Consortium

for

Small-Scale Modelling

Technical Report No. 20

Tracer module in the COSMO model

by

Anne Roches and Oliver Fuhrer

December 2012

Deutscher Wetterdienst
MeteoSwiss
Ufficio Generale Spazio Aereo e Meteorologia
ΕΘΝΙΚΗ ΜΕΤΕΩΡΟΛΟΓΙΚΗ ΥΠΗΡΕΣΙΑ
Instytucie Meteorogii i Gospodarki Wodnej
Administratia Nationala de Meteorologie
ROSHYDROMET
Agenzia Regionale per la Protezione Ambientale del Piemonte
Agenzia Regionale per la Protezione Ambientale dell' Emilia-Romagna
Centro Italiano Ricerche Aerospaziali
Amt für GeoInformationswesen der Bundeswehr

www.cosmo-model.org

Editor: Massimo Milelli, ARPA Piemonte

Tracer module in the COSMO model

Anne Roches**[*] **and Oliver Fuhrer**[**]

[*]C2SM
Universitätstrasse 16
Zürich
Switzerland

[**]MeteoSwiss
Krähbühlstrasse 58
Zürich
Switzerland

Revision	Modifications	Date	Author
1.0	Initial Release	December 2012	Anne Roches (C2SM) Oliver Fuhrer (MeteoSwiss)

Contents

Abstract

A tracer module is implemented in the COSMO model in version 5.0 and higher. This model component offers a set of functionalities commonly required for the treatment of new prognostic variables (tracers) like memory management, input and output support or the computation of advective transport. It allows for a simplified introduction of tracers into the COSMO code, for a coherent treatment of tracers and for a well-structured code without redundancies. Application of this module may range from the tracing of air masses to the handling of microphysics species or aerosols.

1 Introduction

Water in all phases and various isotopic forms, volatile chemical compounds, radionuclides or aerosols are all important components of our atmosphere. The adequate representation of these tracers is thus a crucial theme for atmospheric modeling.
We use the term *tracer* to designate any substance present in the atmosphere as a trace constituent and which can be transported passively, i.e. without influencing the flow (except if the user chooses to implement a coupling term).

A consistent and flexible treatment of tracers in atmospheric models becomes increasingly important as the complexity of these models augments by the introduction of new submodules describing additional processes and implying the consideration of additional species and even including other environmental compartments. A clear trend in the extension of atmospheric models towards Earth's system models has been observed in the past decades. All atmospheric models which have made important steps into this direction handle tracers in general modules providing basic functionalities like memory management or advective transport to the submodules (hereafter called tracer clients) (e.g. Rast, 2009). Example of tracer clients are the microphysics, atmospheric chemistry modules, etc.

Traditionally, adding a new tracer variable into COSMO required extensive and tedious code modifications, since all tracers were treated as individual variables. The result was abundant code redundancy, bugs and inconsistencies in the treatment of the individual tracers. The general tracer module described here should facilitate extensions of the COSMO model by providing a comprehensive API (Application Programming Interface) to the tracer client. Consistency in the tracer treatment is promoted through the unity of the code describing a given process (e.g. advection). This also simplifies code maintenance and introduction of new features.

In this technical report a general description of the new tracer module can be found in section 2. A short user's guide explains how to use the tracer module by showing concrete examples (section 3) whereas section 4 provides some technical details on the implementation for model developers. Section 5 presents the results obtained with the tracer enabled version. Issues that might arise with this version are discussed in section 6 and conclusions are drawn in section 7.

2 Description of the tracer module

The tracer module consists in four new source code files, `src_tracer.f90`, `data_tracer.f90`, `src_tracer_metadata.f90`, `data_tracer_metadata.f90`. Extensive modifications to the existing COSMO source code files had to be performed as well, changing the programming paradigm based on individual variables for an automated treatment of the species (by looping over all tracers), in order to bring the required functionalities (see section 2.2) in an automated form for the tracers.

The tracer API is a collection of few subroutines and functions that can be called by the tracer client from any part of the COSMO code. They serve to define and retrieve tracers as well as their corresponding metadata (attributes storing information about the tracers). All subroutines and functions which can be called by the tracer client are located in the source file `src_tracer.f90` and are described in detail in section 2.3. Useful named constants are made available in the `data_tracer.f90` module. The source code files `src_tracer_metadata.f90` and `data_tracer_metadata.f90` contain low-level functions for handling metadata and are not called directly by the tracer client.

Basic information about the tracer (metadata) such as its name or the operations it has to undergo (e.g. advection, turbulent mixing, etc.) has to be passed to the tracer module upon definition of a new tracer. This is the so-called set of standard metadata. A short description of the mandatory standard metadata and of the optional standard metadata is provided in table 1 and table 2 respectively. No default value is provided for the mandatory metadata whereas the default value is used for the optional standard metadata if the user provides no specification of these items.

Table 1: Basic set of standard metadata: mandatory items

Identifier	**Type**	**Definition**	**Default value**
`T_NAME_ID`	`CHAR`	Tracer name	-
`T_GRBPARAM_ID`	`INT`	GRIB parameter number	-
`T_GRBTABLE_ID`	`INT`	GRIB table number	-
`T_PARENT_ID`	`CHAR`	Parent subroutine name (subroutine which defines the tracer)	-
`T_UNITS_ID`	`CHAR`	Units	-

Table 2: Basic set of standard metadata: optional items

Identifier	**Type**	**Definition**	**Default value**
T_NCSTDNAME_ID	CHAR	Standard name	"undefined"
T_NCLONGNAME_ID	CHAR	Long name	"undefined"
T_ADV_ID	INT	Do advection? Options: - T_ADV_OFF: no advection - T_ADV_ON: advection	T_ADV_OFF
T_DIFF_ID	INT	Do horizontal hyperdiffusion? Options: - T_DIFF_OFF: no horiz. hyperdiffusion - T_DIFF_ON: horiz. hyperdiffusion	T_DIFF_OFF
T_TURB_ID	INT	Do turbulent mixing? Options: - T_TURB_OFF: no turbulent mixing - T_TURB_1D: vertical turb. mixing - T_TURB_3D: 3-dimensional turb. mixing	T_TURB_OFF
T_CONV_ID	INT	Do passive convective transport? Options: - T_CONV_OFF: no passive convective transp. - T_CONV_ON: passive convective transp.	T_CONV_OFF
T_INI_ID	INT	Type of initial condition Options: - T_INI_ZERO: initialize to zero - T_INI_FILE: initialize using data from file - T_INI_USER: user defined	T_INI_ZERO
T_LBC_ID	INT	Type of lateral boundary condition Options: - T_LBC_ZERO: zero value - T_LBC_FILE: values read from file - T_LBC_CST: constant value - T_LBC_ZEROGRAD: zero-gradient condition - T_LBC_USER: user defined	T_LBC_ZERO
T_BBC_ID	INT	Type of bottom boundary condition Options: - T_BBC_ZEROFLUX: zero flux condition - T_BBC_ZEROVAL: zero value condition - T_BBC_SURF_VAL: values provided in a surface field	T_BBC_ZEROFLUX

Table 2: (continued)

Identifier	**Type**	**Definition**	**Default value**
T_RELAX_ID	INT	Do boundary relaxation? Options: - T_RELAX_OFF: no relaxation - T_RELAX_FULL: relaxation at all boundaries - T_RELAX_INFLOW: relaxation at inflow boundary only	T_RELAX_OFF
T_DAMP_ID	INT	Do Rayleigh damping? Options: - T_DAMP_OFF: no damping - T_DAMP_ON: damping	T_DAMP_OFF
T_CLP_ID	INT	Do clipping? Options: - T_CLP_OFF: no clipping - T_CLP_ON: clipping	T_CLP_OFF

2.1 Assumptions

Some basic assumptions have been made about the tracers in order to keep a concise tracer API and ensure a uniform tracer treatment.

The tracers are scalar, 3-dimensional prognostic (i.e. time dependent) variables defined on the full model grid. They are defined along the three main spatial dimensions (rotated longitude `rlon`, rotated latitude `rlat`, model level `level`) of the COSMO grid. The tracers live at the mass-point (center of the grid box) collocated with temperature and pressure (`rlon, rlat, level`). It is not possible to have tracers on the staggered positions of the Arakawa-C/Lorenz COSMO grid (Schättler et al. 2011 or Doms 2011).

2.2 Functionalities of the tracer module

All basic functionalities which are common to all tracers as for example the allocation of the memory and transport by advection are provided. In a broad sense, the tracer module handles the infrastructure operations (related to the definition, the storage and the input/output of a variable) and the operations related to what is referred in COSMO as the "dynamics" (Schättler et al., 2011). It does not handle processes which are specific to a tracer or to a tracer class (basically the operations belonging to the "physics" according to the COSMO vocabulary or to additional submodules).

