

TECHNICAL DOCUMENTATION

Reference Manual and User's Guide

Contents

1	Overview.	4
2	Libraries needed.	5
3	General design.	10
4	Modules.	12
4.1	Module <code>precision</code> .	12
4.2	Module <code>tools</code> .	12
4.2.1	<code>tools_ij2s</code> .	13
4.2.2	<code>tools_s2ij</code> .	13
4.2.3	<code>tools_ijk2s</code> .	14
4.2.4	<code>tools_s2ijk</code> .	15
4.3	Module <code>random</code> .	15
4.3.1	<code>random_parametersup</code> .	16
4.3.2	<code>random_uniform0d</code> .	16
4.3.3	<code>random_normalstd0d</code> .	16
4.3.4	<code>random_normal0d</code> .	17
4.3.5	<code>random_normal1dmatrix</code> .	18
4.3.6	<code>random_normal1dsqrtmatrix</code> .	19
4.3.7	<code>random_meanestimator1d</code> .	20
4.3.8	<code>random_covarianceestimator1d</code> .	20
4.3.9	<code>random_sqrtcovestimator1d</code> .	21
4.4	Module <code>initialize</code> .	22
4.4.1	<code>initialize_parametersup</code> .	22
4.4.2	<code>initialize_initialize1</code> .	22
4.4.3	<code>initialize_initialize2</code> .	23
4.5	Module <code>model</code> .	23
4.5.1	<code>model_parametersup</code> .	23
4.5.2	<code>model_model</code> .	24
4.5.3	<code>model_tangmodel</code> .	24
4.5.4	<code>model_sqrtmodelerror</code> .	25
4.5.5	<code>model_modelerror</code> .	25

4.6	Module observations.	25
4.6.1	observations_parametersup.	26
4.6.2	observations_ifobservations.	26
4.6.3	observations_numberobs.	26
4.6.4	observations_obsvalue.	27
4.6.5	observations_tangobsop.	27
4.6.6	observations_obsop.	27
4.6.7	observations_sqrtcovobs.	28
4.6.8	observations_covobs.	28
4.7	Module parallel.	28
4.7.1	parallel_init.	29
4.7.2	parallel_ranksize.	30
4.7.3	parallel_finalize.	30
4.7.4	parallel_pregrid.	30
4.7.5	parallel_postgrid.	31
4.7.6	parallel_predistribute.	31
4.7.7	parallel_postdistribute.	32
4.7.8	parallel_getrf.	32
4.7.9	parallel_gesv.	33
4.7.10	parallel_gemm.	34
4.7.11	parallel_operator.	35
4.7.12	parallel_gesvd.	36
4.7.13	parallel_svdcut.	37
4.8	Module ekf.	38
4.8.1	ekf_predictor.	38
4.8.2	ekf_corrector.	39
4.9	Module enkf.	40
4.9.1	enkf_predictor.	40
4.9.2	enkf_corrector.	41
4.10	Module rrsqrtkf.	42
4.10.1	rrsqrtkf_predictor.	42
4.10.2	rrsqrtkf_corrector.	43
4.11	Module rrsqrtenkf.	44
4.11.1	rrsqrtenkf_predictor.	45
4.11.2	rrsqrtenkf_corrector.	45
5	How to use the package.	46
5.1	Case ekf.	47
5.2	Case enkf.	48
5.3	Case rrsqrtkf.	48
5.4	Case rrsqrtenkf.	49
5.5	An example of a main program.	50
6	Examples.	52
6.1	Example 0D.	52
6.1.1	Problem.	52
6.1.2	Domain.	52

6.1.3	State vectors.	53
6.1.4	Model.	53
6.1.5	Model errors.	53
6.1.6	Observations.	53
6.1.7	Observation errors.	53
6.1.8	Observation operator and the tangent observation operator.	54
6.1.9	Initialization.	54
6.1.10	Experiment.	54
6.2	Example 2D.	55
6.2.1	Problem.	55
6.2.2	Domain.	56
6.2.3	State vectors.	56
6.2.4	Model.	56
6.2.5	Model errors.	57
6.2.6	Observations.	57
6.2.7	Observation errors.	58
6.2.8	Observation operator and the tangent observation operator.	58
6.2.9	Initialization.	58
6.2.10	Experiment.	59
6.3	Example 3D - Assimilation of CO.	59
6.4	Example 3D - Assimilation of O3.	61

1 Overview.

The package presented here presents a set of Fortran 90 modules that implement the Kalman filter adapted for large scale problems. The methods that are implemented are the Extended Kalman filter (identified as EKF), the Reduced Rank Square Root filter (identified as RRSQRTKF), the Ensemble Kalman filter (identified as ENKF) and the Reduced Rank Square Root Ensemble filter (identified as RRSQRTEKF). Each method and each necessary task into the assimilation (for example model and observations) is coded into modules that can be replaced according to the application.

A list of capabilities is described below:

- **Modularity:** as mentioned before, each task into an assimilation implementation is separated one from another. It means that observations, model and assimilation are different entities. For example, if the observation stations change the location, there is no need to modify the main code, but only a module related with them. If we need to change the model, there is no need to transform all the code, but only the module related with the model. If we want to change the Kalman filter version, a change in the module related with the assimilation will suffice. This will allow the user to make minimum changes in all the codes around a modelling system (model, data formats, libraries, configuration files) in order to avoid programmer bugs.
- **Simplicity:** there are no derived types of variables defined in the code. It is clear that derived types of variables are a useful language tool in order to make powerful codes, but in this case the intention was to determine the global variables and the specific functions associated to the assimilation. The users are encouraged to introduce all the new abstract types of variables and all the complexity they need in those modules that have to be edited, rather than adjust their codes to an existent structure. The code has been prepared for both an expert programmer as well as for a medium programmer.
- **Language:** the modules are programmed in Fortran 90. The choice of the language was made because a big number of large scale models are written either in Fortran 77 or in Fortran 90. Only Fortran standards have been used, and the code will work with almost any Fortran compiler. It has been successfully compiled and executed with the Intel Fortran Compiler, Portland Fortran Compiler, GNU Fortran and g95, and its MPI wrappers.
- **Precision:** the modules can handle single and double precision. Some models have their outputs in single precision, others in double precision, so this is a useful feature. The switch between these two choices is done changing only one parameter in the whole code.
- **Parallelism:** repeated tasks like propagating states (or applying the observation operator, or the tangent observation operator, or the tangent model) are parallelized using the master-slave strategy with MPI (Message Passing Interface). In this case a set of independent tasks are sent to the processors. Once the task is performed, the slave receives a new task from the master. Linear algebra operations are performed using BLACS (Basic Linear Algebra Communication Subprograms) and SCALAPACK (Scalable LAPACK). The global matrices are distributed in a processor grid, then the operation is performed in each processor over local matrices. Finally, the global matrices are rebuilt from local

pieces. The parallelism is included in a module and the user just needs to call the parallel routines implemented there avoiding communicators and memory distribution.

2 Libraries needed.

To compile the modules, we need to compile additional libraries:

- BLAS (Basic Linear Algebra Subroutines): they are routines that provide standard building blocks for performing basic vector and matrix operations. The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations. See [1].
- LAPACK (Linear Algebra Package): it is written in Fortran 77 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. See [2].
- MPI (Message Passing Interface): MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users. See [5].
- BLACS (Basic Linear Algebra Communication Subprograms): it is a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms. See [3].
- SCALAPACK (Scalable LAPACK): it includes a subset of LAPACK routines redesigned for distributed memory MIMD parallel computers. It is currently written in a Single-Program-Multiple-Data style using explicit message passing for interprocessor communication. It assumes matrices are laid out in a two-dimensional block cyclic decomposition. It is designed for heterogeneous computing and it is portable on any computer that supports MPI or PVM. See [4].

The user should notice that if he wants to compile the code from scratch with his own compiler, it will be necessary to compile the libraries from the source code, because it can cause incompatibilities. This is equivalent to set configuration files in the libraries according to the architecture of the machines and the compiler used.

Here below there are hints to compile the libraries with different compilers. Remember that there are items that can change according to the version of compilers and the architecture, so use them as a guide.

- BLAS and LAPACK: you need to edit a make.inc file in the following way:

For g95:

```

FORTRAN = g95
OPTS     = -funroll-loops -O2
LOADER   = $(FORTRAN)
TIMER    = EXT_ETIME

```

For gfortran:

```

FORTRAN = gfortran
OPTS     = -O2
LOADER   = $(FORTRAN)
TIMER    = INT_ETIME

```

For ifort:

```

FORTRAN = ifort
OPTS     = -O3 -fp-port
NOOPT    = -fltconsistency
LOADER   = $(FORTRAN)
LOADOPTS = -O3
TIMER    = EXT_ETIME

```

For pgf90:

```

FORTRAN = pgf90
OPTS     = -Mnounroll -O3
LOADER   = $(FORTRAN)
TIMER    = EXT_ETIME

```

- MPICH: you need to execute the configure script with the following arguments:

```

./configure --prefix=<path to directory where MPICH will be \
            \installed> -fc=<Fortran compiler> -f90=<Fortran 90 \
            \compiler> --with-device=<device> --without-romio

```

- BLACS: you need to edit the Bmake.inc file in the following way:

For g95:

```

BTOPdir      = <path to BLACS source code>
COMMLIB      = MPI
PLAT         = LINUX
MPIdir       = <path to directory where MPI is installed>
INTERFACE    = -Df77IsF2C
SENDIS       =
BUFF         =
TRANSCOMM    = -DCSameF77
WHATMPI      =
SYSERRORS    =
F77          = mpif90
F77NO_OPTFLAGS =
F77FLAGS     = $(F77NO_OPTFLAGS) -O
F77LOADER    = $(F77)
F77LOADFLAGS =
CC           = mpicc
CCFLAGS      = -O
CCLOADER     = $(CC)
CCLOADFLAGS  =

```

For gfortran:

```

BTOPdir      = <path to BLACS source code>
COMMLIB      = MPI
PLAT         = LINUX
MPIdir       = <path to directory where MPI is installed>
INTERFACE    = -DAdd_
SENDIS       =
BUFF         =
TRANSCOMM    = -DCSameF77
WHATMPI      =
SYSERRORS    =
F77          = mpif90
F77NO_OPTFLAGS =
F77FLAGS     = $(F77NO_OPTFLAGS) -O
F77LOADER    = $(F77)
F77LOADFLAGS =
CC           = mpicc
CCFLAGS      = -O
CCLOADER     = $(CC)
CCLOADFLAGS  =

```

For ifort:

```

BTOPdir      = <path to BLACS source code>
COMMLIB      = MPI
MPIdir       = <path to directory where MPI is installed>
INTERFACE    = -DAdd_
SENDIS       =
BUFF         =
TRANSCOMM    = -DCSameF77
WHATMPI      =
SYSERRORS    =
F77          = mpif90
F77NO_OPTFLAGS = -O0
F77FLAGS     = $(F77NO_OPTFLAGS) -O
F77LOADER    = $(F77)
F77LOADFLAGS =
CC           = mpicc
CCFLAGS      = -O
CCLOADER     = $(CC)
CCLOADFLAGS  =

```

For pgf90:

```

BTOPdir      = <path to BLACS source code>
COMMLIB      = MPI

```

```

MPIdir      = <path to directory where MPI is installed>
INTERFACE   = -DAdd_
SENDIS      =
BUFF        =
TRANSCOMM   = -DCSameF77
WHATMPI     =
SYSERRORS   =
F77         = mpif90
F77NO_OPTFLAGS =
F77FLAGS    = $(F77NO_OPTFLAGS) -O
F77LOADER   = $(F77)
F77LOADFLAGS =
CC          = mpicc
CCFLAGS     = -O
CCLOADER    = $(CC)
CCLOADFLAGS =

```

- SCALAPACK: you have to edit the SLmake.inc file in the following way:

For g95:

```

home        = <path to SCALAPACK source code>
BLACSdir    = <path to directory where BLACS is installed>
SMPLIB      = <path to directory where MPI is installed>/libmpich.a
BLACSFINIT  = $(BLACSdir)/blacsF77init_MPI-$(PLAT)-$(BLACSDBGLVL).a
BLACSCINIT  = $(BLACSdir)/blacsCinit_MPI-$(PLAT)-$(BLACSDBGLVL).a
BLACSLIB    = $(BLACSdir)/blacs_MPI-$(PLAT)-$(BLACSDBGLVL).a
F77         = mpif90
CC          = mpicc
NOOPT       =
F77FLAGS    = -funroll-all-loops -O3 $(NOOPT)
DRVOPTS     = $(F77FLAGS)
CCFLAGS     = -O4
SRCFLAG     =
F77LOADER   = $(F77)
CCLOADER    = $(CC)
F77LOADFLAGS =
CCLOADFLAGS =
CDEFS       = -Df77IsF2C -DNO_IEEE $(USEMPI)
BLASLIB     = <directory where BLAS is installed>/libblas.a
LAPACKLIB   = <directory where LAPACK is installed>/liblapack.a

```

For gfortran:

```

home        = <path to SCALAPACK source code>
BLACSdir    = <path to directory where BLACS is installed>
SMPLIB      = <path to directory where MPI is installed>/libmpich.a
BLACSFINIT  = $(BLACSdir)/blacsF77init_MPI-$(PLAT)-$(BLACSDBGLVL).a

```



```

BLACSCINIT   = $(BLACSdir)/blacsCinit_MPI-$(PLAT)-$(BLACSDBGLVL).a
BLACSLIB     = $(BLACSdir)/blacs_MPI-$(PLAT)-$(BLACSDBGLVL).a
F77          = mpif90
CC           = mpicc
NOOPT        =
F77FLAGS     = -funroll-all-loops -O3 $(NOOPT)
DRVOPTS      = $(F77FLAGS)
CCFLAGS      = -O4
SRCFLAG      =
F77LOADER    = $(F77)
CCLOADER     = $(CC)
F77LOADFLAGS =
CCLOADFLAGS  =
CDEFS        = -DAdd_ -DNO_IEEE $(USEMPI)
BLASLIB      = <directory where BLAS is installed>/libblas.a
LAPACKLIB    = <directory where LAPACK is installed>/liblapack.a

```

For ifort:

```

home         = <path to SCALAPACK source code>
BLACSdir     = <path to directory where BLACS is installed>
SMPLIB       = <path to directory where MPI is installed>/libmpich.a
BLACSFINIT   = $(BLACSdir)/blacsF77init_MPI-$(PLAT)-$(BLACSDBGLVL).a
BLACSCINIT   = $(BLACSdir)/blacsCinit_MPI-$(PLAT)-$(BLACSDBGLVL).a
BLACSLIB     = $(BLACSdir)/blacs_MPI-$(PLAT)-$(BLACSDBGLVL).a
F77          = mpif90
CC           = mpicc
NOOPT        =
F77FLAGS     = -O2 $(NOOPT)
DRVOPTS      = $(F77FLAGS)
CCFLAGS      = -O2
SRCFLAG      =
F77LOADER    = $(F77)
CCLOADER     = $(CC)
F77LOADFLAGS =
CCLOADFLAGS  =
CDEFS        = -DAdd_ -DNO_IEEE $(USEMPI)
BLASLIB      = <directory where BLAS is installed>/libblas.a
LAPACKLIB    = <directory where LAPACK is installed>/liblapack.a

```

For pgf90:

```

home         = <path to SCALAPACK source code>
BLACSdir     = <path to directory where BLACS is installed>
SMPLIB       = <path to directory where MPI is installed>/libmpich.a
BLACSFINIT   = $(BLACSdir)/blacsF77init_MPI-$(PLAT)-$(BLACSDBGLVL).a
BLACSCINIT   = $(BLACSdir)/blacsCinit_MPI-$(PLAT)-$(BLACSDBGLVL).a

```

```

BLACSLIB      = $(BLACSdir)/blacs_MPI-$(PLAT)-$(BLACSDBGLVL).a
F77           = mpif90
CC            = mpicc
NOOPT         =
F77FLAGS      = -Mnounroll -O3 $(NOOPT)
DRVOPTS       = $(F77FLAGS)
CCFLAGS       = -O4
SRCFLAG       =
F77LOADER     = $(F77)
CCLOADER      = $(CC)
F77LOADFLAGS  =
CCLOADFLAGS   =
CDEFS         = -DAdd_ -DNO_IEEE $(USEMPI)
BLASLIB       = <directory where BLAS is installed>/libblas.a
LAPACKLIB     = <directory where LAPACK is installed>/liblapack.a

```

3 General design.

The package consists of several Fortran 90 modules, namely:

- **precision**: it defines precision in all types of variables.
- **tools**: it provides tools to simplify tasks in main programs or modules.
- **random**: it is devoted to random generation of numbers and vectors.
- **initialize**: it initializes the filter.
- **model**: it is dedicated to the model.
- **observations**: it sets all things related to observations.
- **parallel**: it provides procedures for parallelization.
- **ekf**: implementation of the Extended Kalman filter.
- **enkf**: implementation of the Ensemble Kalman filter.
- **rrsqrtkf**: implementation of the Reduced Rank Square Root Kalman filter.
- **rrsqrtenkf**: implementation of the Reduced Rank Square Root Ensemble Kalman filter.

