Miniker manual

for Miniker version 102, 2 November 2008

The TEF Collaboration

Table of Contents

Introduction

Miniker is a modeling tool, built especially in order to implement models written following the TEF (Transfer Evolution Formalism) formalism, a mathematical framework for system analysis and simulation. Miniker allows for timewise simulation, system analysis, adjoint computation, Kalman filtering and more.

Miniker uses a fortran preprocessor, `mortran`, designed in the 1970's, to ease model writing using dedicated specific languages. For example partial derivatives are symbolicaly determined by `mortran` macros in Miniker. For the selection of another compile-time features, another set of preprocessor directives, the *cmz directives*, are used. In most cases the user does not need to know anything about that preprocessing that occurs behind the scene, he simply writes down the equations of his model and he is done.

A comprehensive description of the TEF formalism in available on http://www.lmd.jussieu.fr/ZOOM/doc/tef-GB-partA5.pdf). The Miniker software is a reduced version of **ZOOM**, that can only handle a hundreds of variables, but is much easier to use.

Intended audience

The reader should have notions in system dynamics. Moreover a minimal knowledge of programmation and fortran is required. What is required is a basic understanding of variable types, affectation and fortran expressions.

Reading guide

The first chapter is a brief overview of the TEF. The following describes how to write, compile and run a model in Miniker in its basic and comprehensive syntax. Reading up to the section *Controlling the run* is required to be able to use Miniker. In this section it is assumed that Miniker is properly setup. The installation instructions are in the appendix at Appendix A [Installation], page 48.

The next chapter describes advanced features, first a general introduction to features settings and then a description of other model description related features.

The next chapter describes system analysis tools available with Miniker. The sections are independant and each describes how to use a specific feature. If you plan on using these features, you should also read Section 3.1 [Overview of feature setting], page 15.

A final chapter describes advanced features in a development environment using make,

In the appendix the instructions for the installation are described (see Appendix A [Installation], page 48).

Other Manuals and documentation

A programmers'Manual is available (in French), and can be asked for to any member of the collabration. See additional documents in http://www.lmd.jussieu.fr/Zoom/doc or ask for Research texts and articles to members.

1 An overview of the TEF formalism

The TEF (Transfer Evolution Formalism) is based on partitionning and recoupling of model subsystems. It allows the study of the coupling between subsystems by the means of linearization and time discretization.

1.1 Cell and Transfer equations

In the TEF, a model is mathematically represented by a set of equations corresponding to two kinds objects:

1. Cells which are elementary models and correspond to evolution equations such as:

$$\partial_t \eta(t) = g(\eta(t), \varphi(t))$$

   Vector η represent the state variables of cells and the vector φ represent the dependent boundary conditions, *i.e.* the variables considered as boundary conditions by a cell, but depending upon the complete model state. This dependent boundary conditions are required to make the cells correspond to well-posed problems. These variables are often called state variables, and prognostic variables in meteorology.

2. Transfers which are determined by constraint equations such as:

$$\varphi(t) = f(\eta(t), \varphi(t))$$

   These equations are often called algebraic equations, and in meteorology diagnostic equations.

1.2 Linearization and discretization in the TEF

The relations between sub-systems is excessively difficult to exhibit when having to cope with non-linear system. In the TEF, the TLS (Tangent Linear System) is constructed along the trajectory. One considers the system over a small portion along the trajectory, say between t and $t + \delta t$. The variation $\delta\eta$ of η and $\delta\varphi$ of φ is obtained through a Padé approximation of the state-transition matrix. The final form of the algebraic system is closed to the classical Crank-Nicolson scheme:

$$\begin{pmatrix} A & B \\ -C^+ & I - D \end{pmatrix} \begin{pmatrix} \delta\eta \\ \delta\varphi \end{pmatrix} = \begin{pmatrix} \Gamma \\ \Omega \end{pmatrix}$$

The blocks appearing in the Jacobian matrix are constructed with partial derivative of f and g, and with δt. From this system the elimination of $\delta\eta$ leads to another formulation giving the coupling between transfers, and allows for the $\delta\varphi$ computation. The $\delta\varphi$ value is then substitued in $\delta\eta$ to complete the time-step solving process.

2 Miniker model programming

Miniker works by combining the model specification code given by the user and other source files provided in the package. The code is assembled, preprocessed, compiled, linked and the resulting program can be run to produce the model trajectory and dynamic analysis.

The code provided in the package contains a principal program, some usefull subroutines and pieces of code called *sequences* combined with the different codes. Among these sequences some hold the code describing the model and are to be written by the user (sequences are similar to Fortran include files).

2.1 General structure of the code

The sequences used to enter model description hold the mathematical formulae for each cell and transfer component, dedicated derived computations, and time-step steering. During the code generation stage, cmz directives are preprocessed, then the user pseudo-Fortran instructions are translated by `mortran` using macros designed to generate in particular all Fortran instructions that compute the Jacobian matrices used in TEF modelling.

The sequence '`zinit`' contains the mathematical formulation of the model (see Section 2.2.2 [Model equation and parameters], page 4). Another sequence, '`zsteer`', is merged at the end of the time step advance of the simulation, where the user can monitor the time step values and printing levels, and perform particular computations etc. (see Section 2.4.1 [Executing code at the end of each time step], page 12).

2.2 Miniker programming illustrated

The general TEF system writes:

$$\begin{aligned} \partial_t \eta(t) &= g(\eta(t), \varphi(t)) \\ \varphi(t) &= f(\eta(t), \varphi(t)) \end{aligned}$$

To illustrate the model description in Miniker a simple predator-prey model of Lotka-Volterra is used. This model can be written in the following TEF form:

$$\begin{cases} \partial_t \eta_{prey} = a\eta_{prey} - a\varphi_{meet} \\ \partial_t \eta_{pred} = -c\eta_{pred} + c\varphi_{meet} \end{cases}$$
$$\varphi_{meet} = \eta_{prey}\eta_{pred}$$

with two cell equations, *i.e.* state evolution of the prey and predator groups, and one transfer accounting for the meeting of individuals of different group.

2.2.1 All you need to know about mortran and cmz directives

The first stage of code generation consists in cmz directives preprocessing. Cmz directives are used for conditional selection of features, and sequence inclusion. At that point you don't need to know anything about these directives. They are only usefull if you want to take advantage of advanced features (see Section 3.10 [Programming with cmz directives], page 27).

The code in sequences is written in Mortran and the second stage of code generation consists in mortran macro expansion. The mortran language is described in its own manual, here we only explain the very basics which is all you need to know to use Miniker. Mortran basic instructions are almost Fortran, the differences are the following:

- The code is free-form, and each statement should end with a semi-colon `;`.
- Comments may be introduced by an exclamation mark `!` at the beginning of a line, or appear within double quotes `"` in a single line.
- It is possible to use blocs, for `do` or `if` statement for example, and they are enclosed within brackets '`<`' and '`>`'. To be in the safe side, a semi-colon `;` should be added after a closng bracket `>`.

The following fictious code is legal mortran:

```
real
  param;
param = 3.; ff(1) = ff(3)**eta(1);       "a comment"
! a line comment
do inode=1,n_node <eta_move(inode)=0.01; eta_speed(inode)=0.0;>;
```

Thanks to mortran the model code is very simply specified, as you'll see next.

2.2.2 Entering model equation and parameters

The model equation and parameters and some Miniker parameters are entered in the '`zinit`' sequence. The whole layout of the model is given before detailing the keywords.

```
!%%%%%%%%%%%%%%%%%%%%
! Parameters
!%%%%%%%%%%%%%%%%%%%%
  real apar,bpar;        "optional Fortran type declaration"

! required parameters
     dt=.01;             "initial time-step"
     nstep=10 000;       "number of iterations along the trajectory"
     time=0.;            "time initialisation "

! model parameters
     apar = 1.5;
     cpar = 0.7;

! misceallaneous parameters
     modzprint = 1000;    "printouts frequency"

print*,'************************************';
print*,'Lotka-Volterra model with parameters as:';
z_pr: apar,bpar;
print*,'************************************';

!%%%%%%%%%%%%%%%%%%%%
! Transfer definition
```

```
!%%%%%%%%%%%%%%%%%%%%%
! rencontre (meeting)
set_Phi
< var: ff_interact, fun: f_interact = eta_prey*eta_pred;
>;

!%%%%%%%%%%%%%%%%%%%%%
! Cell definition
!%%%%%%%%%%%%%%%%%%%%%

set_eta
< var: eta_prey, fun: deta_prey =   apar*eta_prey - apar*ff_interact;
  var: eta_pred, fun: deta_pred = - cpar*eta_pred + cpar*ff_interact;
>;


!%%%%%%%%%%%%%%%%%%%%%
! Initial states
!%%%%%%%%%%%%%%%%%%%%%
     eta_prey = 1.;
     eta_pred = 1.;
;
    OPEN(50,FILE='title.tex',STATUS='UNKNOWN');   "title file"
    write(50,5000) apar,cpar;
5000;format('Lotka-Volterra par:',2F4.1);
```

Variables and model parameters

The following variables are mandatory:

`dt` The time step.

`time` Model time initialisation.

`nstep` Number of iterations along the trajectory.

There are no other mandatory variables. Some optional variables are used to monitor the printout and ouput of results of the code. As an example, the variable `modzprint` is used to set the frequency of the printout of the model matrix and vectors during the run (see Section 2.4.2 [Controlling the printout and data output], page 14).

User's defined variable and Fortran or Mortran instructions can always be added for intermediate calculus. To avoid conflict with the variables of the Miniker code, the rule is that a users symbol must not have characters '`o`' in the first two symbol characters.

In the predator-prey example there are two model parameters. The fortran variables are called here `apar` for a and `cpar` for c. If a Fortan type definition is needed, it should be set at the very beginning of '`zinit`'. The predator-prey code variable initializations finally reads

```
!%%%%%%%%%%%%%%%%%%%%%
! Parameters
!%%%%%%%%%%%%%%%%%%%%%
  real apar,bpar;        "optional Fortran type declaration"

     dt=.01;
     nstep=10 000;
     time=0.;

! model parameters
     apar = 1.5;
     cpar = 0.7;

     modzprint = 1000;
```

Model equations

The model equations for cells and model equations for transferts are entered in two mortran blocks, one for the transferts, the other for the cell components. The model equations for cells are entered into a `set_eta` block, and the transfer equations are entered into a `set_phi` block.

In each block the couples variable-function are specified. For transfers the function defines the transfer itself while for cells the function describes the cell evolution. The variable is specified with `var:`, the function is defined with `fun:`.

In the case of the predator-prey model, the transfer variable associated with φ_{meet} could be called `ff_interact` and the transfer definition would be given by:

```
set_Phi
< var: ff_interact, fun: f_interact = eta_prey*eta_pred;
>;
```

The two cell equations of the predator-prey model, with name `eta_prey` for the prey (η_{prey}) and `eta_pred` for the predator (η_{pred}) are:

```
set_eta
< var: eta_prey, fun: deta_prey =   apar*eta_prey - apar*ff_interact;
  var: eta_pred, fun: deta_pred = - cpar*eta_pred + cpar*ff_interact;
>;
```

The ‘;’ at the end of the mortran block is important.

The whole model equations are setup with:

```
!%%%%%%%%%%%%%%%%%%%%%
! Transfer definition
!%%%%%%%%%%%%%%%%%%%%%
! rencontre (meeting)
set_Phi
< var: ff_interact, fun: f_interact = eta_prey*eta_pred;
>;

!%%%%%%%%%%%%%%%%%%%%%
! Cell definition
!%%%%%%%%%%%%%%%%%%%%%

set_eta
< var: eta_prey, fun: deta_prey =   apar*eta_prey - apar*ff_interact;
  var: eta_pred, fun: deta_pred = - cpar*eta_pred + cpar*ff_interact;
>;
```

Whenever the user is not concerned with giving a specific name to a function, it is possible to specify the equation only with `eqn:`. Therefore the user may replace an instruction as:

```
var: ff_dump,
fun: f_dump  = - rd*(eta_speed - eta_speed_limiting);
```

with:

```
eqn: ff_dump = - rd*(eta_speed - eta_speed_limiting);
```

In that case, the unnamed function will take the name of the defined variable preceded by the '`$`' sign: `$ff_dump`.

Starting points

The cells equations require state initial conditions. In some case, the transfers may also need starting points although they are determined from the cell values.

In the predator-prey model the starting points for cells are:

```
!     initial state
!     -------------
     eta_prey = 1.;
     eta_pred = 1.;
```

When there is a non trivial implicit relationship between the transfers in the model, it may be usefull or even necessary to set some transfers to non-zero values. This difficulty is only relevant for the very first step of the simulation and will be used as a first guess of φ. The uninitialized transfers having a default compiler-dependant (often zero) value, an initialization to another value may help avoiding singular functions or matrix and ensure convergence in the Newton algorithm used to solve the transfer implicit equation.