The prognostic equation for a tracer can be derived from the prognostic equation for water vapor (equation 3.148 in Doms, 2011):

$$\frac{\partial q^v}{\partial t} = \underbrace{-\left\{\frac{1}{acos\varphi}\left(u\frac{\partial q^v}{\partial \lambda} + vcos\varphi\frac{\partial q^v}{\partial \varphi}\right)\right\} - \dot{\zeta}\frac{\partial q^v}{\partial \zeta}}_{ADV} \underbrace{-\ (S^l + S^f)}_{SRC/SINKS} \underbrace{+ M_{q^v}}_{MIX} \tag{1}$$

Expanding the third term (MIX) of equation (1) using its definition (equation 3.153 in Doms, 2011) and reordering the terms, we get:

$$\frac{\partial q^v}{\partial t} = \underbrace{ADV}_{A} + \underbrace{DIFF}_{B} + \underbrace{TURB}_{C} + \underbrace{CONV}_{D} + \underbrace{RELAX}_{E} + \underbrace{DAMP}_{F} + \underbrace{SRC/SINKS}_{G} \qquad (2)$$

The terms in equation (2) are:

- A: horizontal and vertical advection (as described in chapters 4.4.2 and 4.4.3 of Doms, 2011)
- B: horizontal hyperdiffusion (or computational mixing) (as described in chapter 6.2 of Doms, 2011)
- C: turbulent mixing (as described in chapter 4.3.3 of Doms, 2011 and in chapter 3 of Doms et al., 2011)
- D: moist convection (as described in chapter 6 of Doms et al., 2011)
- E: boundary relaxation (as described in chapter 5.2 of Doms, 2011)
- F: Rayleigh damping (as described in chapter 6.4 of Doms, 2011)
- G: sources and sinks (in the case of qv, microphysical sources and sinks from/to the liquid and solid phases)

Written for an arbitrary tracer, equation (2) becomes:

$$\frac{\partial \psi}{\partial t} = \underbrace{ADV}_{A} + \underbrace{DIFF}_{B} + \underbrace{TURB}_{C} + \underbrace{CONV}_{D} + \underbrace{RELAX}_{E} + \underbrace{DAMP}_{F} + \underbrace{SRC/SINKS}_{G} \qquad (3)$$

Equation (3) is a general equation valid for any tracer ψ. The tracer module takes care of the 6 first terms (A to F). It means that these terms can be computed in an automated way by the tracer module. The tracer client simply has to set correctly the corresponding metadata (table 2). The 7th term, the sources and the sinks, is not computed by the tracer module, since this term is specific for each tracer. The tracer module simply makes available a tendency field that can be used to store the sources and the sinks. It is the responsibility of the tracer client to fill this term with the sources and the sinks appropriated for its tracer. Sources and sinks might be emissions, chemical transformations, phase changes or fluxes from/to the soil for instance.

We describe the functionalities of the tracer module in more details in the following subsections. If a standard metadata controls a functionality, it is added in brackets after the subsection title. The possible values for the standard metadata are listed in table 2.

2.2.1 Tracer definition and memory management

The definition of a new tracer is done by a call to the tracer API (described in section 2.3). Memory required by the tracer, its boundary values and its tendency field is handled internally by the tracer module. Pointers to these data fields can be retrieved by the tracer client by calling the tracer API.

2.2.2 I/O methods

In order to Input/Output (I/O) a field in COSMO, the variable has to have a corresponding entry in the `var` structure in the subroutine `setup_vartab`. The tracer module now automatically does this operation. An entry in the `var` structure is performed for the tracer and this tracer no longer appears explicitly in the code of the subroutine `setup_vartab`.

Any tracer can thus be read from and/or written out to file. However, associated fields like the tendency or a possible surface field cannot be read or written out in an automated way for the moment. This is further discussed in section 6.

The tracers can either be read/written in GRIB format or in NetCDF format like any other field in COSMO. Output is possible on model levels (`ml`), levels of constant pressure (`pl`) or levels of constant height (`zl`).

The relevant configuration switches are the following ones (namelist `INPUT_IO`):

- `yform_read`: determines the input format (GRIB or NetCDF)
- `yform_write`: determines the output format (GRIB or NetCDF)
- `yvarml`, `yvarpl`, `yvarzl`: determines the list of variables to output on the model levels, pressure levels and height levels, respectively

In order to write a tracer out, it must simply be added to the corresponding output list (`yvarml`, `yvarpl`, `yvarzl`). In order to read a tracer from file (as initial or/and as boundary condition), the metadata `T_INI_ID` and `T_LBC_ID` have to be set to `T_INI_FILE` and `T_LBC_FILE`, respectively. The model then automatically searches for the corresponding fields in the initial and boundary condition files and stops execution should they be missing.

2.2.3 Advection (`T_ADV_ID`)

The type of advection used for scalar quantities depends on the dynamical core selected by the user and on namelist switches that further refine the choice of a specific algorithm used for advection. The tracer module currently implements all major existing options and additionally offers the possibility to turn off advective transport for each tracer separately (e.g. for idealized cases or sensitivity studies).

For the Leapfrog core, the standard advection scheme for tracers is a second-order centered difference scheme in the horizontal and an implicit scheme in the vertical. Although certain species are transported using alternative advection schemes (e.g. semi-Lagrangian advection for precipitating microphysics species), the tracer module only implements the standard advection scheme (second order centered difference) for the tracers.

In the Runge-Kutta dynamical core a namelist switch allows a global selection (valid for all tracers) of the advection scheme. A complete list of the schemes is given below and a description of these schemes is provided in Doms (2011).

The relevant configuration switches are the following ones (namelist `INPUT_DYN`):

- `l2tls`: determines if the Leapfrog dynamical core or the Runge-Kutta one should be used
- `y_scalar_advect`: determines which scheme should be used in case of the Runge-Kutta core (van Leer, PPM, Bott2, Bott4, Van Leer Strang, PPM Strang, Bott2 Strang, Bott 4 Strang, Semi-Lagrange 3 MF, Semi-Lagrange 3 SFD)

The tracer module reproduces the advection as it is coded in the original code. Bottom and lateral boundary conditions used in this code part may not be consistent with the choices made by the tracer client (`T_LBC_ID` and `T_BBC_ID`).

2.2.4 Horizontal hyperdiffusion (`T_DIFF_ID`)

Horizontal hyperdiffusion is an artificial diffusion (computational mixing) performed to smooth the solution of the thermodynamical equations obtained by numerical integration. Two different schemes are available in the COSMO model (Doms, 2011) and both have been implemented for the tracers. Horizontal diffusion can also be switched off globally or for each tracer separately (see table 2).

The tracer module reproduces the horizontal diffusion as it is coded in the original code. Lateral boundary conditions used in this code part may not be consistent with the choice made by the tracer client (`T_LBC_ID`).

The relevant configuration switches are the following ones (namelist `INPUT_DYN`):

- `lhordiff`: determines globally if horizontal diffusion should be activated
- `itype_hdiff`: determines which scheme should be used (4th order linear or 4th order linear monotonic with orographic limiter)
- `hd_corr_trcr_in`: determines the correction factor for scalar (tracers) used for the diffusion in the domain
- `hd_corr_trcr_bd`: determines the correction factor for scalar (tracers) used for the diffusion at the boundaries

2.2.5 Turbulent mixing (`T_TURB_ID`)

Different options are available for the turbulent mixing of tracers. Most of them compute the diffusion only in the vertical direction while an option enables to compute additionally the horizontal diffusion. The tracer module only implements the widely used options (i.e. prognostic TKE-based scheme with an implicit treatment using either a Dirichlet or a Neumann boundary condition at the Earths surface) and also the new option to have a 3-dimensional

turbulent mixing. The option to switch off turbulent mixing for each tracer individually is made available.

The relevant configuration switches are the following ones (namelist `INPUT_PHY`):

- `ltur`: determines globally if turbulent mixing should be activated
- `itype_turb`: determines the parameterization used for turbulent diffusion (only value 3 is accepted for the tracers)
- `imode_turb`: determines the mode (implicit vs. explicit) of turbulent diffusion (only value 0 or 1 are accepted for the tracers)
- `l3dturb`: determines if horizontal diffusion should be computed in addition

The type of lower boundary condition is relevant for the turbulent mixing. This condition can be set using the metadata `T_BBC_ID` (see table 2).

More information about the turbulent mixing in the COSMO model can be found in Doms et al. (2011) and Buzzi (2008).

2.2.6 Passive convective transport (`T_CONV_ID`)

Passive transport of scalar quantities by convection was previously not available in the COSMO model. For the tracers, passive transport by convection using the Tiedtke scheme is implemented. It is done in an analogous way as for the momentum components in COSMO and follows ECMWF (2011). Other convection schemes available in COSMO (Kain-Fritsch, Bechtold, shallow-convection based on Tiedtke) are not implemented for the tracers and thus cannot be selected for the tracers. It is possible to activate passive transport by convection for each tracer individually.

The relevant configuration switches are the following ones (namelist `INPUT_PHY`):

- `lconv`: determines globally if subgrid-scale convection should be activated
- `itype_conv`: determines the parameterization used for subgrid-scale convection (only value 0 is accepted for the tracers)

2.2.7 Lateral boundary conditions (`T_LBC_ID`)

Different types of lateral boundary condition (BC) can be applied to the tracers (see table 2). If boundary data coming from a coarser model are available in an input file, these data can be used as boundary condition. If such data are lacking for a tracer, other values can be used at the boundaries. One can choose between a value of zero at all boundaries, a constant value (which is defined upon initialization) or a zero-gradient method, in which the values of the first line of the computing domain are chosen as boundaries. Additionally, the tracer client can also define the boundary values itself.

There is no relevant configuration switch (except for the microphysics species).

2.2.8 Initial condition (T_INI_ID)

Different types of initialization can be applied for the tracers (see table 2). If initialization data coming from a coarser model or from an analysis are available in an input file, these data can be used as initial condition (IC). If such data are lacking for a tracer, an initialization using a value of zero or using a value/function defined by the user can be chosen.

There is no relevant configuration switch (except for the microphysics species).

2.2.9 Boundary relaxation (T_RELAX_ID)

Boundary relaxation is performed in the COSMO model to gradually blend the solution of the model with the boundary values and thus avoid abrupt transitions in the fields and possible contamination of the solution due to numerical noise created by reflection at the boundaries. The scheme used is described in Doms (2011).
Boundary relaxation for each tracer can either be switched off (e.g. in case of idealized studies), can be performed at all boundaries or only at the inflow boundaries (like it is often done for precipitation quantities in COSMO.
Boundary relaxation, even if specified, is not performed in case of constant or zero-gradient lateral boundary conditions.

The relevant configuration switches are the following ones (namelist INPUT_DYN):

- `rlwidth`: specifies the width of the relaxation zone (must be greater than 0 to perform relaxation)

2.2.10 Rayleigh damping (T_DAMP_ID)

Rayleigh damping may be applied in the COSMO model to avoid the reflection of gravity waves, which might occur at the top of the model due to its formulation as a rigid lid. Two types of damping are available in COSMO and for the tracers: damping by relaxing against the boundary fields or by relaxing against the filtered forecast fields (Doms, 2011). Additionally damping can be activated/deactivated on an individual basis for each.
Damping using a relaxation against the boundary fields is not performed, even if specified, in case of constant or zero-gradient lateral boundary conditions.

