IFS DOCUMENTATION – Cy47r1
Operational implementation 30 June 2020

PART VI: TECHNICAL AND COMPUTATIONAL PROCEDURES

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

REVISION HISTORY

Changes since CY45R1

- Add missing MPL routines

Changes since CY43R1

None.

Changes since CY41R2

- Minor revisions to chapter 2 sections: High performance computer architecture; IFS parallelisation issues; EQ REGIONS; FOURIER transform; Legendre transform.
- Deletion of outdated appendicies F and G.

Changes since CY41R1

- New chapter 3.

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 to deliver 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 use 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 1.1 on page 96), or by using the namelist variable `NCONF` in namelist `NAMCTO` (see Table 3.3 on page 100). A detailed description of the recognised values of `NCONF` is given in Table 4.11 on page 105.

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 4.11 on page 105 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 STEP0 (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 4 on page 105), 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.

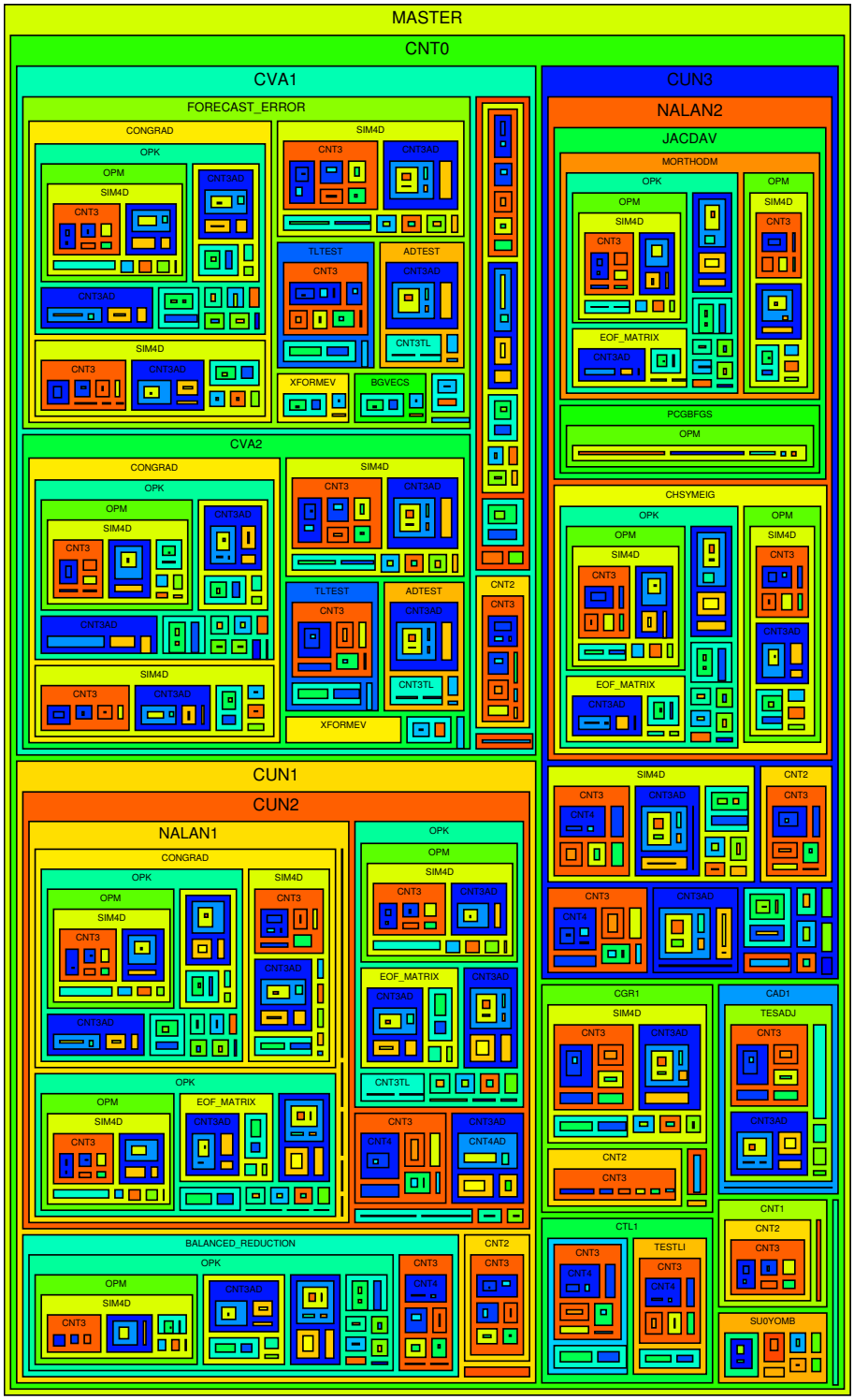


Figure 1.1 *Treemap of the* IFS *Calling Tree.*

Details of the input data files used by the IFS can be found in Section 5 on page 106.