The module **precision** should be included in all the modules listed above and the modules and programs developed by the user because it defines precision in all variables.

The module **tools** includes auxiliary functions. The user can add here all the tools needed to perform the simulation.

The module **random** has functions and subroutines that go from random number generation with uniform distribution to vector random generation with gaussian distribution with a prescribed mean and covariance matrix. Users can include additional functions if needed.

In general, the modules contain a subroutine called `<modulename>_parametersup`. The new parameters, if any, may be added to the list of public global variables in order to make them

visible outside the scope of the module. There are situations where some subroutines are used and others are not. In the last case, the user should not removed them, but remove the body of the subroutine, because they are needed at compilation time.

The module `initialize` initializes the state vector and the covariance matrix (or its square root version). Users can include here auxiliary modules in order to generate the initial condition.

The module `model` includes the model, its tangent version (that it will be used in some versions of the filter), and the covariance matrix of model errors (or its square root). The idea is to build a wrapper around an existing model in order to minimize edition in the model source code. Considerations of handling multiple types of variables, nonlinear run for the tangent model, formats of input data, etc., should be included in this module.

The module `observations` includes all the things related to observations, for example: to determine the number of observations, their values, time in which observations appear, etc. The user can add definition of station locations, types of observations, handling of different kind of observations (for example humidity, temperature and pressure), file formats if the observations are given in a file, etc.

Notice that users can take advantage of Fortran features. A Fortran array can be seen as a vector, thinking in column ordering (for C language, the ordering is performed by rows). Taking this property into account, inside the module `model`, users can think of the state vector as a 3-dimensional array, as it is normally used in large models. But outside the module, the 3-dimensional array is seen as a state vector, so there is no need to code a mapping and make the code less efficient. The same remark applies to the module `observations` and its associated procedures.

The module `parallel` contains a set of subroutines that, in general terms, are wrappers of existing parallel subroutines from MPI, BLACS, PBLAS and SCALAPACK. The objective to do this is that users do not deal directly with parallelization. Users can work with parallel subroutines as if they coded a sequential program. For repeated tasks, like the propagation of covariance matrices (the model has to be called many times), the master-slave strategy is used. For matrix operations, global matrices are distributed into the processor grid, then the matrix operation is performed in each processor using parallel libraries (PBLAS and SCALAPACK), and finally, the local pieces are gathered into the global matrix. In case users need another repeated task, they have to edit the `parallel_operator` subroutine. In case users need another matrix operation, they can follow the same style as mentioned before: distribution, operation, gathering.

The modules `ekf`, `enkf`, `rrsqrtkf` and `rrsqrtenkf` propose implementations of the Kalman filter dividing tasks in prediction and correction of the state vector and covariance matrices. These modules can serve as a guide to code new methods.

Programmers can notice that there are no abstract types of variables defined. This has been done on purpose in order to keep the source code as simple as possible. All the complexity and structures that can be defined in Fortran 90 (or other languages) should be built above these modules. Thinking in this way, the proposed package should be considered as a low-level software that users can include in their own programs.

All the linear algebra computation is performed using the libraries BLAS, LAPACK, PBLAS and SCALAPACK.

As a final remark, each module can be extended, or new modules can be defined and interact with the existing modules, at a low cost in lines of source code. As an example for a real case, see the next sections.

4 Modules.

The library is a set of Fortran 90 modules with the minimum requirements in order to set up the assimilation. The user must provide the model and the observations because they will highly depend on the application. Here below there is a detailed description of each Fortran 90 module.

4.1 Module precision.

The module `precision` is the most basic module. It defines global variables whose purpose is to set the precision in all types of variables.

- Name: `precision`
- Dependencies: none
- Global variables:
 - `low` (integer): number of bytes for single precision variables
 - `high` (integer): number of bytes for double precision variables
 - `plo` (integer): precision for logical variables
 - `pch` (integer): precision for character variables
 - `pin` (integer): precision for integer variables
 - `pre` (integer): precision for real variables
- Adding code: this module can be edited in order to change the precision. For example, if one works in a computer where a real double precision variable has 16 bytes, it must be set: `high = 16` and `pre = high`

4.2 Module tools.

The purpose of the module `tools` is to provide tools to simplify tasks in the main programs. The user can include here its own functions according to the application.

- Name: `tools`
- Dependencies: `precision`
- Global variables:
 - from `precision`: `low`, `high`, `plo`, `pch`, `pin`, `pre`
- Adding code: the user can edit this module adding new tools in order to make them visible in the assimilation modules

The subroutines/functions associated to this module are:

4.2.1 tools_ij2s.

- Purpose: it maps indices of a matrix of size $\mathbf{nx} \times \mathbf{ny}$ to indices of a vector of size $\mathbf{nx} \star \mathbf{ny}$ using Fortran style
- Synopsis: `tools_ij2s(nx , i , j , s)`
- Inputs:
 - \mathbf{nx} (integer): first dimension of a matrix
 - \mathbf{i} (integer): first coordinate of a matrix element ($1 \leq \mathbf{i} \leq \mathbf{nx}$)
 - \mathbf{j} (integer): second coordinate of a matrix element ($1 \leq \mathbf{j} \leq \mathbf{ny}$)
- Inputs/outputs: none
- Outputs:
 - \mathbf{s} (integer): index of the vector element ($1 \leq \mathbf{s} \leq \mathbf{nx} \star \mathbf{ny}$) corresponding to the (\mathbf{i} , \mathbf{j}) matrix element according to the Fortran style
- Calls: none
- Comments: in Fortran a matrix can be seen as a vector considering a predefined order by columns. For example, for a 2×2 matrix would be:

$$\mathbf{A} = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \longleftrightarrow \mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}. \quad (1)$$

Given a position (i, j) in a matrix \mathbf{A} of dimension $n_x \times n_y$, we have to deduce the position s of the element in the corresponding vector \mathbf{v} ordering the matrix by columns as in Fortran 90. Forming \mathbf{v} , at the position (i, j) we have already stored $j - 1$ columns, and then we have to put the last i components. Therefore:

$$s = (j - 1) n_x + i. \quad (2)$$

4.2.2 tools_s2ij.

- Purpose: it maps indices of a vector of size $\mathbf{nx} \star \mathbf{ny}$ to indices of a matrix of size $\mathbf{nx} \times \mathbf{ny}$ using Fortran style
- Synopsis: `tools_s2ij(nx , s , i , j)`
- Inputs:
 - \mathbf{nx} (integer): first dimension of a matrix
 - \mathbf{s} (integer): index of the vector element ($1 \leq \mathbf{s} \leq \mathbf{nx} \star \mathbf{ny}$)
- Inputs/outputs: none
- Outputs:

- **i** (integer): first coordinate of the matrix element corresponding to the **s** vector element according to the Fortran style
- **j** (integer): second coordinate of the matrix element corresponding to the **s** vector element according to the Fortran style
- Calls: none
- Comments: we have to do the inverse process of the previous subroutine. That is, given a component s of the vector **v** of size $n_x \star n_y$, we have to find the position (i, j) in a matrix **A** of size $n_x \times n_y$, where **A** is formed storing the components of **v** in columns in the matrix **A**. For example:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{bmatrix} \longleftrightarrow \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}. \quad (3)$$

Using the remainder's theorem we have that:

$$s = n_x r + t, \quad 0 \leq t < n_x. \quad (4)$$

If $t \neq 0$, then it is clear that $i = t$, but if $t = 0$, it means that we have stored exactly r columns and then we must define $i = n_x$. Once we have i , the number $(s - i) / n_x$ is the number of columns minus 1, therefore $j = (s - i) / n_x + 1$. Thus, we have computed (i, j) from s .

4.2.3 tools_ijk2s.

- Purpose: it maps indices of a matrix of size **nx** \times **ny** \times **nz** to indices of a vector of size **nx** \star **ny** \star **nz** using Fortran style
- Synopsis:
 - `tools_ijk2s(nx , ny , i , j , k , s)`
- Inputs:
 - **nx** (integer): first dimension of a matrix
 - **ny** (integer): second dimension of a matrix
 - **i** (integer): first coordinate of a matrix element ($1 \leq i \leq nx$)
 - **j** (integer): second coordinate of a matrix element ($1 \leq j \leq ny$)
 - **k** (integer): third coordinate of a matrix element ($1 \leq k \leq nz$)
- Inputs/outputs: none

- Outputs:
 - `s` (integer): index of the vector element ($1 \leq s \leq nx * ny * nz$) corresponding to the (`i`, `j`, `k`) matrix element according to the Fortran style
- Calls: none
- Comments: this subroutine is an extension of `tools_ij2s` for a matrix of rank 3.

4.2.4 `tools_s2ijk`.

- Purpose: it maps indices of a vector of size `nx * ny * nz` to indices of a matrix of size `nx × ny × nz` using Fortran style
- Synopsis:
 - `tools_s2ijk(nx , ny , s , i , j , k)`
- Inputs:
 - `nx` (integer): first dimension of a matrix
 - `ny` (integer): second dimension of a matrix
 - `s` (integer): index of a vector component ($1 \leq s \leq nx * ny * nz$)
- Inputs/outputs: none
- Outputs:
 - `i` (integer): first coordinate of the matrix element corresponding to the `s` vector element according to the Fortran style
 - `j` (integer): second coordinate of the matrix element corresponding to the `s` vector element according to the Fortran style
 - `k` (integer): third coordinate of the matrix element corresponding to the `s` vector element according to the Fortran style
- Calls: this subroutine is an extension of `tools_s2ij` for a matrix of rank 3.

4.3 Module `random`.

This module is devoted to generate random numbers and vectors with a specified distribution.

- Name: `random`
- Dependencies:
 - `precision`
- Global variables:
 - from `precision`: `low`, `high`, `plo`, `pch`, `pin`, `pre`
 - `idum` (integer): seed for random generators. It must be set to a negative value
- Adding code: the user does not need to edit this module

The subroutines/functions associated to this module are:

4.3.1 random_parametersup.

- Purpose: it sets the seed for the random generator with uniform distribution as a negative integer related to time

- Synopsis:

– `random_parametersup()`

- Inputs: none
- Inputs/outputs: none
- Outputs: none
- Calls: none

4.3.2 random_uniform0d.

- Purpose: long period ($> 2 \times 10^{18}$) random number generator of L'Ecuyer with Bays-Durham shuffle and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of end-points values). Use global variable `idum` which needs to be initialized with a negative value

- Synopsis:

– `random_uniform0d()`

- Inputs: none
- Inputs/outputs: none
- Outputs:

– `random_uniform0d (real)`: random number with uniform distribution between 0.0 and 1.0

- Calls: none
- Comments: this is a translation of function `ran2.f` of Numerical Recipes (see [6]) adapted to generate random numbers in single and double precision and translated to free format in Fortran 90

4.3.3 random_normalstd0d.

- Purpose: it returns a normally (standard) distributed deviate with zero mean and unit variance, using `random_uniform0d` as the source of uniform deviates

- Synopsis:

– `random_normalstd0d()`

- Inputs: none

- Inputs/outputs: none
- Outputs:
 - `random_normalstd0d` (real): random number with normal (standard) distribution
- Calls:
 - `random_normalstd0d`: from `random`
- Comments: this is a translation of function `gasdev.f` of the Numerical Recipes adapted to generate random numbers in single and double precision and translated to free format in Fortran 90

4.3.4 `random_normal0d`.

- Purpose: it returns a normally distributed deviate with mean `mu` and standard deviation `sigma`, using `random_normalstd0d` as the source of normal deviates
- Synopsis:
 - `random_normal0d(mu , sigma)`
- Inputs:
 - `mu` (real): mean
 - `sigma` (real): standard deviation
- Inputs/outputs: none
- Outputs:
 - `random_normal0d` (real): random number with normal distribution with mean `mu` and variance `sigma`².
- Calls:
 - `random_normalstd0d`: from `random`
- Comments: if x is a random variable with normal distribution with expectation 0 and variance 1, then the variable y defined by:

$$y = \mu + \sigma x, \tag{5}$$

has mean μ and variance σ^2 .

4.3.5 random_normal1dmatrix.

