IFS DOCUMENTATION – Cy38r1
Operational implementation 19 June 2012

PART VI: TECHNICAL AND COMPUTATIONAL PROCEDURES

Table of contents

European Centre for Medium-Range Weather Forecasts
Shinfield Park, Reading, RG2 9AX, England

Chapter 1
Structure, data flow and standards

Table of contents

1.1 INTRODUCTION

Development of what is now called the IFS was started in 1987, with the aim of providing a single software system delivering a state of the art forecast model with an integrated 4D-Var analysis scheme. Since then the code has been in a state of continuous development, incorporating improvements to scientific formulations, modifications to allow efficient utilisation of a range of High Performance Computer (HPC) architectures, and technical changes to the structure and expression of the code to improve both its efficiency and maintainability.

The IFS has, over time, grown to be a large and complex code. This chapter aims to give an overview of the high level technical structure of the IFS, describing the configurations, control structure, data flow and coding standards employed. More detailed technical information is given in the Appendices.

1.2 CONFIGURATIONS

The IFS contains many different functions within a single high level program structure, including:

- 2D and 3D model integrations;
- variational analysis (3D/4D-Var);
- adjoint and tangent linear models;
- calculation of singular vectors.

For any single execution of the program, the function is selected by means of a configuration parameter. The value of this parameter may be supplied to the IFS on the command line option (using the "`-n`" option, see Table A.1 on page 38), or by using the namelist variable `NCONF` in namelist `NAMCTO` (see Table A.3 on page 42). A detailed description of the recognised values of `NCONF` is given in Table A.11 on page 47.

1.3 STRUCTURE

All IFS configurations share a single top-level call tree:

MASTER ▷ CNT0

The routine MASTER calls CNT0 after calling routines to initialise functions such as performance monitoring and error trapping.

CNT0 reads the configuration information from the command line and namelists, and then uses the value of NCONF to call the control routine appropriate for the configuration requested. Table A.11 on page 47 describes which control routine is used for each configuration. Once the required configuration has completed, CNT0 does any necessary housekeeping and clearing up, and after printing any requested execution statistics, exits.

The top level calling tree of the forecast integration configuration looks like this:

MASTER ▷ CNT0 ▷ CNT1 ▷ CNT2 ▷ CNT3 ▷ CNT4

where CNT4 repeatedly calls STEPO (which performs a single timestep of the forecast model) in a timestepping loop. Other configurations will "hook" into this (and into each other) at an appropriate level. For example, 4D-Var has the following calling tree:

MASTER ▷ CNT0 ▷ CVA1 ▷ CVA2 ▷ CONGRAD ▷ SIM4D ▷ CNT3

Here we see SIM4D called by the minimisation function CONGRAD, and SIM4D then performs a forecast integration by calling it at level 3 (CNT3) since the previous control level was level 2 (CVA2).

A graphical representation of the IFS calling tree is shown in Figure 1.1. In this "treemap" diagram, each box represents one subroutine (and all the subroutines called from it), and the size of the box is representative of the number of lines of code it (and its children) contain. The colour of the box is a function of the name of the routine, enabling identification of the same routine that is being called from multiple locations. It can be seen from the treemap that although the forecast model integration (CNT1 and below) only form a small proportion of the code called from CNT0, it is actually called (at CNT3 level) from many parts of the IFS.

1.4 DATA FLOW

The IFS stores fields using both spectral and grid-point representations. The main spectral state variables are all stored in both a spectral representation and also in grid-point space, with both representations held in memory concurrently throughout a model integration. Other variables are stored in a grid point representation only.

1.4.1 Input/Output

For the forecast configuration (NCONF=1, see Section A.4 on page 47), the main state variables are read in from the CSTA routine. This is called from "level 3" control, i.e.

MASTER ▷ CNT0 ▷ CNT1 ▷ CNT2 ▷ CNT3 ▷ CSTA

For all other configurations, the main state variables are read in and/or initialised from the SUVAZX routine, which reads the data into the control variable. This is called from "level 1" control, i.e.

MASTER ▷ CNT0 ▷ CNT1 ▷ SU1YOM ▷ SUVAZX

MASTER ▷ CNT0 ▷ CVA1 ▷ SU1YOM ▷ SUVAZX

Similarly, the observational data is also read in at control "level 1", i.e.

MASTER ▷ CNT0 ▷ CVA1 ▷ SUOBS ▷ MKCMARPL

Postprocessing, diagnostic and coupling data is output from a model integration after the loop over model timesteps in routine CNT4.

Details of the input data files used by the IFS can be found in Section A.5 on page 48.

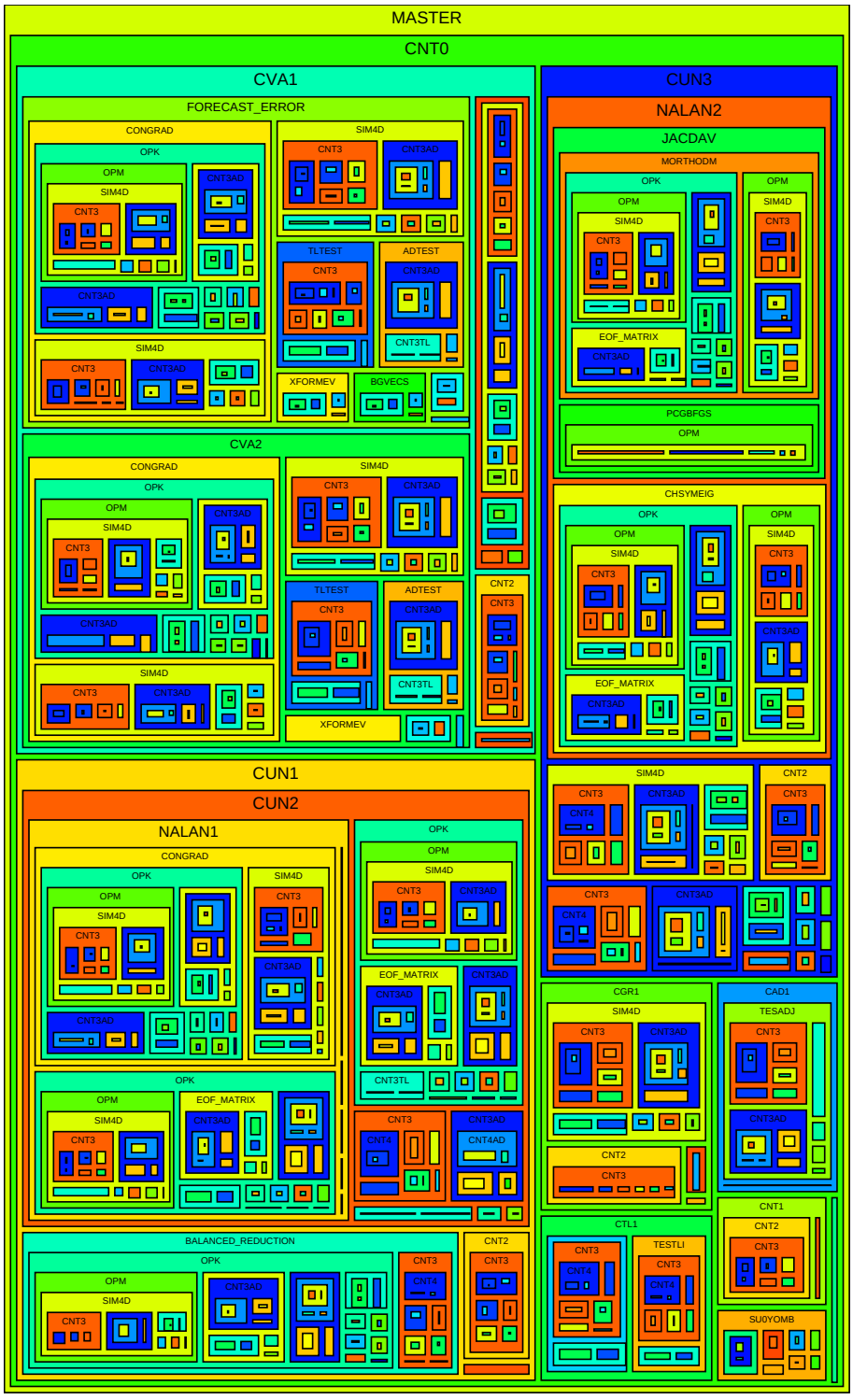

Figure 1.1 *Treemap of the* IFS *Calling Tree.*

1.4.2 Major data structures

The spectral fields are carried in the module YOMSP, in which the arrays SPA3 and SPA2 hold the 3D and 2D state variable spectral fields. Individual fields within these arrays are addressed via pointers which are defined in the same module. The grid point fields have a much more flexible storage structure, which was introduced at cycle 27, and is designed to allow the easy incorporation of new prognostic variables, without the need to know about and modify a large number of routines through IFS. The basic concept is that all the grid point variables are stored within a single structure, and that any routine which performs a generic operation on grid-point data just loops over all the grid point fields within the structure. There is, however, the potential to control action for individual fields by the use of a set of *attributes* which are associated with each field in the structure.

There are two core data-structures:

GMV Contains prognostic variables involved in the semi-implicit (u, v, T, p_s in the hydrostatic model). This can be considered to be a "fixed" data structure, with little reason for modification. The prognostic fields all have a spectral representation, and can be either two or three dimensional. There are no attributes, apart from field pointers, associated with the GMV fields.

GFL Contains all the other variables (currently q, q^l, q^i, a, O_3 in ECMWF's operational model). This is a more flexible structure that can be easily extended. All the fields are three dimensional, with the vertical extent always the number of levels in the model. The fields may have a spectral representation or be pure grid-point fields. A number of attributes are available to govern the treatment of the field in question. All fields have two modes of being accessed; either as part of the GFL structure, or as individual components.

More technical information of the implementation and usage of the GMV and GFL structures can be found in Section A.6 on page 51.

1.5 CODING STANDARDS AND CONVENTIONS

There is a comprehensive document (see Appendix F) which gives a full and complete guide to the "coding norms" to be used when writing and submitting code to the IFS system. In this section, brief highlights of some of the most important features of the coding standards are given, but it is recommended that anyone planning on writing any significant amount of code for IFS refers to the full Coding Standards document in Appendix F.

1.5.1 Style and layout

- Each file should contain only one module or procedure. The filename should be the name (in lowercase letters) of the procedure it contains, with an appropriate extension (eg. .F90 for FORTRAN 90).
- Executable lines should be written in uppercase characters, comments can use a mixture of case as appropriate (but should be in English only). A consistent style should be maintained throughout a subroutine or module.
- Use free-format FORTRAN 90, starting in column 1, but keeping lines to within 80 characters per line.
- Continuation lines are marked by the continuation character & at the end of each line to be continued *and* the start of the continuation line. Use indentation and alignment to maintain readability of long, broken lines.
- Use indentation (spaces only, no tab characters) to make the structure more obvious (ie. loops, IF blocks).
- A procedure should have only one entry and one exit point (the bottom of the procedure). Abnormal termination should be invoked with the ABOR1('Error Message') routine.
- Each data module should begin with a description of the general content of the module and the purpose of each declared variable (one line per variable).
- Each procedure should begin with comments describing:

  - the *purpose* of the procedure;
  - the *interface* details, describing the arguments in the same order they appear in the interface;
  - the *externals* (other subroutines/functions called);
  - the *method* used in the application;
  - a *reference* to further documentation;
  - the *author and date* of creation;
  - details of any *modifications* since the creation, including the author and date.
- The first and last executable statement of every subroutine should be a conditional call to DR_HOOK:
  First: `IF (LHOOK) CALL DR_HOOK('ROUTINE_NAME',0,ZHOOK_HANDLE)`
  Last: `IF (LHOOK) CALL DR_HOOK('ROUTINE_NAME',1,ZHOOK_HANDLE)`
- In a procedure, variables should be declared or USEd in the order:
  - variables USEd from modules;
  - dummy arguments (in the same order as they appear in the argument list), and using the INTENT attribute;
  - local variables.
- Loops should be written only using the DO ... ENDDO construct.
- Use the SELECT CASE construct in preference to IF/ELSEIF/ELSE/ENDIF statements.
- Use the FORTRAN 90 comparison operators rather than the FORTRAN 77 style operators (ie. "less than" should be "<" rather than ".LT.").

1.5.2 Variables

- The use of IMPLICIT NONE is mandatory.
- Each variable should be declared on a separate line, with declarations of variables with similar type and attributes being grouped together. All the attributes of a given variable should be grouped within the same instruction.
- Arrays should be declared using the DIMENSION attribute, with the shape and size of the arrays being declared inside brackets after the variable name on the declaration statement.
- The use of array syntax is not recommended, except for simple operations such as the initialisation of copying of whole arrays.
- Where a MODULE is used to import a variable into a subroutine, the ONLY attribute must be used, so that only those variables actually used by the procedure are imported.
- Derived types should be declared in a module. Such a module should contain ONLY the declaration of a single derived type (or a group of derived types if they are closely related), and any "primitive" operations on the types (such as allocation/deallocation of its components).
- All INTEGER and REAL variables and constants must be declared using explicit KIND, using the parameters defined in the modules PARKIND1 and PARKIND2. Table 1.1 shows the commonly used KIND parameters.
- The first (or first two) letters of every variable name indicate its type and scope, as described in Table 1.2. Prefixes shown in red and/or ~~crossed out~~ indicate those prefixes are not available for that particular variable type/scope.

1.5.3 Banned features

- GO TO should not be used (use instructions such as DO WHILE, EXIT, CYCLE, SELECT CASE instead).
- Use format descriptors rather than the obsolescent FORMAT statement.
- Use MODULEs rather than COMMON blocks.
- Do not change the shape or type of a variable when passing it to a subroutine.
- CHARACTER variables should be declared using the syntax `CHARACTER(LEN=`*n*`) var_name`.
- Arrays must not be declared with implicit *size* (`REAL(KIND=JPRB) :: A(*)`) but can be declared with implicit *shape* (`REAL(KIND=JPRB) :: A(:)`).

Table 1.1 *Commonly used KIND parameters.*

KIND name	SELECTED_*_KIND value(s)	Fortran 77 equivalent	Range *Approx.*	Precision
JPIS	4	INTEGER*2	$\pm 2^{15}$	–
JPIM	9	INTEGER*4	$\pm 2^{31}$	–
JPIB	12	INTEGER*8	$\pm 2^{63}$	–
JPIA[1]	9 *or* 12	INTEGER*4 *or* INTEGER*8	$\pm 2^{31}$ *or* $\pm 2^{63}$	–
JPRM	(6,37)	REAL*4	$\pm 10^{37}$	10^{-7}
JPRB	(13,300)	REAL*8	$\pm 10^{307}$	10^{-15}
JPRH[2]	(13,300) *or* (28,2400)	REAL*8 *or* REAL*16	$\pm 10^{307}$	10^{-15} 10^{-31}

[1]If 64 bit INTEGERs are available, then these are used, otherwise 32 bit INTEGERs are used.
[2]If 128 bit REALs are available, then these are used, otherwise 64 bit REALs are used.

Table 1.2 *Variable Prefix Naming Convention.*

	Fortran Type				
Scope	INTEGER	REAL	LOGICAL	CHARACTER	**Derived type**
MODULE variable	M,N	A,B,E-H, O,Q-X	L ~~LD,LL,LP~~	C ~~CD,CL,CP~~	Y ~~YD,YL,YP~~
Dummy argument	K	P ~~PP~~	LD	CD	YD
Local variable	I	Z	LL	CL	YL
Loop control	J ~~JP~~	-	-	-	-
PARAMETER	JP	PP	LP	CP	YP

1.5.4 I/O

- User supplied configuration variables should be access via a conventional formatted sequential file containing namelists (Unit NULNAM=4).
- Each namelist should be contained is a specific include (.h) file, with the filename being the same as the namelist name (in lowercase).
- Output messages should be written to unit NULOUT, error messages to unit NULERR. Do not explicitly write to units 0,6 or "*".

1.5.5 Parallelisation

- Only use MPL package for message passing, and set the CDSTRING to the name of the caller routine.

Chapter 2
Parallel implementation

Table of contents

2.1 INTRODUCTION

2.1.1 High Performance Computing architecture

Before we describe the IFS parallelisation and its associated code and data structures, it is useful to understand the basic architectures used in a typical High Performance Computing (HPC) environment, as it is these which have largely directed the design of these structures.

The obtainable performance of a Central Processing Unit (CPU) is ultimately constrained by a number of factors; some technological such as the density of transistors on the silicon, thermal characteristics and memory bandwidth, but also fundamental constraints such as the speed of light. To enable increasing performance within current technological parameters, manufacturers have for many decades exploited parallelisation as a cost effective solution, the basic concept being to replicate the basic processing unit many times, having them act in parallel on the problem being solved.

Although the hardware technology and architectures have evolved considerably over the past decades, compiler technology has not always kept pace with these changes. By and large, compilers still produce code for a single processor, and any parallelisation has to be at least directed by, if not explicitly coded by the programmer.

Today's architectures typically contain multiple layers of parallelism, which are described below:

CPU
: The basic computational unit will usually contain a small number of independent functional units. Typically each unit will be capable of performing a small number of basic operations (for example, a CPU may contain two functional units capable of doing add/multiply instructions, one functional unit for divides and one for logical operations). The parallelisation is usually obtained by "pipelining" these units. This can be thought of rather like cars on a conveyor belt in a production line - the data passes from one functional unit to another as they apply their various operations as required on the data. Once the pipeline has filled up, each functional unit will be operating on a different piece of data in the stream, and a result will pop out of the pipeline after every clock tick.

  There are two fundamental CPU architectures which it is useful to consider in a little more detail:

Scalar CPUs are the more traditional design of CPU, where the basic machine instruction operates on single units of data at a time (eg. `ADD Number_A to Number_B`). Typically these CPUs operate at high clock speeds, and their performance is limited by the speed at which data can be fed from memory. To help remove this bottleneck, scalar CPUs have a high performance, low latency memory cache, so that data which has been recently accessed can be accessed again very quickly, without the CPU having to wait long for it to arrive from main memory.

With this design of CPU, the most efficient code is such that all the data accessed within an inner loop can fit within cache - so all the functional units are kept busy, and the pipelines do not have to stop because of delays caused by accessing relatively slow main memory. This generally means the inner loop should be kept quite small (number of iterations).

Vector CPUs are more specialised CPUs, historically designed especially for HPC computer systems, and therefore significantly more expensive than scalar CPUs. In a vector CPU, the basic machine instruction operates on multiple units of data (vectors) at a time (eg. `ADD Vector_A to Vector_B`). These typically have much more complex functional units, and operate at slower clock speeds. It can take a while for a pipeline to fill up and start producing results, but once it does, it is very efficient, performing many floating point operations every clock cycle. The main performance limitation on a Vector CPU is short vectors, where the startup costs dominate and the pipelines never really get a chance to be used efficiently.

With this design of CPU, the most efficient code is such that the inner loop (the vector length) is as long (has as many iterations) as possible, so the pipeline startup costs are minimised.

The parallelisation within a CPU is generally largely exploited by the compiler. However, the programmer does have some control over the efficiency of the parallelisation, as described above; depending on the CPU architecture, the inner loop should perhaps be small or very large in order to gain maximum performance. It is also sometimes necessary (usually more so for a vector CPU) for the programmer to add directives (hints to the compiler, often describing data dependencies) in the code to enable the compiler to make the correct decision on how to pipeline the work in the inner loop.

Node

A node is a collection of CPUs which share a common memory. Any CPU in the node can access any memory on the node without the explicit collaboration of any other CPU on the node.

Although some compilers will attempt to parallelise a code over nodes, a code as complex as IFS needs a programmer to direct the parallelisation. The compiler needs to have identified to it, either:

- Chunks of code operating on independent data, so different CPUs on the same node can perform different computations without having to worry about interactions (data dependencies) with any other CPUs on the node.
- Independent iterations of a loop, so different CPUs on the same node can perform different iterations (or more commonly subsets of the total iterations) of a loop, without having to worry about interactions with any other CPUs on the node. This is the form of node parallelisation commonly exploited in IFS.

This compiler direction is achieved using OpenMP[1], which is a set of directives the programmer inserts in the code to inform the compiler that it is safe to farm out subsets of a loop's iterations to different CPUs on the node.

Starting and completing a parallel OpenMP block of code carries a certain overhead, as the operating system synchronises the CPUs and carries out any other necessary housekeeping. For this reason, it is advantageous to minimize the number of OpenMP loops in a code. In practise, this is achieved by keeping the OpenMP at a high level of the code - so instead of applying OpenMP directives around each loop in a low level computational module, it is more efficient to apply OpenMP around a loop in which this computational module is called (where the loop is over independent data points).

[1] OpenMP is a portable open API available on all commercially available shared memory HPC systems. For further information see `http://www.openmp.org/`.

OpenMP parallelisation is in some ways the easiest kind of parallelisation to implement as it appears to require very little change to code and data structures. However, first appearances can be deceptive, as the devil can be in the detail. A fundamental requirement for a correct and reliable OpenMP parallelisation is that loop iterations are independent, and that the order of execution of the iterations does not affect the final result. When a code can safely be run with different numbers of processors in the OpenMP parallelisation, with reproducibly identical results, it is said to be "Thread Safe". It is usually easy to verify that a simple loop will be thread safe, but as was just explained, IFS typically uses OpenMP at a very high level in the code - the code within an OpenMP parallelised loop can often contain deeply nested subroutine calls to complex and relatively unknown code. Verifying, debugging and fixing the thread safeness of such code is often a non trivial exercise!

Distributed Memory

This is the "top level" of parallelisation, consisting of a number of nodes, where each node has its own independent memory (shared amongst the CPUs in the node as described previously). If any CPU needs to access memory on another node, then an explicit communication (Message Passing) is required with a CPU on that remote node which has direct access to that memory.

Although a number of attempts have been made at automatically parallelising at this level, none have been able to deliver high performance and reliable results, especially to complex codes such as IFS, so programmer parallelisation is required.

Distributed Memory parallelisation requires a considerable knowledge of the dataflow and data dependencies and potentially has a much larger code impact than the shared memory (node) parallelisation:

- The full data structures need to be decomposed so that each node now has a data structure which only holds a subsection of the total data being computed by the application, along with additional metadata that allows a node to know about the subsection of data it has; such as where it is in the total data space, and which nodes contain its neighbouring data.
- Code needs to recognise it is only dealing with a subsection of the data.
- Data dependencies need to be resolved by either communicating data between relevant processors to resolve dependencies, or redecomposing (transposing) the data in such a way that the dependencies can be satisfied by the subsection of data now on the node.

Communicating data between nodes is achieved using MPI (the Message Passing Interface), although in the IFS this is hidden under an interface layer "MPL" (see Appendix B). In IFS this communication mostly takes the form of transposing the data between different computational phases of the model, and is described in more detail later in this chapter.

This communication strategy is an important characteristic of the IFS, and is a fundamental property of the spectral transform method it employs. A purely grid point model, which has data dependencies in many different dimensions during different phases of the model integration typically requires explicit communication to be invasively added throughout the model code, and generally requires special data structures with halo regions for finite difference calculations. In contrast, the IFS already (before parallelisation) has a different data structure for each major computational component of the integration, and in each phase this data structure has at least one data independent dimension (that is, different elements of the given dimension(s) can be safely computed in parallel as they are not interdependent). This means that (generally speaking) there is no communication required within the main computational phases, and the communications can be localised to the transpose/transform steps which move the data between the different data structures/representations used for different phases of the integration.

2.1.2 Overview of IFS parallelisation

Having seen the various levels of architectural parallelisation that are available, we now consider how this is applied to the IFS.

A meteorological model such as IFS may have a basic data structure for grid-point model data which is of the form shown in Listing 2.1.

Listing 2.1 *Basic model data structure.*

```
REAL Model_Data ( 1:Horiz_i,
                  1:Horiz_j,
                  1:Levels_k,
                  1:Fields    )
```

Here we have shown a basic (regular) 3D grid-point field. The IFS of course, also contains reduced grid-point, Fourier and spectral fields, but the same principles that are demonstrated here can also be extended to such fields.

The first step is to consider the distributed memory parallelisation. We need to break up, or "decompose" the data so that every node has a subset. Potentially we could decompose every dimension of "`Model_Data`", and that is what we will consider here. Of course, it is unlikely that it is ever possible to do this, as there will almost always be some kind of dependency in one or more dimensions, depending on the computational algorithm that is being applied to the data. In this case, the dimension(s) containing the dependency(s) would be left undecomposed (or possibly, if decomposition was unavoidable, extra message passing would be introduced to satisfy the data dependencies).

So, decomposing the data in every dimension, we now have, on any one node, an array of the form shown in Listing 2.2.

Listing 2.2 *Decomposed data structure.*

```
REAL Model_Data ( 1:Decomposed_Horiz_i,
                  1:Decomposed_Horiz_j,
                  1:Decomposed_Levels_k,
                  1:Decomposed_Fields )
```

(NB This regular decomposition is actually a simplification of the decomposition actually used by IFS which is described later in this chapter, but will serve to demonstrate the principles used in the parallelisation.)

Of course, there will also be some additional variables associated with this which will describe this node's position in the decomposition, who its neighbours are and other such useful information.

We now come to consider the lower two levels of parallelisation; over the node (shared memory or OpenMP parallelisation), and on the CPU.

The data structure we now have presents a problem. Both of these levels of parallelism are essentially at the loop level. The CPU parallelisation will be over the innermost loop:
`DO i = 1 , Decomposed_Horiz_i`
whilst the node parallelisation will be at an outer loop (`Decomposed_Horiz_j`, `Decomposed_Levels_k` or `Decomposed_Fields` depending on the algorithm and its data dependencies).

An issue now arises, in that we have very little way of controlling the size of these loops, which is a problem for both levels of parallelism. For the innermost loop (CPU parallelism) we would like some control over the number of iterations to maximise the efficiency of the scalar or vector CPU architecture. For the outer loop which is parallelised with OpenMP (and we try to ensure this is as outermost as possible for the efficiency reasons described earlier), we need to ensure that there are at least as many iterations as there are CPUs on the node (otherwise some CPUs would be left with nothing to do). Additionally, we would prefer that there to be many more iterations than CPUs on a node - this will ensure a better load balance of work across the CPUs on a node. (If each CPU only had one iteration of the loop, and the iterations were not all of equal cost, then the total computational cost would be determined by the

slowest iteration. If each CPU is given a number of iterations, then the costs should average out across the CPUs and a better load balance will be achieved.)

With this data structure, the size of the loops is determined by a function of the non-decomposed dimension, and the decomposition in the dimension concerned, which may be different in different parts of the code.

To avoid this performance limitation, the data structure is manipulated in IFS in such a way to give better control over the loop lengths of these performance critical loops. Before we consider how this happens, we will simplify the example, and bring it closer to the grid-point decomposition used in the IFS by removing the decomposition over levels and fields (and replace the variables describing them with the variables used if IFS). In the grid-point part of IFS there are too many dependencies in these dimensions to make them suitable for decomposition. This means we have a structure as shown in Listing 2.3.

Listing 2.3 *Simplified field structure with no decomposition over levels or fields.*

```
REAL Model_Data ( 1:Decomposed_Horiz_i,
                  1:Decomposed_Horiz_j,
                  1:NFLEVG,
                  1:NFIELDS             )
```

The first step of the manipulation is to merge the leading horizontal dimensions (`i,j`) into a single dimension `1:Decomposed_2D_Field` which contains all the (decomposed) points for a single level of a field on this node, as shown in Listing 2.4.

Listing 2.4 *Merged Leading Dimensions.*

```
REAL Model_Data ( 1:Decomposed_2D_Field,
                  1:NFLEVG,
                  1:NFIELDS              )
```

We now split the leading dimension (`Decomposed_2D_Field`) in such a way that we introduce a new (artificial) leading dimension which we can control the length of. In the physical parameterization and Eulerian dynamics code of IFS this inner loop length is called `NPROMA` and the total `Decomposed_2D_Field` is broken up into `NGPBLKS` blocks. We now have the data structure shown in Listing 2.5.

Listing 2.5 *NPROMA blocking.*

```
REAL Model_Data ( 1:NPROMA,
                  1:NFLEVG,
                  1:NFIELDS
                  1:NGPBLKS  )
```

The value of `NPROMA` is chosen by the user at run-time to suit the architecture of the CPU (small values, typically a few 10's for scalar CPUs, and larger values (100's or 1000's) for vector CPUs).

The innermost loop should perform well on the CPU (for scalar CPUs the data size should be small enough to fit in cache, and for vector CPUs the vector will be long enough to minimise overheads), and the outermost loop can be parallelised over the node using OpenMP, giving the typical code structure shown in Listing 2.6.

Listing 2.6 *Loop Parallelisation.*

```
!$ DO PARALLEL
  DO iBlock=1,NGPBLKS  ! This loop is OpenMP'd
    CALL ModelScience(Model_Data(:,:,iBlock))
  ENDDO
!$ END DO PARALLEL

  SUBROUTINE ModelScience(Model_Data)
  REAL Model_Data(NPROMA,NFields)

  DO fld=1,NFields
    DO i=1,NPROMA
      Model_Data(i,fld)=...
    ENDDO
  ENDDO

  RETURN
```

2.1.3 IFS parallelisation issues

(*a*) *IFS algorithmic structure*

The remainder of this chapter will describe in further detail the parallelisation methodology and implementation used within the IFS. It considers the four major algorithmic steps of IFS separately; the parallelisation of each step applies the general principles expressed in this introductory section, but with differences due to the data dependencies of the algorithm in question.

Figure 2.1 gives an overview of these algorithmic steps in a single timestep of an IFS model integration. A timestep is represented by the cycle running around the perimeter of the figure, containing the two main computational blocks, Grid Point Computations and Spectral Calculations, with a set of transpositions and transformations between them. The blocks in the centre of the figure represent the data decomposition employed at any step within the timestep, based on a very simple four node example.

From the figure, it can be seen that the transpose steps involve moving data between processors to form a new decomposition, which enables the following transform or computational step to perform its calculations with all a node's data dependencies satisfied within that node, so that no further communications are required within that step[2].

Note that transpositions never involve global communication, but only communication within each subset, e.g. between P1 and P2 and between P3 and P4, respectively. This improves the communication performance significantly when a large number of processor is used.

(*b*) *Overview of the "TRANS" package*

This package lies at the heart of the IFS decompositions, and is fully described in Appendix C (starting on page 73). The TRANS package is responsible for the necessary transposes and transforms that are needed to move the model data between grid-point space and spectral space as shown in Figure 2.1. Although the data is briefly in Fourier space, the IFS never has sight of this representation, as it is purely internal to the TRANS package[3].

The basic usage of the package is as follows:

[2]This is not completely correct, as the semi-Lagrangian step of the Grid Point Calculations performs some additional communications to satisfy its dynamic data dependencies.

[3]The TRANS package does allow the data in Fourier space to be manipulated by an optional user supplied subroutine.

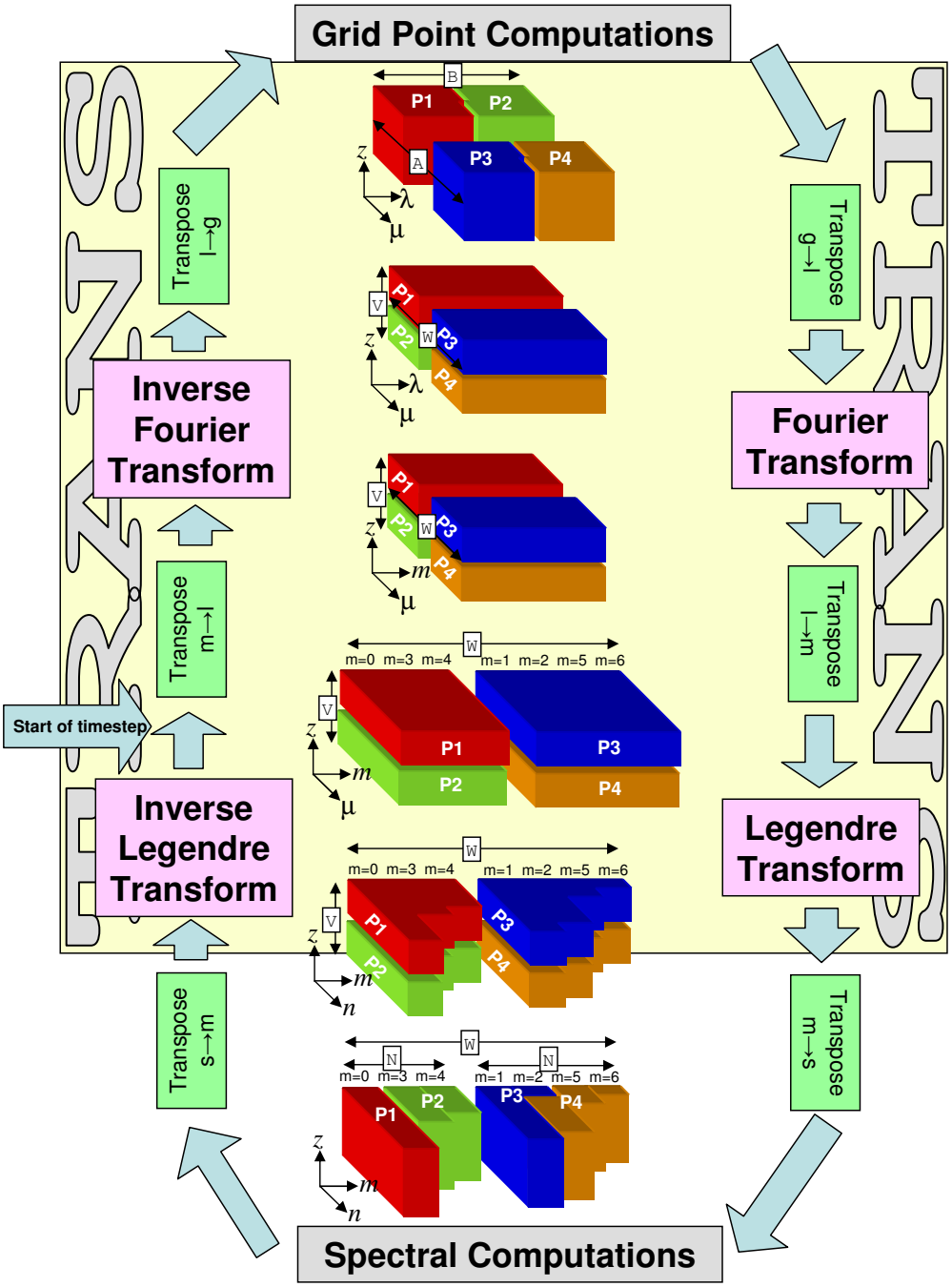

Figure 2.1 IFS *Model Timestep, showing data decompositions.*

Setup Phase
: A call to SETUP_TRANSO is required to initialise the TRANS package. This is then followed by one or more calls to SETUP_TRANS - the arguments supplied describing the grid point and spectral resolutions to be used, and flags describing how the data should be decomposed. The routine TRANS_INQ is then called, which returns, via optional arguments, a complete description of the decomposition(s) used, so that each processor knows what data it is responsible for, and how and who it is to communicate with.

Integration Phase
: During the model integration, there are a small number of TRANS package routines available for moving the data between grid-point and spectral spaced, based on the decompositions and resolutions that were described with SETUP_TRANS.

(c) *Message passing communication*

As was stated earlier, the message passing is achieved using MPI, which is encapsulated within the IFS "MPL" library (see Appendix B). MPI allows a number of different blocking[4] strategies. This is controlled by the variable MP_TYPE in MODULE YOMMP which can take the following values:

MP_TYPE=1
: Blocked mode communication using MPI_SEND/MPI_RECV. For a send operation, this means that the program continues only once MPI guarantees that the array containing data to be sent is safe to be overwritten or destroyed. There is no guarantee the data has safely arrived at its destination, only that it has been copied out of the senders array. An MPI_SEND is completely free to block until the corresponding MPI_RECV has been called on the receiving processor.

  A blocking receive (MPI_RECV) simply means that the program continues only when the message being received has arrived and is contained in the receiving array specified.

MP_TYPE=2
: Buffered mode communication using MPI_BSEND/MPI_BRECV is a little more flexible. Outgoing messages using MPI_BSEND are buffered locally by MPI in a buffer of size MBX_SIZE (defined in module YOMMP), which means that an MPI_BSEND call can return before the corresponding receive has been called on the receiving processor. The sending array is safe to reuse/destroy as all the data to be sent is safe in the MPI buffer.

MP_TYPE=3
: Immediate mode communication using MPI_ISEND/MPI_IRECV is the most flexible. Both send and receive operations return control back to calling programming immediately, and all the communication is performed in the background. Additional MPI calls are required to check or wait for the completion of a communication. The program must be careful not to reuse or destroy the sending array before MPI has confirmed that it is safe to do so, and not to use the data in the receiving array before MPI has confirmed that the data has arrived there.

(d) *Terminology: nodes, processors and CPUs*

In the text that follows we often refer to nodes. This may not necessarily correspond to a physical hardware node on an HPC system, but a subset of this node which is just a part of the total number of CPUs on the hardware's node. This sub-dividing of hardware nodes is usually done to maximise the efficiency of the OpenMP parallelisation. For example, on some IBM systems with 32 CPUs on a hardware node, we may choose to actually run some configurations of IFS with 4 CPUs per MPI task (so we have 32/4=8 MPI tasks, with each MPI task running with 4 OpenMP threads).

Unfortunately, there is sometimes some confusion between the use of the terms "node", "processor" and "CPU" in the code, variable names and documentation. This is because in the days when the distributed memory version of the code was originally being developed, there was no concept of shared memory nodes

[4] Blocking refers to the behaviour whereby a SEND or RECEIVE action "blocks", or waits to complete before allowing the program to continue.

on such architectures, so the data distribution and message passing dealt with "processors" rather than "nodes". In today's IFS the data distribution and message passing happens over and between MPI tasks which are not usually just a single processor, but a group of processors each running a single OpenMP thread, but this is not always obvious! For example, the variable "NPROC" which describes how many MPI tasks are being used for the data decomposition, is NOT necessarily the total number of PROCessors (CPUs) being used for the job, but the number of MPI tasks. The total number of processors (CPUs) is the product of NPROC and the number of CPUs per MPI task.

In the following sections, the terms "MPI task" and "processor" are used interchangeably - this allows a sensible correlation between the documentation and the code/variable names. The term "CPU" is used for describing the individual CPUs within an MPI task.

2.2 GRID POINT COMPUTATIONS

In considering the parallelisation, it is helpful to classify the computations into four categories, each of which has differing requirements, and is considered separately below.

2.2.1 Grid point dynamics and physics

These computations contain only vertical dependencies, so all grid columns can be considered to be independent of each other, allowing an arbitrary distribution of columns to processors.

(*a*) *Decomposition*

The IFS allows a number of different decomposition strategies, which are selected based on the settings of the YOMCTO module variables described in Table 2.1. Two decomposition strategies are available, a 2D scheme (the original strategy used in IFS, LEQ_REGIONS=F) and the EQ_REGIONS scheme which is now the default scheme used (LEQ_REGIONS=T).

Table 2.1 *Variables controlling Grid Point decomposition.*

Variable	Description
NPROC	Total number of processors to be used
LEQ_REGIONS	Logical controlling use of EQ_REGIONS partitioning
NPRGPNS	Number of processors in the North–South direction (LEQ_REGIONS=F)
NPRGPEW	Number of processors in the East–West direction (LEQ_REGIONS=F)
LSPLIT	Allows the splitting of latitude rows

The simplest (LEQ_REGIONS=F) distribution is achieved by setting:

```
NPRGPEW = 1 ; NPRGPNS = NPROC ; LSPLIT = .FALSE.
```

which basically assigns a set of complete latitude rows to every processor. A good static load balance can only be achieved for very specific values of NPROC and a particular model resolution, but even this becomes difficult when the reduced model grid is used. The advantage of this distribution is that it matches the distribution used by the Fourier transforms, so eliminates the transposition between these algorithmic stages.

Some improvement to this distribution can be made by setting:

```
LSPLIT = .TRUE.
```

which allows a line of latitude to be split so that part of it is assigned to one processor and the remainder is assigned to the next processor. This removes the load balance problems, but the amount of parallelism remains limited by approximately 2/3 times the number of latitude rows owing to the FFT and Legendre Transforms. There are also efficiency disadvantages in the semi-Lagrangian message passing, because the long-thin shape of the decompositions results in a relatively large amount of communication required with neighbouring processors.

The best distribution is obtained by setting:

```
NPRGPEW = x
NPRGPNS = y
```

where `x * y = NPROC`

which provides for considerably increased parallelism, and potentially (depending on the values of "x" and "y"), a much "squarer" shape of domain, which results in a reduced communication volume in the semi-Lagrangian scheme.

An example decomposition is shown in a number of figures. In this example, we have set:

```
NPRGPEW = 2
NPRGPNS = 3
LSPLIT = .TRUE.
```

The example shows the decomposition of a representative small reduced grid (this means there are less points near the pole, with the number of grid points per latitude row increasing towards the equator), which has 19 latitude rows[5].

The calculation of the decomposition is carried out over two steps, which are illustrated in Figure 2.2. In the first step, the total number of points in a field is split as equally as possible in the North–South direction (the "A" set). In Figure 2.2 we see that a total of 152 points have been split between the three "A" sets, giving two partitions with 51 points, and one with 50 points. As a consequence of setting `LSPLIT=.TRUE.` there will be some latitude rows split between two "A" sets. This introduces a slight complication in the addressing of some arrays where information is required about each latitude row, as the split rows will appear in two "A" sets. We therefore introduce the concept of "latitude strips" - for most rows, this is a full row of points, but for the split rows, there are two "strips", one for the "A" set with the first section of the row, and the other for the adjacent "A" set with the remaining section of the row. Such arrays, instead of being dimensioned by the total number of latitude rows (`NDGLG` or `NDGL`) are dimensioned by the maximum possible number of "latitude strips", `NDGLH + NPRGPNS - 1`.

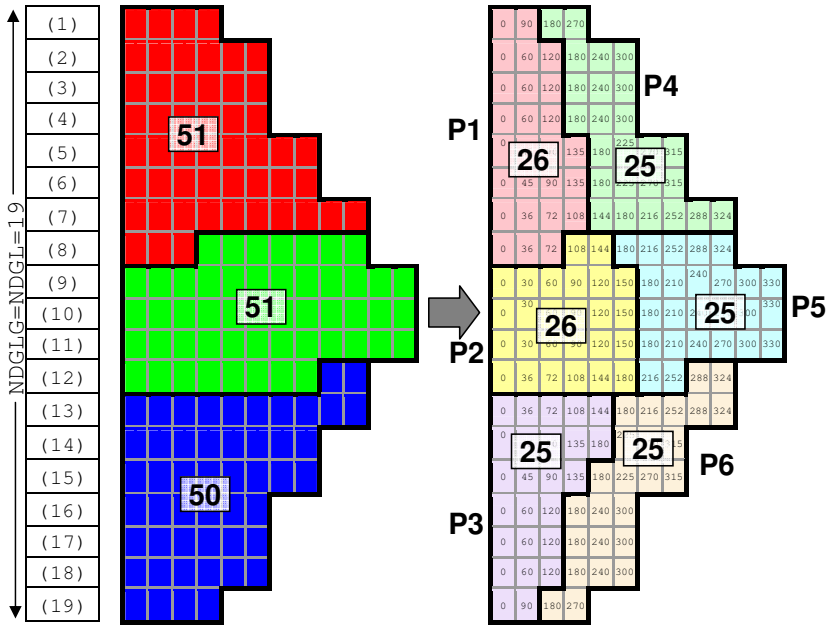

Figure 2.2 *Grid point decomposition, showing the two stages of decomposition on 6 processors.*

[5]Note, that although the IFS code for distributing grid points among processors is fully flexible as exemplified here with Figure 2.2, it is actually used in a more restricted manner due to constrains of the spectral transform method and limitation in the FFT transform routines (in IFS the number of latitude rows (`NDGLG`) must always be an even number).

The next step is the decomposition of each of the three "A" set partitions in the East–West direction (the "B" set), in such a way that each subpartition of the "A" set contains an equal (or as equal as possible) number of points. The algorithm used to achieve this selects points for a subpartition on the criteria that the new point is the point on the partition with the smallest difference in longitude from the previous point added. Again, this is illustrated in Figure 2.2, where each grid box's longitude is shown.

The static distribution is fully described by the namelist variables in Table 2.1, which means that all the required information about distributions and communications required for the various transpose steps can be calculated in the setup phase, by the "TRANS" package, and stored in the IFS MODULE YOMMP. Some of the more widely used variables describing the grid point decomposition are shown in Figures 2.3 and 2.4, and described in Table 2.2.

Figure 2.3 *Variables describing the Grid Point Decomposition.*

Table 2.2 *Variables describing the grid point decomposition.*

Variable	Array dimensions and description
LSPLITLAT	(1:NDGENL) Logical indicating whether a given row on the "A" set is split with another "A" set.
MYFRSTACTLAT	Scalar The first latitude row (global index) on this "A" set (1..NDGLG) (Equivalent to NFRSTLAT(MYSETA))
MYLATS	(1:NDGENL) The latitude row (global index) a given row on this "A" set corresponds to.
MYLSTACTLAT	Scalar The last latitude row (global index) on this "A" set (1..NDGLG) (Equivalent to NLSTLAT(MYSETA))
MYPROC	Scalar Logical processor ID (1 .. NPROC). Note, processor numbering does not follow the normal FORTRANarray ordering (row first), but instead runs in a column first order, so Processor "1" is in the North Western corner, processor "2" is to the South of this and so on.
MYSETA	Scalar Which "A" set (North–South) this processor is in (1..NPRGPNS).
MYSETB	Scalar Which "B" set (East–West) this processor is in (1..NPRGPEW).
NDGENL	Scalar Number of latitude rows handled by this "A" set.
NFRSTLAT	(1:NPRGPNS) The first latitude row (global index) for a given "A" set. (1..NDGLG)
NFRSTLOFF	Scalar Offset of the first latitude row (global index). (Equivalent to MYFRSTACTLAT-1)
NLSTLAT	(1:NPRGPNS) The last latitude row (global index) for a given "A" set. (1..NDGLG)
NPTRFRSTLAT	(1:NPRGPNS) Index of the first latitude strip on the given "A" set. (Used for indexing NSTA and NONL arrays)
NPTRLAT	(1:NDGLG) Index of the first latitude strip of the given global latitude.(Used for indexing NSTA and NONL arrays)
NPTRLSTLAT	(1:NPRGPNS) Index of the last latitude strip on the given "A" set. (Used for indexing NSTA and NONL arrays)
NSTA	(1:NDGLG+NPRGPNS-1 , 1:NPRGPEW) Number of grid points from Greenwich meridian at the start of the given latitude strip on the given "B" set. Counting starts at 1, so for a grid point at the start of a row (ie. on the meridian) NSTA(Index,1)=1
NONL	(1:NDGLG+NPRGPNS-1 , 1:NPRGPEW) Number of grid points on this latitude strip within my "B" set.

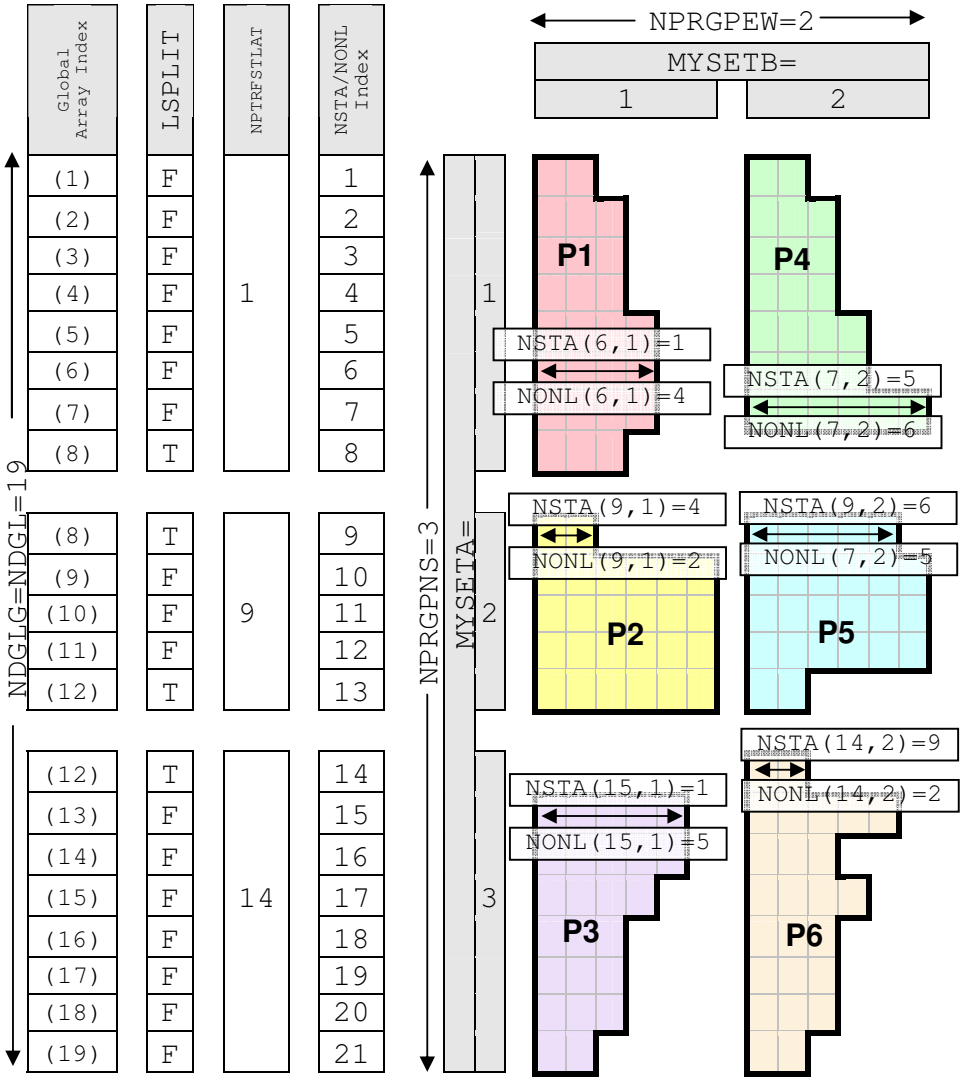

Figure 2.4 *NSTA and NONL in the Grid Point Decomposition.*

2.2.2 EQ_REGIONS

Since the mid-90s IFS has used a 2-dimensional scheme for partitioning grid point space to MPI tasks. While this scheme has served ECMWF well there has nevertheless been some areas of concern, namely, communication overheads for IFS reduced grids at the poles to support the Semi-Lagrangian scheme; and the halo requirements needed to support the interpolation of fields between model and radiation grids.

These issues have been addressed by the implementation of a new partitioning scheme called EQ_REGIONS which is characterised by an increasing number of partitions in bands from the poles to the equator. The number of bands and the number of partitions in each particular band are derived so as to provide partitions of equal area and small 'diameter'. The EQ_REGIONS algorithm used in IFS is based on the work of Paul Leopardi, School of Mathematics, University of New South Wales, Sydney, Australia.

The differences between EQ_REGIONS partitioning and 2D partitioning can be clearly seen in Figure 2.5 and Figure 2.6 for 256 MPI tasks, Figure 2.7 and Figure 2.8 for 512 tasks, and Figure 2.9 and Figure 2.10 for 1024 tasks.

From a code point of view the differences between the old 2D partitioning and the new EQ_REGIONS partitioning are relatively simple. For the 2D scheme, there were loops such as,

```
DO JB=1,NPRGPEW
  DO JA=1,NPRGPNS

  ENDDO
ENDDO
```

where, NPRGPEW and NPRGPNS are the number of EW and NS bands (or sets).

For EQ_REGIONS partitioning loops were simply transformed into,

```
DO JA=1,N_REGIONS_NS
  DO JB=1,N_REGIONS(JA)

  ENDDO
ENDDO
```

where, N_REGIONS_NS is the number of N-S EQ_REGIONS bands, and N_REGIONS(:) and array containing the number of partitions for each band.