The relevant configuration switches are the following ones (namelist INPUT_DYN):

- `lspubc`: determines globally if Rayleigh damping should be performed in the upper layers
- `itype_spubc`: specifies the type of Rayleigh damping
- `rdheight`: specifies the height of the damping layer bottom in meters (must be smaller than the top of the model domain to perform damping)

2.2.11 Clipping (T_CLP_ID)

Several clippings are performed throughout the COSMO code to ensure positive values for positive definite variables. Clipping removes thus negative values in order to ensure positive definiteness. We also offer this functionality for the tracers. It is possible to switch on or off the clipping for each tracer independently.

There is no relevant configuration switch.

Remark: For some species, different types of clipping are used at different locations in the COSMO code. The tracer module does not use different clipping types but consistently use the same method across the whole code.

2.2.12 COSMO options not implemented

Some existing schemes/configurations available in the COSMO model are not implemented for the tracers. These schemes/configurations have been considered as not relevant, outdated or as a major violation of a coherent tracer treatment. It is the case of:

- Semi-Lagrange advection in the Leapfrog dynamical core
- Positive definite advection scheme using Euler time stepping in the Leapfrog dynamical core
- Turbulent mixing in case $itype\_turb \neq 3$ and $imode\_turb \leq 2$
- Convective tracer transport for schemes other than the Tiedtke scheme (Kain-Fritsch, shallow convection, Bechtold)
- Special clipping types (e.g. massflux correction scheme)

2.3 The tracer module API

The tracer module API consists of the subroutines and functions stored in the source file `src_tracer.f90` as well as some predefined constants in the source file `data_tracer.f90`. All tracer module subroutine names start with `trcr_`. They provide methods to the tracer client for handling a specific tracer and its associated metadata or for handling a metadata for all tracers, and also some facilities (e.g. memory management), which are not specific to one tracer client.

A general feature of the tracer module is the ability to access any tracer or metadata by either its name or its index. The second option is much faster since it does not imply string comparison and is also the method to use in loops over all tracers (or over a subset of them).

All tracer module subroutines return an error status code (`ierr`) as first argument. Any non-zero value indicates an error. A function which translates this error code into a human readable string is provided (see `trcr_errorstr` below). It is the tracer client's responsibility to check the status code returned and handle the error gracefully.

2.3.1 Methods for tracer handling

The methods described in this section are the ones the tracer client will use in order to define, access, modify tracers from the client code.

trcr_new This subroutine serves to define a new tracer. It must be called by the tracer client each time a new tracer has to be added to the COSMO model. It should be called from a meaningful location in the code (generally from an `organize_xxx` subroutine). Once the memory has been allocated by the tracer module (see `trcr_alloc` below) no more tracers can be defined.

The subroutine arguments are summarized in table 3 (in the correct order). Mandatory arguments have to be specified in each call of `trcr_new`. The other arguments can be passed or not. If they are not passed, the default value of the corresponding metadata is used (see table 2). Many arguments directly correspond to a metadata of the new tracer. Where this is the case, the corresponding metadata identifier is given in parenthesis after the definition of the argument.

A unique short name for the tracer has to be provided. It serves to identify the tracer and, at the same time, is the variable name for the NetCDF format (Unidata, 2011). The units have to be specified and should follow the SI system (International System of Units) if possible. These are the units used for reading and writing out the tracer. A GRIB parameter number as well as a GRIB table number have to be given too. For standard meteorological variables, a specific GRIB number is dedicated to a specific variable. For some tracers, it might be more complicated since the WMO (World Meteorological Organization) has not attributed a fixed number to them. In that case, a GRIB parameter number and GRIB table number which are not yet occupied have to be chosen. The owner (or parent) name, i.e. the name of the module which is in charge of the tracer, has to be passed as well.

It is also strongly recommended to specify a standard name and a long name for the tracer. This should be done with respect to the CF conventions (Eaton et al., 2011) and a table of possible standard names is available on the Lawrence Livermore National Laboratory web-page[1].

Optionally, it is also possible to get back a unique index which can be used to identify the tracer (as a replacement for the tracer name in the `trcr_get` subroutine).

We also encourage to specify all other optional arguments of `trcr_new` which determine the behavior of the tracer. The possible values for the arguments are summarized in table 2.

[1] http://cf-pcmdi.llnl.gov/documents/cf-conventions/1.6/cf-conventions.pdf

Table 3: Aguments for `trcr_new`

Argument	**Definition**	**Type**	**Intent**	**Mandatory?**
`ierr`	Error status	`INT`	`OUT`	Yes
`yshort_name`	Tracer short name (`T_NAME_ID`)	`CHAR`	`IN`	Yes
`igribparam`	Tracer GRIB parameter number (`T_GRBPARAM_ID`)	`INT`	`IN`	Yes
`igribtable`	Tracer GRIB table number (`T_GRBTABLE_ID`)	`INT`	`IN`	Yes
`yparent`	Name of tracer owner (`T_PARENT_ID`)	`CHAR`	`IN`	Yes
`yunits`	Tracer units (`T_UNITS_ID`)	`CHAR`	`IN`	Yes
`ystandard_name`	Tracer standard name (`T_NCSTDNAME_ID`)	`CHAR`	`IN`	No
`ylong_name`	Tracer long name (`T_NCLONGNAME_ID`)	`CHAR`	`IN`	No
`itype_adv`	Advection specification (`T_ADV_ID`)	`INT`	`IN`	No
`itype_diff`	Horizontal hyperdiffusion specification (`T_DIFF_ID`)	`INT`	`IN`	No
`itype_turbmix`	Turbulent mixing specification (`T_TURB_ID`)	`INT`	`IN`	No
`itype_passconv`	Passive transport by convection specification (`T_CONV_ID`)	`INT`	`IN`	No
`itype_ini`	IC specification (`T_INI_ID`)	`INT`	`IN`	No
`itype_lbc`	Lateral BC specification (`T_LBC_ID`)	`INT`	`IN`	No
`itype_bbc`	Bottom BC specification (`T_BBC_ID`)	`INT`	`IN`	No
`itype_relax`	Relaxation specification (`T_RELAX_ID`)	`INT`	`IN`	No
`itype_damp`	Rayleigh damping specification (`T_DAMP_ID`)	`INT`	`IN`	No
`itype_clip`	Clipping specification (`T_CLP_ID`)	`INT`	`IN`	No
`idx_trcr`	Tracer index	`INT`	`OUT`	No

trcr_get This subroutine enables the tracer client to retrieve a pointer to the tracer in order to perform specific operations (e.g. microphysics, atmospheric chemistry, etc.). The subroutine has to be called in each module or each subroutine (usually once per module should be enough) for each tracer and each time level required.

The subroutine arguments are summarized in table 4 (in the correct order).

Table 4: Aguments for `trcr_get`

Argument	**Definition**	**Type**	**Intent**	**Mandatory?**
`ierr`	Error status	`INT`	`OUT`	Yes
`idx_trcr`	Tracer index	`INT`	`IN`	Yes
or				
`yname`	Tracer name	`CHAR`	`IN`	Yes
`ptr`	Pointer to the tracer data	`REAL, POINTER`	`OUT`	No
`ptr_bd`	Pointer to the tracer boundary data	`REAL, POINTER`	`OUT`	No
`ptr_tens`	Pointer to the tracer tendency data	`REAL, POINTER`	`OUT`	No
`ptr_tlev`	Time level for the pointers	`INT`	`IN`	No

trcr_get_ntrcr This function returns the total number of tracers currently handled by the tracer module (table 5). It has no argument. This function is used to create the upper bound of the loops over all tracers for instance (e.g. `DO iztrcr= 1, trcr_get_ntrcr()`).

Table 5: Aguments for `trcr_get_ntrcr`

Argument	**Definition**	**Type**	**Intent**	**Mandatory?**
result	Number of tracers	`INT`	-	-

trcr_get_index This subroutine returns the unique index identifying the tracer given its name.
The subroutine arguments are summarized in table 6 (in the correct order).

Table 6: Aguments for `trcr_get_index`

Argument	**Definition**	**Type**	**Intent**	**Mandatory?**
`ierr`	Error status	`INT`	`OUT`	Yes
`yname`	Tracer name	`CHAR`	`IN`	Yes
`idx`	Tracer index	`INT`	`OUT`	Yes

trcr_errorstr This function returns a meaningful error message for any error code issued by the tracer module. The input argument is the error code (see table 7). This error number can come from any subroutine of the tracer API listed in this section.

Table 7: Aguments for `trcr_errorstr`

Argument	Definition	Type	Intent	Mandatory?
`ierr`	Error status	`INT`	`IN`	Yes
result	Error message	`CHAR`	-	-

2.3.2 Methods for metadata handling

A tracer metadata describes a property of a tracer substance, link an associated field to it or determines a specific behavior of it. Apart from the standard set of metadata every tracer has (see tables 1 and 2), the tracer module allows the tracer client to define, set and retrieve arbitrary metadata for each tracer substance. This section describes the API methods available to the tracer client for handling the metadata associated with its own tracers. A metadata can be a single `integer`, `real`, `double`, `character`, `logical` as well as a one-dimensional array of any of these types. A metadata can also be a single two-, three- or four-dimensional pointer, pointing to an array of the same data type as the tracer fields (`ireals`). The methods for handling metadata are overloaded and the tracer client has a single interface for all of the above types. The interfaces are described in more details below. For simplicity reasons, we do not mention `real` and `double` for each interface but only `real` in this documentation.

trcr_meta_define Define a new metadata and provide its default value. The subroutine arguments are summarized in table 8 (in the correct order).

