tc`, both called in `print_mpi_phy_config` in a loop over all domains `jg=1,n_dom`:

Listing 2.41: Printing the physics configuration variables

```
CALL print_mpi_phy_config_dt_sd_ed(CG,          '<prc>', &
&                                mpi_phy_config(jg)% dt_<prc>, &
&                                mpi_phy_config(jg)% sd_<prc>, &
&                                mpi_phy_config(jg)% ed_<prc>  )
CALL print_mpi_phy_tc_dt_sd_ed(CG,          '<prc>', &
```

```

&          mpi_phy_tc(jg)% dt_<prc>, &
&          mpi_phy_tc(jg)% sd_<prc>, &
&          mpi_phy_tc(jg)% ed_<prc>  )

```

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

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 according to the DT- and TI-variables 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 in 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 2.42: If clauses to evaluate whether a process has to be called or not

```

IF ( mpi_phy_tc(jg)%dt_<prc> > dt_zero ) THEN
  is_in_sd_ed_interval = &
& (mpi_phy_tc(jg)%sd_<prc> <= datetime_old) .AND. &
& (mpi_phy_tc(jg)%ed_<prc> >  datetime_old)
  is_active = &
& isCurrentEventActive(mpi_phy_tc(jg)%ev_<prc>,datetime_old)
  CALL message_forcing_action('process_<prc>', &
&          is_in_sd_ed_interval, is_active)
  CALL interface_echam_<prc>(is_in_sd_ed_interval, &
&          is_active, &
&          patch, rl_start, rl_end, &
&          field, tend, &
&          datetime_old )
END IF

```

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 2.43: Interface routine for calling a physics process

```

MODULE mo_interface_echam_<prc>
  USE mo_kind          ,ONLY: wp
  USE mo_model_domain  ,ONLY: t_patch
  USE mo_loopindices   ,ONLY: get_indices_c
  USE mo_parallel_config ,ONLY: nprma
  USE mo_run_config    ,ONLY: nlev

  USE mo_echam_phy_memory ,ONLY: t_echam_phy_field,
    t_echam_phy_tend

  IMPLICIT NONE
  PRIVATE
  PUBLIC :: interface_echam_<prc>

```

```

CONTAINS

SUBROUTINE interface_echam_<prc>(is_in_sd_ed_interval,    &
    &                                     is_active,      &
    &                                     patch, rl_start, rl_end, &
    &                                     field, tend,      )

LOGICAL                                ,INTENT(in)      ::
    is_in_sd_ed_interval
LOGICAL                                ,INTENT(in)      :: is_active
TYPE(t_patch)                          ,TARGET ,INTENT(in) :: patch
INTEGER                                ,INTENT(in)      :: rl_start, rl_end
TYPE(t_echam_phy_field)                ,POINTER       :: field
TYPE(t_echam_phy_tend)                 ,POINTER       :: tend

INTEGER :: i_nchdom
INTEGER :: i_startblk,i_endblk
INTEGER :: jb          !< block index
INTEGER :: jcs, jce !< start/end column index within block

i_nchdom = MAX(1,patch%n_childdom)
i_startblk = patch%cells%start_blk(rl_start,1)
i_endblk = patch%cells%end_blk(rl_end,i_nchdom)

DO jb = i_startblk,i_endblk
    CALL get_indices_c( patch,      jb,  i_startblk, &
                       & i_endblk, jcs, jce,      &
                       & rl_start, rl_end      )
    IF ( is_in_sd_ed_interval ) THEN
        IF ( is_active ) THEN
            CALL <prc> (jcs,          jce,      &
                       & nprma,      nlev,    &
                       & <fields>,    <tends> )
        END IF
        ! accumulate tendencies
        tend%<t>(jcs:jce,:,jb) = tend%<t>(jcs:jce,:,jb) + &
            & <tends>(jcs:jce,:,jb)
    END IF
END DO
END SUBROUTINE interface_echam_o3_cariolle
END MODULE mo_interface_echam_o3_cariolle