- Purpose: it returns a set of random vectors with normal distribution with prescribed mean and prescribed covariance matrix
- Synopsis:
 - `random_normal1dmatrix(mean , covariance , sizemean , &numbersamples , samples)`
- Inputs:
 - `mean` (real array of size `sizemean`): mean
 - `covariance` (real array of size `sizemean` \times `sizemean`): covariance matrix
 - `sizemean` (integer): size of the random vectors to be generated
 - `numbersamples` (integer): number of samples to generate
- Inputs/outputs: none
- Outputs:
 - `samples` (real array of size `sizemean` \times `numbersamples`): each column of `samples` stores a random vector
- Calls:
 - `dsyev`: from LAPACK
 - `ssyev`: from LAPACK
 - `dgemv`: from BLAS
 - `sgemv`: from BLAS
 - `random_normal0d`: from `random`
- Comments: this subroutine addresses the problem of generating a random vector $\mathbf{v} \in \mathbb{R}^n$ such that $\mathbf{v} \in \mathcal{N}(\mathbf{x}, \mathbf{P})$, where \mathbf{x} and \mathbf{P} are given. First, we compute the eigenvalues and eigenvectors of the symmetric and semi-positive definite covariance matrix \mathbf{P} , that is $\mathbf{P} = \mathbf{U}\mathbf{D}\mathbf{U}^T$. Here \mathbf{U} is an orthogonal matrix and \mathbf{D} is a diagonal matrix containing the eigenvalues σ_i in ascending order ($\sigma_i \leq \sigma_{i+1}$). Now, we take n random samples and form a vector \mathbf{a} such that $\mathbf{a}(i) \in \mathcal{N}(0, \sqrt{\sigma_i})$, $i = 1 : n$. This samples are independent one from another. Let us define:

$$\mathbf{v} = \mathbf{x} + \mathbf{U}\mathbf{a}. \quad (6)$$

Therefore, the mean of \mathbf{v} is:

$$\bar{\mathbf{v}} = \overline{\mathbf{x} + \mathbf{U}\mathbf{a}} = \mathbf{x} + \mathbf{U}\bar{\mathbf{a}} = \mathbf{x}, \quad (7)$$

and the covariance matrix of \mathbf{v} is:

$$\overline{(\mathbf{v} - \mathbf{x})(\mathbf{v} - \mathbf{x})^T} = \overline{(\mathbf{U}\mathbf{a})(\mathbf{U}\mathbf{a})^T} = \mathbf{U}\overline{\mathbf{a}\mathbf{a}^T}\mathbf{U}^T = \mathbf{U}\mathbf{D}\mathbf{U}^T = \mathbf{P}. \quad (8)$$

Thus, we have been able to generate a random vector with a required mean and covariance matrix.

4.3.6 random_normal1dsqrtmatrix.

- Purpose: it returns a set of random vectors with normal distribution with prescribed mean and prescribed reduced rank square root covariance matrix
- Synopsis:
 - random_normal1dsqrtmatrix(mean , sqrtcov , sizemean , &
 & numbermodes , numbersamples , samples)
- Inputs:
 - mean (real array of size **sizemean**): mean
 - sqrtcov (real array of size **sizemean** \times **numbermodes**): square root of the covariance matrix with reduced rank limited to **numbermodes** columns
 - sizemean (integer): size of the random vectors to be generated
 - numbermodes (integer): number of columns of the reduced rank square root of the covariance matrix
 - numbersamples (integer): number of samples to generate
- Inputs/outputs: none
- Outputs:
 - samples (real array of size **sizemean** \times **numbersamples**): each column stores a random vector
- Calls:
 - dgemv: from BLAS
 - sgemv: from BLAS
 - random_normalstd0d: from random
- Comments: This subroutine looks for random vectors with normal distribution using the reduced rank square root covariance matrix. That is, we need vectors \mathbf{v} such that $\mathbf{v} \in \mathcal{N}(\mathbf{x}, \mathbf{S}\mathbf{S}^T)$, where $\mathbf{S} \in \mathbb{R}^{n \times m}$ is a given reduced rank square root covariance matrix. In our applications we will use $n \gg m$. Let us take:

$$\mathbf{u}(i) \in \mathcal{N}(0, 1), \quad i = 1 : m. \quad (9)$$

As the samples are independent one from another, then $\mathbf{u} \in \mathcal{N}(\mathbf{0}, \mathbf{I})$. Finally, we define $\mathbf{v} = \mathbf{x} + \mathbf{S}\mathbf{u}$. Therefore, the mean of \mathbf{v} is:

$$\bar{\mathbf{v}} = \overline{\mathbf{x} + \mathbf{S}\mathbf{u}} = \mathbf{x} + \mathbf{S}\bar{\mathbf{u}} = \mathbf{x}, \quad (10)$$

and the covariance matrix of \mathbf{v} is:

$$\overline{(\mathbf{v} - \mathbf{x})(\mathbf{v} - \mathbf{x})^T} = \overline{(\mathbf{S}\mathbf{u})(\mathbf{S}\mathbf{u})^T} = \overline{\mathbf{S}\mathbf{u}\mathbf{u}^T\mathbf{S}^T} = \mathbf{S}\mathbf{S}^T. \quad (11)$$

Thus, we have been able to generate a random vector \mathbf{v} with mean equal to \mathbf{x} and covariance matrix equal to $\mathbf{S}\mathbf{S}^T$.

4.3.7 random_meanestimator1d.

- Purpose: it returns the mean of a set of vectors stored by columns in a matrix
- Synopsis:
 - `random_meanestimator1d(size , numbersamples , samples , mean)`
- Inputs:
 - `size` (integer): size of the vectors to which we will compute the mean
 - `numbersamples` (integer): number of vectors
 - `samples` (real array of size `size × numbersamples`): each column of `samples` is a vector
- Inputs/outputs: none
- Outputs:
 - `mean` (real array of size `sizemean`): mean of the vectors stored in `samples`
- Calls: none
- Comments: Given N random vectors $\mathbf{v}^1, \dots, \mathbf{v}^N$, the mean is computed simply using:

$$\frac{1}{N} \sum_{i=1}^N \mathbf{v}^i. \quad (12)$$

4.3.8 random_covarianceestimator1d.

- Purpose: it returns the covariance matrix of a set of vectors stored by columns in a given matrix
- Synopsis:
 - `random_covarianceestimator1d(size , numbersamples , samples , &mean , covariance)`
- Inputs:
 - `size` (integer): size of the random vectors
 - `numbersamples` (integer): number of random vectors
 - `samples` (real array of size `size × numbersamples`): each column of `samples` is a vector
 - `mean` (real array of size `size`): mean of the vectors stored by columns in `samples`
- Inputs/outputs: none
- Outputs:
 - `covariance` (real array of size `size × size`): covariance matrix of the vectors stored by columns in the matrix `samples`

- Calls:
 - `dgemm`: from BLAS
 - `sgemm`: from BLAS
- Comments: Given N random vectors $\mathbf{v}^1, \dots, \mathbf{v}^N$ with mean \mathbf{x} , the covariance matrix \mathbf{P} can be estimated by:

$$\begin{aligned} \mathbf{P} &= \frac{1}{N-1} \sum_{i=1}^N (\mathbf{v}^i - \mathbf{x})(\mathbf{v}^i - \mathbf{x})^T = \\ &= \frac{1}{N-1} \begin{pmatrix} \vdots & & \vdots \\ \mathbf{v}^1 - \mathbf{x} & \cdots & \mathbf{v}^N - \mathbf{x} \\ \vdots & & \vdots \end{pmatrix} \begin{pmatrix} \vdots & & \vdots \\ \mathbf{v}^1 - \mathbf{x} & \cdots & \mathbf{v}^N - \mathbf{x} \\ \vdots & & \vdots \end{pmatrix}^T. \end{aligned} \quad (13)$$

4.3.9 `random_sqrtcovestimator1d`.

- Purpose: it returns the reduced rank square root of the covariance matrix of a set of vectors stored by columns in a matrix
- Synopsis:


```
- random_sqrtcovestimator1d( size , numbersamples , samples , &
    &mean , sqrtcovariance )
```
- Inputs:
 - `size` (integer): size of the vectors
 - `numbersamples` (integer): number of vectors
 - `samples` (real array of size `size × numbersamples`): each column of `samples` is a vector
 - `mean` (real array of size `size`): mean of the vectors
- Inputs/outputs: none
- Outputs:
 - `sqrtcovariance` (real array of size `size × numbersamples`): reduced rank square root covariance matrix of the vectors stored by columns in the matrix `samples`
- Calls: none
- Comments: Given N random vectors $\mathbf{v}^1, \dots, \mathbf{v}^N$, we want to estimate the square root \mathbf{S} of the covariance matrix \mathbf{P} . From (13) is easy to deduce that:

$$\mathbf{S} = \frac{1}{\sqrt{N-1}} \begin{pmatrix} \vdots & & \vdots \\ \mathbf{v}^1 - \mathbf{x} & \cdots & \mathbf{v}^N - \mathbf{x} \\ \vdots & & \vdots \end{pmatrix}. \quad (14)$$

Also see [7], pp. 1048.

4.4 Module initialize.

This module initializes the filter, and it has to be edited by the user in order to adapt the assimilation for a specific case.

- Name: `initialize`
- Dependencies:
 - `precision`
- Global variables:
 - from `precision`: `low`, `high`, `plo`, `pch`, `pin`, `pre`
 - `dimspacestate` (integer): dimension of the space state
 - `numbersamples` (integer): number of samples
 - `modesanalysis` (integer): number of analysis modes
 - `first` (logical): flag to determine the first step
- Adding code: the user has to edit this module in order to initialize the filter

The subroutines/functions associated to this module are:

4.4.1 `initialize_parametersup`.

- Purpose: it initializes the global variables of this module
- Synopsis:
 - `initialize_parametersup()`
- Inputs: none
- Inputs/outputs: none
- Outputs: none

4.4.2 `initialize_initialize1`.

- Purpose: it sets the state and the covariance matrix at initial time
- Synopsis:
 - `initialize_initialize1(state , covariance)`
- Inputs: none
- Inputs/outputs: none
- Outputs:
 - `state` (real array of size `dimspacestate`): initial state
 - `covariance` (real array of size `dimspacestate` × `dimspacestate`): initial covariance matrix

4.4.3 initialize_initialize2.

- Purpose: it sets the state and the square root of the covariance matrix at initial time
- Synopsis:
 - `initialize_initialize2(state , sqrtcov)`
- Inputs: none
- Inputs/outputs: none
- Outputs:
 - `state` (real array of size `dimspacestate`): initial state
 - `sqrtcov` (real array of size `dimspacestate` \times `modesanalysis`): initial square root of the covariance matrix

4.5 Module model.

This module is devoted to the propagation model.

- Name: `model`
- Dependencies:
 - `precision`
- Global variables:
 - from `precision`: `low`, `high`, `plo`, `pch`, `pin`, `pre`
 - `modesmodel` (integer): number of model modes
 - `modelerror` (real array): covariance matrix of model errors
 - `sqrtmodelerror` (real array): square root of the covariance matrix of model errors
- Adding code: the user has to edit this module in order to set things related to the model

The subroutines/functions associated to this module are:

4.5.1 model_parametersup.

- Purpose: it sets the `modesmodel` variable and auxiliary variables that the user may include as global variable
- Synopsis:
 - `model_parametersup()`
- Inputs: none
- Inputs/outputs: none
- Outputs: none

4.5.2 `model_model`.

- Purpose: it applies the propagation model from time step index `t` to time step index `l+1`
- Synopsis:
 - `model_model(l , statein , stateout)`
- Inputs:
 - `l` (integer): time step index
 - `statein` (real array): state before the application of the model
- Inputs/outputs: none
- Outputs:
 - `stateout` (real array): state after applying the model
- Comments: The arrays `statein` and `stateout` have to be allocated before calling this subroutine. Inside it, `statein` and `stateout` can be of any rank, but when it is called from a module devoted to assimilation, these arrays are considered as vectors. This duality simplifies the matrix-vector mapping, and the identification is done in the Fortran style.

4.5.3 `model_tangmodel`.

- Purpose: it applies to a state vector the tangent version of the model that goes from time step index `l` to time step index `l+1`
- Synopsis:
 - `model_tangmodel(l , statein , stateout)`
- Inputs:
 - `l` (integer): time step index
 - `statein` (real array): state vector at time step index `l`
- Inputs/outputs: none
- Outputs:
 - `stateout` (real array): state vector after the application of the tangent model
- Comments: The arrays `statein` and `stateout` have to be allocated before calling this subroutine. Inside it, `statein` and `stateout` can be of any rank, but when it is called from a module devoted to assimilation, these arrays are considered as vectors. This duality simplifies the matrix-vector mapping, and the identification is done in the Fortran style.

4.5.4 model_sqrtmodelerror.

- Purpose: it sets the square root of the covariance matrix of model errors at time step index 1, that is, it sets the global variable `sqrtmodelerror`

- Synopsis:

– `model_sqrtmodelerror(1)`

- Inputs:

– 1 (integer): time step index

- Inputs/outputs: none

- Outputs: none

4.5.5 model_modelerror.

- Purpose: it sets the covariance matrix of model errors at time step index 1, that is, it sets the global variable `modelerror`

- Synopsis:

– `model_modelerror(1)`

- Inputs:

– 1 (integer): time step index

- Inputs/outputs: none

- Outputs: none

4.6 Module observations.

This module sets all things related to observations

- Name: `observations`

- Dependencies:

– `precision`

- Global variables:

– from `precision`: `low`, `high`, `plo`, `pch`, `pin`, `pre`

– `modesobs` (integer): number of observation modes

– `numberobs` (integer): number of observations

– `ifobs` (logical): flag to determine if there are observations

– `obsvalue` (real array of size `numberobs`): observation values

- `covobs` (real array of size `numberobs` \times `numberobs`): covariance matrix of observation errors
- `sqrtcovobs` (real array of size `numberobs` \times `modesobs`): square root of the covariance matrix of observation errors
- Adding code: the user has to edit this module in order to incorporate all things related to observations

The subroutines/functions associated to this module are:

4.6.1 `observations_parametersup`.

- Purpose: it sets the number of observation modes `modesobs`
- Synopsis:
 - `observations_parametersup()`
- Inputs: none
- Inputs/outputs: none
- Outputs: none

4.6.2 `observations_ifobservations`.

- Purpose: it decides if at time step index `l` there are observations or not, setting the global variable `ifobs`
- Synopsis:
 - `observations_ifobservations(l)`
- Inputs:
 - `l` (integer): time step index
- Inputs/outputs: none
- Outputs: none

4.6.3 `observations_numberobs`.

- Purpose: it sets the number of observations at time step index `l`, that is, the variable `numberobs`
- Synopsis:
 - `observations_numberobs(l)`
- Inputs:
 - `l` (integer): time step index

- Inputs/outputs: none
- Outputs: none

4.6.4 observations_obsvalue.

- Purpose: it sets the observation values at time step index `l`, that is, the variable `obsvalue`
- Synopsis:

– `observations_obsvalue(l)`

- Inputs:
 - `l` (integer): time step index
- Inputs/outputs: none
- Outputs: none

4.6.5 observations_tangobsop.

- Purpose: it applies to a state vector the tangent version observation operator at time step index `l`
- Synopsis:

– `observations_tangobsop(l , vectorin , vectorout)`

- Inputs:
 - `l` (integer): time step index
 - `vectorin` (real array): state vector
- Inputs/outputs: none
- Outputs:
 - `vectorout` (real array): state vector after the application of the tangent observation operator

4.6.6 observations_obsop.

- Purpose: it applies to a state vector the observation operator at time step index `l`
- Synopsis:

– `observations_obsop(l , vectorin , vectorout)`

- Inputs:
 - `l` (integer): time step index
 - `vectorin` (real array): state vector

- Inputs/outputs: none
- Outputs:
 - `vectorout` (real array): state vector after the application of the observation operator

4.6.7 `observations_sqrtcovobs`.

- Purpose: it sets the square root of the covariance matrix of observation errors at time step index 1, that is, the variable `sqrtcovobs`
- Synopsis:
 - `observations_sqrtcovobs(1)`
- Inputs:
 - 1 (integer): time step index
- Inputs/outputs: none
- Outputs: none

4.6.8 `observations_covobs`.

- Purpose: it sets the covariance matrix of observation errors at time step index 1, that is, the variable `covobs`
- Synopsis:
 - `observations_covobs(1)`
- Inputs:
 - 1 (integer): time step index
- Inputs/outputs: none
- Outputs: none

4.7 Module `parallel`.

This module provides procedures for parallelization.