Table 8: Aguments for `trcr_meta_def`

Argument	Definition	Type	Intent	Mandatory?
`ierr`	Error status	`INT`	`OUT`	Yes
`yname`	Metadata name	`CHAR`	`IN`	Yes
`ydefault`	Default metadata value	`INT`	`IN`	Yes
or				
”	”	`REAL`	”	”
or				
”	”	`DOUBLE`	”	”
or				
”	”	`CHAR`	”	”
or				
”	”	`LOGICAL`	”	”
or				
”	”	`POINTER`	”	”
`iidx`	Metadata index	`INT`	`OUT`	No
`protect`	Flag to protect the metadata from possible deletion	`LOGICAL`	`IN`	No

The name of the metadata has to be provided as well as a default value. The default value

will be applied to the metadata for all tracers, except if another value is specified for a given tracer (using `trcr_meta_set`, see below). The type of the default value determines the type of the metadata.
Be careful when defining a pointer metadata. The default value has to be a pointer (generally a null pointer) of the type `REAL(KIND=ireals)` and of the correct dimension.

Optionally, the index of the metadata in the metadata structure can be obtained for more efficient access and it is also possible to protect the metadata from deletion by another tracer client.

trcr_meta_set Store values for a given metadata (identified by its name or index) for a specific tracer or for all tracers at once. The subroutine arguments are summarized in table 9 (in the correct order).

Table 9: Aguments for `trcr_meta_set`

Argument	**Definition**	**Type**	**Intent**	**Mandatory?**
`ierr`	Error status	`INT`	`OUT`	Yes
`itrcr`	Tracer index	`INT`	`IN`	Yes
or				
`ytrcr`	Tracer name	`CHAR`	`IN`	Yes
or				
nothing				
`imeta`	Metadata index	`INT`	`IN`	Yes
or				
`ymeta`	Metadata name	`CHAR`	`IN`	Yes
`ydata`	Specified value	`INT`	`IN`	Yes
or				
”	”	`REAL`	”	”
or				
”	”	`CHAR`	”	”
or				
”	”	`LOGICAL`	”	”
or				
”	”	`POINTER`	”	”

The name or the index of the metadata to set has to be provided as well as the value to store. If a tracer name or a tracer index is specified, then the value given will be used only for this tracer. If the tracer name or index is omitted, it is assumed that the tracer client wants to store metadata for all tracers and the last dimension of the value array needs to correspond to the number of tracers. The values have to be of the same type (`INT`, `REAL`, `CHAR`, `LOGICAL`, `POINTER`) than the default value provided to `trcr_meta_define`. In case of pointers, the dimension has to correspond to the dimension of the default value provided to `trcr_meta_define`.

trcr_meta_get Retrieve a given metadata for a specific tracer or for all tracers at once. The subroutine arguments are summarized in table 10 (in the correct order).

Table 10: Aguments for `trcr_meta_get`

Argument	Definition	Type	Intent	Mandatory?
`ierr`	Error status	`INT`	`OUT`	Yes
`itrcr`	Tracer index	`INT`	`IN`	Yes
or				
`ytrcr`	Tracer name	`CHAR`	`IN`	Yes
or				
nothing				
`imeta`	Metadata index	`INT`	`IN`	Yes
or				
`ymeta`	Metadata name	`CHAR`	`IN`	Yes
`ydata`	Specified value	`INT`	`OUT`	Yes
or				
”	”	`REAL`	”	”
or				
”	”	`CHAR`	”	”
or				
”	”	`LOGICAL`	”	”
or				
”	”	`POINTER`	”	”

The name or the index of the metadata has to be provided and its value will be returned. If a tracer name or a tracer index is specified, then the value is returned only for this tracer. But if the tracer name or index is omitted, the value of the specified metadata will be returned for all tracers. The value array needs to be dimensioned accordingly.

2.3.3 Infrastructure methods

These methods are low-level methods used to manipulate the data structures and the information related to the tracers and the metadata. These methods are called only once in the code and should not be called by the tracer client, as they are already implemented in the model code.

trcr_init This subroutine is called at the beginning of the initialization phase and defines and initializes the mandatory metadata with default values. It initializes some indices and nullifies some pointers used in the tracer module.

trcr_alloc This subroutine allocates the memory for the data structures used to save the tracers, their tendency and boundary fields and it initializes these structures. It is called only once during the allocation phase of the model. After a call to this routine, no more tracers can be defined and a call to `trcr_new` will result in an error.

trcr_setup_vartab This subroutine is called from `organize_data.f90` and mimics the COSMO subroutine `setup_vartab`. This is a necessary operation to do I/O with the tracers.

trcr_print This subroutine is called only once from `lmorg.f90` after the allocation phase. This routine produces a summary of all tracers and of the associated metadata in the standard output of the COSMO model. The standard output should be checked by all tracer clients to make sure that their tracers are correctly handled (i.e. that the values of the metadata have been set according to their needs).

trcr_cleanup This subroutine deallocates the memory allocated in `trcr_alloc`. It is called only once at the end of the COSMO program. After a call to this routine, any tracer data is lost an all calls to `trcr_get` or `trcr_meta_get` will result in an error.

3 A short user's guide

In this section, we first show briefly how we replaced the existing microphysics variables by the tracer module. This example should help users who already have their tracers coupled to the COSMO model (e.g. COSMO-ART) in making use of the tracer module. In a second example, we outline the main steps to introduce a new tracer into the model.

3.1 The microphysics example

We have replaced the current treatment of all microphysics species (water vapor: `qv`, cloud water: `qc`, cloud ice: `qi`, rain: `qr`, snow: `qs`, graupel: `qg`) by the tracer module. The microphysics species are no longer handled as individual global variables but are now packed in the data structures related to the tracers and can only be defined, accessed and modified using the tracer methods.

3.1.1 The generic code parts

In all code parts where actions that can now be performed by the tracer module were executed, we could simply remove the code lines related to the microphysics species. This is for instance the case of the allocation (`src_allocation.f90`), the filling of the `var` structure (`src_setup_vartab.f90`) or the handling of the I/O for initial and boundary conditions (`organize_data.f90`). These actions are common to all tracers.

An example of a code section which is generic for all tracers can be found in Fig. 1. In this code section, the list of the variables that need initial conditions to be read from file is generated. On the left panel, the original code is shown. The generation of the variable list implied to add each variable explicitly (i.e. using its name). In the new version, shown on the right panel, the metadata `T_INI_ID` is tested for each tracer and if this metadata indicates that IC have to be read from file (`T_INI_FILE`) then the tracer name is added to the list.

Figure 1: Replacement of the microphysics species in a generic code section
left panel: original code (without tracer); right panel: new code (with tracer)

3.1.2 The specific code parts

In code parts that are specific to the microphysics, the original code could be kept almost unchanged. The main change is the access to the data. In the original code, a `USE` statement of the module `data_fields.f90` is performed in order to get access to the microphysics variables (`qv`, `qc`, etc.) and these variables can then be used directly. In the new version, some `trcr_get` have to be performed instead to retrieve the microphysics variables.

An example can be seen in Fig. 2. This code section is extracted from the subroutine `hydci_pp`, which computes the rate of change of temperature, cloud water, cloud ice, vapor, rain and snow due to microphysical processes related to the formation of grid scale precipitation.
On the left panel (original code version), it becomes clear that each microphysics specie is accessed using the corresponding global variable defined in the source file `data_fields.f90` (`USE data_fields` statement). It is then used in the code at the time level `nx`.
In the tracer version (shown on the right panel), local pointers are defined for each microphysical species for each required time level. Then the data are accessed using the statements `CALL trcr_get` for each variable at the required time level. These pointers are 3-dimensional pointers. The local pointers can then be used instead of the global variables in the parameterization.

It is crucial to notice that the tracer client now has to work with pointers instead of allocated fields. This implies for instance that all instances of `IF(ALLOCATED(my_var))` have to be replaced by `IF(ASSOCIATED(my_var))`. The validity of a pointer (which points on a given variable for a given time step) is not longer than one time step, whereas an allocated variable

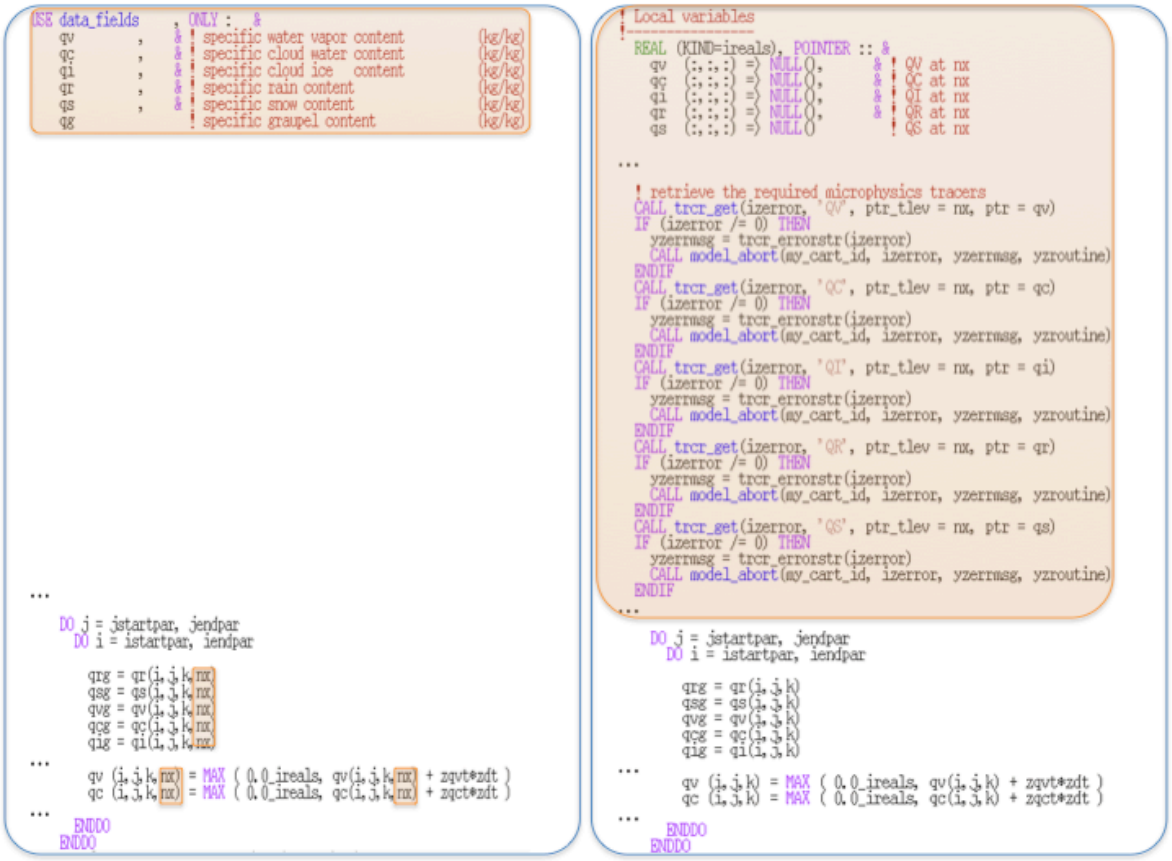

Figure 2: Replacement of the microphysics species in a specific code section
left panel: original code (without tracer); right panel: new code (with tracer)