In total some 100 IFS routines were modified with such transformations. It should be noted that the above loop transformation supports both 2D and EQ_REGIONS partitioning, i.e. to use 2D partitioning, a simple namelist variable would be set LEQ_REGIONS=F, which would result in the following initialisation,

```
N_REGIONS_NS=NPRGPNS
N_REGIONS(:)=NPRGPEW
```

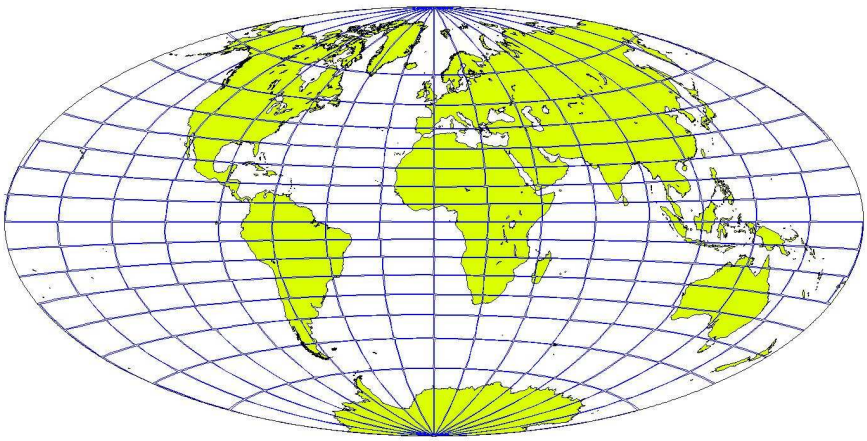

Figure 2.5 *2D partitioning, for 256 MPI tasks.*

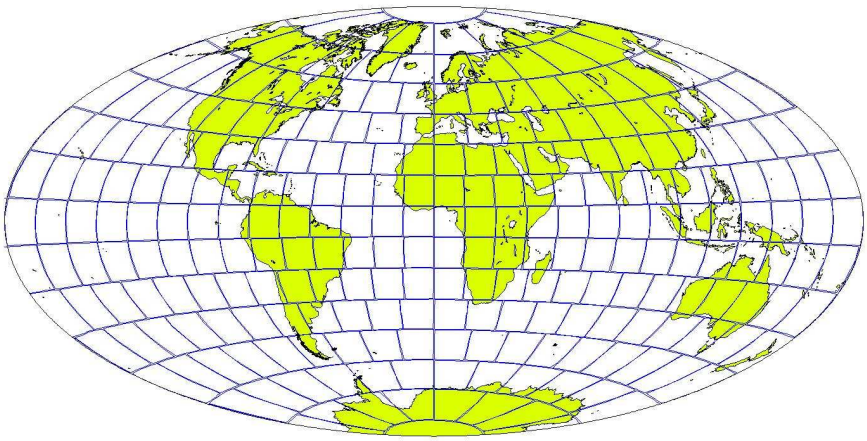

Figure 2.6 *EQ_REGIONS partitioning, for 256 MPI tasks.*

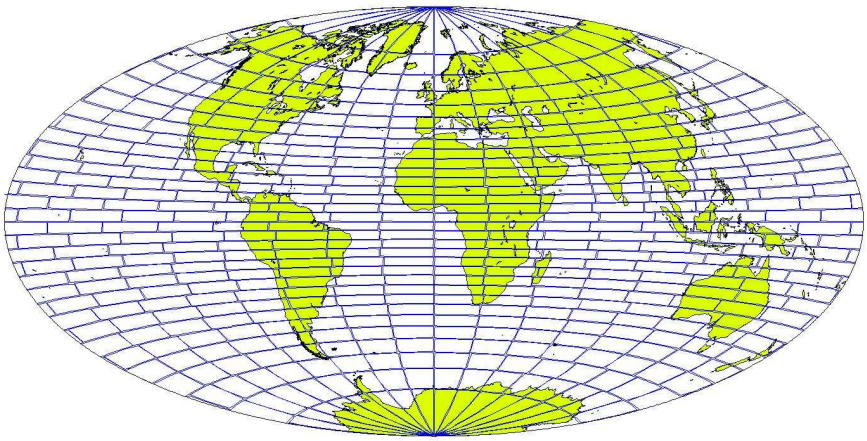

Figure 2.7 *2D partitioning, for 512 MPI tasks.*

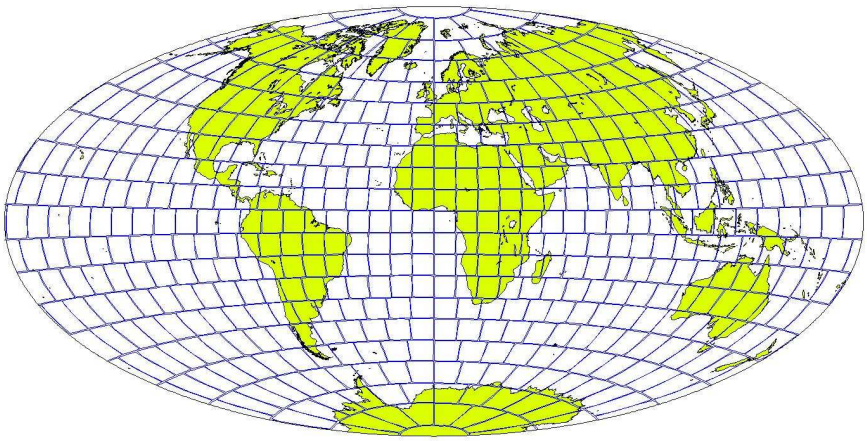

Figure 2.8 *EQ_REGIONS partitioning, for 512 MPI tasks.*

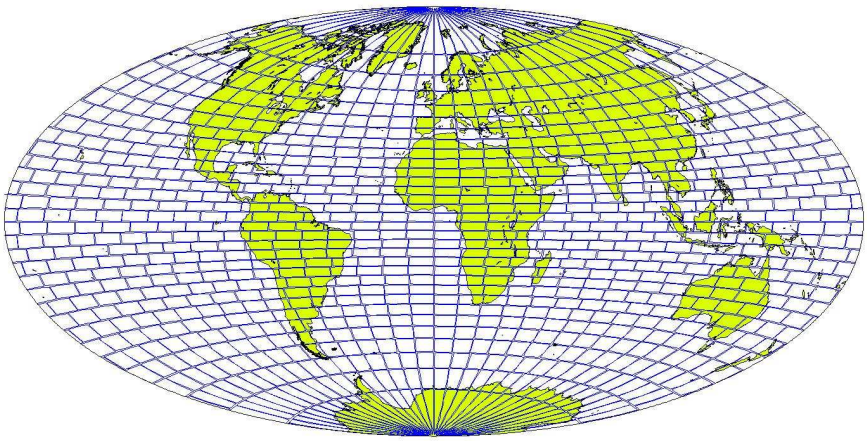

Figure 2.9 *2D partitioning, for 1024 MPI tasks.*

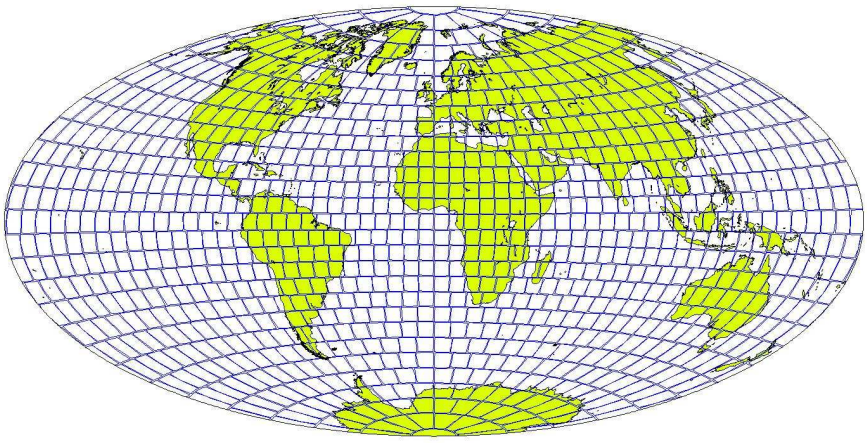

Figure 2.10 *EQ_REGIONS partitioning, for 1024 MPI tasks.*

The description of the EQ_REGIONS algorithm, and mathematical proof are described in great detail in the original paper by Leopardi. This algorithm results in partitions of equal area and small 'diameter'. However, this would not be sufficient for an IFS implementation, as the density of grid-points on the globe varies with the latitude, the greatest density being at the poles and the least density at the equator. This imbalance has been measured at 13% for a T799 model with 512 partitions when using the EQ_REGIONS algorithm to provide the bounds information (start/end latitude, start/end longitude) for each partition.

The solution to this imbalance issue was to use the EQ_REGIONS algorithm to only provide the band information, i.e. the number of N-S bands and the number of partitions per band. Then the IFS partitioning code would use this information in a similar way to that used for 2D partitioning, resulting in an equal number of grid-points per partition. With this approach there was only ONE new data structure (N_REGIONS(:)) used to store the number of partitions in each band.

The characteristic features of this partitioning approach are square-like partitions for most of the globe and polar caps together with a significant improvement in the convergence at the poles.

2.2.3 Radiation

The radiation calculations are performed on a lower resolution grid, in order to reduce their computational cost. The radiation grid is decomposed using the same algorithm as the "normal" grid, and the necessary data is interpolated to and from this grid.

2.2.4 Semi-Lagrangian advection

(*a*) *Introduction*

The semi-Lagrangian calculations in the IFS consist of two parts called by routine CALL_SL:

LAPINEA Computation of a trajectory from a grid point backwards in time to determine the departure point.

LAPINEB Interpolation of various fields to the departure point.

For a distributed memory parallelisation both these parts require access to grid-point data held on neighbouring processors and message passing is required to obtain these data.

The grid-column data that could potentially be required on a processor from neighbouring processors (called the halo) is calculated from the model time step TSTEP and a conservative estimate of the global maximum wind likely to be encountered (VMAX2 (m/sec)). Typical values for VMAX2 are 150-200 m/sec. The advantage of using a fixed (large) VMAX2 is that semi-Lagrangian communication tables can be calculated once and for all during the setup phase after the distribution of grid columns to processors has been defined. This determines the width of the SL halo which is NSLWIDE.

This pre-determined pattern then allows efficient block transfers of halo data between processors. The disadvantage is that a large amount of the halo data that is communicated may not really be required because the wind speed may be much lower than VMAX2. This problem is addressed by an "on-demand" scheme, in which only the data that will be required for the interpolations is exchanged before LAPINEB. However, the full halos must be exchanged for the fields used for calculation of the departure points before LAPINEA. The halo points required for the interpolations in LAPINEB are determined in LASCAW - where a 2-D integer array called MASK_SL is set for all points in the stencil round each departure point.

The semi-Lagrangian communication tables are calculated by the subroutine SLRSET called from SLCSET within SUSC2.

A processor can have any continuous block of grid columns on the sphere (see Figure 2.11) and so a processor's halo cannot be described only with NSLWIDE. SLCSET is called on each processor to calculate arrays which describe the halo of grid-point columns required by itself, based on VMAX2 and TSTEP and on the additional stencil requirements of the semi-Lagrangian interpolation method. Once this is done, SLRSET is called to exchange this halo information with other processors so that each processor knows

what data needs to be sent and received and which processors to communicate with. All this is done once at initialization time.

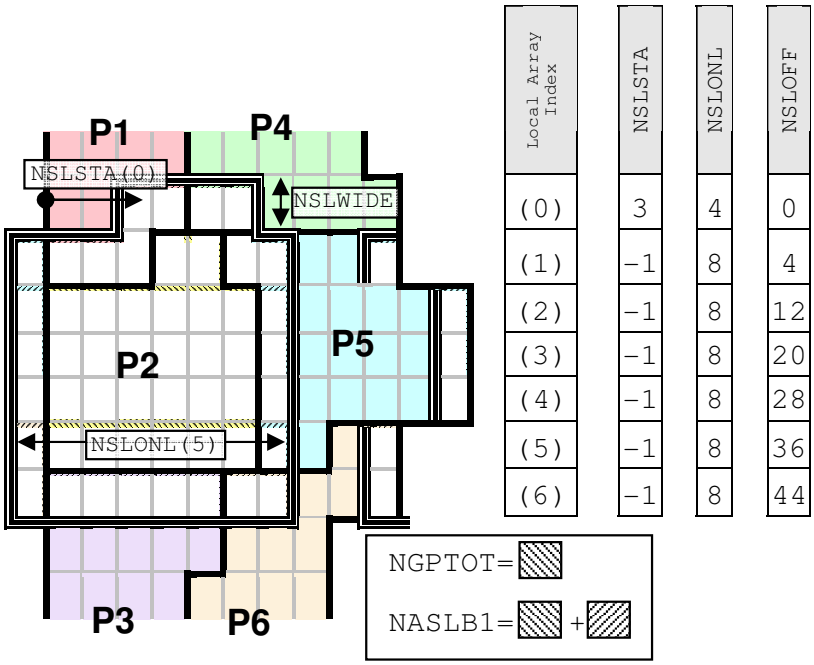

Figure 2.11 *The semi-Lagrangian Halo (For processor 2).*

For each model time step the full halo for the fields needed for calculation of the departure point in LAPINEA is constructed by calling SLCOMM1 to perform the message passing and SLEXTPOL1 to initialize those halo grid points that only require to be copied from other grid points held on the local processor. To simplify the interpolation routines, halo points are cyclically mirrored for complete latitudes in the east-west direction, and mirror-extended near the poles.

After the calculation of the departure point and before the interpolations in LAPINEB, SLCOMM2A is called to perform the message passing to initialise the required halo points and SLEXTPOL2 called to initialise the non-message passed part of the halo.

(*b*) *SLCSET*

SLCSET is called to set array variables to describe the SL halo for the local processor. NSLSTA, NSLONL and NSLOFF, are dimensioned by the number of latitudes that cover the halo and core region (see Figure 2.11) and are, briefly:

NSLSTA(JN) Starting grid point (most westerly relative to Greenwich) for halo on relative latitude JN (is negative if the area starts west of Greenwich)

NSLONL(JN) Number of halo and core (i.e. belonging to this processor) grid points on relative latitude JN

NSLOFF(JN) Offset from beginning of SL buffer to first halo grid point on relative latitude JN

The semi-Lagrangian buffer PB1 contains the variables needed for semi-Lagrangian interpolation. It has a 1-dimensional data structure with the storage is organized from north towards south. The total size is calculated in SLCSET and called NASLB1. To improve vector efficiency and cache performance the "horizontal" collapsed dimension is the innermost loop in the semi-Lagrangian buffer. NASLB1 is just the container size and it may be increased slightly in SLCSET to avoid bank conflicts on vector machines. The

second dimension represents the fields in the semi-Lagrangian buffer and will vary according to the chosen semi-Lagrangian configuration. This strategy makes it simple to add new fields to the semi-Lagrangian buffer - no changes in the message-passing routines are needed.

The calculation of the halo is done as follows.

For each latitude:

(1) The minimum (i.e. most westerly) and maximum (i.e. most easterly) angles on the sphere are determined for the local processor's core region by considering NSLWIDE latitudes to the north and south.
(2) The angular distance a particle can travel on the sphere (given the maximum wind speed VMAX2 and timestep TSTEP) is then subtracted and added respectively from the above minimum and maximum angles.
(3) The angular distances are converted to grid points and at the same time a further grid point is added to satisfy the requirements of the interpolation method used. For more complex interpolation methods more points are required.
(4) NSLSTA, NSLONL and NSLOFF are then updated for this latitude, such that the number of grid points required for the halo and core region is never greater than the number of grid points on the whole latitude plus the extra points (IPERIOD) required for the interpolation. In addition, the NSLWIDE latitudes at the north and south poles are forced to require full latitudes to simplify the design.

To aid debugging, space is also reserved on each latitude for NSLPAD grid points east and west of the halo. As these points are initialized to HUGE, any attempt to use this data in an interpolation routine will result in an immediate floating point exception, which can simplify the detection of programming errors in SL interpolation routines. NSLPAD is 0 by default.

NSLCORE contains the position of each core point in the SL buffer.

NSLEXT is used to simplify a SL buffer (lat, lon) offset calculation in LASCAW. This reduces an "IF test" (to account for phase change over poles) and a "modulo function", to a simple array access. As a result LASCAW becomes more efficient and more maintainable.

(c) *SLRSET*

SLRSET is called by SLCSET at initialization time to determine the detailed send- and receive-list information that will be used later by SLCOMM1 and SLCOMM2A during model execution. This is achieved by a global communication where send and receive lists are exchanged in terms of global (lat,lon) coordinates.

The data structures initialized by SLRSET are as follows:

NSLPROCS is a scalar which defines the number of processors that the local processor has to communicate with during SL halo communication.

NSLCOMM contains the list of processors that the local processor has to communicate with, and is dimensioned 1: NSLPROCS.

NSENDNUM, NRECVNUM contain the number of send and receive (lat, lon) pairs that the local processor has to communicate. The difference between elements (N) and (N+1) contain the number of entries that apply to processor N.

NRLSTLAT, NRLSTLON describe the global latitude and longitude of the grid-point columns to be received during SL halo communication. Columns to be received from processor N start at entry NRECVNUM(N) in these arrays.

NSLSTLAT, NSLSTLON describe the global latitude and longitude of the grid-point columns to be sent during SL halo communication. Columns to be sent to processor N start at entry NSENDNUM(N) in these arrays.

(*d*) *SLCOMM1 and SLCOMM2A*

SLCOMM1 and SLCOMM2A are called at each model time step to obtain grid-point halo data from neighbouring processors for the semi-Lagrangian calculations.

SLCOMM1 communicates the full halo before the departure point calculations in LAPINEA.

SLCOMM2A uses an "on-demand" scheme to communicate only the required halo points for fields used in the interpolations in LAPINEB.

The "on-demand" scheme in SLCOMM2A works by using a MASK_SL array set in LASCAW which is non-zero for all points needed in the interpolation. The MASK_SL is stored as the superposition in the vertical of all points required on each level - although different sets of points are needed for each level, having a single horizontal mask considerably simplifies the buffer creation. Fields which have linear interpolation - KFIELD_TYPE=1 - communicate a 2×2 stencil and fields which have cubic interpolation - KFIELD_TYPE=2 - communicate a 4×4 stencil.

The message passing in SLCOMM2A is in 2 steps: firstly a list of points required by "the processor" is communicated to the surrounding processors, and then the required points are communicated back from the surrounding processors to "the processor".

The packing and unpacking of the message passing buffers is parallelised using OpenMP.

For LIMP_NOOLAP=.T. the message passing is done with non-blocking MPI_ISENDs, blocking MPI_RECVs followed by a MPI_WAIT on the send requests and the message passing is not overlapped with the buffer creation.

For LIMP_NOOLAP=.F. the message passing is allowed to overlap with the buffer packing.

2.3 FOURIER TRANSFORM

For the Fast Fourier Transforms (FFTs), the Fourier coefficients are fully determined for each field from the gridpoint data on a latitude. The individual latitudes are all independent as are the vertical levels and the fields. In practice, independence across the fields is not exploited in the current code, so the quantity of exploitable parallelism is limited to the product of the number of latitudes and the number levels. For a typical operational resolution T511L60 configuration, this is tens of thousands. This approach allows an efficient serial FFT routine to be used and precludes fine-grain parallelism which is unavoidable in parallel FFT implementations.

The decomposition of latitude rows / zonal waves is described in Figure 2.12, where the rows are distributed as equally as possible across the "W" set. The "W" set's size, NPRTRW, is almost always set equal to NPRGPNS, the size of the "A" set in gridpoint space. The distribution of rows over the "W" set is similar to the distribution of rows over the "A" set in gridpoint space, the major difference being that there is no concept of "split" rows in wave space, as the FFT algorithm requires a full row of data to operate on. The variables used to describe the decomposition of latitude rows are detailed in Table 2.3.

Table 2.3 *Variables describing the decomposition of Fourier latitudes.*

Variable	Array dimensions and description
MYSETW	Scalar Which "W" set (zonal waves) this processor is in (1..NPRTRW)
NDGLL	Scalar Number of Fourier latitude on this processor.
NPROCL	(1:NDGL) The "W" set responsible for a given global Fourier latitude.
NPTRLS	(1:NPRTRW) Global index of first Fourier latitude for a given "W" set.

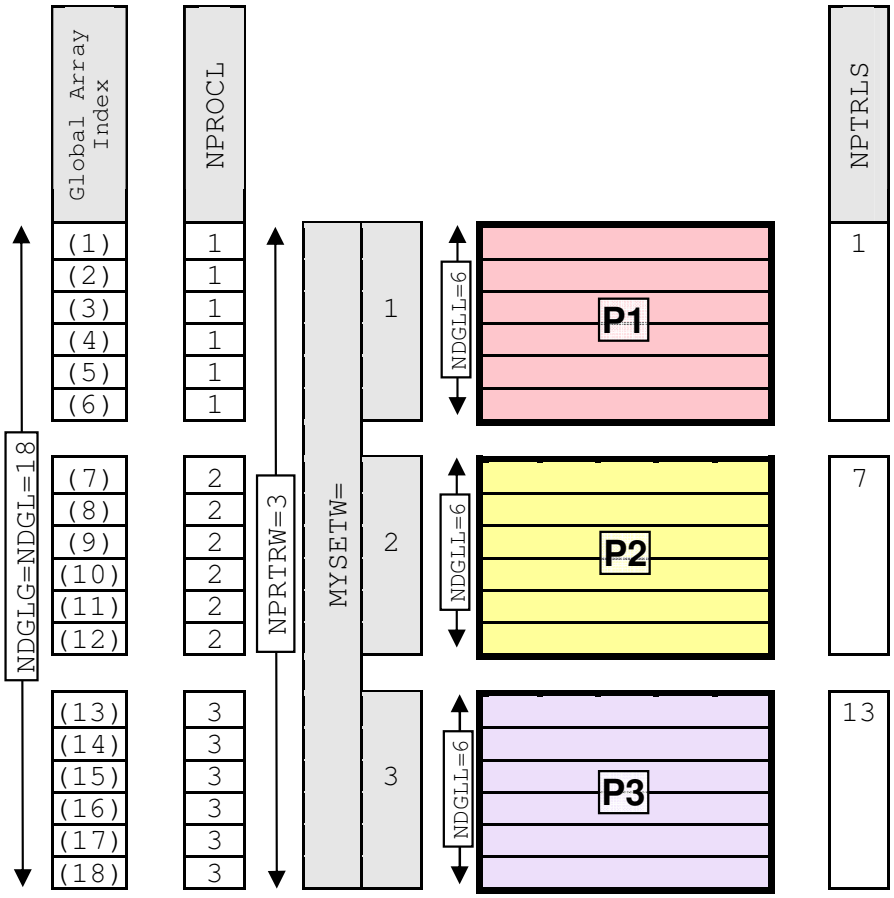

Figure 2.12 *Decomposition of zonal waves in Fourier space. Shown here is a single (first) vertical level.*

The decomposition over levels, which is also shared with the Spectral Transforms/Computations is described in Figure 2.13, where the vertical levels are distributed as equally as possible across the "V" set. The "V" set's size, `NPRTRV`, is almost always set equal to `NPRGPEW`, the size of the "B" set in gridpoint space. The variables used to describe the decomposition of vertical levels are detailed in Table 2.4.

2.4 LEGENDRE TRANSFORM

The spectral wave numbers are distributed in a round-robin fashion over the "W" set as shown in Figure 2.14. The top half of the figure shows all the wave numbers for an example T21 representation, with the colours of each zonal wavenumber m indicating which "W" it will belong to. The lower half of the figure shows the decomposed spectral data (for a single level), with each "W" set having as equal as possible number of spectral co-efficients.

The variables used to describe the decomposition of spectral wave numbers are shown in Table 2.5.

The spectral data for the Legendre transforms are decomposed vertically, with the same decomposition as is employed for the FFTs, described in Figure 2.13 and Table 2.4.

In almost all parts of IFS, it is sufficient to have a subset of the spectral coefficients, namely the subset this processor is responsible for (as shown in Figure 2.14). However, a global view is required when initial data is read, when post processed spectral fields are gathered, and when the spectral cost function contributions are accumulated. The global spectral data structure (see Figure 2.15) is designed so that local parts from each processor (in processor order) within a "W" set are stored next to each other. To avoid memory waste, the data are stored in a one-dimensional structure.

The variables used to describe this global data structure are shown in Table 2.6.

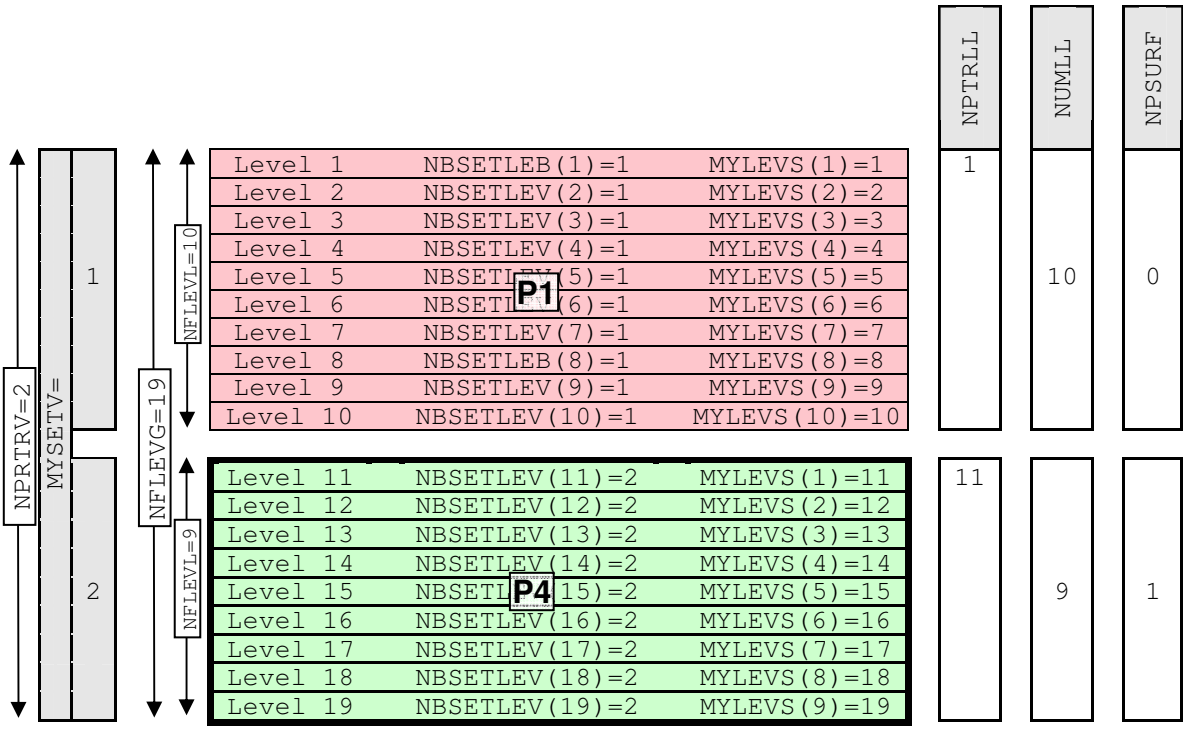

Figure 2.13 *Decomposition of vertical levels used in Fourier and spectral space. Shown here is a single "W" set.*

Table 2.4 *Variables describing the decomposition of levels in Fourier and spectral space.*

Variable	Array dimensions and description
MYSETV	Scalar Which "V" set (Vertical levels) this processor is in (1..NPRTRV)
NFLEVL	Scalar Number of vertical levels on this "V" set.
NPSP	Scalar Set to "1" on the "V" set member containing the surface pressure (and any other surface fields) which take part in the spectral transform.
NPSURF	(1:NPRTRV) Contains the value of NPSP for a given "V" set.
NPTRLL	(1:NPRTRV+1) The first level treated by a given "V" set. (For coding simplicity, NPTRLL(NPRTRV+1) = NPTRLL(NPRTRV))
NUMLL	(1:NPRTRV+1) The number of levels treated by a given "V" set. (For coding simplicity, NUMLL(NPRTRV+1) = 0)

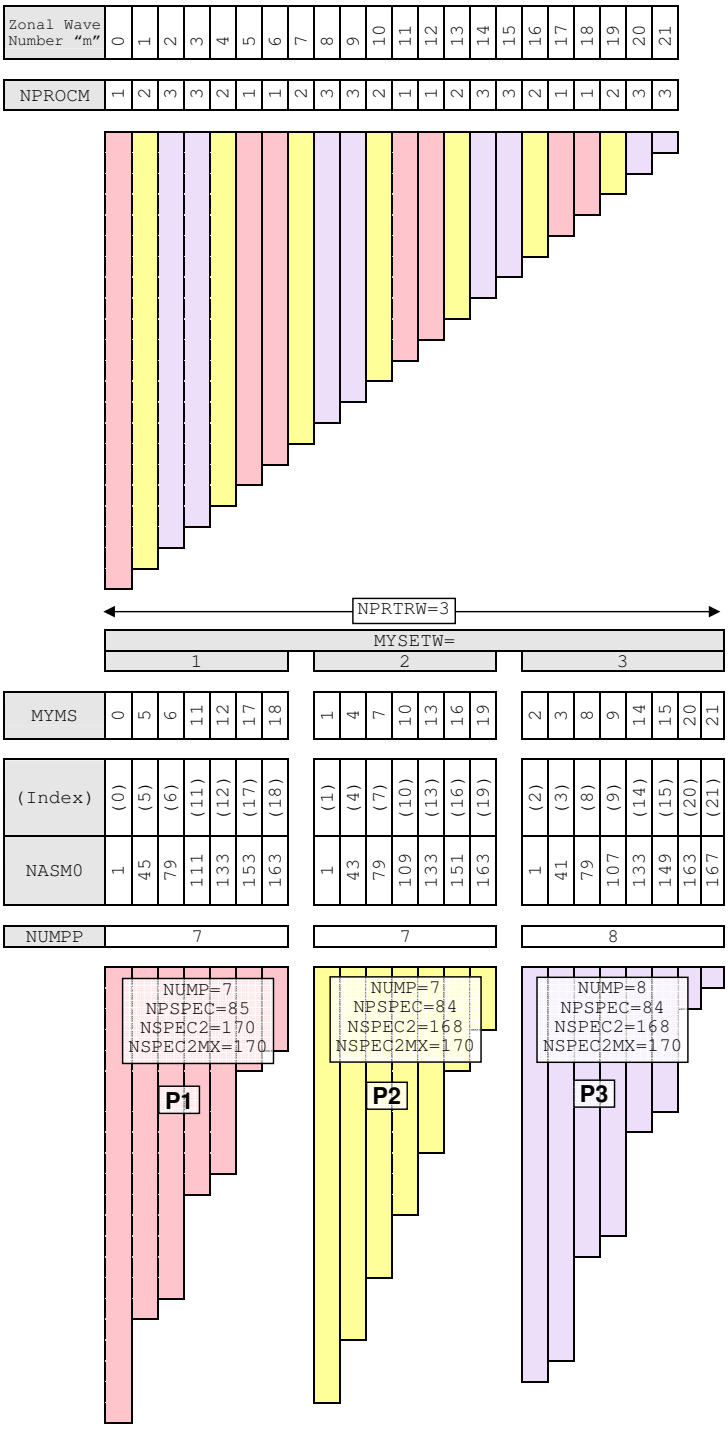

Figure 2.14 *Distribution of zonal wave numbers (T21 spectral triangle).*

Table 2.5 *Variables describing the decomposition of levels in Fourier and spectral space.*

Variable	Array dimensions and description
MYMS	(1:NUMP) Ordered list of the zonal wave numbers m on a given "W" set.
MYSETW	Scalar Which "W" set (spectral waves) this processor is in (1..NPRTRW)
NASM0	(1:NSMAX) Address in spectral array of a given zonal wave number m. For each "W" set, only the subset (NUMP) of wave numbers on that "W" set are defined.
NPROCM	(1:NSMAX) Gives process which is responsible for Legendre transforms, NMI and spectral space calculations for a given zonal wave number m.
NSPEC	Scalar Number of real spectral coefficients on this "W" set.
NSPEC2	Scalar Number of complex spectral coefficients on this "W" set. (This is simply 2*NSPEC)
NSPEC2MX	Scalar Maximum number of complex spectral coefficients over the "W" sets.
NUMP	Scalar Number of spectral wave numbers on this "W" set.
NUMPP	(1:NPRTRW) Number of spectral wave numbers on a given "W" set.

Table 2.6 *Variables describing the global representation of spectral data.*

Variable	Array dimensions and description
NALLMS	(1:NSMAX) Gives the real spectral wave number for a given wave in the global spectral wave structure.
NDIM0G	(0:NSMAX-1) Gives the index into the global spectral wave structure of the first coefficient for a given spectral wave number.
NPOSSP	(1:NPRTRW) Gives the index into the global spectral wave structure of the first wave number of a given "W" set.
NPTRMS	(1:NPRTRW) First wavenumber index of a given "W" set.

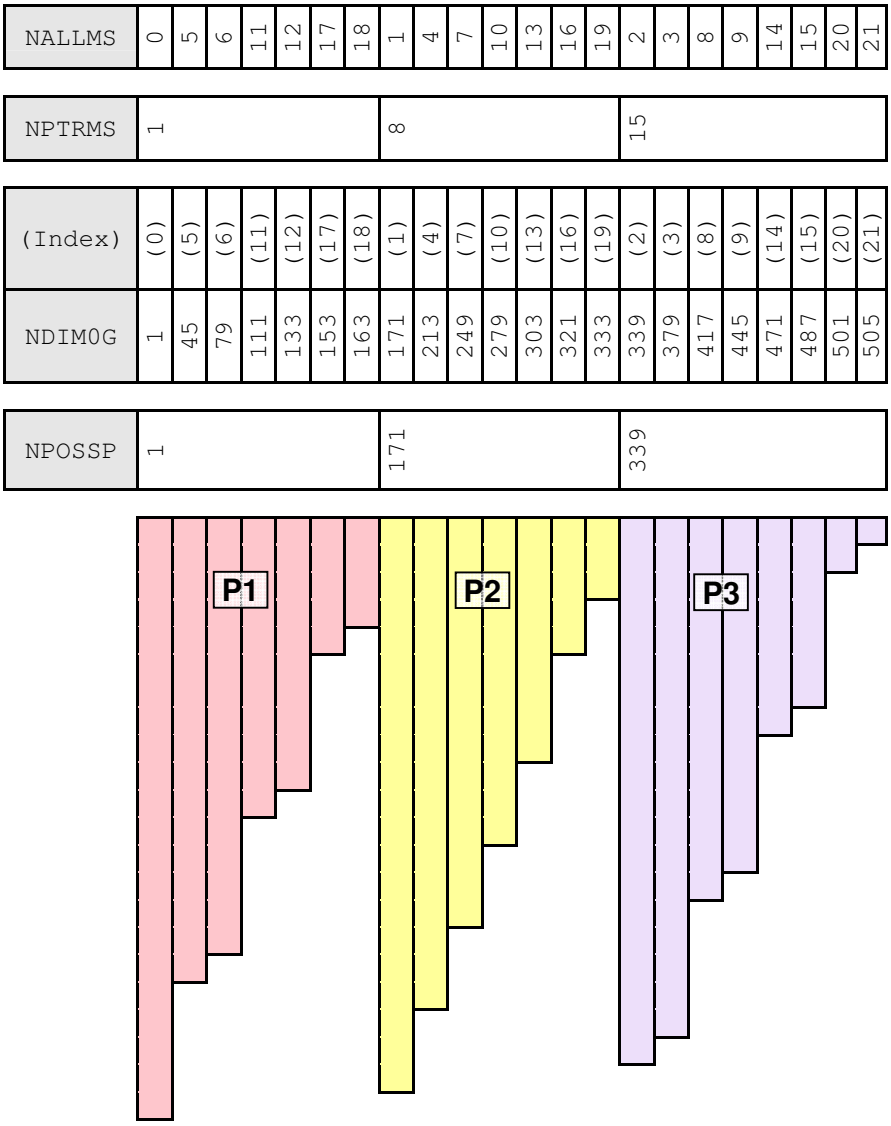

Figure 2.15 *Global representation of T21 spectral triangle.*

2.5 SEMI IMPLICIT SPECTRAL CALCULATIONS

The semi implicit spectral calculation have only vertical dependencies so spectral coefficient columns can be distributed without constraints amongst the NPRTRN(=NPRTRV) processors, as is shown in Figure 2.16. Unlike the other transforms and transposes we have already discussed, the transpose required for the semi implicit spectral calculations is not carried out within the TRANS package, but is coded directly in IFS.

To achieve good load balance, spectral waves are usually cut in the middle (as shown in Figure 2.16). However, for some configurations (where LIMPF=.TRUE.), there are dependencies between the total wave number n within a zonal wave number m, and for these cases splitting in the middle of a wavenumber is not possible, which restricts the load-balanced parallelism to one half of the spectral truncation.

The variables used to describe the decomposition of the semi implicit spectral calculations are shown in Table 2.7.

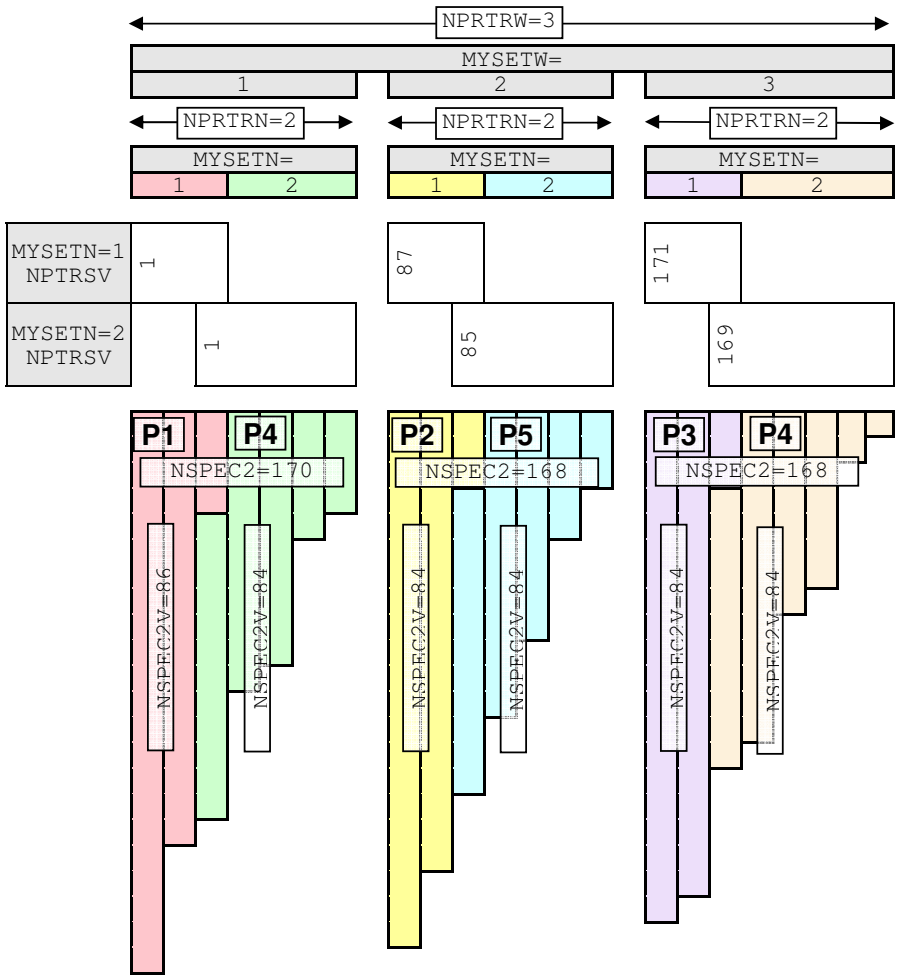

Figure 2.16 *Decomposition of spectral data for the semi implicit calculations.*

Table 2.7 *Variables describing the decomposition used for the semi implicit spectral calculations.*

Variable	Array dimensions and description
MYSETN	Scalar Which "N" set (spectral wave coefficients) this processor is in. (1..NPRTRN)
NPTRSV	(1:NPTRW+1 Pointer to first spectral wave column to be handled by each "W" set.
NSPEC2	Scalar Total number of complex spectral coefficients on this "W" set.
NSPEC2V	Scalar Number of complex spectral coefficients on this processor.
NVALUE	(1:NSPEC2 n (total wave number) value for a given NSPEC2 coefficient on this processor.

Appendix A
Structure, data flow and standards

Table of contents

A.1 COMMAND LINE OPTIONS

The primary way to control options within IFS is the namelist input file. However, since there are a very large number of options, it is convenient to be able to specify certain standard configurations in a simple way by supplying a small number of UNIX style flags on the command line. Any configuration supplied via the command line will automatically override equivalent namelist supplied configuration. The available flags are shown in Table A.1.

A.2 CDCONF SETTINGS

Within subroutines `STEPO`, `STEPOAD` and `STEPOTL`, extensive use is made of a character string `CDCONF` for controlling the logic flow of transformations between spectral and grid-point space. `CDCONF` is a 9 character variable where each character controls a specific sub-area within IFS, as indicated in Table A.2.

Table A.1 *Command line options.*

Option	Namelist Variable	Description
-c	NCONF	Job Configuration (see Table A.11 on page 47)
-v	LECMWF	Model version: ecmwf or meteo
-e	CNMEXP	Experiment identifier (max 4 characters)
-t	TSTEP	Time step (seconds) Default set according to model resolution and advection scheme
-f	NSTOP	Forecast length: dxxxxx - run for xxxxx days hxxxxx - run for xxxxx hours txxxxx - run for xxxxx timesteps
-a	LSLAG / LSVENIN	Advection scheme: eul: Eulerian sli: Interpolating semi-Lagrangian slni: Non-interpolating in the vertical, semi-Lagrangian
-m	LUELAM	Model type: arpifs: ARPEGE/IFS aladin: ALADIN

Table A.2 *CDCONF settings.*

Character position	Sub-area *(Description)*	Value	Description
1.....	IOPACK *(IO handling)*	A	Write out model level post-processed data
		B	Retrieve trajectory information
		C	Write out pressure level post-processed data
		F	"A"+"R"
		I	Store/write out increment (incremental 3D/4D Var)
		J	Read increment (incremental 3D/4D Var)
		L	Lagged model level post-processing
		R	Read restart file
		T	Store trajectory
		V	Read in the inputs for sensitivity job
		E,M,U, Y,Z	FullPos

continued on next page . . .

Table A.2 *CDCONF settings (continued . . .).*

Character position	Sub-area *(Description)*	Value	Description
.2.....	LTINV *(Inverse Legendre Transform)*	A	SPA3 Derivatives & Fourier Data T0
		B	SPA5 Derivatives & Fourier Data T0
		C	SPA7 Derivatives & Fourier Data T0
		D	SPA3 Derivatives & Fourier Data T5
		E	SPA5 Derivatives & Fourier Data T5
		F	SPA7 Derivatives & Fourier Data T5
		G	No SPA3 Derivatives & Fourier Data T0
		H	No SPA5 Derivatives & Fourier Data T0
		I	No SPA7 Derivatives & Fourier Data T0
		J	No SPA3 Derivatives & Fourier Data T5
		K	No SPA5 Derivatives & Fourier Data T5
		L	No SPA7 Derivatives & Fourier Data T5
		P	FullPos
..3.....	FTINV *(Inverse Fourier Transform)*	A	T0 Derivatives & Fourier Data T0
		B	No T0 Derivatives & Fourier Data T0
		C	T5 Derivatives & Fourier Data T5
		D	No T5 Derivatives & Fourier Data T5
		I	No T1 Derivatives & Fourier Data T0
		P	FullPos
...4.....	CPG *(Grid point computations)*	A	"Normal" timestep
		B	Additional computations for post-processing surface fields
		E,F	Adiabatic NNMI iteration
		M,X	NNMI iteration or initial fluxes

continued on next page . . .

Table A.2 *CDCONF settings (continued . . .).*

Character position	Sub-area *(Description)*	Value	Description
....5....	POS *(Post processing)*	`A`	Pressure level post-processing
		`H`	Height (above orography) level post-processing
		`T`	Potential temperature level post-processing
		`V`	Potential vorticity level post-processing
		`M`	Model level post-processing
		`S`	Eta level post-processing
		`L`	End of vertical post-processing
.....6...	OBS *(Comparison with observations)*	`A`	Add squares of grid-point values (analyses error calculation)
		`B`	Subtract squares of grid-point values (analyses error calculation)
		`C,V`	Computation of observation equivalents (GOM arrays)
		`G,W`	Normalisation by standard deviations of background error
		`I`	Grid-point calculations for CANARI
		`X`	Multiplication by standard deviations of background error (inverse of `G`,`W`)
		`Y`	Modifies the background errors to have a prescribed global mean profile and (optionally) to be separable
		`Z`	Generate background errors of humidity
......7..	FTDIR *(Direct Fourier Transform)*	`A`	Standard transform
		`B`	Pressure level post-processing
		`C`	Model level post-processing
		`P`	FullPos
.......8.	LTDIR *(Direct Legendre Transform)*	`A`	Standard transform
		`B`	Pressure level post-processing
		`C`	Model level post-processing
		`P`	FullPos
		`T`	Tendencies (result in SPT arrays)
		`G`	Similar to "`A`", but goes from vorticity and divergence in Fourier space to spectral space, instead of starting from the wind components

continued on next page . . .

Table A.2 *CDCONF settings (continued . . .).*

Character position	**Sub-area *(Description)***	**Value**	**Description**
.....9	SPC *(Spectral space computations)*	`A`	Semi-implicit and horizontal diffusion
		`F`	Filtering of spectral fields
		`I`	Only semi-implicit (for NMI)
		`P`	Filtering of FullPos fields

A.3 CONTROL NAMELISTS

Namelist input is provided in a text file `fort.4`. Within this file, the namelists can be in any order. The file is read multiple times to extract the namelist parameters in the order that the IFS code reads them. All namelists must always be present in `fort.4`. However, it is permissible for a namelist to be empty (see NAME2 in the example below). The general format is:

```
&NAME1
variable_name=value,
..
/
&NAME2
/
```

A.3.1 Index of namelists

Tables A.3 to A.10 index the major namelists used by IFS. For further details of the contents of any of the namelists described here, look in the following files in the IFS source repository:

`namelist/na*<namelist_name>.h`: Definition of all the variables within a namelist.

`module/yom<namelist_name>.h`: FORTRAN module containing all the namelist variables associated with a namelist, with a description of the purpose/usage of each variable.

Table A.3 *Top Level Control Namelists.*

Namelist	Description	Read in Subroutine
NAMCT0	Control parameters, constant during model run	SUCT0/SUMPINI
NAMCT1	Overriding control switches	SU1YOM
NAMDIF	Difference two model states	SUDIF
NAMGFL	GFL field descriptors	SUDIM1
NAMMCC	Climate version	SUMCC/SUDIM
NAMRCF	Restart control file	RERESF/WRRESF
NAMRES	Restart time parameters	SURES
NAMTLEVOL	Tangent linear perturbation evolution switches	SUPHLI

Table A.4 *Physics / Radiation Namelists.*

Namelist	Description	Read in Subroutine
NAEPHY	ECMWF Physics	SUOPHY
NAERAD	ECMWF Radiation	SUECRAD
NAMCUMFS	Simplified convection scheme	SUCUMF
NAMDPHY	Physics dimension	SUDIM
NAMPHY	ARPEGE atmospheric physical parameters	SUOPHY
NAMPHY0	ARPEGE atmospheric physical parameters	SUPHY0
NAMPHY1	ARPEGE ground physics parameters	SUPHY1
NAMPHY2	ARPEGE vertical physics definition	SUPHY2
NAMPHY3	ARPEGE radiation physical constants	SUPHY3
NAMPHYDS	Physics fields setup	SUPHYDS
NAMRAD15	ARPEGE climate version of ECMWF radiation	SUECRAD15
NAMRCOEF	Radiation coefficients control	SUOPHY
NAMSIMPH1	ARPEGE linear physics parameterisation	SUOPHY
NAMSTOPH	Parameterisation top limits / mesospheric drag parameters	SURAND1
NAMTOPH	Mesospheric drag parameterisation (ARPEGE)	SUTOPH
NAMTRAJP	ECMWF linear physics	SUOPHY
NAMVDOZ	ARPEGE physics	SUPHY1
NAPH1C	Switch for simple physics	SUOPHY

Table A.5 *Dynamics / Numerics / Grids Namelists.*

Namelist	Description	Read in Subroutine
NAMCLTC	Dimensions of the input grid for SST NESDIS analysis	INCLITC
NAMDIM	Dimension / truncation	SUDIM
NAMDYN	Dynamics and hyperdiffusion	SUDYN
NAMDYNA	Dynamics	SUDYNA
NAMGEM	Transformed sphere (geometry / coordinate definition)	SUGEM1A/INCLIB
NAMRGRI	Reduced grid description	SURGRI
NAMSWE	Shallow water configuration	SUSPECB
NAMVV1	Vertical co-ordinate descriptor	SUVERT

Table A.6 *Assimilation / Initialisation / Observation Namelists.*

Namelist	Description	Read in Subroutine
NAMANCS	Analysis constants	SUEDFI
NAMDFI	Digital filtering control	SUEDFI
NAMDMSP	Satellite data descriptor	GETSATID
NAMGMS	Satellite data descriptor	GETSATID
NAMGOES	Satellite data descriptor	GETSATID
NAMHCP	Hours of synoptic reference trajectory corrections	SUHCP
NAMINI	Overriding switches for initialization	SUEINI
NAMJG	Assimilation, first guess constraint	SUJB
NAMJO	Jo control	DEFRUN
NAMLCZ	Lanczos eigensystem	SULCZ
NAMMETEOSAT	Satellite data descriptor	GETSATID
NAMMKODB	Make ODB run parameters	DEFRUN/OBADAT
NAMMODERR	Model error coefficients	SUDIM1
NAMMTS	TOVS radiation	SUMTS
NAMNASA	NASA satellite IDs	GETSATID
NAMNMI	Normal mode initialisation	SUNMI
NAMNN	Neural network bias correction	SUNNE
NAMNUD	Nudging	SUNUD
NAMOBS	Observation control	DEFRUN
NAMRINC	Incremental Variational description	SURINC
NAMSCC	Observation screening control	DEFRUN
NAMSENS	Sensitivity job	SUVAR
NAMSKF	Simplified Kalman Filter	SUSKF
NAMSSMI	SSMI Parameters	INISSMIP
NAMTESTVAR	VAR test configuration	TESTVAR
NAMTOVS	Satellite data descriptor	GETSATID
NAMVAR	Variational assimilation	SUVAR
NAMVARBC	Variational bias correction parameters	SUVARBC
NAMVFP	Variable LARCHFP	SUVAR
NAMVRT1	Switches for variational assimilation	SUVAR

Table A.7 *Diagnostic / Post-Processing Namelists.*

Namelist	Description	Read in Subroutine
NAMAFN	FullPos	SUAFN
NAMCAPE	FullPos CAPE calculation	SUCAPE
NAMCFU	Flux accumulation control	SUCFU
NAMCHET	Diagnostics on physical tendencies	SUCHET
NAMCHK	Grid -point evolution diagnostics	SUECHK
NAMDDH	Diagnostic (horizontal domain)	SUNDDH
NAMFPC	FullPos	SUFPC
NAMFPD	FullPos	SUFPD
NAMFPDYH	FullPos	SUFPDYN
NAMFPDYP	FullPos	SUFPDYN
NAMFPDYS	FullPos	SUFPDYN
NAMFPDYT	FullPos	SUFPDYN
NAMFPDYV	FullPos	SUFPDYN
NAMFPF	FullPos	SUFPF
NAMFPG	FullPos	SUFPG1
NAMFPIOS	FullPos	SUFPIOS
NAMFPPHY	FullPos	SUFPPHY
NAMFPSC2	FullPos	SUFPSC2
NAMPPC	Post-processing control	SUPP
NAMSCM	Single Column Model profile extraction	SUSCM
NAMSTA	Temperature extrapolation configuration	SUSTA
NAMXFU	Instantaneous flux control	SUXFU

Table A.8 *I/O Namelists.*

Namelist	Description	Read in Subroutine
NAMFA	GRIB packing options	SUFA
NAMGRIB	GRIB coding descriptor	SUGRIB
NAMIOMI	Minimisation I/O scheme	SUIOS
NAMIOS	I/O control	SUIOS
NAMOPH	Permanent file information	SUOPH
NAMVWRK	I/O scheme for trajectory	SUVWRK

Table A.9 *Computational Namelists.*

Namelist	Description	Read in Subroutine
NAM_DISTRIBUTED_VECTORS	Initialize chunksize for distributed vectors	SUMPINI
NAMPAR0	Parallel version control	SUMPINI
NAMPAR1	Parallel version control	SUMPO

Table A.10 *Other Namelists.*

Namelist	Description	Read in Subroutine
NACOBS	CANARI	DEFRUN
NACTAN	Analysis area for CANARI	DEFRUN
NACTEX	CANARI	CANALI
NACVEG	CANARI	CANALI
NAIMPO	CANARI	CANALI
NALORI	CANARI	CANALI
NAM_CANAPE	CANARICANAPE parameters	CANALI
NAMCLA	Climatological constants	INCLIO
NAMCLI	Climatological constants	INCLIO
NAMCOK	CANARI	CANALI
NAMMCUF	Digital filter specification	SUMCUF
NAMPONG	Vertical plane sponge configuration	SUPONG
NAMPRE	CANARI	CANAMI
NAMRIP	Real time parameters	SURIP
NAMTRANS	Transform configuration	SUTRANS

A.4 NCONF: IFS CONFIGURATION PARAMETER

The NCONF parameter is supplied by the namelist namct0 (see Table A.3 on page 42), or this value can be overridden on the command line (using the "-n" option, see Table A.1 on page 38). NCONF controls the function of any single execution of the IFS. Table A.11 describes the valid values for NCONF. An "[O]" or an "[R]" in the description indicates the configuration is routinely used in an **O**perational or **R**esearch context respectively.