- Name: `parallel`
- Dependencies:
 - `precision`
 - `MPI`
 - `model`
 - `observations`

- Global variables:
 - from precision: `low`, `high`, `plo`, `pch`, `pin`, `pre`
 - from MPI: variables from `mpif.h`
 - from model: `modesmodel`, `modelerror`, `sqrtmodelerror`
 - from observations: `modesobs`, `numberobs`, `ifobs`, `obsvalue`, `covobs`, `sqrtcovobs`
 - `ierror` (integer): variable to detect errors
 - `rank` (integer): which machine
 - `nproc` (integer): number of processors
 - `block_size` (integer): block size
 - `nprow` (integer): number of rows in the processor grid
 - `npcol` (integer): number of columns in the processor grid
 - `context` (integer): it is a universe where messages exist and do not interact with other context's messages
 - `myrow` (integer): calling processor's row number in the processor grid
 - `mycol` (integer): calling processor's column number in the processor grid
- Adding code: the user does not need to edit this module

The subroutines/functions associated to this module are:

4.7.1 `parallel_init`.

- Purpose: initialization of MPI
- Synopsis:
 - `parallel_init()`
- Inputs: none
- Inputs/outputs: none
- Outputs: none
- Calls:
 - `mpi_init`: from MPI
- Comments: this subroutine is a wrapper of the `mpi_init` subroutine

4.7.2 `parallel_ranksizes`.

- Purpose: it gets rank and size
- Synopsis:
 - `parallel_ranksizes()`
- Inputs: none
- Inputs/outputs: none
- Outputs: none
- Calls:
 - `mpi_comm_rank`: from MPI
 - `mpi_comm_size`: from MPI
- Comments: this subroutine is a wrapper of the `mpi_comm_rank` and `mpi_comm_size` subroutines

4.7.3 `parallel_finalize`.

- Purpose: finalization of MPI
- Synopsis:
 - `parallel_finalize()`
- Inputs: none
- Inputs/outputs: none
- Outputs: none
- Calls:
 - `mpi_finalize`: from MPI
- Comments: this subroutine is a wrapper of the `mpi_finalize` subroutine

4.7.4 `parallel_pregrid`.

- Purpose: preparation of the grid
- Synopsis:
 - `parallel_pregrid()`
- Inputs: none
- Inputs/outputs: none
- Outputs: none

- Calls:
 - `sl_init`: from SCALAPACK
 - `blacs_gridinfo`: from BLACS
- Comments: this subroutine is a wrapper of the `sl_init` and `blacs_gridinfo` subroutines

4.7.5 `parallel_postgrid`.

- Purpose: finalization of the grid
- Synopsis:
 - `parallel_postgrid()`
- Inputs: none
- Inputs/outputs: none
- Outputs: none
- Calls:
 - `blacs_gridexit`: from BLACS
- Comments: this subroutine is a wrapper of the `blacs_gridexit` subroutine

4.7.6 `parallel_predistribute`.

- Purpose: it distributes a global matrix and gets descriptors
- Synopsis:
 - `parallel_predistribute(m_global , n_global , a_global , &a_desc , m_local , n_local , a_local)`
- Inputs:
 - `m_global` (integer): number of rows of the global matrix
 - `n_global` (integer): number of columns of the global matrix
 - `a_global` (real array of size `m_global` \times `n_global`): global matrix
- Inputs/outputs: none
- Outputs:
 - `a_desc` (integer array of size 9): BLACS context handle identifying the created process grid
 - `m_local` (integer): number of rows of the local matrix
 - `n_local` (integer): number of columns of the local matrix
 - `a_local` (pointer to a real array of size `m_local` \times `n_local`): local matrix

- Calls:
 - descinit: from SCALAPACK
 - blacs_barrier: from BLACS

4.7.7 parallel_postdistribute.

- Purpose: it gathers local pieces into a global matrix
- Synopsis:
 - parallel_postdistribute(m_global , n_global , a_global , a_local)
- Inputs:
 - m_global (integer): number of rows of the global matrix
 - n_global (integer): number of columns of the global matrix
- Inputs/outputs:
 - a_local (pointer to a real array): pointer to a local matrix
- Outputs:
 - a_global (real array of size m_global \times n_global): global matrix
- Calls:
 - indx12g: from PBLAS
 - sgsum2d: from PBLAS
 - dgsum2d: from PBLAS

4.7.8 parallel_getrf.

- Purpose: it performs a LU factorization with partial pivoting
- Synopsis:
 - parallel_getrf(ma_global , na_global , a_global , ipiv_global)
- Inputs:
 - ma_global (integer): number or rows of the global matrix
 - na_global (integer): number of columns of the global matrix
- Inputs/outputs:
 - a_global (real array of size ma_global \times na_global): global matrix
- Outputs:
 - ipiv_global (integer array of size ma_global): permutations

- Calls:
 - `parallel_pregrid`: from `parallel`
 - `parallel_predistribute`: from `parallel`
 - `parallel_postdistribute`: from `parallel`
 - `parallel_postgrid`: from `parallel`
 - `psgetrf`: from SCALAPACK
 - `pdgetrf`: from SCALAPACK
 - `descset`: from SCALAPACK
 - `pslapiv`: from SCALAPACK
 - `pdlapiv`: from SCALAPACK
 - `dgetrf`: from LAPACK
 - `sgetrf`: from LAPACK
- Comments: this subroutine is a wrapper of the `pdgetrf` and `psgetrf` subroutines

4.7.9 `parallel_gesv`.

- Purpose: it gets a solution of a linear system
- Synopsis:
 - `parallel_gesv(na_global , a_global , nrhs_global , b_global)`
- Inputs:
 - `na_global` (integer): number of rows of the matrix
 - `nrhs_global` (integer): number of right hand sides
- Inputs/outputs:
 - `a_global` (real array of size $\text{na_global} \times \text{na_global}$): on entry the coefficient matrix, on exit the factors of the LU decomposition
 - `b_global` (real array of size $\text{na_global} \times \text{nrhs_global}$): on entry the right hand side matrix, on exit the solution of the system
- Outputs: none
- Calls:
 - `parallel_pregrid`: from `parallel`
 - `parallel_predistribute`: from `parallel`
 - `parallel_postdistribute`: from `parallel`
 - `parallel_postgrid`: from `parallel`
 - `psgesv`: from SCALAPACK

- pdgesv: from SCALAPACK
- descset: from SCALAPACK
- pslapiv: from SCALAPACK
- pdlapiv: from SCALAPACK
- dgesv: from LAPACK
- sgesv: from LAPACK

- Comments: this subroutine is a wrapper of the `pdgesv` and `psgesv` subroutines

4.7.10 `parallel_gemm`.

- Purpose: matrix multiplication $C = \alpha * op(A) * op(B) + \beta * C$
- Synopsis:
 - `parallel_gemm(transa , transb , ma_global , na_global , a_global , &mb_global , nb_global , b_global , mc_global , nc_global , c_global & , alpha , beta)`
- Inputs:
 - `transa` (character*1): indicates if `a_global` has to be transposed or not
 - `transb` (character*1): indicates if `b_global` has to be transposed or not
 - `ma_global` (integer): number of rows of the matrix `a_global`
 - `na_global` (integer): number of columns of the matrix `b_global`
 - `a_global` (real array of size `ma_global` \times `na_global`): matrix A
 - `mb_global` (integer): number of rows of the matrix `b_global`
 - `nb_global` (integer): number of columns of the matrix `b_global`
 - `b_global` (real array of size `mb_global` \times `nb_global`): matrix B
 - `mc_global` (integer): number of rows of the matrix `c_global`
 - `nc_global` (integer): number of columns of the matrix `c_global`
 - `alpha` (real): number `alpha`
 - `beta` (real): number `beta`
- Inputs/outputs:
 - `c_global` (real array of size `mc_global` \times `nc_global`): matrix C
- Outputs: none
- Calls:
 - `parallel_pregrid`: from `parallel`
 - `parallel_predistribute`: from `parallel`
 - `parallel_postdistribute`: from `parallel`

- `parallel_postgrid`: from `parallel`
- `psgemm`: from PBLAS
- `pdgemm`: from PBLAS
- `dgemm`: from BLAS
- `sgemm`: from BLAS
- Comments: this subroutine is a wrapper of the `pdgemm` and `psgemm` subroutines

4.7.11 `parallel_operator`.

- Purpose: parallelization of certain loops including operators using the master-slave strategy. The different options are:
 - `'tangmodelT'`: it applies the tangent model to each row of `matrixin` and stores the result in each column of `matrixout`
 - `'tangmodelN'`: it applies the tangent model to each column of `matrixin` and stores the result in each column of `matrixout`
 - `'tangobsopT'`: it applies the tangent observation operator to each row of `matrixin` and stores the result in each column of `matrixout`
 - `'tangobsopN'`: it applies the tangent observation operator to each column of `matrixin` and stores the result in each column of `matrixout`
 - `'modelN'`: it applies the model to each column of `matrixin` and stores the result to each column of `matrixout`
 - `'obsopN'`: it applies the observation operator to each column of `matrixin` and stores the result to each column of `matrixout`
- Synopsis:
 - `parallel_operator(l , matrixin , matrixout , option)`
- Inputs:
 - `l` (integer): step time index
 - `matrixin` (real array): matrix before operator is applied
 - `option` (character*): option to choose the operator
- Inputs/outputs: none
- Outputs:
 - `matrixout` (real array): matrix after operator is applied
- Calls:
 - `mpi_recv`: from MPI
 - `mpi_send`: from MPI
 - `mpi_bcast`: from MPI

- model_tangmodel: from model
- model_model: from model
- observations_tangobsop: from observations
- observations_obsop: from observations

4.7.12 parallel_gesvd.

- Purpose: it performs a singular value decomposition $A = U \star \text{SIGMA} \star VT$
- Synopsis:
 - parallel_gesvd(jobu , jobvt , ma_global , na_global , a_global , &
 &mu_global , nu_global , u_global , mvt_global , nvt_global , &
 &vt_global , msv_global , sv_global)
- Inputs:
 - ma_global (integer): number or rows of A
 - na_global (integer): number or columns of A
 - mu_global (integer): number of rows of U
 - nu_global (integer): number of columns of U
 - mvt_global (integer): number of rows of VT
 - nvt_global (integer): number of columns of VT
 - msv_global (integer): number of singular values
- Inputs/outputs:
 - jobu (character*1): flag to determine if parts of the decomposition are skipped
 - jobvt (character*1): flag to determine if parts of the decomposition are skipped
 - a_global (real array of size ma_global \times na_global): matrix A
- Outputs:
 - u_global (real array of size mu_global \times nu_global): matrix U
 - vt_global (real array of size mvt_global \times nvt_global): matrix VT
 - sv_global (real array of size msv_global): vector of singular values
- Calls:
 - parallel_pregrid: from parallel
 - parallel_predistribute: from parallel
 - parallel_postdistribute: from parallel
 - parallel_postgrid: from parallel
 - pdgesvd: from SCALAPACK
 - psgesvd: from SCALAPACK

- dgesvd: from LAPACK
- sgesvd: from LAPACK
- Comments: this subroutine is a wrapper of the `psgesvd` and `pdgesvd` subroutines

4.7.13 `parallel_svdcut`.

- Purpose: given a matrix of size `rows` \times `colsin`, the subroutine reduces its rank to `colsout` (`colsin` \geq `colsout`) taking the directions associated with the leading singular values
- Synopsis:
 - `parallel_svdcut(rows , colsin , colsout , matrixin , matrixout)`
- Inputs:
 - `rows` (integer): number of rows of the matrix `matrixin`
 - `colsin` (integer): number of columns of the matrix `matrixin`
 - `colsout` (integer): number of columns of the reduced matrix `matrixout`
 - `matrixin` (real array of size `rows` \times `colsin`): matrix to be cut
- Inputs/outputs: none
- Outputs:
 - `matrixout` (real array of size `rows` \times `colsout`): matrix `matrixin` reduced to `colsout` columns
- Calls:
 - `parallel_gemm`: from `parallel`
 - `parallel_gesvd`: from `parallel`
- Comments: In [8] the traditional reduction step using the SVD is explained, and a new reduction strategy is proposed “focusing on specific characteristics of a stochastic system with an atmospheric chemistry model”. In [9] the same strategy using SVD is implemented in the LOTOS model.

Given a square root matrix $\mathbf{S}_1 \in \mathbb{R}^{n \times m_1}$, the idea is to find a matrix $\mathbf{S}_2 \in \mathbb{R}^{n \times m_2}$ with $n \gg m_1 \geq m_2$, such that $\mathbf{S}_1 \mathbf{S}_1^T \approx \mathbf{S}_2 \mathbf{S}_2^T$. A SVD of \mathbf{S}_1 is:

$$\mathbf{S}_1 = \mathbf{U} \mathbf{D} \mathbf{V}^T, \quad \mathbf{U} \in \mathbb{R}^{n \times n}, \quad \mathbf{D} \in \mathbb{R}^{n \times m_1}, \quad \mathbf{V} \in \mathbb{R}^{m_1 \times m_1}. \quad (15)$$

The elements of \mathbf{D} are the singular values in descending order. If we compute $\mathbf{S}_1 \mathbf{S}_1^T = (\mathbf{U} \mathbf{D}) (\mathbf{U} \mathbf{D})^T$, then $\mathbf{U} \mathbf{D}_{(1:n, 1:m_2)}$ is a possible square root with the rank limited to m_2 columns, getting rid of the columns associated to the lower singular values. The problem is that for large scale applications, we could not store the \mathbf{U} matrix. Notice that from equation (15) we have that $\mathbf{S}_1 \mathbf{V} = \mathbf{U} \mathbf{D}$. If we compute $\mathbf{S}_1^T \mathbf{S}_1 = \mathbf{V} \mathbf{D}^T \mathbf{D} \mathbf{V}^T$ we can get the matrix \mathbf{V} computing a SVD of $\mathbf{S}_1^T \mathbf{S}_1$ (a matrix of dimension $m_1 \times m_1$). Thus, the reduced matrix \mathbf{S}_2 can be defined as:

$$\mathbf{S}_2 = \mathbf{S}_1 \mathbf{V}_{(1:n, 1:m_2)}. \quad (16)$$

4.8 Module ekf.

This module implements the Extended Kalman filter.

- Name: `ekf`
- Dependencies:
 - `precision`
 - `model`
 - `observations`
 - `parallel`
 - `initialize`
- Global variables:
 - from `precision`: `low`, `high`, `plo`, `pch`, `pin`, `pre`
 - from `model`: `modesmodel`, `modelerror`, `sqrtmodelerror`
 - from `observations`: `modesobs`, `numberobs`, `ifobs`, `obsvalue`, `covobs`, `sqrtcovobs`
 - from `parallel`: `ierror`, `rank`, `nproc`, `block_size`, `nprow`, `npcol`, `context`, `myrow`, `mycol`
 - from `initialize`: `dimspacestate`, `numbersamples`, `modesanalysis`, `first`
- Adding code: the user does not need to edit this module

The subroutines/functions associated to this module are:

4.8.1 ekf_predictor.