```

The most important steps are to get the indices of the blocks which have to be handled (*i_startblk* and *i_endblk*). The physics process has then to be called for each of these blocks. The indices of the elements in one block start at *jcs* = 1 and run to *jce*. There should never occur a *jcs* \neq 1, but this may change in the future. The variable *nprma* is the maximum block length and is normally used to declare 2d-arrays which have the dimensions *a(nprma,nlev)*. The first if clause (*IF (is_in_sd_ed_interval)*) inside the loop assures that a tendency is added if and only if we are between the start and end date of the physics process *<prc>*. The ten-

dencies `<tends>` have to be stored over several time steps since they are re-used in subsequent time steps even if the process is not active, i.e. the tendencies are not re-calculated. In fact, that is what the second if clause does: Calculate the process if and only if it is also “active” (IF (`is_active`)), i.e. our actual time step is the time step at which the process has to be re-calculated according to `mpi_phy_config(jg)%dt_<prc>`.

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 parameters 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 2.44: 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

Listing 2.45: `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 2.46: Conversion of DT- and TI-variables from strings 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 2.47: 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 occurring 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 2.48: 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 2.49: 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 2.48. 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 into ICON: Parameter data sets consisting of arrays that have to be known at every grid point, 2d- or 3d-data sets of surface properties or atmospheric composition, or zonal mean values of data that depend on latitude only. Their vertical coordinate may be a pressure coordinate or the data may be given at 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. However, we should think a bit about the problems that may accompany the handling of such data.

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.

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 coordinate demands interpolation. If these data are to be used in the ECHAM–physics part, a vertical pressure coordinate 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: Technically, the easiest way would be to extend and interpolate the zonal mean values to 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, this may then be of the order of several TB. Reading several TB into a parallel program turns out to be a nightmare and this cannot be a sensible method anymore. In that case, interpolation inside ICON is certainly the better option.

I would like to present 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 (typical example are parameter fields). In that case, they are read on an 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 2.50: 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 2.51) 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. 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 netcdf-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's `.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 2.51: 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:ndom)` 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 on an 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` as 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`). The parameter list is as follows:

Listing 2.52: 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 2.53.

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 call is different from the previous case, and the specific function `openInputFile_bcast` is called by this interface.

Listing 2.53: Function `openInputFile` for opening a netcdf file for reading on the i/o processor

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

2.3.8 Defining new “streams”

We already know the datastructures `prm_field` or `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` of type `t_echam_phy_field` has a “cousin” `prm_field_list` of type `t_var_list` that contains a lot of 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 2.54). 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 2.54: 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`. In principle you can search for such a stream by its name, but all these searches include string comparisons

that are very slow compared to floating point operations. This means, that we should avoid such searches except during the initialization phase of ICON. 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 the list. 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, restart files or is a stream the variables of which have to be used for initialization. The `patch_id` indicates the model domain (so the refinement step) to which it belongs. The variable `vlevel_type` indicates whether it is 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 2.55).

Listing 2.55: 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 2.56: 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. What is particularly interesting for us, is the “meta data” information. It contains the following:

```
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.

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 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_domain` such that each element is of type `t_var_list`. Similarly, `prm_field` has `n_domain` 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` that is in each domain `jk`:

```
prm_field_list(jk)%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 `jk` at that position in the program as shown in the following example, respectively:

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

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

```
prm_field_list(jk)%p%first_list_element
```

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

```
prm_field_list(jk)%p%first_list_element%next_list_element
prm_field_list(jk)%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(jk)%p%first_list_element%field%r_ptr(:,:,:,,:)
prm_field_list(jk)%p%first_list_element%field%i_ptr(:,:,:,,:)
prm_field_list(jk)%p%first_list_element%field%l_ptr(:,:,:,,:)
!metadata:
prm_field_list(jk)%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(jk)%p%first_list_element%field%info%name
prm_field_list(jk)%p%first_list_element%field%info%loutput
prm_field_list(jk)%p%first_list_element%field%info%cf
prm_field_list(jk)%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`, so for the real field:

Listing 2.57: Array of $i - 1$ st stream element

```
prm_field_list(jk)%p%first_list_element% $\overbrace{\text{next\_list\_element}\% \dots \% \text{next\_list\_element}}^{i-1 \text{ times}}\%$ 
```