Table A.11 *NCONF:* IFS *Configuration Parameter.*

Configuration (NCONF range)	Control routine	NCONF value	Model Description
Integration (0-99)	CNT1	1	[O] 3D primative equation (P.E.) model
		2	[O] 3D P.E. model and comparison with observations (LOBSC1=.TRUE.)
Variational Analysis (100-199)	CVA1	131	[O] Incremental 3D/4D-Var
2D Integration (200-299)	CNT1	201	[R] Shallow water model
		202	Vorticity equation model
		203	Linear gravity wave model
Test of the adjoint (400-499)	CAD1	401	[R] Test of adjoint with 3D P.E. model
		421	[R] Test of adjoint with shallow water model
Test of the tangent linear model (500-599)	CTL1	501	[R] Test of tangent linear with 3D P.E. model
		521	[R] Test of tangent linear with shallow water model
Eigenvalue / vector solvers for unstable model (600-699)	CUN1	601	[O] Eigenvalue/vector solver (singular vector comp.)
CANARI Optimal interpolation (700-799)	CAN1	701	Optimal interpolation with CANARI
Sensitivity (800-899)	CGR1	801	[O] Sensitivity with 3D P.E. model
		821	Sensitivity with shallow water model
Preparation of Initial Conditions / Interpolations (900-999) (MÉTÉO-FRANCE *only*)	CPREP1	901	Converts GRIB file to FA file
	CPREP5	903	Convert from MARS (un-gribbed) to FA (unrotated)
	CPREP1	911	Converts GRIB file to FA file
	CPREP1	912	Converts GRIB file to FA file
	INCLIO	923	Initialisation of climatological fields
		926	Change of geometry
	INCLITC	931	NESDIS sea surface temperature
	CSEAICE	932	Compute Sea ice concentration field
	CORMASS	940	Compute mass correction
	CPREP2	951	Difference between two model states
	CPREP3	952	Compute wind and grid-point fields
	CPREPAD	953	Compute grid-point gradient fields

A.5 INITIAL DATA

The starting conditions are supplied to the IFS in a number of files which follow a specific naming convention. The file names used and their contents vary according to the model configuration being run. In the following descriptions an *experiment identifier* ***xxid*** consisting of any four alphanumeric characters is used. Note that the ARPEGE version of the IFS (LECMWF=.FALSE. in namelist NAMCTO) uses different file names and file formats and is not documented here.

The initial fields are supplied in GRIB format and contained in the files described below.

ICMSH*xxid*INIT Contains upper-air spectral format fields on model levels

Individual fields are selected based on the GFL *attribute*, particularly the LSP and LGP *attributes* which specifies if a given field is in grid point or spectral space (see Table A.16 on page 53).

These fields are read from the file in the routine SUSPECG.

Table A.12 *ICMSHxxidINIT: Upper-air spectral format fields.*

ECMWF GRIB code	IFS variable (GRIB name)	Description	Levels
129	NGRB<Z>	Geopotential	1
152	NGRB<LNSP>	Log surface pressure	1
130	NGRB<T>	Temperature	NFLEV
133	NGRB<Q>	Specific humidity	NFLEV
138	NGRB<VO>	Vorticity (relative)	NFLEV
155	NGRB<D>	Divergence	NFLEV
203	NGRB<O3>	Ozone	NFLEV

ICMGG*xxid*INIT Contains surface fields on the model Gaussian grid.

These fields are read from the file in the routine SUGRIDG.

Table A.13 *ICMGGxxidINIT: Surface fields on model Gaussian grid.*

ECMWF GRIB code	IFS variable (GRIB name)	Description
27	NGRB<CVL>	Low vegetation cover
28	NGRB<CVH>	High vegetation cover
29	NGRB<TVL>	Type of low vegetation
30	NGRB<TVH>	Type of high vegetation
31	NGRB<CI>	Sea-ice cover
32	NGRB<ASN>	Snow albedo
33	NGRB<RSN>	Snow density
34	NGRB<SSTK>	Sea surface temperature
35	NGRB<ISTL1>	Ice surface temperature: Layer 1
36	NGRB<ISTL2>	Ice surface temperature: Layer 2
37	NGRB<ISTL3>	Ice surface temperature: Layer 3
38	NGRB<ISTL4>	Ice surface temperature: Layer 4
39	NGRB<SWVL1>	Volumetric soil water: Layer 1
40	NGRB<SWVL2>	Volumetric soil water: Layer 2
41	NGRB<SWVL3>	Volumetric soil water: Layer 3
42	NGRB<SWVL4>	Volumetric soil water: Layer 4
129	NGRB<Z>	Geopotential (at the surface orography) *(If not provided in a spectral field)*

continued on next page . . .

Table A.13 *ICMGGxxidINIT: Surface fields on model Gaussian grid. (continued . . .)*

ECMWF GRIB code	IFS variable (GRIB name)	Description
139	NGRB<STL1>	Soil temperature: Layer 1
141	NGRB<SD>	Snow depth
148	NGRB<CHNK>	Charnock parameter *(Coupled wave model only: LWCOU and LWCOU2W)*
159	NGRB<NGRBBLH>	Boundary layer height
160	NGRB<SDOR>	Standard deviation of orography
161	NGRB<ISOR>	Anisotropy of sub-gridscale orography
162	NGRB<ANOR>	Angle of sub-gridscale orography
163	NGRB<SLOR>	Slope of sub-gridscale orography
170	NGRB<STL2>	Soil temperature: Layer 2
172	NGRB<LSM>	Land-sea mask
173	NGRB<SR>	Surface roughness
174	NGRB<AL>	Albedo
183	NGRB<STL3>	Soil temperature: Layer 3
198	NGRB<SRC>	Skin reservoir content
234	NGRB<LSRH>	Logarithm of surface roughness length for heat
235	NGRB<SKT>	Skin temperature
236	NGRB<STL4>	Soil temperature: Layer 4
238	NGRB<TSN>	Temperature of snow layer

`ICMGGxxidINIUA` Contains upper air fields in grid point space.

All fields in this file have NFLEV levels.

Table A.14 shows only the "guaranteed" fields. The GFL fields are also included in this file, and are read in as directed by the GFL *attributes*. For further details, see the routine SUGRIDUG where this file is read in.

Table A.14 *ICMGGxxidINIUA: Upper air fields in grid point space.*

ECMWF GRIB code	IFS variable (GRIB name)	Description
246	NGRB<CLWC>	Cloud liquid water content
247	NGRB<CIWC>	Cloud ice water content
248	NGRB<CC>	Cloud cover

`ICMCLxxidINIT` Contains climate forcing surface fields on the model Gaussian grid.

These fields are used in "perfect surface" long (climate) integrations. In such integrations the surface conditions subject to seasonal variations (which in a normal forecast integration would be defined by the data assimilation and kept constant during the forecast) are changed regularly during the integration, based on the values contained in the file.

Note that the fields in the file must follow a certain pattern, otherwise the model will fail in the first time step. The fields must be in ascending time order, and the time spanned by them should be large enough to cover the model integration period. Furthermore, they should come at regular intervals, either every nth day (LMCCIEC[1] =.FALSE., or every month (LMCCIEC=.TRUE.).

[1] LMCCIEC, defined in namelist NAMMCC (see Table A.3 on page 42) is set to .TRUE. (the default value) if the climate fields are to be interpolated in time.

Routine SUMCC initialises switches for the climate configuration, and selects which fields will be required, storing the required GRIB codes in the array NCLIGC (in module YOMMCC). Routine UPDCLIE is called throughout the climate integration, and reads in the required fields from the file, and uses them to update the model fields. Alternatively, if the OASIS coupler is used, then the SST and sea ice data are obtained from the OASIS coupler rather than the file.

Table A.15 *ICMCLxxidINIT: Climate surface fields on model Gaussian grid.*

ECMWF GRIB code	GRIB name	Description
31	CI	Sea-ice fraction
139	STL1	Soil temperature: Layer 1
174	AL	Albedo

A.6 GMV AND GFL STRUCTURES IMPLEMENTATION AND USAGE

A.6.1 GMV structure

(*a*) *GMV code implementation*

There are three FORTRAN modules used by the GMV implementation:

YOMGV Contains the main grid-point GMV arrays, which are all allocatable arrays with one of two different layouts:

- Multilevel arrays, dimensioned: (NPROMA[2], NFLEVG[3], nflds[4], NGPBLKS[5]):
  GMV : Multilevel fields at t and $t - dt$
  GMVT1 : Multilevel fields at $t + dt$
  GMV5 : Multilevel fields trajectory
  GMV_DEPART : Multilevel fields departure (for 3D FGAT)
  YT0, YT9, YT1, YPH9, YT5, YAUX : "pointers" to fields
- Single level arrays, dimensioned: (NPROMA, nflds, NGPBLKS) [all single level variable names end in "S"]:
  GMVS : Single level fields at t and $t - dt$
  GMVT1S : Single level fields at $t + dt$
  GMV5S : Single level fields trajectory
  GMVS_DEPART : Single level fields departure (for 3D FGAT)

TYPE_GMVS Contains the type description of the user defined types used to address the GMV arrays (YT0, YT9, YT1, YPH9, YT5 and YAUX).

GMV_SUBS Contains subroutines used for setting up the GMV structure.

(*b*) *GMV usage*

The addressing of individual GMV fields is done using the user-defined types YT0, YT9, YT1, YPH9 and YT5 for the different time levels. The following code fragment taken from the time filtering for the Eulerian model (GPTF1) illustrates its usage:

```
PGMV(JL,JK,YT9%MU)    = REPS1*PGMV(JL,JK,YT9%MU)    + &
                      & ZREST*PGMV(JL,JK,YT0%MU)
PGMV(JL,JK,YT9%MV)    = REPS1*PGMV(JL,JK,YT9%MV)    + &
                      & ZREST*PGMV(JL,JK,YT0%MV)
PGMV(JL,JK,YT9%MDIV)  = REPS1*PGMV(JL,JK,YT9%MDIV)  + &
                      & ZREST*PGMV(JL,JK,YT0%MDIV)
```

The "pointers" MU, MV and MDIV point to u, v and divergence. The user-defined types YT0 and YT9 indicate the time levels t and $t - dt$ respectively. As this code is taken from a subroutine where a specific NPROMA block of grid points has been passed down, the forth dimension of GMV (the block number) is absent.

A.6.2 GFL structure

(*a*) *GMV code implementation*

There are three modules used in the GFL implementation:

[2] NPROMA: Size of a "computational block" (see Chapter 2 for more details).
[3] NFLEVG: Number of vertical levels.
[4] nflds: Total number of fields stored within this array.
[5] NGPBLKS: Number of NPROMA blocks each level is split in to.

YOMGFL Contains the main grid-point GFL arrays, which are all allocatable arrays with the same layout: (NPROMA , NFLEVG , nflds , NGPBLKS)
where only nflds varies from array to array. The spectral counterpart to the GFL grid-point array, SPGFL can be found in module YOMSP.

GFL : GFL array for t and $t - dt$

GFLT1 : GFL array for $t + dt$

GFLSLP : GFL array used by semi-Lagrangian physics

GFL5 : GFL array for trajectory

GFL_DEPART : GFL array for departures (3D FGAT)

TYPE_GFLS Contains the type definitions for structures holding the GFL attributes. There are three type definitions:

TYPE_GFLD : Overall descriptor, dimensions etc.

TYPE_GFL_COMP : Individual field attributes

TYPE_GFL_NAML : Individual field attributes for NAMELIST input

GFL_SUBS Contains the subroutines for setting up the GFL structure

YOM_YGFL Contains the structures holding the GFL attributes. There is one, YGFL of type TYPE_GFLD, and a number of structures of type TYPE_GFL_COMP.

YGFLC : An array containing the descriptors of all GFL components. All other individual component descriptors are pointers into this array

YQ : Specific humidity - q

YI : Ice water - q^i

YL : Liquid water - q^l

YA : Cloud fraction - a

YO3 : Ozone - O_3

YEXT(:) : Extra variables

(*b*) *GFL usage: attributes and pointers*

One of the main concepts introduced with the GFL structure is the use of *attributes* to govern the behaviour of the individual components contained within it. The intention is that the individual components of GFL (i.e. ozone) should only be referred to when absolutely necessary, e.g. when calling an ozone chemistry routine. In all other instances the following approach should be followed: loop over all fields in the GFL structure and perform the action defined by the setting of the appropriate GFL attribute. A typical example is shown below, taken from the Eulerian dynamics:

```
DO JGFL=1,YGFL%NUMFLDS  ! All fields in the GFL
 IF (YGFKC(JGFL)%LCDERS .AND.  &  ! horionztal derivs
&    YGFKC(JGFL)%LADV) THEN       ! advected field
  DO JLEV=1,NFLEVG-1 ! Vertical levels
   DO JROF=KSTART,KPROF ! Horizontal dimension
    ZDT=PDT*PVCASRSF(JROF,JLEV)
    PGFLT1(JROF,JLEV,YGFLC(JGFL)%MP1) = &
   & PGFLT1(JROF,JLEV,YGFLC(JGFL)%MP1)-ZDT* &
   & (PGFL(JROF,JLEV,YGFLC(JGFL)%MPL)*PUT0(JROF,JLEV) &
   & +PGFL(JROF,JLEV,YGFLC(JGFL)%MPM)*PVT0(JROF,JLEV))
   ENDDO
  ENDDO
 ENDIF
ENDDO
```

Here, the two attributes LCDERS (field has horizontal derivatives) and LADV (field to be advected) are tested in order to decide whether to horizontally advect the field or not. The advantage of this approach is twofold; when a new component is introduced the routine in question does not have to be modified and the coding becomes more compact.

The above example also shows the use of the GFL field "pointers" where the following pointers have been used:

YGFLC(JGFL)%MP1 : Points to the location of field JGFL in the $t + dt$ GFL array (PGFLT1)

YGFLC(JGFL)%MPL : Points to the location of the zonal derivative of field JGFL in the main GFL array (PGFL)

YGFLC(JGFL)%MPM : Points to the location of the meridional derivative of field JGFL in the main GFL array (PGFL)

The use of these pointers is compulsory as the layout of the different GFL arrays is different and often contains more fields than there are components in the GFL structure.

The following tables show the general attributes (Table A.16) and pointers (Table A.17). When it is realised that the existing attributes and/or pointers are not sufficient for a piece of code it is important that a new attribute/pointer is added rather than relying on some other ad. hoc. switch, or writing code for a specific component of GFL, when (theoretically) the same code could apply to other GFL components.

Table A.16 *GFL attributes.*

Attribute	Description
CNAME	ARPEGE field name
CSLINT	Semi-Lagrangian interpolation "type"
IGRBCODE	Gribcode of the field
LACTIVE	True if field is in use
LADJUST0	True if field is thermodynamically adjusted at t *[LAM specific (AROME/ALADIN)]*
LADJUST1	True if field is thermodynamically adjusted at $t + dt$ *[LAM specific (AROME/ALADIN)]*
LADV	True if field is to be advected
LBIPER	True if the field must be biperiodised inside the transforms *[LAM specific (AROME/ALADIN)]*
LCDERS	True if derivatives are required
LCOUPLING	True if field is to be coupled by Davies relaxation *[LAM specific (AROME/ALADIN)]*
LGP	True if field is a grid-point field
LGPINGP	True if grid-point field input as grid-point
LREQIN	True if field required in input
LSP	True if field is a spectral field
LSLP	True if field has S.L. physics representation
LT1	True if field has $t + dt$ representation
LT5	True if field forms part of trajectory
LT9	True if field has $t - dt$ representation

(*c*) *Adding a new attribute*

(i) Add the attribute to the type definition (TYPE_GFL_COMP in module TYPE_GFLS).

Table A.17 *GFL pointers.*

Pointer	Description
MP	Basic field
MPL	Zonal derivative
MPM	Meridional derivative
MPSLP	Semi-Lagrangian physics
MPSP	Spectral space
MP1	Field at $t + dt$
MP5	Trajectory
MP5L	Zonal derivative - trajectory
MP5M	Meridional derivative - trajectory
MP9	Field at $t - dt$
MP_SPL	Spline interpolation
MP_SL1	Field in SLBUF1

(ii) Update one of the setup routines (DEFINE_GFL_COMP or SET_GFL_ATTR in module GFL_SUBS). It is preferable to use routine SET_GFL_ATTR unless the attribute has to be known very early in the setup stage.
(iii) Use the attribute.

(*d*) *Adding a new component*

(i) Add the new component in module YOM_YGFL
(ii) Decide the required attributes and setup the component by adding calls to routines DEFINE_GFL_COMP and SET_GFL_ATTR for the new component.

(*e*) *Time stepping*

The time stepping of the variables with a spectral representation (all GMV prognostic variables and optionally part of GFL) takes place implicitly during the spectral transforms. The input for the inverse transforms are the spectral arrays (SPA3 and SPA2) and the output is the t part of GMV and GFL. The $t + dt$ GMV and GFL arrays (GMV1, GMVT1S and GFLT1) are created in grid-point space and transformed back to spectral space (SPA3 and SPA2) by the direct transforms. The time stepping of the pure grid-point GFL variables takes place at the end of SCAN2MDM (a simple copy from GFLT1 to GFL).

Appendix B
Message Passing Library (MPL)

Table of contents

B.1 INTRODUCTION

In the past, it has proved very beneficial to have a subroutine layer between the application code and the message passing library calls themselves. Benefits include:

- Some details (e.g. error handling) can be hidden.
- Flexibility is enhanced since changes can be made (e.g. vendor specific code) without impacting the application code.

MPL supersedes the original MPE library developed for IFS use. It provides for greater flexibility and future enhancement. In particular, it provides support for several different flavours of MPI point-to-point message-passing techniques. This version supports:

- Blocking-standard.
- Blocking-buffered.
- Non-blocking-standard.

Several FORTRAN 90 language features are utilised:

- By using MODULEs, optional keyword parameters allow subroutine parameter lists to be considerably shortened without losing flexibility.
- Data is passed as a FORTRAN 90 object so that the type and length of the message content is deduced by the routine. The following data types are supported:

REAL*4 : 1 or 2 dimensional arrays

REAL*8 : 1 or 2 dimensional arrays

INTEGER [1] : Scalar

All routines which wish to call MPL routines must contain:

```
USE MPL_MODULE
```

This permits errors in the calling sequence to be identified at compile time.

The process numbering convention used in the application is assumed to begin with 1. MPI uses a numbering convention commencing with 0. Therefore, all MPL routines which refer to a process number subtract one from the user supplied value before passing to the relevant MPI routine.

In the following descriptions of MPL parameters, the interface description includes only the required parameters. Keywords are in `UPPER CASE`, user supplied values are in `lower case`.

The choice of keywords follows the source code naming conventions used within IFS for subroutine parameters (see Table 1.2 on page 8):

- INTEGERs commence with K
- REAL commence with P
- CHARACTER commence with CD
- LOGICAL commence with LD

[1] All references to type INTEGER in this appendix refer to the default INTEGER type for the particular platform being used.

B.2 MPL_ABORT

Aborts from a parallel environment with an (optional) message

Purpose

Called to terminate a parallel execution and print a suitable message.

Interface

```
CALL MPL_ABORT
```

Input Arguments

Required
None

Optional

CDMESSAGE
Character string to be printed

Output Arguments

Required
None

Optional
None

B.3 MPL_BARRIER

Barrier synchronisation

Purpose

Blocks the caller until all group members have called it

Interface

```
CALL MPL_BARRIER
```

Input Arguments

Required

None

Optional

KCOMM

Communicator number if different from MPI_COMM_WORLD or from that established as the default by an MPL communicator routine

CDSTRING

Character string for ABORT messages used when KERROR is not provided

Output Arguments

Required

None

Optional

KERROR

Return error code
If not supplied, MPL_BARRIER aborts when an error is detected.

B.4 MPL_BROADCAST

Message Broadcast

Purpose

Broadcasts a message from the process with rank KROOT to all processes in the group.
NOTE: Unlike MPI_BCAST, only the KROOT process sends the message.
Messages are sent assuming a process numbering convention of 1 to N unless an alternative has been supplied to MPL_BUFFER_METHOD (KPROCIDS=...)

Interface

```
CALL MPL_BROADCAST (buffer,KTAG=itag,KROOT=iroot)
```
Alternatives are: PBUF=buffer, or KBUF=ibuffer

Input Arguments

Required

PBUF
: Buffer containing message
(may be type (REAL*4), REAL*8 or INTEGER).

KTAG
: Message tag.

KROOT
: Root process number.

Optional

KCOMM
: Communicator number if different from MPI_COMM_WORLD or from that established as the default by an MPL communicator routine

CDSTRING
: Character string for ABORT messages used when KERROR is not provided

Output Arguments

Required

None

Optional

KERROR
: Return error code
If not supplied, MPL_BROADCAST aborts when an error is detected.

B.5 MPL_BUFFER_METHOD

Establish message passing default method

Purpose

Optional Routine
Override the message passing default method and allocate an attached buffer if required.

Interface

```
CALL MPL_BUFFER_METHOD (KMP_TYPE=itype)
```

Input Arguments

Required

KMP_TYPE

Buffering type, possible values (defined as parameters in MPL_DATA_MODULE) are :

- JP_BLOCKING_STANDARD (*default* for VPP platforms)
- JP_BLOCKING_BUFFERED (*default* for all other platforms)

Optional

KMBX_SIZE

Size (in bytes) of attached buffer
(*Only if* KMP_TYPE= JP_BLOCKING_BUFFERED)

KPROCIDS

Array of processor identifiers.
For use if the application uses a processor numbering convention different from 1 to N.

Output Arguments

Required

None

Optional

KERROR

Return error code
If not supplied, MPL_BUFFER_METHOD aborts when an error is detected.

B.6 MPL_COMM_CREATE

Create a new communicator

Purpose

Create a new communicator and set as default

Interface

```
CALL MPL_COMM_CREATE
```

DEFERRED IMPLEMENTATION

B.7 MPL_END

Terminate a parallel execution

Purpose

Cleans up all of the MPI state.
Subsequently, no other MPI routine can be called.
(MPL_END may be called more than once.)

Interface

```
CALL MPL_END
```

Input Arguments

Required
None

Optional
None

Output Arguments

Required
None

Optional

KERROR
Return error code
If not supplied, MPL_END aborts when an error is detected.

B.8 MPL_INIT

Initialises the Message passing environment

Purpose

Must be called before any other MPL routine.
(MPL_INIT may be called more than once.)

Interface

```
CALL MPL_INIT
```

Input Arguments

Required

None

Optional

KOUTPUT

Level of printing for MPL routines:

=0 : None

=1 : Intermediate (*Default*)

=2 : Full trace

KUNIT

FORTRAN unit to receive printed trace.

Output Arguments

Required

None

Optional

KERROR

Return error code
If not supplied, MPL_INIT aborts when an error is detected.

KPROCS

Number of processes which have been initialised in the MPI_COMM_WORLD communicator.

B.9 MPL_MESSAGE

Prints message

Purpose

Creates an ASCII message for printing and optionally aborts.
(Used from within other MPL routines.)

Interface

```
CALL MPL_MESSAGE(CDMESSAGE='.....')
```

Input Arguments

Required

CDMESSAGE
: Character string containing message

Optional

KERROR
: Error number

CDSTRING
: Optional additional message prepended to CDMESSAGE.

LDABORT
: Forces ABORT if .TRUE.

Output Arguments

Required

None

Optional

None

B.10 MPL_MYRANK

Find rank in current communicator

Purpose

Returns the rank of the calling process in the currently active communicator.

Interface

```
IRANK = MPL_MYRANK()
```

Input Arguments

Required
None

Optional
None

Output Arguments

Required
None

Optional
None

B.11 MPL_NPROC

Find number of processors in current communicator

Purpose

Returns the number of processes in the currently active communicator.

Interface

```
INUMP = MPL_NPROC()
```

Input Arguments

Required

KCOMM

Communicator number if different from MPI_COMM_WORLD.

Optional

None

Output Arguments

Required

None

Optional

None

B.12 MPL_PROBE

Check for incoming message.

Purpose

Look for existence of an incoming message.

Interface

```
CALL MPL_PROBE
```

Input Arguments

Required

None

Optional

KSOURCE

Rank of process sending the message.
(*Default* is MPI_ANY_SOURCE.)

KTAG

Tag of incoming message.
(*Default* is MPI_ANY_TAG.)

KCOMM

Communicator number.
(*Default* is MPI_COMM_WORLD.)

LDWAIT

C

ontrols the blocking behaviour:

=.TRUE. : Waits for a message to be available *(Default)*.

=.FALSE. : Return immediately and set LDFLAG to indicate if a message exists.

CDSTRING

Character string for ABORT messages used when KERROR is not provided.

Output Arguments

Required

None

Optional

KERROR

Return error code
If not supplied, MPL_PROBE aborts when an error is detected.

LDFLAG

Must be supplied if LDWAIT=.TRUE.
Returns .TRUE. if a message exists.

B.13 MPL_RECV

Receive a message

Purpose

Receive a message from a named source into a buffer.
The data may be:

- Scalar REAL or INTEGER.
- 1D Array of REAL*4, REAL*8 or INTEGER.
- 2D Array of REAL*4 or REAL*8.

Interface

```
CALL MPL_RECV(buffer)
```
Alternatively, PBUF=buffer or KBUF=ibuf.

Input Arguments

Required

PBUF
: Buffer to receive the message.
(Can be of type REAL*4, REAL*8 or INTEGER.)

Optional

KTAG
: Message tag.
(*Default* is MPI_ANY_TAG.)

KCOMM
: Communicator tag.
(*Default* is MPI_COMM_WORLD.)

KMP_TYPE
: Buffering type.
(*Default* is the value provided to MPL_BUFFER_METHOD.)

KSOURCE
: Rank of process sending the message.
(*Default* is MPI_ANY_SOURCE.)

CDSTRING
: Character string for ABORT used when KERROR is not provided.

Output Arguments

Required

None

Optional

KREQUEST
: Communication request identifier (required when buffering type is non-blocking).

KFROM
: Rank of process sending the message.

KRECVTAG
: Tag of received message.

KOUNT
: Number of items in received message.

KERROR
: Return error code
If not supplied, MPL_RECV aborts when an error is detected.

B.14 MPL_SEND

Send a message

Purpose

Send a message to a named source from a buffer.
The data may be:

- Scalar REAL or INTEGER.
- 1D Array of REAL*4, REAL*8 or INTEGER.
- 2D Array of REAL*4 or REAL*8.

Interface

```
CALL MPL_SEND(buffer,KTAG=itag,KDEST=iproc)
```
Alternatively, PBUF=buffer or KBUF=ibuffer.

Input Arguments

Required

PBUF
: Buffer containing message.
(Can be of type REAL*4, REAL*8 or INTEGER.)

KTAG
: Message tag.

KDEST
: Rank of process to receive the message.

Optional

KCOMM
: Communicator tag.
(*Default* is MPI_COMM_WORLD or the default established by an MPL communicator routine.)

KMP_TYPE
: Buffering type.
(*Default* is the value provided to MPL_BUFFER_METHOD.)

CDSTRING
: Character string for ABORT used when KERROR is not provided.

Output Arguments

Required

None

Optional

KREQUEST
: Communication request identifier (required when buffering type is non-blocking).

KERROR
: Return error code
If not supplied, MPL_SEND aborts when an error is detected.

B.15 MPL_WAIT

Waits for completion

Purpose

Returns control when the operation(s) identified by the request is completed.
Normally used in conjunction with non-blocking buffering type.

Interface

`CALL MPL_WAIT(buffer,KREQUEST=ireq)`
Alternatively, `PBUF=buffer` or `KBUF=buffer`.

Input Arguments

Required

PBUF

Array with same size and shape as buffer.
Used for MPL_SEND or MPL_RECV.

KREQUEST

Scalar or array containing communication request identifier(s) as provided by MPL_SEND or MPL_RECV.

Optional

CDSTRING

Character string for `ABORT` used when KERROR is not provided.

Output Arguments

Required

None

Optional

KOUNT

Number of items in received message.

KERROR

Return error code
If not supplied, MPL_WAIT aborts when an error is detected.

Appendix C
The TRANS package

Table of contents

C.1 INTRODUCTION

As from cycle 23r4 the spectral transforms have been broken out of the IFS to form a separate library (`libtrans`), the source residing in its own VOB (`trans`). There are several reasons for this change. One is to make the IFS more modular, the transforms form a non-scientific part that could easily be separated from the rest. Another reason is to make the efficient spectral transforms previously buried within the IFS usable by other codes.

The routines in the transform package are divided into two groups, externally callable routines and internal routines. The internal routines are all FORTRAN 90 module procedures and are not described in this documentation. The externally callable routines are not module procedures but an explicit interface block is needed in order to call them. This makes it possible to use FORTRAN 90 features like assumed shape arrays and optional arguments. The interface blocks can all be found in the `trans` VOB (in the interface directory). The assumed shape arrays are used to avoid having to pass dimensions and to ensure that arrays are dimensioned correctly. When calling a transform routine it always transforms the whole of the arrays passed, e.g.

```
CALL INV_TRANS(PSPSCALAR=SPEC,PGP=GP)
```

would transform all the fields of SPEC into GP. The spectral array passed into the transform routines must be of rank 2, the first dimension for the number of fields and the second for the spectral coefficients.

In the following description of the individual routines, where arguments are arrays, "`(:,...)`" is used to show the rank of the array. Following IFS coding norms argument names starting with "`K`" denotes integer arguments, starting letter "`P`" indicates real arguments and "`L`" logical arguments. Arguments names in italics indicate that the argument in question is only of interest in the case of using more than one processor. When the word "GLOBAL" is used in the following documentation it refers to viewing the data as a whole, not the part of it that is available on an individual processor in the distributed case.

Some examples are given after the description of the routines to demonstrate their typical usage.

C.2 SETUP_TRANS0

General setup routine for transform package.

Purpose

Resolution independent part of setup of transform package. Must be called before SETUP_TRANS.

Interface

```
CALL SETUP_TRANS0(...)
```

Input Arguments

Required

None

Optional

KOUT
: Unit number for listing output. (*Default : 6*)

KERR
: Unit number for error messages. (*Default : 0*)

KPRINTLEV
: Level of output to KOUT:
0 : No output
1 : Normal output
2 : Debug output
(*Default : 0*)

KMAX_RESOL
: Maximum number of different resolutions for this run. (*Default : 1*)

KPRGPNS
: Number of processors in N–S direction in grid-point space. (*Default : 1*)

KPRGPEW
: Number of processors in E–W direction in grid-point space. (*Default : 1*)

KPRTRW
: Number of processors in wave direction in spectral space. (*Default : 1*)

KCOMBFLEN
: NSize of communication buffer (in 8 byte words). (*Default : 1800000*)

LDIMP
: Use immediate message passing. (*Default : .FALSE.*)

LDIMP_NOOLAP
: Use immediate message passing with no overlap between communications and computations. (*Default : .FALSE.*)

LDMPOFF
: Switch off message passing. (*Default : .FALSE.*)

Output Arguments

Required

None

Optional

None

The total number of (MPI) processors has to be equal to KPRGPNS*KPRGPEW.

C.3 SETUP_TRANS

Setup transform package for specific resolution.

Purpose

Setup for making spectral transforms. Each call to this routine creates a new resolution up to a maximum of KMAX_RESOL as set up in SETUP_TRANSO. You need to call SETUP_TRANSO before this routine can be called. The optional parameter KRESOL used in subsequent calls to other transform routines refers to the nth defined resolution.

Interface

```
CALL SETUP_TRANS(...)
```

Input Arguments

Required

KSMAX
: Spectral truncation required.

KDGL
: Number of Gaussian latitudes.

Optional

KLOEN(:)
: Number of points on each Gaussian latitude. (*Default : 2*KDGL*)

LDSPLIT
: True if split latitudes in grid-point space. (*Default : .FALSE.*)

LDLINEAR_GRID
: True if linear grid. (*Default : .FALSE.*)

KAPSETS
: Number of apple sets in the distribution (*Default : 0*)

KTMAX
: Truncation order for tendencies (*Default : KSMAX*)

Output Arguments

Required

None

Optional

KRESOL
: The resolution identifier.

KSMAX, KDGL, KTMAX and KLOEN are GLOBAL variables describing the resolution in spectral and grid-point space.

C.4 DIR_TRANS

Direct spectral transform (from Gaussian grid to spectral space).

Purpose

Interface routine for the Direct Spectral Transform.

In the following description "NF_UV_G" is the GLOBAL number of u/v type fields and NF_SCALARS_G is the GLOBAL number of *scalar* valued fields. When the fields are not distributed over processors, NF_UV_G is given by the length of PSPVOR and PSPDIV and NF_SCALARS_G by the length of PSPSCALAR. If the fields are distributed over processors (the case where KPRTRW < KPRGPNS*KPRGPEW in SETUP_TRANSO), the arguments KVSETUV and KVSETSC describing the distribution have to be present and their respective lengths give NF_UV_G and/or NF_SCALARS_G.

There are two alternative ways of specify the grid point fields. The original way is to use the PGP array. The alternative way is to use a combination of the PGPUV, PGP3A, PGP3B and PGP2 arrays. The reason for introducing these alternative ways of calling DIR_TRANS is to avoid unnecessary copies where your data structures don't fit in to the "PSPVOR, PSPDIV, PSPSCALAR, PGP" layout. The use of any of these precludes the use of PGP and vice versa.

Interface

```
CALL DIR_TRANS(....)
```

Input Arguments

Required

PGP(:,:,:)

Grid point fields.
PGP must be dimensioned (KPROMA,NF_GP,NGPBLKS) where KPROMA is the blocking factor(see below), NF_GP the total number of fields in gridpoint space and NGPBLKS the number of KPROMA blocks. The default for KPROMA is the total number of gridpoints on a processor in which case NGPBLKS is 1.
The ordering of the output fields is as follows (all parts are optional depending on the input switches):

u : NF_UV_G fields (if PSVOR and PSPDIV present)

v : NF_UV_G fields (if PSVOR and PSPDIV present)

scalar **fields** : NF_SCALARS_G fields (if PSPSCALAR present)

or a combination of the following arrays:

PGPUV(:,:,:,:)

The "u-v" related grid-point variables in the order described for PGP. The second dimension of PGPUV should be the same as the GLOBAL first dimension of PSPVOR, PSPDIV (in the IFS this is the number of levels). PGPUV need to be dimensioned (NPROMA,ILEVS,IFLDS,NGPBLKS) where IFLDS is the number of "variables" (u,v).

PGP3A(:,:,:,:)

Grid-point array directly connected with PSPSC3A dimensioned (NPROMA,ILEVS,IFLDS,NGPBLKS) where IFLDS is the number of "variables" (the same as in PSPSC3A).

PGP3B(:,:,:,:)

Grid-point array directly connected with PSPSC3B dimensioned (NPROMA,ILEVS,IFLDS,NGPBLKS) where IFLDS is the number of "variables" (the same as in PSPSC3B).

PGP2(:,:,:)
Grid-point array directly connected with PSPSC2 dimensioned (NPROMA,IFLDS,NGPBLKS) where IFLDS is the number of "variables" (the same as in PSPSC2).

Optional

KRESOL
Resolution identifier which is to be used (*Default : First defined resolution*)

KPROMA
Requested blocking factor for gridpoint output. Used to divide the gridpoint array in chunks of a size suitable for further computations (to control memory usage, vectorization etc.)
(*Default : Number of gridpoints on processor (KGPTOT from TRANS_INQ)*)

KVSETUV(:)
Indicating which "field-set" in spectral space owns a *vor/div* field. Equivalent to NBSETLEV in the IFS. The length of KVSETUV should be the GLOBAL number of u/v fields which is the dimension of u and v related fields in grid-point space.

Either

KVESETSC(:)
Indicating which "field-set" in spectral space owns a *scalar* field. As for KVSETUV this argument is required if the total number of processors is greater than the number of processors used for distribution in spectral wave space.

Or a combination of *KVSETSC3A(:)*
As KVESETSC for PSPSC3A (distribution on first dimension).

KVSETSC3B(:)
As KVESETSC for PSPSC3C (distribution on first dimension).

KVSETSC2(:)
As KVESETSC for PSPSC2 (distribution on first dimension).

Output Arguments

Required Either PSPVOR and PSPDIV or PSPSCALAR (or as an alternative to PSPSCALAR a combination of PSPSC3A, PSPSC3B and PSPSC2) has to be present (see below).

Optional

PSPVOR(:,:)
Spectral *vorticity*.

PSPDIV(:,:)
Spectral *divergence*.

Either

PSPSCALAR(:,:)
Spectral *scalar* valued fields.

Or a combination of

PSPSC3A(:,:,:)
Alternative to use of PSPSCALAR, see PGP3A above.

PSPSC3B(:,:,:)
Alternative to use of PSPSCALAR, see PGP3B above.

PSPSC2(:,:)
Alternative to use of PSPSCALAR, see PGP2 above.

For PSPVOR, PSPDIV and PSPSCALAR the first dimension is the field dimension and the second is for the spectral coefficients. In the case of one processor the ordering of the spectral coefficients is the same as the one obtained when decoding/encoding a GRIB field using the GRIBEX routine.

C.5 DIR_TRANSAD

Adjoint of direct spectral transform.

See DIR_TRANS. Only differences are that PGP becomes an output argument and PSPVOR, PSPDIV and PSPSCALAR become input arguments.

C.6 INV_TRANS

Inverse spectral transform (spectral to gridpoint).

Purpose

Interface routine for the inverse spectral transform. Also for computing gridpoint u and v from *vorticity* and *divergence* and for computing N–S and E–W derivatives of fields.

In the following description "NF_UV_G" is the GLOBAL number of u/v type fields and NF_SCALARS_G is the GLOBAL number of *scalar* valued fields. When the fields are not distributed over processors, NF_UV_G is given by the length of PSPVOR and PSPDIV and NF_SCALARS_G by the length of PSPSCALAR. If the fields are distributed over processors (the case where KPRTRW < KPRGPNS*KPRGPEW in SETUP_TRANSO), the arguments KVSETUV and KVSETSC describing the distribution have to be present and their respective lengths give NF_UV_G and/or NF_SCALARS_G.

There are two alternative ways of specify the grid point fields. The original way is to use the PGP array. The alternative way is to use a combination of the PGPUV, PGP3A, PGP3B and PGP2 arrays. The reason for introducing these alternative ways of calling DIR_TRANS is to avoid unnecessary copies where your data structures don't fit in to the "PSPVOR, PSPDIV, PSPSCALAR, PGP" layout. The use of any of these precludes the use of PGP and vice versa.

Interface

```
CALL INV_TRANS(...)
```

Input Arguments

Required

None

Optional

PSPVOR(:,:)
: Spectral *vorticity*.

PSPDIV(:,:)
: Spectral *divergence*.

Either

PSPSCALAR(:,:)
: Spectral *scalar* valued fields.

Or a combination of

PSPSC3A(:,:,:)
: Alternative to use of PSPSCALAR, see PGP3A below.

PSPSC3B(:,:,:)
: Alternative to use of PSPSCALAR, see PGP3B below.

PSPSC2(:,:)
: Alternative to use of PSPSCALAR, see PGP2 below.

FSPGL_PROC
: External procedure to be executed in Fourier space before transposition.

LDSCDERS
: Indicating if derivatives of *scalar* variables are required. (*Default : .FALSE.*)

LDVORGP
: Indicating if grid-point *vorticity* is required. (*Default : .FALSE.*)

LDDIVGP
: Indicating if grid-point *divergence* is required. (*Default : .FALSE.*)

LDUVDER

Indicating if E–W derivatives of u and v are required. (*Default : .FALSE.*)

KPROMA

Required blocking factor for gridpoint output. Used to divide the gridpoint array in chunks of a size suitable for further computations (to control memory usage, vectorization etc.) *Default : Total number of gridpoints for one field.*

KVSETUV(:)

Indicating which "field-set" in spectral space owns a vor/div field. Equivalent to NBSETLEV in the IFS. The length of KVSETUV should be the GLOBAL number of u/v fields which is the dimension of u and v related fields in grid-point space.

Either

KVESETSC(:)

Indicating which "field-set" in spectral space owns a *scalar* field. As for KVSETUV this argument is required if the total number of processors is greater than the number of processors used for distribution in spectral wave space.

Or a combination of

KVSETSC3A(:)

As KVESETSC for PSPSC3A (distribution on first dimension).

KVSETSC3B(:)

As KVESETSC for PSPSC3C (distribution on first dimension).

KVSETSC2(:)

As KVESETSC for PSPSC2 (distribution on first dimension).

KRESOL

Resolution identifier which is required. (*Default : First defined resolution.*)

Output Arguments

Required

PGP(:,:,:)

Grid point fields.
PGP must be dimensioned (KPROMA,NF_GP,NGPBLKS) where KPROMA is the blocking factor(see above), NF_GP the total number of fields in gridpoint space and NGPBLKS the number of KPROMA blocks. The default for KPROMA is the total number of gridpoints on a processor in which case NGPBLKS is 1.
The ordering of the output fields is as follows (all parts are optional depending on the input switches):

vorticity : NF_UV_G fields (if PSVOR / PSPDIV present and LDVORGP).

divergence : NF_UV_G fields (if PSVOR / PSPDIV present and LDDIVGP).

u : NF_UV_G fields .(if PSVOR / PSPDIV present)

v : NF_UV_G fields (if PSVOR / PSPDIV present).

scalar **fields** : NF_SCALARS_G fields (if PSPSCALAR present).

N–S derivative of *scalar* **fields** : NF_SCALARS_G fields (if PSPSCALAR present and LDSCDERS.)

E–W derivative of u : NF_UV_G fields (if PSVOR / PSPDIV present and LDUVDER.)

E–W derivative of v : NF_UV_G fields (if PSVOR / PSPDIV present and LDUVDER.)

E–W derivative of *scalar* **fields** : NF_SCALARS_G fields (if PSPSCALAR present and LDSCDERS.)

or a combination of the following arrays:

PGPUV(:,:,:,:)
: The "u-v" related grid-point variables in the order described for PGP. The second dimension of PGPUV should be the same as the GLOBAL first dimension of PSPVOR, PSPDIV (in the IFS this is the number of levels). PGPUV need to be dimensioned (NPROMA,ILEVS,IFLDS,NGPBLKS) where IFLDS is the number of "variables" (u,v).

PGP3A(:,:,:,:)
: Grid-point array directly connected with PSPSC3A dimensioned (NPROMA,ILEVS,IFLDS,NGPBLKS) where IFLDS is the number of "variables" (the same as in PSPSC3A).

PGP3B(:,:,:,:)
: Grid-point array directly connected with PSPSC3B dimensioned (NPROMA,ILEVS,IFLDS,NGPBLKS) where IFLDS is the number of "variables" (the same as in PSPSC3B).

PGP2(:,:,:)
: Grid-point array directly connected with PSPSC2 dimensioned (NPROMA,IFLDS,NGPBLKS) where IFLDS is the number of "variables" (the same as in PSPSC2).

Optional

None

C.7 INV_TRANSAD

Adjoint of inverse spectral transform (spectral to gridpoint).

See INV_TRANS. Only differences are that PGP becomes an input argument and PSPVOR, PSPDIV and PSPSCALAR become output arguments.

C.8 TRANS_END

Terminate transform package.

Purpose

Terminate transform package and release all allocated arrays.

Interface

```
CALL TRANS_END
```

Input Arguments

Required
None

Optional
None

Output Arguments

Required
None

Optional
None

C.9 TRANS_INQ

Extract information from the transform package.

Purpose

Interface routine for extracting information from the Transform Package.

Interface

```
CALL TRANS_INQ(...)
```

Input Arguments

Required

None

Optional

KRESOL

Resolution identifier for which info is required. (*Default : First defined resolution*)

Output Arguments

Required

None

Optional

Spectral Space

KSPEC

Number of complex spectral coefficients on this processor.

KSPEC2

2*KSPEC (for use as second dimension of PSPVOR etc.)

KSPEC2G

Global KSPEC2

KSPEC2MX

Maximum KSPEC2 among all processors.

KNUMP

Number of spectral waves handled by this processor.

KGPTOT

Total number of grid columns on this processor.

KGPTOTG

Total number of grid columns on the globe.

KGPTOTMX

Maximum number of grid columns on any of the processors.

KGPTOTL(NPRGPNS:NPRGPEW)

Number of grid columns on each processor.

KMYMS(:)

This processor's spectral zonal wavenumbers.

KASM0(0:)

Address in a spectral array of (m, n=m).

KUMPP(:)

Number of wave numbers each wave set is responsible for.

KPOSSP(:)
Defines partitioning of global spectral fields among processors.

KPTRMS(:)
Pointer to the first wave number of a given "A" set.

KALLMS(:)
Wave numbers for all wave-set concatenated together to give all wave numbers in wave-set order.

KDIMOG(0:)
Defines partitioning of global spectral fields among processors.

Grid-point Space

KFRSTLAT(:)
First latitude of each "A" set in grid-point space.

KLSTTLAT(:)
Last latitude of each "A" set in grid-point space.

KFRSTLOFF
Offset for first lat of own "A" set in grid-point space.

KPTRLAT(:)
Pointer to the start of each latitude.

KPTRFRSTLAT(:)
Pointer to the first latitude of each "A" set in NSTA and NONL arrays.

KPTRLSTLAT(:)
Pointer to the last latitude of each "A" set in NSTA and NONL arrays.

KPTRFLOFF
Offset for pointer to the first latitude of own "A" set NSTA and NONL arrays, i.e. NPTRFRSTLAT(MYSETA)-1.

KSTA(:,:)
Position of first grid column for the latitudes on a processor. The information is available for all processors. The "B" sets are distinguished by the last dimension of NSTA(). The latitude band for each "A" set is addressed by NPTRFRSTLAT(JASET), NPTRLSTLAT(JASET), and NPTRFLOFF=NPTRFRSTLAT(MYSETA) on this processors "A" set. Each split latitude has two entries in NSTA(,:) which necessitates the rather complex addressing of NSTA(,:) and the overdimensioning of NSTA by NPRGPNS-1. For further details, see the discussion on the grid point decomposition in Section (a) on page 17.

KONL(:,:)
Number of grid columns for the latitudes on a processor. Similar to NSTA() in data structure and addressing.

LDSPLITLAT(:)
.TRUE. if latitude is split in grid point space over two "A" sets.

Fourier Space

KULTPP(:)
Number of latitudes for which each "A" set is calculating the FFT's.

KPTRLS(:)
Pointer to first global latitude of each "A" set for which it performs the Fourier calculations.

Legendre Polynomials

PMU(:)
sin(Gaussian latitudes).

`PGW(:)`
Gaussian weights.

`PRPNM(:,:)`
Legendre polynomials on this processor.

`KLEI3`
First dimension of Legendre polynomials.

`KSPOLEGL`
Second dimension of Legendre polynomials.

`KPMS(0:NSMAX)`
Address for Legendre polynomial for a given m.

C.10 EXAMPLES

Both the following examples are for running on a single processor only. For more complex examples see the IFS code (CY23R4 or later), routines SUTRANS, SUMP, TRANSINV_MDL, TRANSDIR_MDL etc.

To compile a program similar to these examples you need to have the directory containing the interface blocks visible with a view corresponding to the version of the trans library you are using and this directory added to your search path for include files. To load you need to load with `-ltrans -lifsaux -lmpi_serial` (and your own libraries) and a suitable load path.

The example shown in Listing C.1 transforms 10 spectral fields into grid-point. The routines `INI_NLOEN`, `READSPEC` and `WRITEGRID` are user routines (not shown).

Listing C.1 *Transforms 10 fields from spectral to grid point.*

```
PROGRAM EXAMPLE1

IMPLICIT NONE
INTEGER NLOEN(320),IFLDS
REAL,ALLOCATABLE :: SPEC(:,:),GP(:,:,:)

INTERFACE
#include "setup_trans0.h"
#include "setup_trans.h"
#include "trans_inq.h"
#include "inv_trans.h"
END INTERFACE

CALL INI_NLOEN(NLOEN) ! Initialize array describing
                      ! reduced grid
CALL SETUP_TRANS0
CALL SETUP_TRANS(KSMAX=319,KDGL=320,KLOEN=NLOEN)
CALL TRANS_INQ(KSPEC2=NSPEC2,KGPTOT=NGPTOT)
IFDLS=10

ALLOCATE(SPEC(IFLDS,NSPEC2))
CALL READSPEC(SPEC,IFLDS,NSPEC2) ! Read in spectral
                                 ! fields
ALLOCATE(GP(NGPTOT,IFLDS,1))
CALL INV_TRANS(PSPSCALAR=SPEC,PGP=GP)

CALL WRITEGRID(GP,IFLDS) ! Write out gridpoint fields

END
```

The second example, shown in Listing C.2 transforms grid-point u and v fields into spectral *vorticity* and *divergence*. Note the dimension of `GP` as `2*IFLDS` to accommodate u followed by v. The routines `INI_NLOEN`, `READGRID` and `WRITESPEC` are user routines (not shown).

Listing C.2 *Transforms grid point u and v to spectral vorticity and divergence.*

```
PROGRAM EXAMPLE2

IMPLICIT NONE
INTEGER NLOEN(320),IFLDS
REAL,ALLOCATABLE :: SPECVOR(:,:),SPECDIV,GP(:,:,:)

INTERFACE
#include "setup_trans0.h"
#include "setup_trans.h"
#include "trans_inq.h"
#include "dir_trans.h"
END INTERFACE

CALL INI_NLOEN(NLOEN) ! Initialize array describing
                      ! reduced grid
CALL SETUP_TRANS0
CALL SETUP_TRANS(KSMAX=319,KDGL=320,KLOEN=NLOEN)
CALL TRANS_INQ(KSPEC2=NSPEC2,KGPTOT=NGPTOT)
IFDLS=10

ALLOCATE(GP(NGPTOT,2*IFLDS,1))
CALL READGRID(GP,IFLDS) ! Read in u and v
                        ! gridpoint fields

ALLOCATE(SPECVOR(IFLDS,NSPEC2))
ALLOCATE(SPECDIV(IFLDS,NSPEC2))
CALL DIR_TRANS(PSPVOR=SPECVOR,PSPDIV=SPECDIV,PGP=GP)

! Write out spectral fields
CALL WRITESPEC(SPECVOR,SPECDIV,IFLDS,NSPEC2)

END
```

Appendix D
FullPos user guide

Author: R. El Khatib
METEO-FRANCE - CNRM/GMAP

Table of contents

D.1 INTRODUCTION

FULLPOS is a powerful and sophisticated post-processing package. It is intended to be used for operation and research as well.

FULLPOS has two main parts: the vertical interpolations, then the horizontal interpolations. In between, a spectral treatment is sometimes possible for the dynamic fields.

D.1.1 Organisation of this manual

This manual contains information about the installation, the use and the management of the code of FULLPOS.

It is assumed that the user has some familiarity with the configuration 001 of ARPEGE/IFS or ALADIN and understands the basic features of post-processing operations.

Much of the information presented in this document is also available inside the code via the comments, especially in the data modules.

D.1.2 Reporting bugs

If you find any bugs or deficiencies in this software, then please write a short report and send it to the author.

FULLPOS has so many features that it is difficult to validate all the possible namelists configurations.

If you have wishes for further developments inside FULLPOS, then please write a short report as well, that could be discussed.

D.1.3 Summary of features

FULLPOS is a post-processing package containing many features. The following is just a small list of the main available features:

- Multiple fields from the dynamics, the physics, the cumulated fluxes or the instantaneous fluxes.
- Post-processing available on any pressure level, height (above output orography) level, potential vorticity level, potential temperature level or model level.
- Multiple latitudes X longitudes output subdomains, or one Gaussian grid with any definition, or one grid of kind 'ALADIN', with any definition.
- Multiple possible optimisations of the memory or the CPU time used, through specific I/O schemes, vectorisation depth, distribution and various other segmentations.
- Possible spectral treatment for all the fields of a given post-processing level type.
- Customization of the names of the post-processed fields.
- Support for computing a few other fields without diving deeply into the code of FULLPOS.
- Ability to perform post-processing in-line (ie: during the model integration) or off-line (out of the model integration).
- Ability to make ARPEGE or ALADIN history files, starting from a file ARPEGE or a file ALADIN (processes "927", "E927" and "EE927").

D.1.4 Acknowledgements

Thanks to Alain Joly who invented first the "French POS" concept which became FULLPOS, and to Jean-François Geleyn who has adopted my point of view about this internal new post-processing. Credit and thanks to Jean Pailleux who convinced ECMWF to let METEO-FRANCE implement this software in ARPEGE/IFS; to Mats Hamrud for his advice on vertical scannings, his help for long distance debugging and the re-usable code he has written on I/O scheme, spectral transforms and horizontal scanning; to Vincent Cassé for these long talks about interpolations and how the so-called "semi-Lagrangian buffers" work; to Jean-Marc Audoin and Eric Escalière who helped me to write a part of the code; to Patrick Le Moigne and Jean-Daniel Gril who spent time to let me try to understand the geometry of ALADIN. Congratulations and thanks to Gabor Radnoti who managed in the huge task to implement FULLPOS

inside ALADIN to Jaouad Boutahar and Mehdi Elabed for their debugging in FULLPOS. Many thanks to Jean-Noël Thépaut who believed in the use of FULLPOS for the incremental variational analysis. Thanks to you all who will use FULLPOS and be happy of it (...and maybe find out residual bugs?).