- Purpose: it makes the prediction step of the Extended Kalman filter from time step index `l` to time step index `l+1`
- Synopsis:
 - `ekf_predictor(l , state , covariance)`
- Inputs:
 - `l` (integer): time step index
- Inputs/outputs:
 - `state` (real array of size `dimspacestate`): on input the analysis at time step index `l`, on output the forecast at time step index `l+1`
 - `covariance` (real array of size `dimspacestate` \times `dimspacestate`): on input the covariance matrix of analysis errors at time step index `l`, on output the covariance matrix of forecast errors at time `l+1`
- Outputs: none

- Calls:

- `model_model`: from `model`
- `model_modelerror`: from `model`
- `parallel_operator`: from `parallel`

4.8.2 `ekf_corrector`.

- Purpose: it makes a correction step of the Extended Kalman filter at time step index `l+1`
- Synopsis:

```
– ekf_corrector( l , state , covariance )
```

- Inputs:

- `l` (integer): time step index

- Inputs/outputs:

- `state` (real array of size `dimspacestate`): on input the forecast at time step index `l+1`, on output the analysis at time step index `l+1`
- `covariance` (real array of size `dimspacestate` × `dimspacestate`): on input the covariance matrix of forecast errors at time step index `l+1`, on output the covariance matrix of analysis errors at time step index `l+1`

- Outputs: none

- Calls:

- `parallel_operator`: from `parallel`
- `parallel_getrf`: from `parallel`
- `parallel_gemm`: from `parallel`
- `observations_covobs`: from `observations`
- `observations_obsop`: from `observations`
- `dgetrs`: from LAPACK
- `sgetrs`: from LAPACK

- Comments:

- `numberobs` in this context is the number of observations at time step index `l+1`
- `obsvalue` in this context is the vector of observation values at time step index `l+1`
- `covobs` is set to the covariance matrix of observation errors at time step index `l+1` during execution, and at the end is destroyed

4.9 Module `enkf`.

This module implements the Ensemble Kalman filter.

- Name: `enkf`
- Dependencies:
 - `precision`
 - `random`
 - `model`
 - `observations`
 - `parallel`
 - `initialize`
- Global variables:
 - from `precision`: `low`, `high`, `plo`, `pch`, `pin`, `pre`
 - from `random`: `idum`
 - from `model`: `modesmodel`, `modelerror`, `sqrtmodelerror`
 - from `observations`: `modesobs`, `numberobs`, `ifobs`, `obsvalue`, `covobs`, `sqrtcovobs`
 - from `parallel`: `ierror`, `rank`, `nproc`, `block_size`, `nprow`, `npcol`, `context`, `myrow`, `mycol`
 - from `initialize`: `dimspacestate`, `numbersamples`, `modesanalysis`, `first`
- Adding code: the user does not need to edit this module

The subroutines/functions associated to this module are:

4.9.1 `enkf_predictor`.

- Purpose: it makes the prediction step of the Ensemble Kalman filter
- Synopsis:
 - `enkf_predictor(l , state , covariance , samples)`
- Inputs:
 - `l` (integer): time step index
- Inputs/outputs:
 - `state` (real array of size `dimspacestate`): on input the analysis at time step index `l`, on output the forecast at time step index `l+1`
 - `covariance` (real array of size `dimspacestate` × `dimspacestate`): on input the covariance matrix of analysis errors at time step index `l`, on output the covariance matrix of forecast errors at time step index `l+1`

- `samples` (real array of size `dimspacestate` \times `numbersamples`): on input the members of the ensemble representing possible states of the analysis at time step index `l`, on output the members of the ensemble representing possible states of the forecast at time step index `l+1`
- Outputs: none
- Calls:
 - `random_normal1dmatrix`: from `random`
 - `random_meanestimator1d`: from `random`
 - `random_covarianceestimator1d`: from `random`
 - `model_modelerror`: from `model`
 - `parallel_operator`: from `parallel`

4.9.2 `enkf_corrector`.

- Purpose: it makes the correction step of the Ensemble Kalman filter
- Synopsis:
 - `enkf_corrector(l , state , covariance , samples)`
- Inputs:
 - `l` (integer): time step index
- Inputs/outputs:
 - `state` (real array of size `dimspacestate`): on input the forecast at time step index `l+1`, on exit the analysis at time step index `l+1`
 - `covariance` (real array of size `dimspacestate` \times `dimspacestate`): on input the covariance matrix of forecast errors at time step index `l+1`, on output the covariance matrix of analysis errors at time step index `l+1`
 - `samples` (real array of size `dimspacestate` \times `numbersamples`): on input the members of the ensemble representing possible states of forecast at time step index `l+1`, on output the members of the ensemble representing possible states of analysis at time step index `l+1`
- Outputs: none
- Calls:
 - `observations_covobs`: from `observations`
 - `parallel_operator`: from `parallel`
 - `parallel_gesv`: from `parallel`
 - `parallel_gemm`: from `parallel`
 - `random_normal1dmatrix`: from `random`

- `random_meanestimator1d`: from `random`
- `random_covarianceestimator1d`: from `random`
- Comments:
 - `numberobs` in this context is the number of observations at time step index `l+1`
 - `obsvalue` in this context is the vector of observation values at time step index `l+1`
 - `covobs` is set to the covariance matrix of observation errors at time step index `l+1` during execution, and at the end is destroyed

4.10 Module `rrsqrtkf`.

This module implements the Reduced Rank Square Root Kalman filter.

- Name: `rrsqrtkf`
- Dependencies:
 - `precision`
 - `model`
 - `observations`
 - `parallel`
 - `initialize`
- Global variables:
 - from `precision`: `low`, `high`, `plo`, `pch`, `pin`, `pre`
 - from `model`: `modesmodel`, `modelerror`, `sqrtmodelerror`
 - from `observations`: `modesobs`, `numberobs`, `ifobs`, `obsvalue`, `covobs`, `sqrtcovobs`
 - from `parallel`: `ierror`, `rank`, `nproc`, `block_size`, `nprow`, `npcol`, `context`, `myrow`, `mycol`
 - from `initialize`: `dimspacestate`, `numbersamples`, `modesanalysis`, `first`
- Adding code: the user does not need to edit this module

The subroutines/functions associated to this module are:

4.10.1 `rrsqrtkf_predictor`.

- Purpose: it makes the prediction step of the Reduced Rank Square Root Kalman filter
- Synopsis:
 - `rrsqrtkf_predictor(l , state , sqrtcov , sqrtcovfor)`
- Inputs:
 - `l` (integer): time step index

- `sqrtcov` (real array of size `dimspacestate` \times `modesanalysis`): reduced rank square root of the covariance matrix of analysis errors at time step index 1
- Inputs/outputs:
 - `state` (real array of size `dimspacestate`): on input the analysis at time step index 1, on output the forecast at time step index 1+1
- Outputs:
 - `sqrtcovfor` (real array of size `dimspacestate` \times (`modesanalysis` + `modesmodel`)): on output the reduced rank square root of the covariance matrix of forecast errors at time step index 1+1
- Calls:
 - `model_model`: from `model`
 - `model_sqrtmodelerror`: from `model`
 - `parallel_operator`: from `parallel`

4.10.2 `rrsqrtkf_corrector`.

- Purpose: it makes the correction step of the Reduced Rank Square Root Kalman filter
- Synopsis:
 - `rrsqrtkf_corrector(1 , state , sqrtcov , sqrtcovfor)`
- Inputs:
 - 1 (integer): time step index
 - `sqrtcovfor` (real array of size `dimspacestate` \times (`modesanalysis` + `modesmodel`)): on input the reduced rank square root of the covariance matrix of forecast errors at time step index 1+1, on output the reduced rank square root of the covariance matrix of analysis errors at time step index 1+1 without reducing rank
- Inputs/outputs:
 - `state` (real array of size `dimspacestate`): on input the forecast at time step index 1+1, on exit the analysis at time step index 1+1
- Outputs:
 - `sqrtcov` (real array of size `dimspacestate` \times `modesanalysis`): reduced rank square root of the covariance matrix of analysis errors at time step index 1+1
- Calls:
 - `observations_sqrtcovobs`: from `observations`
 - `observations_obsop`: from `observations`
 - `parallel_operator`: from `parallel`

- `parallel_gemm`: from `parallel`
- `parallel_gesv`: from `parallel`
- `parallel_svdcut`: from `parallel`
- Comments:
 - `numberobs` in this context is the number of observations at time step index `l+1`
 - `obsvalue` in this context is the vector of observation values at time step index `l+1`
 - `sqrtcovobs` is set to the reduced rank square root of the covariance matrix of observation errors at time step index `l+1` during execution, and at the end is destroyed

4.11 Module `rrsrtenkf`.

This module implements the Reduced Rank Square Root Ensemble Kalman filter.

- Name: `rrsrtenkf`
- Dependencies:
 - `precision`
 - `random`
 - `model`
 - `observations`
 - `parallel`
 - `initialize`
- Global variables:
 - from `precision`: `low`, `high`, `plo`, `pch`, `pin`, `pre`
 - from `random`: `idum`
 - from `model`: `modesmodel`, `modelerror`, `sqrtmodelerror`
 - from `observations`: `modesobs`, `numberobs`, `ifobs`, `obsvalue`, `covobs`, `sqrtcovobs`
 - from `parallel`: `ierror`, `rank`, `nproc`, `block_size`, `nrow`, `ncol`, `context`, `myrow`, `mycol`
 - from `initialize`: `dimspacestate`, `numbersamples`, `modesanalysis`, `first`
- Adding code: the user does not need to edit this module

The subroutines/functions associated to this module are:

4.11.1 `rrsrtenkf_predictor`.

- Purpose: it makes the prediction step of the Reduced Rank Square Root Ensemble Kalman filter

- Synopsis:

– `rrsrtenkf_predictor(l , state , sqrtcov , samples , sqrtcovaux)`

- Inputs:

- `l` (integer): time step index
- `sqrtcov` (real array of size `dimspacestate` × `modesanalysis`): reduced rank square root of the covariance matrix of analysis errors at time step index `l`

- Inputs/outputs:

- `state` (real array of size `dimspacestate`): on input the analysis at time step index `l`, on output the forecast at time step index `l+1`
- `samples` (real array of size `dimspacestate` × `numbersamples`): on input the members of the ensemble representing possible states of analysis at time step index `l`, on output the members of the ensemble representing possible states of forecast at time step index `l+1`

- Outputs:

- `sqrtcovaux` (real array of size `dimspacestate` × `numbersamples`): on output the reduced rank square root of the covariance matrix of forecast errors at time step index `l+1`

- Calls:

- `random_normal1dsqrtmatrix`: from `random`
- `random_meanestimator1d`: from `random`
- `random_sqrtcovestimator1d`: from `random`
- `parallel_operator`: from `parallel`
- `model_sqrtmodelerror`: from `model`

4.11.2 `rrsrtenkf_corrector`.

- Purpose: it makes the correction step of the Reduced Rank Square Root Ensemble Kalman filter

- Synopsis:

– `rrsrtenkf_corrector(l , state , sqrtcov , samples , sqrtcovaux)`

- Inputs:

- `l` (integer): time step index

- Inputs/outputs:
 - `state` (real array of size `dimspacestate`): on input the forecast at time step index `l+1`, on exit the analysis at time step index `l+1`
 - `samples` (real array of size `dimspacestate` \times `numbersamples`): on input the members of the ensemble representing possible states of forecast at time step index `l+1`, on output the members of the ensemble representing possible states of analysis at time step index `l+1`
 - `sqrtcovaux` (real array of size `dimspacestate` \times `numbersamples`): on input the reduced rank square root of the covariance matrix of forecast errors at time step index `l+1`, on output the reduced rank square root of the covariance matrix of analysis errors at time step index `l+1` without reducing rank
- Outputs:
 - `sqrtcov` (real array of size `dimspacestate` \times `modesanalysis`): reduced rank square root of the covariance matrix of analysis errors at time step index `l+1`
- Calls:
 - `parallel_operator`: from `parallel`
 - `parallel_gemm`: from `parallel`
 - `parallel_gesv`: from `parallel`
 - `parallel_svdcut`: from `parallel`
 - `observations_sqrtcovobs`: from `observations`
 - `random_normal1dsqrtmatrix`: from `random`
 - `random_meanestimator1d`: from `random`
 - `random_sqrtcovestimator1d`: from `random`
- Comments:
 - `numberobs` in this context is the number of observations at time step index `l+1`
 - `obsvalue` in this context is the vector of observation values at time step index `l+1`
 - `sqrtcovobs` is set to the reduced rank square root of the covariance matrix of observation errors at time step index `l+1` during execution, and at the end is destroyed

5 How to use the package.

In order to set up an assimilation run, there are optional and mandatory modules, as well as subroutines. We will distinguish the cases according to the Kalman filter method:

5.1 Case ekf.

The mandatory modules are: `precision`, `model`, `observations`, `parallel`, `initialize` and `ekf`. The user has to edit the following subroutines:

- `model_model(1 , statein , stateout)`: first define rank and size of `statein` and `stateout`. Then set `stateout` as the model (going from time step index 1 to time step index 1+1) applied to `statein`.
- `model_tangmodel(1 , statein , stateout)`: first define rank and size of `statein` and `stateout`. Then set `stateout` as the tangent model (going from time step index 1 to time step index 1+1) applied to `statein`.
- `model_modelerror(1)`: set variable `modelerror` as the covariance matrix of model errors at time step index 1. No allocation required because it is performed in the `ekf` module.
- `observations_ifobservations(1)`: set variable `ifobs`. This flag is `TRUE` in case there are observations at time step index 1 and `FALSE` if not.
- `observations_numberobs(1)`: set variable `numberobs` as the number of observations at time step index 1.
- `observations_obsvalue(1)`: set variable `obsvalue` as a vector (or array) containing observation values at time step index 1. No allocation is required because it is performed in the main program.
- `observations_tangobsop(1 , vectorin , vectorout)`: first define rank and size of `vectorin` and `vectorout`. Then set `vectorout` as the tangent observation operator at time step index 1) applied to `vectorin`.
- `observations_obsop(1 , vectorin , vectorout)`: first define rank and size of `vectorin` and `vectorout`. Then set `vectorout` as the observation operator at time step index 1) applied to `vectorin`.
- `observations_covobs(1)`: set variable `covobs` as the covariance matrix of observation errors at time step index 1. No allocation required because it is performed in the `ekf` module.
- `initialize_parametersup()`: set variable `dimspacestate` as the size of the state vector.
- `initialize_initialize1(state , covariance)`: set `state` as the initial state vector and `covariance` as the initial covariance matrix.

The remaining subroutines that are included in the modules and not mentioned above should be kept without code in the subroutine body.

5.2 Case `enkf`.

The mandatory modules are: `precision`, `random`, `model`, `observations`, `parallel` and `initialize`. The user has to edit the following subroutines:

- `model_model(l , statein , stateout)`: modify in the same way as in the `ekf` case.
- `model_modelerror(l)`: modify in the same way as in the `ekf` case.
- `observations_ifobservations(l)`: modify in the same way as in the `ekf` case.
- `observations_numberobs(l)`: modify in the same way as in the `ekf` case.
- `observations_obsvalue(l)`: modify in the same way as in the `ekf` case.
- `observations_tangobsop(l , vectorin , vectorout)`: modify in the same way as in the `ekf` case.
- `observations_obsop(l , vectorin , vectorout)`: modify in the same way as in the `ekf` case.
- `observations_covobs(l)`: modify in the same way as in the `ekf` case.
- `initialize_parametersup()`: set variable `dimspacestate` as the size of the state vector, set variable `numbersamples` as the number of samples in the ensemble, and set variable `first` to `TRUE`.
- `initialize_initialize1(state , covariance)`: modify in the same way as in the `ekf` case.