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 7 on page 112.

1.4.3 Integration of Atlas field library

In preparation of performance engineering work focused on future accelerator architectures, the Atlas library has been chosen to provide an object-based field API to provide active control over data placement and layout in memory. To prepare for this migration a new object-based field API has been introduced in cycle 46 through parts of the PHYS_EC component that wraps existing GMV and GFL fields to demonstrate the necessary API changes throughout the scientific code. More details on the new API can be found in Section 7 on page 112.

1.5 CODING STANDARDS AND CONVENTIONS

In this section, brief highlights of some of the most important features of the IFS coding standards are given, to be used when writing and submitting code to the IFS system. It is recommended that anyone planning on writing any significant amount of code for IFS refers to the up-to-date Coding Standards document to be found online (at the time of writing: http://www.umr-cnrm.fr/gmapdoc/IMG/pdf/coding-rules.pdf).

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 `USE`d in the order:
  - variables `USE`d 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.

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

1.5.3 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(:)).

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.

In addition, the CPUs in current general purpose HPC architectures achieve parallelism through vectorisation. 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`). Although these CPUs offer high levels of performance, to achieve the maximum performace requires that the vector units are continually provided with data to operate on. This can be difficult to achieve as the memory bandwidth is not sufficient to deliver this data at the rate that would be required. To achieve the best performance it is important to use data that is already present in the local cache of the CPU since the bandwidth to cache is much higher than to main memory. For this reason it is typically better to operate on short vectors so that once data is loaded from main memory into local cache it can be re-used multiple times without being flushed from cache.

An alternative type of computer architecture makes use of an accelerator which is connected to a general purpose host processor. Modern accelerator processors such as Graphics Processing Units (GPUs) have much higher levels of floating point performace though multiple large vector units, albeit at a lower clock-speed than general purpose processors. GPUs also have much higher memory bandwidth allowing them to operate efficiently with long vectors.

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 parallelise the work in the inner loop. In addition, for accelerator architectures the movement of data between host and accelerator must be managed by the programmer.

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.

In practice, a node contains one or more physical processor chips, with each chip containing multiple CPUs (also known as cores). Each physical processor chip typically has some memory attached directly, and it is also able to access the memory attached to the other chips within the node. Access to another chip's memory is not as fast as accessing a chip's own memory. For this reason this type of node is known as a Non-uniform memory access (NUMA) node, and a single chip with its own memory is known as a NUMA region. The best performance is often achieved when the on-node parallelism is limited to a single NUMA region, and the other NUMA regions are treated as different nodes.

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

[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 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).

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 computer architecture of the (small values, typically a few 10's for general purpose processors, and larger values (100's or 1000's) for systems with large vector units such as GPUs).

The innermost loop should perform well on the CPU (for general purpose processors the data size should be small enough to fit in cache, 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 145). 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

[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.