Special thanks to the workstation "Nout", to Edit_file and the mouse on NOS-Ve with which the code is typed, and to the user friendly Crisp editor under UNIX environment, with which this manual has been typed.

D.2 BASIC USAGE

D.2.1 Getting started

(*a*) *Installing the software*

FULLPOS is embedded in the software ARPEGE/IFS/ALADIN. It needs the auxiliary library for the I/Os and some low-level calculations, and the external spectral transforms packages TFL and TAL (the last one is needed for running FULLPOS ALADIN only).

(*b*) *Preparing the namelists file*

The namelists file should correspond to the ARPEGE/IFS/ALADIN cycle you are running.

FULLPOS is using a few specific namelists which are: NAMAFN, NAMFPC, NAMFPD, NAMFPG, NAMFPF, NAMFPIOS, NAMFPSC2, NAMFPEZO NAMCAPE.

All these namelists are specific to FULLPOS, except NAMAFN which is a little bit more general.

FULLPOS is also using model variables from the namelists NAMCT0 NAMDIM NAMDYN NAMPAR0 NAMPAR1 NAMOPH NAMFA NAMCT1.

Furthermore it is indirectly interfaced with the model via the namelists NAMPHY, NAMDPHY, NAMINI, NAMCFU and NAMXFU.

(*c*) *Running the software*

To run the software anyhow, you have to control that the next basic namelist variables are properly set:

NCONF :

Definition : General configuration of the ARPEGE/IFS/ALADIN software. This parameter is also accessible as a command line option: `-c`

Scope : Integer which *must* be 1 to enable the post-processing.

Default value : in namelist the default value is 1; if the command line option is used there is no default value.

Namelist location : NAMCT0

CNMEXP :

Definition : Name of the experiment. This parameter is also accessible as a command line option: `-e`

Scope : string of 4 characters.

Default value : in namelist the default value is `'0123'`; if the command line option is used there is no default value.

Namelist location : NAMCT0

LECMWF :

Definition : Control of setup version. (Set `.TRUE.` for ECMWF setup and `.FALSE.` for MÉTÉO-FRANCE setup). This parameter is also accessible as a command line option: `-v`

Scope : in namelist: boolean; in command line: character string which can be either `'ecmwf'` (for LECMWF=.TRUE.) or `'meteo'` (for LECMWF=.FALSE.).

Default value : in namelist the default value is `.TRUE.`; if the command line option is used there is no default value.

Namelist location : NAMCT0

LELAM :

Definition : Control of the limited area vs. global version of the model. (Set .TRUE. for ALADIN and .FALSE. for ARPEGE/IFS). This parameter is also accessible as a command line option: -m

Scope : in namelist: boolean; in command line: character string which can be either 'arpifs' (for LELAM=.FALSE.) or 'aladin' (for LELAM=.TRUE.).

Default value : in namelist the default value is .FALSE.; if the command line option is used the default value is 'arpifs'.

Namelist location : NAMCTO

LFPOS :

Definition : Main control of FULLPOS software; set LFPOS=.TRUE.— to activate it.

Scope : Boolean.

Default value : .FALSE.

Namelist location : NAMCTO

N1POS :

Definition : Post-processing outputs control switch. Set N1POS=1 to switch on, and N1POS=0 to switch off.

Scope : Integer between 0 and 1.

Default value : 1

Namelist location : NAMCT1

NFRPOS, NPOSTS :

Definition : Post-processing outputs monitor, working as follows:

- if NPOSTS(0) = 0 then the post-processing runs every NFRPOS time steps (including time 0).
- if NPOSTS(0) > 0 then NPOSTS(0) is the number of post-processing events and the post-processing runs on the time steps NPOSTS(1)*NFRPOS, NPOSTS(2)*NFRPOS, ...NPOSTS(NPOSTS(0))*NFRPOS.
- if NPOSTS(0) < 0 then -NPOSTS(0) is the number of post-processing events and the post-processing runs on the hours -NPOSTS(1)*NFRPOS, -NPOSTS(2)*NFRPOS, ...-NPOSTS(NPOSTS(0))*NFRPOS.

Scope : Respectively positive integer, and integer array sized 0 to 240.

Default value : If LECMWF=.FALSE. and NCONF=1 and the command line is used then NFRPOS=1 and NPOSTS is set for output at hours 0, 6, 12, 18, 24, 30, 36, 48, 60 and 72. Else NFRPOS=NSTOP and NPOSTS(:)=0 (outputs at first and last time step).

Namelist location : NAMCTO

If you do not specify anything else, then FULLPOS will run, but you will not get any output file since you did not ask for any output field!

Imagine now that you add in the namelist NAMFPC the following variables:

```
CFP3DF='GEOPOTENTIEL','TEMPERATURE',
RFP3F=50000.,85000.,
```

then you will get a post-processing file which will contain the geopotential and the temperature at 500 hPa and 850 hPa on the model grid (stretched Gaussian grid in the case of ARPEGE, geographical "C+I" grid in the case of ALADIN. The output file will be a file ARPEGE/ALADIN named PF${CNMEXP}000+*nnnn*, where ${CNMEXP} is the name of the experiment (CNMEXP(1:4)), and *nnnn* the forecast range.

D.2.2 Leading namelists and variables

The namelists variables and the set-up have been built in order to use the namelists default values as far as possible, and to respect a hierarchy.

This section will describe the purpose of the main post-processing namelists and will detail the basic variables in these namelists.

(*a*) *NAMFPC*

This is the main namelist for the post-processing. It contains the list of the fields to post-process, the format of the output subdomain(s) (spectral coefficients, Gaussian grid, LAM grid or LAT × LON grids), and various options of post-processing.

CFPFMT :

Definition : format of the output files.

Scope : character variable which can be either 'MODEL', 'GAUSS', 'LELAM' or 'LALON' respectively for spectral coefficients, a global model grid, a LAM grid a set of LAT × LON grids.

Default value : 'GAUSS' in ARPEGE/IFS 'LELAM' in ALADIN.

CFPDOM :

Definition : names of the subdomains.

Scope : array of 10 characters; if CFPFMT is 'MODEL', 'GAUSS' or 'LELAM' then you can make only one output domain; otherwise you can make up to 15 subdomains.

Default value : CFPDOM(1)='000'; CFPDOM(i)=' ' for i greater than 1. This means that by default, you ask for only one output (sub-)domain.

CFP3DF :

Definition : ARPEGE names of the 3D dynamics fields.

Scope : array of 12 characters, maximum size: 98 items. The reference list of these fields is written in Section D.6.1 on page 127.

Default value : blank strings (no 3D dynamics fields to post-process).

CFP2DF :

Definition : ARPEGE names of the 2D dynamics fields.

Scope : array of 16 characters, maximum size: 78 items. The reference list of these fields is written in Section (a) on page 128.

Default value : blank strings (no 2D dynamics fields to post-process).

CFPPHY :

Definition : ARPEGE names of the surface grid-point fields from physical parameterisations.

Scope : array of 16 characters, maximum size: 328 items. The reference list of these fields is written in Section (b) on page 129.

Default value : blank strings (no surface fields to post-process).

CFPCFU :

Definition : ARPEGE names of the cumulated fluxes.

Scope : array of 16 characters, maximum size: 63 items. The reference list of these fields is written in Section (c) on page 130.

Default value : blank strings (no cumulated fluxes to post-process).

CFPXFU :

Definition : ARPEGE names of the instantaneous fluxes.

Scope : array of 16 characters, maximum size: 63 items. The reference list of these fields is written in Section (d) on page 132.

Default value : blank strings (no instantaneous fluxes to post-process).

RFP3P :

Definition : post-processing pressure levels.

Scope : array of real values, maximum size: 31 items. Unit: Pascal.

Default value : None.

RFP3H :

Definition : post-processing height levels above orography.

Scope : array of real values, maximum size: 127 items. Unit: meter.

Default value : None.

RFP3TH :

Definition : post-processing potential temperature levels.

Scope : array of real values, maximum size: 15 items. Unit: Kelvin.

Default value : None.

RFP3PV :

Definition : post-processing potential vorticity levels.

Scope : array of real values, maximum size: 15 items. Unit: Potential Vorticity Unit.

Default value : None.

NRFP3S :

Definition : post-processing *eta* levels.

Scope : array of real values, maximum size: 200 items. Unit: adimensional.

Default value : None.

Notice:

- If you ask for fluxes you do not need to specify anything particular in the namelists NAMCFU or NAMXFU: these namelists will be automatically modified by FULLPOS in order to get the required fluxes.
- If you ask for spectral coefficients then the upper air grid-point fields, the surface grid point fields and the fluxes will be written on the model Gaussian grid.

(*b*) *NAMFPD*

This namelist defines the boundaries and the horizontal dimensions of each output subdomain. Many default values are available through a clever use of the previous namelist NAMFPC.

Note that if you ask for the model horizontal geometry (CFPFMT='MODEL'), all these parameters will be reset by the program; so you should not try to choose them yourself.

NLAT, NLON :

Definition : respectively number of latitudes and longitudes for each output (sub-)domain (corresponding respectively to the variables NDGLG and NDLON of a model grid).

Scope : arrays of integers.

Default value : It depends on the variables CFPFMT and LELAM as shown in Table D.1 on page 96.

RLATC, RLONC :

Definition : respectively latitude and longitude of the center of each output (sub-)domain (if CFPFMT='GAUSS' then these variables are useless).

Scope : arrays of reals; unit: degrees.

Default value : It depends on the variable CFPFMT.

If CFPFMT='LALON' then refer to Table D.2 on page 97;
elseif CFPFMT='LELAM' then refer to Table D.3 on page 98.

RDELY, RDELX :

Definition : respectively the mesh size in latitude and longitude for each output (sub-)domain (if
VarValCFPFMT'GAUSS' then these variables are useless).

Scope : arrays of reals; unit: degrees if CFPFMT='LALON', meters if CFPFMT='LELAM'.

Default value : It depends on the variable CFPFMT.
If CFPFMT='LALON' then refer to Table D.2 on page 97;
elseif CFPFMT='LELAM' then refer to Table D.3 on page 98.

NFPGUX, NFPLUX :

Definition : respectively number of geographical latitude rows and longitude rows for each output (sub-)domain (these variables are useful only if CFPFMT='LELAM': they correspond to the definition of the so-called "C+I" area while NLAT and NLON are corresponding to the area "C+I+E").

Scope : arrays of integers.

Default value : It depends on the variable FPDOM. Refer to Table D.3 on page 98.

Table D.1 *Default values for NLAT and NLON according to CFPFMT and LELAM.*

(NLAT,NLON) CFPFMT	'GAUSS'	LELAM	'LALON'
LELAM			
.FALSE.	(NDGLG,NDLON)	See Table D.3	See Table D.2
.TRUE.	(32,64)	(NFPGUX,NFPLUX)	See Table D.2

Table D.2 *Default values for* LAT × LON *subdomains according to the value of* CFPDOM.

CFPDOM	NLAT	NLON	RLATC	RLONC	RDELY	RDELX
'HENORD'	60	180	45.	179.	1.5	2.
'HESUDC'	60	180	-45.	179.	1.5	2.
'HESUDA'	30	90	-45.	178.	3.	4.
'ATLMED'	65	129	-48.75	-20.	0.75	1.
'EURATL'	103	103	45.75	2.	0.5	2/3
'ZONCOT'	81	81	48.75	0.	0.375	0.5
'FRANCE'	61	61	45.75	2.	0.25	1/3
'GLOB15'	121	240	0.	179.25	1.5	1.5
'EURAT5'	105	149	46.	5.	0.5	0.5
'ATOUR10'	81	166	40.	-17.5	1.	1.
'EUROC25'	105	129	48.	1.	0.25	0.25
'GLOB25'	73	144	0.	178.75	2.5	2.5
'EURSUD'	41	54	38.25	-19/3	0.5	2/3
'EUREST'	39	73	50.75	16/3	0.5	2/3
'GRID25'	21	41	50.	0.	2.5	2.5
'MAROC'	158	171	31.05	-6.975	23.7/157	25.65/170
'OCINDIEN'	67	89	-16.5	66.	1.5	1.5
'REUNION05'	61	141	-20.	65.	0.5	0.5
else - case ARPEGE	0	0	0.	0.	0.	0.
else - case ALADIN	NDGUXG	NDLUXG	computed	computed	computed	computed

Table D.3 *Default values for* LAM *subdomains according to the value of* CFPDOM.

CFPDOM	NLAT	NLON	RLATC	RLONC
'BELG'	61	61	50.44595488554766	4.90727841961041
'SLOV'	37	37	46.05017943078632	13.52668207859151
'MARO'	149	149	31.56059442218072	-7.00000000285346
'OPMA'	97	97	31.56059442218072	-7.00000000285346
'LACE'	181	205	46.24470063381371	16.99999999944358
'ROUM'	61	61	44.77301981937139	25.00000000483406
'FRAN'	189	189	45.31788242335041	1.27754303826285
else - case ARPEGE	169	169	46.46884540633992	2.57831063089259
else - case ALADIN	NDGUXG	NDLUXG	EDELY	EDELX
CFPDOM	NFPGUX	NFPLUX	RDELY	RDELX
'BELG'	61	61	12715.66669793411	12715.66669793552
'SLOV'	37	37	26271.55175398597	26271.55175829969
'MARO'	149	149	18808.17793051683	18808.17792427479
'OPMA'	97	97	31336.13991686922	31336.13988918715
'LACE'	181	205	14734.91380550296	14734.913810093
'ROUM'	61	61	33102.6285617361	33102.62857952392
'FRAN'	189	189	12715.67301977791	12715.66779231173
else - case ARPEGE	169	169	12715.6635946432	12715.66736292664
else - case ALADIN	NDGUXG	NDLUXG	EDELY	EDELX
CFPDOM	FPLONO	FPLATO		
'BELG'	2.57831001	46.46884918		
'SLOV'	17.0	46.24470064		
'MARO'	-7.0	31.56059436		
'OPMA'	-7.0	31.56059436		
'LACE'	17.0	46.24470064		
'ROUM'	25.0	44.77301983		
'FRAN'	25.0	44.77301983		
else - case ARPEGE	2.57831001	46.46884918		
else - case ALADIN	ELONO	ELATO		

(*c*) *NAMFPG*

This namelist defines the geometry of the output subdomain(s). It is used mostly when the output geometry is a Gaussian grid or a LAM grid. Default geometry is the model geometry.

Note that if you ask for the model horizontal geometry (CFPFMT='MODEL'), all these parameters will be reset by the program; so you should not try to choose them yourself.

NFPMAX :

Definition : A truncation order which definition depends on the variable CFPFMT:

- If CFPFMT='GAUSS' it is the truncation order of the output grid.
- If CFPFMT='LELAM' it is the *meridional* truncation order of the output grid.
- If CFPFMT='LALON' it is the truncation used to filter in spectral space the post-processed fields.

Scope : array of integers. Maximum size 15 items.

Default value :

- If CFPFMT='GAUSS' then NFPMAX is computed like for a quadratic grid: so that 3*NFPMAX(:)+1 $\geq$ NLON(:)
- If CFPFMT='LELAM' then NFPMAX is computed like for a quadratic grid: so that 3*NFPMAX(:)+1 $\geq$ NLAT(:)
- If CFPFMT='LALON' NFPMAX is computed like for a quadratic grid: so that 3*NFPMAX(:)+1 $\geq$ min(NLAT(:), NLON(:))

NMFPMAX :

Definition : Truncation order in the *zonal* direction (used only if CFPFMT='LELAM').

Scope : integer.

Default value : If; else if CFPFMT='LELAM' then NMFPMAX is computed like for a quadratic grid: so that 3*NMFPMAX+1 $\geq$ NLON(1)

FPMUCEN, FPLOCEN :

Definition : respectively Sine of the latitude, and longitude of either the pole of interest if CFPFMT='GAUSS', or the location of the observed cyclone (for bogussing purpose — refer to Section D.4.4 on page 122 —) if CFPFMT='LELAM'. This variable is useless if CFPFMT='LALON'.

Scope : reals; unit: adimentional for FPMUCEN, and radians for FPLOCEN

Default value : in ARPEGE/IFS respectively RMUCEN and RLOCEN. In ALADIN respectively sin(ELAT0))— and ELON0.

NFPHTYP :

Definition : reduction of the Gaussian grid. Used only if CFPFMT='GAUSS'.

Scope : Integer which value can be either 0 (for a regular grid) or 2 (for a reduced grid).

Default value : NFPHTYP=NHTYP in ARPEGE/IFS if NLAT(1)=NDGLG; otherwise NFPHTYP=0.

NFPRGRI :

Definition : number of active points on each parallel of a Gaussian grid. Used only if CFPFMT='GAUSS'. Reduced grids can be computed thanks to the procedure surgery[1].

Scope : Integer array to be filled from subscript 1 to NLAT(i)/2 (Northern hemisphere only): subscript 1 corresponds to row the nearest to the pole; subscript NLAT(i)/2 corresponds to the row the nearest to the equator. Both hemisphere are assumed to be symmetric.

Default value : NFPRGRI(1:(NLAT(1)+1)/2)=NRGRI(1:(NDGLG+1)/2) if NLAT(1)=NDGLG; else NFPRGRI(1:NLAT(1))=NLON(1).

[1] http://intra.cnrm.meteo.fr/gmod/modeles/procedures/surgery.html

FPSTRET :

Definition : stretching factor. Used only if CFPFMT='GAUSS'.

Scope : Real value. Unit: adimensional.

Default value : FPSTRET=RSTRET in ARPEGE/IFS FPSTRET=1. in ALADIN.

NFPTTYP :

Definition : Transformation type (used to rotate or deform model fields). This variable is useless if CFPFMT='LALON'.

- If NFPTTYP=1 then the pole of interest is at the North pole of the geographical Earth.
- If NFPTTYP=2 and CFPFMT='GAUSS' in ARPEGE/IFS then the pole of interest is anywhere else on the geographical Earth.
- If NFPTTYP=2 and CFPFMT='LELAM' in ALADIN the cyclone is moved to the location of the observed cyclone (for bogussing purpose — refer to Section D.4.4 on page 122 —).

Scope : Integer which value can be only 1 or 2.

Default value : In ARPEGE/IFS and if CFPFMT='GAUSS': NFPTTYP=NSTTYP. In all other cases NFPTTYP=1.

FPNLGINC :

Definition : non-linear grid increment. Used only if CFPFMT='GAUSS' to compute the value: NLON(1)-1)/NFPMAX(1).

Scope : Real value between 2. (linear grid) and 3. (quadratic grid).

Default value : FPNLGINC=1.

FPLAT0, FPLON0 :

Definition : respectively the geographic latitude and longitude of reference for the projection (used only if CFPFMT='LELAM').

Scope : Real values. Unit: degrees.

Default value : It depends from the variable CFPDOM. Refer to Table D.3 on page 98.

NFPLEV :

Definition : number of vertical levels.

Scope : Integer between greater or equal to 1, and limited to 200.

Default value : NFPLEV=NFLEVG

FPVALH, FPVBH :

Definition : respectively the "A" and "B" coefficients of the vertical coordinate system.

Scope : real arrays. Unit: FPVALH is in Pascal; FPVBH is adimensional.

Default value : if NFPLEV=NFLEVG then
FPVALH(1:NFPLEV)=VALH(1:NFLEVG) and
FPVBH(1:NFPLEV)=VBH(1:NFLEVG) (model levels). Else the program will attempt to recompute FPVALH and FPVBH to fit with NFPLEV, using vertical levels that may have been used in operations in the past.

FPVP00 :

Definition : Reference pressure.

Scope : real value. Unit: Pascal.

Default value : FPVP00=VP00.

D.2.3 Output files handling

(*a*) *File structure*

Output files are ARPEGE/ALADIN files.

- If you ask for a Gaussian grid in output (CFPFMT='GAUSS') you will get a file ARPEGE.
- If you ask for a LAM grid (CFPFMT='LELAM') you will get a file ALADIN.
- If you ask for LAT × LON grids (CFPFMT='LALON') you will get files ALADIN with the only particularity that the output geometry is not projected.
- If you ask for the model geometry (CFPFMT='MODEL') you can get either spectral or gridpoint data.

Notice: to plot LAM or LAT × LON grids you can use the graphic procedure chagal[2].

(*b*) *File name*

There is one post-processing file for each post-processing time step and each (sub-)domain.

The output files are named: PF${CNMEXP}${CFPDOM}+*nnnn*, where:

PF is a prefix

$CNMEXP is the so-called "name of the experiment" (value: CNMEXP(1:4))

$CFPDOM is the name of the output (sub-)domain (CFPDOM)

nnnn is the time stamp.

Example: if you ask for post-processing at time 0, with CNMEXP='FPOS' and CFPDOM'ANYWHERE', then the output file will be named: PFFULLANYWHERE+0000.

(*c*) *File content*

To read a field in an output file, you have to specify through the subroutine FACILE the name of the field you wish to get.

For a "surface" field, this name is the ARPEGE field name that has been defined in the namelist NAMFPC; it is a string of 16 characters.

For an upper air field, this name is also the ARPEGE field name that has been defined in the namelist NAMFPC (string of 12 characters), but furthermore, you must specify the kind of post-processing level ("prefix" of the field) and the value of this level. There are 5 possibilities, according to the level type as shown in Table D.4 on page 101.

Table D.4 *Prefix, unit and number of letters to write upper air fields prefix.*

Level type	Prefix	Unit	Number of letters for level value
Pressure	P	Pascal	5
Height	H	Meter	5
Potential vorticity	V	deciPVU	3
Potential temperature	T	Kelvin	3
Eta	S	-	3

Example: temperature at 2 PVU is V020TEMPERATURE

Warning: fields on pressure levels bigger or equal to 1000 hPa are written out with truncated names; for example, temperature at 1000 hPa is P00000TEMPERATURE while P00500TEMPERATURE could be as well the temperature at 5 hPa or the temperature at 1005 hPa!

[2] http://www.cnrm.meteo.fr/aladin/concept/Chagal0.html

D.3 ADVANCED USAGE

The purpose of this chapter is to describe supplementary namelists variables which users may need, but which are either too complex, or too rarely needed to warrant complicating the previous chapter.

D.3.1 Scientific options

(*a*) *Spectral fit on dynamic fields*

If you wish to post-process surface dynamic fields or upper air dynamic fields on pressure levels, potential temperature levels or potential vorticity levels, it is possible to perform a spectral fit between the vertical interpolations and the horizontal interpolations. The spectral fit will remove the nemerical noise which has been generated by the vertical interpolation and which is beyond the model truncation.

`LFITP` :

Definition : Spectral fit of post-processed fields on pressure levels.

Scope : Boolean.

Default value : `.TRUE.`

Namelist location : `NAMFPC`

`LFITT` :

Definition : Spectral fit of post-processed fields on potential temperature levels.

Scope : Boolean.

Default value : `.FALSE.`

Namelist location : `NAMFPC`

`LFITV` :

Definition : Spectral fit of post-processed fields on potential vorticity levels.

Scope : Boolean.

Default value : `.FALSE.`

Namelist location : `NAMFPC`

`LFIT2D` :

Definition : Spectral fit of 2D post-processed fields.

Scope : Boolean.

Default value : `.TRUE.`

Namelist location : `NAMFPC`

Notice:

- If you wish to post-process upper air dynamic fields on height levels or hybrid levels, it is not possible to apply such spectral fit because the horizontal interpolations are performed *before* the vertical interpolation in order to respect the displacement of the planetary boundary layer.
- If you post-process dynamic fields which are not represented by spectral coefficients in the model, then these fields will not be spectrally fitted, even if the corresponding key `LFITxx` is `.TRUE.` In the same way, if you post-process a specific dynamic field which is represented by spectral coefficients in the model, then this field will be spectrally fitted whenever the corresponding key `LFITxx` is `.TRUE.`. However it is possible to change the native representation of a field: refer to Section (a) on page 111.

(*b*) *Tuning of the spectral filters*

Several fields can be smoothed via tunable filters activated in spectral space (refer to Section D.9 on page 136 for the formulation of these filters). These parameters are contained in the specific namelist NAMFPF.

LFPBED, RFPBED :

Definition : Respectively switch and intensity of the filter on the so-called "derivative" fields, that is: horizontal derivatives or those which are built after horizontal derivatives (absolute and relative vorticites, divergence, vertical velocity, stretching and shearing deformations, potential vorticity and all fields interpolated on potential vorticity levels).

Scope : Respectively boolean and real. Unit: adimensional.

Default value : LFPBED=.TRUE.; RFPBED—≈3.08[3] in ARPEGE/IFS, RFPBED=6. in ALADIN.

NFMAX :

Definition : Truncation threshold of each (sub-)domain for the filter on the so-called "derivative" fields (used only in ARPEGE/IFS if the model is stretched).

Scope : Integer array. Maximum size: 15 items.

Default value : If CFPFMT='GAUSS' then NFMAX(1)=NFPMAX(1)*FPSTRET.
Else if CFPFMT='MODEL' then NFMAX(1)=NFPMAX(1)*FPSTRET *which means that the fields will never be filtered.*
Else NFPMAX is computed like for a quadratic grid:
so that 3*NFMAX(:)+1 $\geq$ min(NLAT(:),NLON(:)

LFPBEG, RFPBEG :

Definition : Respectively switch and intensity of the filter on geopotential.

Scope : Respectively boolean and real. Unit: adimensional.

Default value : LFPBEG=.TRUE.; RFPBEG=4. in ARPEGE/IFS, RFPBEG=6 in ALADIN.

LFPBET, RFPBET :

Definition : Respectively switch and intensity of the filter on temperature.

Scope : Respectively boolean and real. Unit: adimensional.

Default value : LFPBET=.TRUE.; RFPBET=4. in ARPEGE/IFS, RFPBET=6 in ALADIN.

LFPBEP, RFPBEP :

Definition : Respectively switch and intensity of the filter on medium sea level pressure.

Scope : Respectively boolean and real. Unit: adimensional.

Default value : LFPBEP=.TRUE.; RFPBEP=4. in ARPEGE/IFS, RFPBEP=6. in ALADIN.

LFPBEH, RFPBEH :

Definition : Respectively switch and intensity of the filter on relative humidity.

Scope : Respectively boolean and real. Unit: adimensional.

Default value : LFPBEH=.TRUE.; RFPBEH=4. in ARPEGE/IFS, RFPBEH=6. in ALADIN.

Notice:

- Only one filter can be appled to a given field; consequently, in case of ambiguity in the choice of filter (example: geopotential on an iso-PV surface), only the "derivative" filter is applied.
- Filters are applied even if the post-processed fields should be represented in spectral coefficients.

[3]This odd value stands here for a historical continuity reason.

(*c*) *Climatology*

In horizontal interpolations the usage of auxiliary climatology data improves the accuracy of the upper air fields when interpolated on surface-dependent levels, and of several surface fields. Section D.8 on page 135 explains how to make such files.

NFPCLI :

Definition : Usage level for climatology data:

- If NFPCLI=0 climatology data are not used.
- If NFPCLI=1 the horizontal interpolations use the surface geopotential and the land-sea mask of a target climatology file. In this case the climatology file name in the local script should be: "const.clim.CFPDOM*(i)*" where i is the (sub-)domain subscript.
- If NFPCLI=3 the horizontal interpolations use a larger set of climatology surface fields, including constant and monthly values. In this case two climatology files are used: one with the source geometry and one with the target geometry. In the local script the source climatology file name should be: "Const.Clim" while the target climatology file name should be: "const.clim.CFPDOM*(i)*" where i is the (sub-)domain subscript.

Table D.5 on page 104 lists the climatology fields read in function of the namelist keys.

Scope : Integer which value can be only 0, 1 or 3.

Default value : NFPCLI=0

Namelist location : NAMFPC

Table D.5 *Climatology fields read in function of the namelist keys.*

Field	Namelist keys
surface geopotential	NFPCLI ≥ 1
land-sea mask	NFPCLI ≥ 1 and (LMPHYS or LEPHYS)
surface temperature	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
relative surface wetness	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
deep soil temperature	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
relative deep soil wetness	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
snow depth	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
albedo	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
emissivity	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
standard deviation of surface geopotential	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
percentage of vegetation	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
roughness length	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
anisotropy coefficient of topography	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
direction of the main axis of topography	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
type of vegetation	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
minimum stomatal resistance	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
percentage of clay	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
percentage of sand	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
root depth	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
leaf area density	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
thermal roughness length	NFPCLI ≥ 3 and (LMPHYS or LEPHYS)
surface snow albedo	NFPCLI ≥ 3 and (LMPHYS or LEPHYS) and LVGSN
surface snow density	NFPCLI ≥ 3 and (LMPHYS or LEPHYS) and LVGSN

RFPCORR :

Definition : Critical difference of surface geopotential between the model and the source climatology in order to correct surface temperature through the standard vertical profile.

Scope : Real. Unit: J/kg.

Default value : 300.*g.

Namelist location : NAMFPC

RFPCSAB :

Definition : Critical difference of sand percentage between the model and the source climatology in order to compute the relative soil moisture.

Scope : Real. Unit: adimensional.

Default value : 0.01.

Namelist location : NAMFPC

RFPCD2 :

Definition : Critical difference of depth between the model and the source climatology in order to compute the relative soil moisture.

Scope : Real. Unit: m.

Default value : 0.001 m.

Namelist location : NAMFPC

LFPMOIS :

Definition : Month selected while using climatology data (used only if NFPCLI *ge* 3):

- if LFPMOIS=.FALSE. then the month is the one of the model (forecast).
- if LFPMOIS=.TRUE. then the month is the one of the input initial file. This option should lead to less accurate fields but it enables in-line post-processing[4].

Scope : Boolean.

Default value : .FALSE.

Namelist location : NAMFPC

(*d*) *Optional pronostic fields*

The model is able to run with optional pronostic fields. These fields would be interpolated by the post-processing if they are declared as *present* in the model. But if they are not, then the post-processing would create and fulfill them as it can.

NFPASS :

Definition : Number of spectral passive scalars in the model.

Scope : Integer between 0 and 5.

Default value : 0

Namelist location : NAMDIM

LNHDYN :

Definition : Control of the non-hydrostatic model; if LNHDYN=.TRUE. then pressure departure and vertical divergence fields are read in and thus interpolated. Else pressure departure and vertical divergence are created. Pressure departure field is then fulfilled with zero, while vertical divergence is diagnosed.

Scope : Boolean. To run the model with this option you need the ALADIN software.

Default value : .FALSE.

Namelist location : NAMCTO

[4]The post-processing is performed during the direct model integration.

LSPQ, LGPQ :

Definition : Respectively spectral and gridpoint atmospheric specific humidity represented as pronostic variables in the model.

Scope : Boolean. Possible values: any pair of booleans except (.TRUE.,.TRUE.).

Default value : if LECMWF=.TRUE. then (LSPQ,LGPQ)=(.FALSE.,.TRUE.).
Else (LSPQ,LGPQ)=(.TRUE.,.FALSE.).

Namelist location : NAMDIM

LSPL, LGPL :

Definition : Respectively spectral and gridpoint atmospheric liquid water represented as pronostic variables in the model.

Scope : Boolean. Possible values: any pair of booleans except (.TRUE.,.TRUE.).

Default value : if LECMWF=.TRUE. then (LSPL,LGPL)=(.FALSE.,.TRUE.)
Else (LSPL,LGPL)=(.FALSE.,.FALSE.).

Namelist location : NAMDIM

LSPI, LGPI :

Definition : Respectively spectral and gridpoint atmospheric solid water (ice) represented as pronostic variables in the model.

Scope : Boolean. Possible values: any pair of booleans except (.TRUE.,.TRUE.).

Default value : if LECMWF=.TRUE. then (LSPI,LGPI)=(.FALSE.,.TRUE.)
Else (LSPI,LGPI)=(.FALSE.,.FALSE.).

Namelist location : NAMDIM

LSPA, LGPA :

Definition : Respectively spectral and gridpoint cloud fraction represented as pronostic variables in the model.

Scope : Boolean. Possible values: any pair of booleans except (.TRUE.,.TRUE.).

Default value : if LECMWF.TRUE. then (LSPA,LGPA)=(.FALSE.,.TRUE.)
Else (LSPA,LGPA)=(.FALSE.,.FALSE.).

Namelist location : NAMDIM

LSPO3, LGPO3 :

Definition : Respectively spectral and gridpoint ozone mixing ratio represented as pronostic variables in the model.

Scope : Boolean. Possible values: any pair of booleans except (.TRUE.,.TRUE.).

Default value : (.FALSE.,.FALSE.).

Namelist location : NAMDIM

(*e*) *Adiabatic post-processing*

To run the post-processing in the adiabatic model, you should carefully remove the physical fields from the model, by setting the following variables in namelists:

```
/NAMPHY
   LSOLV=.FALSE.,
   LFGEL=.FALSE.,
   LFGELS=.FALSE.,
   LMPHYS=.FALSE.,
   LNEBN=.FALSE.,
/END
/NAMDPHY
   NVSO=0,
   NVCLIV=0,
   NVRS=0,
   NVSF=0,
   NVSG=0,
   NCSV=0,
   NVCLIN=0,
   NVCLIP=0,
/END
```

(*f*) *Horizontal interpolations*

It is possible to control the kind of horizontal interpolations, for dynamic fields on one side, and for physical fields and fluxes on the other side:

NFPINDYN :

Definition : control of horizontal interpolations for dynamic fields:

- NFPINDYN=12: quadratic interpolations
- NFPINDYN=4: bilinear interpolations
- NFPINDYN=0: to adopt the nearest point rather than interpolating.

Scope : Integer which value can be only 0, 4 or 12.

Default value : 12

Namelist location : NAMFPC

NFPINPHY :

Definition : control of horizontal interpolations for physical fields and fluxes:

- NFPINPHY=12: quadratic interpolations
- NFPINPHY=4: bilinear interpolations
- NFPINPHY=0: to adopt the nearest point rather than interpolating.

Scope : Integer which value can be only 0, 4 or 12.

Default value : 12

Namelist location : NAMFPC

Notice: setting NFPINPHY=NFPINDYN=0 enables to run the post-processing without any climatology, even when any ISBA field is requested.

(*g*) *The problem of lakes and islands*

When the output resolution is so that a single gridpoint lake or island is created, the horizontal interpolations taking into account the land/sea nature will not work properly since no neighbouring

points will be of the same nature as the target point; hence all the neighbouring points will be used in the interpolation process. This can lead to irrealistic temperatures or water contents.

To avoid this, an alternative option has been developed:

LFPLAKE :

Definition : Special treatment for lake and islands; when it is set to `.TRUE.` the surface and deep soil temperatures and water contents will be modified as follows:

- values on isolated lakes or islands gridpoint created by the interpolations will be overwritten by the climatology data
- values on any lake gridpoint, *as identified by the climatology*, will be overwritten by the climatology data (to improve the existing quality of the climatology data over lakes, when it is possible).

Scope : Boolean.

Default value : `.FALSE.`

Namelist location : NAMFPC

Notice: the positive impact of the feature still need be proved.

(*h*) *Computation of CAPE*

The computation of the Convective Available Potential Energy (CAPE) is widely tunable:

NFPCAPE :

Definition : Kind of computation:

- NFPCAPE=1: computation starts from the lowest model level
- NFPCAPE=2: computation starts from the most unstable model level
- NFPCAPE=3: computation starts from the recomputed temperature and relative moisture at 2 meters
- NFPCAPE=4: computation starts from the analysed temperature and relative moisture at 2 meters.

Scope : Integer which value can be only 1,2 3 or 4.

Default value : 2

Namelist location : NAMFPC

NCAPEITER :

Definition : Number of iterations in the Newton's loops.

Scope : Integer.

Default value : 2

Namelist location : NAMCAPE

NETAPES :

Definition : Number of intermediate layers used for calculation of vertical ascent between two model pressure levels.

Scope : Integer.

Default value : 2

Namelist location : NAMCAPE

GCAPEPSD :

Definition : Depth of layer above the ground in which most unstable parcel is searched for (used with NFPCAPE=2 only).

Scope : Real. Unit: Pascal.

Default value : 30000 Pa.

Namelist location : NAMCAPE

GCAPERET :

Definition : Fraction of the condensate which is retained (ie: which does not precipitate).

Scope : real value between 0. and 1.

Default value : GCAPERET=0. ("irreversible" or pseudo-adiabatic moist ascent: clouds condensates precipitate instantaneously and thus does not affect the buoyancy).

Namelist location : NAMCAPE

(*i*) *Miscellaneous*

LFPQ :

Definition : To control the interpolation of relative versus specific humidity on height or *eta* levels. Relative humidity is considered to have better conservative properties through interpolations than mixing ratio, even if it is not a conservative quantity. If LFPQ=.FALSE. the relative humidity is interpolated then the specific humidity is deducted. If LFPQ=.TRUE. the specific humidity is interpolated then the relative humidity is deducted.

Scope : Boolean.

Default value : .FALSE. (this is the recommended value).

Namelist location : NAMFPC

RFPVCAP :

Definition : Minimum pressure of model level to provide an equatorial cap for fields computed on potential vorticity levels.

Scope : Real. Unit: Pascal.

Default value : if LECMWF=.TRUE. then RFPVCAP=8900. Pa; else RFPVCAP=15000. Pa

Namelist location : NAMFPC

NDLNPR :

Definition : Discretization of δ ($\ln p$). Set NDLNPR=1 to adopt the proper discretization to conform the non-hydrostatic model or whenever you post-process on "non-hydrostatic" field (pressure departure, vertical divergence or true vertical velocity). orticity levels.

Scope : Integer which value can be only 0 or 1.

Default value : 1

Namelist location : NAMDYN

D.3.2 Optimizing the performance

NPROMA :

Definition : working length of the model data rows. Refer to Section D.10.2 on page 138 for more information.

Scope : positive or negative integer but not zero nor a power of 2, and limited (in absolute value) to the biggest helpful value (ie: the number of model gridpoints in the current processor). When it is negative the absolute value is used; when it is positive the program will try to increase it in the limit of 10 % in an attempt to improve even more the optimization.

Default value : if LECMWF=.TRUE. then NPROMA=2047, else NPROMA=67.

Namelist location : NAMDIM

NFPROMAG :

Definition : working length of the post-processing data rows. Refer to Section D.10.2 on page 138 for more information.

Scope : positive integer but not zero nor a power of 2, and limited to the biggest helpful value (ie: the number of post-processing gridpoints in the current processor).

Default value : internally computed as the mean of the helpful values gathered among all processors.

Namelist location : NAMFPSC2

NFPROMEL :

Definition : working length of the post-processed extension zone data rows. Refer to Section D.10.2 on page 138 for more information.

Scope : positive integer but not zero nor a power of 2, and limited to the biggest helpful value (ie: the number of gridpoints in the post-processed extension zone of the current processor).

Default value : internally computed as the biggest helpful value.

Namelist location : NAMFPEZO

NPROC :

Definition : Number of processors used for the distribution per nodes.

Scope : Integer between 1 and the maximum number of processors of the machine.

Default value : 0 (So this parameter *must* be set explicitly!)

Namelist location : NAMPARO

LMPOFF :

Definition : Control of message passing libraries. Set LMPOFF=.TRUE. to avoid entering message passing subroutines when NPROC=1.

Scope : Boolean.

Default value : .FALSE.

Namelist location : NAMPARO

NPRTRW, NPRTRV :

Definition : Numbers of processors used respectively for the waves distribution and the vertical distribution in spectral space.

Scope : Integers greater than zero and so that NPRTRW*NPRTRV=NPROC. For the time being the vertical distribution is not working, so (NPRTRW,NPRTRV) must be (NPROC,1).

Default value : 0 (So these parameters *must* be set explicitly!)

Namelist location : NAMPARO

NPRGPNS, NPRGPEW :

Definition : Numbers of processors used respectively for the North–South and East–West gridpoint distributions.

Scope : Integers greater than zero and so that NPRGPNS*NPRGPEW=NPROC.
For the time being the East–West distribution is not working in ARPEGE/ALADIN, so (NPRGPNS,NPRGPEW) must be (NPROC,1).

Default value : 0 (So these parameters *must* be set explicitly!)

Namelist location : NAMPARO

NSTRIN, NSTROUT :

Definition : Numbers of processors used respectively for unpacking input data from file and for packing output data to file.

Scope : Integers between 1 and NPROC. The best performance in ARPEGE/ALADIN is obtained with NSTRIN=NPROC and NSTROUT≈NPROC/2.

Default value : if LECMWF=.TRUE. then (NSTRIN,NSTROUT)=(1,0).
Else (NSTRINNSTROUT)=(NPROC,1).

Namelist location : NAMPAR1

NSTREFP :

Definition : Number of processors used for the distribution of the post-processed extension zone (for LAM outputs only).

Scope : Integer between 1 and NPROC.

Default value : 1

Namelist location : NAMFPEZO

LSPLIT :

Definition : Control of latitude row splitting. set LSPLIT=.TRUE. to improve the balance of distribution.

Scope : Boolean. This option does not work in ALADIN (LSPLIT must be .FALSE.).

Default value : .TRUE.

Namelist location : NAMPAR1

NFPXFLD :

Definition : Chunk size of global fields while gathering the post-processed distributed fields before writing out to output files. Refer to Section D.10.1 on page 138 for more information.

Scope : Integer greater than zero and limited to the biggest helpful value (ie: the number of post-processed fields).

Default value : internally computed as the biggest helpful value.

Namelist location : NAMFPIOS

D.3.3 Output fields conditioning

(*a*) *Horizontal representation of dynamic fields*

For any post-processed dynamic field it is possible to choose the horizontal representation (spectral or gridpoint), providing the field can be computed in both representation. This is independent from the representation of the field in the model. So it is a way to convert fields from spectral space to gridpoint space or vice-versa):

TFP_{*}%LLGP :

Definition : Horizontal representation of fields: .TRUE. for gridpoint, .FALSE. for spectral.

Scope : Boolean. "{*}" represents the field generic identificator (there is one variable per dynamic field).

Default value : Refer to Section D.6.1 on page 127 for upper air fields, and to Section (a) on page 128 for 2D fields.

Namelist location : NAMAFN

LFITS :

Definition : Spectral fit of post-processed fields on eta levels. This key is active *only* if CFPFMT='MODEL' (ie: spectral coefficients in output). Setting LFITS=.FALSE. enables to write out all upper air dynamic fields in gridpoints.

Scope : Boolean. This key is getting obsolescent.
Better use the individual keys TFP_{*}%LLGP.

Default value : .TRUE.

Namelist location : NAMFPC

(*b*) *Encoding data in output file*

NBITPG :

Definition : Default number of bits for packing fields.

Scope : Integer which value can be either -1, or any positive number between 1 and 64. If NBITPG=-1 then the default value is internally computed by the FA (File ARPEGE) software.

Default value : 24; if NBITPG=-1 the actual default value will be 16.

Namelist location : NAMFA

NSTRON :

Definition : Default threshold for the truncation beyond which the spectral fields are packed.

Scope : Integer which value can be either -1, or any positive number between 1 and the model truncation NSMAX.

Default value : 10; if NSTRON=-1 the actual default depends on the model truncation NSMAX.

Namelist location : NAMFA

NPULAP :

Definition : "Dolby exposant" for the packing of spectral fields.

Scope : Integer between -5 and +5.

Default value : 1

Namelist location : NAMFA

NB{*} :

Definition : Number of bits for packing physical fields and fluxes.

Scope : Integer. "{*}" represents the field generic identificator (there is one variable per field).

Default value : Refer to Section (b) on page 129. Notice: surface geopotential should not be packed in the model in order to keep consistency between spectral and gridpoint orography.

Namelist location : NAMAFN

TFP_{*}%IBITS :

Definition : Number of bits for packing dynamic fields.

Scope : Integer. "{*}" represents the field generic identificator (there is one variable per dynamic field).

Default value : Refer to Section D.6.1 on page 127 for upper air fields, and to Section (a) on page 128 for 2D fields. Notice: surface geopotential should not be packed in the model in order to keep consistency between spectral and gridpoint orography.

Namelist location : NAMAFN

NFPGRIB :

Definition : GRIBlevel for fields encoding in the post-processing ARPEGE/ALADIN files:

- NFPGRIB=0: no packing at all. This value has priority over the numbers of bits for packing.
- NFPGRIB=1: standard GRIBencoding.
- NFPGRIB=2: a modified GRIBencoding for ARPEGE/ALADIN files.

Refer to the documentation on the ARPEGE/ALADIN files for more information (available in French[5] or in English[6]).

Scope : Integer between 0 and 2.

Default value : 2

Namelist location : NAMFPC

(*c*) *Customized complexions*

NCADFORM :

Definition : Auto-documentation format for the ALADIN files: set NCADFORM=0 for the EGGX new style format and NCADFORM=1 for the EGGX old style format.

Scope : Integer which value can be only 0 or 1.

Default value : 0

Namelist location : NAMOPH

LFPRH100 :

Definition : Representation of relative humidity: set LFPRH100=.TRUE. to get a percentage rather than a ratio.

Scope : Boolean.

Default value : LFPRH100=LECMWF

Namelist location : NAMFPC

LFPLOSP :

Definition : Representation of surface pressure: set LFPLOSP=.TRUE. to fill surface pressure with its logarithm.

Scope : Boolean.

Default value : if LECMWF=.TRUE. then LFPLOSP=.FALSE.; else LFPLOSP=.FALSE. except for the so-called configurations ((e)e)927 (See Chapter D.4 on page 118).

Namelist location : NAMFPC

D.3.4 Selective namelists

In normal use, at each post-processing time step all the post-processing fields are written out at all post-processing levels and for all output (sub-)domains. However it is possible to specify a more selective list of fields to write out, by choosing for each field the exact list of post-processing levels, and for each post-processing level of each field the exact list of (sub-)domains.

This is achieved by filling a specific namelist file currently named the *selection file*. In the local script the selection file should write: "xxtDDDDHHMM" where DDDD, HH and MM specify respectively the day (on 4 digits), the hour (on 2 digits) and the minute (on 2 digits) of the forecast. Furthermore in the local script the working directory should contain a file named dirlst listing the content of the working directory (as generated by the command ls).

The selection files should contain the following namelist blocks:

[5] http://intra.cnrm.meteo.fr/gmod/modeles/Tech/fa/synopsis.html
[6] http://intra.cnrm.meteo.fr/gmod/modeles/Tech/fa/manual.html

(i) NAMFPPHY
(ii) NAMFPDY2
(iii) NAMFPDYP
(iv) NAMFPDYH
(v) NAMFPDYV
(vi) NAMFPDYT
(vii) NAMFPDYS

Finally the following variables should be documented:

CNPPATH :

Definition : directory where the selection files stand.

Scope : string of 120 characters.

Default value : blank string (no selection files).

Namelist location : NAMCTO in the namelist file.

CLPHY :

Definition : selected physical fields names.

Scope : array of 16 characters, maximum size: 328 items. All the selected fields should be present in the array CFPPHY.

Default value : blank string (no fields).

Namelist location : NAMFPPHY in the selection file.

CLDPHY :

Definition : selected subdomains for each selected physical field.

Scope : array of ((15 *(10 + 1))-1) characters. Maximum size: 328 items. It should contain for each selected physical field the list of selected subdomains separated with the character ":". All the selected subdomains should be present in the array CFPDOM.

Default value : blank string (ALL subdomains).

Namelist location : NAMFPPHY in the selection file.

CLCFU :

Definition : selected cumulated fluxes names.

Scope : array of 16 characters, maximum size: 63 items. All the selected fields should be present in the array CFPCFU.

Default value : blank string (no fields).

Namelist location : NAMFPPHY in the selection file.

CLDCFU :

Definition : selected subdomains for each selected cumulated flux.

Scope : array of ((15 *(10 + 1))-1) characters. Maximum size: 63 items. It should contain for each selected cumulated flux the list of selected subdomains separated with the character ":". All the selected subdomains should be present in the array CFPDOM.

Default value : blank string (ALL subdomains).

Namelist location : NAMFPPHY in the selection file.

CLXFU :

Definition : selected instantaneous fluxes names.

Scope : array of 16 characters, maximum size: 63 items. All the selected fields should be present in the array CFPXFU.

Default value : blank string (no fields).

Namelist location : NAMFPPHY in the selection file.

CLDXFU :

Definition : selected subdomains for each selected instantaneous flux.

Scope : array of ((15 *(10 + 1))-1) characters. Maximum size: 63 items. It should contain for each selected instantaneous flux the list of selected subdomains separated with the character ":". All the selected subdomains should be present in the array CFPDOM.

Default value : blank string (ALL subdomains).

Namelist location : NAMFPPHY in the selection file.

CL2DF :

Definition : selected dynamic 2D fields names.

Scope : array of 16 characters, maximum size: 78 items. All the selected fields should be present in the array CFP2DF.

Default value : blank string (no fields).

Namelist location : NAMFPDY2 in the selection file.

CLD2DF :

Definition : selected subdomains for each selected dynamic 2D field.

Scope : array of ((15 *(10 + 1))-1) characters. Maximum size: 78 items. It should contain for each selected dynamic 2D field the list of selected subdomains separated with the character ":". All the selected subdomains should be present in the array CFPDOM.

Default value : blank string (ALL subdomains).

Namelist location : NAMFPDY2 in the selection file.

CL3DF :

Definition : selected upper air dynamic fields names.

Scope : array of 12 characters, maximum size: 98 items. All the selected fields should be present in the array CFP3DF.

Default value : blank string (no fields).

Namelist location : NAMFPDYP for pressure levels, NAMFPDYH for height levels, NAMFPDYV for potential vorticity levels, NAMFPDYT for isentropic levels and NAMFPDYS for *eta* levels. All in the selection file.

IL3DF :

Definition : the *subscripts* of the selected post-processing levels for each selected upper air dynamic field.

Scope : integer array of strictly positive values, maximum size: 98 items. All the selected subscripts should correspond to an effective post-processing level.

Default value : 0

Namelist location : NAMFPDYP for pressure levels, NAMFPDYH for height levels, NAMFPDYV for potential vorticity levels, NAMFPDYT for isentropic levels and NAMFPDYS for *eta* levels. All in the selection file.

CLD3DF :

Definition : selected subdomains for each selected level of each selected upper air dynamic field.

Scope : bi-dimensional array of ((15 *(10 + 1))-1) characters. Maximum size: (200 , 78) items. It should contain for each selected level of each selected upper air dynamic field the list of selected subdomains separated with the character ":". All the selected subdomains should be present in the array CFPDOM.

Default value : blank string (ALL subdomains).

Namelist location : NAMFPDYP for pressure levels, NAMFPDYH for height levels, NAMFPDYV for potential vorticity levels, NAMFPDYT for isentropic levels and NAMFPDYS for *eta* levels. All in the selection file.

Section D.7 on page 133 shows an example of selection file.

D.3.5 Miscellaneous

(*a*) *Customization of names*

CN{*} :

Definition : ARPEGE/ALADIN field names for each surface fields or fluxes.

Scope : String of 16 characters. "{*}" represents the field generic identificator (there is one variable per field).

Default value : Refer to Section (b) on page 129.

Namelist location : NAMAFN

TFP_{*}%CLNAME :

Definition : ARPEGE/ALADIN field names for dynamic fields.

Scope : String of 16 characters. "{*}" represents the field generic identificator (there is one variable per field). However the string length is limited to 12 characters for upper air fields.

Default value : Refer to Section D.6.1 on page 127 for upper air fields, and to Section (a) on page 128 for 2D fields.

Namelist location : NAMAFN

CFPDIR :

Definition : Prefix of the output files names.

Scope : String of 180 characters. for instance you can set a UNIXpath.

Default value : 'PF'

Namelist location : NAMFPC

LINC :

Definition : Control of the time stamp of the output files names: .TRUE. to write the stamp in hours, .FALSE. to write it in time steps.

Scope : Boolean.

Default value : .FALSE.

Namelist location : NAMOPH

(*b*) *Traceback*

LTRACEFP :

Definition : post-processing traceback: set LTRACEFP=.TRUE. to get more information printed out on the listing (for debugging purpose). This option is coupled with the variable NPRINTLEV.

Scope : Boolean.

Default value : .FALSE.

Namelist location : NAMFPC

NPRINTLEV :

Definition : verbose option for the listing.

Scope : Integer between 0 (minimum prints) and 2 (maximum prints).

Default value : 0

Namelist location : NAMCTO

LFPNORM :

Definition : Control of the norms of the output fields (mean, minimum and maximum value for each field and each (sub-)domain).

Scope : Boolean.

Default value : .TRUE.

Namelist location : NAMFPC

LRFILAF :

Definition : verbose option to control the content of any ARPEGE/ALADIN files used. Set LRFILAF=.TRUE. to get the content of the files at each I/O operation.

Scope : Boolean.

Default value : .TRUE.

Namelist location : NAMCT1

D.4 THE FAMILY OF CONFIGURATIONS 927

D.4.1 What it is

The "configuration 927" is the way how to use FULLPOS to change the geometry and/or the resolution of a history spectral file. Actually, it is not a true configuration of the software ARPEGE/IFS/ALADIN, since the parameter NCONF should remain equal to 1; let us rather call it a configuration of the post-processing. In such configuration the horizontal interpolations are performed systematically before the vertical interpolations, and the dynamic variables are (usually) written out as spectral coefficients in the target spectral geometry[7].