The cell and transfer arrays

Sometime it is easier to iterate over an array than to use the cell or transfer variable name. This is possible because there is a correspondence between the variable names and the

fortran array `eta(.)` for the cell variables and the fortran array `ff(.)` for the transfer variables[1].

The index of the variable is determined by the order of appearance in the variable definition blocks. It is reminded in the output, as explained later (see Section 2.3.3 [Simulation and output], page 11).

The number of cells is in the integer `np` variable, and the number of transfer is in the integer `mp` variable.

title file

For some graphics generation, a file with name '`title.tex`' is required which sets the title. The following instructions take care of that:

```
      OPEN(50,FILE='title.tex',STATUS='UNKNOWN');
      write(50,5000) apar,cpar;
5000;format('Lotka-Volterra par:',2F4.1);

      close(50);
```

In that case the parameter values are written down, to differenciate between different runs. This step is in general not needed.

2.3 Setting and running a model

In this section it is assumed that a programming environment has been properly setup. This environment may use either cmz or make to drive the preprocessing and compilation. You can skip the part related with the environment you don't intend to use.

For instructions regarding the installation, see Appendix A [Installation], page 48.

2.3.1 Setup a model and compile with cmz

The user defined sequences are '`KEEP`' in the cmz world. The most common organization is to have a cmz file in a subdirectory of the directory containing the '`mini_ker.cmz`' cmz file. In this cmz file there should be a '`PATCH`' called '`zinproc`' with the KEEPs within the patch. The KEEP must be called '`$zinit`'.

From within cmz in the directory of your model the source extraction, compilation and linking will be triggered by a `mod` command. This macro uses the '`selseq.kumac`' information to find the '`mini_ker.cmz`' cmz file. `mod` shall create a directory with the same name than the cmz file, '`mymodel/`' in our example. In this directory there is another directory '`cfs/`' containing the sources extracted from the cmz file.

The file '`mymodel_o.tmp`' contains all the mortran code generated by cmz with the sequences substituted, including the '`$zinit`'. The fortran produced by the preprocessing and splitting of this file is in files with the traditional '`.f`' suffix. The principal program is in '`principal.f`'. An efficient way of getting familiar with mini_ker methods is looking at the '`mymodel_o.tmp`' where all sequences and main Mortran instructions are gathered.

[1] In fact the variables names are transformed into fortran array elements by mortran generated macros, so the symbolic names defined in the mortran blocks never appears in the generated fortran code, they are replaced by the fortran arrays.

Symbolic derivation is noted as `F_D(expression)(/variable)`, and the resulting Fortran code is in '`principal.f`'.

`mod` also triggers compilation and linking. The object files are in the same '`cfs/`' directory and the executable is in the '`mymodel/`' directory, with name '`mymodel.exe`'.

2.3.2 Setup a model and compile with make

With make, the sequences are files ending with '`.mti`' (for mortran include files), called, for example, '`zinit.mti`'. They are included by `mortran` in other source files. You also need a '`Makefile`' to drive the compilation of the model.

If you don't need additional code or libraries to be linked with your model you have two alternatives.

1. The simplest alternative is to run the `start_miniker` script with the model file name as argument. It should copy a '`zinit.mti`' file ready to be edited and a Makefile ready to compile the model. For the predator prey model, for example, you could run

   ```
   $ start_miniker predator
   ```

2. Otherwise you can copy the Makefile from '`template/Makefile`' in the directory containing the sequences. You should then change the compiled model file name, by changing the value of the `model_file_name` variable to the name of your choice in the Makefile. It is set to '`mymodel`' in the template. For the predator-prey model, it could be set like

   ```
   model_file_name = predator
   ```

   If you want the executable model file to be built in another directory, you could set

   ```
   model_file_name = some_dir/predator
   ```

   The other items set in the default Makefile should be right.

The preprocessing and the compilation are launched with

```
make all
```

The mortran files are generated by the cmz directive preprocessor from files found in the package source directories. The mortran files end with '`.mtn`' for the main files and '`.mti`' for include files. They are output in the current directory. The mortran preprocessor then preprocess these mortran files and includes the sequences. The resulting fortran code is also in the current directory, in files with a '`.f`' suffix. Some fortran files ending with '`.F`' may also be created by the cmz directive preprocessor. The object files resulting from the compilation of all the fortran files (generated from mortran or directly from fortran files) are there too.

In case you want to override the default sequences or a subroutine file you just have to create it in your working directory along with the '`zinit.mti`'. For example you could want to create or modify a '`zsteer.mti`' file (see Section 2.4.1 [Executing code at the end of each time step], page 12), a '`zcmd_law.mti`' file (see Section 4.2.2 [Control laws], page 32), a '`monitor.f`' file (see Section 3.2.1 [Turning the model into a subroutine], page 15) to take advantage of features presented later in this manual.

More in-depth discussion of using make to run Miniker is covered in Chapter 5 [Advanced use of Miniker with make], page 42. For example it is also possible to create files that are to be preprocessed by the cmz directive preprocessor and separate source files and generated

files. This advanced use is more precisely covered in Section 3.10 [Programming with cmz directives], page 27.

2.3.3 Running a simulation and using the output

Once compiled the model is ready to run, it only has to be executed. On standard output informations about the states, transfers, tangent linear system and other jacobian matrices are printed. For example the predator-prey model could be executed with:

```
./predator > result.lis
```

The correspondance between the symbolic variables and the basic vectors and functions are printed at run time:

```
---------------- Informing on Phi definition -----------------
  Var-name,           Function-name,       index in ff vector
           ff_interact              f_interact  1
--------------------------------------------------

--------------- Informing on Eta definition -----------------
 Var-name,           Function-name,       index in eta vector
              eta_prey               deta_prey  1
              eta_pred               deta_pred  2
```

A summary of the model equations are in '`Model.hlp`' file. For the same example:

```
====================== set_Phi

    1 ff_interact f_interact           eta_pray*eta_pred
====================== set_Eta

    1 eta_pray    deta_pray            apar*eta_pray-apar*ff_interact
    2 eta_pred    deta_pred            -cpar*eta_pred+cpar*ff_interact
```

when other general functions are specified with `f_set`, it can appear also in the same help file when replaced by `fun_set`.

As far as possible, all data printed in the listing are associated with a name related to a variable. Here is an extract:

```
Gamma :-8.19100E-02-1.42151E-01 3.87150E-02
        eta_courant eta_T_czcx  eta_T_sz
       ------------------------------------------------
Omega : 0.00000E+00 0.00000E+00 0.00000E+00 0.00000E+00
        courant_L   T_czcx      Psi_Tczc    Psi_Tsz
       ------------------------------------------------
```

for the two known vectors of the system, and:

```
 >ker : Matrice de couplage       4 4 4 4
courant_L Raw(1,j=1,4):   1.000     -9.9010E-03  0.000       0.000
T_czcx    Raw(2,j=1,4): -2.7972E-02   1.000      0.000      9.9900E-04
Psi_Tczcx Raw(3,j=1,4):  0.1605      9.7359E-02  1.000     -5.7321E-03
Psi_Tsz   Raw(4,j=1,4):   0.000     -0.1376     5.7225E-03   1.000
          Var-Name      courant_L   T_czcx      Psi_Tczc    Psi_Tsz
          ------------------------------------------------------------
```

where the `couplage` (coupling matrix) is given that corresponds to the matrix coupling the four transfer components after $\delta\eta$ has been eliminated from system. It is computed in the subprogram '`oker`' (for kernel) which solves the system.

Basic results are output in a set of '`.data`' files. The first line (or two lines) describes the column with a '`#`' character used to mark the lines as comments (for `gnuplot` for example). In the '`.data`' files, the data are simply separated with spaces. Each data file has the `time` variable values as first column.[1]. Following columns give the values of `eta(.)` in '`res.data`', `dEta(.)` in '`dres.data`' – the step by step variation of `eta(.)` – and `ff(.)` in '`tr.data`'.

Along the simulation the TEF Jacobian matrices are computed. A transfer variables elimination process also leads to the definition of the classical state advance matrix of the system (the corresponding array is `aspha(.,.)` in the code). This matrix is output in the file '`aspha.data`' that is used to post-run dynamics analyses. The matrix columns are written column wise on each record. See Section 4.5 [Stability analysis of fastest modes], page 39. See Section 4.6 [Generalized tangent linear system analysis], page 40. It is not used in the solving process.

Other '`.data`' files will be described later.

2.3.4 Doing graphics

Since the data are simply separated with spaces, and comment lines begin with '`#`', the files can be vizualised with many programs. With `gnuplot`, for example, to plot `eta(`*`n`*`)`, the `gnuplot` statement could be:

```
plot "res.data" using 1:(n+1)
```

The similar one for `ff(`*`n`*`)`:

```
plot "tr.data" using 1:(n+1)
```

For people using `PAW`, the CERN graphical computer code, Miniker prepares kumacs that allow to read process the '`.data`' files in the form of *n-tuples* (see the *PAW manual* for more information). In that cas, the flag `sel paw` has to be gievn in the '`selsequ.kumac`'. The generated n-tuples are ready to use only for vector dimension of at most 10 (including the variable `time`). These kumacs are overwritten each time the model is run. Usaually, gnuplot has to be preferred, but when using surfaces and histograms, PAW is better. The '`gains.f`' (and '`go.xqt`' is provided as an example in the Miniker files.

2.4 Controlling the run

It is possible to add code that will be executed at the end of each time step. It is also possible to specify which time step leads to a printout on standard output. For maximal control, the code running te model may be turned into a subroutine to be called from another fortran (or C) program, this possibility is covered in Section 3.2 [Calling the model code], page 15.

2.4.1 Executing code at the end of each time step

The code in the sequence '`zsteer`' is executed at the end of each time step. It is possible to change the time step length (variable `dt`) verify that the non linearity are not too big,

[1] '`dres.data`' has another time related variable as second column: `dt`, the time step that can vary in the course of a simulation.

or perform discontinuous modifications of the states. One available variable `res` might be usefull for time step monitoring. At the end of the time step, as soon as φ has been computed, a numerical test is applied on a pseudo relative quadratic residual between $\varphi = f(\eta(t-dt)+d\varphi$ (`ffl`), where $d\varphi$ is given by the system resolution in `ker`,and $\varphi = f(\eta), \varphi)$, Fortran variable (`ff`):

```
! =======================================================
! test linearite ffl - ff
! =======================================================
if (istep.gt.1)
< res=0.; <io=1,m; res = res +(ffl(io)-ff(io))**2/max(one,ff(io)*ff(io)); >;
  if (res .gt. TOL_FFL)
  < print*,'*** pb linearite : res > TOL_FFL a istep',istep,res,' > ',TOL_FFL;
    do io=1,m < z_pr: io,ff(io),ff(io)-ffl(io); >;
  >;
>;
```

This test hence applies only for non linearities in tranfer models. Nevertheless, `res` might be usefull to monitor the time step `dt` in `ZSTEER` and eventually go backward one step (`goto :ReDoStep:`). This can more appropriatly be coded in the (empty in default case) sequence `zstep`, inserted just before time-advancing states and `time` variables in '`principal`'.

It is also possible to fix the value of the criterium `TOL_FFL` in '`zinit`' different from its default value of 10^{-3} – independent of the Fortran precision.

Many other variables are available, including

`istep` The step number;

`couplage(.)`
The TEF coupling matrix between transfers;

`H` The Jacobian matrix corresponding with:

$$\partial_\eta g(\eta(t), \varphi(t));$$

`Bb` The Jacobian matrix corresponding with:

$$\partial_\varphi g(\eta(t), \varphi(t));$$

`Bt` The Jacobian matrix corresponding with:

$$\partial_\eta f(\eta(t), \varphi(t));$$

`D` The Jacobian matrix corresponding with:

$$\partial_\varphi f(\eta(t), \varphi(t));$$

`aspha` The state advance matrix;

`dneta`
`dphi` the variable increments;

One should be aware of that the linearity test concerns the preceding step. We have yet no example of managing the time-step.

2.4.2 Controlling the printout and data output

The printout on standard output is performed if the variable `zprint` of type `logical` is true. Therefore it is possible to control this printout by setting `zprint` false or true. For example the following code, in sequence '`zsteer`', triggers printing for every `modzprint` time step and the two following time steps:

```
ZPRINT = mod(istep+1,modzprint).eq.0;
Zprint = zprint .or. mod(istep+1,modzprint).eq.1;
Zprint = zprint .or. mod(istep+1,modzprint).eq.2;
```

The data output to '`.data`' files described in Section 2.3.3 [Running a simulation and using the output], page 11 is performed if the `logical` variable `zout` is true. For example the following code, in '`zsteer`', triggers output to '`.data`' files every `modzout` step.

```
Zout = mod(istep,modzout).eq.0;
```

3 Advanced Miniker programming

3.1 Overview of additional features setting

It is possible to enable some features by selecting which code should be part of the principal program. Each of these optionnal features are associated with a *select flag*. For example double precision is used instead of simple precision with the '`double`' select flag, the model is a subroutine with the select flag '`monitor`', the Kalman filter code is set with '`kalman`' and the 1D gridded model capabilities are associated with '`grid1d`'. To select a given feature the cmz statement `sel` *`select_flag`* should be written down in the '`selseq.kumac`' found in the model directory. With make either the corresponding variable should be set to 1 or it should be added to the `SEL` make variable, depending on the feature.