3.1.3 The microphysics specificities in generic code parts

Major difficulties when replacing the existing treatment of the microphysics species by the tracer module methods arose in the code sections generic to all tracers. Indeed, numerous inconsistencies are present in the advection, in the turbulent mixing, in the treatment of the lateral boundary conditions and in the time stepping for the microphysics species. Although these operations should usually be handled in a consistent way for all tracers, we could not harmonize the treatment of the microphysics species for backwards compatibility reasons. We had to find a solution to handle these specificities without altering the integrity of the tracer module. We made use of the metadata to distinguish these special cases. **The use of these special metadata is not recommended to other tracer clients and they will be removed from the code over time.**

We list below the metadata that we have defined to reproduce the original treatment of the microphysics species, the possible values and the default (i.e. the standard treatment for other tracers). We do not recommend using these metadata for other tracers without intensive testing. Indeed, it is not clear at all for most of them why the behavior they are activating is required for some microphysical species and if it would be meaningful for other species. Caution is thus required when manipulating these metadata.

CLP_10E-12 At the end of the turbulent mixing for the Leapfrog dynamical core, a clipping is performed to ensure that the positive definite quantities are not exhibiting negative

values. In case of cloud ice (qi), another clipping method than the usual one is performed. Any value smaller than 10E-12 is set to zero instead of clipping only the negative values (i.e. smaller than 0). The metadata CLP_10E-12 is set to FALSE for all tracers except for qi. It means that we perform the usual clipping (clipping of the negative values only) for all tracers except for qi (for which we clip any value smaller than 10E-12).

Remark: this metadata is only active for qi if the standard metadata T_CLP_ID has been set to T_CLP_ON.

MASSFLX_CLP At the end of the turbulent mixing for the Leapfrog dynamical core and the Runge Kutta core, a clipping is performed to ensure that the positive definite quantities are not exhibiting negative values. In case of water vapor (qv) and cloud water (qc), a mass-flux correction scheme can be used instead of the usual clipping method. This is activated by a hard-coded internal switch (lmassf_diffusion) in the original code. The metadata MASSFLX_CLP is set to FALSE for all tracers except for qv and qc. It means that we perform the usual clipping for all tracers except for qv and qc (for which we redistribute the clipped mass).

Remark: this metadata is only active for qv and qc if the standard metadata T_CLP_ID has been set to T_CLP_ON.

ADD_CLP_ADV When using the advection in the Runge-Kutta core not written in conservation form (ltrcr_conserv_form=.FALSE.), a clipping is performed at the end of the advection routine (advection_pd) but only for the precipitating species (qr, qs, qg). The metadata ADD_CLP_ADV is set to TRUE for these species whereas it is set to FALSE for any other tracer, thus activating this additional clipping only for the three species mentioned above. This metadata is of course only active if advection is performed (i.e. if the metadata T_ADV_ID has been set to T_ADV_ON).

BD_0GRAD_FORCED Several types of lateral boundary condition can be chosen using the metadata T_LBD_ID. This type is then used at several locations in the code, among others in the advection. A routine (lateral_boundaries_zerograd in the new code, western_boundary, eastern_boundary, southern_boundary and northern_boundary in the original code) in the advection part of the Runge Kutta core (subroutine advection_pd) sets the lateral boundaries in case of a zero-gradient boundary condition type. This routine should thus only be called if T_LBC_ID is set to T_LBC_ZEROGRAD. However this is done in any case for qi, qr, qs and qg in the original code except if the boundaries are read from a file. We thus have to define a metadata which overwrites the boundary settings in the advection (for Runge Kutta) for these species except if the boundaries are read from file (T_LBC_FILE). BD_0GRAD_FORCED is thus set to FALSE for all tracers except for the above mentioned species. This metadata is of course only active if advection is performed (T_ADV_ON).

BD_SET_FORCED The type of lateral boundary condition is also used at the end of the dynamics computation (section 6.3.1 of lmorg.f90). At this location, the boundaries are set in case a zero-value (T_LBC_ZERO) or zero-gradient (T_LBC_ZEROGRAD) boundary condition has been chosen. For the precipitating species however, this boundary settings is done also if the boundaries have been read from file and the type (zero-value or zero-gradient) is defined according to the namelist switch itype_lbc_qrsg in the original code. In order to reproduce

this behavior, the switch BD_SET_FORCED is introduced. It is set to zero for all species except for the precipitating species. For them, it is set to 1 or 2 according to the value of the namelist switch itype_lbc_qrsg. A value of 1 means a zero-gradient boundary condition whereas a value of 2 means a zero-value boundary condition.

DAMP_FORCED The type of lateral boundary condition plays a role for boundary relaxation and Rayleigh damping. In case of constant boundary conditions or in case of zero-gradient boundary conditions, no relaxation and no Rayleigh daming should be done. For the precipitating species, Rayleigh damping (restoration to filtered fields) is performed nevertheless also when the boundary conditions are set to a zero-gradient type. In order to reproduce this behavior, the switch DAMP_FORCED is introduced. It is set to FALSE for all species except for the precipitating species. For these species, it is set to TRUE in case Rayleigh damping against filtered fields has been selected (itype_spubc of 2 and T_DAMP_ON) and the lateral boundary conditions have been set to a zero-gradient type.

SP_ADV_LF Advection in Leapfrog is normally done using centered differences and a Leapfrog time stepping scheme. The vertical advection is done implicitly. This default is applied to all tracers and to qv and qc. In the original code however, qi, qr, qs and qg are advected using others schemes. qi is advected using a positive definite scheme based on an Euler forward time stepping scheme. The vertical advection is also positive definite and is solved explicitly. For the precipitating species, a semi-Lagrange scheme with a Leapfrog time stepping is used. We use the metadata SP_ADV_LF to reproduce these various schemes. The default, SP_ADV_LF=1, is applied to all tracers except the species mentioned above; the normal Leapfrog scheme is used. For qi, the metadata SP_ADV_LF is set to 2, in order to make use of the Euler forward time stepping scheme. For the precipitating species, SP_ADV_LF is set to 3, inducing the use of the semi-Lagrange scheme.

Remark: A correct treatment of T_ADV_OFF is guaranteed only for the default Leapfrog scheme.

3.1.4 Treating the surface field

For any prognostic variable, there are possibly associated fields: the tendency fields, the boundary field, a surface field or an emission field for instance. In the current COSMO code, the only field that can be associated directly to a variable is the boundary field. This is dictated by the way COSMO defines a variable (i.e. by the composition of the var structure). It means that only the boundary field can be handled properly in an automated way.

However, in order to treat correctly the bottom boundary condition in case a surface field should be used (T_BBC_ID set to T_BBC_SURF_VAL), we need an automated method to relate a surface field to the tracer. The method currently used is constrained by the variable handling in COSMO and is not very satisfying. The tracer client has to define the surface field (in data_fields.f90), to allocated it (in src_allocation.f90), to fill the var structure for it (in src_setup_vartab), to update it (in lmorg.f90) and to deallocate it (in src_allocation.f90) himself or herself. In the routine organize_data, we define a pointer metadata (SURF_FIELD). We then search for each tracer having a surface field as bottom boundary condition (T_BBC_SURF_VAL) a field in the var structure named like the tracer field but with an extension _S (QV_S for QV for instance). If such a field is found, the pointer

SURF_FIELD is associated to this field.

This method is inflexible: the name of the surface field must be constructed using the name of the tracer and the _S extension. And it is laborious, the tracer client having to perform lots of operations to handle this field. Care is required when a surface field should be used as bottom boundary condition.

3.2 How to introduce a new tracer?

The introduction of a new tracer is relatively straightforward using the tracer module. The general workflow to introduce a new tracer can be summarized in:

- Defining the tracer: name, GRIB information, set of dynamics operation to undergo, type of initial and boundary conditions
- Defining associated metadata: determine if some information has to be attached to the tracer and if yes, define and set these metadata
- Defining initial and boundary conditions if needed (in case they are not read from file and should not have a zero value or a zero-gradient)
- Retrieving the tracer to perform various actions: computation of the source/sink terms of the tracer and update of the tracer, use of the tracer to compute derived quantities
- Adding the tracer to the output list in the namelist INPUT_IO

Each operation listed above has to take place in a meaningful location in the COSMO code. Some basic knowledge of the workflow in COSMO can help to use the tracer module correctly. We describe roughly the COSMO workflow in section 4.2.

3.2.1 Define the tracer and its metadata

The tracer client has to call trcr_new from an appropriate location. This appropriate location is usually organize_physics for the physical tracers, src_artifdata for artificial tracers and the "organizing" routine of associated modules (e.g. the "organizing" routine of the aerosol module in case of aerosols). This has to occur before the allocation phase but after the initialization of the tracer module (i.e. in section 2.1. of lmorg.f90). An example of an "organizing" routine can be found in Appendix A.

A crucial task for the tracer client is to choose the type of initial and boundary conditions for the tracer, to define which processes (advective transport, convective transport, etc.) it should undergo. In case of user-defined boundary conditions, the user has to retrieve the boundary field using the procedure trcr_get and assign a value to the corresponding pointer.