The remaining subroutines that are included in the modules and not mentioned above should be kept without code in the subroutine body.

5.3 Case `rrsqrtkf`.

The mandatory modules are: `precision`, `model`, `observations`, `parallel` and `initialize`. The user has to edit the following subroutines:

- `model_parametersup()`: set variable `modesmodel` as the number of modes considered for the covariance matrix of model errors.
- `model_model(l , statein , stateout)`: modify in the same way as in the `ekf` case.
- `model_tangmodel(l , statein , stateout)`: modify in the same way as in the `ekf` case.
- `model_sqrtmodelerror(l)`: set variable `sqrtmodelerror` as the square root of the covariance matrix of model errors.
- `observations_parametersup()`: set variable `modesobs` as the number of modes considered for the covariance matrix of observation errors.
- `observations_ifobservations(l)`: modify in the same way as in the `ekf` case.

- `observations_numberobs(1)`: modify in the same way as in the `ekf` case.
- `observations_obsvalue(1)`: modify in the same way as in the `ekf` case.
- `observations_tangobsop(1 , vectorin , vectorout)`: modify in the same way as in the `ekf` case.
- `observations_obsop(1 , vectorin , vectorout)`: modify in the same way as in the `ekf` case.
- `observations_sqrtcovobs(1)`: set variable `sqrtcovobs` as the square root of the covariance matrix of observation errors.
- `initialize_parametersup()`: set variable `dimspacestate` as the size of the state vector, set variable `modesanalysis` as the number of modes for the covariance matrix of analysis errors.
- `initialize_initialize2(state , sqrtcov)`: set `state` as the initial vector and `sqrtcov` as the initial square root of the covariance matrix.

The remaining subroutines that are included in the modules and not mentioned above should be kept without code in the subroutine body.

5.4 Case `rrsqrtenkf`.

The mandatory modules are: `precision`, `random`, `model`, `observations`, `parallel` and `initialize`. The user has to edit the following subroutines:

- `model_parametersup()`: modify in the same way as in the `rrsqrtnkf` case.
- `model_model(1 , statein , stateout)`: modify in the same way as in the `ekf` case.
- `model_sqrtmodelerror(1)`: modify in the same way as in the `rrsqrtnkf` case.
- `observations_parametersup()`: modify in the same way as in the `rrsqrtnkf` case.
- `observations_ifobservations(1)`: modify in the same way as in the `ekf` case.
- `observations_numberobs(1)`: modify in the same way as in the `ekf` case.
- `observations_obsvalue(1)`: modify in the same way as in the `ekf` case.
- `observations_tangobsop(1 , vectorin , vectorout)`: modify in the same way as in the `ekf` case.
- `observations_obsop(1 , vectorin , vectorout)`: modify in the same way as in the `ekf` case.
- `observations_sqrtcovobs(1)`: modify in the same way as in the `rrsqrtnkf` case.
- `initialize_parametersup()`: set variable `dimspacestate` as the size of the state vector, set variable `numbersamples` as the number of samples in the ensemble, set variable `modesanalysis` as the number of modes for the covariance matrix of analysis errors and set variable `first` to `TRUE`.

- `initialize_initialize2(state , sqrtcov)`: modify in the same way as in the `rrsqrtkf` case.

5.5 An example of a main program.

Once the user has set all the subroutines related to the Kalman filter implementation to be used, a main program is required in order to set the simulation. Below there is a description of a general main program using `ekf` module. For other implementations the main program is very similar. For specific details, please refer to the files attached to this documentation.

program main

```

    !* INCLUDING MODULES
    use precision
    use random
    use model
    use observations
    use parallel
    use initialize
    use ekf
    ...

    implicit none

    !* DECLARATION OF VARIABLES
    ...

    !* INITIALIZATION OF PARALLELIZATION
    call parallel_init()

    !* GETTING PARALLELIZATION PARAMETERS
    call parallel_ranksize()

    !* SETTING PARAMETERS AND BROADCASTING
    if ( rank == 0_pin ) then
        call initialize_parametersup()
    end if
    call mpi_bcast(...)
    ...

    !* ALLOCATIONS
    allocate(...)
    ...

    !* INITIALIZATION
    if ( rank == 0_pin ) then
        call initialize_initialize1( state , covariance )
    
```

```

end if
call mpi_bcast(...)

!* LOOP

do l = 1 , ...

    !* PREDICTION
    call ekf_predictor( l , state , covariance )

    !* SET IFOBS
    if ( rank == 0_pin ) then
        call observations_ifobservations( l + 1_pin )
    end if
    call mpi_bcast(...)

    if ( ifobs ) then !* IN CASE THERE ARE OBSERVATIONS

        !* SET NUMBEROBS
        if ( rank == 0_pin ) then
            call observations_numberobs( l + 1_pin )
        end if
        call mpi_bcast(...)

        !* ALLOCATE OBSVALUE
        allocate( obsvalue( numberobs ) )

        !* SET OBSVALUE
        if ( rank == 0_pin ) then
            call observations_obsvalue( l + 1_pin )
        end if
        call mpi_bcast(...)

        !* CORRECTION
        call ekf_corrector( l , state , covariance )

        !* DEALLOCATE OBSVALUE
        deallocate( obsvalue )

    end if

end do

!* DEALLOCATIONS
...

!* END PARALLELIZATION

```

```

    call parallel_finalize()

end program

```

6 Examples.

This section presents four examples of application of the package. In the first example the model is the Euler's method for solving ordinary differential equations and we assimilate the numerical solution. In the second example a finite difference explicit scheme to solve a partial differential equation is considered as the model and the numerical solution is assimilated. In the third example we consider the model MATCH [10] and CO concentrations are assimilated. Finally, the fourth example is devoted to the model POLAIR [11] and O3 concentrations are considered for assimilation.

The tests were compiled with the following compilers: g95, GNU Fortran, Intel Fortran Compiler and Portland Fortran Compiler. The examples run sequentially in a PC running linux, Pentium 4, 2.4 GHz., and 512MB of RAM. The examples were also tested in a cluster with 16 nodes, 2 Intel(R) Itanium(TM) 2 CPU 1.6 GHz with 4 GB RAM, the operative system is Rocks 4.2.1 IA64, data interconnect is $2 \times$ GB Ethernet (Cooper) and compute interconnect is infiniband DDR 4×10 Gbps.

6.1 Example 0D.

6.1.1 Problem.

We will consider an ordinary differential equation:

$$\begin{cases} \dot{x}(t) &= f(t, x), & t > a, \\ x(a) &= x_a. \end{cases} \quad (17)$$

If in (17) we set:

$$a = 0, \quad (18)$$

$$x_a = 2, \quad (19)$$

$$f(t, x) = \cos(t), \quad (20)$$

we can check that the exact solution of (17) is:

$$x_{sol}(t) = 2 + \sin(t). \quad (21)$$

6.1.2 Domain.

Let us choose Δt positive and let us define a number of time steps n_t . Therefore, the time domain is discretized as:

$$t_l = a + (l - 1) \Delta t, \quad l = 1 : n_t. \quad (22)$$

In the experiment, the setting is:

$$n_t = 250, \quad (23)$$

$$\Delta t = 0.1. \quad (24)$$

6.1.3 State vectors.

The state vector will be of size 1 and it is defined as $\mathbf{x}_l = [x]$ where $x \in \mathbb{R}$ represents an estimation of $x_{sol}(t_l)$.

The true state at time step index l is defined as: $\mathbf{x}_l^t = [x(t_l)]$.

6.1.4 Model.

Using the Euler's method the model that propagates the state forward in time can be defined as:

$$M_{l \rightarrow l+1}(\mathbf{x}) = [x + \Delta t f(t_l, x)]. \quad (25)$$

It can be easily deduced that the tangent model is:

$$\mathbf{M}_{l \rightarrow l+1}(\mathbf{x}) = \left[1 + \Delta t \frac{\partial f}{\partial x}(t_l, x) \right]. \quad (26)$$

6.1.5 Model errors.

Using a Taylor's expansion, it is easy to obtain an approximation of the model error:

$$\begin{aligned} M_{l \rightarrow l+1}(\mathbf{x}_l^t) - \mathbf{x}_{l+1}^t &\doteq [q_l] \approx \\ &\approx \left[-\frac{(\Delta t)^2}{2} \left(\frac{\partial f}{\partial t}(t_l, x_{sol}(t_l)) + \frac{\partial f}{\partial x}(t_l, x_{sol}(t_l)) f(t_l, x_{sol}(t_l)) \right) \right]. \end{aligned} \quad (27)$$

Then, the covariance matrix of model errors can be set as:

$$\mathbf{Q}_l = [q_l^2]. \quad (28)$$

If we consider the square root of the covariance matrix of model errors, we should set:

$$\mathbf{S}_l^m = [|q_l|]. \quad (29)$$

6.1.6 Observations.

We define the observation vector as \mathbf{y} . We will observe the same state vector. The observations will be given each 5 time steps. The number of observations `numberobs` for each observation step is 1. The observation values are built from a perturbation of the exact solution, that is,

$$\mathbf{y}_l \in \mathcal{N}(x_{sol}(t_l), \sigma_l), \quad \sigma_l = c |x_{sol}(l)|, \quad (30)$$

where c is a positive constant. Thus σ_l represents a fraction of the absolute value of the exact solution. In the experiment we set $c = 0.3$.

6.1.7 Observation errors.

We can set the covariance matrix of observation errors to:

$$\mathbf{R}_l = [\sigma_l^2]. \quad (31)$$

If we consider the square root of the covariance matrix of observation errors, we should set:

$$\mathbf{S}_l^o = [|\sigma_l|]. \quad (32)$$

6.1.8 Observation operator and the tangent observation operator.

As we observe the same state vector, the observation operator as well as the tangent observation operator is the identity function:

$$H_l(\mathbf{x}) = \mathbf{x} = \mathbf{H}_l(\mathbf{x}). \quad (33)$$

6.1.9 Initialization.

The state vector will be initialized with a perturbation of x_a . The objective is to apply the assimilation in order to get a better estimation of the exact solution. In the experiment, the initial conditions were:

$$\mathbf{x} = [4], \quad \mathbf{P} = [4], \quad \mathbf{S} = [|2|]. \quad (34)$$

6.1.10 Experiment.

A comparison between the exact solution, the numerical solution (with and without assimilation), and the observations, is shown in the figure 1. This figure is a clear example of the need of performing data assimilation. The errors in the initial conditions can take the model far away from the solution. That is why we need to correct the numerical solution using observations. Notice that reduced rank methods give the same results than the full methods because we are in 0D. Some auxiliary settings are listed below:

```
!* precision parameters
plo = 4
pch = 1
pin = 4
pre = 8

!* random parameters
idum = -1

!* model parameters
modesmodel = 1

!* observation parameters
modesobs = 1

!* initialization parameters
dimspacestate = 1
numbersamples = 100
modesanalysis = 1
first          = .TRUE.
```

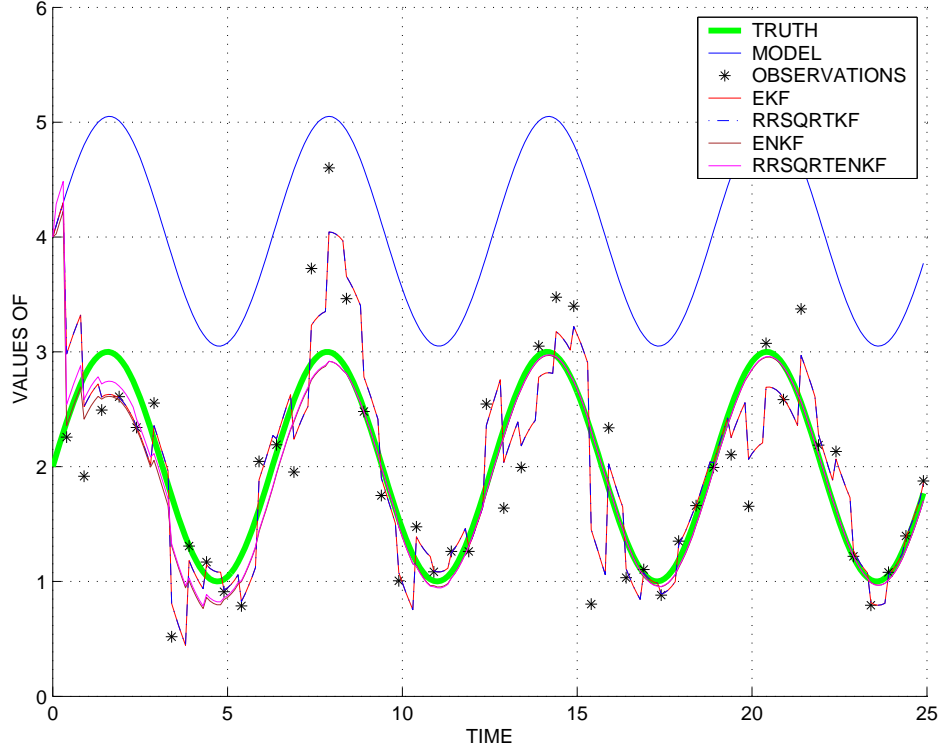


Figure 1: Comparisons of the exact, model, and assimilation solution.

6.2 Example 2D.

6.2.1 Problem.

The problem consists of solving the following equation:

$$\begin{cases} \frac{\partial c}{\partial t} + u \frac{\partial c}{\partial x} + v \frac{\partial c}{\partial y} - \alpha \frac{\partial^2 c}{\partial x^2} - \beta \frac{\partial^2 c}{\partial y^2} = f, & x \in \Omega, \quad t \in [0, \infty), \\ c = g, & x \in \partial\Omega, \quad t = 0, \end{cases} \quad (35)$$

where the functions u, v, α, β, f and g , and the space domain Ω are known.