Other features don't need different or additional code to be used. Most of the features are enabled by setting specific logical variables to '`.true.`'. This is the case for `zback` for the adjoint model, `zcommand` if the command is in a file and `zlaw` if it is a function and `zkalman` for the Kalman filter. These select and logical flags are described in the corresponding sections.

In cmz an alternative of writing select flags to '`selseq.kumac`' is to drive the compilation with `smod` *`sel_flag`*. In that case the *sel_flag* is selected and the files and executable goes to a directory named '`sel_flag`'.

The select flags are taken into account during cmz directives preprocessing. Therefore you have the possibility to use these flags to conditionnaly include pieces of code. In most cases you don't need to include code conditionally yourself though, but if you want to, this is covered in Section 3.10 [Programming with cmz directives], page 27.

3.2 Calling the model code

When the model code is a subroutine, it can be called from another fortran program unit (or another program), and the model will be run each time the subroutine is called. This technique could be used, for example to perform optimization (see Section 4.2 [Adjoint model and optimisation with Miniker], page 31), or to run the model with different parameters.

3.2.1 Turning the model into a subroutine

With cmz, one has to do a

```
sel monitor
```

in the '`selseq.kumac`' file and create the KEEP that call the model code. See Section 3.1 [Selecting features], page 15.

With make '`monitor`' should be added to the `SEL` variable in the '`Makefile`', for example:

```
SEL = monitor
```

A file that call the principal subroutine should also be written, using the prefered language of the user. The additional object files should then be linked with the Miniker objects. To that aim they may be added to the `miniker_user_objects` variable.

3.2.2 Calling the model subroutine

The model subroutine is called '`principal`' and is called with the following arguments:

`principal` (*Cost, ncall, integer_flag, file_suffix, info, idxerror*) [Subroutine]

Where *Cost* is a real number, `real` or `double precision`, and is set by the `principal` subroutine. It holds the value of the cost function if such function has been defined (the use and setting of a cost function is covered later, see Section 4.2.3 [Cost function coding and adjoint modeling], page 33). *ncall* is an integer which corresponds with the number of call to `principal` done so far, it should be initialized to 0 and its value should not be changed, as it is changed in the `principal` subroutine. *integer_flag* is an integer that can be set by the user to be accessed in the `principal` subroutine. For example its value could be used to set some flags in the '`zinit`' sequence. *file_suffix* is a character string, that is suffixed to the output files names instead of '`.data`'. If the first character is the null character '`char(0)`', the default suffix, '`.data`' is appended. *info* and *idxerror* are integer used for error reporting. *idxerror* value is 0 if there was no error. It is negative for an alert, positive for a very serious error. The precise value determines where the error occured. *info* is an integer holding more precise information about the error. It is usually the information value from lapack. The precise meaning of these error codes is in table 3.1.

Source of error or warning	`info`	`idxerror`
state matrix inversion in ker	inversion	1
time advance system resolution in ker	system	2
transfer propagator, $(I - D)$ inversion	inversion	3
kalman analysis state matrix advance in phase space, $(I - D)$ inversion	inversion	21
kalman analysis variance covariance matrix non positive	Choleski	22
kalman analysis error matrix inversion	inversion	23
kalman error matrix advance	system	24
transfers determination linearity problem for transfers		-1
transerts determination Newton D_loop does not converge		-2

table 3.1: Meaning of error codes returned by principal.

In general more information than the provided arguments has to be passed to the `principal` subroutine, in that case a `common` block, to be written in the '`zinit`' sequence can be used.

3.3 Describing 1D gridded model

Specific macros have been built that allow generic description of 1D gridded models. Because of the necessity of defining left and right limiting conditions, the models are partitionned in three groups for cell and transfer components. In the following example, a chain of masselottes linked by springs and dumps is bounded to a wall on the left, and open at right. The TEF formulation of the problem is written in the phase space (position-shift, velocity) for node k, with bounding conditions:

$$\begin{cases} \partial_t \eta_k^{pos} = \eta_k^{vel} \\ \partial_t \eta_k^{vel} = \left(\varphi_k^{spr} - \varphi_{k+1}^{spr} + \varphi_k^{dmp} - \varphi_{k+1}^{dmp}\right)/m_k \end{cases}$$

$$\begin{cases} \varphi_k^{spr} = -k_k(\eta_k^{pos} - \eta_{k-1}^{pos}) \\ \varphi_k^{spr} = -d_k(\eta_k^{vel} - \eta_{k-1}^{vel}) \end{cases}$$

$$\begin{cases} \eta_0^{pos} = 0 \\ \eta_0^{vel} = 0 \\ \varphi_{N+1}^{spr} = 0 \\ \varphi_{N+1}^{dmp} = 0 \end{cases}$$

where m_k is the mass of node k, r_k and d_k the rigidity of springs and dumping coefficients. There are N nodes in the grid, from 1 to N, and two nodes outside of the grid, a limiting node 0, and a limiting node $N+1$. The limiting node corresponding with node 0 is called the *down* node, while the limiting node corresponding with node $N+1$ is called the *up* node. Other models not part of the 1D grid may be added if any.

To enable 1D gridded models, one should set the select flag '`grid1d`'. In cmz it is achieved setting the select flag in '`selseq.kumac`', like

```
sel grid1d
```

With make, the `SEL` variable should contain `grid1d`. For example to select `grid1d` and `monitor`, it could be

```
SEL = grid1d,monitor
```

3.3.1 Setting dimensions for 1D gridded model

In that case the number of nodes, the number of states and tranferts per node, and the number of limiting transfers and states are required. These dimensions has to be entered in the '`DimEtaPhi`' sequence. The parameters for cells are

`n_node` Number of cell nodes in the 1D grid.

`n_dwn` Number of limiting cells with index -1, *i.e.* number of cells in the limiting down node.

`n_up` Number of limiting cells with index +1, *i.e.* number of cells in the limiting up node.

`n_mult` Number of cells in each node (multiplicity).

The parameters for transfers, are similarly `m_node`, `m_dwn`, `m_up`, `m_mult`. The layout of their declaration should be respected as the precompiler matches the line. Also this procedure is tedious, it should be selected for debuging processes (use the flag `sel dimetaphi` in "selsequ.kumac". Otherwise, the dimensioning sequence will be automaticaly generated, which is smart but can lead to diffculty in interpreting syntax errors. Once a model is correctly entred, turn off the sel flag and further modifications will automatically generate the proper dimensions. The correctness of dimensionning should nevertheless always be checked in `principal.f`, where you can also check that null valued parameters as `lp, mobs, nxp` will suppress parts of the code - this is signaled as Fortran comment cards.

In our example, there are three grids of cell and transfer variables (`n_node=m_node=3`). There are two cells and two transfers in each node (`n_mult=2` and `m_mult=2`). There is no limiting condition for the states in the down node therefore `n_up=0`. There is no transfer for the first limiting node, and therefore `m_dwn=0`. There are two states in the limiting node 0, the down node, `n_dwn=2`, and two transfers in the limiting last node the node up, and `m_up=2`:

```
! ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! nodes parameters, and Limiting Conditions (Low and High)
! ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      parameter (n_node=3,n_dwn=2,n_up=0,n_mult=2);
      parameter (m_node=3,m_dwn=0,m_up=2,m_mult=2);
! ----------------------------------------------------
```

3.3.2 1D gridded Model coding

The model code and parameters go in the '`zinit`' sequence.

Parameters

A value for the Miniker parameters and the model parameters should be given in '`zinit`', in our example we have

```
!%%%%%%%%%%%%%%%%%%%%%%
! Parameters
!%%%%%%%%%%%%%%%%%%%%%%
real rk(n_node),rd(n_node),rmassm1(n_node);

data rk/n_node*1./;
data rd/n_node*0.1/;
data rmassm1/n_node*1./;
     dt=.01;
     nstep=5 000;
     modzprint = 1000;
     time=0.;
```

Limiting conditions

There are four mortran blocks for `node` and `up` and `down`, both for states and transfers:

`set_dwn_eta`
: down node cells

`set_up_eta`
: up node cells

`set_dwn_phi`
: down node transfers

`set_up_phi`
: up node transfers

The following scheme illustrates the example:

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%================================================
! Maillage convention inode
!%%%%%%%%%%%%%%%%%%%%%%%%%%                                 Open ended
!(2 Down    Phi    Eta                         (n_node)
! Eta)  \|       .-----.       .-----.          .-----.        /
! wall  \|-\/\/\-|     |-\/\/\-|     |  . . .  -|     |-\/\/\- |dummy
!  pos  \|--***--|  1  |--***--|  2  |  . . .  -|  n  |--***-- |Phis
! speed \|   1   |_____|   2   |_____|      n   |_____|  n+1   \(2 Up Phi)
!
```

Two states are associated with the down node, they correspond to the position and speed of the wall. As the wall don't move these states are initialized to be 0, and the cells are stationnary cells, therefore these values remain 0.

```
! Down cells (wall)
! -----------------
eta_pos_wall = 0; eta_speed_wall = 0.;

set_dwn_eta
< var: eta_pos_wall,  fun: deta_pos_wall  = 0.;
  var: eta_speed_wall, fun: deta_speed_wall= 0.;
>;
```

There are 2 limiting transfers in the up node. They correspond with an open end and are therefore set to 0.

```
! limiting Transfers : dummy ones
! -------------------------------
set_Up_Phi
< var:ff_dummy_1, fun: f_dummy_1=0.;
  var:ff_dummy_2, fun: f_dummy_2=0.;
>;
```

Starting points

The cell node state values are initialized. They are in an array indexed by the `inode` variable. In the example the variable corresponding with position is `eta_move` and the variable corresponding with speed is `eta_speed`. Their initial values are set with the following mortran code

```
!---------------
! Initialisation
!---------------
;
do inode=1,n_node <eta_move(inode)=0.01; eta_speed(inode)=0.0;>;
```

If any transfer needs to be given a first-guess value, this is also done using `inode` as the node index.

Grid node equations

Each node is associated with an index `inode`. It allows to refer to the preceding node, with `inode-1` and the following node `inode+1`. The node states are declared in `set_node_Eta` block and the transfers are in `set_node_Phi` blocks.

In the example, the cells are declared with

```
! node cells
! ----------
;
set_node_Eta
< var: eta_move(inode),  fun: deta_move(inode) = eta_speed(inode);
  var: eta_speed(inode),
  fun: deta_speed(inode) = rmassm1(inode)
                             *( - ff_spring(inode+1) + ff_spring(inode)
                               - ff_dump(inode+1)  + ff_dump(inode)
                              );
>;
```

Note that the `inode` is dummy in the `var:` definition and can as well be written as: `var: eta_move(.)`.

The transfers are (`ff_spring` corresponds with springs and `ff_dump` with dumps):

```
!%%%%%%%%%%%%%%%%%%%%%
! Transfer definition
!%%%%%%%%%%%%%%%%%%%%%
! node transfers
! --------------
! convention de signe spring : comprime:= +
set_node_Phi
< var: ff_spring(.),
  fun:
   f_spring(inode)= -rk(inode)*(eta_move(inode) - eta_move(inode-1));
  var: ff_dump(.),
  fun:
   f_dump(inode)  = -rd(inode)*(eta_speed(inode) - eta_speed(inode-1));
>;
```

The limiting states and transfers are associated with the states or transfers with index `inode+1` or `inode-1` appearing in node cell and transfer equations (`inode-1` for down limiting conditions and `inode+1` for up limiting conditions) in their order of appearance. In our example, in the `eta_speed` state node equation `ff_spring(inode+1)` appears before `ff_dump(inode+1)` and is therefore associated with `ff_dummy_1` while `ff_dump(inode+1)` is associated with the `ff_dummy_2` limiting transfer, as `ff_dummy_1` appears before `ff_dummy_2` in the limiting up transfers definitions. Verification of the grid index coherence should be eased with the following help printed in the listing header:

```
 --------------- Informing on Dwn Eta definition ---------------
Var-name,            Function-name, index in eta vector
        eta_pos_wall        deta_pos_wall  1 [
      eta_speed_wall      deta_speed_wall  2 [

 -------------- Informing on Eta Nodes definition --------------
Var-name,     Function, k2index of (inode: 0 [ 1,...n_node ] n_node+1)
           eta_move           deta_move    1 [   3 ...   7 ]   9
          eta_speed          deta_speed    2 [   4 ...   8 ]  10