In some cases, it might be necessary to define additional metadata. It might be useful for instance to have the molar mass of some chemical tracers. These metadata can be defined and set at the same place using the procedures trcr_meta_define and trcr_meta_set.

3.2.2 Retrieve the tracer and its metadata

Wherever the tracer client needs to use its tracer (e.g. in parameterizations of physical or chemical processes), it has to retrieve it using `trcr_get`. The required time step has to be specified. It is also possible to retrieve the tendency or the boundary data using the same procedure. An example can be found in Appendix B.

4 Implementation details

This section details some design choices made internally in the tracer module. It is important to note that this information is subject to change at any time and that the tracer client should not write code that assumes any of the information given here.

4.1 Data structures

The main data structures used to manage the tracers at a low level are allocatable arrays of type `REAL`:

- `trcr_data` is a 5-dimensional array dimensioned with `(ie, je, ke, n_trcr, n_tlev)` and contains the tracer fields.
- `trcr_data_bd` is a 5-dimensional array dimensioned with `(ie, je, ke, n_bd, 2)` and contains the boundary field for the tracers.
- `trcr_data_tens` is a 4-dimensional array dimensioned with `(ie, je, ke, n_trcr)` and contains the tendency field for the tracers.

The number of tracers, `n_trcr`, and the number of boundary fields for the tracers, `n_bd`, are updated by each call to `trcr_new`. The method `trcr_get` accesses these data structures and returns 3-dimensional pointers for the tracer data and the tracer tendency and a 4-dimensional pointer for the tracer boundaries.

4.2 Workflow

The COSMO workflow can be roughly described as follows:

1. Initialization phase
   (a) Initialization of the environment and domain decomposition
   (b) Read of the namelist variables
   (c) Initialization of the tracer module
   (d) Definition of the tracers and of their metadata
   (e) Allocation
   (f) Fill in of the `var` structure (I/O table)
   (g) Definition of the variable list to read for IC, BC and for the restart
   (h) Read (or computation) of the IC and of the two first BC sets
   (i) Initialization of the output variable list
   (j) Print of the tracer list and metadata in the standard output
   (k) Initialization of the various packages (dynamics, physics, assimilation,)
   (l) (Model initialization by digital filtering)
2. Time stepping loop
   (a) Time step initialization (read new BC data if needed and interpolate them in time, swap time levels, reset the tendencies to zero)
   (b) Computation of the physical parameterizations (first set)
   (c) Computation of the dynamics
   (d) Setting of special boundaries
   (e) Computation of the assimilation
   (f) Computation of budgets
   (g) Computation of the boundary relaxation
   (h) Computation of the physical parameterizations (second set)
   (i) Nullification of small values for the tracers
   (j) Boundary exchanges
   (k) Diagnostics computation
   (l) Results output (if needed)
3. Finalization phase
   (a) Deallocation of the memory
   (b) Collection of the timings
   (c) Finalization of the MPI environment

The operations on the tracers cannot be performed in any order but they have to fit into the COSMO workflow.

In order to check that the main tracer operations are done in the right sequence, the tracer module internally uses a status variable which is checked upon every call to the tracer module to ensure a legal calling sequence (state machine). This variable can take the value "start", "define", "alloc" or "finish". At the beginning of the code, the status is "start". `trcr_init` can only be called with this status. At the end of `trcr_init`, the status is changed to "define". `trcr_new` and `trcr_alloc` can only be called for the status "define". At the end of the allocation routine (`trcr_alloc`), the status is changed to "alloc". At this point, it is no longer allowed to define new tracers. The filling of the `var` structure (`trcr_setup_vartab`) and the printing of the tracer list and associated metadata in the standard output have to be performed for the status "alloc". The subroutines which access the tracer structure, namely `trcr_get_index`, `trcr_check_index` and `trcr_get`, can be called for the status "define" or "alloc", depending on the arguments. `trcr_cleanup` has to be called with the status "alloc" and the status is changed to "finish" at the end of this subroutine.

4.3 Metadata module

The metadata management of the tracer module is handled by a general metadata backend in the source code files `src_tracer_metadata.f90` and `data_tracer_metadata.f90`. The tracer module defines a metadata container (of type `t_metadata`) and pipes all metadata requests from the methods `trcr_meta_define`, `trcr_meta_set` and `trcr_meta_get` to methods of the metadata backend. The metadata backend is not logically linked to the tracer module and can — in principle — be used for other metadata storage tasks.

Internally, all metadata is stored in a buffer of type `CHARACTER`. The different metadata are typecast using the Fortran `TRANSFER` intrinsic. Depending on the compiler dependent implementation, this allows for efficient access to metadata avoiding data copies.

Access to metadata can potentially be slow when done via name of the tracer and name of the metadata. This is especially the case, when a large number of tracers and metadata have been defined, implying a larger number of string comparisons. Accessing metadata in this way from within loops should be avoided at all costs. Access by tracer index and metadata index will be considerably more efficient. Further efficiency improvements can be achieved with metadata accesses for all tracers at once (omitting the tracer index or name in the metadata access methods, such as `trcr_meta_get`).

There are some internally hardcoded limits of the metadata module, such as the maximum length of the metadata name and the maximum length of strings that can be stored. If these limits are exceeded by any call to the metadata API, an appropriate error is issued. The user can then increase the limits and recompile.

4.4 Performance considerations

The performance has been tested on several platforms (Cray XT systems, Cray XE systems, NEC SX6, Mac OS). The performance of the model code is not changed significantly by our implementation on all tested platforms except for the NEC SX6, where a slowdown of about 10% has been observed. The main reason is vectorization issues. Some optimizations will be done in the future in order to achieve comparable runtime as with the original code. An improvement of the boundary exchange (bulk exchange instead of individual exchange for each tracer) is planned to prevent major performance issues when dealing with numerous tracers.

An experiment has been conducted to analyze the rise of the computing time by adding tracers successively, from zero additional tracers up to 100 additional tracers. For this experiment, COSMO-7 simulations of 24 hours forecast (as run operationally at MeteoSwiss but without any output) have been performed on a Cray XE6 system using 480 processing units. The resulting timings for a various number of tracers are illustrated in Fig. 3.
With zero additional tracers, one hour of simulation takes 21.45 seconds. Fitting a linear regression curve (least squares method) on the data, a slope of 0.7388 is obtained (equation: $time = 20.63 + 0.7388 * num\_tracers$). It means that adding one tracer costs about 0.7388 seconds more (out of 21.45 seconds), i.e. an increase of 3.5% of the computing time per additional tracer is observed. The computing time behaves extremely linearly with respect to the number of tracers.

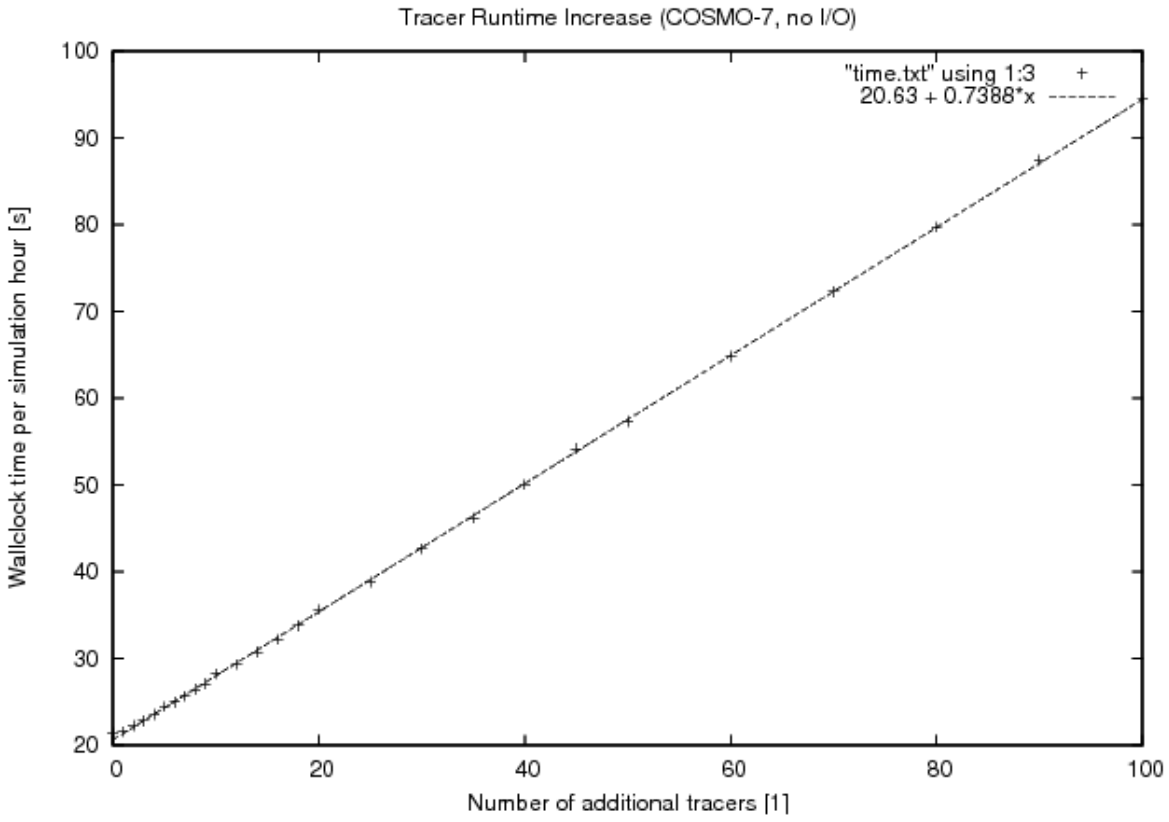

Figure 3: Runtime increase as a function of the tracer number

5 Results

The implementation of the tracer module is a purely technical change made to the code to improve its structure. It does not change the results.

After some code refactoring (mainly operation reordering), we could reproduce most of the results when handling the microphysics species with the tracer module. Some discrepancies have been observed and it turned out that these differences in the results were related to inconsistencies in the treatment of these species in the original code. Making use of the metadata mechanism, it has been possible to reproduce these specificities (inconsistencies) without degrading the integrity of the tracer module itself (see section 3.1.3).