If in (35) we set:

$$u = 1, \quad (36)$$

$$v = 1, \quad (37)$$

$$\alpha = 1, \quad (38)$$

$$\beta = 1, \quad (39)$$

$$\begin{aligned} f(t, x, y) = & 1000 \cos(1000t) \sin(5xy) e^{-\frac{1}{2}x} + 10 \sin(1000t) \cos(5xy) y e^{-\frac{1}{2}x} - \\ & - \frac{3}{4} \sin(1000t) \sin(5xy) e^{-\frac{1}{2}x} + 5 \sin(1000t) \cos(5xy) x e^{-\frac{1}{2}x} + \\ & + 25 \sin(1000t) \sin(5xy) y^2 e^{-\frac{1}{2}x} + 25 \sin(1000t) \sin(5xy) x^2 e^{-\frac{1}{2}x}, \end{aligned} \quad (40)$$

$$g(t, x, y) = 3 + \sin(1000t) \sin(5xy) e^{-\frac{1}{2}x}. \quad (41)$$

we can check that the exact solution of (17) for any Ω is:

$$c_{sol}(t, x, y) = 3 + \sin(1000t) \sin(5xy) e^{-\frac{1}{2}x}. \quad (42)$$

In the experiment we choose:

$$\Omega = [-1, 1] \times [-1, 1]. \quad (43)$$

6.2.2 Domain.

Let us set Δt , Δx and Δy positive numbers, and let us define the number of time steps n_t and the number of nodes n_x and n_y in x and y directions respectively. The domain is discretized as:

$$t_l = (l - 1) \Delta t, \quad l = 1 : n_t, \quad (44)$$

$$x_i = -1 + (i - 1) \Delta x, \quad i = 1 : n_x, \quad (45)$$

$$y_j = -1 + (j - 1) \Delta y, \quad j = 1 : n_y. \quad (46)$$

In the experiment we set:

$$n_t = 1110, \quad (47)$$

$$n_x = 21, \quad (48)$$

$$n_y = 21. \quad (49)$$

6.2.3 State vectors.

The state vector will be of size $n_x n_y$ and it is defined as a matrix of size $n_x \times n_y$ representing an estimation of the exact solution in the discretized domain. That is, at time step index l the true state is given by:

$$\mathbf{x}_{ij}^t = c_{sol}(t_l, x_i, y_j), \quad i = 1 : n_x, \quad j = 1 : n_y \quad (50)$$

The dimension of the space state is $n_x n_y$.

6.2.4 Model.

To solve the problem, we use an explicit finite difference scheme. Thus, we can define the model that propagates the state as:

$$\begin{aligned} [M_{l \rightarrow l+1}(\mathbf{x}_l)]_{ij} &= \mathbf{x}_{lij} - \frac{u_{lij} \Delta t}{2 \Delta x} (\mathbf{x}_{li+1j} - \mathbf{x}_{li-1j}) - \frac{v_{lij} \Delta t}{2 \Delta y} (\mathbf{x}_{lij+1} - \mathbf{x}_{lij-1}) + \\ &+ \frac{\alpha_{lij} \Delta t}{(\Delta x)^2} (\mathbf{x}_{li+1j} - 2\mathbf{x}_{lij} + \mathbf{x}_{li-1j}) + \\ &+ \frac{\beta_{lij} \Delta t}{(\Delta y)^2} (\mathbf{x}_{lij+1} - \mathbf{x}_{lij} + \mathbf{x}_{lij-1}) + f_{lij} \Delta t, \\ l &= 1 : n_t - 1, \quad i = 2 : n_x - 1, \quad j = 2 : n_y - 1. \end{aligned} \quad (51)$$

In the border the model will be a perturbation of the exact boundary condition at time step $l + 1$, that is,

$$[M_{l \rightarrow l+1}(\mathbf{x}_l)]_{ij} \in \mathcal{N}(c_{sol}(t_{l+1}, x_i, y_j), \sigma_{l+1ij}^b), \quad \sigma_{l+1ij}^b = c |c_{sol}(t_{l+1}, x_i, y_j)|, \quad (52)$$

$$i = 1, \quad j = 1 : n_y, \quad (53)$$

$$i = n_x, \quad j = 1 : n_y, \quad (54)$$

$$i = 1 : n_x, \quad j = 1, \quad (55)$$

$$i = 1 : n_x, \quad j = 1 : n_y, \quad (56)$$

where c is a positive constant. Thus σ_{l+1ij}^b represents a fraction of the absolute value of the exact solution. In the experiment we set $c = 0.5$.

The tangent model is the same as the model, but in the boundary is set to zero.

6.2.5 Model errors.

If we make an analysis of the discretization errors (for example, using a Taylor's expansion), we deduce that the model error for an interior node (i, j) is given by:

$$\begin{aligned} [M_{l \rightarrow l+1}(\mathbf{x}_l^t) - \mathbf{x}_{l+1}^t]_{ij} &\doteq q_{lij} \approx \\ &\approx \frac{(\Delta t)^2}{2} \frac{\partial^2 c_{sol}}{\partial t^2}(t_l, x_i, y_j) + u_{lij} \frac{\Delta t \Delta x^2}{6} \frac{\partial^3 c_{sol}}{\partial x^3}(t_l, x_i, y_j) + \end{aligned} \quad (57)$$

$$+ v_{lij} \frac{\Delta t \Delta y^2}{6} \frac{\partial^3 c_{sol}}{\partial y^3}(t_l, x_i, y_j) - \alpha_{lij} \frac{\Delta t \Delta x^2}{12} \frac{\partial^4 c_{sol}}{\partial x^4}(t_l, x_i, y_j) - \quad (58)$$

$$- \beta_{lij} \frac{\Delta t \Delta y^2}{12} \frac{\partial^4 c_{sol}}{\partial y^4}(t_l, x_i, y_j). \quad (59)$$

For a boundary node, the model error is given by:

$$[M_{l \rightarrow l+1}(\mathbf{x}_l^t) - \mathbf{x}_{l+1}^t]_{ij} = \sigma_{l+1ij}^b{}^2. \quad (60)$$

For the case of the square root of the covariance matrix of model errors we consider 200 modes.

6.2.6 Observations.

For the example, we set the following measuring stations:

- Station 1 located at nodes $i_1 = 6$ and $j_1 = 3$.
- Station 2 located at nodes $i_2 = 14$ and $j_2 = 3$.
- Station 3 located at nodes $i_3 = 3$ and $j_3 = 6$.
- Station 4 located at nodes $i_4 = 10$ and $j_4 = 6$.
- Station 5 located at nodes $i_5 = 18$ and $j_5 = 6$.
- Station 6 located at nodes $i_6 = 6$ and $j_6 = 10$.
- Station 7 located at nodes $i_7 = 14$ and $j_7 = 10$.
- Station 8 located at nodes $i_8 = 3$ and $j_8 = 14$.
- Station 9 located at nodes $i_9 = 10$ and $j_9 = 14$.
- Station 10 located at nodes $i_{10} = 18$ and $j_{10} = 14$.
- Station 11 located at nodes $i_{11} = 6$ and $j_{11} = 18$.
- Station 12 located at nodes $i_{12} = 14$ and $j_{12} = 18$.

We define the observation vector as \mathbf{y} , where \mathbf{y} is of size 12 containing the measurement of each station. We will observe the same variable. The observations will be given each 5 time steps. The number of observations `numberobs` for each observation step is 12. The observation values are built from a perturbation of the exact solution, that is,

$$\mathbf{y}_{ls} \in \mathcal{N}(c_{sol}(t_l, x_{is}, y_{js}), \sigma_{ls}^o), \quad \sigma_{ls}^o = c |c_{sol}(t_l, x_{is}, y_{js})|, \quad s = 1 : 12, \quad (61)$$

where c is a positive constant. Thus σ_{ls}^o represents a fraction of the absolute value of the exact solution. In the experiment we set $c = 0.3$.

6.2.7 Observation errors.

We can set the covariance matrix of observation errors to a diagonal matrix, that is:

$$\mathbf{R}_{lss} = \sigma_{ls}^{o^2}, \quad s = 1 : 12. \quad (62)$$

The number of observation modes is set to 12, that is:

$$\mathbf{S}_{lss}^o = |\sigma_{ls}^o|, \quad s = 1 : 12. \quad (63)$$

6.2.8 Observation operator and the tangent observation operator.

As we are observing the same type of variables, the observation operator is simply a projection:

$$H_l(\mathbf{x}) = \begin{bmatrix} \mathbf{x}_{i_1 j_1} \\ \mathbf{x}_{i_2 j_2} \\ \mathbf{x}_{i_3 j_3} \\ \mathbf{x}_{i_4 j_4} \\ \mathbf{x}_{i_5 j_5} \\ \mathbf{x}_{i_6 j_6} \\ \mathbf{x}_{i_7 j_7} \\ \mathbf{x}_{i_8 j_8} \\ \mathbf{x}_{i_9 j_9} \\ \mathbf{x}_{i_{10} j_{10}} \\ \mathbf{x}_{i_{11} j_{11}} \\ \mathbf{x}_{i_{12} j_{12}} \end{bmatrix} \quad (64)$$

The tangent observation operator is the same as the observation operator because it is linear.

6.2.9 Initialization.

The state vector will be initialized with a perturbation of the exact solution at $t = 0$. That is, for a node (i, j) :

$$\mathbf{x}_{ij} \in \mathcal{N}(c_{sol}(t_0, x_i, y_j), \sigma_{ij}^I), \quad \sigma_{ij}^I = c |c_{sol}(t_0, x_i, y_j)|, \quad i = 1 : n_x, \quad j = 1 : n_y. \quad (65)$$

The covariance matrix at initial time is defined as a diagonal matrix. In the row corresponding to the node (i, j) we set the covariance matrix element as $\sigma_{ij}^{I^2}$. For the square root of the initial covariance matrix we take 200 modes.

6.2.10 Experiment.

A list of auxiliary settings is listed below:

```
!* precision parameters
plo = 4
pch = 1
pin = 4
pre = 8

!* random parameters
idum = -1

!* initialization parameters
dimstate = 441
numbersamples = 50
first      = .TRUE.
```

We can see a comparison of the methods in figure 2. The graphics show the percentage of the relative error of the model and assimilation against the true solution in the last step. For example, for the case of the Extended Kalman filter, the figure 2 shows a plot of

$$\left| \frac{\mathbf{x}_{ij}^{ekf} - \mathbf{x}_{ij}^t}{\mathbf{x}_{ij}^t} \right| * 100. \quad (66)$$

The maximum values of the plots are:

- Model: 77%
- EKF: 10%
- ENKF: 15%
- RRSQRTKF: 26%
- RRSQRTENKF: 27%

We see that without assimilation the model produces an error of 77%. The EKF gives the best results, as well as the ENKF. The EKF is a little better than ENKF, because the ENKF uses a finite set of possible states. The reduced rank square methods are comparable, and the error is around 26%. That is because we are getting rid of a certain number of columns in the square root covariance matrices. From the experiments one can see that the reduced rank methods need a period of time in order to have an effective reduction of the error. The reason for that might be that the modes that represent the covariance matrices need to adjust themselves to represent the dynamical system.

6.3 Example 3D - Assimilation of CO.

The package has been used to assimilate CO concentrations in the area of Santiago de Chile. The implementation was done using the MATCH model [10]. The version of the filter applied was the Reduced Rank Square Root Kalman filter with 50 samples.

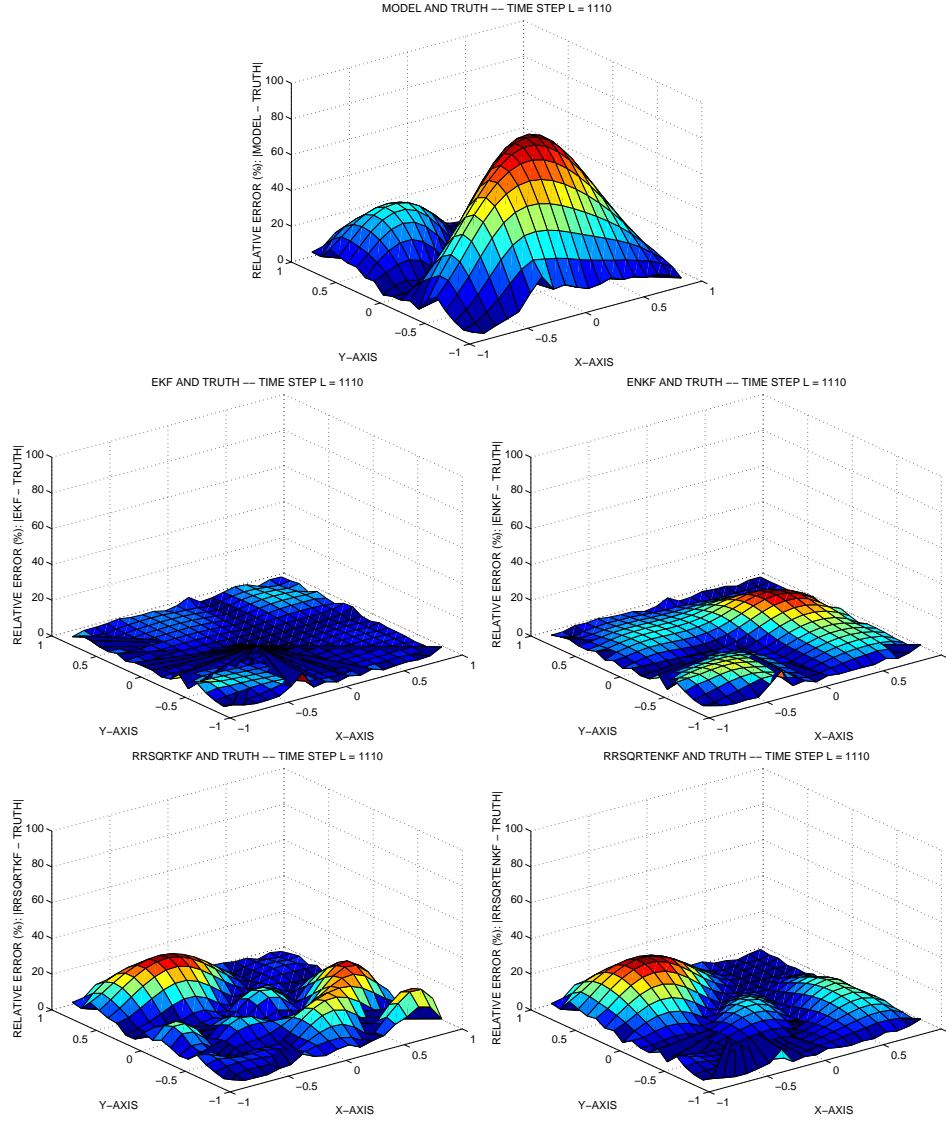


Figure 2: Comparison of methods.

A grid of 41×41 is considered in the horizontal domain, and 16 levels in the vertical component. Then, the dimension of the space state is set to 26896. For the initialization, the model was run for a period of three days considering an atmosphere free of CO at the beginning. The initial state vector is set to the last output of the MATCH initial run, adding an error of 100%. The simulation period was 13 days starting at June 17th, 1999, and ending at June 30th of the same year, performing an analysis each 3 hours (the time step for observations). The meteorological fields were generated using the HIRLAM model [12] with a resolution of 0.01 degrees (≈ 1 km.) and at 1 hour of time resolution.

The CO emissions were generated by MODEM [13]. See figure 3.

There are 8 monitoring stations located at different positions in Santiago. The observations are taken from the measuring stations at the surface level, at intervals of 3 hours, where the analysis step was performed. In figure 3 we show the eighth monitoring stations and the domain of simulation: (1) Seminario, (2) Independencia-Recoleta and (5) Parque OHiggins are in the city center of Santiago; (3) La Florida takes measurements in the east and south area; (4)

Las Condes - Vitacura is monitoring the north-east sector; (6) Pudahuel-Cerro Navia and (7) Cerrillos register measurements at the west side of the city; and (8) El Bosque is located at the south. The error in the observations was set to 30% of the measured value.