 ---------------- Informing on Up  Phi  definition -------------
Var-name,             Function-name, index in ff vector
         ff_dummy_1           f_dummy_1 ]    7
         ff_dummy_2           f_dummy_2 ]    8
        ff_move_sum          f_move_sum ]    9
       ff_speed_sum         f_speed_sum ]   10
 ------------------------------------------------------

-------------- Informing on Phi Nodes definition ---------------
Var-name,     Function, k2index of (inode: 0 [ 1,...m_node ] m_node+1)
          ff_spring            f_spring   -1 [   1 ...   5 ]   7
            ff_dump              f_dump    0 [   2 ...   6 ]   8
 ------------------------------------------------------
```

All variable names and functions are free but has to be different. Any particular node-attached variable k is referred to as: '`(inode:k)`', where k has to be a Fortran expression allowed in arguments. The symbol '`inode`' is reserved. As usual other Fortran instructions can be written within the Mortran block '`< >`' of each `set_` block.

3.4 Double precision

The default for real variables is the `real` Fortran type. It is possible to use double precision instead. In that case all the occurences of '`real` ' in mortran code is substituted with '`double precision` ' at precompilation stage, and the Lapack subroutine names are replaced by the double precision names. Eventual users'declaration of `complex` Fortran variables is also changed to `double complex` .

This feature is turned on by `sel double` in '`selseq.kumac`' with cmz and `double = 1` in the '`Makefile`' with make.

In order for the model to run as well in double as in simple precision, some care should be taken to use the generic intrinsic functions, like `sin` and not `dsin`. No numerical constant should be passed directly to subroutines or functions, but instead a variable with the right type should be used to hold the constant value, taking advantage of the implicit casts to the variable type.

3.5 Partial Derivatives

The partial derivative rules are included in a `Mortran` macro series in '`Derive_mac`' of Miniker files. When using an anusual function, one should verify that the corersponding rules are in that file. It is easy to understand and add new rules in analogy with the already existing ones.

For instance, suppose one wants to use the intrinsic Fortran function `abs()`. Its derivatives uses the other function `sign()` this way:

```
&'(ABS(#))(/#)' = '((#1)(/#2)*SIGN(1.,#1))'
```

In such cases when one is adding a new rule, it is important to use the generic function names only (i.e. `sin` not `dsin`), because when compilating Miniker in the double precision version, or complex version, the generic names will correctly handle the different variable types - which is not the case when coding with specific function names.

3.5.1 Derivating a power function

Partial derivative of a function in exponent is not secure in its Fortran form `g(x,y)**(f(y))`. It should be replaced by `power(g,f)` of the Miniker '`mathlib`', or by the explicit form `exp(f(y)*log(g(x,y)))`.

Its derivative will have the following form:

$$\begin{aligned}\partial_x f^g &= g f^{g-1}\partial_x f + f^g \log f \partial_x g \\ &= f^{g-1}(g\partial_x f + f\partial_x g)\end{aligned}$$

and is in the macros list already defined in: '`DERIVE_MAC`'.

3.6 Rule of programming non continuous models

Some models may originally be non continuous, as the ones using a Fortran instruction `IF`. Some may use implicitly a step function on a variable. In such cases, the model has to be set in a derivable form, and use a "smooth step" instead. One should be aware of that this apparently mathematical treatment currently indeed leads to a physical question about the macroscopic form of a physical law. At a macroscipic level, a step function is usually a nonsense. Taking the example of phase-change, a fluid volume does not change phase at once, and a "smooth change of state" is a correct macroscopic model.

Miniker provides with the smooth step function *Heavyside*[1] in the Miniker '`mathlib`':

```
Delta = -1."K";
A_Ice =  heavyside("in:" (T_K-Tf), Delta, "out:" dAIce_dT);
```

in this example, `Tf` is the ice fusion-temperature, `A_ice` gives the ice-fraction of the mesh-volume of water at temperature `T_k`. The smooth-step function is a quasi hyperbolic tangent function of x/Δ, normalised from 0 to 1, with a maximum slope of 2.5, see figure Figure 3.1.

[1] This naming is a joke for "Inert" Heaviside function.

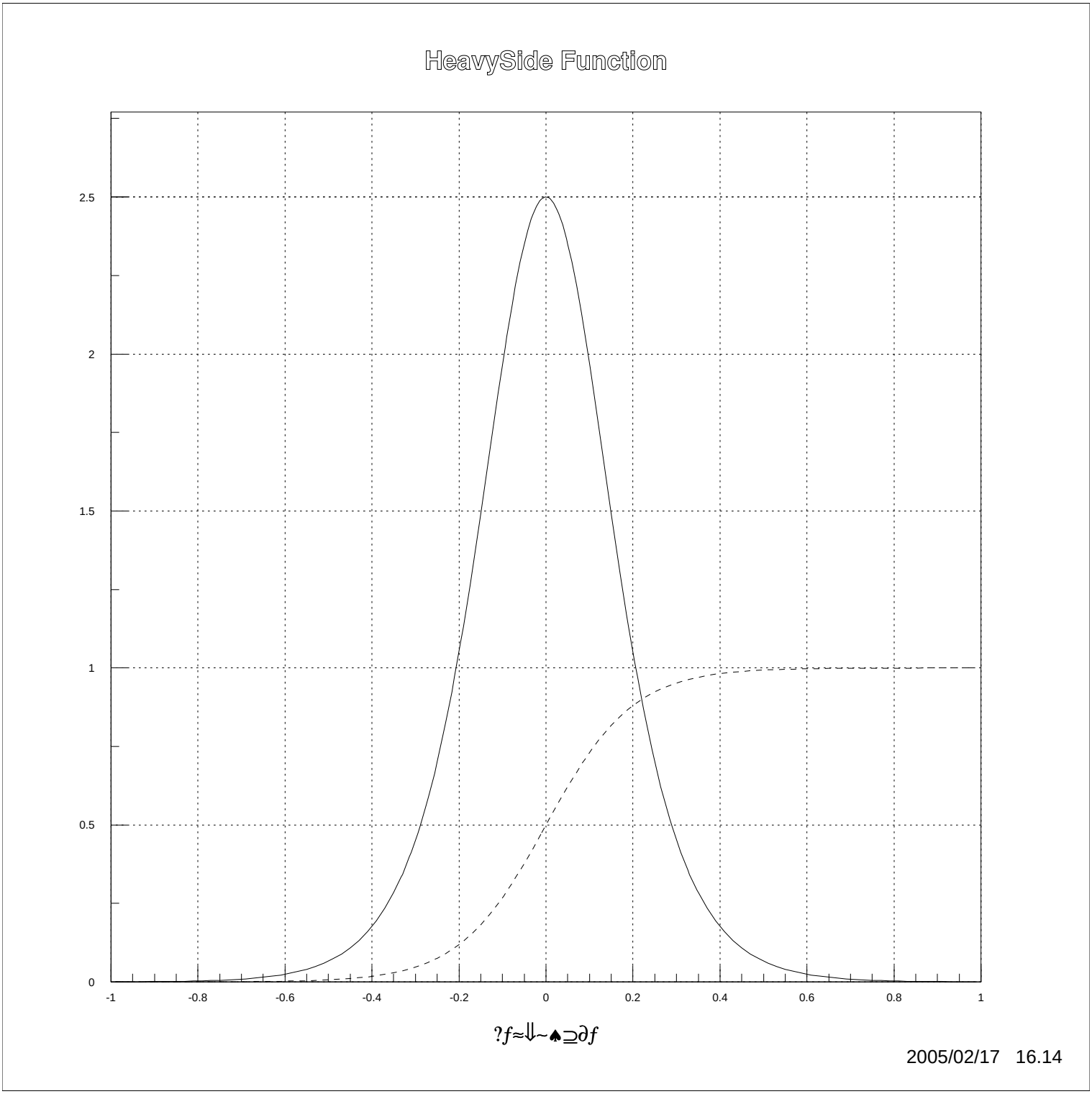

Figure 3.1: Heaviside function and derivative

For `Mortran` to be able to symbolicaly compute the partial derivarives, the rule is in the table of macros as:

```
&'(HEAVYSIDE(#,#,#))(/#)' = '((#1)(/#4)*HEAVYDELTA(#1,#2,#3))'
```

which uses the Foratn entry point `HeavyDelta` in the Fortrsan function `heavyside`.

Another type of problem arises when coding a `var=min(f(x),g(x))` Fortran instruction. In such a case one does not want a derivative and one will code:

```
var = HeavySide(f(x)-g(x),Delta,dum)*g(x) + (1.-HeavySide(f(x)-g(x),Delta,dum)*f(x);
```

or equivalently:

```
var = HeavySide(f(x)-g(x),Delta,dum)*g(x) + HeavySide(g(x)-f(x),-Delta,dum)*f(x);
```

Warning: the value of the argument *Delta* is important because it will fix the maximum slope of the function that will appear as a coefficient in the Jacbian matrices.

3.7 Parameters

It is possible to specify some Fortran variables as specific model parameters. Model parameters may be used in sensitivity studies (see Section 4.1.1 [Sensitivity to a parameter], page 30) and in the adjoint model (see Section 4.2.4 [Sensitivity of cost function to parameters], page 33). Nothing special is done with parameters with Kalman filtering.

The parameters are fortran variables that should be initialized somewhere in '`zinit`'. For a variable to be considered as a parameter, it should be passed as an argument to the `Free_parameters` macro. For example if `apar` and `cpar` (from the predator example) are to be considered as parameters, `Free_parameters` should be called with:

```
Free_parameter: apar, cpar;
```

When used with grid1d models (see Section 3.3 [Describing 1D gridded model], page 17) the `inode` number may appear in parenthesis:

```
Free_parameter: rd(1), rk(2);
```

3.8 Observations and data

Some support for observations and interactions with data is available. The observations are functions of the model variables. They don't have any action on the model result, but they may (in theory) be observed and measured. The natural use of these observations is to be compared with data that correspond with the values from real measurements. They are used in the Kalman filter (see Section 4.3 [Kalman filter], page 34).

The (model) observation vector is noted ω and the observation function is noted h:

$$\omega = h(\eta, \varphi)$$

3.8.1 Observations

The observation functions are set in a `set_probe` block in the '`zinit`' sequence.

For example suppose that, in the predator-prey model, we only have access to the total population of preys and predators, we would have:

```
set_probe
< eqn: pop = eta_pred + eta_pray;
```

```
>;
```

The number of observations is put in the integer variable `mobs`. The observation vector corresponds with the part of the `ff(.)` array situated past the regular transferts, `ff(mp+.)`, and is output in the file '`obs.data`'.

3.8.2 Data

Currently this code is only used if the Kalman code is activated. This may be changed in the future.

The convention for data is that whenever some data are available, the logical variable `zgetobs` should be set to '`.true.`'. And the `vobs(.)` vector should be filled with the data values. This vector has the same dimension than the observation vector and each coordinate is meant to correspond with one coordinate of the observation vector.

This feature is turned on by setting the logical variable `zdata` to '`.true.`', and the `zgetobs` flag is typically set in the '`zsteer`' sequence (see Section 2.4.1 [Executing code at the end of each time step], page 12). Every instant data are available (`zgetobs` is true) the observations are written to the file '`data.data`'. With the Kalman filter more informations are output to the '`data.data`' file, see Section 4.3.2.2 [Kalman filter results], page 36.

3.9 Entering model size explicitely

It is possible to enter the model dimensions explicitely, instead of generating them automatically, as it was done previously. This feature is turned on by `sel dimetaphi` in '`selseq.kumac`' with cmz and `dimetaphi` added to the `SEL` variable in the '`Makefile`' with make.

3.9.1 The explicit size sequence

The dimension of the model is entered in the sequence '`dimetaphi`', using the fortran `parameter np` for `eta(.)` and `mp` for `ff(.)`. For the Lotka-Volterra model, we have two cell components and only one transfer.