```
%field%r_ptr(:, :, :, :)
```

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

Listing 2.58: 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 2.57 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 memory area under two different access names, the one of Listing 2.57 and the one of Listing 2.58.

Our next task will be to explain how we can build such a linked list and how we may connect it to a “simple datastructure” like the `prm_field`. The “simple datastructure” will be referred to as Fast Access Datastructure (FAD). This datastructure 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_<task>_memory.f90`. In the same module, the stream or linked list must be defined and the connection must be established between the FAD and the linked list. 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_<task>_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 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_<task>_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_<task>_list (patch_array)
TYPE(t_patch), INTENT(IN) :: patch_array(:)
INTEGER :: ndomain, ist
ndomain=SIZE(patch_array)
ALLOCATE(<fad>(ndomain), STAT=ist)
IF (ist/=SUCCESS) THEN
CALL finish('construct_<task>_list_of_<mo_<task>_memory.f90', &
&'allocation_of_<fad>_array_failed')
END IF
ALLOCATE(<fad>_list(ndomain), STAT=ist)
IF (ist/=SUCCESS) THEN
CALL finish('construct_<task>_list_of_<mo_<task>_memory.f90', &
&'allocation_of_<fad>_list_array_failed')
END IF
END SUBROUTINE construct_<task>_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_<task>_list` that will be called in a loop over the domains. This loop will be in `construct_<task>_list`

```

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

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

```

- (v) The new subroutine `new_<task>_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 datastructure `<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 datastructures 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, &
  &          ZA_HYBRID
USE mo_io_config,     ONLY: lnetcdf_flt64_output
USE mo_parallel_config,ONLY: nprma
...
SUBROUTINE new_<task>_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 datastructures 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 = (/nprma, p_patch%nlev, p_patch%nblks_c/)
CALL add_var(<fad>_list, '<varname>', <fad>%<var>, &
            &GRID_UNSTRUCTURED_CELL, ZA_HYBRID, &
            &cf_desc, grib2_desc, &
            &ldims=shape3d, &
            &vert_interp = &
            &create_vert_interp_metadata( &
            &vert_intp_type=vintp_types("P", "Z", "I") )&
            &)
END SUBROUTINE new_<task>_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 in the subroutine `add_var`. Once a default value is assigned, it cannot be changed by a second call to `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 `lnetcdfflt64_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 and spurious questions of your colleagues. 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>` datastructure. Exactly this allows the fast access later because this array will be associated with the array of the linked list. The following two arguments describe the the horizontal grid and the vertical grid. We always use variables on the “unstructured” icosahedral grid, the vertical coordinate is always `ZA_HYBRID` or `ZA_HYBRID_HALF` for the layer interfaces, and `ZA_SURFACE` for 2d- (surface) fields. 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 the program. This includes the deallocation of the linked list and the corresponding `<fad>`:

```
USE mo_var_list, ONLY: delete_var_list
SUBROUTINE destruct_<task>_list
  INTEGER :: ndomain, jg, ist
  ndomain = SIZE(<fad>)
  DO jg = 1, ndomain
    CALL delete_var_list ( <fad>_list(jg) )
  END DO
  DEALLOCATE(<fad>_list, STAT=ist)
  IF (ist/=SUCCESS) THEN
    CALL finish('destruct_<task>_list_of_mo_<task>_memory.f90', &
      &'deallocation_of_<fad>_list_array_failed')
  END IF
  DEALLOCATE(<fad>, STAT=ist)
  IF (ist/=SUCCESS) THEN
    CALL finish('destruct_<task>_list_of_mo_<task>_memory.f90', &
      &'deallocation_of_<fad>_array_failed')
  END IF
END SUBROUTINE destruct_<task>_list
```

- (vii) The last step is to introduce the subroutines `construct_<task>_list` and `destruct_<task>_list` into ICON. Clearly, these routines have to be called out-

side 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_<task>_list` is `SRC/atm_phy_echam/mo_echam_phy_init.f90`. The clean-up would then be in `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. Majewski, 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.
- [3] B. Zängl, D. Reinert, F. Prill, M. Giorgetta, L. Kornblueh, L. Linardakis, S. Müller, and S. Rast. ICON user’s guide. Report, Deutscher Wetterdienst, Max Planck Institute for Meteorology, Karlsruhe Institute of Technology, Hamburg, 2016.

Listings

1.1	Archive file of the ICON model	2
1.2	Generation of run scripts from basic run file and experiment file	12
1.3	Basic SLURM commands to submit jobs	12
1.4	Example for giving an individual frequency to the radiation call	21
1.5	Example for giving start and end date and an individual frequency to the radiation call	22
2.1	Declaration of real variables	30
2.2	Modules in ICON	30
2.3	type statement	31
2.4	type	31
2.5	components	31
2.6	Derived type of a vectorfield	32
2.7	Usage of “netsted” derived types	32
2.8	Passing derived types into subprograms: calls of subroutines	32
2.9	Passing derived types into subprograms	33
2.10	Recursive data types	33
2.11	Variables of type tracer to generate a linked list	33
2.12	Linked list of tracers	34
2.13	Use statement for the extensions of various operators for DT-variables	36
2.14	Usage of type <code>datetime</code>	36
2.15	Usage of extended operators for DT-variables	37
2.16	Testing of the ICON code — <code>exp.atm.amip.test</code>	38
2.17	The <code>message</code> subroutine to output messages and continue the execution of the ICON code	39
2.18	The <code>finish</code> subroutine to print a message and stop the ICON program	40

2.19	Namelist module – Declaration of namelist	40
2.20	Read namelist in restart	41
2.21	Read namelist in initial run	41
2.22	Store namelist for restart	42
2.23	Conversion of non-character variables (single values) into a string	42
2.24	Call of reading namelist in ICON	43
2.25	Grid information as stored in <code>p_patch</code>	44
2.26	Type for geometric information <code>t_grid_geometry_info</code>	45
2.27	Information about grid cells provided by the type <code>t_grid_cells</code>	46
2.28	Geographical coordinates of cell centres	46
2.29	Usage of maximum block length <code>nproma</code>	46
2.30	Variables describing the state of (ECHAM) physics in ICON	47
2.31	Components of <code>prn_field</code> and <code>prn_tend</code> all at t if not stated differently	47
2.32	Type <code>t_nh_state</code> for the description of the state of the nonhydrostatic atmosphere	48
2.33	Type <code>t_nh_prog</code> that hosts the prognostic variables	49
2.34	Density of the atmosphere as state variable of the nonhydrostatic dynamic core	49
2.35	Type <code>t_nh_diag</code> containing diagnostic variables from the dynamics	49
2.36	Specific components of <code>t_mpi_phy_config</code> for a process <code><prc></code>	50
2.37	Specific components of <code>t_mpi_phy_tc</code> for a process <code><prc></code>	50
2.38	Initialization of <code>mpi_phy_config</code>	51
2.39	Check TI- and DT-variables given by namelist <code>mpi_phy_nml</code>	51
2.40	Conversion of TI- and DT-variables into <code>mtime</code> compatible format for process <code><prc></code>	51
2.41	Printing the physics configuration variables	51
2.42	If clauses to evaluate whether a process has to be called or not	52
2.43	Interface routine for calling a physics process	52
2.44	Maximum string lengths of DT- and TI-variables	54
2.45	<code>mtime</code> library compatible format of DT- and TI-variables	54
2.46	Conversion of DT- and TI-variables from stings into <code>mtime</code> library compatible format	54
2.47	Conversion of <code>mtime</code> library compatible variables into strings	54
2.48	Data type for time interpolation weights	55

2.49	Calculation of time interpolation weights	55
2.50	Reading a time-dependent 3d-data field on the ICON grid	56
2.51	Function <code>openInputFile</code> for opening a netcdf file for distributed read	57
2.52	Read a general field and send it to all processors	58
2.53	Function <code>openInputFile</code> for opening a netcdf file for reading on the i/o processor	59
2.54	Type <code>t_var_list</code> for “stream” information	59
2.55	Types <code>t_list_element</code> and <code>t_var_list_element</code> for linked lists	60
2.56	Type <code>t_var_list_element</code> containing information about individual list elements	60
2.57	Array of i – 1st stream element	62
2.58	Zonal wind as member of <code>prm_field</code>	63