As shown in the fancy picture D.1 on page 118, gobbleenv below,

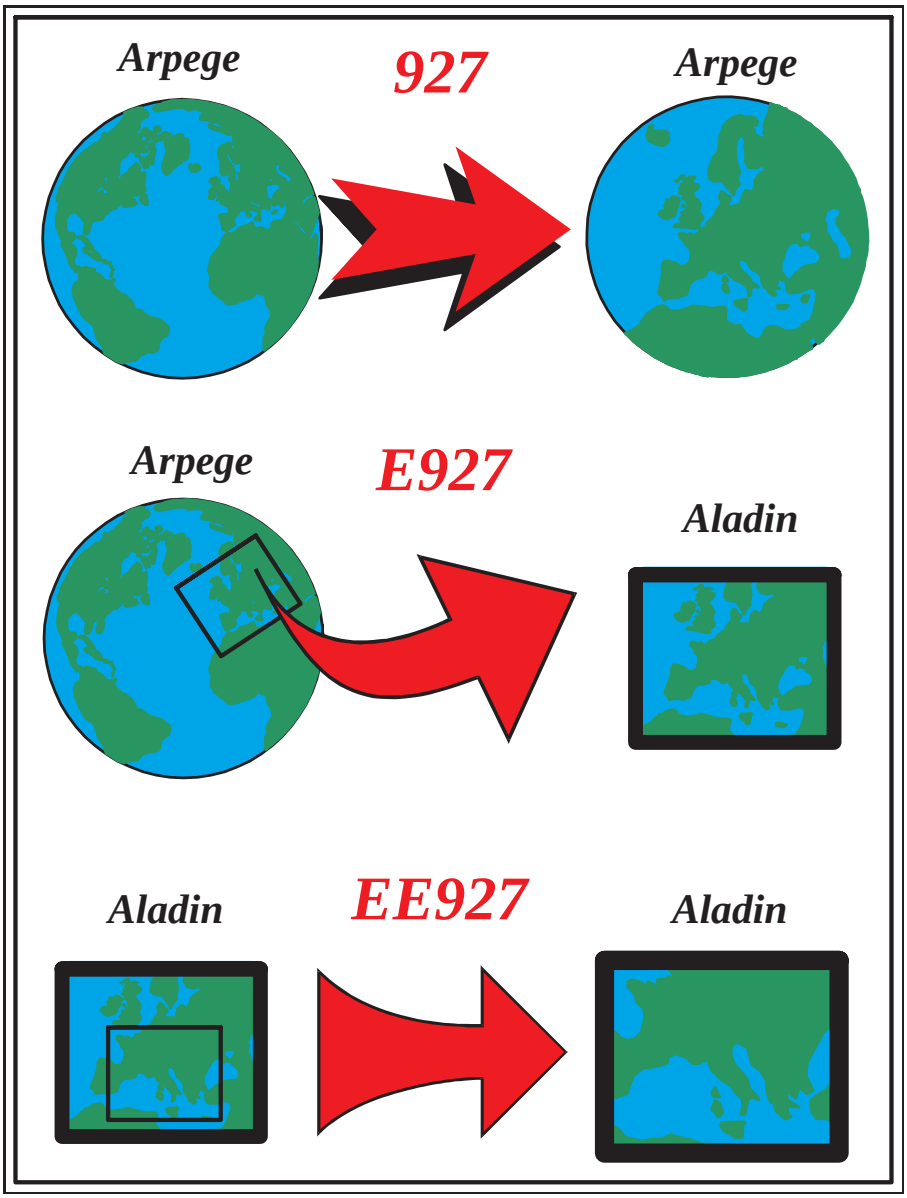

Figure D.1 *The configuration 927, E927 and EE927.*

- Configuration "927" is to make a file ARPEGE, starting from a file ARPEGE (mostly used to change the resolution, the stretching and the pole of stretching in the 4D-Var suite).

[7]It is the change of spectral geometry which makes this configuration so special in the context of the software state.

- Configuration "E927" is to make a file ALADIN, starting from a file ARPEGE (for coupling ALADIN to ARPEGE).
- Configuration "EE927" is to make a file ALADIN, starting from a file ALADIN (for ALADIN nesting).

D.4.2 How it works

The configurations 927 are working only off-line[8].

Such "configurations" are activated through a specific key:

LFPSPEC :

Definition : Control of the configuration 927. Set LFPSPEC=.TRUE. to activate the process.

Scope : Boolean.

Default value : .FALSE.

Namelist location : NAMFPC

Notice:

- To run the configuration 927 (ARPEGEto ARPEGE) you have to run the *model* ARPEGE.
- To run the configuration E927 (ARPEGEto ALADIN) you have to run the *model* ARPEGE (setting LELAM=.FALSE. or -m arpifs in command line) with the *software* ALADIN.
- To run the configuration EE927 (ALADINto ALADIN) you have to run the *model* ALADIN.

Warning! The configurations 927 create a working file named "ncf927". If your script contains executions of configurations 927 inside a loop, then this file should be deleted before the beginning of each iteration.

D.4.3 Namelists parameters

The recommended namelists parameters to set for the configuration 927 are the following:

```
&NAMCTO
   LFPOS=.T.,
   NPRINTLEV=1, (verboosity)
   NOPGMR=0, LSIDG=.F., (memory savings)
   NSPPR=0, (CPU savings)
/END
&NAMCT1
   N1HIS=0, (no history file in output)
   LRFILAF=.F., (I/O savings)
/END
&NAMINI
   NEINI=0, (no initialization on input data)
/END
&NAMFA
  NSTRON=-1, NBITPG=16, (proper file encoding)
/END
&NAMAFN (Let this namelist empty)
/END
&NAMFPC
   LTRACEFP=.TRUE.,
   LFPSPEC=.T.,
   CFPFMT='GAUSS',
   NFPCLI=3,
```

[8] Out of the direct model integration.

```
   LFPMOIS=.FALSE.,
   CFP3DF(1)='TEMPERATURE',
   CFP3DF(2)='FONC.COURANT',
   CFP3DF(3)='POT.VITESSE',
   CFP3DF(4)='HUMI.SPECIFIQUE',
   CFP2DF(1)='SURFPRESSION',
   CFP2DF(2)='SPECSURFGEOPOTENTIEL',
   CFPPHY(1)='SURFTEMPERATURE',
   CFPPHY(2)='PROFTEMPERATURE ',
   CFPPHY(3)='PROFRESERV.EAU ',
   CFPPHY(4)='SURFRESERV.NEIGE',
   CFPPHY(5)='SURFRESERV.EAU ',
   CFPPHY(6)='SURFZO.FOIS.G ',
   CFPPHY(7)='SURFALBEDO ',
   CFPPHY(8)='SURFEMISSIVITE ',
   CFPPHY(9)='SURFET.GEOPOTENT',
   CFPPHY(10)='SURFIND.TERREMER',
   CFPPHY(11)='SURFPROP.VEGETAT',
   CFPPHY(12)='SURFVAR.GEOP.ANI',
   CFPPHY(13)='SURFVAR.GEOP.DIR',
   CFPPHY(14)='SURFIND.VEG.DOMI',
   CFPPHY(15)='SURFRESI.STO.MIN',
   CFPPHY(16)='SURFPROP.ARGILE',
   CFPPHY(17)='SURFPROP.SABLE',
   CFPPHY(18)='SURFEPAIS.SOL',
   CFPPHY(19)='SURFIND.FOLIAIRE',
   CFPPHY(20)='SURFRES.EVAPOTRA',
   CFPPHY(21)='SURFGZO.THERM',
   CFPPHY(22)='SURFRESERV.INTER',
   CFPPHY(23)='PROFRESERV.GLACE',
   CFPPHY(24)='SURFRESERV.GLACE',
   NRFP3S=1,2,3,4,5,6,7,8,9,10,11,12, ... (fill it up to NFPLEV)
/END
&NAMFPD
   NLAT= (fill it yourself)
   NLON= (fill it yourself)
/END
&NAMFPG
   FPMUCEN= (fill it yourself)
   FPLOCEN= (fill it yourself)
   NFPHTYP= (fill it yourself)
   NFPRGRI= (fill it yourself if NFPHTYP=2)
   FPSTRET= (fill it yourself)
   NFPTTYP= (fill it yourself)
   NFPMAX= (fill it yourself)
   NFPLEV= (fill it yourself)
   FPVALH= (fill it yourself)
   FPVBH= (fill it yourself)
/END
```

The recommended namelists parameters to set for the configuration E927 or EE927 are the following:

```
&NAMCTO
   LFPOS=.T.,
   NPRINTLEV=1, (verboosity)
```

```
   NOPGMR=0, LSIDG=.F., (memory savings)
   NSPPR=0, (CPU savings)
/END
&NAMCT1
   N1HIS=0, (no history file in output)
   LRFILAF=.F., (I/O savings)
/END
&NAMINI
   NEINI=0, (no initialization on input data)
/END
&NAMFA
  NSTRON=-1, NBITPG=18, (proper file encoding)
/END
&NAMAFN
   TFP_U%CLNAME='WIND.U.PHYS',
   TFP_V%CLNAME='WIND.V.PHYS',
/END
&NAMFPC
   LTRACEFP=.TRUE.,
   LFPSPEC=.T.,
   CFPFMT='GAUSS',
   NFPCLI=3,
   LFPMOIS=.FALSE.,
   CFP3DF(1)='TEMPERATURE',
   CFP3DF(2)='FONC.COURANT',
   CFP3DF(3)='POT.VITESSE',
   CFP3DF(4)='HUMI.SPECIFIQUE',
   CFP2DF(1)='SURFPRESSION',
   CFP2DF(2)='SPECSURFGEOPOTENTIEL',
   CFPPHY(1)='SURFTEMPERATURE',
   CFPPHY(2)='PROFTEMPERATURE ',
   CFPPHY(3)='PROFRESERV.EAU ',
   CFPPHY(4)='SURFRESERV.NEIGE',
   CFPPHY(5)='SURFRESERV.EAU ',
   CFPPHY(6)='SURFZO.FOIS.G ',
   CFPPHY(7)='SURFALBEDO ',
   CFPPHY(8)='SURFEMISSIVITE ',
   CFPPHY(9)='SURFET.GEOPOTENT',
   CFPPHY(10)='SURFIND.TERREMER',
   CFPPHY(11)='SURFPROP.VEGETAT',
   CFPPHY(12)='SURFVAR.GEOP.ANI',
   CFPPHY(13)='SURFVAR.GEOP.DIR',
   CFPPHY(14)='SURFIND.VEG.DOMI',
   CFPPHY(15)='SURFRESI.STO.MIN',
   CFPPHY(16)='SURFPROP.ARGILE',
   CFPPHY(17)='SURFPROP.SABLE',
   CFPPHY(18)='SURFEPAIS.SOL',
   CFPPHY(19)='SURFIND.FOLIAIRE',
   CFPPHY(20)='SURFRES.EVAPOTRA',
   CFPPHY(21)='SURFGZO.THERM',
   CFPPHY(22)='SURFRESERV.INTER',
   CFPPHY(23)='PROFRESERV.GLACE',
   CFPPHY(24)='SURFRESERV.GLACE',
   NRFP3S=1,2,3,4,5,6,7,8,9,10,11,12, ... (fill it up to NFPLEV)
/END
```

```
&NAMFPD
   NLAT= (fill it yourself)
   NLON= (fill it yourself)
   RLATC= (fill it yourself)
   RLONC= (fill it yourself)
   RDELX= (fill it yourself)
   RDELY= (fill it yourself)
   NFPLUX= (fill it yourself)
   NFPGUX= (fill it yourself)
/END
&NAMFPG
   FPLONO= (fill it yourself)
   FPLATO= (fill it yourself)
   NFPMAX= (fill it yourself)
   NMFPMAX= (fill it yourself)
   NFPLEV= (fill it yourself)
   FPVALH= (fill it yourself)
   FPVBH= (fill it yourself)
/END
```

Furthermore, if you intend to make a non-hydrostatic history file, you should add the following parameters:

```
&NAMCTO
  LNHDYN=.TRUE. or .FALSE. (depending whether your input file is hydrostatic or not)
/END
&NAMDYN
   NDLNPR=1,
/END
&NAMFPC
   CFP3DF(5)='PRESS.DEPART',
   CFP3DF(6)='VERTIC.DIVER',
/END
```

D.4.4 Bogussing

A procedure has been developed in order to try and improve the forecast of tropical cyclone in ARPEGE/ALADIN: it is called "bogussing", or "configuration 927E". This configuration is working in 3 steps:

(i) Bogussing of ALADIN a configuration EE927 is run in adiabatic mode with translation activated to move the model cyclone (actually the minimum of surface pressure in the model) to the observed location (refer to NFPTTYP, FPMUCEN and FPLOCEN). In order not to translate the orography, one should first lower the orography to zero, then translate, and finally re-set the original orography.

(ii) ARPEGE background: this is a file ARPEGE which should contain the fields of a given ARPEGE history file, all in gridpoint representation. Furthermore the surface pressure should be the true one, not its logarithm. This file aims to be used for the third step:

(iii) Bogussing of ARPEGE this configuration is a kind of "reverse configuration E927": starting from the ARPEGE background file and the ALADIN bogussed file, a new ARPEGE file is build, containing the local translation of fields in the vicinity of the tropical cyclone.

To run this configuration 927E (ALADIN to ARPEGE) you have to run the model ALADIN (setting LELAM=.TRUE. or `-m aladin` in command line) with the namelist of a configuration 927 in adiabatic mode and with the incremental process as described below:

NFPINCR :

Definition : Control of incremental post-processing. Set NFPINCR=1 to activate the incremental process.

Scope : Integer which value can be only 0 or 1.

Default value : 0

Namelist location : NAMFPC

You will have also to provide 3 input files:

ELSCF${CNMEXP(1:4)}ALBC : the ALADIN file before bogussing

ICMSH${CNMEXP(1:4)}INIT : the ALADIN bogussed file

BGPX${CNMEXP(1:4)}${CFPDOM} : the ARPEGE background file

Remark: this "incremental" process can be considered like the "tangent linear post-processing of the poor", as it does not really works on increments.

D.5 EXPERT USAGE

Once you have a good knowledge of FULLPOS, you can tune various parameters of namelists as you wish, combine scripts, and even modify the code.

This section will shortly describe some examples of clever use of the software.

D.5.1 Appending fields to a file

Imagine you wish to post-process a given field on a thousand pressure levels: the software will fail because the maximum number of output levels is limited to a reasonable value.

However you can easily overcome this limitation by slicing the list of post-processing levels: that way you would submit a bunch of jobs, targeting the same output file. Since the output file is not sequential but indexed-sequential, the file will not be overwritten at the beginning of each job: instead the fields will be appended to one another.

You can also use this trick to append fields to your own input file: to do that you just have to copy your input file to the output file before starting the post-processing job.

D.5.2 Derivatives on model levels

If you try to postprocess derivatives on *eta* levels (like the potential vorticity on the model levels) and you do interpolate on the horizontal (for instance: from a file ARPEGE to a file ALADIN), the software will fail because derivatives will be missing: this is because the horizontal derivatives are available only in the model geometry.

A way to overcome this limitation is to first change the geometry of your input file to the geometry of your output file (using the configurations 927), then to post-process on the new "model" grid (`CFPFMT='MODEL'` and `LFITS=.FALSE.`).

Unfortunately this does not work if the target geometry is LAT × LON! In this case you have to trick the software so that the field you wish to interpolate will be considered as a passive scalar field; this can be achieved in two steps:

(i) You should create a history file with the supplementary fields you wish to interpolate; this can be achieved either by running a configuration of the kind "927" in which namelist you will request the supplementary fields, or by running a normal post-processing job in the model geometry (`CFPFMT='MODEL'`) and using the "appending fields" trick (refer to the previous section).

(ii) If they are spectral you can post-process your supplementary fields as model passive scalar fields (setting `NFPASS` and the field descriptors `TFP_SCVA()`). Else you can still trick the software by activating the pronostic field for gridpoint cloud fraction (setting `LGPA=.TRUE.`) and feeding the cloud fraction with one of your supplementary field through a proper setting of `TFP_CLF%CLNAME`. Notice: this is possible only because there is — by "chance"! — no control of the interpolations overshoot for cloud fraction. (In principle the interpolated cloud fraction should be controlled in order to remain between 0. and 1.)

D.5.3 3D physical fluxes

Fluxes are not yet post-processable as 3D fields. However you can post-process them in off-line mode[9] by activating the pronostic field for gridpoint cloud fraction (setting `LGPA=.TRUE.`) and feeding the cloud fraction with one of them through a proper setting of `TFP_CLF%CLNAME`. Notice: this is possible only because there is — by "chance"! — no control of the interpolations overshoot for cloud fraction. (In principle the interpolated cloud fraction should be controlled in order to remain between 0. and 1.)

[9] Out of the direct model integration.

D.5.4 Free-use fields

FULLPOS provides the environment to post-process your personal fields once you have created them in the software. This may be done with a minimum of modifications in the software. The environment should be documented through the following namelists variables:

CNPFSU :

Definition : Generic for surface physical free-use fields.

Scope : array of 16 characters; maximum size: 15 items.

Default value : Refer to Section (b) on page 129.

Namelist location : NAMAFN

NBFSU :

Definition : Number of bits for packing surface physical free-use fields.

Scope : Integer array; maximum size: 15 items.

Default value : Refer to Section (b) on page 129.

Namelist location : NAMAFN

TFP_FUA%CLNAME :

Definition : Dynamic upper air free-use fields names.

Scope : array of 16 characters;; maximum size: 30 items.

Default value : Refer to Section D.6.1 on page 127.

Namelist location : NAMAFN

TFP_FUA%IBITS :

Definition : Number of bits for packing dynamic upper air free-use fields.

Scope : Integer array; maximum size: 30 items.

Default value : Refer to Section D.6.1 on page 127.

Namelist location : NAMAFN

TFP_FUA%LLGP :

Definition : Control of the horizontal representation for dynamic upper air free-use fields: `.TRUE.` for gridpoint representation; `.FALSE.` for spectral representation.

Scope : Boolean array; maximum size: 30 items.

Default value : Refer to Section D.6.1 on page 127.

Namelist location : NAMAFN

TFP_FSU%CLNAME :

Definition : Dynamic surface free-use fields names.

Scope : array of 16 characters;; maximum size: 15 items.

Default value : Refer to Section (a) on page 128.

Namelist location : NAMAFN

TFP_FSU%IBITS :

Definition : Number of bits for packing dynamic surface free-use fields.

Scope : Integer array; maximum size: 15 items.

Default value : Refer to Section (a) on page 128.

Namelist location : NAMAFN

TFP_FSU%LLGP :

Definition : Control of the horizontal representation for dynamic surface free-use fields .TRUE. for gridpoint representation; .FALSE. for spectral representation.

Scope : Boolean array; maximum size: 15 items.

Default value : Refer to Section (a) on page 128.

Namelist location : NAMAFN

Dynamic fields should then be computed in the subroutines POS (for interpolations on pressure levels, isentropic levels or PV levels) or ENDPOS (for interpolations on height or *eta* levels).

You can possibly control the result of the horizontal interpolations in the subroutine FPCORDYN.

The fields will be treated as fitable non-derivatives: in other words they will be concerned by the keys LFITP, LFITV, LFITT, LFITS and LFIT2D.

D.6 FIELD DESCRIPTORS

D.6.1 Upper air dynamic fields descriptors

This section details the content of a part of the namelist NAMAFN which contains the descriptors of the upper air dynamic fields. The descriptor %CLNAME serves to fill the array CFP3DF in the namelist NAMFPC.

Field	TYPE NAME	%CLNAME	%IBITS	%LLGP
Absolute Vorticity.....	TFP_ABS	ABS_VORTICIT	24	F
Atmospheric liquid water..	TFP_W	LIQUID_WATER	24	T
Atmospheric solid water...	TFP_S	SOLID_WATER	24	T
Cloud fraction.....	TFP_CLF	CLOUD_FRACTI	24	T
Divergence.....	TFP_DIV	DIVERGENCE	24	F
Equiv. pot. temperature...	TFP_ETH	THETA_EQUIVA	24	F
Free upper air field n 01.	TFP_FUA(01)	UPPER_AIR.01	24	F
Free upper air field n 02.	TFP_FUA(02)	UPPER_AIR.02	24	F
Free upper air field n 03.	TFP_FUA(03)	UPPER_AIR.03	24	F
	(truncated list - 30 variables)			
Geopotential.....	TFP_Z	GEOPOTENTIEL	24	F
Montgomery potential.....	TFP_MG	MONTGOMERY G	24	F
Ozone.....	TFP_O3MX	OZONE	24	F
Passive scalar nr 01.....	TFP_SCVA(01)	#001.SCALAR	24	F
Passive scalar nr 02.....	TFP_SCVA(02)	#002.SCALAR	24	F
Passive scalar nr 03.....	TFP_SCVA(03)	#003.SCALAR	24	F
	(truncated list - 5 variables)			
Potential temperature.....	TFP_TH	TEMPE_POTENT	24	F
Potential Vorticity.....	TFP_PV	POT_VORTICIT	24	F
Pressure Departure.....	TFP_PD	PRESS.DEPART	24	F
Pressure.....	TFP_P	PRESSURE	24	F
Pseudo Vertic. Divergence.	TFP_VD	VERTIC.DIVER	24	F
Relative humidity.....	TFP_HU	HUMI_RELATIV	24	F
Shearing Deformation.....	TFP_SHD	SHEAR_DEFORM	24	F
Specific humidity.....	TFP_Q	HUMI.SPECIFI	24	F
Stream function.....	TFP_KHI	FONC.COURANT	24	F
Stretching Deformation....	TFP_STD	STRET_DEFORM	24	F
Temperature.....	TFP_T	TEMPERATURE	24	F
True Vertical NH Velocity.	TFP_VW	VERT.VELOCIT	24	F
U-momentum of wind.....	TFP_U	VENT_ZONAL	24	F
Velocity potential.....	TFP_PSI	POT.VITESSE	24	F
Vertical velocity.....	TFP_VV	VITESSE_VERT	24	F
Vorticity.....	TFP_VOR	VORTICITY	24	F
V-momentum of wind.....	TFP_V	VENT_MERIDIE	24	F
Wet bulb pot. temperature.	TFP_THPW	THETA_PRIM_W	24	F
Wind velocity.....	TFP_WND	WIND_VELOCIT	24	F

Notice: Vertical velocity “omega” is expressed in Pa/s and true vertical velocity “w” is expressed in m/s

(*a*) *2D dynamic fields descriptors*

This section details the content of a part of the namelist NAMAFN which contains the descriptors of the 2D dynamic fields. The descriptor %CLNAME serves to fill the array CFP2DF in the namelist NAMFPC.

```
   Field                  :    TYPE NAME    %CLNAME      %IBITS %LLGP
Altitude of iso-t=0 ......:     TFP_HT0B SURFISOT0.MALTIT   24     T
Altitude of iso-tprimw=0 .:     TFP_HTPW SURFISOTPW0.MALT   24     T
CAPE......................:     TFP_CAPE SURFCAPE.POS.F00   24     T
CIEN......................:     TFP_CIEN SURFCIEN.POS.F00   24     T
Free surface field nr 01..:  TFP_FSU(01) SURF2D.01          24     F
Free surface field nr 02..:  TFP_FSU(02) SURF2D.02          24     F
Free surface field nr 03..:  TFP_FSU(03) SURF2D.03          24     F
                            (truncated list - 15 variables)
HU cls....................:     TFP_RCLS CLSHU.RELATI.POS   24     T
ICAO jet pressure.........:     TFP_PJET JETPRESSURE        24     T
ICAO Tropopause pressure..:     TFP_PCAO ICAOTROP.PRESSUR   24     T
ICAO Tropo. temperature...:     TFP_TCAP ICAOTROP.TEMPERA   24     T
Log. of Surface pressure..:     TFP_LNSP LOG.SURF.PRESS     24     F
Map factor................:     TFP_GM   MAP_FACTOR        24     T
Maxi. rel. moist. in cls..:     TFP_HUX  CLSHUREL.MAX.POS   24     T
Maxi. temperature in cls..:     TFP_TX   CLSTEMPE.MAX.POS   24     T
Mean sea level pressure...:     TFP_MSL  MSLPRESSURE        24     F
Mini. rel. moist. in cls..:     TFP_HUN  CLSHUREL.MIN.POS   24     T
Mini. temperature in cls..:     TFP_TN   CLSTEMPE.MIN.POS   24     T
Module of gusts...........:     TFP_FGST CLSRAFALES.POS     24     T
Module of wind cls........:     TFP_FCLS CLSWIND_VELO.POS   24     T
Pressure of iso-t=0 ......:     TFP_PT0B SURFISOT0.PRESSU   24     T
Q cls.....................:     TFP_QCLS CLSHU.SPECIF.POS   24     T
Surface geopotential......:     TFP_FIS  SPECSURFGEOPOTEN   64     F
Surface pressure..........:     TFP_SP   SURFPRESSION       24     F
Surface Vertical Velocity.:     TFP_WWS  SURFVERT.VELOCIT   24     F
T cls.....................:     TFP_TCLS CLSTEMPERATU.POS   24     T
Total water vapour........:     TFP_TWV  SURFTOT.WAT.VAPO   24     T
Tropo. Folding Indicator..:     TFP_FOL  TROPO_FOLD_INDIC   24     T
U cls.....................:     TFP_UCLS CLSVENT_ZONA.POS   24     T
U gusts...................:     TFP_UGST CLSURAFALES.POS    24     T
U-momentum of ICAO jet....:     TFP_UJET JETVENT_ZONAL      24     T
V cls.....................:     TFP_VCLS CLSVENT_MERI.POS   24     T
V gusts...................:     TFP_VGST CLSVRAFALES.POS    24     T
V-momentum of ICAO jet....:     TFP_VJET JETVENT_MERIDIEN   24     T
```

(*b*) *Surface physical fields descriptors*

This section details the content of a part of the namelist NAMAFN which contains the descriptors of the surface physical fields. The descriptor %CLNAME serves to fill the array CFPPHY in the namelist NAMFPC.

```
Albedo ............................... CNAL   = SURFALBEDO          NBAL   = 24
Analysed RMS of geopotential ........  CNPCAAG= SURFETA.GEOPOTEN    NBPCAAG= 24
Anisotropy coeff. of topography ...... CNACOT = SURFVAR.GEOP.ANI    NBACOT = 24
Clim. relative deep soil wetness ..... CNCDSW = PROFPROP.RMAX.EA    NBCDSW = 24
Clim. relative surface soil wetness .. CNCSSW = SURFPROP.RMAX.EA    NBCSSW = 24
Deep soil temperature ................ CNDST  = PROFTEMPERATURE     NBDST  = 24
Deep soil wetness .................... CNDSW  = PROFRESERV.EAU      NBDSW  = 24
Direction of main axis of topography . CNDPAT = SURFVAR.GEOP.DIR    NBDPAT = 24
Emissivity ........................... CNEMIS = SURFEMISSIVITE      NBEMIS = 24
Forecasted RMS of geopotential ....... CNPCAPG= SURFETP.GEOPOTEN    NBPCAPG= 24
Frozen deep soil wetness ............. CNFDSW = PROFRESERV.GLACE    NBFDSW = 24
Frozen superficial soil wetness ...... CNFSSW = SURFRESERV.GLACE    NBFSSW = 24
Index of vegetation .................. CNIVEG = SURFIND.VEG.DOMI    NBIVEG = 24
Interception content ................. CNIC   = SURFRESERV.INTER    NBIC   = 24
INTERPOLATED surface temperature ..... CNRDST = INTSURFTEMPERATU    NBRDST = 24
Land/sea mask ........................ CNLSM  = SURFIND.TERREMER    NBLSM  = 24
Leaf area index ...................... CNLAI  = SURFIND.FOLIAIRE    NBLAI  = 24
OUTPUT Grid-point geopotential ....... CNGFIS = SURFGEOPOTENTIEL    NBGFIS = 64
Percentage of clay within soil ....... CNARG  = SURFPROP.ARGILE     NBARG  = 24
Percentage of land ................... CNLAN  = SURFPROP.TERRE      NBLAN  = 24
Percentage of sand within soil ....... CNSAB  = SURFPROP.SABLE      NBSAB  = 24
Percentage of vegetation ............. CNVEG  = SURFPROP.VEGETAT    NBVEG  = 24
Relaxation deep soil wetness ......... CNRDSW = RELAPROP.RMAX.EA    NBRDSW = 24
Resistance to evapotranspiration ..... CNHV   = SURFRES.EVAPOTRA    NBHV   = 24
Roughness length of bare surface
(times g)............................. CNBSR  = SURFZOREL.FOIS.G     NBBSR  = 24
Snow albedo .......................... CNALSN = SURFALBEDO NEIGE    NBALSN = 24
Surface snow density ................. CNSNDE = SURFDENSIT.NEIGE    NBSNDE = 24
Snow depth ........................... CNSD   = SURFRESERV.NEIGE    NBSD   = 24
Soil depth ........................... CND2   = SURFEPAIS.SOL       NBD2   = 24
Standart deviation of orography
(times g) ............................ CNSDOG = SURFET.GEOPOTENT     NBSDOG = 24
Stomatal minimum resistance .......... CNRSMIN= SURFRESI.STO.MIN    NBRSMIN= 24
Surface albedo for non snowed areas .. CNBAAL = SURFALBEDO.COMPL    NBBAAL = 24
Surface relative moisture ............ CNPSRHU= SURFHUMI.RELATIV    NBPSRHU= 24
Surface roughness (times g) .......... CNSR   = SURFZ0.FOIS.G       NBSR   = 24
Surface soil wetness ................. CNSSW  = SURFRESERV.EAU      NBSSW  = 24
Surface temperature .................. CNST   = SURFTEMPERATURE     NBST   = 24
Thermal roughness length (times g) ... CNZ0H  = SURFGZ0.THERM      NBZ0H  = 24
U-momentum of vector anisotropy ...... CNPADOU= SURF.U.ANISO.DIR   NBPADOU= 24
V-momentum of vector anisotropy ...... CNPADOV= SURF.V.ANISO.DIR   NBPADOV= 24
Free field #01 ....................... CNPFSU = SURFFREE.FIELD01    NBFSU  = 24
Free field #02 ....................... CNPFSU = SURFFREE.FIELD02    NBFSU  = 24
                            (truncated list - 15 variables)
```

(c) *Cumulated fluxes descriptors*

This section details the content of a part of the namelist NAMAFN which contains the descriptors of the cumulated fluxes. The descriptor %CLNAME serves to fill the array CFPCFU in the namelist NAMFPC.

```
Boundary Layer Dissipation ..... CNCBLD = SURFDISSIP SURF    NBCBLD = 24
Clear sky longwave radiative flux .... CNCTHC = SURFRAYT THER CL   NBCTHC = 24
Clear sky shortwave radiative flux ... CNCSOC = SURFRAYT SOL CL    NBCSOC = 24
Contribution of Convection to Cp.T ... CNCCVS = SURFCFU.CT.CONVE   NBCCVS = 24
Contribution of Convection to Q ...... CNCCVQ = SURFCFU.Q.CONVEC   NBCCVQ = 24
Contribution of Convection to U ...... CNCCVU = SURFTENS.CONV.ZO   NBCCVU = 24
Contribution of Convection to V ...... CNCCVV = SURFTENS.CONV.ME   NBCCVV = 24
Contribution of Turbulence to Cp.T ... CNCTUS = SURFCFU.CT.TURBU   NBCTUS = 24
Contribution of Turbulence to Q ...... CNCTUQ = SURFCFU.Q.TURBUL   NBCTUQ = 24
Convective Cloud Cover ............... CNCCCC = ATMONEBUL.CONVEC   NBCCCC = 24
Convective precipitation ............. CNCCP  = SURFPREC.EAU.CON   NBCCP  = 24
Convective Snow Fall ................. CNCCSF = SURFPREC.NEI.CON   NBCCSF = 24
Deep soil water content run-off ...... CNCDRU = PROFRUISSELLEMEN   NBCDRU = 24
Duration of total precipitations ..... CNCDUTP= SURFTIME.PREC.TO   NBCDUTP= 24
Evapotranspiration ................... CNCETP = SURFEVAPOTRANSPI   NBCETP = 24
Flux d eau dans le sol ............... CNCEAS = SURFEAU DANS SOL   NBCEAS = 24
Flux de chaleur dans le sol .......... CNCCHS = SURFCHAL. DS SOL   NBCCHS = 24
High Cloud Cover ..................... CNCHCC = ATMONEBUL.HAUTE    NBCHCC = 24
Interception water content run-off.... CNCIRU = SURFRUISS. INTER   NBCIRU = 24
Large Scale Precipitation ............ CNCLSP = SURFPREC.EAU.GEC   NBCLSP = 24
Large Scale Snow fall ................ CNCLSS = SURFPREC.NEI.GEC   NBCLSS = 24
Latent Heat Evaporation .............. CNCLHE = SURFFLU.LAT.MEVA   NBCLHE = 24
Latent Heat Sublimation .............. CNCLHS = SURFFLU.LAT.MSUB   NBCLHS = 24
Liquid specific moisture ............. CNCLI  = ATMOHUMI LIQUIDE   NBCLI  = 24
Low Cloud Cover ...................... CNCLCC = ATMONEBUL.BASSE    NBCLCC = 24
Medium Cloud Cover ................... CNCMCC = ATMONEBUL.MOYENN   NBCMCC = 24
Melt snow ............................ CNCFON = SURFFONTE NEIGE    NBCFON = 24
Snow mass ............................ CNCSNS = SURFRESERV NEIGE   NBCSNS = 24
Snow Sublimation ..................... CNCS   = SURFFLU.MSUBL.NE   NBCS   = 24
Soil Moisture ........................ CNCWS  = SURFCONTENU EAU    NBCWS  = 24
Solid specific moisture .............. CNCICE = ATMOHUMI SOLIDE    NBCICE = 24
Surface down solar flux .............. CNCSOD = SURFRAYT DIFF DE   NBCSOD = 24
Surface down thermic flux ............ CNCTHD = SURFRAYT THER DE   NBCTHD = 24
Surface downward moon radiation ...... CNCSMR = SURFRAYT.LUNE.DE   NBCSMR = 24
Surface Latent Heat Flux ............. CNCSLH = SURFCHAL LATENTE   NBCSLH = 24
Surface parallel solar flux .......... CNCSOP = SURFRAYT DIR SUR   NBCSOP = 24
Surface Sensible Heat Flux ........... CNCSSH = SURFFLU.CHA.SENS   NBCSSH = 24
Surface solar radiation .............. CNCSSR = SURFFLU.RAY.SOLA   NBCSSR = 24
Surface Thermal radiation ............ CNCSTR = SURFFLU.RAY.THER   NBCSTR = 24
Surface water content run-off......... CNCSRU = SURFRUISSELLEMEN   NBCSRU = 24
Tendency of Surface pressure ......... CNCTSP = SURFPRESSION SOL   NBCTSP = 24
Top clear sky longwave radiative flux  CNCTTHC= SOMMRAYT THER CL   NBCTTHC= 24
Top clear sky shortwave radiative flux CNCTSOC= SOMMRAYT SOL CL    NBCTSOC= 24
Top mesospheric enthalpy ............. CNCTME = TOPMESO ENTH       NBCTME = 24
Top parallel solar flux .............. CNCTOP = TOPRAYT DIR SOM    NBCTOP = 24
Top Solar radiation .................. CNCTSR = SOMMFLU.RAY.SOLA   NBCTSR = 24
Top Thermal radiation ................ CNCTTR = SOMMFLU.RAY.THER   NBCTTR = 24
Total Cloud cover .................... CNCTCC = ATMONEBUL.TOTALE   NBCTCC = 24
Total Ozone .......................... CNCTO3 = ATMOOZONE TOTALE   NBCTO3 = 24
Total precipitable water ............. CNCQTO = ATMOHUMI TOTALE    NBCQTO = 24
Transpiration ........................ CNCTP  = SURFTRANSPIRATIO   NBCTP  = 24
```

```
U-momentum of Gravity-Wave Drag stress CNCUGW = SURFTENS.DMOG.ZO   NBCUGW = 24
U-momentum of Turbulence stress ...... CNCUSS = SURFTENS.TURB.ZO   NBCUSS = 24
V-momentum of Gravity-Wave Drag stress CNCVGW = SURFTENS.DMOG.ME   NBCVGW = 24
V-momentum of Turbulence stress ...... CNCVSS = SURFTENS.TURB.ME   NBCVSS = 24
Water Evaporation .................... CNCE   = SURFFLU.MEVAP.EA   NBCE   = 24
```

Notice: precipitations are expressed in kg/m^2 (equivalent to mm)

(*d*) *Instantaneous fluxes descriptors*

This section details the content of a part of the namelist NAMAFN which contains the descriptors of the instantaneous fluxes. The descriptor %CLNAME serves to fill the array CFPXFU in the namelist NAMFPC.

```
CAPE out of the model ................. CNXCAPE= SURFCAPE.MOD.XFU   NBXCAPE= 24
Contribution of Convection to Cp.T ... CNXCVS = S000FL.CT CONVEC   NBXCVS = 24
Contribution of Convection to Q ...... CNXCVQ = S000FL.Q  CONVEC   NBXCVQ = 24
Contribution of Convection to U ...... CNXCVU = S000FL.U  CONVEC   NBXCVU = 24
Contribution of Convection to V ...... CNXCVV = S000FL.V  CONVEC   NBXCVV = 24
Contribution of Gravity Wave Drag to U CNXGDU = S000FL.U ONDG.OR   NBXGDU = 24
Contribution of Gravity Wave Drag to V CNXGDV = S000FL.V ONDG.OR   NBXGDV = 24
Contribution of Turbulence to Cp.T ... CNXTUS = S000FL.CT TURBUL   NBXTUS = 24
Contribution of Turbulence to Q ...... CNXTUQ = S000FL.Q  TURBUL   NBXTUQ = 24
Contribution of Turbulence to U ...... CNXTUU = S000FL.U  TURBUL   NBXTUU = 24
Contribution of Turbulence to V ...... CNXTUV = S000FL.V  TURBUL   NBXTUV = 24
Convective Cloud Cover ................ CNXCCC = SURFNEBUL.CONVEC   NBXCCC = 24
Convective precipitation .............. CNXCP  = S000PLUIE CONVEC   NBXCP  = 24
Convective Snow Fall .................. CNXCSF = S000NEIGE CONVEC   NBXCSF = 24
Gusts out of the model ................ CNXGUST= CLSRAFAL.MOD.XFU   NBXGUST= 24
Height of the PBL out of the model
(times g) ............................. CNXPBLG= CLPGEOPO.MOD.XFU   NBXPBLG= 24
High Cloud Cover ...................... CNXHCC = SURFNEBUL.HAUTE    NBXHCC = 24
Large Scale Precipitation ............. CNXLSP = S000PLUIE STRATI   NBXLSP = 24
Large Scale Snow fall ................. CNXLSS = S000NEIGE STRATI   NBXLSS = 24
Low Cloud Cover ....................... CNXLCC = SURFNEBUL.BASSE    NBXLCC = 24
Maximum relative moisture at 2 meters  CNXX2HU= CLSMAXI.HUMI.REL   NBXX2HU= 24
Maximum temperature at 2 meters ...... CNXX2T = CLSMAXI.TEMPERAT   NBXX2T = 24
Medium Cloud Cover .................... CNXMCC = SURFNEBUL.MOYENN   NBXMCC = 24
Minimum relative moisture at 2 meters  CNXN2HU= CLSMINI.HUMI.REL   NBXN2HU= 24
Minimum temperature at 2 meters ...... CNXN2T = CLSMINI.TEMPERAT   NBXN2T = 24
MOCON out of the model ................ CNXMOCO= CLPMOCON.MOD.XFU   NBXMOCO= 24
Relative Humidity at 2 meters ......... CNX2RH = CLSHUMI.RELATIVE   NBX2RH = 24
Specific Humidity at 2 meters ......... CNX2SH = CLSHUMI.SPECIFIQ   NBX2SH = 24
Surface solar radiation ............... CNXSSR = S000RAYT.SOLAIRE   NBXSSR = 24
Surface Thermal radiation ............. CNXSTR = S000RAYT.TERREST   NBXSTR = 24
Temperature at 2 meters ............... CNX2T  = CLSTEMPERATURE     NBX2T  = 24
Top Solar radiation ................... CNXTSR = SOMMRAYT.SOLAIRE   NBXTSR = 24
Top Thermal radiation ................. CNXTTR = SOMMRAYT.TERREST   NBXTTR = 24
Total Cloud cover ..................... CNXTCC = SURFNEBUL.TOTALE   NBXTCC = 24
U-momentum of gusts out of the model . CNXUGST= CLSU.RAF.MOD.XFU   NBXUGST= 24
U-momentum of wind at 10 meters ...... CNX10U = CLSVENT.ZONAL      NBX10U = 24
V-momentum of gusts out of the model . CNXVGST= CLSV.RAF.MOD.XFU   NBXVGST= 24
V-momentum of wind at 10 meters ...... CNX10V = CLSVENT.MERIDIEN   NBX10V = 24
Wind velocity at 10 meters ............ CNX10FF= CLSWIND.VELOCITY   NBX10FF= 24
```

D.7 SELECTION FILE EXAMPLE

To get the following fields:

- Model orography on domains FRANCE and EUROC25 at time h00
- Surface pressure on domain EUROC25 at times h00 and h03
- Geopotential at 500 hPa on domains FRANCE and EUROC25 at time h00
- Geopotential at 850 hPa on domains FRANCE and EUROC25 at time h03
- Temperature at 850 hPa on domain FRANCE at time h00
- Temperature at 500 hPa on domain EUROC25 at time h00 and h03
- Potential vorticity at 300 K on domain FRANCE at time h00

You would first have the following parameters in the namelist file:

```
/NAMCT0
   CNPPATH='.',
/END
/NAMFPC
   CFP2DF='SPECSURFGEOPOTEN','SURFPRESSION',
   CFP3DF='GEOPOTENTIEL','TEMPERATURE','POT_VORTICIT',
   RFP3P(1)=500.,
   RFP3P(2)=850.,
   RFP3T(1)=300.,
   CFPDOM='FRANCE','EUROC25',
/END
```

Then you would add in your script:

```
/bin/cat <EOF>> xxt00000000
/NAMFPPHY
/END
/NAMFPDY2
   CL2DF(1)='SPECSURFGEOPOTEN',
   CLD2DF(1)='FRANCE:EUROC25',
   CL2DF(2)='SURFPRESSION',
   CLD2DF(2)='EUROC25',
/END
/NAMFPDYP
   CL3DF(1)='GEOPOTENTIEL',
   ILD3DF(1,1)=1,
   CLD3DF(1,1)='FRANCE:EUROC25',
   CL3DF(2)='TEMPERATURE',
   ILD3DF(1,2)=1,2,
   CLD3DF(1,2)='EUROC25',
   CLD3DF(2,2)='FRANCE',
/END
/NAMFPDYH
/END
/NAMFPDYV
   CL3DF(1)='POT\_VORTICIT',
   ILD3DF(1,1)=1,
   CLD3DF(1,1)='FRANCE',
/END
/NAMFPDYT
/END
/NAMFPDYS
```

```
/END
EOF

/bin/cat <EOF>> xxt00000300
/NAMFPPHY
/END
/NAMFPDY2
   CL2DF(1)='SURFPRESSION',
   CLD2DF(1)='EUROC25',
/END
/NAMFPDYP
   CL3DF(1)='GEOPOTENTIEL',
   ILD3DF(1,1)=2,
   CLD3DF(1,1)='FRANCE:EUROC25',
   CL3DF(2)='TEMPERATURE',
   ILD3DF(1,2)=1,
   CLD3DF(1,2)='EUROC25',
/END
/NAMFPDYH
/END
/NAMFPDYV
/END
/NAMFPDYT
/END
/NAMFPDYS
/END
EOF


/bin/ls > dirlst
```

D.8 MAKING CLIMATOLOGY FILES

You need to run the configuration 923 (ARPEGE/IFS) for a Gaussian grid, or the configuration E923 (ALADIN) for a LAM grid or a LAT × LON grid.

You should not forget to specify in the namelists of the configuration 923/E923 the definition(s) of your output (sub-)domain(s). Remember that in the case of LAT × LON grids there is no extension zone (set NDGL=NDGUX and NDLON=NDLUX in NAMDIM) and the geometry is not plane (set LRPLANE=.FALSE. in NAMCTO).

Finally do not forget that in the case of any gridpoint output for ordinary post-processing the surface geopotential should not be spectrally fitted (set LKEYF=.FALSE. in NAMCLA).

D.9 SPECTRAL FILTERS

There are two formulations used to smooth the fields.

The first one — nicknamed *thx* because it uses the hyperbolic tangent function — is used in ARPEGE/IFS only to smooth the fields which are horizontal derivatives, or which are built upon horizontal derivatives, especially when the model is stretched. It looks like a smoothed step function:

$$f(n) = \frac{1 - \tanh(e^{-k}(n - n_0))}{2}$$

where n is a given wavenumber in the *unstretched* spectral space, k is the intensity of the filter and n_0 is the truncation threshold: this function roughly equals 1 if n is less than n_0, and roughly equals 0 if it is bigger.

Figure D.2 on page 136 illustrates this spectral filter. gobbleenv The next figure illustrates this spectral filter:

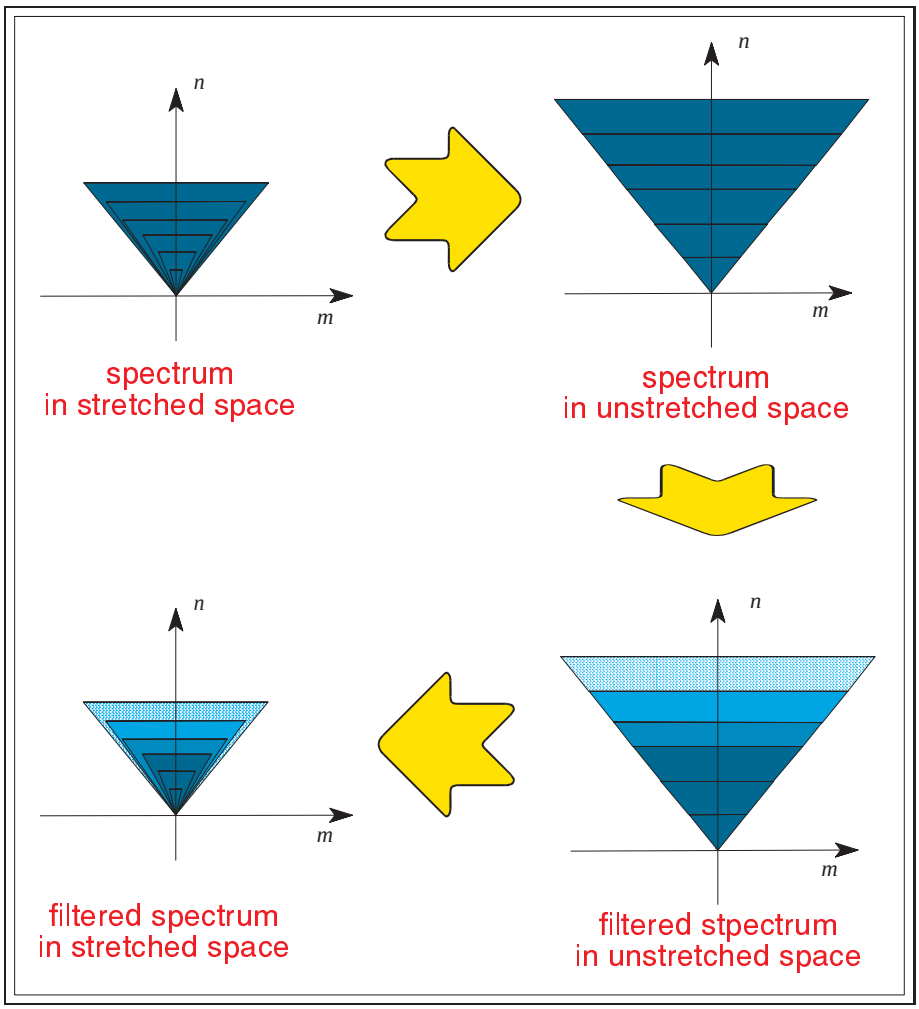

Figure D.2 *Illustration of the spectral filter for derivatives in* ARPEGE/IFS.

The second one is a Gaussian function. In ARPEGE/IFS it writes:

$$f(n) = e^{\frac{-k}{2}(n/N)^2}$$

where n is a given wavenumber, k is the intensity of the filter and N represents the model triangular truncation.

In ALADIN it writes:

$$f(n,m) = e^{\frac{-k}{2}((n/N)^2+(m/M)^2)}$$

where (n, m) is a given pair of wavenumbers, k is the intensity of the filter and (N,M) represent the model elliptic truncation.

In ALADIN this Gaussian filter is used to filter any field ("derivative" or not).

Figure D.3 on page 137 illustrates this spectral filter. gobbleenv The next figure illustrates this spectral filter:

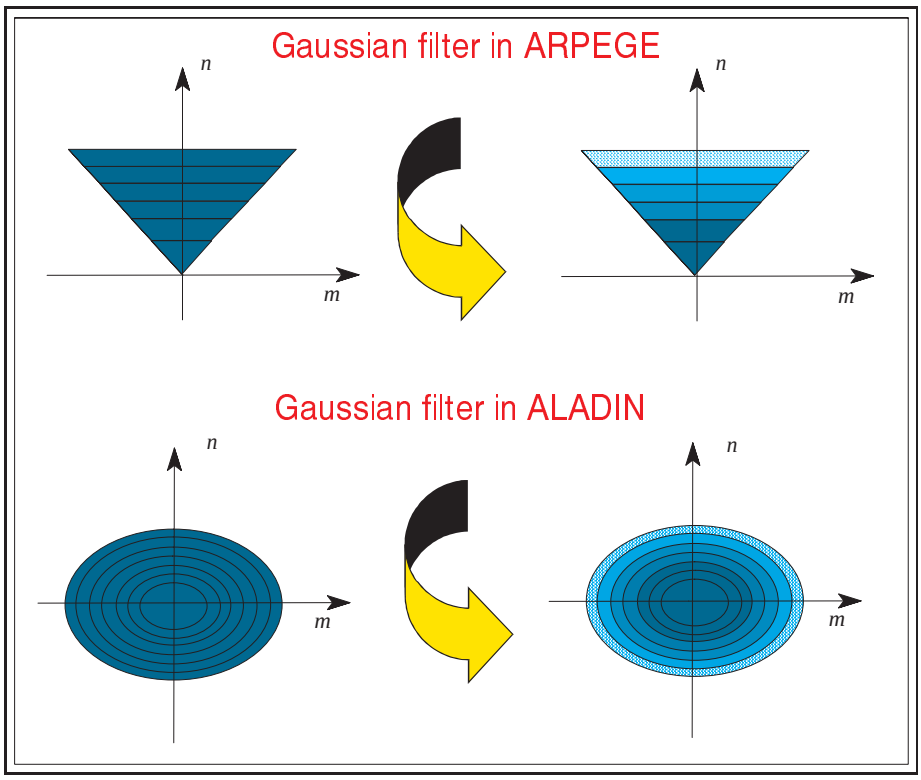

Figure D.3 *Illustration of the Gaussian spectral filter.*

D.10 OPTIMIZATION OF THE PERFORMANCE

D.10.1 Communications

To write post-processed fields in an output file, you first gather the distributed pieces of these fields from the different processors.

Rather than gathering the fields one after the other, the fields are grouped in chunks, and these chunks are treated one after the other.

The variable NFPXFLD is the maximum size of these chunks. Lowering it should save memory to the detriment of inter-processors communications, and vice versa.

D.10.2 Segmentation

Several variables control the segmentation of the software arrays:

NPROMA is the elementary size of the gridpoint rows in the model geometry. In the post-processing it is in use mostly during the vertical interpolations.

NFPROMAG is the elementary size of the gridpoint rows in the post-processing geometry. It is used mostly during the horizontal interpolations.

NFPROMEL is the elementary size of the gridpoint rows in the post-processed extension zone for LAM output. It is used only in ALADIN during the computation of the post-processed extension zone.

By definition all these variables control a part of the vectorization depth as well as memory cost. The bigger these variables are, the deeper the vectorization is, in detriment to the memory cost. On non-vector machines it is better to use small values for these parameters in order to fit the cache memory. They should not be a power of 2 to avoid memory bank conflicts. One should refer to the machine constructor to choose the best values for these variables.

Appendix E
FullPos technical guide

Author: R. El Khatib
METEO-FRANCE - CNRM/GMAP

Table of contents

E.1 FOUNDER PRINCIPLES

E.1.1 Basic concept

FULLPOS is a non-independent software: it is designed to serve specifically the ARPEGE/ALADIN post-processing.

To get a post-processing fully consistent with the model itself, FULLPOS software has been completely embedded inside the ARPEGE/ALADIN software, in order that it can (and it should!) re-use model operators. This should also simplify a few maintenance operations[1].

The reliability of this concept has been ensured by the existence of a previous internal post-processing software (currently pointed out by the name of its leading subroutine: `POS`). However the target of this previous internal software was limited to vertical pressure interpolations for a few specific fields to be written out as spherical harmonics, while FULLPOS is designed to be a comprehensive post-processing tool.