```
parameter (np=2,mp=1);
```

You should not change the layout of the parameter statement as the mortran preprocessor matches the line.

You also have to provide other parameters even if you don't have any use for them. If you don't it will trigger fortran errors. It includes the `maxstep` parameter that can have any value but 0, `lp` and `mobs` that should be 0 in the example, and `nxp`, `nyp` and `nzp` that should also be 0. The layout is the following:

```
parameter (np=2,mp=1);
parameter (mobs=0);

parameter (nxp=0,nyp=0,nzp=0);
parameter (lp=0);
parameter (maxstep=1);
```

If there are observations, (see Section 3.8.1 [Observations], page 24), the size of the observation vector is set in the '`dimetaphi`' sequence by the `mobs` parameter. For example if there is one observation:

```
parameter (mobs=1);
```

To specify parameters (see Section 3.7 [Parameters], page 24), the number of such parameters has to be declared in '`dimetaphi`' with the parameter `lp`. Then, if there are two parameters, they are first declared with

```
parameter (lp=2);
```

3.9.2 Entering the model equations, with explicit sizes

When sizes are explicit, another possibility exists for entering the model equations. The use of symbolic names, as described in [Model equations], page 6 is still possible, and it also becomes possible to set directly the equations associated with the `eta(.)` and `ff(.)` vectors.

In case the symbolic names are not used, the model equations for cells and transfers are entered using a mortran macro, `f_set`[2], setting the `eta(.)` evolution with `deta_tef(.)` and the transfer definitions `ff(.)` with `Phi_tef(.)`.

`f_set` *Phi_tef*(`i`) = *f*(*eta*(.),*ff*(.)) [Macro]

This macro defines the transfer *i* static equation. `f` is a fortran expression which may be function of cell state variables, '`eta(1)`'...'`eta(np)`' and transfers '`ff(1)`'...'`ff(mp)`'.

In the case of the predator-prey model, the transfer definition for φ_{meet} is:

```
f_set Phi_tef(1) = eta(1)*eta(2);
```

`f_set` *deta_tef*(`i`) = *g*(*eta*(`i`),*ff*(.)) [Macro]

This macro defines the cell state component *i* time evolution model. `g` is a expression which may be function of cell state variables, '`eta(1)`'...'`eta(np)`' and transfers '`ff(1)`'...'`ff(mp)`'.

The two cell equations of the predator-prey model are, with index 1 for the prey (η_{prey}) and index 2 for the predator (η_{pred}):

```
f_set  deta_tef(1) = apar*eta(1)-apar*ff(1);
f_set  deta_tef(2) = - cpar*eta(2) + cpar*ff(1);
```

The whole model is:

```
!%%%%%%%%%%%%%%%%%%%%%%
! Transfer definition
!%%%%%%%%%%%%%%%%%%%%%%
! rencontres (meeting)
    f_set Phi_tef(1) = eta(1)*eta(2);

!%%%%%%%%%%%%%%%%%%%%%%
! Cell definition
!%%%%%%%%%%%%%%%%%%%%%%
! eta(1) : prey
```

[2] `fun_set`, or equivalently `f_set`, is a general mortran macro associating a symbol with a fortran expression. Here, it is the name of the symbol (`eta`) that has a particular meaning for the building of the model.

```
! eta(2) : predator

    f_set  deta_tef(1) = apar*eta(1)-apar*ff(1);
    f_set  deta_tef(2) = - cpar*eta(2) + cpar*ff(1);
```

The starting points for cells are entered like:

```
!     initial state
!     -------------
     eta(1) = 1.;
     eta(2) = 1.;
```

If there are observations, they are entered as special transferts with index above `mp`, for example:

```
f_set Phi_tef(mp+1) = ff(1) ;
```

3.10 Programming with cmz directives

3.10.1 Cmz directives used with Miniker

The main feature of cmz directive is to use code conditionnaly for a given select flag. For example when the double precision is selected (see Section 3.4 [Double precision], page 21) the use of the conditionnal `double` flag may be required in case there is a different subroutine name for different types. If, for example, the user use the subroutine `smysub` for simple precision and `dmysub` for double precision the following code is an example of what could appear in the user code:

```
+IF,double
 call dmysub(eta);
+ELSE
 call smysub(eta);
+ENDIF
```

For a complete reference on cmz directives see the appendix Appendix B [Cmz directives reference], page 51.

3.10.2 Using cmz directives in Miniker

In cmz the KEEP and DECK have their cmz directives preprocessed as part of the source files extraction. And the +KEEP and +DECK directives are automatically set when creating the KEEP or DECK. With make, files with these directives has to be created within the files that are to be preprocessed by the cmz directives preprocessor.

To be processed by make, a file that contains cmz directives should have a file suffix corresponding with the language of the resulting file and with the normal file suffix of that language. More precisely '`cm`' should be added before the normal file suffix and after the '`.`'. Therefore if the resulting file language is associated with a suffix '*`.suf`*', the file with cmz directives should have a '`.cm`*`suf`*' suffix. The tradition is to have a different suffix for main files and include files. To add directories searched for *cmfiles* (files with cmz directives) they should be added to the `CMFDIRS` makefile variable, separated by '`:`'.

Rules for preprocessing of the files are defined in the file '`Makefile.miniker`' for the file types described in table 3.2:

language	**file type**	**cmfile suffix**	**suffix**	**language**
fortran	main/deck	.cmf	.f	ftn
fortran preprocessed	main/deck	.cmF	.F	f77
fortran preprocessed	include/keep	.cminc	.inc	f77
mortran	main/deck	.cmmtn	.mtn	mtn
mortran	include/keep	.cmmti	.mti	mtn

table 3.2: Association between file language, file type, file suffixes and language identifier in cmz directives. A main file is called a *deck* in cmz and an include file is called a *keep*.

4 Dynamic analysis of systems in Miniker

4.1 Automatic sensitivity computation

An obvious advantage of having acces to the Jacobian matrices along the system trajectory concerns automatic sensitivity analyses, as either:

- the sensitivity of all variables to perturbation in the initial condition of one state variable;
- the same sensitivities to an initial pulse (or step) on a transfer;
- the same sensitivities to a series of pulses (or steps) on a transfer;
- the same for a change in a parameter, eventually during the run;
- the sensitivity of the matrix of advance in state space to a change in a parameter.

This is declared in Zinit as:

```
! -------------
! Sensitivities
! -------------
Sensy_to_var
< var: eta_pray, pert: INIT;
  var: eta_pred, pert: INIT;
>;
```

Each variable at origin of a perturbation is declared as `var:`, and the type of perturbation in `pert:`. Here, INIT conditions are only allowed because the two variables are states variables. For transfers, `pert: pulse` corresponds to an initial pulse, `pert: step_resp` and `pert: step_eff` to initial steps, the difference between `_resp` (response form) and `_eff` (effect form) concerns the diagonal only of the sensitivity matrix (see Feedback gains in non-linear models).

Non initial perturbation can also be asked for:

```
    Sensy_to_var
    <
!*     var: eta_courant_L, pert: init at 100;
!*     var: ff_T_czcx,     pert: pulse at 100 every 20;
!*     var: ff_Psi_Tczcx,  pert: step_eff;
!*     var: ff_Psi_Tczcx,  pert: step_Resp at 10 every 100;
! *** premiers tests identiques a lorhcl.ref
      var: ff_courant_L , pert: step_eff;
      var: ff_T_czcx    , pert: step_eff;
      var: ff_Psi_Tczcx , pert: step_eff;
      var: ff_Psi_Tsz   , pert: pulse at 100 every 50;
    >;
```

In this example taken from '`lorhcl`', a sensitivity can increase so as to trespass the Fortran capacity, so that each sensitivity vector (matrix column) can be reset at some time-increment `at III every JJJ;`

It is noteworthy that these sensitivity analyses are not based on difference between two runs with different initial states or parameter values, but on the formal derivatives of the

model. This method is not only numerically robust, but is also rigorously funded as based on the TLS of the model[1].

If the `dimetaphi` sequence is built by the users, he should declare the number of perturbing variables as `nxp=`:

```
parameter (nxp=np,nyp=0,nzp=0);
```

here, all state variables are considered as perturbing variables.

The sensitivity vectors are output in the result files '`sens.data`' for cells and '`sigma.data`' for transfers. In those files the first column corresponds again with time, and the other columns are relative sensitivities of the cell states (in '`sens.data`') and transfers (in '`sigma.data`') with respect to the initial value of the perturbed state.

In our predator-prey example, the second column of '`sens.data`' will contain the derivative of $\eta_1(t)$ with respect to $\eta_1(t=0)$. Drawing the second column of '`sens.data`' against the first one gives the time evolution of the sensitivity of `eta-pred` to a change in the initial value of `eta-pray`. One can check in that it is set to 1 at $t=0$:

```
#    Sensy_to: eta_pray         3         eta_pred         5
# time \\  of: eta_pray     eta_pred     eta_pray     eta_pred
  0.00000E+00  1.00000E+00  0.00000E+00  0.00000E+00  1.00000E+00
  1.00000E-02  9.90868E-01  1.11905E-02 -1.26414E-02  9.98859E-01
```

The two last columns are the state sensitivity to a change in initial conditions of the number of predators.

In the same way, the *j+1*th column of '`sigma.data`' will be the derivative of $\phi_j(t)$ with respect to $\eta_i(t=0)$. Here:

```
#    Sensy_to: eta_pray     eta_pred
# time \\  of: ff_interact  ff_interact
  0.00000E+00  1.60683E+00  8.47076E-01
  1.00000E-02  1.59980E+00  8.18164E-01
```

the unique transfer variable gives rise to two sensitivity columns.

Sensitivity studies are usefull to assess the predictability properties of the corresponding system.

4.1.1 Sensitivity to a parameter

A forward sensitivity to a parameter will be computed when specified as described in Section 3.7 [Parameters], page 24. For example, suppose that the sensitivity to an initial change in the `apar` parameter of the predator model is of interest.

The sensitivity calculs is turned on as a forward parameter specified on the `Free_parameter` list:

```
Free_parameter: [fwd: apar, cpar];
```

The result are in '`sensp.data`' for cells and '`sigmap.data`' for transfers.

```
#    Sensy_to: pi_prandtl       3            4         pi_rayleigh_     6
# time \\  of: eta_courant_ eta_T_czcx   eta_T_sz     eta_courant_ eta_T
```

[1] For a short introduction to automatic sensitivity analysis, see the document: http://lmd.jussieu.fr/zoom/doc/sensibilite.ps, in French, or ask for the more complete research document to a member of the TEF-ZOOM collaboration

```
0.00000E+00  0.00000E+00  0.00000E+00  0.00000E+00  0.00000E+00  0.000
2.00000E-03 -4.77172E-03 -3.99170E-05  3.55971E-05 -9.94770E-05 -1.004
```

In the above example from '`lorhcl`' sensitivity of the three states with respect to an initial change in two parameters are independantly given (first line also numbers the column to easy gnuplot using).

4.1.2 Advance matrix sensitivity

It is possible to look at the sensitivity of the matrix of advance in states space (the matrix `aspha`) with regard to a parameter. The parameter must be accounted for in the parameter number and be in the parameter list, flagged as the matrix `mx` parameter, like in

```
Free_parameter: [mx: apar], cpar;
```

This feature is associated with a selecting flag, '`dPi_aspha`'. One gets the result in the matrix `d_pi_aspha(.,.)` of dimension (`np`,`np`).

This matrix may be used to compute other quantities, for example it may be used to compute the sensitivity of the eigenvalues of the state-advance matrix with regard to the `[fwd]` parameter. These additional computations have to be programmed by the user in '`zsteer`' with matrices declared and initialized in '`zinit`'. An example is given in the example '`lorhcl`' provided with the Miniker installation files, following a method proposed by Stephane Blanco.

4.2 Adjoint model and optimisation with Miniker

In the following a possible use of Miniker for optimisation is discussed. More precisely the use of adjoint and control laws in Miniker are presented. Optimisation isn't the only application of these tools, but it is the most common one. In that case the adjoint may be used to determine the gradient of a functional to perturbations in the control laws, and an optimisation process can use this information to search for the optimum. Another application of the adjoint is to compute the sensitivity of a cost function to parameters (the ones declared in the `free_parameters:`' list. Note that the cost function can be sensitive to probe's variables, even if these are uncoupled with standard variables in the forward calculations; this is the case when minimizing a quadratic distance function between probes (from the model) and the corresponding measurements.

The code is close transcription of the mathematical calculus described in http://www.lmd.jussieu.fr/ZOOM/doc/Adjoint.pdf . It essentialy reverse time and transpose the four Jacobian matrices: states and transfers are saved in array dimensionned with `maxstep` Fortran parameter.

4.2.1 Overview of optimisation with Miniker

In the proposed method, Miniker is run twice, one time forward and then backward to determine the trajectory and the adjoint model. After that the control laws are modified by a program external to Miniker. The same steps are repeated until convergence. More pecisely,

forward The command law $h(t)$ is given (by an explicit law or taken from a file). The trajectory is computed in a classical way, with the additionnal computation of the functional to be optimised, J, prescribed with specific `f_set` macros. The states, transfers and control laws are stored.

backward The adjoint variable is computed from the last time T backward. The time increment is re-read as it could have changed during the forward simulation. The system is solved by using the same technics as in the forward simulation, but with a negative time step.

external phase
Now the command should be corrected. This step isn't covered here, but, for example, minuit the optimisation tool from the CERN could be used. In order to ease such a use of Miniker, the principal program has to be compiled as a subroutine to be driven by an external program (see Section 3.2 [Calling the model code], page 15).

The functionnal J to be optimised is defined as

$$J = \psi[\eta(T), \varphi(T), h(T)] + \int_0^T l[\eta(\tau), \varphi(\tau), h(\tau)]\, d\tau$$

Where ψ is the final cost function, l is the integrand cost function and h represents the control laws variations.

The general use of the adjoint model of a system is the determination of the gradient of this J functional to be optimised, with respect to perturbations of the original conditions of the reference trajectory, that is, along its GTLS[2].

4.2.2 Control laws

Each control law is associated with one cell or transfer equation, meaning that a command associated with an equation does not appear in any other equation. It is still possible to add commands acting anywhere by defining a transfer equal to that command.

The control laws associated with states are in the `ux_com(.)` array, control laws associated with transfers are in the `uy_com(.)` array. The control laws may be prescribed even when there is no adjoint computed, nor any optimisation, and they are used during simulation, in which case they will act as external sources. To enable the use of commands, the logical flag `Zcommand` should be `.true.`.

The command can be given either as:

1. a table of numerical values in the files '`uxcom.data`' and '`uycom.data`'.
2. a function of the problem variables. To turn that feature on the logical flag `Zlaw` should be set to `.true.` in '`zinit`'. The sequence '`zcmd_law`' should hold the code filling the `ux_com(.)` and `uy_com(.)` arrays, as the code from that sequence is used whenever the control laws are needed. In that case the files '`uxcom.data`' and '`uycom.data`' will be filled by the command values generated by the function along the trajectory.

For example in the Lotka-Volterra model, the parameter `apar` could be a control variable. In that case, `apar` would be defined as the variable `ux_com(1)`, and either entered as a law in the sequence '`zcmd_law`' , either written in the file '`uxcom.data`' step by step. In that case, there must be a perfect corresponodence between time of the commands and time of the run.

[2] General Tangent Linear System, i.e. the TLS circulating along a trajectory. See the explanation in the document http://www.lmd.jussieu.fr/Zoom/doc/Adjoint.pdf (in French).

4.2.3 Cost function coding and adjoint modeling

First of all the flag `zback` should be set to `.true.` in order to allow adjoint model computation:

```
Zback=.true.;
```

The two functions `cout_Psi` corresponding with the final cost and `cout_l` corresponding with the integrand cost are set up with the `f_set` macros.

`f_set` *cout_Psi* = *f*(*eta*(.),*ff*(.),*ux_com*(.),*uy_com*(.)) [Macro]

This macro defines the final cost function. `f` is a fortran expression which may be function of cell state variables, '`eta(1)`'...'`eta(np)`', transfers '`ff(1)`'...'`ff(mp)`', state control laws '`ux_com(1)`'...'`ux_com(np)`', and transfer control laws '`uy_com(1)`'...'`uy_com(mp)`'.

`f_set` *cout_l* = *f*(*eta*(.),*ff*(.),*ux_com*(.),*uy_com*(.)) [Macro]

This macro defines the integrand cost function. `f` is a fortran expression which may be function of cell state variables, '`eta(1)`'...'`eta(np)`', transfers '`ff(1)`'...'`ff(mp)`', state control laws '`ux_com(1)`'...'`ux_com(np)`', and transfer control laws '`uy_com(1)`'...'`uy_com(mp)`'.

For example, the following code sets a cost function for the masselottes model:

```
! Initialisation
  F_set cout_Psi = eta_move(inode:1);
!and f_set cout_l integrand in the functionnal
  F_set cout_l = 0.;
```

In that example the functional is reduced to the final value of the first state component. Here, the adjoint vector will correspond to the final sensitivity (at $t = 0$) of that component (here the first masselotte position) to a perturbation in all initial conditions[3].

The following variables are set during the backward phase, and output in the associated files:

var	**file**	**explanation**
`v_adj(.)`	'`vadj.data`'	adjoint to `eta(.)`
`w_adj(.)`	'`wadj.data`'	adjoint to `ff(.)`
`wadj(mp+.)`	'`gradmuj.data`'	adjoint to `ff(mp+.)`
`graduej(.)`	'`gradxj.data`'	adjoint to `ux_com(.)`
`gradufj(.)`	'`gradyj.data`'	adjoint to `uy_com(.)`
`hamilton`	'`hamilton.data`'	time increment, hamiltonian, cost function increment

4.2.4 Sensitivity of cost function to parameters

The sensitivity of the cost function to all the parameters given as arguments of `Free_parameters` is computed. For the predator model the sensitivity of a cost function consisting in the integral of the predator population with respect with `apar` an `cpar` is obtained with a number of parameters set to 2 in '`dimetaphi`':

[3] For detailed explanation of the adjoint model, see the document in pdf or .ps.gz

```
parameter (lp=2);
```

And the cost function and `Free_parameters` list in '`zinit`':

```
f_set cout_Psi = eta(2);
f_set cout_l = eta(2);
Free_parameters: apar,cpar;
```

`apar` and `cpar` also have to be given a value. The result is output in '`gradpj.data`'.

4.3 Kalman filter

The Kalman filter allows for data assimilation along the model run. In that case it is assumed that there is a real-world model with stochastic perturbations on the states, and that noisy observations are available. The situation implemented in Miniker corresponds to a continuous stochastic perturbation on the state, and discrete noisy observations. In the TEF this leads to:

$$\begin{aligned}\partial_t \eta(t) &= g(\eta(t), \varphi(t)) + W(t)\mu \\ \varphi(t) &= f(\eta(t), \varphi(t)) \\ \omega(t) &= h(\eta(t), \varphi(t)) + \nu\end{aligned}$$

The observations ω are available at discrete time steps $t = s_i$. The stochastic perturbation on state, μ is characterized by a variance-covariance matrix Q and the noise on the observation, ν has a variance-covariance matrix R. W relates states with stochastic perturbations. At each time step the Kalman filter recomputes an estimation of the state and the variance-covariance matrix of the state.

In the following we use the example of a linear model with perturbation on state and observation of state. The model has 3 states and 3 corresponding transfers (equal to the states), but the error on the state is of dimension 2. The 3 states are observed. The corresponding equations read:

$$\begin{cases} \partial_t \eta_1 = a_{11}\eta_1 + a_{12}\varphi_2 + a_{13}\varphi_3 + W_{11}\mu_1 + W_{12}\mu_2 \\ \partial_t \eta_2 = a_{21}\varphi_1 + a_{22}\eta_2 + a_{23}\varphi_3 + W_{21}\mu_1 + W_{22}\mu_2 \\ \partial_t \eta_3 = a_{31}\varphi_1 + a_{32}\varphi_2 + a_{33}\eta_3 + W_{31}\mu_1 + W_{32}\mu_2 \end{cases}$$

$$\begin{cases} \varphi_1 = \eta_1 \\ \varphi_2 = \eta_2 \\ \varphi_3 = \eta_3 \end{cases}$$

$$\begin{cases} \omega_1 = \varphi_1 + \nu_1 \\ \omega_2 = \eta_2 + \nu_2 \\ \omega_3 = \eta_3 + \nu_3 \end{cases}$$

4.3.1 Coding the Kalman filter

First of all the Kalman filter code should be activated. The observations code is also required (see Section 3.8.1 [Observations], page 24). If cmz is used the code should be selected with the select flag kalman in the '`selseq.kumac`':

```
sel kalman
```

With make the `kalman` variable should be set to 1:

```
kalman = 1
```

The kalman code is actually used by setting the flag `zkalman` to `.true.`, for example in the '`zinit`':

```
zkalman = .True.;
```

With the Kalman filter the dimension of estimated states, of the error on the state and of the observation, the W matrix, the observation function, the initial variance-covariance matrices on the state and the variance-covariance matrices of errors have to be given.

4.3.1.1 Kalman filter vectors dimensions

These dimensions should be set in the '`zinit`' sequence. The size of the estimated states is given by the parameter `nkp`. You can set this to `np` if all the states are estimated, but in case there are some deterministic state variables, `nkp` may be less than `np`. In that case the first `nkp` elements of `eta(.)` will be estimated using the Kalman filter.

The error on state dimension is associated with the parameter `nerrp` and the size of the observations vector is `mobs` (see Section 3.8.1 [Observations], page 24). In our example the dimensions are set with:

```
parameter (nkp=np);
parameter (mobs=3);
parameter (nerrp=2);
```

All the states are estimated, there are 3 observation functions and the error on the state vector is of dimension 2.

If the sizes are set explicitely, the parameters should be set in '`dimetaphi`'.

4.3.1.2 Error and observation matrices

Initial variance-covariance matrix on the state

The variance-covariance on the state matrix is `covfor(.,.)`. The initial values have to be given for this matrix, as in our example:

```
covfor(1,1) = 1000.; covfor(1,2) = 10.; covfor(1,3) = 10.;
covfor(2,1) = 10.; covfor(2,2) = 5000.; covfor(2,3) = 5.;
covfor(3,1) = 10.; covfor(3,2) = 5.; covfor(3,3) = 2000.;
```

This matrix is updated by the filter at each time step because the states are pertubated by some noise, and when assimilation takes place as new information reduce the error.

Observations and error on state matrix

The matrix that relates errors on states vector components to states, corresponding with W is `mereta(.,.)`. In our example it is set by:

```
mereta(1,1) = 1.;  mereta(1,2) = 0.;
mereta(2,1) = 0.;  mereta(2,2) = 1.;
mereta(3,1) = 0.5;  mereta(3,2) = 0.5;
```

The observation functions are set by a `f_set` macro with `Obs_tef(.)` as described in Section 3.8.1 [Observations], page 24. In our example the observation functions are set by:

```
f_set Obs_tef(1) = ff(1) ;
f_set Obs_tef(2) = eta(2);
f_set Obs_tef(3) = eta(3);
```

Error variance-covariance matrices

The variance-covariance matrix on observation noise is `covobs(.,.)` set, in our example, by:

```
covobs(1,1) = 0.3; covobs(1,2) = 0.; covobs(1,3) = 0.;
covobs(2,1) = 0.; covobs(2,2) = 0.1; covobs(2,3) = 0.;
covobs(3,1) = 0.; covobs(3,2) = 0.; covobs(3,3) = 0.2;
```

The variance-covariance matrix on state noise is `coveta(.,.)` set, in our example, by:

```
coveta(1,1) = 0.2; coveta(1,2) = 0.001;
coveta(2,1) = 0.001; coveta(2,2) = 0.1;
```

These matrices are not changed during the run of the model as part of the filtering process. They may be changed by the user in '`zsteer`'.

4.3.2 Kalman filter run and output

4.3.2.1 Feeding the observations to the model

The observations must be made available to the model during the run. These observations are set in the `vobs(.)` array, and the assimilation (also called the analysis step of the filter) takes place if the logical variable `zgetobs` is `.true.` (see Section 3.8.2 [Data], page 25).

These steps are typically performed in the '`zsteer`' sequence. In this sequence there should be some code such that when there are data ready to be assimilated, `zgetobs` is set to `.true.` and the data is stored in `vobs(.)`, ready for the next step processing.

4.3.2.2 Kalman filter results

The estimated states and transfers are still in the same '`.data`' files, '`res.data`' and '`tr.data`' and there is the additional file with observations, called '`obs.data`' (see Section 3.8.1 [Observations], page 24). Each time `zgetobs` is `.true.` the data, and the optimally weighted innovations are output in the file associated with data, '`data.data`' (see Section 3.8.2 [Data], page 25).

4.3.3 Executing code after the analysis

The analysis takes place before the time step advance when `zgetobs` is `.true.`. It may be usefull to add some code after the analysis and before the time step advance. For example the analysis may lead to absurd values for some states or parameters, it could be usefull to correct them in that case. The sequence included after the analysis is called '`kalsteer`'. At this point, in addition to the usual variables the following variables could be usefull:

`etafor(.)`
: The state before the analysis.

`kgain(.)`
: The Kalman gain.

`innobs(.)`
: The innovation vector (observations coherent with the states minus data values).

`covana(.,.)`
: The variance-covariance error matrix after the analysis.

At each time step the derivative of the observation function with respect to transfer and cells variables are recomputed. The elimination of transfers is also performed to get the partial derivative of the observation function of the equivalent model, with states only, with respect to the states. In other words, the Kalman filter does not follow the TEF formalism, because the advance of the var-covar matrix could not yet be set in the TEF form.

`obspha(.,.)`
: derivative of observation function in state space with respect to cell variables.

4.4 Feedback gain

The feedback dynamic gain associated with a feedback loop can be expressed as the inverse Borel transform of the coefficient of the reduced scalar coupling matrix, $g(\tau)$, associated with a transfer. A Borel sweep provides this $g(\tau)$. Therefore it is an interesting tool for the characterization of the feedback loop[4].

As explained in the ZOOM web page document http://www.lmd.jussieu.fr/ZOOM/doc/Feedback_Gain.pdf, this allows for the calculation of the dynamic gain and factor of any feedback that goes through a unique transfer variable. An example of the conclusions that can be drawn from such an analysis is provided in the same document.

For linear systems – whose GTLS are autonomous along the whole trajectory – the τ function of the feedback gain is independent of the position on the system trajectory. But in general it is dependant, and one can analyse the function $g(\tau; t)$ defined on a segment t of the trajectory.

The document introducing the TEF-ZOOM technique explains how a Crank-Nicolson scheme for the time discretisation symbolically gives the solution of the Borel transform of the system. One can identify the `dt` variable with the Borel τ within a factor 2. Hence, to numerically study the τ dependency of the transform of various coefficients in the system coupling matrix at one point in time, one can calculate the Borel transform of the TLS solutions by making a time-step sweep.

The function $g(\tau; t)$ is simply output for the feedback gain attached to a unique `ff(k)` transfer variable. All the relevant informations should be entered in the '`zinit`' sequence.

4.4.1 Specifying the Borel sweep

First of all the logical flag `ZBorel` should be raised:

```
ZBorel=.true.;
```

The index of the studied transfer is given in the `index_ff_gain` variable

```
index_ff_gain=7;
```

At each time step a Borel sweep may be performed. The time steps of interest are specified with three variables, one for the first step, one for the last step and one for the number of steps between two Borel sweeps:

[4] More generally, the Borel sweep allows the numerical study of the dependency in τ of the Borel transform of various coefficients in the system coupling matrix.

`istep_B_deb`
: First time step for the Borel sweep.

`istep_B_fin`
: Last time step for the Borel sweep.

`istep_B_inc`
: Number of time steps between Borel sweeps.

In the following examples Borel sweeps are performed from the time step 1000 up to the time step 1200, with a sweep at each time step:

```
istep_B_deb=1000;
istep_B_fin=1200;
istep_B_inc=1;
```

For each Borel sweep, the range of the τ variable should be set. As this is a multiplicative variable the initial value, a multiplicative factor and the number of values are to be given.

`tau_B_ini`
: Initial value for τ.

`tau_B_mult`
: Multiplicative factor for sweep in *tau*.

`itau_max` Number of τ values.

For example, in the following, at each time step, the Borel transform will be computed for τ values starting at 0.2 and then multiplied a hundred times by $\sqrt{\sqrt{2}}$

```
tau_B_ini=0.2;
tau_B_mult=sqrt(sqrt(2.));
itau_max=100;
```

When the initial value of τ is set to a negative value (*i.e.* `tau_B_ini=-0.2;`), the Borel sweep will first be applied with `itau_max` negative values for `-0.2`, `tau_B_mult*(-0.2)`,..., then for the zero value, and finally for the symetric positive values, resulting in `2*itau_max+1` values for τ.

The whole example reads

```
! -------------------
! Feedback gain
! Borel
! -------------------
ZBorel=.true.;
if ZBorel
<  istep_B_deb=1000;
   istep_B_fin=1200;
   istep_B_inc=1;
;
   index_ff_gain=7;
   tau_B_ini=0.2;
   tau_B_mult=sqrt(sqrt(2.));
   itau_max=100;
```

```
    z_pr/Borel/:tau_B_mult,tau_B_ini*(tau_B_mult)**itau_max;
>;
```

Instead of using the index of the transfer in `index_ff_gain` it is possible to specify the name of the transfer.In that case the transfer is specified by the `zborel for` macro. For example if the transfer selected for the feedback gain computation is *b_transfer*, it can be selected with:

```
zborel for: b_transfer;
```

4.4.2 Borel sweep results

The file '`tau_Borel.data`' gives the τ values of the *tau* sweep, and the file '`gains.data`' records the feedback gain function values of $g(\tau)$, with one line for each sweep along the trajectory. In the 1.01 version, a new feature is also provided giving the poles and residuals of the Borel transform in the file '`vpgains.data`'. Consult the subroutine `Boreleig` for (not definitive) output description.

One can easily obtain the surface contours of $g(t, \tau)$ using the Fortran program provided as '`gains.f`' and its compilation shell '`gains.xqt`', that builds 2D histograms for PAW, in which one uses the '`borels.kumac`' provided kumac.

4.5 Stability analysis of fastest modes

The preceding analyses are done along with a simulation. One has also the possibility of using in a more classical fashion the state advance matrix A_{st}, after the end of the simulation. Code to perform the SVD (Singular Value Decomposition) of the state matrix A_{st} and also of $A_{st} + A_{st}^{\dagger}$ is provided with Miniker. The singular elements of these two matrices correspond to the most rapid modes of instability of the perturbed system.

The Singular value decomposition of a matrix is noted

$$UwV^{\dagger}$$

An executable file, '`sltc.exe`' is generated and running this file will produce the corresponding results.

4.5.1 Singular Value Decomposition with cmz

The cmz macro `smod SLTC` prepares a main program ('`circul`' of +PATCH SLTC), provided as a base for user's own analysis, in the directory '`sltc/`'.

4.5.2 Singular Value Decomposition with make

To compile the singular value decomposition executable with `make` you can do

```
make sltc.exe
```

If you want to have a separate directory for the SVD, you should copy the sequence '`dimetaphi.inc`' (or make a link to that file) to the directory. You should also copy the file '`Makefile.sltc`' from the '`template/`' directory in this directory, rename it '`Makefile`' and set the Miniker directory path in the `miniker_dir` variable. For example, if the Miniker directory is in '`/u/src/mini_ker`':

```
miniker_dir = /u/src/mini_ker
```

4.5.3 Singular Value Decomposition run and output

As it is, the '`sltc.exe`' executable generated by the compilation determines the SVD. This program requires '`title.tex`' (see [Title file], page 8) to transmit a title for output and graphics, and '`aspha.data`' (see Section 2.3.3 [Running a simulation and using the output], page 11) to access the state matrix. To get access to these files (in case they are not in the current directory) it is possible to make a link to the corresponding files in the model directory. Once it is done the program may be run:

```
./sltc.exe
```

The files '`u.data`', '`w.data`', and '`v.data`' holds the singular elements for A_{st} (U, w and V), and '`us.data`', '`ws.data`', and '`vs.data`' holds the singular elements of $A_{st} + A_{st}^{\dagger}$. The corresponding macros '`.kumac`' for PAW[5] are also generated.

4.6 Generalized linear tangent system analysis

The state matrix A_{st} may also be used to compute the GTLS propagator (or state transition matrix applied to perturbation), after the simulation. The algorithm is a finite product of 5th order development of $\Phi(t + \delta t, t) = \exp A_{st}\delta t$. Numerous element of analysis are given, in particular the determination of the Lyapunov exponents of the system.

An executable file, '`sltcirc.exe`' is generated and running this file will produce the corresponding results.

4.6.1 Generalized tangent linear system with cmz

The cmz macro `smod SLTCIRC` prepares a main program ('`circule`' of +PATCH SLTCIRC), in the directory '`sltcirc/`'.

4.6.2 Generalized tangent linear system with make

To compile the GTLS analysis executable with `make` you can do

```
make sltcirc.exe
```

If you want to have a separate directory for the GTLS analysis, you should copy the sequence '`dimetaphi.inc`' (or make a link to that file) to the directory. You should also copy the file '`Makefile.sltcirc`' from the '`template/`' directory in this directory and rename it '`Makefile`' and set the Miniker directory path in the `miniker_dir` variable.

4.6.3 Generalized tangent linear system analysis run and output

The '`sltcirc.exe`' executable generated by the compilation computes the elements of analysis of the system. This program requires '`title.tex`' to transmit a title for output and graphics (see [Title file], page 8), '`aspha.data`' to access the state matrix and '`dres.data`', because time-step can be changed along the simulation (see Section 2.3.3 [Running a simulation and using the output], page 11)[6]. To get access to these files (in case they are not in the current directory) it is possible to make a link to the corresponding files in the model directory. Once it is done the program may be run:

```
./sltcirc.exe
```

[5] Explanation in the research paper about SLTC (Al1 2003) available on request.

[6] cf our research texts about propagator analyses in SLTC, and "les Gains sur champs (Al1 2003-2004)"

The following table gives the correspondence between variable name, result file and ntuple number, with a short explanation:

var	**file**	**ntuple**	**explanation**
`p(.,.)`	'`phit.data`'	55	propagator from 0 to t, $\Phi(t,0)$
`up(.,.)`	'`uphit.data`'	50	Left singular vectors U in the SVD of Φ
`wp(.)`	'`wphit.data`'	51	singulat values w in the SVD of Φ
`vp(.,.)`	'`vphit.data`'	52	Right Singular Vectors V in the SVD of Φ
`wr(.)`	'`wr.data`'	53	real part of eigen values of $\Phi(t,0)$
`wi(.)`	'`wi.data`'	54	imaginary part of eigen values of $\Phi(t,0)$
`lwp(.)`	'`lwphit.data`'	67	Lyapunov exponents

5 Advanced use of Miniker with make

5.1 Make variables

The '`Makefile.miniker`' Makefile provided in the distribution should be included as it defines a lot of important variables and rules.

The following make variables can be set by the user:

`miniker_dir`
: that variable should hold the Miniker sources directory. If you installed Miniker that variable should be set to '`$(includedir)/mini_ker`'. If you use the sources right from the sources directory it should be set to the sources package directory.

`MTNDIRS` This variable can hold a '`:`' delimited list of directories that will be searched for mortran include files.

`CMFDIRS` This variable can hold a '`:`' delimited list of directories that will be searched for cmz directive include files.

`SEL` This variable holds a '`,`' delimited list of select flags, for example `monitor`, `grid1d`, `debug`.

`LDADD` This variable can be used to add libraries flags and files. It is used in the default linking command/rule.

`miniker_user_objects`
: This variable should hold a space separated list of additional object files to be linked with the model and helper object files.

`CAR2TXTFLAGS`
: cmz directives preprocessor flag.

`kalman` This variable should be set to 1 if you want to use the kalman filter (see Section 4.3 [Kalman filter], page 34).

`double` This variable should be set to 1 if you want to have a double precision code (see Section 3.4 [Double precision], page 21).

The following variables are allready set and may be used (some are set by ./configure see Section A.4.2 [Configuration], page 49):

`miniker_principal_objects`
: The list of object files needed for the model build, together with some helper object files often used but not strictly required for the linking.

`DEPDIR` The name of a hidden directory containing the dependencies computed for the main mortran files.

`F77`
`FC`
`FFLAGS`

`LDFLAGS` Compiler and linker related variables set by ./configure.

`LIBS` This variable should hold the link flags and files required to build Miniker, set by ./configure.

`CAR2TXT`
`MORTRAN`
`MTNFLAGS`
`MTNDEPEND`
Preprocessor and preprocessor flags, set by ./configure.

5.2 Rules

The following rules are defined in the '`Makefile.miniker`' file.

`miniker-clean`
remove the fortran files generated from the mortran files. Remove the object files.

`miniker-mtn-clean`
remove the mortran files generated from the files with cmz directives.

Various rules to preprocess files with cmz directives and mortran files and to compile fortran files.

If the user needs a mortran main file, he may take advantage of the rule used to compute the dependencies of a mortran file. If the file is called, say, '`mtnfile.mtn`' leading to '`mtnfile.f`', the following include should lead to the automatic creation, updating and inclusion of a file describing the dependencies of '`mtnfile.mtn`' in the '`Makefile`':

```
include $(DEPDIR)/mtnfile.Pf
```

5.3 Linking rule

The rule used for the linking of the model file is not in the '`Makefile.miniker`' file but should be provided in the user '`Makefile`' for more flexibility. The default rule uses the variables `miniker_user_objects` for additional object files and `LDADD` for additionnal linking flags and files, those variables are there to be changed by the user.

The object files required by the Miniker code are in the make variable `miniker_principal_objects`, this variable is also used. The value of the variables `FC` for the Fortran compiler, `FFLAGS` for the Fortran compiler flags and `LDFLAGS` for the linker flags should be set to right values; `LIBS` should also be right and hold the link flags and link files required to compile the Miniker model. These variables are set by by `./configure` during configuration (see Section A.4.2 [Configuration], page 49) and used in the default rule:

```
$(model_file): $(miniker_user_objects) $(miniker_principal_objects)
        $(FC) $(FFLAGS) $(LDFLAGS) $^ $(LDADD) $(LIBS) -o $@
```

In case this isn't right it may be freely changed. You should certainly refer to the section "Top" in *GNU Make Manual* manual to understand what that rule exactly means and make your own.

Concepts index

P

R

S

T

U

V

Z

Variables, macros and functions index

T

V

Z

Appendix A Installation

A.1 Programming environments

Miniker is not a traditionnal software in that it isn't a library or an interpreter but rather a set of source and macro file that combines with the user model code and enable to build a binary program corresponding with the model. It requires a build environment with a preprocessor, a compiler and facilities that automate these steps.

Two different environment are proposed. One use `cmz` (http://wwwcmz.web.cern.ch/wwwcmz/index.html), while the other is based on `make`. Other libraries are needed, the CERN Program Library (cernlib) and lapack.

A.2 Common requisites

Whatever method is used a fortran 77 compiler is required. The compilers that have been used so far are g77, gfortran and the sun solaris compiler.

When usng CMZ, the CERN Program Library, available at http://wwwasd.web.cern.ch/wwwasd/cernlib/, has to be installed. With make, internal source files copied from the cernlib may be used instead but then some examples won't be available, since they rely on some mathematical functions provided by the CERN library. On windows, in case you want to use the compiler from the GNU compiler collection with cygwin or MINGW/MSYS you can use the binaries provided at http://zyao.home.cern.ch/zyao/cernlib.html. On Mac OS X, the cernlib provided by fink (package `cernlib-devel`) can be used.

You should also have LAPACK, available at http://www.netlib.org/lapack/. LAPACK can also be installed as part of the CERN Library or as part of the http://math-atlas.sourceforge.net/ implementation. On most linux distributions a lapack package is available. On Mac OS X, the ATLAS implementation provided by fink or the frameworks from Xcode can be used.

A.3 Miniker with cmz

First of all you have to get the cmz file '`mini_ker.cmz`' and put it in a directory. In that same directory you should create a directory for each of your models. In the model directory you should copy the file '`selseq.kumac`' available with Miniker, and create your own cmz file for your model, called for example '`mymodel.cmz`'. You should also have installed the kumac macro files handling mortan compilation, the associated shell scripts and the mortran preprocessor.

A.4 Miniker with make

A.4.1 Additional requirements for Miniker with make

The package has been tested with GNU `make` and solaris `make`.

Suitable preprocessors should also be installed. Two preprocessors are required, one that preprocess the cmz directives, and a mortran preprocessor. A cmz directives processor written in `perl`, is distributed in the `car2txt` package available at http://www.environnement.ens.fr/perso/dumas/mini_ker/software.html. A

mortran package with a command able to preprocess a mortran file given on the command line with a syntax similar with the cpp command line syntax is also required. Such a mortran is available at http://www.environnement.ens.fr/perso/dumas/mini_ker/software.html.

A.4.2 Configuration

The package is available at http://www.environnement.ens.fr/perso/dumas/mini_ker/software.html. It is available as a compresssed tar archive. On UNIX, with GNU tar it may be unpacked using

```
$ tar xzvf mini_ker-1.02.00.7.tar.gz
```

The detection of the compiler, the preprocessors (car2txt and mortran), and the libraries are performed by the configure script. This script sets the apropriate variables in makefiles. It can be run with:

```
$ cd mini_ker-1.02.00.7
$ ./configure
```

If the output of ./configure doesn't show any error it means that all the components are here. It is possible to give ./configure switches and also specify environment variables (see also ./configure --help):

--disable-cernlib
: Use the internal cernlib source files, even if a cernlib is detected.

--with-static-cernlib
: This command line switch forces a static linking with the cernlib (or a dynamic linking if set to no).

--with-cernlib
: This command line switch can be used to specify the cernlib location (if not detected or you want to use a specific cernlib).

--with-blas
--with-lapack
: With this command switch, you can specify the location of the blas and lapack libraries.

  For example, on mac OS X this can be used to specify the blas and lapack from the Apple frameworks:

```
./configure \
--with-blas=/System/Library/Frameworks/vecLib.framework/versions/A/vecLib \
--with-lapack=/System/Library/Frameworks/vecLib.framework/versions/A/vecLib
```

F77
FC
FFLAGS
LDFLAGS
: Classical compiler, compiler flags and linker flags.

MORTRAN
: This environment variable holds the mortran preprocessor command (default is mortran).

MTNFLAGS
: This environment variable holds command line arguments for the mortran preprocessor. It is empty in the default case.

`MTN` This environment variable may be used to specify the mortran executable name and/or path, it should be used by the `mortran` commmand. (default is empty, which leads to a mortran executable called `mtn`).

`MTNDEPEND`
This environment variable may be used to specify the mortran dependencies checker executable. It should be used by the `mortran` commmand. (default is empty, which leads to a mortran dependencies checker called `mtndepend`).

After a proper configuration, if `make` is run then the example models should be build. You have to perform the configuration only once.

A.4.3 Installation with make

Miniker can be installed by running

```
make install
```

It should copy the sources and the '`Makefile.miniker`' file in a '`mini_ker`' directory in the `$(includedir)` directory, and copy the templates in '`$(datadir)/mini_ker`'. The default for `$(includedir)` is '`/usr/local/include`' and the default for `$(datadir)` is '`/usr/local/share`', these defaults may be changed by `./configure` switches '`--prefix`', '`--includedir`' and '`--datadir`'. See `./configure --help` and the '`INSTALL`' file for more informations. The helper script '`start_miniker`' should also be installed.

The installation is not required to use comfortably Miniker. Indeed the only thing that changes with the sources and the '`Makefile.miniker`' directory location is the `miniker_dir` variable in a project `Makefile`.

Appendix B Cmz directives reference

The cmz directives are described together with the other features of cmz in the cmz manual at http://wwwcmz.web.cern.ch/wwwcmz/, the important ones are nevertheless recalled here, especially for those that use make and don't need the whole features of cmz.

After the description of the generic features, we turn to the cmz directive of interest. There are three kinds of cmz directives that are of use within Miniker: one kind that introduce files, the other for conditionnal compilation and the third for sequence inclusion.

B.1 Cmz directives general syntax

The cmz directives always begin with a '`+`' in the first column, optionnaly followed by any number of '`_`' that may be used for indentation, then the directive label, case insensitive, followed by the directive arguments separated by '`,`'. The arguments are also case insensitive. Optional spaces may be around directive arguments. An optionnal '`.`' ends the directive arguments and begin a comment, everything that follows that '`.`' is ignored.

B.2 Conditional expressions

A directive argument common to all the directives is the conditionnal expression. A conditionnal expression may be true or false, it is a combination of select flags. the select flags are combined with logical operators. A select flag itself is true if it was selected. A select flag *selflag* is selected by using the `sel` *`selflag`* instruction in cmz. It is selected by passing the `-D` *`selflag`* command line switch to the call of the cmz directives preprocessor when using make.

A '`-`' negates the expression that follows. Parenthesis '`(`' and '`)`' are used for the grouping of subexpressions. '`|`' and '`,`' are for the boolean or: an expression with a or is true if the expression on the left or the expression on the right of the or is true. '`&`' is for the boolean and: an expression with an and is true if the expression on the left and the expression on the right are true.

The grouping is left to right when there is no parenthesis, with or and '`&`' having the same precedence. Therefore

```
a&b|c     ≡     (a&b)|c
a|b&c     ≡     (a|b)&c
a|b&c  is not   a|(b&c)
a&b|c  is not   a&(b|c)
```

B.3 File introduction directives

A file (or sequence) introduction directive appears at the beginning of the file. There are two different directives, one is `DECK` for normal files, the other is `KEEP` for include files (sequences). The first argument is the name of the file. The file name may not be larger than 32 characters and is converted to lower case in the general case. The optionnal following arguments may be of 2 type (and may be mixed, separated by '`,`'):

conditional
: A conditionnal is introduced by `IF=` followed by a conditionnal expression described in Section B.2 [Conditional expressions], page 51. The file is preprocessed if the conditionnal expression is true.

language specification

A language specification is introduced by a `T=`. The most common languages are '`mtn`' for the mortran, '`ftn`' for fortran not preprocessed, '`f77`' for preprocessed fortran, '`c`' for the c language and '`txt`' for text files. In general the language of the file determines the name of files the preprocessed file is extracted to, the comment style and the command for inclusions.

It is a common practice to have wrong language type in `KEEP` as the language may be determined from the `DECK` that include them with cmz, or from their file name with make. This is not recommended and considered a bad practice.

Such a directive will always appear in cmz, as it is built-in. It is recommended to have one when using make too, even though it is not required in most cases. Indeed make uses the file name directly and finds the language and file type by looking at the file extension. make should then pass the language type with a `--lang` *lang* command line switch when calling the cmz directives preprocessor. With make, the convention is to have '`cm`' added before the normal file suffix and after the '.'. The table table 3.2 shows the matching between suffixes, file type and file language.

For example, a file beginning with

```
+Deck, subroutine_foo, If=monitor&-simple, T=f77.
```

is a main preprocessed fortran file that will only be generated if '`monitor`' is selected and '`simple`' is not selected. The file to be preprocessed by make should have the '`.cmF`' suffix, and be called '`subroutine_foo.cmF`'.

A file beginning with

```
+KEEP,inc_common,If=monitor|interface,T=mtn
```

is an mortran include file that should be processed only if '`monitor`' or '`interface`' is selected. The file to be preprocessed by make should have the '`cmmti`' suffix and be called '`inc_common.cmmti`'. The resulting file when make is used will be called '`inc_common.mti`'.

B.4 Conditional directives

Conditional directives may be used to conditionnaly skip blocks of code. There are 4 conditional directives: `if`, `elseif`, `else` and `endif`. `+if` begins a conditional directives sequence, with argument a conditional expression. If the expression is true the block of code following the `+if` is output in the resulting file, up to another conditional directive, if it is false the code block is skipped. If the expression is false and the following conditional directive is `+elseif`, the same procedure is followed with the argument of `+elseif` which is also a conditionnal expression. More than one `+elseif` may follow a `+if`. If a `+if` or `+elseif` expression is true the following code block is output and all the following `+elseif` code blocks are skipped. If all the `+if` and `+elseif` expressions are false and the following coditionnal directive is `+else` then the block following the `+else` is output. If a previous expression was true the code block following the `+else` is skipped. The last code block is closed by `+endif`.

Conditionnal directives may be nested, a `+if` begins a deeper conditionnal sequences directives that is ended by the corresponding `+endif`.

The simplest example is:

```
     some code;
+IF,monitor
     code output only if monitor is true;
+ENDIF
```

If '`monitor`' is selected, the `+if` block is output, it leads to

```
     some code;
     code output only if monitor is true;
```

If '`monitor`' isn't selected the `+if` block is skipped, it leads to

```
     some code;
```

An example with `+else` may be:

```
+IF,double
 call dmysub(eta);
+ELSE
 call smysub(eta);
+ENDIF
```

If '`double`' is selected the code output is `call dmysub(eta);`, if '`double`' isn't selected the code output is `call dmysub(eta);`.

Here is a self explanatory example of use of `+elseif`:

```
+IF,monitor
  code used if monitor is selected;
+ELSEIF,kalman
  code used if kalman is selected and monitor is not;
+ELSE
  code used if kalman and monitor are not selected;
+ENDIF
```

And last an example of nested conditional directives:

```
+IF,monitor
  code used if monitor is selected;
+_IF,kalman. deep if
    code used if monitor and kalman are selected;
+_ELSE. deep else
    code used if monitor is selected and kalman is not;
+_ENDIF. end the deep conditionnals sequence
+ELSE
  code used if monitor is not selected;
+_IF,kalman
    code used if monitor is not selected but kalman is;
+_ELSE
    code used if monitor and kalman are not selected;
+_ENDIF
  other code used if monitor is not selected;
+ENDIF
```

B.5 File inclusion directive

The file (sequence) inclusion directive is `seq`. The argument of `seq` is an include files '`,`' separated list. The include files are `Keep` in cmz. The following optional arguments may be mixed:

conditional
: A conditionnal is introduced by `IF=` followed by a conditionnal expression described in Section B.2 [Conditional expressions], page 51. The directive is ignored if the conditionnal expression is false.

T=noinclude
: When this argument is present the text of the sequence will always be included in the file where the `+seq` appears.

When there is no `T=noinclude` argument, the `+seq` directive may be replaced with an inclusion command suitable for the language of the file being processed, if such command has been specified.

For example if we have the following sequence

```
+KEEP,inc,lang=C
typedef struct incstr {char* msg};
```

And the following code in the file being processed:

```
+DECK,mainf,lang=C
+SEQ,inc
int main (int argc, char* argv) { exit(0); }
```

the processing of '`mainf`' should lead to the file '`mainf.c`', containing an include command for '`inc`':

```
#include "inc.h"
int main (int argc, char* argv) { exit(0); }
```

In case the `+seq` has the `T=noinclude`:

```
+DECK,mainf,lang=C
+SEQ,inc,T=noinclude
int main (int argc, char* argv) { exit(0); }
```

The processing of '`mainf`' should lead to the file '`mainf.c`' containing the text of '`inc`':

```
typedef struct incstr {char* msg};
int main (int argc, char* argv) { exit(0); }
```

B.6 The '`self`' directive

The `self` directive is an obsolete directive that may be used for conditionnal skipping of code. For a better approach see Section B.4 [Conditional directives], page 52. The optionnal argument of `+SELF` is `If=` followed by a conditionnal expression. If the conditionnal expression is true the code following the directive is output, if it is false the code is skipped up to any directive (including another `+SELF`) except `+seq`.

Appendix C Copying This Manual

C.1 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

C.1.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.