The new code, including the handling of the microphysics species by the tracer module, has been extensively tested for several dozen of configurations. All tests have been successfully passed, i.e. bit-identical results have been obtained for all of them when comparing the original code and the tracer enabled code.

Parallelization has not been altered by our developments and the major configuration options of the original COSMO code are still available in combination with the tracer module.

6 Issues and needs for further refinements

The tracer module offers a facilitated tracer treatment. However it has been implemented in an existing code structure and only parts of this structure have been refactored. A larger effort would be needed to achieve more flexibility in the tracer definition and to reach a solution that could satisfy all tracer clients.

Technical and scientific aspects are still insufficiently well-handled for some tracer clients. From the scientific part, the main issue for a satisfying tracer treatment is related to conservation properties. Mass conservation is not guaranteed with the advection schemes implemented in COSMO. Other properties like the ratio between tracers are not conserved at all. This is not acceptable for several tracer clients.

From the technical side, the tracer module considers all tracers as being equal. We do not have a concept of tracer family for the moment. The possibility for differentiated treatment is left to the user, who can attach a new metadata, which controls a special behavior, to his set of tracers.

Another technical limitation of the current COSMO code is related to the inflexible way of defining variables. Each field that has to be read/written has to appear in the `var` structure. This structure contains among others the GRIB number for the element but also some pointers to the variable itself and to the boundary field. This means that a boundary field can be associated to each variable but no other field can be attached to it. A more general way to define a field would help to handle associated fields for the tracers. Usually tracer clients want to have a surface field, an emission, a tendency, a flux or other types of field like a sedimentation velocity attached to their species. For the moment, only the boundary field is completely handled by the tracer module. The tendency field can also be considered as fully covered by the tracer module but cannot be written to the output (which was also the case in the original COSMO version). A surface field can be attached to the tracer field if the user has chosen a surface value as bottom boundary condition but the tracer client still has to declare, allocate and define this surface field himself (see section 3.1.4). This solution is not satisfying in a long-term perspective. Emissions or other fields cannot be attached to the tracers in an easy way for the moment. Replacing the `var` structure by a more flexible datatype would alleviate this problem.

Currently, metadata are defined within the code, requiring a new compilation each time a metadata is changed. Handling some of the metadata via namelists would eliminate the need for recompilation.

7 Conclusions

A major code modification has been implemented in the COSMO model for a better treatment of passive, scalar prognostic variables (tracers). The implementation of this tracer module is a step towards an efficient, consistent and user-friendly tracer treatment in COSMO. The module has been successfully used for handling the microphysics species (of both the one-moment and the two-moment microphysics schemes) present in the model as well as for tracing air masses (work of Bojan Skerlak, ETH Zürich) and for the forecast of snow water content (Frick, 2012).

The mechanism to attach metadata to a tracer is a powerful functionality, which has been intensively used for the handling of the microphysics species. Indeed, the specificities (inconsistent treatment) present in some code parts for the microphysics species could be reproduced without attempting at the tracer module coherence and operating mode. Replacing the existing treatment of the microphysics species by the tracer module could therefore be done without altering the results.

Although the new module provides rather flexible methods for prognostic variables and convenient extension possibilities, the general definition of a field in COSMO remains inflexible. New methods for variable definition would greatly improve the structure of the COSMO code and open new doors for future developments.

Acknowledgments

We would like to thank Ulrich Schättler, Ulrich Blahak, Michael Baldauf, Jochen Förstner, Matthias Raschendorfer, Axel Seifert, Heini Wernli, Bernhard and Heike Vogel, Daniel Reinert, Christoph Knote, Astrid Kerkweg and Pirmin Kaufmann for useful discussions, suggestions and support. We are grateful to Claudia Frick and Bojan Skerlak who volunteered to serve as guinea pigs!

References

Buzzi M., 2008: Challenges in Operational Numerical Weather Prediction at High Resolution in Complex Terrain.
Diss. ETH No. 17714
[Available online at: *http://e-collection.library.ethz.ch*].

Doms G., 2011: A description of the Nonhydrostatic Regional COSMO-Model.
Part I: Dynamics and Numerics. *Tech. Rep.*
[Available online at: *http://cosmo-model.org*].

Doms G. et al., 2011: A description of the Nonhydrostatic Regional COSMO-Model.
Part II: Physical Parameterization. *Tech. Rep.*
[Available online at: *http://cosmo-model.org*].

Eaton B. et al., 2011. NetCDF Climate and Forecast (CF) Metadata Conventions, version 1.6, 5 December, 2011.

ECMWF, 2011: IFS Documentation - Cy37r2, Operational implementation 18 May 2011.
Part IV: Physical Processes. *IFS Documentation*
[Available online at: *http://www.ecmwf.int*].

Frick C. 2012: The numerical modeling of wet snowfall events.
Diss. ETH No. 20624
[Available online at: *http://e-collection.library.ethz.ch*].

Rast S., 2009. Using and Programming ECHAM5 - a first introduction.
[Available online at: *www.mpimet.mpg.de/fileadmin/staff/rastsebastian/echam_5.5_lecture.pdf*]

Schättler U. et al., 2011: A description of the Nonhydrostatic Regional COSMO-Model.
Part VII: User's guide. *Tech. Rep.*
[Available online at: *http://cosmo-model.org*].

Unidata, 2011. NetCDF Users Guide. *http://www.unidata.ucar.edu/software/netcdf/docs/netcdf.html*

Appendix A: an organizing routine

```
!==============================================================================
!+ External procedure for organizing the calls to the xxx packages
!------------------------------------------------------------------------------

SUBROUTINE organize_xxx (yaction, ierror, yerrmsg)

!------------------------------------------------------------------------------
!
! Description:
!   This procedure is the driving routine for calling the xxx
!   parametrizations.
!
! Method:
!
! Current Code Owner:
!
! History:
! Version    Date       Name
! ---------- ---------- ----
! Code Description:
! Language: Fortran 90.
! Software Standards: "European Standards for Writing and
! Documenting Exchangeable Fortran 90 Code".
!==============================================================

USE ...

USE data_tracers,   ONLY:  T_ADV_ON      , T_DIFF_ON, T_TURB_OFF,  &
                           T_CONV_OFF    , T_INI_FILE, T_LBC_CST,  &
                           T_BBC_SURF_VAL, T_RELAX_FULL,           &
                           T_DAMP_ON     , T_CLP_ON, T_ERR_NOTFOUND

USE src_tracers, ONLY: trcr_new, trcr_errorstr,                    &
                       trcr_meta_define, trcr_meta_set, trcr_get

!==============================================================

IMPLICIT NONE

!==============================================================

! Parameter list:
CHARACTER (LEN= *),       INTENT(IN)           ::                  &
  yaction      ! action to be performed

INTEGER (KIND=iintegers), INTENT(OUT)          ::                  &
  ierror       ! error status

CHARACTER (LEN= *),       INTENT(OUT)          ::                  &
  yerrmsg      ! error message

! Local variables:
INTEGER (KIND=iintegers)   ::   izerr

CHARACTER (LEN=25)         ::   yzroutine

! Tracer pointers
REAL (KIND=ireals), POINTER :: xcomp_bd(:,:,:,:) => NULL()
!------------------------------------------------------------------------------
!- End of header
```

```
!------------------------------------------------------------------------------
!- Begin Subroutine organize_xxx
!------------------------------------------------------------------------------
yzroutine = 'organize_xxx'
izerr     = 0_iintegers

...

!------------------------------------------------------------------------------
! Section xxx: Tracer definition
!------------------------------------------------------------------------------


IF (yaction == 'tracer') THEN

  ! init error value
  izerr = 0_iintegers

  ! Define all tracers required by the module xxx
  ! ---------------------------------------------

  ! define component X

  CALL trcr_new(                                          &
    ierr             = izerr,                             &
    yshort_name      = 'X_COMPONENT',                     &
    iGRIBparam       = 222,                               &
    iGRIBtable       = 6,                                 &
    yparent          = 'organize_xxx',                    &
    yunits           = 'kg kg-1',                         &
    ystandard_name   = 'x_component',                     &
    ylong_name       = 'specific content of component x', &
    itype_adv        = T_ADV_ON,                          &
    itype_diff       = T_DIFF_ON,                         &
    itype_turbmix    = T_TURB_OFF,                        &
    itype_passconv   = T_CONV_OFF,                        &
    itype_ini        = T_INI_FILE,                        &
    itype_lbc        = T_LBC_USER,                        &
    itype_bbc        = T_BBC_SURF_VAL,                    &
    itype_relax      = T_RELAX_OFF,                       &
    itype_damp       = T_DAMP_OFF,                        &
    itype_clip       = T_CLP_ON)

  IF (izerr /= 0_iintegers) THEN
    ierror  = izerr
    yerrmsg = trcr_errorstr(izerr)
    RETURN
  ENDIF


  ! Define and set all additional metadata required by the module xxx
  ! -----------------------------------------------------------------

  ! define metadata for molar mass [kg mol-1]
  CALL trcr_meta_define(izerr, 'MOL_MASS', -999_ireals)
  IF (izerr /= 0_iintegers) THEN
    ierror  = izerr
    yerrmsg = trcr_errorstr(izerr)
    RETURN
  ENDIF

  ! set molar mass for component X to 58.443 g mol-1
  CALL trcr_meta_set(izerr, 'X_COMPONENT', 'MOL_MASS', 58.443E-3)
```

```
  IF (izerr /= 0_iintegers) THEN
    ierror  = izerr
    yerrmsg = trcr_errorstr(izerr)
    RETURN
  ENDIF

!-------------------------------------------------------------------------
! Section xxx: Other operations
!-------------------------------------------------------------------------

ELSEIF (yaction == 'xxx') THEN

...
!-------------------------------------------------------------------------
! Section xxx: Define the boundaries [kg kg-1]
!-------------------------------------------------------------------------

ELSEIF (yaction == 'boundaries') THEN

  ! retrieve the boundaries of component X
  CALL trcr_get(izerr, 'X_COMPONENT', ptr_bd=xcomp_bd)
  IF (izerr/= 0) THEN
    ierror = izerr
    yerrmsg = trcr_errorstr(izerr)
    RETURN
  ENDIF

  ! set the boundaries of component X to 0.8 kg kg-1
  x_comp_bd(:,:,:,:) = 0.8_ireals

ENDIF


!-------------------------------------------------------------------------
! End of module procedure organize_xxx
!-------------------------------------------------------------------------

END SUBROUTINE organize_xxx
*******************************************************************
```