FULLPOS is also designed to serve both operations (which implies: high efficiency to be run in real time situation) and research (which means: the ability to process various elaborated fields on various grids and vertical levels).

E.1.2 Scientific layouts

This section will list the processes that have taken part in the elaboration of FULLPOS. Presented prior to the architecture of the software, it should help understanding the conception of the code.

- Dynamical fields should be post-processable on pressure levels (P), potential vorticity levels (P_v), isentropic levels (θ), eta levels (η) including other definitions than the model originating eta levels, and on height levels above a given orography (z).

[1] This is less and less true.

- Fields post-processed on P, P_v and θ levels should be interpolated vertically first, then horizontally. In between, it should be possible (as an option) to fit the fields in spectral space (to remove the numerical noise induced by the vertical interpolations). For a few specific fields (like geopotential, medium sea-level pressure, ...) for which the formulation of vertical interpolation induces potential inconsistencies, a filter in spectral space should be optionally performed.
- Fields post-processed on η and z levels should respect the profile of the boundary layer; therefore they should be interpolated horizontally first, then re-adjusted with respect to the orography of the target grid.
- To ensure the inter-consistency of fields interpolated on η or z levels, only the model primitive variables (U, V, T, q, P_s currently) should be horizontally interpolated: the other fields should be recomputed from the interpolated model primitive variables.
- Fields on horizontal surfaces should be homogenous; in other words the small scale information should not pollute the interpretation of the output fields. This means that the fields which are composed of derivatives (like vorticity, divergence, vertical velocity as the integral of the divergence, but also any field on P_v levels) should be filtered in a spectral space of homogenous resolution[2]. The intensity of this filter should depend of the output grid resolution[3].
- Physical fields from the model, including cumulated fluxes and instantaneous diagnostics, should be post-processable: their interpolations require often specific treatments, like the land/sea aspect (only points of the same nature should serve the interpolations), the control of the validity domain for the output values (for instance: the interpolated land/sea mask should be either 0. or 1.), or the interdependencies of the post-processed physical fields (for instance: deep soil temperature should be interpolated as its anomaly with respect to surface temperature, which implies to interpolate surface temperature prior to deep soil temperature).
- It should be possible to interpolate "physico-dynamic" fields: these are fields defined on a surface and computed with model physical surface fields as well as upper air dynamic fields (CAPE is one of them). If computed on the model originating grid, this should be easy (because the environment is then very similar to the model gridpoint environment), but if computed on another grid, this implies to interpolate horizontally almost all the model pronostic fields (upper air as well as surface).
- It should be possible to use FULLPOS to make full history files (for coupling, nesting, multi-incremental variational purpose, or even bogussing). This means that the fields (mainly the physical fields and the dynamic fields post-processed on η levels) should be interpolated with respect to a target orography (ie: the orography of the output grid) which should be spectrally fitted in the output spectral geometry; and also that it should be possible to write out the dynamic fields as spectral coefficients for the target geometry in this case.
- While doing bogussing, to prevent from getting "walls" at the border of the bogussing area, the target grid should get only the interpolated increments from the source fields.
- It should be possible to use climatology data in order to interpolate with a better accuracy a few surface fields; instead of a straightforward interpolation, we would interpolate the anomaly of a field with respect to the climatology, or even: we would impose the whole climatology field if it is a constant field (land-sea mask for instance).
- In case of gridpoint outputs on a complete ALADIN grid, the extension zone should be computed as well, taking into account the realism of the physical fields.
- In the initial design, horizontal interpolations had to be quadratic exclusively[4].
- Wind-related fields (like vorticity, divergence, etc) should be computed from the wind components so that all these fields are consistent.

[2] Actually not done for wind on PV levels.

[3] Not effective for aladin. Is it a bug? However we usually write out only one grid from aladin. Anyhow this should be harmonized with ARPEGE.

[4] Things are supposed to change in the future, so that each field could have a specific interpolation kind: quadratic, bilinear or no interpolation but the value of the model nearest point is adopted.

E.1.3 Technical requirements

Beside the scientific aspects, various technical aims had to be achieved:

- it should be possible to post-process during the model direct integration ("in-line post-processing") as well as after ("off-line post-processing"), both solution giving the same results. This implies that FULLPOS should not be a specific ARPEGE/ALADIN configuration but a package which could be called inside the direct model temporal loop, and that the packing of fields in history files should be considered.
- FULLPOS should benefit, as the model does, from the (super)computer hardware architecture, that is: the memory distribution today and probably OPEN-MP tomorrow[5].
- FULLPOS should be cheap.
- FULLPOS should be modular and should not spread itself all over the code[6].
- FULLPOS users interface should be ergonomic. This should not mean that the users interface should be restricted to a limited number of namelists parameters, but rather that the namelists should be easy to set, with meaningful parameters.
- FULLPOS should stick to the ARPEGE/ALADIN interfaces standards: namelists parameters for the users interface and ARPEGE/ALADIN files for the I/O data.
- the list of post-processing fields, levels and horizontal domains per post-processing time range should be flexible.
- The horizontal output format has been restricted to: either one gaussian grid, or one ALADIN grid, or a set of LATLON grids, or one definition of spectral coefficients.
- One should be able to pack each post-processing field on a tunable specific number of bits.

E.1.4 Technical limitations

Nobody is perfect, and the code is not, either: before understanding a few technical choices concerning the conception of FULLPOS, one should remember the following technical limitations existing at the time this software has been first conceived:

- The software has been elaborated upon a FORTRAN 77 compiler; FORTRAN 90 was not available at that time.
- To extend the possibilities of the software, ARPEGE/ALADIN system was currently using FORTRAN Cray extensions, like a memory manager system based on the pre-allocation of a heap; from this heap it was possible to allocate arrays. This system was not so flexible as the `ALLOCATE` statement of the FORTRAN 90 language.
- Spectral transform were not modular.
- The ARPEGE/ALADIN was originally designed for a multi-processor vector machine with limited central memory, using multitasking and having fast I/Os; while the architecture of today is distributed, not always vector, and with relatively slow I/Os but with a large central memory.

[5] Formerly: the multitasking and the I/O scheme.

[6] This has been a failure definitely, partly because of the technical limitations/constraints at the time of the first conception,. However things has changed so that FULLPOS is getting more concentrated and modular. Its "externalization" from the code arpege/ifs/aladin is even under consideration.

E.2 GENERAL CONCEPTION

E.2.1 Architecture

This section will describe the main subroutines involved in FULLPOS and how they are articulated between one another.

(*a*) *General implementation*

FULLPOS is included in the configuration 001 of ARPEGE/IFS/ALADIN: this is to enable the in-line post-processing as well as the off-line post-processing (in the latter case, it is enough to run the model on zero time step but with the post-processing activated).

However it remain theoretically possible to implement FULLPOS in any other configuration. For instance, one can imagine to implement it CANARI (configuration 701).

A logical key: LFPOS has been implemented in order to select either FULLPOS or the previous internal post-processing. In the near future, this old post-processing should be removed, but this will not be so easy since it is also used for the so-called "movies". Note also that LFPOS and the binary namelist variable N1POS (controlling the main post-processing flow) could then be merged.

The general mechanism of the code follows the "control routines cascade" (CNT0 to CNT4) of the model (see Figure E.1 on page 142), though all the subroutines are not used. The next list describes the purpose of each control subroutine:

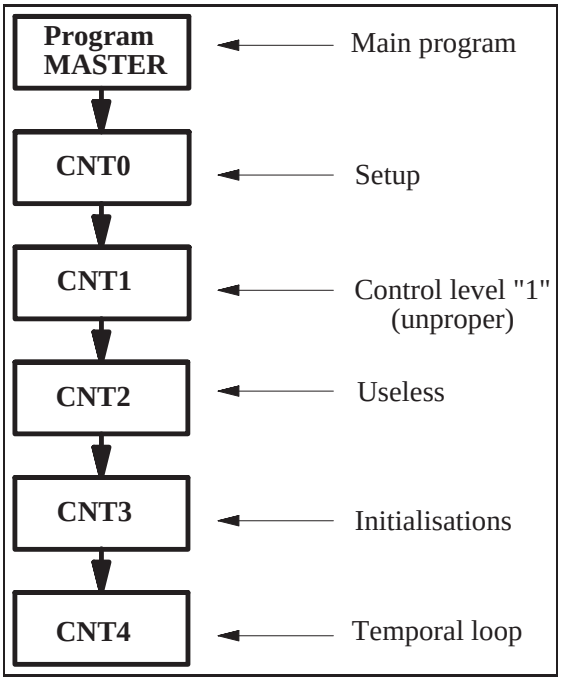

Figure E.1 *General mechanism of the post-processing inside the model.*

MASTER : Main program; it does not do anything but calls the control subroutine CNT0.

CNT0 : Setup (control of level "0"); it initializes the constants or the namelists variables as scalars or arrays.

CNT1 : Control of level "1".FULLPOS is concerned by this subroutine for two reasons. The first one is because the namelist NAMCT1, which contains two variables used by FULLPOS (N1POS and LRFILAF), is read below this subroutine; the second one is because the key LFPOS is still used in this subroutine

for a completely old-fashioned reason. These situations are not proper, and in future developments FULLPOS should no more be concerned by CNT1[7].

CNT2 : Useless for FULLPOS.

CNT3 : Initializations (control of level "3"); it reads the initial conditions data (and possibly the needed climatology data) and compute the working arrays which would depend on these data.

CNT4 : Temporal loop (control of level "4"); it initializes the time-dependent post-processing variables (like the list of fields to be post-processed for the current model step) and performs the post-processing inside the model temporal loop.

When FULLPOS is configured to make history files, things are more complicated since the code has been conceived with non-modular spectral transform: the control cascade should be invoked two times in order to change the setup of the spectral transforms.

The first control cascade will be called the external part, while the second control cascade will be called the internal part.

This mechanism is achieved by a supplementary control subroutine named CPREP4 which is called once by CNT4 (in the external part) and once by CNT3 (in the internal part). The location of CPREP4 in CNT4 for the external part is justified by the fact that this mechanism could potentially work in the "in-line" mode, though this possibility has never been used, and has even been removed from the code. The location of CPREP4 in CNT3 for the internal part is justified by the fact that this part is really internal: it is "out of the temporal loop"; actually this internal part is justified only because the spectral transforms were not modular at the time of this conception. At the end of the external part, CPREP4 calls again CNT0 after it has released all the allocated arrays. So we have to cope with a recursive cascade of subroutines (see Figure E.2 on page 144). This needs supplementary control items in order to be able to leave this never-ending loop:

LFPSPEC : namelist variable which should be set to `.TRUE.` for getting FULLPOS configured to make history files. Note that this so-called "configuration" is not a real one, since the code is still implemented in the configuration 001. However, this system is well-known as configuration (e)(e)927 for historical reasons (it has replaced the previous true configuration 926).

LFPART2 : internal key telling whether the running part is the external one (`LFPART2 = .FALSE.`) or the internal one (`LFPART2 = .TRUE.`; also called: "part 2").

`ncf927` : pilot file to control the never-ending loop on CNT0: its existence means that the post-processing is over.

(*b*) *Setup*

Still following the general structure of the code ARPEGE/ALADIN, the setup of FULLPOS is split in two main subroutines: SUOYOMA and SUOYOMB. One should remind that the original reason for splitting the setup was because of the memory management handling on the former Cray computer. Today there is no reason for such splitting, but we have to live with the past, before reforming it.

Concerning the post-processing, it was mostly important to make a hierarchical setup, in order to have an easy setup using consistent default values as much as possible. Sometimes, this becomes contradictory with the principle of modularity; as a consequence, in some places, the setup of FULLPOS is spread over the setup of the model. In the future this matter of fact should be reduced by a progressive reorganization of the setup of the model.

Notice: in the case of the family of configurations 927, the internal call to CNT0 (case: `LFPART2=.TRUE.`) should be limited to what needs to be re-initialized only, because of the change of geometry. For instance

[7] LFPOS has already been removed from CNT1 in the cycle 26.

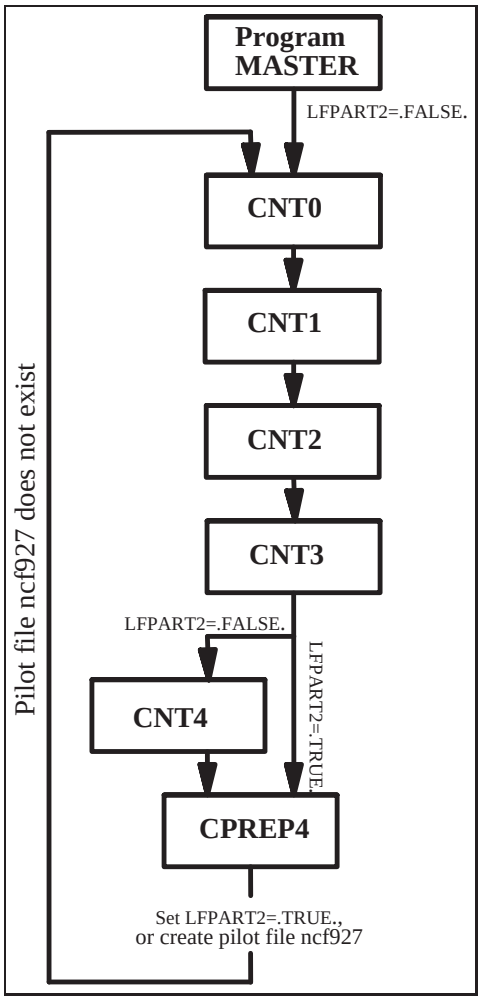

Figure E.2 *General mechanism of the so-called configuration (e)(e)927.*

the message passing should not be re-initialized. One will also notice a call to the synchronization barrier: this is because the I/O operations should be finished before starting the internal part. In short one will notice that the key `LFPART2=.TRUE.` is often spoiling the code. It would be advantageous to reconsider this "second part" so that instead of recalling the whole control subroutines cascade, it would recall only its needed parts.

SUOYOMA This subsection will list the subroutines which are (more or less) specific to FULLPOS in the first part of the setup. Besides, Figure E.3 on page 145 will show the scheme of this subroutine. One should not forget that the post-processing is also using a large amount of variables coming from the model itself (like logical unit numbers). In the scope of the externalization of FULLPOS, it will be necessary to create a data module to transfer these model variables to internal post-processing variables.

SUAFN (SetUp Arpege Field Names): To initialize extensive descriptors about the fields which are post-processable (which fields and how they should be treated). This subroutine is interfaced with a namelist (NAMAFN) and consequently initializes the corresponding module YOMAFN. It calls three subroutines:

- SUAFN1: setup default values, using some model variables as a background; so it must be called after SUDIM1.
- SUAFN2: control the users values and complete with supplementary internal control variables.
- SUAFN3: print out the initialized descriptors.

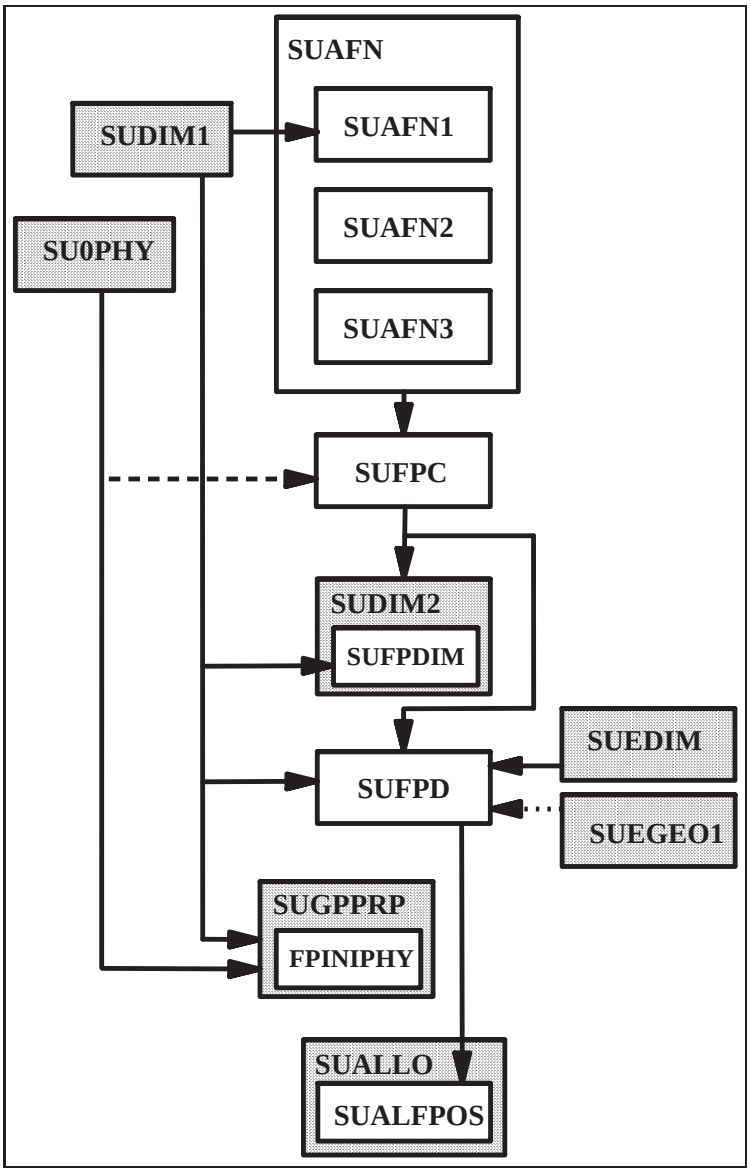

Figure E.3 *Setup, first part: SU0YOMA. The greyish areas correspond to model subroutines.*

Remarks:

- YOMAFN is now used in the model to read non-hard-coded ARPEGE field names. This is embarrassing because it makes both the model and FULLPOS less independent. The solution should be to create a derived type variable in the model to setup the ARPEGE field names and possibly other characteristics (GRIBcode and the minimum number of bits the field should be encoded with). These derived type would then be used to setup FULLPOS.
- SUAFN and its relatives treat all the post-processing fields (from dynamics or physics): this matter of fact is penalizing the maintenance. It would be better to split this ensemble in two parts: one for dynamics and one for physics; then each part could be re-unified in one single module.

SUFPC (SetUp FullPos Computation): originally designed to setup the list of fields to post-process as well as all scientific or technical options used for the post-processing. This subroutine has become a mess, depending on a lot of model variables. It should be split one of these days in one subroutine for the various post-processing options, and one subroutine for the list of fields to post-process.

SUFPDIM (SetUp FullPos Dimensioning): originally designed to setup dimensions of arrays to be shared with the model (in the spirit of re-using the model allocated arrays to save memory). This subroutine was highly related to the shared memory architecture and the internal spectral transforms, that is why it is called inside SUDIM2. Today it has two new purposes:

- To complete the descriptors in YOMAFN with information from the model settings (more exactly: which variables are primitive); for that it needs to be run after SUDIM1 and SUFPC.
- To compute static dimensions of data arrays now specific to FULLPOS, so it needs to be called after SUFPC.

Remarks:

- In the cycle 26, this subroutine has just been removed, because it is enough and more efficient to work with dynamic dimensioning only. This was not possible at the time of the conception because the I/O scheme needed static dimensionings for the workfiles.
- There is a dirty trick in SUDIM1: the namelist NAMFPC is read there in order to initialize the variable NFPCLI. This is the consequence of a hurried (and hurting) split of SUDIM in SUDIM1 and SUDIM2. The cleaning might come from the future split of SUFPC, but a more robust solution would be to have an independent gridpoint buffer to contain the input climatology.
- The fact that a part of a descriptor is initialized in SUAFN* and another part in SUPDIM proves that this descriptor should be split!

SUFPD (SetUp FullPos Domains): to initialize the dimensions and bounds of the output post-processing (sub)-domains. To take advantage of default values which would be the model dimensions, this subroutine needs to be called after SUDIM1 for ARPEGE and after SUEDIM and SUEGEO1 for ALADIN. Unfortunately SUEGEO1 is called late in SUOYOMB for the time being. The solution is probably to move SUFPD inside SUBFPOS (see SUOYOMB below).

FPINIPHY (FullPos INItialization of physics): to control that all the model physical fields pointers which will be used by the post-processing are initialized. This is a fragile subroutine, which should be merged with parts of SUFPC (tests about physical aspects) and it should rather work on the physics logical keys and dimensions (from the namelists NAMPHY and NAMDPHY, actually prior to reading these namelists) rather than pointers values (we do not know what is the "undefined" value). For the time being it is called by SUGPPRP in order to get this "undefined" pointer value.

SUALFPOS (SetUp ALLocations of FullPOS): to allocate various arrays. The location of this subroutine inside SUALLO and at the bottom of SUOYOMA is purely the consequence of the conception on the former Cray machine. In the cycle 26, this subroutine has been moved inside SUBFPOS below SUOYOMB (see below). However the call to this subroutine is conditioned by the initialization of the model and the post-processing dimensioning.

SUOYOMB In the second part of the setup, things are fortunately much more condensated as there are less interactions with the model itself.

Figure E.4 on page 147 shows the scheme of this subroutine.

SUCFUFP (SetUp Cumulated FlUxes for FullPos): To activate the model cumulated fluxes switches which will be needed by the post-processing. It is called inside SUCFU. This routine is typically an input–output interface between the model and FULLPOS (finally like FPINIPHY described above).

SUXFUFP (SetUp X —instantaneous— FlUxes for FullPos): same as SUCFUFP for the instantaneous fluxes; called inside SUXFU. Same remark.

SUBFPOS (SetUp part B of FullPOS): this subroutine contains a lot of specific subroutines for FULLPOS. Here is a rough description of them:

- SUFPG (SetUp FullPos Geometry): To initialize the geometric parameters of the output grids ... and the post-processing gridpoint distribution as well!

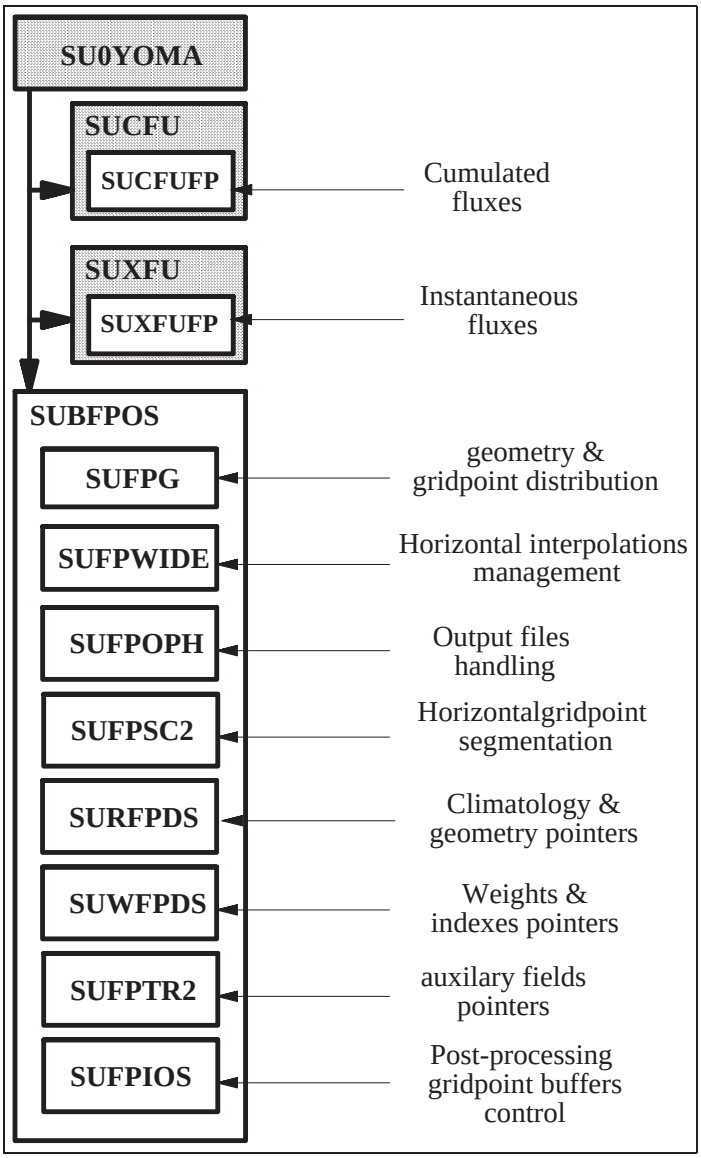

Figure E.4 *Setup, second part: SUOYOMB. The greyish areas correspond to model subroutines.*

- SUFPWIDE (SetUp FullPos WIDE): To initialize the parameters and working arrays needed for the horizontal interpolations mechanism.
- SUFPF (SetUp FullPos Filters): To initialize the spectral filters parameters.
- SUFPOPH (SetUp FullPos Output Parameters Handling): To setup output files names and their autodocumentation part. Note that the files logical unit numbers are initialized in the model subroutine SULUN (A clear problem of interfacing the model and the post-processing).
- SUFPSC2 (SetUp FullPos SCan 2): To initialize the horizontal segmentation in the post-processing gridpoint calculation and some corresponding control arrays.
- SURFPDS (SetUp Real fields FullPos DeScriptors): To initialize fields pointers in output buffers for the target climatology and target geometry. Notice: there were no reason to put altogether these fields (climatology and geometry) but to limit I/O operations in the framework of the former I/O scheme. Today it would be advantageous to split climatology and geometry because it would improve the modularity and the understanding of the code.
- SUWFPDS (SetUp Weights FullPos DeScriptors): To initialize fields pointers in output buffers for the horizontal interpolations weights, addresses and indexes. Note that fields in this buffer are all considered as real fields, though some of them are actual integer fields. In the future it would be advantageous to build a more clever field organization (using either the Fortran

function TRANSFER in order to save memory, or just by creating a integer gridpoint buffer: now very easy).

- SUFPTR2 (SetUp Fullpos PoinTeR 2): To initialize the "auxiliary fields" pointers, that is the special surface fields which should be horizontally post-processed prior to the current flow of post-processed fields, because they will have to be used for the current post-processing flow.
- SUFPIOS (SetUp Fullpos IO scheme): To initialize the control parameters for each "post-processing gridpoint buffer".
  Remarks:
  - This subroutine was originally designed to initialize the former I/O schemes for the post-processing gridpoint buffers, but its purpose has now completely changed.
  - This subroutine, which is reading the namelist NAMFPIOS containing a unique variable, should be merged with SUFPSC2 which also reads the namelist NAMFPSC2 containing a unique variable; so one of these two namelists could be removed.
  - It would be more robust to setup each buffer descriptors at the time the buffer is allocated. Actually this has been already done for cycle 26. To go further, the next step would be to define the post-processing gridpoint buffers as derived types, together with their operators (allocation and initialization, deallocation, and even reading in and writing out).

  Notice: One will notice that in SUBFPOS a few variables need to be initialized even if FULLPOS is not active (LFPOS=.FALSE.): this clearly shows that these variables have an active role in subroutines which are common to the model and the post-processing. In the scope of the externalization of FULLPOS, this should disappear.

(*c*) *Initializations*

Figure E.5 on page 148 shows the scheme of this subroutine.

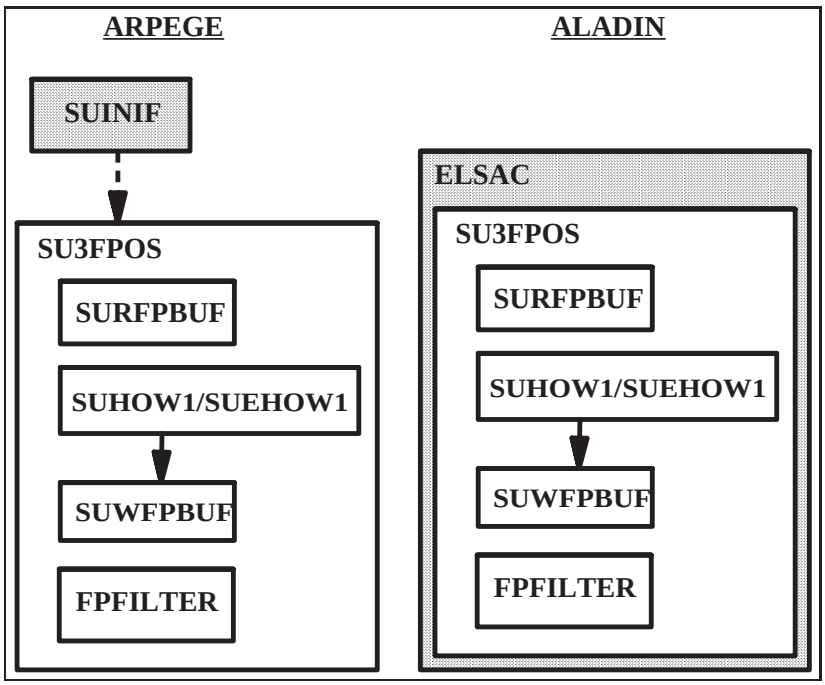

Figure E.5 *Initializations: CNT3. The greyish areas correspond to model subroutines.*

The initializations part is composed of two main parts:

- The initialization of the input meteorology data (including possibly climatology data) which is common with the model: SUINIF for ARPEGE and ELSAC for ALADIN.
- RoutineNameSU3FPOS (SetUp level 3 FullPOS): to initialize the following data buffers/arrays:

- The buffer containing the output climatology and geometry (refer to SURFPDS). This is performed by the subroutine SURFPBUF.
- The buffer containing the weights and indexes for horizontal interpolations (refer to SUWFPDS). This is performed by the subroutines SUEHOW1/SUHOW1/SUWFPBUF.
- The matrixes for the spectral filters (performed by the subroutine FPFILTER).

Remarks:

- The occurrence of a piece of code in CNT3 is justified only if it depends on the initial condition fields; that is why SURFPBUF could (and should!) be moved away from it, and put into SUBFPOS for instance.
- In cycle 26 FPFILTER has been moved away and put into the control subroutine for dynamic post-processing (see DYNFPOS later) because this place is the best one to limit the memory cost (special notice: the array should be allocated if necessary at the beginning of the scans "Vertical then Horizontal", and systematically deallocated at the end of this scan for an optimal memory management).
- The occurrence of SUWFPBUF inside CNT3 is justified because to compute the land-sea-mask-dependent interpolations weights, we need first to interpolated the model land-sea mask (and possibly the surface temperature) when the output climatology is not at disposal.
- SU3FPOS had to be moved inside ELSAC for ALADIN because the computation of weights, if it uses the model surface temperature, should be performed before the digital filters initialization of the model (true?).
- A possibility to concentrate FULLPOS could be just to move SUBFPOS inside CNT3?

(*d*) *Temporal loop*

Figure E.6 on page 149 shows the scheme of this subroutine.

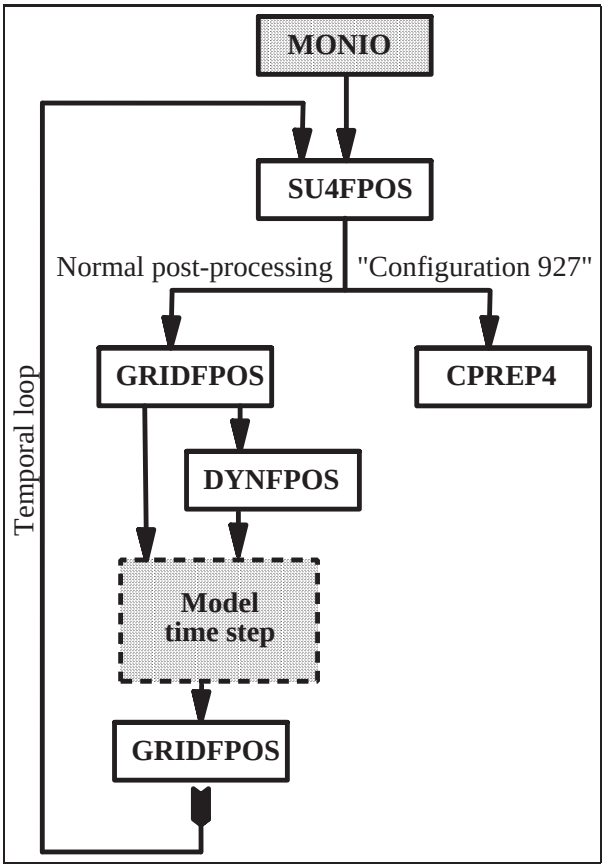

Figure E.6 *Temporal loop: CNT4. The greyish areas correspond to model subroutines.*

The temporal loop subroutine is composed of the following parts:

MONIO (MONitoring of I/O): To stamp the model time steps with the post-processing events; it is called before the temporal loop. This subroutine is not specific to FULLPOS, and thus it is penalizing its modularity. The solution would be first to move out from the model the variables N1POS, NFRPOS and NPOSTS, and put them into a post-processing-specific module.

SU4FPOS (SetUp level 4 FullPOS): To setup the time-dependent variables for the post-processing (ie: the actual list of fields to post-process).

GRIDFPOS (GRIDpoint FullPOS): to perform the post-processing of physical fields (surface fields or fluxes). It can be invoked either at the beginning of the time step, or at its end (Notice: the conditions of call are not clear since ECMWF never use the cumulated fluxes buffer; this must be the residual effects of an old-fashioned specification related to the shift between instantaneous physical fields and the fluxes, in contradiction with the specification for in-line/off-line reproductibility).

DYNFPOS (DYNamic FullPOS): to perform the post-processing of dynamic fields. It is called at the beginning of the model time step to have the correct clock. It should be called after GRIDFPOS (non-lagged call) in order to have the needed surface fields already post-processed before starting the physico-dynamical post-processing. In the scope of the externalization of FULLPOS it will be interesting to have the model STEPO sequences completely independent from DYNFPOS (no overlap on the configuration of IOPACK).

Note that the non-lagged call to GRIDFPOS could simply be put at the beginning of DYNFPOS.

CPREP4 (Configuration PREParatory level 4): to control the mechanism of the so-called "configuration 927".

Notice: the logical key NFPCTO is an internal key to control the so-called "bogussing" configuration, which will be described later. For the moment, it is enough to know that NFPCTO is greater or equal to 1 for ordinary post-processing.

(*e*) *Control of the configurations 927*

This subroutine controls the whole family of "configurations 927", that is "927" (ARPEGE to ARPEGE), "e927" (ARPEGE to ALADIN), "ee927" (ALADIN to ALADIN), and "bogussing" (ALADIN to ARPEGE).

Figure E.7 on page 151 shows the scheme of this subroutine.

The main ingredients are again GRIDFPOS and DYNFPOS.

- In the first part (LFPART2=.FALSE.) the horizontal post-processing is performed.
- Then all the allocated arrays are released (FREEMEM) and the program is re-run from the (CNTO).
- Finally in the second part (LFPART2=.TRUE.) the vertical post-processing is performed.

Remark: again here we can guess that the key LFPART2=.TRUE. will complicate the setup since we need to select from the namelists the horizontal aspects first, then the vertical aspects.

The bogussing consists in a local modification of a history file ARPEGE by a file ALADIN. For that, we actually need to compute the increments to add to the ARPEGE file. Since we do not have coded yet the tangeant linear of FULLPOS we should perform first the horizontal interpolations on two separate ALADIN files (which difference corresponds to the increments).

The bogussing is activated by the key NFPINCR in namelist (0 for no bogussing, 1 for bogussing). Its control is realized by the key NFPCTO which can have the following values:

- NFPCTO=-2: first part of the "external" part (for bogussing only)
- NFPCTO=-1: last part of the "external" part (for all)
- NFPCTO=0: second ("internal") part of the post-processing (for all).

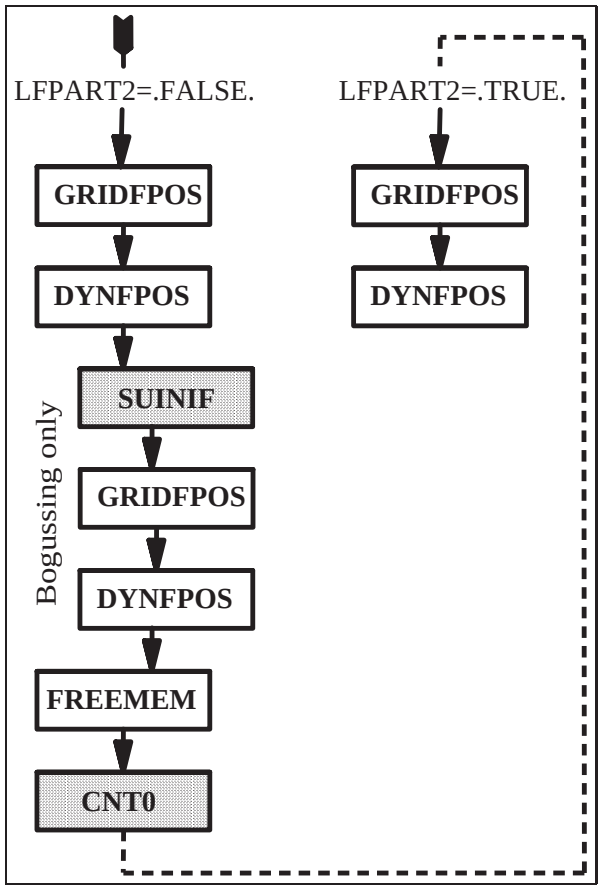

Figure E.7 *Control of the configurations 927: CPREP4. The greyish areas correspond to model subroutines.*

However, and for preliminary test, the bogussing has been enabled out of the "927 configurations system", that is for normal post-processing, through the following options (see CNT4 in ALADIN):

- NFPCTO=1: last normal external part of the post-processing (for all)
- NFPCTO=2: first external part of the post-processing (for bogussing only).

(*f*) *Physical fields post-processing*

The post-processing for physical fields is split in two flows:

(i) Post-processing of auxiliary surface fields: this is an intermediate step before the actual post-processing, as these auxiliary surface fields should be re-used for the post-processing of physical fields and physico-dynamical fields.
(ii) Post-processing of physical fields. This step is completed by the computation of the extension zones of the post-processed fields (if the output domain includes such an area).

Figure E.8 on page 152 shows the scheme of this subroutine.

Following the mechanism of gridpoint calculations in the model and the horizontal interpolations for the semi-Lagrangian scheme, the horizontal interpolations for the post-processing have been embedded inside the model subroutine SCAN2H, invoked with a specific configuration string.

The biperiodicization of post-processed fields is treated by a specific subroutine: FPEZO2H. Note that the content of this subroutine is "unbalanced" since both biperiodicizations for the auxiliary surface fields and the physical fields are treated within the same call, while the symmetry of the code structure indicates clearly that the two biperiodicizations should be performed separately.

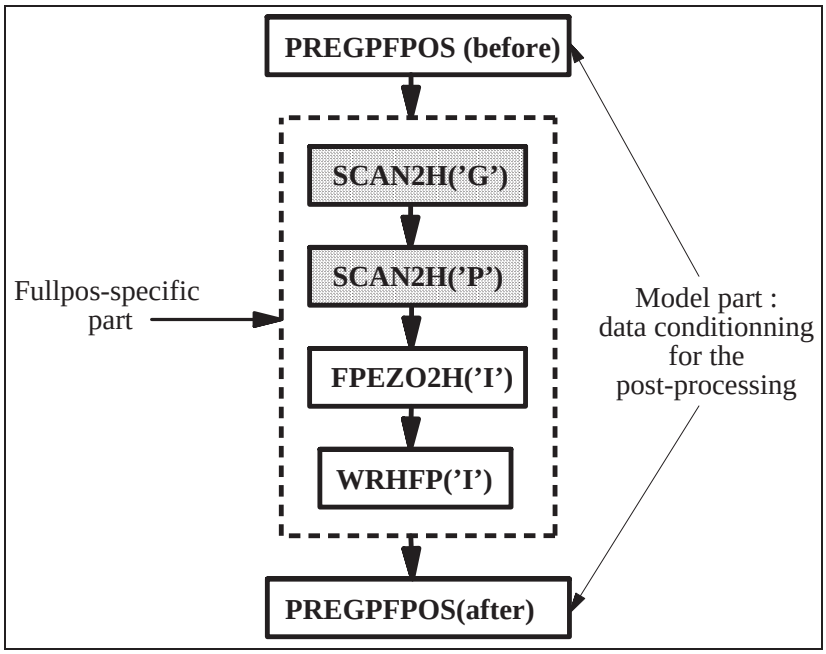

Figure E.8 *Physical fields post-processing: GRIDFPOS. The greyish areas correspond to model subroutines.*

The writing out to files of the gridpoint post-processed fields is performed by the leading subroutine WRHFP (for ARPEGE/ALADIN at least), with a configuration letter.

Remark: GRIDFPOS is actually a model-dependent subroutine, because it contains the subroutine PREGPFPOS which purpose is to pack the data before, and restore them after, in order to enable the in-line/off-line reproductibility of the post-processing. All that is between the two invocation of PREGPFPOS is purely specific to FULLPOS, and thus it should be isolated in a specific subroutine; while PREGPFPOS and the container subroutine should not be subject to externalization, unless PREGPFPOS would be an external module called by the post-processing (another problem of interfacing!).

Further notices about PREGPFPOS:

- There is no reason to have one single subroutine for both operations "before" and "after": there should be one subroutine for "before" (PREGPFPOS) and one for "after" (POSTGPFPOS?); but both subroutines could be contained in the same module.
- To avoid evident duplication of code, the subroutine should treat only one gridpoint buffer, and then called three times.

(*g*) *Dynamic and physico-dynamic post-processing*

Figure E.9 on page 153 describes this subroutine.

The dynamic and physico-dynamic post-processing is based on the following concepts:

- It uses the model subroutine STEPO to combine spectral transforms, gridpoint calculations, spectral calculations and I/O operations.
- A "post-processing step" is composed of a succession of calls to STEPO, since one single call to this subroutine would not be enough to treat at once the vertical interpolations, the horizontal ones, the spectral fit and the spectral filters.
- To each kind of post-processing level corresponds a "post-processing step".

There are three kinds of post-processing steps:

The post-processing on P, P_v or θ levels : it is composed of three calls to STEPO:

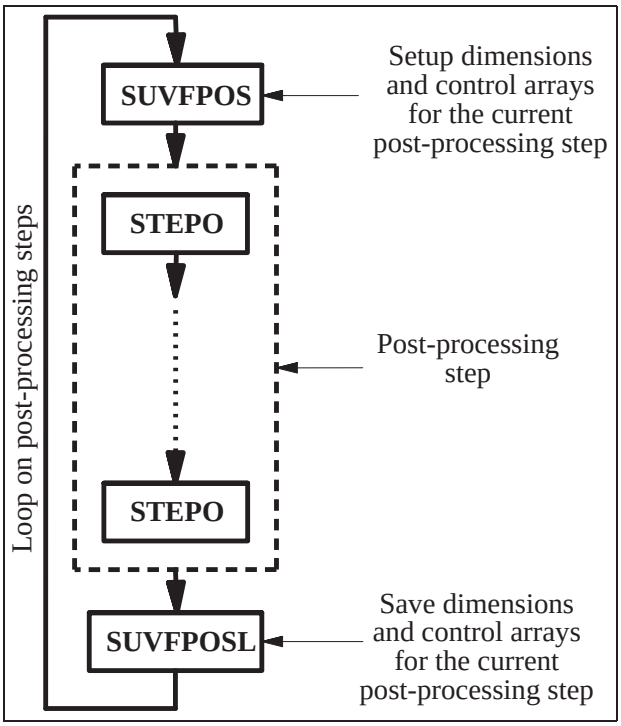

Figure E.9 *Dynamical and physico-dynamical post-processing: DYNFPOS.*

(i) vertical interpolations and spectral filters
(ii) horizontal interpolations
(iii) outputs

The post-processing on z or η levels : it is composed of four calls to STEPO:

(i) computation of the model primitive fields on the model levels
(ii) horizontal interpolations of the model primitive fields on the model levels
(iii) computation and vertical adjustment of post-processing fields
(iv) outputs

Notice: the first two sequences have been separated because such a conception was easier and faster. However they could now be merged: the idea would be to pipe the output array from POS with the input core array for horizontal interpolations. Anyhow this must not lead to duplication of code (VPOS to be re-visited?).

The post-processing on the originating model primitive fields and levels : this is an internal post-processing step dedicated to the external part of the "configuration 927", and composed of three calls to STEPO:

(i) computation of the model primitive fields on the model levels
(ii) horizontal interpolations of the model primitive fields on the model levels
(iii) outputs

Notice: this post-processing step is very close from the beginning of the previous one, except that the data flow in output is different (this is justifying another configuration sequence).

Remarks:

- The mechanism of STEPO is so that the writing out of the post-processing fields, for a given post-processing step, is always "postponed" to the beginning of the next post-processing step. This is

complicating the code because the descriptors control arrays, which are used to drive a given post-processing step, are overwritten (by the following post-processing step) before the current step is over; that is why (a part of) these descriptors must be saved before starting a new post-processing step.

- In the scope of the externalization and the simplification of FULLPOS, the model subroutine STEPO should be abandoned for the post-processing, and a new specific one should be written, with a well-adapted configuration string, like the following one:

  0. Model fields (adapted) inverse transforms
  1. Vertical interpolator
  2. Direct spectral transforms
  3. Spectral filters
  4. Inverse spectral transforms
  5. Horizontal interpolator
  6. Residual computations and vertical adjustments
  7. Biperiodicizator
  8. Output fields preconditioner
  9. Outputs

  The management of the I/Os is still wide open: what kind and conditionment for the input data and the output data: should the model inverse transform or the writing out to files be parts of the post-processing package or not?
- The output configuration letters from the subroutine SUFPCONF should be arrays dimensioned to the maximum number of scans, stored in a module and initialized in the setup (SUBFPOS).
- The double call to SUVFPOS is odd: there should be one (simplified?) subroutine to find out whether or not the sequence on model fields and levels should be performed or not.
- The loop on post-processing steps should be split in two parts: one for the P, P_v, θ levels and internal sequences; the other one for the z and η levels. In between, spectral matrixes arrays should be released in order to save memory.

Table E.1 on page 155 shows the STEPO configuration letters used for the post-processing.

However, all the combinations are not possible: they are build in function of the horizontal output format and the vertical kind of levels, as described in Table E.2 on page 156. Note that whenever the configuration letter "P" occurs, it can be replaced by "0" if the spectral fit for the current post-processing level has been switched off. But once the spectral fit is active, the program will always run through the spectral computations (even if the filters are inactive) because there happen arrays transfers.

Notice: the "configurations 927" (corresponding to LFPSPEC=.TRUE.) are normally performed only for η levels outputs.

Figure E.10 on page 157 shows the ingredients of the subroutine STEPO used for the post-processing.

WRHFP, WRSFP (WRite Horizontal fullPos, WRite Spectral FullPos): To write out post-processed data. These subroutines are embedded inside IOPACK but are specific to FULLPOS .

ESPFP : to precondition some spectral fields before writing them out. This subroutine is specific to ALADIN inFULLPOS: it is used in the case of plane geometry to go from the reduced variables to the geographical ones.

FPEZO2H : to perform the biperiodicization of the fields if needed (see GRIDFPOS above).

PRESPFPOS (PREpare SPectral FullPOS): to pack the spectral data and the surface fields before, and restore them after, in order to enable the in-line/off-line reproductibility of the post-processing. Like the subroutine PREGPFPOS, it should rather be considered as a model-specific subroutine.

TRANSINVH/ETRANSINVH : Inverse spectral transforms. Note that the data can be either the model trajectory or a post-processing flow (externalization to be prospected).

Table E.1 *STEPO configuration letters used for the post-processing.*

Sequence	Function	Possible values
(1)	Outputs	"E" : P levels "U" : z levels "Y" : P_v levels "M" : θ levels "Z" : η levels
(2:3)	Inverse spectral transforms	"A" : on model fields "P" : on post-processing fields "0" : No inverse transforms
(4)	Not used	"0"
(5)	Gridpoint computations	"B" : vertical interpolations on P levels "V" : vertical interpolations on P_v levels "T" : vertical interpolations on θ levels "H" : vertical interpolations on z levels (above model surface) "S" : vertical interpolations on η levels (above model surface) "Z" : vertical interpolations on z levels (above output surface) "E" : vertical interpolations on η levels (above output surface) "M" : computation of the model fields on the model levels "A" : horizontal interpolations of vertically post-processed fields "G" : horizontal interpolations of auxiliary surface fields "P" : horizontal interpolations of physical fields "I" : horizontal interpolations of model fields
(6)	Not used	"0"
(7:8)	Direct spectral transforms	"P" : on post-processing fields "0" : No inverse transforms
(9)	Spectral computations	"P" : active "0" : not active

SCAN2H : the model gridpoint head subroutine is invoked with a specific configuration string (see GRIDFPOS above).

TRANSDIRH/ETRANSDIRH : Direct spectral transforms. They are performed on post-processing data exclusively. Post-processing-specific subroutines are embedded inside.

SPOS/ESPOS : (Spectral POSt-processing): to perform computation in spectral space for the post-processing. These routines are not only used to filter fields, but also for other operations. They are embedded inside the model subroutines SPCH and ESPCH.

(*h*) *Gridpoint calculations*

This section will describe the mechanism of post-processing computations in gridpoint space, which is embedded inside the model subroutines SCAN2H and SCAN2MDM. In the scope of the externalization of FULLPOS, it will be necessary to leave these interfaces.

Horizontal interpolations Following the mechanism of "semi-Lagrangian buffers" at the time of the shared memory architecture, the horizontal interpolations have been conceived as a succession of two subroutines, respectively:

HPOS (Horizontal POSt-processing): to copy the fields to be interpolated in a "core-array".

Table E.2 *Possible combinations of* STEPO *sequences.*

	Gridpoint	Spectral (CFPFMT='MODEL')	"927" (LFPSPEC=.TRUE.)
P	. A A 0 B 0 P P P 0 P P 0 A 0 0 0 0 E	. A A 0 B 0 P P P E	. A A 0 M 0 0 0 0 0 0 0 0 A 0 0 0 0 Z 0 0 0 0 0 0 0 0 0 A A 0 B 0 P P P E 0 0 0 0 0 0 0 0
P_v	. A A 0 V 0 P P P 0 P P 0 A 0 0 0 0 Y	. A A 0 V 0 P P P Y	. A A 0 M 0 0 0 0 0 0 0 0 A 0 0 0 0 Z 0 0 0 0 0 0 0 0 0 A A 0 V 0 P P P Y 0 0 0 0 0 0 0 0
θ	. A A 0 T 0 P P P 0 P P 0 A 0 0 0 0 M	. A A 0 T 0 P P P M	. A A 0 M 0 0 0 0 0 0 0 0 A 0 0 0 0 Z 0 0 0 0 0 0 0 0 0 A A 0 T 0 P P P M 0 0 0 0 0 0 0 0
z	. A A 0 M 0 0 0 0 0 0 0 0 I 0 0 0 0 0 0 0 0 Z 0 0 0 0 U	. A A 0 H 0 P P P U	. A A 0 M 0 0 0 0 0 0 0 0 A 0 0 0 0 Z 0 0 0 0 0 0 0 0 0 A A 0 H 0 P P P U 0 0 0 0 0 0 0 0
η	. A A 0 M 0 0 0 0 0 0 0 0 I 0 0 0 0 0 0 0 0 E 0 0 0 0 Z	. A A 0 S 0 P P P Z	. A A 0 M 0 0 0 0 0 0 0 0 A 0 0 0 0 Z 0 0 0 0 0 0 0 0 0 A A 0 Z 0 P P P Z 0 0 0 0 0 0 0 0

HPOSLAG : (Horizontal POSt-processing, LAGged part): to interpolate the fields (the suffix "LAG" recalls the former shared memory architecture where the call to this subroutine had to be synchronized with the corresponding call to HPOS).

Figure E.11 on page 158 describes this general mechanism.

Between these two subroutines, the "core-array" is surrounded by a "halo" of data from the neighbouring processors. This part is common with the model. In the scope of the externalization of FULLPOS, it is supposed to be externalized as well; it could be the opportunity to write a specific interpolator, well-adapted to an irregular string of output points.

Today HPOSLAG has become a useless interface which calls a main subroutine for the management of horizontal interpolations: FPOSHOR (FullPOS HORizontal).

Notice: in a new modular framework, the general organization should be composed of three successive subroutines:

(i) HPOS: to fill the core array
(ii) HPOSLAG: to fill the halo
(iii) FPOSHOR: to interpolate.