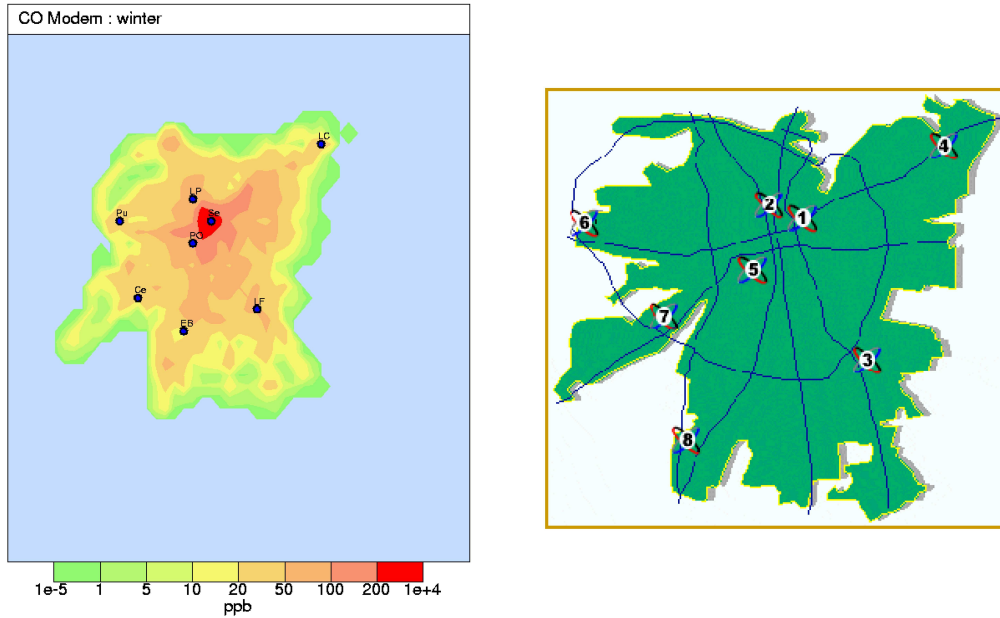


Figure 3: Emissions generated by MODEM and stations.

After 100 hours of simulation running assimilation, we obtain the figure 4, where we can see a comparison of the model, observations, truth and assimilation:

6.4 Example 3D - Assimilation of O3.

Polair3D [11] is a 3D Eulerian chemistry transport model developed at ENPC (École Nationale des Ponts et Chaussées). It has been used for passive transport [14], impact at European scale, photochemistry [15] and mercury chemistry. The model has been validated through comparisons with other models and with measurements provided by campaigns. Several chemical mechanisms are available like RADM2, RACM, EURORADM, MOCA and CBMIV. All the parameterizations are computed in preprocessed steps, so Polair3D is a numerical platform for solving advection-diffusion-reaction partial differential equations.

Polair3D is part of the Polyphemus system, although it can be used independently. Even when assimilation is already implemented in Polyphemus, we have used the libraries presented here with very little effort. The Polair3D model has about 65000 lines of Fortran 77 source code, and in order to implement assimilation, the user needs to write about 1000 lines of Fortran code, that is, about 1.5% of modifications.

The simulations uses a twin experiment, that is, one simulation is considered as the “truth”, from which we will build the observations perturbing the solution. Then we perturb some input data, and the objective is to recover truth using assimilation.

Before the user adds assimilation, it must be understood how the model works, how it is compiled, and how it is run. For the case of Polair3D, the model is called through a script called POLAIR written using the Korn Shell, which function is mainly compilation and execution. Program options are passed through hidden files and symbolic links. Following that script, users can write a Makefile in order to compile the code and avoid scripting.

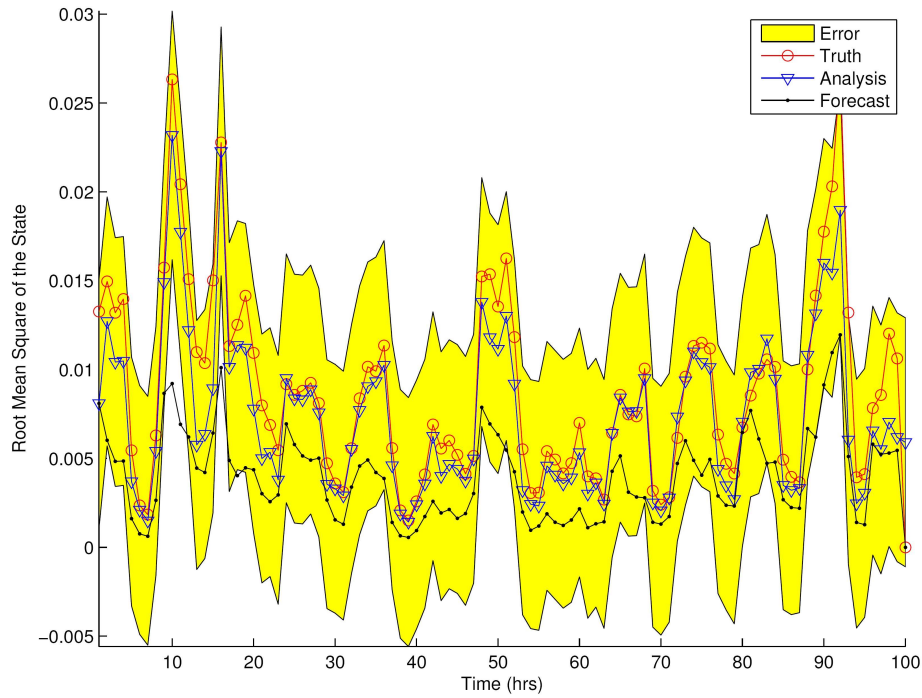


Figure 4: Results of the CO assimilation.

The second thing to be taken into account, is to find in the source code the main program, and the subroutine that performs a time step. For Polair3D the main program is called `CTMASTER.f`, and its function is simply initialization and calling to the subroutine that runs the model, namely, `ctm.f`. This last subroutine has to be edited. The subroutine `ctm.f` can be divided into the following parts: (a) declarations, (b) initializations and (c) time loop. In the time loop we can identify a section which is in charge of the preprocessing, propagation and preparation for the following time step (for example, writing files).

According to section 5.4 we need to edit the following functions:

- `model_parametersup`: we set 50 model modes.

```
subroutine model_parametersup()
  implicit none
  modesmodel = 50_pin
end subroutine model_parametersup
```

- `model_model`: this a wrapper of the subroutine that propagates the state one step forward in time.

```
subroutine model_model( l , statein , stateout )
  implicit none
  integer( kind = pin ) , intent( in ) :: l
  real( kind = pre ) , dimension( nx , ny , nz ) :: statein
  real( kind = pre ) , dimension( nx , ny , nz ) :: stateout
  dlcg( : , : , : , 50_pin ) = statein
```

```

    !* this is the subroutine that propagates the concentrations dlcg
    call polair_calcconc()
    !* the index 50 corresponds to the O3 species
    stateout = dlcg( : , : , : , 50_pin )
end subroutine model_model

```

- `model_sqrtmodelerror`: set the square root of the covariance matrix of model errors.

```

subroutine model_sqrtmodelerror( l )
  implicit none
  integer( kind = pin ) , intent( in ) :: l
  integer( kind = pin ) :: i
  sqrtmodelerror = 0.0_pre
  do i = 1 , modesmodel
    sqrtmodelerror( i , i ) = 1.0_pre
  end do
end subroutine model_sqrtmodelerror

```

- `observations_parametersup`: setting some parameters.

```

subroutine observations_parametersup()
  implicit none
  obsstep = 1_pin !* observations at every step
  no = 1800_pin / obsstep !* number of time observations
  errorobs = 0.05_pre !* observations considered with a 5% of error
  modesobs = 35_pin !* we consider 35 observation modes
end subroutine observations_parametersup

```

- `observations_ifobservations`: set the `ifobs` variable.

```

subroutine observations_ifobservations( l )
  implicit none
  integer( kind = pin ) , intent( in ) :: l
  if ( mod( l , obsstep ) == 0_pin ) then
    ifobs = .true.
  else
    ifobs = .false.
  end if
end subroutine observations_ifobservations

```

- The following subroutine is auxiliar and sets up the location of 35 measuring stations.

```

subroutine observations_stationsup()
  implicit none
  integer( kind = pin ) :: s
  nstat = 35_pin
  ...
end

```

- `observations_numberobs`: set the number of observations as the number of stations.

```
subroutine observations_numberobs( l )
  implicit none
  integer( kind = pin ) , intent( in ) :: l
  numberobs = nstat
end subroutine observations_numberobs
```

- `observations_obsvalue`: the observation values are built from a twin experiment.

```
subroutine observations_obsvalue( l )
  implicit none
  integer( kind = pin ) , intent( in ) :: l
  ...
  ...
  do s = 1_pin , numberobs
    aux = dble( o3_truth( indexstationsx( s ) , indexstationsy( s ) , &
      &indexstationsz( s ) , 1 ) )
    sigmaobs( s ) = errorobs * abs( aux )
    obsvalue( s ) = random_normal0d( aux , sigmaobs( s ) )
  end do
end subroutine observations_obsvalue
```

- `observations_obsop`:

```
subroutine observations_obsop( l , vectorin , vectorout )
  implicit none
  integer( kind = pin ) , intent( in ) :: l
  real( kind = pre ) , dimension( nx , ny , nz ) :: vectorin
  real( kind = pre ) , dimension( numberobs ) :: vectorout
  integer( kind = pin ) :: s
  do s = 1_pin , numberobs
    vectorout( s ) = vectorin( indexstationsx( s ) , &
      &indexstationsy( s ) , indexstat&
      &ionsz( s ) )
  end do
end subroutine observations_obsop
```

- `observations_tangobsop`: it is the same as the observation operator because it is linear.
- `observations_sqrtcovobs`: setting the square root of the covariance matrix of observation errors.

```
subroutine observations_sqrtcovobs( l )
  implicit none
  integer( kind = pin ) , intent( in ) :: l
  integer( kind = pin ) :: s
```



```

    sqrtcovobs = 0.0_pre
    do s = 1_pin , modesobs
        sqrtcovobs( s , s ) = sigmaobs( s )
    end do
end subroutine observations_sqrtcovobs

```

- `initialize_parametersup()`: set assimilation parameters.

```

subroutine initialize_parametersup()
    implicit none
    dimspacestate = nx * ny * nz
    numbersamples = 50_pin
    first = .true.
end subroutine initialize_parametersup

```

- `initialize_initialize2`: set initializations.

```

subroutine initialize_initialize2( state , sqrtcov )
    implicit none
    real( kind = pre ) , dimension( dimspacestate ) :: state
    real( kind = pre ) , dimension( dimspacestate , modesana&
        &lysis ) :: sqrtcov
    integer :: i , j , k , m
    call polair_init()
    m = 1_pin
    do k = 1_pin , nz
        do j = 1_pin , ny
            do i = 1_pin , nx
                state( m ) = dlcg( i , j , k , 50_pin )
                m = m + 1
            end do
        end do
    end do
    sqrtcov = 0.0_pre
    do m = 1 , modesanalysis
        sqrtcov( m , m ) = 1.0_pre
    end do
end subroutine initialize_initialize

```

In order that the code is consistent, we need to add three modules:

- `module_polaircommon.f`: it gathers all the common files already existing in the Polair3d source code,
- `module_polairformat.f90`: it is a set of subroutines to read/write Polair3D input/output files,
- `module_polairinit.f90`: it is a wrapper of the initialization Polair3D files,

- `module_polair.f90`: it is a re-writting of the `ctm.f` file, considering the three main tasks: preprocessing, propagation and postprocessing.

These four modules have to be included in the main program according to Section 5.5:

```
program main
```

```
use precision
use random
use polair
use model
use observations
use initialize
use rrsqrtenkf

implicit none

!* declarations
...

!* parallel initialization
call parallel_init()
call parallel_ranksize()

!* Polair3d initialization
call polairinit_init()

!* calling all the parameters and broadcasting
call random_parametersup()
...

!* allocations
...

!* filter initialization
call initialize_initialize( state , sqrtcov )

!* time loop
do jt = 1 , nstop

    !* preparing propagation / this subroutine is a wrapper
    call polair_previous()

    !* prediction
    if ( jt == 1_pin ) then
        first = .true.
        call rrsqrtenkf_predictor( jt , state , sqrtcov , samples , &
                                   &sqrtcovaux )
```

```

    else
        call rrsqrtenkf_predictor( jt , state , sqrtcov , samples , &
                                &sqrtecovaux )
    end if

    !* decide if there are measurements
    call observations_ifobservations( jt + 1_pin )
    if ( ifobs ) then

        !* getting number of observations
        call observations_numberobs( jt + 1_pin )

        !* getting observations
        allocate( obsvalue( numberobs ) )
        call observations_obsvalue( jt + 1_pin )

        !* correction
        call rrsqrtenkf_corrector( jt , state , sqrtcov , samples , &
                                &sqrtecovaux )
        deallocate( obsvalue )

    end if

    !* write for the next step
    dlcg( : , : , : , 50_pin ) = state

    !* postprocess after propagation / this subroutine is a wrapper
    call polair_post()

end do

!* deallocations
...

!* finalize parallelization
call parallel_finalize()

end program main

```

Summarizing, the steps needed for the implementation were: (a) understanding how the model is compiled and run (b) identification of the subroutine that makes the propagation of one time step and what it is needed to call it (c) coding of modules if necessary, (d) writing wrappers of the model, (e) choosing assimilation options, (f) writing the main program and (g) writing a Makefile.

Even when the settings can be improved in order to have a better assimilation, the explanation above is useful because it serves as a guide to adapt models to assimilation with a very little lines of source code.

References

- [1] <http://www.netlib.org/blas/>
- [2] <http://www.netlib.org/lapack/>
- [3] <http://www.netlib.org/blacs/>
- [4] http://www.netlib.org/scalapack/scalapack_home.html
- [5] <http://www-unix.mcs.anl.gov/mpi/>
- [6] <http://www.nr.com/>
- [7] D. Treebushny and H. Madsen, *On the Construction of a Reduced Rank Square-Root Kalman Filter for Efficient Uncertainty Propagation*, Future Generation Computer Systems, vol. 21, pp. 1047–1055 (2005).
- [8] A. Segers, A. Heemink, M. Verlaan and M. van Loon, *A Modified RRSQRT-Filter for Assimilating Data in Atmospheric Chemistry Models*, Environmental Modelling & Software, vol. 15, pp. 663–671 (2000).
- [9] M. van Loon and A. Heemink, *Kalman Filtering for Non Linear Atmospheric Chemistry Models: First Experiences*, Technical Report MAS-R9711, CWI, Amsterdam (1997).
- [10] <http://www.smhi.se/sgn0106/if/FoU1/en/models/match/match.html>
- [11] <http://cerea.enpc.fr/polair3d/>
- [12] <http://hirlam.org/>
- [13] http://www.sectra.cl/contenido/metodologia/transporte_medioambiente/estimacion_emisiones_fuentes_moviles_modem.asp
- [14] J. -P. Issartel and J. Baverel, *Inverse Transport for the Verification of the Comprehensive Test Ban Treaty*, Atmos. Chem. Phys., 3, 475–486, 2003.
- [15] K. N. Sartelet, J. Boutahar, D. Quélo, I. Coll, P. Plion and B. Sportisse, *Development and validation of a 3D Chemistry-Transport Model, POLAIR3D, by comparison with data from ESQUIF campaign*, Proceedings of the 6th Gloream Workshop: Global and Regional Atmospheric Modelling, 2002.