Appendix B: Module using a tracer variable

```
!+ Source module for the parameterization lambda
!------------------------------------------------------------------------------

MODULE src_lambda

!------------------------------------------------------------------------------
!
! Description:
!
! Current Code Owner:
!
! History:
!
! Code Description:
! Language: Fortran 90.
! Software Standards: "European Standards for Writing and
! Documenting Exchangeable Fortran 90 Code".
!==============================================================================
!
! Declarations:
!
! Modules used:
!
!
!------------------------------------------------------------------------------

USE ...

!------------------------------------------------------------------------------

USE src_tracers,      ONLY: trcr_get, trcr_errorstr, trcr_meta_get

!==============================================================================

IMPLICIT NONE

!==============================================================================
!
! Declarations
!
...
!
!==============================================================================

CONTAINS

!==============================================================================
!  Parameterization driver. Advances the process one time step ahead.
!------------------------------------------------------------------------------

SUBROUTINE lambda_driver

!------------------------------------------------------------------------------
!
! Description:
!
!
!
!==============================================================================
```

```
! Declarations

!  Local variables of type INTEGER

...

!  Local variables of type REAL

...

!  Tracer pointers
REAL (KIND=ireals), POINTER :: &
  xcomp  (:,:,:) => NULL() , &     ! component X at tlev=nx
  xtens  (:,:,:) => NULL()         ! tendency of component X

!  Metadata variables
REAL (KIND=ireals) :: molmass_x


CHARACTER(LEN=25) :: yzroutine = 'lambda_driver'

!==============================================================================
!  Start calculations
!------------------------------------------------------------------------------

IF(l2tls) THEN
  nx = nnew
ELSE
  nx = nnow
ENDIF

!------------------------------------------------------------------------------
! Retrieve pointer to required tracers
!------------------------------------------------------------------------------

CALL trcr_get(izerror, 'X_COMPONENT', ptr_tlev = nx, ptr = xcomp, ptr_tens=xtens)
IF (izerror /= 0) THEN
  yzerrmsg = trcr_errorstr(izerror)
  CALL model_abort(my_cart_id, izerror, yzerrmsg, yzroutine)
ENDIF

!------------------------------------------------------------------------------
!  Compute derived quantities using component X
!------------------------------------------------------------------------------

DO j = jstarts, jends
  DO i = istarts, iends
    phi_s(i,j) = xcomp(i,j,ke) * lambda + alpha
  ENDDO
ENDDO

...

!------------------------------------------------------------------------------
!  Use molar mass
!------------------------------------------------------------------------------

CALL trcr_meta_get(izerror, 'X_COMPONENT', 'MOL_MASS', molmass_x)
IF (izerror /= 0) THEN
  yzerrmsg = trcr_errorstr(izerror)
  CALL model_abort(my_cart_id, izerror, yzerrmsg, yzroutine)
ENDIF
```

```
DO k=1,ke
  DO j = jstarts, jends
    DO i = istarts, iends
      xppm(i,j,k) = xcomp(i,j,k) * rho(i,j,k) * molmass_x * r_gas_univ * &
                    t (i,j,k,nnew) / (p0(i,j,k) + pp(i,j,k,nnew))
    END DO
  END DO
END DO


...

!-------------------------------------------------------------------------------
!  Update the tendency of component X
!-------------------------------------------------------------------------------

DO k=1,ke
  DO j = jstarts, jends  ! DO loops over COSMO horizontal grid points
    DO i = istarts, iends
      xtens(i,j,k) = xtens(i,j,k) + (zeta(i,j,k)-phi_s(i,j))*  &
                     (hhl(i,j,k)-hsurf(i,j))*beta^2
    END DO
  END DO
END DO

...
!-------------------------------------------------------------------------------
!  Update component X
!-------------------------------------------------------------------------------

xcomp(:,:,:) = xcomp(:,:,:) + 42.0_ireals

!-------------------------------------------------------------------------------
!  End calculations
!===============================================================================

END SUBROUTINE lambda_driver

!===============================================================================

END MODULE src_lambda
********************************************************************************
```

List of COSMO Newsletters and Technical Reports

(available for download from the COSMO Website: www.cosmo-model.org)

COSMO Newsletters

No. 1: February 2001.

No. 2: February 2002.

No. 3: February 2003.

No. 4: February 2004.

No. 5: April 2005.

No. 6: July 2006.

No. 7: April 2008; Proceedings from the 8th COSMO General Meeting in Bucharest, 2006.

No. 8: September 2008; Proceedings from the 9th COSMO General Meeting in Athens, 2007.

No. 9: December 2008.

No. 10: March 2010.

No. 11: April 2011.

No. 12: April 2012.

COSMO Technical Reports

No. 1: Dmitrii Mironov and Matthias Raschendorfer (2001):
Evaluation of Empirical Parameters of the New LM Surface-Layer Parameterization Scheme. Results from Numerical Experiments Including the Soil Moisture Analysis.

No. 2: Reinhold Schrodin and Erdmann Heise (2001):
The Multi-Layer Version of the DWD Soil Model TERRA_LM.

No. 3: Günther Doms (2001):
A Scheme for Monotonic Numerical Diffusion in the LM.

No. 4: Hans-Joachim Herzog, Ursula Schubert, Gerd Vogel, Adelheid Fiedler and Roswitha Kirchner (2002):
LLM ¯ the High-Resolving Nonhydrostatic Simulation Model in the DWD-Project LITFASS.
Part I: Modelling Technique and Simulation Method.

No. 5: Jean-Marie Bettems (2002):
EUCOS Impact Study Using the Limited-Area Non-Hydrostatic NWP Model in Operational Use at MeteoSwiss.

No. 6: Heinz-Werner Bitzer and Jürgen Steppeler (2004):
Documentation of the Z-Coordinate Dynamical Core of LM.

No. 7: Hans-Joachim Herzog, Almut Gassmann (2005):
Lorenz- and Charney-Phillips vertical grid experimentation using a compressible non-hydrostatic toy-model relevant to the fast-mode part of the 'Lokal-Modell'.

No. 8: Chiara Marsigli, Andrea Montani, Tiziana Paccagnella, Davide Sacchetti, André Walser, Marco Arpagaus, Thomas Schumann (2005):
Evaluation of the Performance of the COSMO-LEPS System.

No. 9: Erdmann Heise, Bodo Ritter, Reinhold Schrodin (2006):
Operational Implementation of the Multilayer Soil Model.

No. 10: M.D. Tsyrulnikov (2007):
Is the particle filtering approach appropriate for meso-scale data assimilation ?

No. 11: Dmitrii V. Mironov (2008):
Parameterization of Lakes in Numerical Weather Prediction. Description of a Lake Model.

No. 12: Adriano Raspanti (2009):
COSMO Priority Project "VERification System Unified Survey" (VERSUS): Final Report.

No. 13: Chiara Marsigli (2009):
COSMO Priority Project "Short Range Ensemble Prediction System" (SREPS): Final Report.

No. 14: Michael Baldauf (2009):
COSMO Priority Project "Further Developments of the Runge-Kutta Time Integration Scheme" (RK): Final Report.

No. 15: Silke Dierer (2009):
COSMO Priority Project "Tackle deficiencies in quantitative precipitation forecast" (QPF): Final Report.

No. 16: Pierre Eckert (2009):
COSMO Priority Project "INTERP": Final Report.

No. 17: D. Leuenberger, M. Stoll and A. Roches (2010):
Description of some convective indices implemented in the COSMO model.

No. 18: Daniel Leuenberger (2010):
Statistical analysis of high-resolution COSMO Ensemble forecasts in view of Data Assimilation.

No. 19: A. Montani, D. Cesari, C. Marsigli, T. Paccagnella (2010):
Seven years of activity in the field of mesoscale ensemble forecasting by the COSMO–LEPS system: main achievements and open challenges.

No. 20: A. Roches, O. Fuhrer (2012):
Tracer module in the COSMO model.

COSMO Technical Reports

Issues of the COSMO Technical Reports series are published by the *COnsortium for Small-scale MOdelling* at non-regular intervals. COSMO is a European group for numerical weather prediction with participating meteorological services from Germany (DWD, AWGeophys), Greece (HNMS), Italy (USAM, ARPA-SIMC, ARPA Piemonte), Switzerland (MeteoSwiss), Poland (IMGW), Romania (NMA) and Russia (RHM). The general goal is to develop, improve and maintain a non-hydrostatic limited area modelling system to be used for both operational and research applications by the members of COSMO. This system is initially based on the COSMO-Model (previously known as LM) of DWD with its corresponding data assimilation system.

The Technical Reports are intended

- for scientific contributions and a documentation of research activities,
- to present and discuss results obtained from the model system,
- to present and discuss verification results and interpretation methods,
- for a documentation of technical changes to the model system,
- to give an overview of new components of the model system.

The purpose of these reports is to communicate results, changes and progress related to the LM model system relatively fast within the COSMO consortium, and also to inform other NWP groups on our current research activities. In this way the discussion on a specific topic can be stimulated at an early stage. In order to publish a report very soon after the completion of the manuscript, we have decided to omit a thorough reviewing procedure and only a rough check is done by the editors and a third reviewer. We apologize for typographical and other errors or inconsistencies which may still be present.

At present, the Technical Reports are available for download from the COSMO web site (www.cosmo-model.org). If required, the member meteorological centres can produce hard-copies by their own for distribution within their service. All members of the consortium will be informed about new issues by email.

For any comments and questions, please contact the editor:

Massimo Milelli
Massimo.Milelli@arpa.piemonte.it