The subroutine FPOSHOR, as shown on Figure E.12 on page 159, contains the following elements:

SC2RDGFP : To extract data from a "fullpos buffer". Three of them are used here:

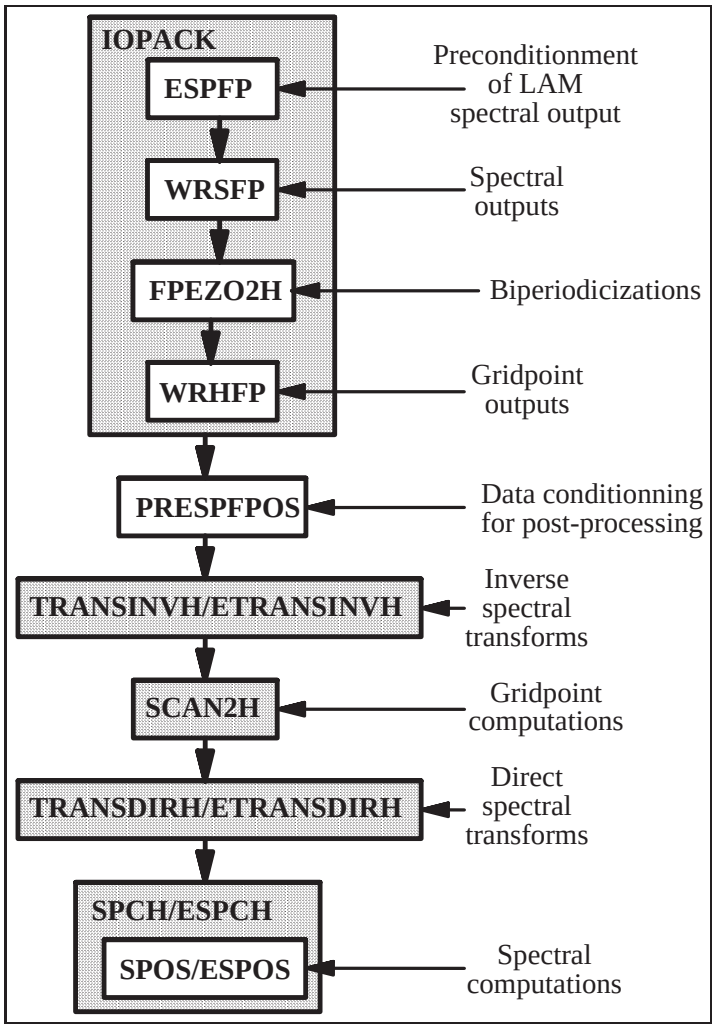

Figure E.10 *Elements of* STEPO *used for the post-processing. The greyish areas correspond to model subroutines.*

(i) The weights and indexes for interpolations (refer to SUWFPDS and SUWFPBUF).
(ii) The output climatology and geometry (refer to SURFPDS and SURFPBUF).
(iii) The auxiliary (pre-interpolated) surface fields (refer to SUFPTR2 and SCAN2H('G')).

FPSCAW : To compute the exact addresses of the neighbouring input points for interpolations.

FPINTDYN/FPINTPHY : Fields basic interpolators, respectively for dynamics and physics.

FPCORDYN/FPCORPHY : Fields basic correctors after interpolations, respectively for dynamics and physics.

FPGEO : To convert wind-related fields to the output compass and map factor.

SC2WRGFP : To write out the interpolated data in a "fullpos buffer".

FPCLIPHY : To use the output climatology rather than interpolate (for the appropriate fields only!)

FPNILPHY : A strange subroutine which controls that the remaining fields are proper for interpolations, and which perform the appropriate computations for those which should not be interpolated.

Remark: to enable the use of all this mechanism for the pre-interpolation of the model land-sea mask (instead of duplicating the code — refer to CPCLIMI —), the buffer containing the weights and indexes should be split: one for the land-sea-dependent fields and one for the "standard" fields.

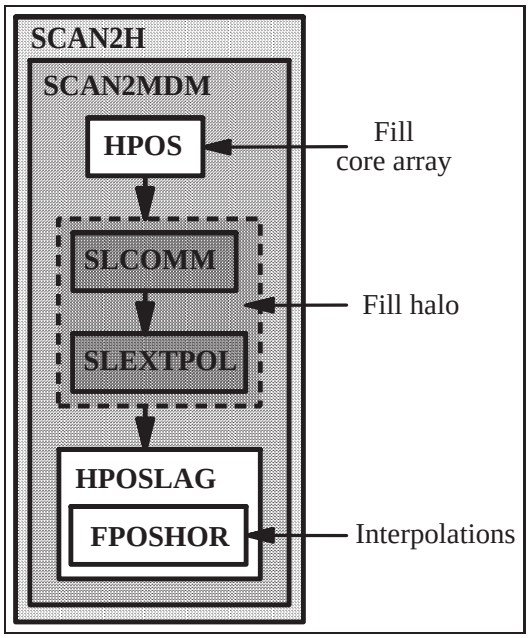

Figure E.11 *General mechanism of horizontal interpolations. The greyish areas correspond to model subroutines.*

Vertical interpolations, re-adjustments and physico-dynamic calculations The vertical interpolations, which are performed on the model grid, are simply controlled by a post-processing-specific interface named VPOS; as shown on Figure E.13 on page 159, it contains the following main subroutines:

POS : To perform vertical interpolations.

PHYMFPOS (PHYsical Meteo-France POStprocessing): To compute physico-dynamic fields on the model grid (used only if the output grid is the model grid).

Notice:

- The conditions of calls to POS and PHYMFPOS are unproper: they should be limited to the occurrence of any field subject to vertical interpolations for POS, and to the occurrence of any field subject to physico-dynamic computations on the model grid for PHYMFPOS. QFPTYPE to be extended in this sense?
- The interface between VPOS and the model has become odd: while the physical fields are read inside VPOS, the dynamic ones are obtained via the subroutine interface. This should be harmonized (all the model data to be read inside).

The vertical re-adjustments, which are performed on the output grids (because they have to be computed after the horizontal interpolations of the model primitive fields), are simply controlled by a post-processing-specific interface named ENDVPOS; as shown on Figure E.14 on page 160, it contains the following subroutines:

SC2RDGFP : To extract data from a "fullpos buffer". Three of them are used here:

(i) The horizontally interpolated model dynamic fields.
(ii) The output climatology and geometry (refer to SURFPDS and SURFPBUF).
(iii) The auxiliary (pre-interpolated) surface fields (refer to SUFPTR2 and SCAN2H('G')).

ENDPOS (END POSt-processing): To perform vertical re-adjustments and physico-dynamic calculations

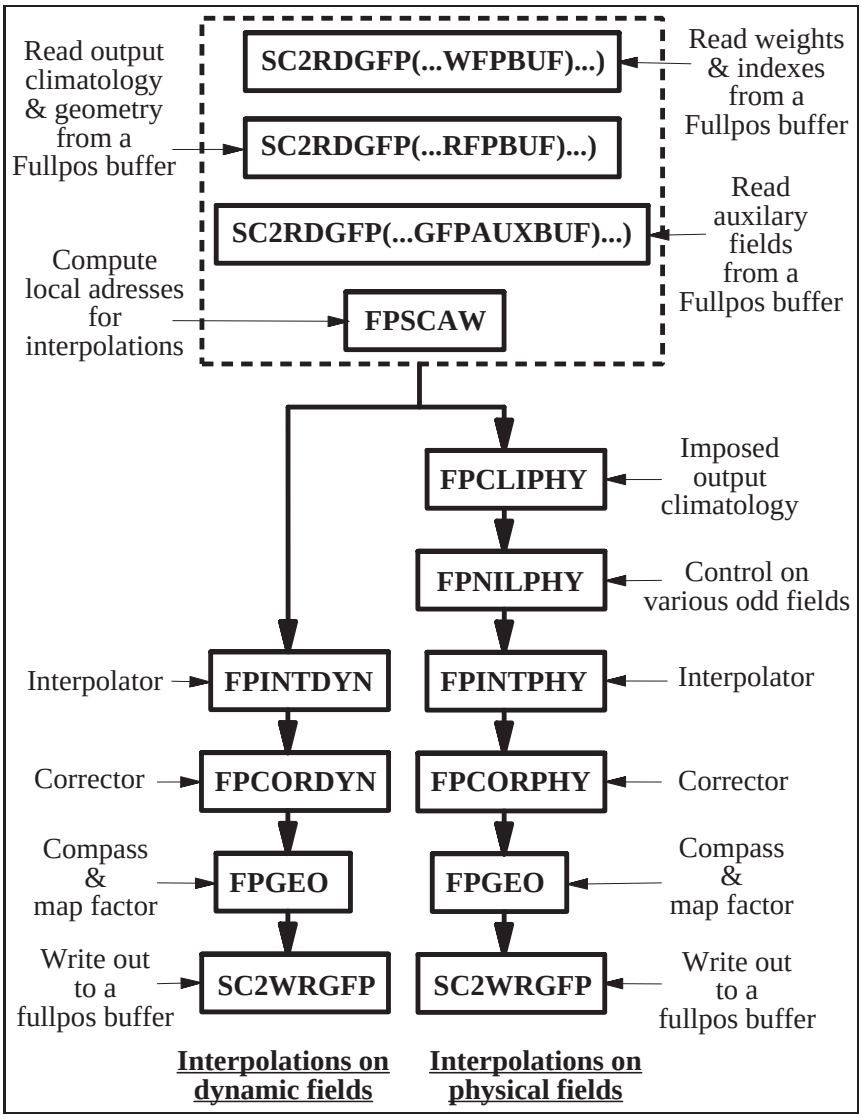

Figure E.12 *Horizontal interpolations management: FPOSHOR.*

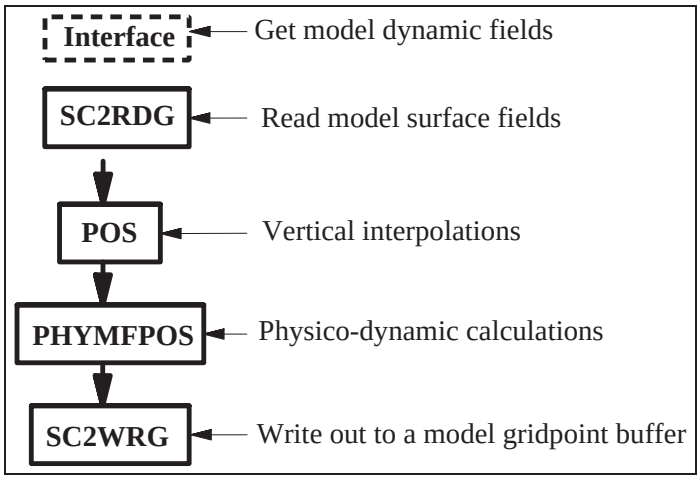

Figure E.13 *Vertical interpolations and physico-dynamic calculations on input grid: VPOS.*

SC2WRGFP : To write out the interpolated data in a "fullpos buffer".

Notice:

- In ENDVPOS the loop on subrows should be put outside the subroutine to make easier a further distribution (OPEN-MP).
- POS/PHYMFPOS on one side, and ENDPOS on the other side are similar. In the scope of a further harmonization of the code, one should investigate how to re-organize all these subroutines.

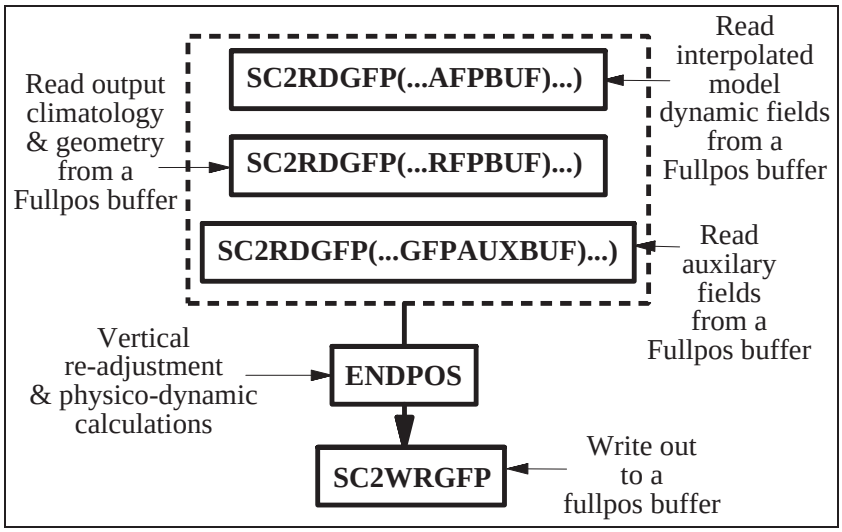

Figure E.14 *Re-adjustments and physico-dynamic calculations on output grids: ENDVPOS.*

Biperiodicization The biperiodicization is performed in the same spirit as the horizontal interpolations: somehow the "interpolations" are replaced by "extrapolations".

The control subroutine, as seen before, is named FPEZO2H. Figure E.15 on page 161 describes its mechanism.

EPOS (Extension zone POSt-processing): To fill a "fullpos buffer" with fields to biperiodicize. This subroutine is the counterpart for HPOS.

ENDEPOS (END Extension zone POSt-processing): To correct the extended fields after the biperiodicization. This subroutine is the counterpart of a part of FPOSHOR.

FPEZONE (FullPos Extension ZONE): To perform the biperiodicization itself. This subroutine has the specificity that its distribution is odd: since the calculation itself is not (yet?) distributed, the distribution is performed on the global fields.

Notice:

- FPEZO2M is nothing but an interface to control the former multitasking system. It should be removed now.
- FPEZONE is supposed to be externalized, or more exactly: its main subroutine FPBIPER. This externalization should naturally fit the spectral transforms ... and their distribution! Therefore the distribution of FPBIPER should be investigated: then FULLPOS would be adapted.
- For the clarity of the code, it would be nice to have a straight symmetry between the interpolations and the biperiodicization! But this depends on the biperiodicization distribution.

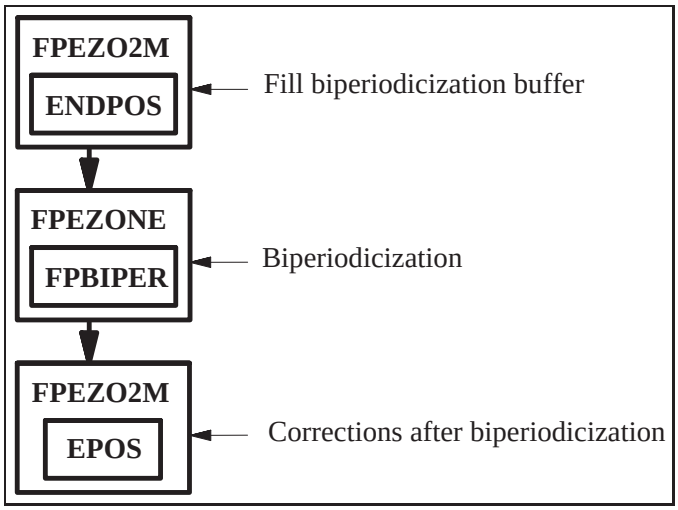

Figure E.15 *Biperiodicization: FPEZO2H.*

E.2.2 Data flow

This section will describe the principal data arrays used in FULLPOS.

(*a*) *Spectral arrays*

FULLPOS is (indirectly) using the model spectral arrays (SPA2 and SPA3) as the input of the model inverse spectral transforms. It uses also specific spectral arrays which will contain the spectrally fitted fields to be filtered or written out to files. As FULLPOS uses the tfl/tal spectral transforms packages, these arrays are all shaped and distributed in the same manner.

Open question: in the scope of the externalization of FULLPOS, should we consider that the model inverse transforms is an external subroutine used inside the post-processing package? This could be a solution to improve the portability of the externalized FULLPOS in other applications (with 100% gridpoint input data for instance).

There are three specific spectral arrays:

SPAFP : it contains the vertically post-processed fields to be spectrally fitted which are not "derivative fields" to be filtered in the homogeneous high resolution space. SPAFP is output from TRANSDIRH and input to SPOS/ESPOS.

SPDFP : it contains the vertically post-processed fields to be spectrally fitted which are "derivative fields" to be filtered in the homogeneous high resolution space. SPDFP is output from TRANSDIRH and input to SPOS/ESPOS.

SPBFP : it contains the vertically post-processed spectral fields which are "derivative fields" filtered in the homogeneous high resolution space. In this array, for each field, there is one spectrum per field and per subdomain. SPBFP is output from SPOS/ESPOS and input to TRANSINVH.

(See Figure E.16 on page 162.)

Remarks:

- Though there were no reasons to do it, the spectral data flow in ALADIN has been split like in ARPEGE to stick to the same code structure.
- SPAFP and SPDFP had been split for technical reasons, after an old conception. In cycle 26, all these arrays have been unified in one single array SPBFP, thanks to a more clever monitoring of the post-processed fields.

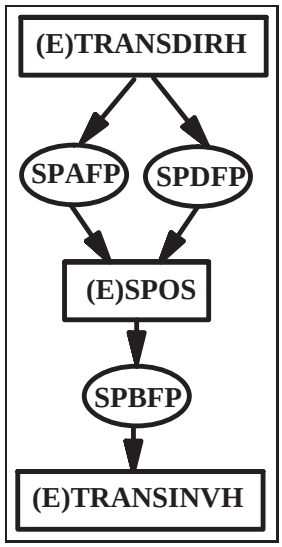

Figure E.16 *Spectral post-processing data flow.*

(*b*) *Gridpoint buffers*

By "gridpoint buffers" one should understand the model gridpoint buffers, that is those which are shaped to fit the model geometry and its distribution.

These buffers (or arrays) are used to interface:

- the model and the post-processing in general (GPPBUF, GFUBUF, XFUBUF for the physical fields; GPP and GPUABUF for the dynamic fields; various geographical data arrays from the module YOMGC);
- the vertical post-processing and the spectral transforms (GPP is re-used as output from VPOS);
- the inverse spectral transforms and the horizontal post-processing (GPP is used as output from the inverse spectral transforms and input to HPOS);
- the vertical post-processing and the horizontal interpolations (GAUXBUF used for the fields which are not concerned by the spectral fit).

Figure E.17 on page 163 gives more information about the relations between the main subroutines and these buffers/arrays. Note that the model array GPP is re-used as input/output forFULLPOS: this is embarrassing for the externalization.

(*c*) *Fullpos buffers*

By "Fullpos buffers" we shall point out the gridpoint buffers specially designed for the post-processing output.

How are they shaped?

Figure E.18 on page 164 helps to understand the problem:
Given the model gridpoint area and its distribution, it appears that from one processor to another, the number of "output points" (ie: the points where to the model fields will be interpolated) can be quite different. This is even more obvious when the input model is stretched and the target grid is not, or when the output grid is centered on the pole of interest of the input model (which is typically the case of the ARPEGE/ALADIN coupling).

At the time of the shared memory code and the I/O scheme, this was really a complication because to a model "row" (with a fixed size NPROMA) we had to affect a post-processing "row" of points with a variable size. To solve this problem we had to consider that the post-processing packets were composed of subrows of fixed size NFPROMAG.

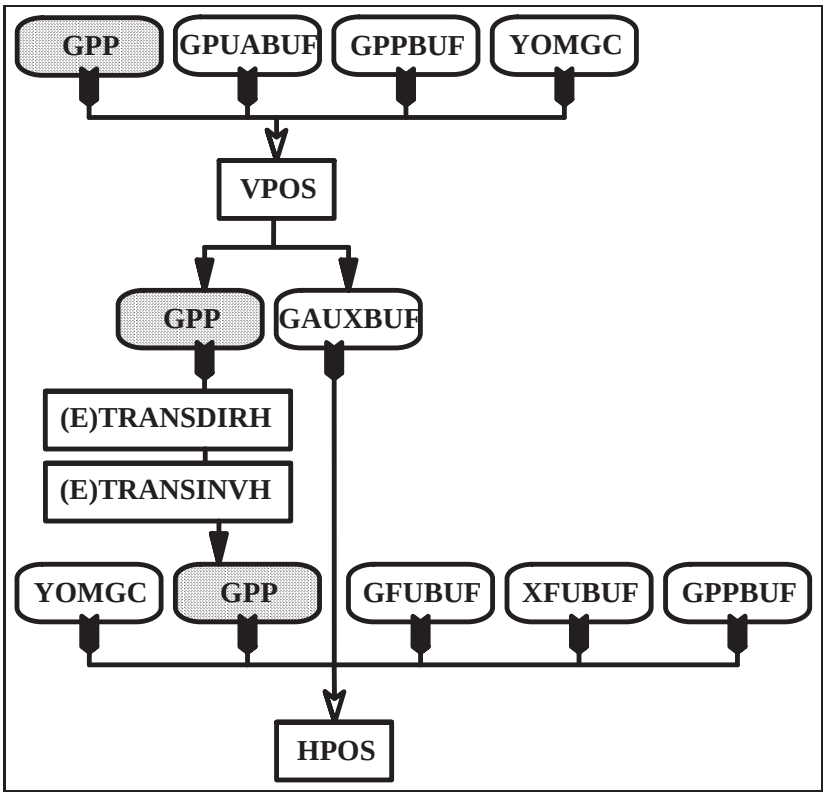

Figure E.17 *Gridpoint buffers interactions.*

Things have been much easier to conceive since the removal of the I/O scheme and the new distributed architecture: the "Fullpos buffer" is very close to the model "Gridpoint buffer", since only the characteristics have different numerics: the size is NFPRGPL instead of NGPTOT; the segmentation is NFPROMAG instead of NPROMA (however, this last distinction has been purely formal since the distributed memory architecture, while it was of prime importance for the parallelism in the shared memory architecture).

Remarks:

- The code is still containing a lot of items which were used for the shared memory architecture and which have become completely useless today (cleaning to be done!).
- "Fullpos buffers" and "Gridpoint buffers" should remain formal different structures to make the externalization of FULLPOS easier. "Fullpos buffers" should even be able to setup "Gridpoint buffers"-like structures on demand, in order to help the externalization (so that buffers like GAUXBUF, which are internal to FULLPOS, would not need the model structure). Another possibility would be to create an external common structure for the model and the post-processing (to be archived in xrd library?).
- "Fullpos buffers" are also already used to handle the biperiodicization, with different characteristics: NFPEL instead of NFPRGPL and NFPROMEL instead of NFPROMAG.
- Distribution on horizontally interpolated data may be better balanced. But how to do it is a big deal! The solution would be to distribute as equally as possible the geographically sorted string of post-processing points, and then to adapt the construction of the cores and halos. This leads to a re-distribution of the model gridpoints after the vertical interpolations and before the horizontal ones, and thus to a post-processing-specific horizontal interpolator system (good for externalization!): see Figure E.19 on page 164.

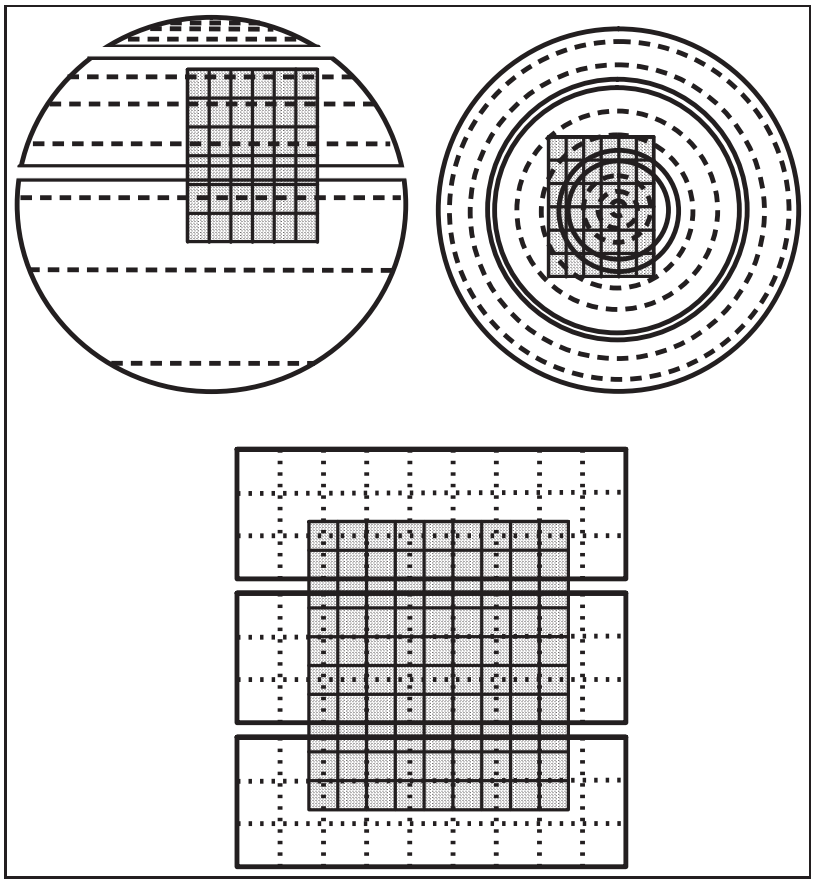

Figure E.18 *Gridpoints repartitions between model and output grids: the greyish areas corresponds to an output grid; the background areas corresponds to the input model, with a physical separation between processors. Top left represents a typical coupling from a stretched grid, Top right a typical coupling from a rotated grid, bottom a typical nesting.*

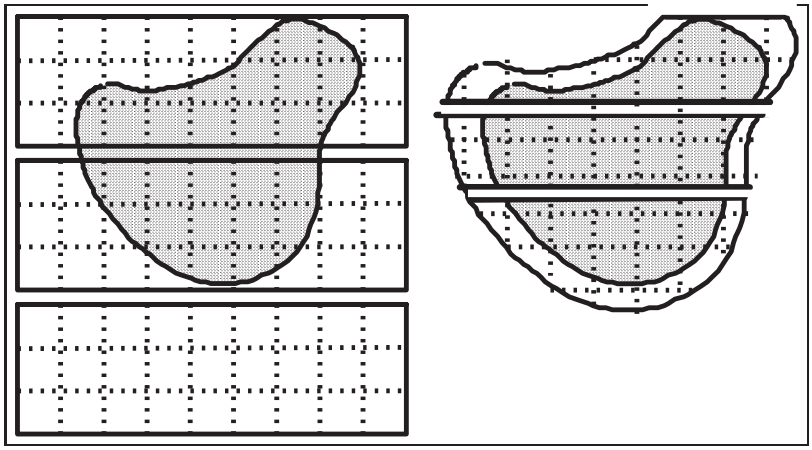

Figure E.19 *"Next generation" fullpos horizontal interpolator: the cores and halos are re-built to fit the post-processing distribution.*

As shown in Figure E.20 on page 165, these buffers are used to interface the vertical interpolations (FPOSHPOR, according to its configuration), the vertical re-adjustment and physico-dynamic calculations on the target grids (ENDVPOS), and finally the writing out to files (WRHFP).

The used buffers are:

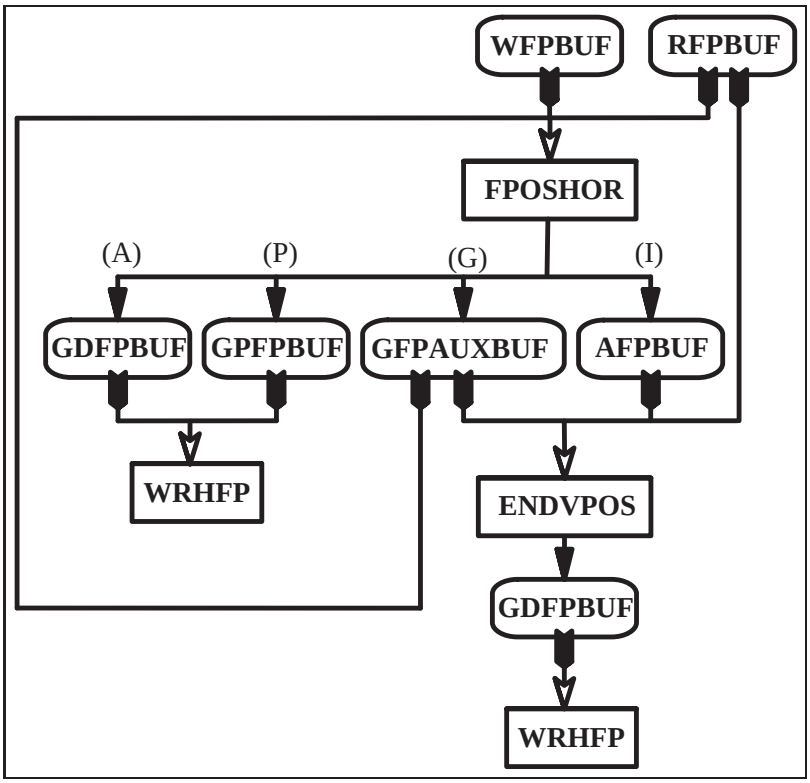

Figure E.20 *Fullpos buffers interactions in the horizontal interpolations.*

`WFPBUF` : weights and indexes for interpolations

`RFPBUF` : output climatology and geometry

`GFPAUXBUF` : auxiliary surface fields

`GPFPBUF` : output physical fields

`GDFPBUF` : output dynamic fields

`AFPBUF` : horizontally interpolated model fields

Notice: `GPFPBUF` and `GDFPBUF` are never used simultaneously; so they should be merged in a single buffer (`FFPBUF` like "Final FullPos BUFfer"?).

As shown in Figure E.21 on page 166, these buffers are also used to interface the biperiodicizations; the used buffer, named `EZOBUF`, contains either the extensions zone of the physical fields or the extensions zones of the dynamic fields (according to the configuration of `FPEZO2H`).

E.2.3 Monitoring

This section will describe how the post-processed fields are monitored through the software.

(*a*) *Physical fields*

The monitoring of the physical fields is rather easy, since there are only horizontal interpolations (at least for the time being, since the upper air fluxes are not yet post-processable). However it has become old-fashioned with regards to the FORTRAN 90 facilities, and it is heavy to use. This monitoring should be revisited to make the maintenance easier and to enable the treatment of upper air fluxes.

The characteristics are the following:

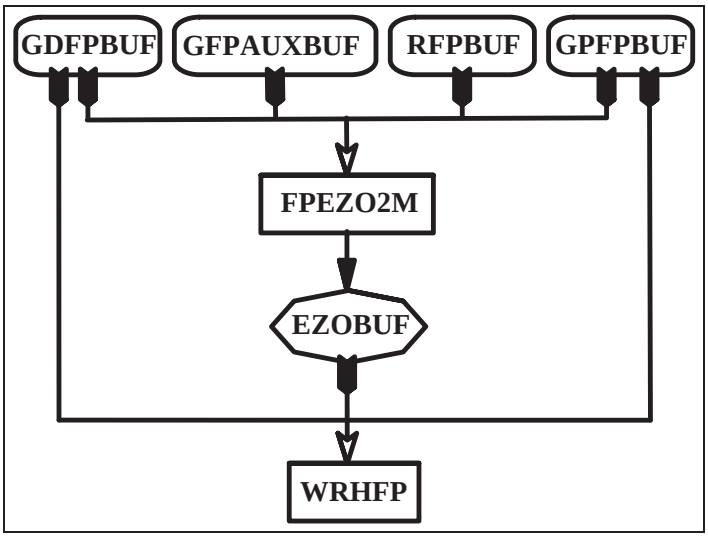

Figure E.21 *Fullpos buffers interactions in the biperiodicizations.*

- The fields are stored in the following order:

  (i) Surface fields (from GPPBUF)
  (ii) Cumulated fluxes (from GFUBUF)
  (iii) Instantaneous fluxes (from XFUBUF)

  This rule should be followed through the whole code in order to locate the fields.
- The fields are characterized by:

  - The ARPEGE/ALADIN field name.
  - The number of bits for packing before writing out to file.

  The setup is performed in YOMAFN, NAMAFN, SUAFN1, SUAFN2 and SUAFN3 (Note that the tables numbering is fragile). Then the fields will be always recognized by their ARPEGE/ALADIN name.

The steps to be followed in order to add a new field are then the next ones:

- For a given post-processing field, all the needed model fields should be at disposal: FPINIPHY/SUFPCFU/SUFPXFU according to the field kind.
- Each field is associated to one or more model fields to fill the interpolations buffer: HPOS.
- Each field is associated to one or more interpolated fields to fill the biperiodicization buffer: FPFILLB.
- Interpolations and control of interpolations may occurs in the following subroutines: FPINTPHY, FPNILPHY, FPCLIPHY, FPCORPHY, FPGEO.

(*b*) *Dynamic fields*

The monitoring of the dynamic fields now uses complex derived types from the FORTRAN 90 language. The aim is to setup all the characteristics which will be used to monitor the fields, so that the fields could be handled in an almost blind way.

The monitoring is realized with two kinds of specific derived types (both declared in the module file `fullpos_descriptors.F90`)

(i) `fullpos_descriptor`:
a static derived type, describing the fixed characteristics through the post-processing for all the fields which can be requested. This type is set in YOMAFN, NAMAFN, SUAFN1, SUAFN2 and SUAFN3. Notice: the item %ILMOD is set in SUFPDIM; this item, together with %LLGP and %IBIT are a bit odd since they depend on the model namelist (to be revisited). This type is partly external, as it is partly accessible from the namelist NAMAFN.

There is one type for each field, and one type array containing all of them (named: TFP_DYNDS). The fields are then recognized by their rank in this type array (it plays the role of an internal code).

(ii) fullpos_request:
a dynamic derived type named QFPTYPE, describing the changing characteristics of each field along the post-processing. This type is set in CPVPOSPR, SUVPOS and UPDVPOS (in cycle 26: only in CPVPOSPR and SUVPOS). It is then used in a large amount of subroutines. It is a purely internal variable type. The purpose of SUVPOS is to set the fields characteristics, while CPVPOSPR initializes the pointer of each field in each space. This last routine is crucial to the dataflow between the gridpoint space and the spectral space: it enables the gridpoint fields to be stored in the proper order for the spectral transforms.

The steps to be followed in order to add a new field are then the next ones:

- To add a new fullpos_descriptor type for the new field and set it in YOMAFN, NAMAFN, SUAFN1, SUAFN2 and SUAFN3.
- To possibly modify SUFPDIM if the field should be considered as a new pronostic field.
- To compute the field in POS/ENDPOS using the fullpos_request type QFPTYPE and the individual fullpos_descriptor types.
- To possibly control the horizontal interpolation of this field in FPCORDYN, using the individual fullpos_descriptor types.

Appendix F
Coding standards

Table of contents

F.1 INTRODUCTION

NWP software, which is often used for both operations and research, aims to be homogenous, portable, modifiable and maintained with much flexibility and ease.

This represents a lot of constraints, to be shared by all participants of the ARPEGE/IFS and ARPEGE/ALADIN projects, and even more when cross-collaborations with other projects are included.

While the computing environment of the ARPEGE/IFS software was originally a shared-memory, multi-processor vector machine, things have changed so that the computing environment is much wider: ranging from a single workstation to a cluster of high performance servers: scalar or vector processors, distributed memory or partially shared memory machines.

Last but not least, scientific and technical developments provide a strong justification for a perpetual evolution of the codes in order for the software to perform optimally, and this should be achieved using a minimum of time and human resources.

All these requirements need a set of agreed coding standards, that will be used by all participants of the project.

First of all we have to choose a language. But it is not enough to "just code in the same language". There are many ways to write the same program. Though different styles can give the same numerical result, they would not be equivalent once we consider further criteria telling that the software should be:

- well studied and well analysed
- well written (... but what does it mean?!)
- properly documented
- portable
- efficient
- flexible
- possibly exchangeable

For a long time the best solution to achieve this was to use the so-called "DOCTOR norm" (DOCumentary ORiented norm) by J. K. Gibson from ECMWF. Time has proven its efficiency.

Then the evolution of machines and languages as well as the increase of European collaborations has lead us to modify or extend the original norms. A European standard has been developed in order to facilitate the exchanges of code between meteorological organizations.

We can point out a few aspects of these norms:

- The existence of comments and especially comments at the start of each routine. Those concerning the modifications appear to be very useful to trace the history of the code. However, the 3-level structure to enable an automatic extraction of the comments never seems to have been used. Nevertheless an automatic extractor to document the ARPEGE/IFS/ALADIN namelist variables now exists.
- The structure in sections and sub-sections with homogenous labels improve the readability of the code, which is even more important than the norm itself.
- The convention of prefixes in order to rapidly identify the type (integer, real ...) and the nature (local or shared, dummy or not, ...) of the variables appeared to be the most popular aspect of the norm.

This document aims to collect all the conventions and customs used in the ARPEGE/IFS/ALADIN software. Hopefully it could be the starting point to code an automatic verificator of the norm. Finally only the norms tested in this automatic verificator would be the official norms while the others would be just recommendations. Developers should be aware that some rules could cause a lot of merging problems, especially where the same lines have been re-shuffled by more than one developers. Therefore we would not advocate to have all the standards systematically enforced by an automatic corrector.

In this document each item is referenced by a label composed of a topic and a number. There are four defined topics:

PRES for the presentation of the code,

NORM for the respect of the norm,

CTRL for the control of the code,

CCPT for the conception of the code.

This document was originally authored by R. El Khatib (MÉTÉO-FRANCE - CNRM/GMAP). It has been cosmetically edited for the IFS documentation.

F.2 SPECIFICATIONS

A well-thought out program is less difficult to code, produces fewer bugs and is often easier to maintain. So it is essential to specify the work with care. Three important aspects should be considered: the documentation, the code conception and its further enhancements, the code validation and maintenance.

F.2.1 Documentation

Documentation is essential: modification and maintenance will be much easier if everyone can understand not only the code, but also the design spirit behind the code. When changing the code the documentation should be updated immediately to avoid misunderstandings.

For the international cooperation to work, the documentation should be written in English.

There should be two kinds of documentation:

An external documentation which will be provided for a package of subroutines rather than an individual one. It should be written outside the code and divided into three parts (Andrews *et al.*, 1995):

(i) *Scientific documentation* describing the scientific aspects of the problem and the solution adopted in the current software. **This documentation should not refer to the code itself**.
(ii) *Technical documentation* describing the implementation of the solution adopted in the scientific documentation. This documentation should include a calling tree and a description (name, purpose) of all the modules which are used (subroutines, functions, data modules). This documentation should take over from the scientific one when technical aspects are concerned. Information for testing, modifying and maintaining the code should be included inside such documentation.
(iii) *A users guide* describing the user interface, switches and tunable variables of the software (access, default values and range).

Internal documentation which should be provided for individual modules (data modules or procedures). This documentation can be divided into three categories:

(i) *Header comments* stating briefly the purpose of the module, the author, references to external documentation, list of modifications (author and purpose) since the creation of the module. When dummy arguments are used, this header should describe them.
(ii) *Section comments* splitting the code into logical sections (that may be related to the scientific documentation). They indicate, section by section, the purpose of the code.
(iii) *Supplementary comments* which should help reading the code. There should not be many of them: source code which is interspersed with many comments is difficult to follow and to understand. If the code has been well designed and if the related external documentation has been properly written before coding, then the header and section comments should be sufficient.

Both kinds of documentation should be updated at the same time in order to avoid misunderstandings.

F.2.2 Code conception

- At the design stage, one should consider how the system should be tested, how it could be modified later and how it will be maintained. Future objectives of the system — not only the immediate objectives — should be considered: future enhancements should be anticipated and planned if possible, and should be made possible with a minimum of disturbance to the whole system.
- The different parts of the system should be analysed and planned. The parts should be designed in a modular way, with interaction between them based on a hierarchical and tree-like structure. There should not be any duplication of code: neither real duplications (when a piece of code is copied then pasted) nor virtual duplications (when more than one procedure has the same purpose). Duplication of code increases the problems of maintenance. Whenever code duplication is found the incriminated part of code should be re-designed.
- The relationship between modules should be simple. Individual modules should not be complex. Whenever a module is becoming complex after enhancements, the system should be re-examined

and the modularity re-designed. The longer subroutines are, the less readable they are and the more difficult they are to maintain.
- Derived types should be used where appropriate as that they make the code more robust and easier to maintain. Derived types naturally contribute to a more object-oriented code.
- Dataflow is a recurrent problem whenever portability is concerned. Therefore special attention should be given to it. Data inputs, outputs or transfers should be confined to a set of data handling modules, separated from the application modules.
- Non-standard statements of the language should not be used. In case they have to be, they should be confined into a subset of modules to limit the problems of portability. The same rule should apply to the invocation of routines coming from an external package.

F.2.3 Code validation and maintenance

- While validating and evaluating new code, the following questions should be considered:
  - Does the source code comply with the coding standards?
  - Is the code easy to understand?
  - Is the code unnecessarily complex?
  - Is the interface to the subroutine straightforward?
  - Does the routine produce the expected results?
  - Can modifications be made easily if required?
  - Are ALL error cases detected and properly acted upon?
  - Are ALL aspects of the calculation (ie: direct model, tangent linear, adjoint, limited area model versus global, ECMWF versus MÉTÉO-FRANCE setups) properly treated?
  - Is the routine efficient in terms of memory and CPU consumption?
  - Are the final integrated tests (ie: the operational configurations of the software) successful?
- The primary maintenance documentation is the source code. Only the source code is guaranteed to be up to date. Thus, **it is of vital importance to update source code comments while modifying the source code.**
- When code undergoes development for scientific and technical reasons, at some point it is likely to become unnecessarily complex. It is important to recognise this and when it occurs review and if necessary re-design and re-write the code package concerned.

F.2.4 Current code framework

- The ARPEGE/IFS/ALADIN code is written in FORTRAN 90 and C. The following document mainly applies to the part of the code written in FORTRAN, this being the main coding language in which the bulk of the code is written.
- The code is designed to perform well on both vector and cache based processors. This feature has to be maintained for the foreseeable future.
- It is parallelised for distributed memory computers using MPI.
- It is also parallelised for shared memory computers using OpenMP inside MPI. This implies that the code inside the OpenMP regions has to be written in a thread-safe way.

F.3 DESIGN

F.3.1 Typewriting style

Following the European standards (Andrews *et al.*, 1995) which are more restrictive than the ANSI standard the FORTRAN keywords may be written in upper case only, or with initial letter in upper case and the rest in lower case. The names of variables may be written in mixed lower or upper case and the names of namelists, modules, programs or subroutines may be written in mixed lower or upper case. No recommendation has been made concerning comments.

However the survey of the existing code shows that the whole executable lines are preferably written with upper case characters. Comments are preferably written with lower case characters except the first letter, or the first letter after each full stop. Emphasized words are written in upper case characters and bracketed with asterisks[1].

It is recommended to stick to a conventional typewriting style inside the whole code because this convention enables the developers to concentrate upon the semantic of the code and makes easier the use of automatic tools to manipulate the code (Hill, 1995).

In view of the apparent lack of rule concerning the typewriting style, the recommendation is to stick to the apparent habit:

PRES(01) Executable lines should be written using upper case characters.

PRES(02) Comments should be written with lower case characters except the first letter, or the first letter after each full stop. Emphasized words may be written in upper case characters.

The minimum recommendation would be to follow a consistent style throughout each module or subroutine.

F.3.2 Basic layout

(*a*) *Executable statements*

NORM(01) The code should be written in FORTRAN 90 free format, at least as far as science is concerned.
The use of FORTRAN ("FORmula TRANslator") is almost obligatory since:

- it fits the scientific topics
- it is portable
- it is well-known by most of the developers!
- there exist well optimised FORTRAN compilers for vector and scalar processors.

C language can advantageously be used for low-level subroutines.

PRES(03) The code should start at the column 1, unless it comes to any of the indentation norms as they will be described below.

PRES(04) The ending statement of a module or subroutine should repeat its name. For example: "`SUBROUTINE SUCTO ... END SUBROUTINE SUCTO`"

NORM(02) Tabulations must not be used: this ensures the code will look indented as desired whenever ported (also the use of the tab character is non ANSI).

PRES(05) One should avoid writing more than one statement per line (ie: avoid using the separator ";"). More than one statement per line might penalize the readability. If several statements should be grouped together then one may write a tiny subroutine that would be private and contained in the current subroutine.

[1] Apparently the heritage of an automatic documentation extractor.

CCPT(01) Subroutines should not have more than 300 executable statements (Gibson, 1986) (there are projects which are even more strict, see Hill (1995)). Each subroutine should have a maximum score of 400, based on the following measure (Gibson, 1986):

- each subroutine call has score 5
- other executable lines have score 1

There is no recommendation for a minimum score per subroutine.

CCPT(02) When the code is modified, it is easier to add or remove lines than modify existing ones. This is of special importance when merging code modifications from several developers. Therefore one should write the code in such a way that consecutive lines are as independent as possible. This would make the future merge of source code easier. Unfortunately this can make the code unnecessarily long. So this rule should be applied with care. Obviously it fits short codes.

(*b*) *Comments*

PRES(06) The comments should be written in English. They should not be written twice (for example: English and the programmer's native language), because it makes the code less readable.

PRES(07) Blank lines should remain empty: they should not start with "`!`"

(*c*) *Entry point and exit points*

CTRL(01) Each procedure should contain one entry point and at most two kinds of exit points: one normal return and one abnormal termination.

CTRL(02) The entry point should be at the top of the procedure.

CTRL(03) The normal return should be the bottom of the procedure. Consequently the use of the statement `RETURN` is discouraged. If there is only one `RETURN` statement at the end of a procedure it should be removed.

CTRL(04) Abnormal termination should be invoked with the specific subroutine `ABOR1` because this subroutine enables one to flush the output buffer and to release the processors which are not affected by the abnormal termination.

There can be more than one abnormal termination in the body of the subroutine.

Listing F.1 shows examples of entry/exit points.

F.3.3 Header comments

(*a*) *Data modules*

CCPT(03) Each data module should begin with documentation describing the general content of the module and the purpose of each declared variable.

PRES(08) In order to improve the readability, the namelist variables in a data module should be separated from the internal ones.

PRES(09) Each description line should be independent to enable an automatic extraction of the documentation.

PRES(10) The documentation should be separated from the starting statement with a blank line and it should finish with a comment line filled with minus signs.

Listing F.2 shows an example of a data module.

Listing F.1 *Examples of entry/exit points and errors handling.*

```
SUBROUTINE ERROR_DETECTION
   :
   :
USE MHOOK     , ONLY : LHOOK
   :
   :
IF (LHOOK) CALL DRHOOK('ERROR_DETECTION',0)
   :
   :
IF (IWORD /= ILEN) THEN
  CALL ABOR1 ('ERROR_DETECTION : MESSAGE 1 '// &
 &             'RECEIVED WITH WRONG LENGTH')
ENDIF
   :
   :
   :
IF (IERR > 0) THEN
  CL='MESSAGE 2 RECEIVED WITH WRONG LENGTH'
  WRITE(NULOUT,*) CL
  WRITE(NULERR,*) CL
  CALL ABOR1 ('FROM ERROR_DETECTION')
ENDIF
   :
   :
   :
IF (LHOOK) CALL DRHOOK('ERROR_DETECTION',1)

END SUBROUTINE ERROR_DETECTION
```

(*b*) *Procedures*

PRES(11) Each procedure should begin (Andrews *et al.*, 1995) with a documentation header as a set of comments containing:

- the *purpose* of the procedure
- the *interface* details, describing the dummy arguments in the same order as they are in the interface
- the *externals* or other subroutines called
- the *method* used in the application, where there is no further documentation to refer to
- a *reference* to further documentation
- the *author and date* of creation of the procedure
- the *modifications* applied since the creation of the procedure, with the author and date of modifications

PRES(12) The header documentation should be separated from the entry point statement with an empty line and it should finish with a comment line filled with minus signs.

PRES(13) The *modifications* comments should start with the template:
"`! Modifications`"
and should end with the template:
"`! End Modifications`".
In between all the modifications description should be written in the same style:
day(2 digits), month(3 characters), year(four digits) separated with a minus sign, then the author, then a description.

Listing F.2 *Example of data modules.*

```
MODULE YOMDATA

! Module showing the coding standards in
! ARPEGE/IFS/ALADIN.

!   NUMBER : Key telling what to do :
!            NUMBER = 0 => don't do anything
!            NUMBER = 1 => do this
!            NUMBER = 2 => do that
!   VALUE  : Tunable variable to do what you wish
!   ARRAY  : Mysterious data array

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

USE PARKIND1 , ONLY : JPIM     ,JPRB

IMPLICIT NONE

SAVE

INTEGER(KIND=JPIM)            :: NUMBER
REAL(KIND=JPRB)               :: VALUE
REAL(KIND=JPRB), ALLOCATABLE :: ARRAY(:,:)

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

END MODULE YOMDATA
```

Listing F.3 shows a header documentation for a procedure.

F.3.4 Declaring variables

(*a*) *Layout*

NORM(03) The use of `IMPLICIT NONE` statement is mandatory. It improves the portability of the code and helps in the detection of errors.

NORM(04) Hard-coded variables increase the problem of maintenance and can even be the cause of bugs, especially when they are used in a subroutine interface.
Hence it is much better to write: `CALL POSNAM(NULNAM,CLNAME)` than `CALL POSNAM(4,'NAMCTO')`

PRES(14) The declaration of variables should be separated from the header documentation with an empty line. It should finish with a comment line filled with minus signs.

PRES(15) The declarations of variables should be grouped according to their type and attributes.

NORM(05) The *statement* `DIMENSION` should not be used (but the *attribute* `DIMENSION` can be). The shape and size of arrays should be declared inside brackets after the variable name on the declaration statement.

NORM(06) The notation "`::`" should be systematically used after the type and attribute declaration, and before the name of the variable.

PRES(16) All the attributes of a given variable should be grouped within the same instruction. This makes it possible to visualize in a glance the characteristics of a variable or of a family of variables.

Listing F.3 *Example of header documentation and variables declarations in a procedure.*

```
SUBROUTINE CODESTY(KERR)

! Purpose :
! -------|\\
!   *CODESTY* : CODE STYLE : Show coding standards

! Interface :
! ---------
!   KERR    : Output error code of the subroutine

! Externals :
! ---------
!   None.

! Method :
! ------

! Reference :
! ---------
!   Coding standards in Arpege/Ifs/Aladin.

! Author :
! ------
!   19-Jul-2002 Ryad El Khatib         *METEO-FRANCE*

! Modifications :
! -------------
!   30-Oct-2003 M. Hamrud   Cleaning for Cycle 28
!   30-Feb-0000 A.N. Other  Imaginary modification ;-)
! End Modifications
!-----------------------------------------------------

USE PARKIND1 , ONLY : JPIM

USE YOMDATA  , ONLY : NUMBER    ,VALUE

IMPLICIT NONE

INTEGER(KIND=JPIM),INTENT(OUT)::KERR

INTEGER(KIND=JPIM),PARAMETER::JPLEN=16 ! Length of
                                       ! local message
CHARACTER(LEN=JPLEN)::CLMESS           ! Local message

!-----------------------------------------------------
 :
 :
END SUBROUTINE CODESTY
```

PRES(17) Templates like "`Dummy scalar arguments :`", or "`Local integer arrays :`", etc. on top of any group of variables declarations are not necessary: the FORTRAN attributes declarations, if used as recommended in this document, are self-documenting. Also the complete list of such templates is so wide that using them can make the code less readable.

CCPT(04) Actually unused variables (local in a used module) should be removed from the current procedure: this makes the code clearer and can reduce the dependencies complexity.

(*b*) *Kinds*

NORM(07) Variables or constants are preferably declared with explicit kind.
In practice conventional parameters have been defined for various kinds (see modules `PARKIND1` and `PARKIND2`): refer to Table F.1.

Table F.1 *Conventional kind parameters for `INTEGER`s and `REAL`s.*

`KIND` value	`KIND` parameter
`SELECTED_INT_KIND(2)`	`JPIT`
`SELECTED_INT_KIND(4)`	`JPIS`
`SELECTED_INT_KIND(9)`	`JPIM`
`SELECTED_INT_KIND(12)`	`JPIB`
`SELECTED_INT_KIND(18)`	`JPIH`
`SELECTED_REAL_KIND(2,1)`	`JPRT`
`SELECTED_REAL_KIND(4,2)`	`JPRS`
`SELECTED_REAL_KIND(6,37)`	`JPRM`
`SELECTED_REAL_KIND(13,300)`	`JPRB`
`SELECTED_REAL_KIND(28,2400)`	`JPRH`

Refer to Listing F.4 for an example of constants usage.

(*c*) *Specifications for data modules*

CTRL(05) In data modules all variables should be saved in order to preserve their values. This is achieved by the use of the statement `SAVE`.

PRES(18) Each variable should be declared separately.

Refer to Listing F.2 for an example of declarations in a data module.

(*d*) *Specifications for procedures*

NORM(08) The variables should be used or declared in the following order:

(i) the variables used from modules (this is enforced by FORTRAN standard)
(ii) the dummy arguments
(iii) the local variables

NORM(09) When using a data `MODULE`, the resources should be restricted to the actually used variables in order to avoid latent conflicts. This is achieved by the use of the keyword `ONLY`.

PRES(19) While enumerating the used variables, the items should be regularly spaced in order to respect a general alignment: this improves the readability. The space used is currently 9 characters per variable, but it could be a multiple of 9 to preserve the general alignment when long names are used. If lines should be broken the separating commas should be at the end of the lines, not the beginning of them.

PRES(20) The declaration of dummy arguments and the presentation of the dummy arguments in the subroutine interface should be the same, in order to improve the readability.

PRES(21) If lines should be broken in the `SUBROUTINE` variable lists then the separating commas should be at the end of the lines, not the beginning of them.

PRES(22) New lines in the `CALL` variable lists should be the same as "line breaks" in the subroutine arguments list.

Refer to Listing F.3 for examples of variables declarations in a procedure.

(*e*) *DOCTOR naming conventions*

The purpose of the following naming conventions is to convey, through the use of prefix letters, the type, status and scope of all variables within the program. Since the original definition of the DOCTOR system, a few minor changes have been made to reflect:

- the increase use of facilities of FORTRAN, especially `CHARACTER` type
- the rationalization in the light of experience
- the wish to restrict prefixes to a single letter as far as possible.

The use of a prefix convention to indicate the status or scope of the variable enables differentiation at a glance.

NORM(10) The type of a variable is indicated by the first — or first two — letter(s) which compose(s) its name, according to Table F.2.

Table F.2 *Naming conventions.*

Type	Variable in data module	Dummy argument	Local variable	Loop control	Any `PARAMETER`
`INTEGER`	`M,N`	`K`	`I`	`J` but not `JP`	`JP`
`REAL`	`A,B,E-H,O,Q-X`	`P` but not `PP`	`Z`	-	`PP`
`LOGICAL`	`L` but not `LD,LL,LP`	`LD`	`LL`	-	`LP`
`CHARACTER`	`C` but not `CD,CL,CP`	`CD`	`CL`	-	`CP`
Derived	`Y` but not `YD,YL,YP`	`YD`	`YL`	-	`YP`

Note:

NORM(11) Double precision variables, which are prefixed with `D` according to the DOCTOR norm, are no more used in the current software: instead of that, the type and kind of the variables are declared explicitly.

NORM(12) Elementary variables composing a derived type should follow the naming conventions for local or global variables.

(*f*) *Further naming conventions*

Names of variables should be as meaningful as possible.

At the time the DOCTOR norm was first specified, the distributed memory machines were not in operations. While programming on a distributed memory machine, the developers have to consider a new scope of variables:

- those which are *local* in the sense of the distribution: such variables can be shared by several subroutines and so they can be declared in a data module. But their values may differ between processors.
- those which are *global* in the sense of the distribution: such variables can be local to a subroutine. But they have a *physical* meaning so that their values will be the same on all processors.

Concerning this scope the naming convention is the following:

CCPT(05) Variables which are suffixed with the letter `L` are local in the sense of the distribution.

CCPT(06) Variables which are suffixed with the letter `G` are global in the sense of the distribution.

Note: the reverse assumption is not true, that is: variables which are local in the sense of the distribution are not always suffixed with `L` (actually *all* variables are local in the sense of the distribution). In the same way variables which are global in the sense of the distribution are not always suffixed with `G`. This is justified since the distribution concerns the data and not all the applications: the mentioned rule applies only to those variables related to the dimensions concerned by the distribution. For instance the model time step is not governed by this rule. Also a loop index would not be governed by this rule, while the loop bounds could be.

Table F.3 gives an example of such variables.

Table F.3 *Example of local versus global variables.*

	Variable local to a subroutine	**Variable in a data module**
Local variable in the sense of the distribution	A loop index	The number of gridpoints treated by a processor
Global variable in the sense of the distribution	A value gathered among all processors	The total number of gridpoints in the model

F.3.5 General coding norms

(*a*) *Section comments and supplementary comments*

PRES(23) The body of the code should be split into numbered sections and subsections. The numbering should be so that the M^{th} subsection of the N^{th} section would be labelled `N.M`.

PRES(24) Each *section* should be clearly separated from the previous one and should begin with its section number and an underlined title.

PRES(25) Each *subsection* should be clearly separated from the previous one and should begin with its subsection number and title.

PRES(26) Supplementary comments should be placed either immediately before or on the same line as the code they are commenting.

(*b*) *Banned features*

Several FORTRAN features should not — or no more — be used, as their past usage showed their detrimental effect in programming, or because they are becoming obsolescent and thus can disappear in future versions of the compilers.

NORM(13) `GOTO` should not be used because it is detrimental to the readability and is obsolescent. FORTRAN 90 provides instructions like `DO WHILE`, `EXIT`, `CYCLE` and the conditional block `SELECT CASE` which can replace `GOTO`.

NORM(14) FORMAT statement should not be used any more as it is becoming obsolescent. Format descriptors should be used instead. For example, one can replace:

```
WRITE(*,99) 'Hello !'
99 FORMAT(A7)
```
by
```
CLFMT='(A7)'
WRITE(*,CLFMT) 'Hello !'
```

NORM(15) COMMON should not be used. MODULE should be used instead, because it is a more robust, flexible statement.

NORM(16) EQUIVALENCE should not be used because it may cause problems of readability or sometimes portability. POINTER or TYPE data can replace it.

NORM(17) COMPLEX type should not be used since the resulting code is not efficient (Clochard, 1988).

CTRL(06) One should not implicitly change the shape of an array while passing it into a subroutine, because this works only after assumptions about how the data is stored. In such situations the code should be properly re-written. If this is not possible RESHAPE should be used instead, but this statement involves extra cost.

CTRL(07) One should not implicitly change the type of a variable while passing it into a subroutine, because this works only after assumptions about how the data is stored. In such situations one should use TRANSFER instead.

NORM(18) To declare a character string, the syntax CHARACTER*n should no more be used because it is becoming obsolescent. Hence the syntax should be:
CHARACTER(LEN=n).

NORM(19) Arrays should not be declared with implicit *size*, ie.:
`REAL(KIND=JPRB) :: A(*)`
but they may be declared with implicit *shape*, ie.:
`REAL(KIND=JPRB) :: A(:)`
Note that such declaration requires an interface block.

(c) *Loops*

NORM(20) One should use only the "block loop" construct, ie starting with DO and ending with ENDDO. Loop boundaries should stand out, finishing with ENDDO statement, in order to make future modifications inside the loop easier.

PRES(27) DO and DO WHILE loops should be indented with 2 blank spaces to improve the readability.

PRES(28) In case of complex loops nesting, it is recommended to use a character label for each loop.

CCPT(07) Loops should be as plain as possible: complexity may destroy the vectorization of the loop.

Listing F.4 shows indentations for loops.

(d) *Conditional blocks*

CTRL(08) Use the SELECT CASE statement when possible, rather than IF-ELSEIF-ELSE-ENDIF statements because the condition relies on the value of only one expression which is compared to constant values, and thus overlapping values can be detected at compilation time.

PRES(29) Conditional blocks should be indented with 2 blank spaces to improve the readability.

Listing F.4 *Example of computations in a procedure.*

```
SUBROUTINE COMPUTE
   :
   :
! 1. Initialization
!    --------------

IST   = LBOUND(ZA)
IEND  = UBOUND(ZA)
ZSCAL = 5._JPRB

ZB(:) = 2._JPRB
ZC(:) = 4._JPRB

! 2. Computation and selection
!    -------------------------

DO JI=IST,IEND
  IF (LDONE(JI)) CYCLE
  ZA(JI) = ZB(JI) + ZC(JI)
  ZD(JI) = 1.0_JPRB-LOG(ZA(JI))
  ZE(JI) = ZSCAL*ZB(JI) &
   &     + (ZA(JI)-ZD(JI))
ENDDO

SELECT CASE (NOPTION)
  CASE(1:)
    SELECT CASE (ALL(LDONE))
      CASE(.FALSE.)
        CALL ROUTINE(         &
         & NOPTION,IST,IEND, &
         & ZA,ZB,ZC,ZD,ZE)
    END SELECT
  CASE DEFAULT
    CALL ABOR1('COMPUTE : ILLEGAL VALUE NOPTION')
END SELECT

CALL MPL_BARRIER(CDSTRING='COMPUTE:')
   :
   :
END SUBROUTINE COMPUTE|\\
```

PRES(30) Nesting of conditional blocks should not be more than 3 levels deep: deeper nesting destroys the understandability of the code (Gibson, 1986). In case of complex nesting, it is recommended to use a character label for each elementary blocks.

PRES(31) Conditional block boundaries should stand out, in order to make future modifications below a condition easier. However, if the conditional instruction is nothing but a plain branch like `EXIT` or `CYCLE` then this recommendation may be ignored.

Refer to Listing F.4 for examples of usage of conditional blocks.

(*e*) *Linebreaking*

PRES(32) Though FORTRAN 90 allows up to 132 characters on a line, the length should be limited to 80 characters per line in order for the code to be viewed easily on any terminal, or to be easily read, when printed on A4 paper.

NORM(21) The continuation character "`&`" should appear both at the end of each line to be continued and at the beginning of each continuation line. In this way we have a systematic rule which allows the inclusion of blank space.

PRES(33) The continuation lines should be indented with one supplementary blank space to improve the readability.

PRES(34) Lines should be broken in a readable manner (ie: do not break a variable name). It is better to start a continuation line with an operator rather than to end one with an operator.

PRES(35) The continuation characters should be aligned on the same columns to improve the readability.

Refer to Listing F.4 for examples of linebreaking.

(*f*) *Dynamic memory usage*

CCPT(08) The use of dynamic memory (automatic or explicit allocation) is preferred to static one (arrays dimensioned with `PARAMETER` statement) because:

- it enables the re-use (and thus the saving) of memory
- it enables the same executable file to be run for different resolutions (which is basically the configuration of a multi-incremental 4D-var assimilation)

However, to avoid potential memory inefficiency, further recommendations should be considered while using dynamic memory allocation:

NORM(22) Automatic arrays should be preferred to explicitly `ALLOCATE`d arrays/`POINTER`s (except for very large arrays[2]) because they are automatically released at the end of the subroutine they are declared in.

NORM(23) Local arrays `ALLOCATE`d explicitly in a subroutine must be explicitly `DEALLOCATE`d before leaving the subroutine.

CCPT(09) One should not repeat sequences like: `ALLOCATE`, `DEALLOCATE`, `ALLOCATE` again ... many times: it is better to compute the maximum size of the array and allocate it once.

(*g*) *Symbolic comparison operators*

NORM(24) The FORTRAN 90 specific comparison operators should be used because since this syntax is closer to the mathematical notation the resulting code should be more readable. Table F.4 lists them.

Table F.4 FORTRAN 90 *specific comparison operators.*

Less than	Less equal	Equal	Not equal	Greater equal	Greater than
<	<=	==	/=	>=	>

[2]Huge automatic arrays can break the stack limit.

NORM(25) The operators == and /= should not be used to compare real variables because the result depends of the precision of the machine. This kind of comparison should be used only when absolutely necessary.
Instead of:
```
(Z1 == Z2)
```
one should write:
```
( ABS(Z1-Z2) < ZSCAL*SPACING(Z1) )
```
where ZSCAL is a scaling factor greater than 1.

(*h*) *Fortran 90 intrinsic functions and procedures*

The FORTRAN 90 language provides a large number of predefined functions or procedures. These can make the code shorter, more readable, more portable and sometimes more efficient. Table F.5 recalls several of these functions and their behaviors for zero-element arrays.

Table F.5 *Some of the predefined functions or procedures specific to* FORTRAN 90.

Fortran function	Purpose	Behaviour for zero-element arrays
ADJUSTL	To adjust a string on the left side without leading blank (not leading "space") characters	normal
ADJUSTR	To adjust a string on the right side without trailing blank (not trailing "space") characters	normal
ALL	To find out if all the values of an array are .TRUE.	.TRUE.
ANY	To find out if any value of an array is .TRUE.	.FALSE.
COUNT	To count the number of true elements in an array	0
DOT_PRODUCT	Scalar product of two vectors	0
EPSILON	Precision of the machine	normal
HUGE	largest number of the machine	normal
MAXLOC	To localize the maximum value in an array	0
MAXVAL	To find out the maximum value in an array	less equal (- HUGE)
MINLOC	To localize the minimum value in an array	0
MINVAL	To find out the minimum value in an array	HUGE
RESHAPE	To reshape an array	Possible error
SHAPE	Shape of an array	0
SIZE	Size of an array	0
SUM	To sum the content of an array	0
SYSTEM_CLOCK	To get information from the system clock	-
TINY	Smallest number of the machine	normal
TRANSFER	To transfer a variable into another type	?
TRIM	To remove the trailing blank (not the trailing "space") characters of a string	Error

NORM(26) *generic* names should be used for intrinsic procedures, not specific names.

(*i*) *Fortran 90 array syntax*

FORTRAN 90 array syntax makes the code more compact and sometimes more readable, but in most cases the result is slower, or at least not faster than the FORTRAN 77 style DO loops.

The reason is the compiler's inability to fuse several array statements and re-use common sub-expressions, registers, etc. With the current level of maturity of FORTRAN 90 compilers there is no reason to believe that the situation will improve dramatically in the future.
Therefore:

CCPT(10) The use of array syntax is not recommended, except for simple operations, like initializing or copying whole arrays.

Refer to Listing F.4 for examples of recommended computations for arrays. Note that in the F90 style, `ZA(:)` has a precise meaning: it means that we consider the whole array. The lower and upper bound are then respectively `LBOUND(ZA)` and `UBOUND(ZA)`.

(*j*) *Dummy and actual arguments*

In FORTRAN 90 there are two ways of associating arguments when a subroutine is called, the FORTRAN 90 way and for compatibility the FORTRAN 77 way (Adams *et al.*, 1992). The main difference lies in the way arrays are passed, in the FORTRAN 90 way it is by strict type, kind, rank and extent matching whereas in the FORTRAN 77 way it is done by Array Element Sequence association. It is important to know that the FORTRAN 90 way is only used when you have an explicit interface block and the arrays are declared with assumed shape.

The FORTRAN 90 way of passing arguments is much more secure, the compiler will detect any mismatch between actual and dummy arguments thanks to the explicit interface block. Unfortunately the use of explicit interface blocks/module procedures is still very limited within the ARPEGE/IFS/ALADIN code, one reason being that it introduces more dependencies between separately compiled units and thus increases the complexity and possibly cost of the compiling system.

CTRL(09) When an explicit interface block is being used for a routine, the interface body should be in an independent separate file (ie: starting with "`INTERFACE`" and ending with "`END INTERFACE`" and introduced in the calling routine with an `#include` statement. The interface body should be extracted from the routine itself by an automatic procedure to ensure that they conform.

NORM(27) The `INTENT` attribute should be used for all dummy arguments: this improves the auto-documentation and the security of the code.

CTRL(10) The number of dummy arguments should be kept as small as possible. As excessive number of arguments degrades the readability and increases the problems of maintenance whenever arguments are added or removed.
To retain the modularity of subroutines there are alternatives:

- to re-design a set of elementary arguments as a new derived `TYPE`;
- to identify the arguments which are internal to a set of subroutines and to use them via a data module. Care has to be taken that this does not cause problems with the thread safeness of the code.

The standard should be: the number of dummy arguments should not exceed 9.

CTRL(11) The arguments of a subroutine should be presented following a conventional order because this improves the readability, the maintainability and sometimes also the portability (the dimensioning of dummy arguments should appear ahead in order to improve the portability). For the time being such a rule has been applied only for the physical package of MÉTÉO-FRANCE, with the convention: input, then input/output, then output arguments. Other orders can be considered, for instance: to order the arguments according to their types and attributes, including the `INTENT` attribute. Concerning the tangent linear and adjoint subroutines the initial recommendation was to follow the order of arguments as in the direct code, then to add the trajectory variables in the same order. Such rule can conflict with other general norms.
Finally the achievable norm in this context seems to be: the arguments of a subroutine should be ordered *at least* with `INTEGER` scalar first.

NORM(28) The preferred method for passing array subsections is to use an explicit interface block. This method allows array sections to be passed safely with no extra cost. If instead an array section is passed when using the FORTRAN 77 way of passing arguments extra copying will take place before and after the subroutine call (the compiler will generate the code) incurring extra cost. If the

bounds of the array section is specified wrongly this copying may also cause memory overwriting. Thus when using the FORTRAN 77way of passing arguments, when the section of the array you want to pass is contiguous in memory and there is no explicit interface block declared, one should pass the start address of the section e.g. `ZARG(1,2)` rather than the array section - `ZARG(:,2)`. Passing array sections, not contiguous in memory, should be completely avoided when using the FORTRAN 77 way of passing arguments.
However the use of an explicit interface block with assumed shape arrays can cause memory overwriting too, because the compilers become unable to check bounds. Thus when using assumed shape arrays one should take care that the array subscript never goes out of the interval given by the function `LBOUND` and `UBOUND`.

NORM(29) Use of array sections (using FORTRAN 90 array syntax) is encouraged when calling intrinsic routines, as all intrinsic FORTRAN 90 subroutines have explicit interfaces. This makes the code more readable and enables the compiler to check bounds properly.

Listing F.5 illustrates the handling of dummy arguments.

Listing F.5 *Example of dummy arguments handling.*

```
SUBROUTINE PROCEDURE(KLEN,PIN1,PIN2,PIN3,PIN4,PIN5, &
 &                   PIN6,PIN7,POUT1,POUT2)
  :
  :
USE PARKIND1 , ONLY : JPIM     ,JPRB

IMPLICIT NONE

INTEGER(KIND=JPIM), INTENT(IN)  :: KLEN
REAL(KIND=JPRB),    INTENT(IN)  :: PIN1(KLEN), &
 &                                 PIN2(KLEN), &
 &                                 PIN3(KLEN)
REAL(KIND=JPRB),    INTENT(IN)  :: PIN4(KLEN), &
 &                                 PIN5(KLEN), &
 &                                 PIN6(KLEN)
REAL(KIND=JPRB),    INTENT(IN)  :: PIN7(KLEN)
REAL(KIND=JPRB),    INTENT(OUT) :: POUT1(KLEN), &
 &                                 POUT2(KLEN)

INTEGER(KIND=JPIM), PARAMETER :: JPNAME=9
INTEGER(KIND=JPIM), PARAMETER :: JPMESS=30
CHARACTER(LEN=JPNAME), PARAMETER :: CLNAME='PROCEDURE'
CHARACTER(LEN=JPMESS) :: CLMESS= &
 &                   ' : INVALID NUMBER OF ARGUMENTS'
!-----------------------------------------------------

! 1. Computation
!    -----------
  :
  :
END SUBROUTINE PROCEDURE
```

F.3.6 Specific coding norms

(*a*) *Naming modules, procedures, namelists and derived types*

Names of modules, procedures, namelists or types should be as meaningful as possible.

CTRL(12) Conventional prefixes or suffixes are recommended for names. Refer to Table F.6.

Table F.6 *Conventional prefixes and suffixes.*

Prefix	Entities	Suffix
TYPE_	Types names	
TYPE_	Types definitions modules	S
PAR	Parameters modules	
YOE	Data modules specific to ECMWF physics	
QA	Data modules specific to CANARI	
YEM	Data modules specific to ALADIN	
TPM_	Data modules specific to spectral transforms packages	
MPL_	Data modules specific to MPL (message passing) package	
YOM	Data modules not specific to ECMWF physics, CANARI, ALADIN, spectral transforms or MPL package	
	Procedure modules	_MOD
NAE	Namelists specific to ECMWF physics	
NEM	Namelists specific to ALADIN	
NAC	Namelists specific to CANARI	
NAM	Namelists not specific to ECMWF physics, ALADIN or CANARI	
SUEC	Setup procedures specific to ECMWF physics	
SUE, not SUEC	Setup procedures specific to ALADIN	
SU, not SUE	Setup procedures not specific to ECMWF physics or ALADIN	
SL	Calculation procedures for any horizontal interpolations system	
LA	Calculation procedures specific to the semi-Lagrangian scheme	
AC	Calculation procedures specific to ARPEGE/ALADIN physics ("Arpege Calcul")	
PP	Calculation operators for the post-processing or the analysis	
FP	Procedures specific to FULLPOS	
CA	Procedures specific to CANARI	
FA	Procedures specific to the Files Arpege package (FA)	
LFI	Procedures specific to the Indexed Files Library (LFI)	
MPL_	Procedures specific to the Message Passing Library (MPL)	
SI	Procedures specific to the semi-implicit scheme	
GNH	Procedures specific to non-hydrostatic gridpoint calculations	
CP or GP	Non-specific gridpoint calculation procedures	
SP	Spectral calculation procedures	
COMM, GATH, ISND, IRCV, OSND, ORCV, BR, DI or TR	Procedures dealing with inter-nodes communications ("COMMunicate", "GATHer", "Input SeND", "Input ReCeiVe", "Output SeND", "Output ReCeiVe", "BRoadcast", "DIstribute", "TRanspose")	
RE or RD	Procedures to read data	
WR	Procedures to write data	
E	Procedures specific to ALADIN ("Elliptic")	
	Tangent linear of a procedure	TL
	Adjoint of a procedure	AD
	Inverse of a procedure	IN

Note: additive standards concern the radical of names:

CTRL(13) The radical of a TYPE definition module should be the name of the type it defines[3]. For instance the type TYPE_GFLD is defined in the module TYPE_GFLDS.

CTRL(14) The radical of a procedure module should be the name of the procedure it encapsulates. For instance the module SUPOL_MOD encapsulates the procedure SUPOL.

CTRL(15) For a subroutine in the spherical geometry of ARPEGE/IFS, its counterpart subroutine in the toroidal geometry of ALADIN should have the same name prefixed with an "E".

CTRL(16) Considering a NAMELIST, its content should be saved in a specific data module and initialized in a specific subroutine. All three should be named with the same radical. For example: the content of the namelist NAMCTO is saved in the module YOMCTO and initialized in a subroutine SUCTO.

(*b*) *Error handling*

Proper management of the errors during the execution of the program help finding them more quickly.

CTRL(17) On error detection, a brief message describing the error should be written out to the conventional error file and output file with logical unit numbers are respectively NULERR and NULOUT. On one hand it is important to write out the error message on NULERR otherwise if only processors other than 1 abort we have no information about the abort, unless we ask for all the output files (one per task). But in this case the number of files can be so large that the debugging would not be easier.
On the other hand, writing twice the error message (on NULERR and NULOUT) can confuse the user, and there can even be a huge number of identical error messages in the listing if all processors abort for the same reason.

CTRL(18) Then, if abnormal termination is required, the subroutine ABOR1 should be called with a message (a character string) as argument, indicating the error location. The use of the subroutine ABOR1 gives time to flush the output buffer and to release the processors not causing ABOR1. It writes out a message on NULERR, and possibly on NULOUT if an argument is provided.

CTRL(19) Sometimes it can be advantageous to postpone the abnormal termination until the end of the subroutine in order to output all the errors detected to the output file before actually aborting.

NORM(30) The statement STOP should not be used in case of an error because it reports a normal termination code.

Refer to Listing F.1 for examples of error handling.

(*c*) *"Hook" function*

CTRL(20) Each subroutine should start and end with a conditional call to a "hook" subroutine. One main usage for it may be finding really awful bugs where we do not get any traceback because the stack has been trashed, but there are also many other potential uses, for statistics gathering, doing 'checksumming' for early catching of problems, etc.

Listing F.1 shows an example of "hook" function.

(*d*) *Handling universal constants*

CTRL(21) Universal constants are stored in a data module named YOMCST. To access them one should use this module.

CTRL(22) Universal constants should not be redefined at any other place in the code, to avoid any potential inconsistency after a redefinition.

[3] ambiguous if there are more than one type defined in the module.

CTRL(23) Universal constants should not be accessed via dummy arguments because there would be a risk to overwrite them through the subroutine interface.

CTRL(24) To make it more robust all universal constants should be declared and initialized in a unique module (fusion of the module YOMCST and the subroutine SUCST of today).

(*e*) *Purpose and usage of the key LECMWF*

In order to simplify user namelist files a different default setup is performed according to the value of the logical key LECMWF. If LECMWF is .TRUE. then the selected default setup corresponds to the framework of ECMWF; else it corresponds to the framework of MÉTÉO-FRANCE.

CCPT(11) The key LECMWF should appear only in the setup routines and should be used only to initialize namelists variables in order to preserve the scientific flexibility of the code.

(*f*) *Purpose and usage of the key LELAM*

The logical key LELAM enables the selection of the limited area model (ALADIN) instead of the global model (ARPEGE/IFS). Thus this key controls branches of the code related to the limited-area versus global aspects of the model.

CCPT(12) The key LELAM should be used only in the setup and control subroutines (ie: not below SCAN2MDM) in order to minimise the scientific generality of the code.

CCPT(13) The code below the key LELAM should be modular as far as possible in order to preserve the visibility of the ALADIN specific code from those who are not ALADIN partners.

CCPT(14) Use of LELAM should be as rare as possible. If a routine uses lots of LELAM keys then it should have its own ALADIN counterpart subroutine called under a single LELAM key.

F.3.7 Purpose and usage of the key LRPLANE

The logical key LRPLANE selects the plane geometry instead of the spherical one. Therefore this key has a strong relationship with the key LELAM.

CCPT(15) Contrary to the key LELAM, the key LRPLANE can be used at any place in the code, but to preserve the scientific generality of the code it should not replace the key LELAM. It can be used outside a LELAM section to treat in a general way low-level parts of the code (for example: in the semi-Lagrangian scheme).

Note that LELAM=.TRUE. together with LRPLANE=.FALSE. would indicate that ALADIN is run in spherical latitudes-longitudes geometry instead of the usual projected plane. This facility has been abandoned in practice for quite a few years but should remain possible in principle.

(*a*) *Model settings*

CTRL(25) User variables for setting up the model should be accessed via a conventional formatted sequential file containing namelists. Its logical unit number is: NULNAM (NULNAM=4).

CCPT(16) Namelist variables should be read from the namelist file and initialized only at one place in the software, in order to prevent redefinition of variables.

CCPT(17) To enable an easy control of the variables used in the program, all the namelist variables should be printed out to the listing file and not be redefined later in the code.

(*b*) *Output messages*

CTRL(26) Messages should be written to the conventional formatted sequential file with logical unit number: NULOUT. The standard output ("*" or unit 6) should not be used as it would mix the messages coming from different processors.

CCPT(18) Important messages may be written out to the standard error file which logical number in the software is NULERR. In that case messages coming from the different processors will be mixed.

CCPT(19) Verbosity should be controlled by the specific namelist variable NPRINTLEV, running between 0 (minimum prints and default value) to 2 (maximum prints).

F.3.8 I/O raw data

CCPT(20) Observation files are binary files to be handled with the ODB software.

CCPT(21) Restart files are binary files to be handled with the PBIO software, which uses C I/O and gives pure binary files without any Fortran record structure.

CCPT(22) Movies (MÉTÉO-FRANCE only) are FORTRAN sequential binary files.

CCPT(23) In the ECMWF framework other user's I/O raw data should be accessed via GRIB files, using the PBIO software.

CCPT(24) In the ARPEGE/ALADIN framework other user's I/O raw data should be accessed either via FA files if the horizontal format of the data corresponds to the model settings; else via LFI files. These are unformatted indexed sequential files.

CCPT(25) It is recommended to use the logical key LARPEGEF rather than the key LECMWF to select the files format FA/LFI versus GRIB.

More generally, the recommendation is to use C I/O to improve the portability (today almost all computers adhere to the IEEE standard).

F.3.9 Message passing interface

CCPT(26) One should use the MPL package as interface for any message passing.

CTRL(27) For an easier control of the code, each MPL subroutine call should have its argument CDSTRING explicitly documented as the name of caller routine. Listing F.4 shows an example of this.

F.4 SOURCE CODE MANAGEMENT

The source code is stored in a database managed by the PERFORCE software package.

Among other advantages, the use of this software makes it possible to maintain an accurate view of the history of the code, and to simplify and make code merging operations safer. Thus, **it is of vital importance to use Perforce to modify the code.**

A few standards should be considered while handling the source code files:

CTRL(28) The whole source code is partitioned into *projects*. Below each project the source code is partitioned into directories. Each directory contains elementary files which are either compilable source files or pieces of source files ("include files") to be included in other source files.

CTRL(29) Each elementary file should contain only one `MODULE` or only one procedure: this makes the maintenance easier (but a procedure may include more than one subroutine via the instruction `CONTAINS`).

CTRL(30) All procedures which are internal to a package should be encapsulated inside a `MODULE`: through the recompilation of the dependencies this enables the compiler to check automatically the interfaces for all the depending procedures. This has already been done for the spectral transform package.

CCPT(27) Each elementary file should be put in the proper project and below the directory which best fits its topic. For example: dynamics routines should be put in the ARPEGE/IFS project directory `adiab`.

CCPT(28) `NAMELIST` statements should be declared in a module containing the namelist variables (data part) as well as the subroutine initializing these variables (via the `CONTAINS` statement): this would make the maintenance and developments easier.

CTRL(31) The basename of each compilable source file should be the name (in lowercase letters) of the `MODULE` or `SUBROUTINE` it contains. For example: the file `suct0.F90` contains the subroutine `SUCT0`.

CCPT(29) Derived `TYPE`s should be declared in a `MODULE` because this manner is more robust than using the attribute `SEQUENCE` and it makes the maintenance easier (no duplication of code). There should be one module dedicated to the declaration of each derived type (or group of derived types if they are closely related), and vice-versa. These modules could also contain "primitive" operations on the type(s) like allocation or deallocation of its components, etc. The *structures* defined by this or these type(s) should not be in this module, only type(s) definitions and basic operations on the type(s) should be.

CTRL(32) Each `NAMELIST` should be contained in a specific include file, which basename should be the name of the namelist (in lowercase letters). For example: the file `namct0.h` contains the namelist `NAMCT0`.

CTRL(33) Each explicit interface should be contained in a specific include file, with basename the name of the subroutine it contains. For example: the file `suspec.h` contains the interface block of the subroutine `SUSPEC`. Note: an explicit interface is necessary whenever a `POINTER` variable is used as a dummy argument. Interfaces should be computer-generated.

CTRL(34) Useless files should be deleted.

F.5 INDEX OF STANDARDS FOR THE PRESENTATION OF THE CODE

F.6 INDEX OF STANDARDS FOR THE RESPECT OF THE NORM

F.7 INDEX OF STANDARDS FOR THE CONTROL OF THE CODE

F.8 INDEX OF STANDARDS FOR THE CONCEPTION OF THE CODE

Appendix G

The Perforce source code management system user guide

Table of contents

G.1 BACKGROUND

G.1.1 Introduction

The main Source Code Management (SCM) system used by RD was changed from CLEARCASE to PERFORCE at CY29R1. The main reason for the change was the difficulties caused by not having access to CLEARCASE on all platforms, specifically on the local desktop (CLEARCASE was available only on the IBM servers). The change to PERFORCE leads to more natural working practices where the editing and preparation of branches is done on the desktop and for the build (under PREPIFS) the source code is directly available on the HPC.

G.1.2 Perforce

PERFORCE is a commercial product chosen for its availability on a wide range of platforms. PERFORCE is a pure client/server SCM system. The PERFORCE command line client (p4) talks to a server (p4d) where the central file repository, or "depot" is situated. You can also access PERFORCE through a visual client (p4v). For further information see the Perforce web-site:
http://www.perforce.com/index.html.

G.1.3 P4 within ECMWF's Research Department

The aim within the Research Department (RD) for the move from CLEARCASE to PERFORCE was to keep as close to established working practices as possible. To facilitate this a new script (q2) replaces the established set of scripts that RD users used to access CLEARCASE (selbranch, findfiles etc.).

To get a description of the q2 commands, type:

```
q2 help all
```

You will see that most q2 commands have aliases that are the same as the old CLEARCASE commands even though the "real" names were changed where they no longer made sense.

G.2 GETTING STARTED

Before getting started it is important to understand one important difference between ClearCase and Perforce. Whereas in ClearCase you had "views", in Perforce you have "clients" and you always do your editing etc. on local copies of the files extracted (synced) from the Perforce depot. The place where your local files will reside is referred to as your client workspace. It is strongly recommended that you always use your local desktop for the interactive access to Perforce.

What follows is some simple scenarios of what you may want to try to get started. Note that p4 commands are native Perforce client commands whereas q2 commands are ecmwf's customized commands.

G.2.1 Create a branch, edit a file, add another file and then submit

(i) To create a branch, a client and change your current working directory to your client's "root" (see above):
`q2 create_branch -r CY29R1 -b test1 -p ifs`
(equivalent of ClearCase `selbranch`*)*
(ii) `cd` to the directory where you want to edit a file
(iii) Open the file for edit (like checking out in ClearCase):
`p4 open <Filename>.F90`
(iv) Edit `<Filename>` using editor of your choice.
(v) Create a new file `<NewFilename>.F90` locally (by copying or editing).
(vi) To tell Perforceyou want to add this file:
`p4 add <NewFileName>.F90`
(equivalent of ClearCase `addfile`*)*
(vii) To update the depot on the server (similar to checking in with ClearCase, but here you do all the changes in one go rather than individual files):
`q2 submit`
or
`p4 submit`

Until you issue the `submit` your edits are purely local and can be lost if you lose your local disk. Also the changed files can not be seen by the build environment (PrepIFS) until they have been submitted. If you use "`p4 submit`" your editor will open on a form where you have to fill in a description of your change. It is not recommended that you submit after every edit of a file but only after you have edited all the files you intend to change or as a precaution before going home.

G.2.2 Create a branch, edit a file and then submit using p4v

(i) `q2 create_branch -r CY29R1 -b test1 -p ifs -x`
(equivalent of ClearCase `selbranch -x`*)*
(ii) Use `p4v` to do your open for edit (right-click on file).
(iii) To add files using `p4v` the file has to exist so `touch` it in line mode or save it with new name using your editor.
(iv) To submit in `p4v`, click red triangle and then default. You will have to write something on the comment line otherwise the submit will fail.

The first time you use `p4v` you should go into "tools `->` preferences" where you can change you preferred editor, fonts etc. The editor can be different for each file type so the first time you try to open a file with a new suffix you will be prompted for which editor (or other application) to use. If you are a `vi` user specifying `vi` will not work as `vi` does not open a window by default. In this case change the editor in `p4v` to `/home/rd/rdx/bin/xvi` which will start an `xterm` with `vi`.

G.2.3 Migrate branch from Clearcase to Perforce

(i) Select a ClearCase View on a ClearCase host (leda,metis,ecgate ...) using `selview`.
(ii) Migrate the branch to Perforce by typing:
`q2 cc_to_p4_branch`

(iii) After the migrate has completed use `q2 select_client` in a local window to access the branch you just migrated.

`q2 select_client` is the approximate equivalent of `selview`. With PERFORCE you have "clients" not views, the main difference is that you always work with local copies of the files in the depot.

G.2.4 Selected p4 and q2 commands

Below find some selected `p4` and `q2` commands. Recall that `p4` is the PERFORCE command client whereas `q2` is an ECMWF script created to simplify the use of `p4`. A `q2` command normally issues one or more `p4` commands. For more info type:
`p4 help` or `q2 help`.

If you use `p4v` you don't need to use any of the `p4` commands, they are all accessible from the `p4v` interface. The `q2` commands are not available inside `p4v`, they have to be issued from the command line.

`p4 edit`
: Open existing file for edit

`p4 add`
: Open new file for adding to depot

`p4 delete`
: Open existing file for delete

`p4 opened`
: List currently open files

`p4 revert`
: Revert changes to open files

`p4 submit`
: Submit your changes

`q2 create_branch`
: Create branch and client

`q2 add_projects`
: Add additional projects to branch and client

`q2 select_client`
: Select which client/branch to use

`q2 cc_to_p4_branch`
: Migrate branch from CLEARCASE to PERFORCE

`q2 merge_branch`
: Merge branch into current client

`q2 find_files`
: Find files modified on branch

`q2 p4v`
: Start `p4v` after `select_client` (if you forgot `-x`)

`q2 submit`
: Wrapper for `p4 submit`

`q2 rm_private`
: Remote private files

`p4login`

Get/update `p4` certificate. If you use `q2` commands `p4login` is automatically issued but you may need to execute `p4login` if you get a message like "`ticket expired`". Note "`p4login`" **not** "`p4 login`".

The `p4login` script is written in house to ensure perforce security without users having to actually type in passwords.

G.3 q2 COMMANDS

What follows is the help text for all q2 commands.

G.3.1 q2 add_projects

Add project(s) to current branch and client specs

Usage:

```
q2 add_projects  [-g pgroup]
                 [-p project1[:project2:...]]
                 [-r release]
```

Arguments:

`-g pgroup`
: Project group (release stream)
[Default: ifs]

`-p project`
: Project(s) to add

`-r release`
: Release to branch from
[Default: release used for existing projects]

Use this script if you already have a branch but need to modify more projects. If you do not use the `-p` option you will be prompted to select project(s).

G.3.2 q2 addprojects

`q2 addprojects` is an alias for `q2 add_projects`

G.3.3 q2 addprojs

`q2 addprojs` is an alias for `q2 add_projects`

G.3.4 q2 add_to_client

Add project(s) to current client without branching

Usage:

```
q2 add_to_client   [-g pgroup]
                   [-p project1[:project2:...]]
                   [-r release]
```

Arguments:

`-g pgroup`
: Project group (release stream)
[Default: ifs]

`-p project`
: Project(s) to add

`-r release`
: Release

Use this script to add projects to your client for read-only access. You will not be able to modify these projects as you have not created a branch for them.

G.3.5 q2 add_vpath

Add the VPATH directories to your client spec and sync

Usage:

```
q2 add_vpath
```

Arguments:

None

Add all directories found in the VPATH for your projects to your client specification and sync. This is needed in order to be able to e.g. locally compile your modified routines using gmake.

G.3.6 q2 addvpath

q2 addvpath is an alias for q2 add_vpath

G.3.7 q2 cc_to_p4_branch

Migrate a branch from ClearCase *to* Perforce

Usage:

```
q2 cc_to_p4_branch  [-b branch_tag]
```

Arguments:

`-b branch_tag`
: Perforce branch tag
[Default: ClearCase *branch tag]*

You have to be in a ClearCase view (on an IBM server) to run this script.
A Perforce branch and client will be created.
The client will temporarily create a client workspace on `$SCRATCH` but at the end this will be removed and the client root be reset to its normal location (on your desktop).

G.3.8 q2 check_norms

Check conformance with coding norms

Usage:

```
q2 check_norms  [-p project]
                [-m suppress]
                [-w] [-W] [-i] [-I]
```

Arguments:

`-p project`
: Check project <Project>
[Default: ifs]

`-m suppress`
: Messages to be suppressed
[Default: PRES-32]

`-w`
: Print warning messages
[Default: on]

`-W`
: Suppress warning messages

`-i`
: Print information messages
[Default: on]

`-W`
: Suppress information messages

You have to be in a client view to run `q2 check_norms`.

G.3.9 q2 client_info

Print out info from client spec

Usage:

```
q2 client_info  [-c client]
                [-p] [-R] [-m] [-o] [-b]
```

Arguments:

`-c client`
: PERFORCE client name
[Default: $P4CLIENT]

`-R`
: Print root

`-p`
: Print projects

`-o`
: Print owner

`-m`
: Print mapping

`-b`
: Print branch

G.3.10 q2 create_branch

Create branch and client

Usage:

```
q2 create_branch  -b branchtag
                  [-r release]
                  [-p project1[:project2:...]]
                  [-g pgroup]
                  [-B parent]
                  [-R] [-D] [-x]
```

Arguments:

`-b branchtag`
: User definable part of branch name

`-r release`
: Release from which to branch

`-p project1[:project2:...]`
: Create branch for listed project(s)

`-g pgroup`
: Project group (release stream)
[Default: ifs]

`-B parent`
: Parent branch for creation of secondary branch

`-R`
: Do not include release in branch name

`-D`
: Do not prompt for branch descriptor

`-x`
: Start p4v after setting client

Creates a branch specification and a client specification.
The client name will be `client_branch` name, the branch name will be `userid_release_branchtag` unless `-R` is specified where instead the branch name will be `userid_branchtag`.
After running `p4 integrate` to setup the branch and submitting, sets the view and optionally "`-x`" forks p4v.

G.3.11 q2 createbranch

`q2 createbranch` is an alias for `q2 create_branch`

G.3.12 q2 selbranch

`q2 selbranch` is an alias for `q2 create_branch`

G.3.13 q2 diff_branch

Diff branch from release it come from

Usage:

```
q2 diff_branch  [-b branchname]
                [-p project]
                [-f file]
                [-d] [-x]
```

Arguments:

`-b branchname`
: PERFORCE branch identifier
[Default: current branch]

`-p project`
: Restrict to project <project>

`-f file`
: Diff only this file

`-g pgroup`
: Diff only this file

`-d`
: Options for `diff` (ignore leading blanks, ignore whitespace etc).

`-x`
: Use `xdiff` instead of line difference

G.3.14 q2 diffbranch

`q2 diffbranch` is an alias for `q2 diff_branch`

G.3.15 q2 find_files

Find files that are modified on branch

Usage:

```
q2 find_files  [-b branchname]
               [-p project]
               [-n] [-O] [-d]
```

Arguments:

`-b branchname`
: PERFORCE branch identifier
[Default: current branch]

`-p project`
: Restrict to project <`project`>

`-n`
: Non-extended path names only

`-O`
: Suppress listing open files

`-d`
: Find also deleted files

If used for current branch `q2 find_files` will also list ALL open files (unless `-O`)

G.3.16 q2 findfiles

`q2 findfiles` is an alias for `q2 find_files`

G.3.17 q2 help

General help information for q1

Usage:

```
q2 help  command_name | all
```

Arguments:

`command_name`
: Help on specific q2 `command`

`all`
: Help on all q2 commands

G.3.18 q2 import_from_tar

Import changes from tar-file into project

Usage:

```
q2 import_from_tar  -p project
                    -t tarfile
                    [-d]
```

Arguments:

`-p project`
: Import into project <project>

`-t tarfile`
: Path name of tarfile

`-d`
: Delete all files from project not found on tar file

The tar file MUST be created inside the project i.e. so that what you get when you untar it are the subdirectories of the project. You must already have `$P4CLIENT` set and the client spec set to include the project you want to import. If you want files that are in your view but not in the tar file to be deleted set the `-d` flag.

G.3.19 q2 list_branches

List branches containing certain patterns

Usage:

```
q2 list_branches  [-u user]
                  [-r release]
                  [-s substring]
                  [-U]
```

Arguments:

`-u user`
: Branches for specific user id
[Default: nar]

`-r release`
: Branches for specific release

`-s substring`
: String anywhere in branch tag

`-U`
: Any user (overrides default for user)

Simple filter for output of `p4` branches.

G.3.20 q2 branches

`q2 branches` is an alias for `q2 list_branches`

G.3.21 q2 list_changes

List changes

Usage:

```
q2 list_changes   [-u user]
                  [-c]
                  [-s substring]
                  [-U]
```

Arguments:

`-u user`
: Changes for specific user id
[Default: nar]

`-c`
: Changes only for current client (`$P4CLIENT`

`-s substring`
: String anywhere in change descriptor

`-U`
: Any user (overrides default for user)

Lists submitted and pending changes (using `p4 changes`). Optionally filters for user, current client and arbitrary string in change descriptor. To obtain further information about specific change use `p4 describe changelist <#>` where `<#>` is the changelist number or use `p4v`.

G.3.22 q2 local_changes

Finds differences between your local workspace and the depot

Usage:

```
q2 list_changes  [-F]
```

Arguments:

`-F`
: Disable filtering out uninteresting files e.g. `.o` etc.

You need to be in a view with `$P4CLIENT` set to run this script. Unless `-F`, `q2 local_changes` filters out files with suffixes "`.o`", "`.list`" or "`.lst`". Also filters out files names ending with "`#`" or "`~`". If run with open changelist (files yet to be submitted) the output of this script becomes confusing. It is suggested that you first submit your changes.

G.3.23 q2 ls_private

List private files in your current client workspace

Usage:

```
q2 ls_private
```

Arguments:

None

You have to have $P4CLIENT set to run this command.

G.3.24 q2 lsprivate

q2 lsprivate is an alias for q2 ls_private

G.3.25 q2 merge_branch

Merge another branch into current branch

Usage:

```
q2 merge_branch  -b branchname
                 [-p project]
                 [-f file]
                 [-r] [-n]
```

Arguments:

`-b branchname`
: PERFORCE branch identifier

`-p project`
: Restrict to project <`project`>

`-f file`
: Merge file <`file`>only

`-r`
: Resolve

`-n`
: Automatic merge

By default `q2 merge_branch` will be manual. It is recommended to use `p4v` to resolve and merge. In PERFORCE merging is a three stage process: integrate, resolve and optionally merge. In manual mode this script will only do the integrate part and then do a submit which will fail. You then need to use either `p4v` (recommended) or `p4 resolve` in order to be able to submit.
In automatic mode (`-n`) this script will also do the resolve (in auto-resolve mode). If there are conflicts again the submit will fail and you will need to resolve manually using one of the methods outlined above.

G.3.26 q2 mergebranch

`q2 mergebranch` is an alias for `q2 merge_branch`

G.3.27 q2 p4v

Start p4v in the background with current client,user and port

Usage:

```
q2 p4v
```

Arguments:

None

G.3.28 q2 recreate_client

Re-create deleted client for existing branch

Usage:

```
q2 recreate_client  [-b branchname]
```

Arguments:

`-b branchname`
PERFORCE branch identifier

This script is for re-creating client for an existing branch. Normally to be used when client has been deleted but the branch spec still exists.

G.3.29 q2 reintegrate

Integrate with current branch spec

Usage:

```
q2 reintegrate  [-n] [-r]
```

Arguments:

`-n`
: Automatic resolve

`-r`
: Resolve

The intention with this script is to pick up back-stitches in release or for use in a secondary branch where the primary branch has changed. Avoid running this with open files as you will get a mixed changelist.

G.3.30 q2 rm_private

Remove private files in your current client workspace

Usage:

```
q2 rm_private  [-f]
```

Arguments:

`-f`
: Do not ask for confirmation before removing file

You have to have `$P4CLIENT` set to run this command.

G.3.31 q2 rmprivate

`q2 rmprivate` is an alias for `q2 rm_private`

G.3.32 q2 select_client

Select client and set view

Usage:

```
q2 select_client  [-c client]
                  [-u user]
                  [-x]
```

Arguments:

`-c client`
: Client name (unless provided you will be prompted)

`-u user`
: Select other user's client

`-x`
: Start p4v after setting up view

Normal usage is
`q2 select_client [-x]`
This will offer you a choice of your existing clients and set the view according to your choice.
Setting the view involves setting $P4CLIENT, syncing and changing directory to the client's working directory. The script will then start a shell with these settings. With the "`-x`" option it will also start p4v with the appropriate settings. With the "`-u`" option you can create a temporary client to view the branch of another user. Please note that you will not see changes that this user has yet to submit.

G.3.33 q2 selclient

q2 selclient is an alias for q2 select_client

G.3.34 q2 submit

Submit changes

Usage:

```
q2 submit  [-d description]
           [-n]
```

Arguments:

`-d description`
: Line describing your edits

`-n`
: No description

G.3.35 q2 unsync_client

Remove local workspace for client

Usage:

```
q2 unsync_client   [-c client]
                   [-r] [-d] [-f]
```

Arguments:

`-c client`
: Client name
[Default: $P4CLIENT]

`-r`
: Also remove local files in workspace tree

`-d`
: Delete client specification

`-f`
: Do not ask for confirmation of actions

If you use the "`-d`" flag the client specification will be deleted. This does not mean that submitted changes you have on your branch will be lost, the branch specification is still there and the client can be recreated using `q2 recreate_client`.

G.3.36 q2 remove_client

`q2 remove_client` is an alias for `q2 unsync_client`

G.4 LOCAL COMPILE OF PERFORCE BRANCH ON LINUX WORKSTATION

There are two scripts available in ~`rdx/bin` which are described here, which can be used for local compilation and syntax checking.

Note: You must type:
`q2 select_client`
before compiling using either of the scripts presented here.

G.4.1 compile.p

Compiles all files in a branch.
The script emulates a single task in the PrepIFS *compile suite*

Usage:

```
compile.p  [-t task]
           [-p project]
           [-d debug_level]
           [-c changelist]
           [...dir(s)...]
```

Arguments:

`-t task`
: Compile code for a particular PrepIFS task (eg. `odbsqlcompiler`) if different from the specified project.

`-p project`
: Compile code for a particular IFS project (eg. `ifs`,`trans`,`surf` etc.)
[Default: ifs]

`-d debug_level`
: If specified, switches on the debug symbol information ("`-g`" in most compilers). `debug_level` can take the following values:

0 : Debug with optimisation

1 : Debug with limited optimisation

2 : Debug with no optimisation

Note: Not all compilers support all of these options.

`-c changelist`
: Synchronise the client and compile a previous version of the branch as identified by the `changelist` number.

`...dir(s)...`
: If specified, compiles only the selected directories of the chosen `project`.

The user must take care of the order of compilation if modifications exist across several projects, eg. compile `ifsaux` before `ifs` or `surf` before `ifs`.

Examples:

`compile.p`
: Compiles all modifications in the default project (`ifs`) and puts new `libifs.a` in `/var/tmp/tmpdir/$USER/p4w/$BRANCH/libs`.

`compile.p -p odb -t odbsqlcompiler`
: Compiles and generates the executable `odb98.x`.

`compile.p -p surf module`
: Compiles all modifications in the `module` directory of the project `surf`.

G.4.2 compile.i

Compiles one file or one directory in a branch.
The script compiles individual files or directories or more generally to execute any available `make target` *from the* `Makefile` *in the* `scripts/build` *directory.*

Usage:

```
compile.i  [-t task]
           [-p project]
           [-d ]
           [...target(s)...]
```

Arguments:

`-t task`
: Compile code for a particular PrepIFS task (eg. `odbsqlcompiler`) if different from the specified project.
[Default: Environment variable `$TASK`*]*

`-p project`
: Compile code for a particular IFS project (eg. `ifs`,`trans`,`surf` etc.)
[Default: Environment variable `$PROJECT` *or ifs]*

`-d`
: Switches on the `make` debug output to identify potential problems with `make`, eg. dependencies, missing files, make rules etc.

`...target(s)...`
: If specified, executes the specified `target(s)` from the `Makefile`.

Note: Some targets only work if `compile.p` has been executed at least once.

Examples:
The following examples are based on modifications made to the `surf` project. There, before we start issuing any compile commands, we must first be in the correct directory:
`cd /var/tmp/tmpdir/$USER/p4w/naw_CY29R1_test_odb/surf/module`

`compile.p -p surf`
: Compiles all modified files in the current directory.

`compile.i -p surf alldepend`
: Creates the dependencies for all files (modified or not) in the current directory.

`compile.i -p surf build`
: Compiles all files (modified or not) in the current directory.

`compile.i -p surf make-src dependency allloc`
: Finds out all modified files in the current directory, generates the dependencies and compiles the files.

`compile.i -p surf info`
: Prints information on compile relevant environment variables for this project.

`compile.i -p surf arloc`
: Makes new `libsurf.a` library in `/var/tmp/tmpdir/$USER/p4w/$BRANCH/libs`.

`compile.i -p surf buildtar`
: Creates tarfile of source from branch.

G.4.3 Other helpful notes

(i) To clean the local client from all private (compile leftover) files:
`q2 rmprivate -f`
(The "`-f`" option means do not ask for confirmation.)
(ii) `.o` and `.mod` and listing files are generated where the corresponding source is found.
(iii) Executables are put in `$BINS` or `/var/tmp/tmpdir/$USER/p4w/$BRANCH/bin`.
(iv) Libraries are created in `$LIBS` or `/var/tmp/tmpdir/$USER/p4w/$BRANCH/libs`.
(v) The order in which different projects need to be compiled may be inferred from the triggers defined in `scripts/def/gen.def` or from the `VPATH`.
(vi) To find the current branch, e.g. `/var/tmp/tmpdir/$USER/p4w/naw_CY29R1_test`
type
`q2 client_info -R`
(vii) To change compiler options change `Makefile.in.linux` in `scripts/build/arch`.

G.5 IFS DEBUGGING ENVIRONMENT FOR PERFORCE ON WORKSTATIONS

G.5.1 Set up test environment

To set up your own test environment to run the Forecast Model and/or Adjoint Test for T21, login to your workstation and enter:

```
q2 select_client
ifs_setup_perforce
```

This asks which cycle, and copies a test environment to:
`/var/tmp/tmpdir/$USER/p4w/<branch_name>/ifs_t21`

G.5.2 Compile Perforce branch

Compile complete PERFORCE branch and put new library in
`/var/tmp/tmpdir/$USER/p4w/<branch_name>/libs`
by entering:

```
q2 select_client
compile.p
```

Compile all new routines in a directory and put all the "`.o`'s" in library by entering:

```
cd /var/tmp/tmpdir/$USER/p4w/<branch_name>/ifs/phys_ec
compile.i
compile.i arloc
```

G.5.3 To build `ifsMASTER` and run forecast or adjoint test

To create the executable enter:

```
cd /var/tmp/tmpdir/$USER/p4w/<branch_name>/ifs_t21
./mkabs
```

To run the Forecast, enter:

```
./ifs_run -d
```
(the "`-d`" is optional, it allows you to run using Totalview. The default is to run without Totalview.

To run the adjoint test, enter: `./ifs_adj`

References

Adams, J. C., Brainerd, W. S., Martin, J. T., Smith, B. T. and Wagener, J. L. (1992). FORTRAN 90 *Handbook*. McGraw-Hill.

Andrews, P., Cats, G., Dent, D., Gertz, M. and Ricard, J.-L. (1995). *European standards for writing and documenting exchangeable* FORTRAN 90 *code*. URL http://www.met-office.gov.uk/research/nwp/numerical/fortran90/f90_standards.html.

Clochard, J. (1988). Norme de codage "DOCTOR" pour le projet ARPEGE. *Note de travail "ARPEGE" nr 4*.

Gibson, J. K. (1986). Standards for software development and maintenance. *ECMWF Operations department technical memorandum nr 120*.

Hill, L. (1995). *Règles essentielles pour l'utilisation du langage* FORTRAN 90, CNES.