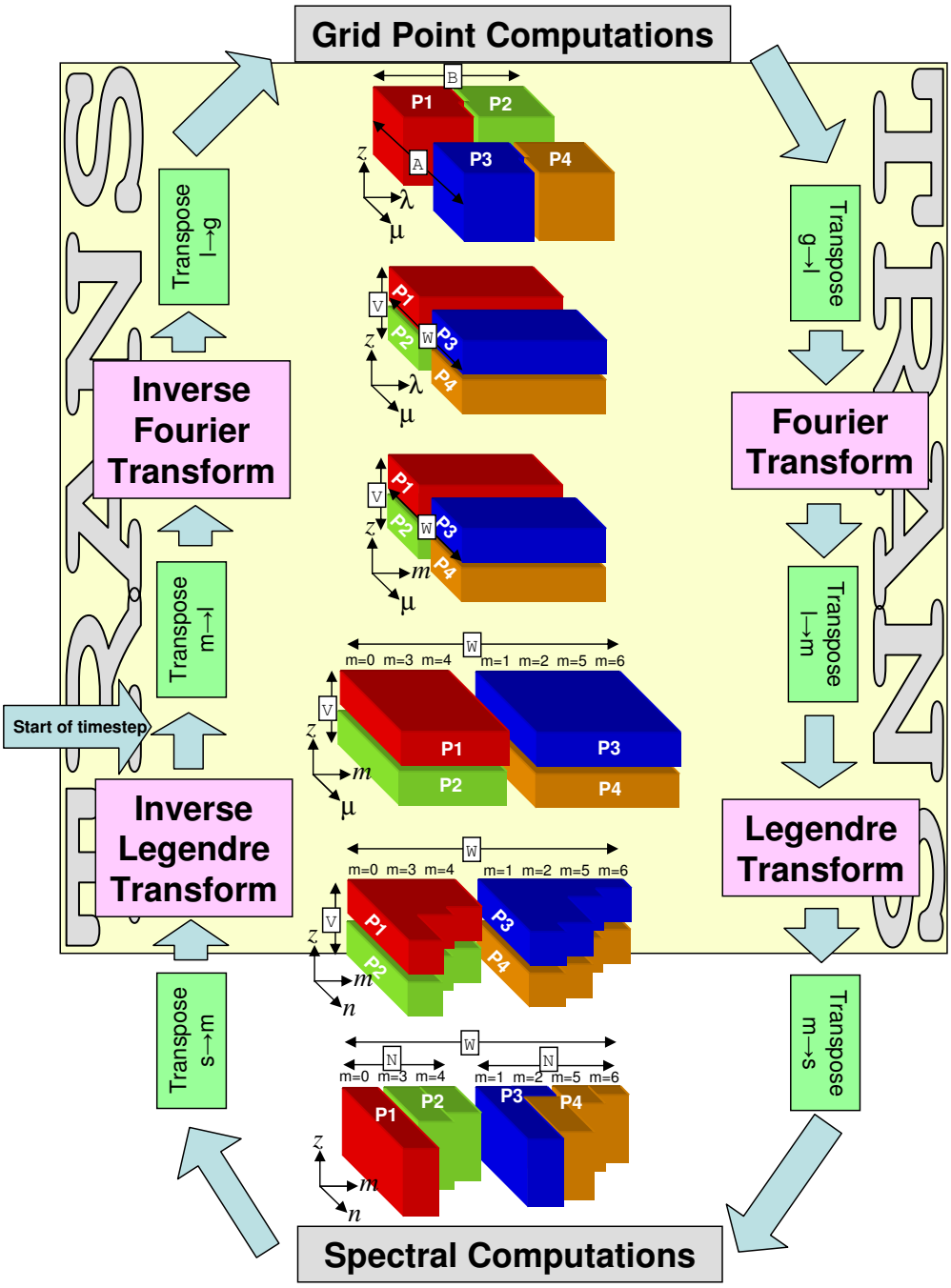


Figure 2.1 IFS *Model Timestep, showing data decompositions.*

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:

Setup Phase

A call to SETUP_TRANS0 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

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

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.

Unfortunately, there is sometimes some confusion between the use of the terms "node", "processor", "processing element (PE)" 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 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

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

total number of PROCessors (CPUs) being used for the job, but the number of MPI tasks. The total number of processors (PEs or 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 YOMCT0 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`.

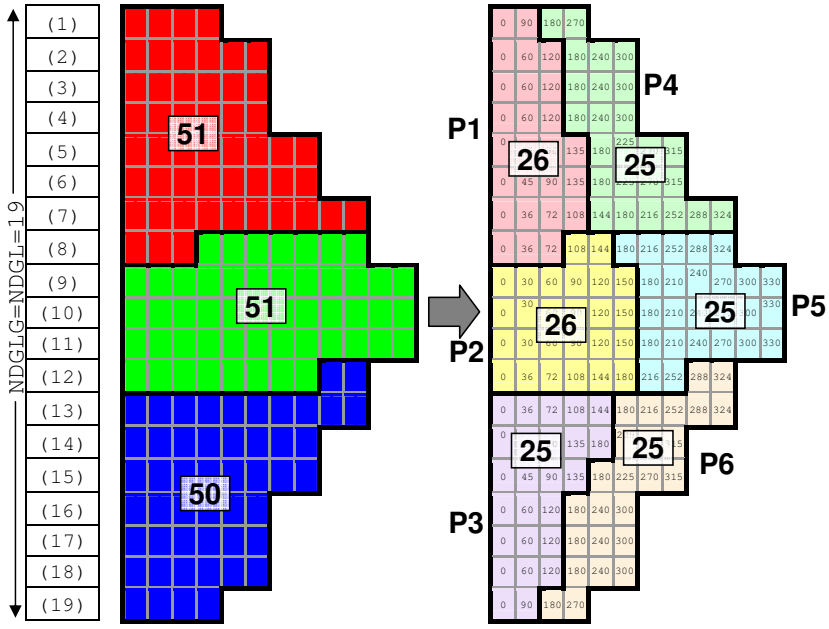


Figure 2.2 *Grid point decomposition, showing the two stages of decomposition on 6 processors.*

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

[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 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.

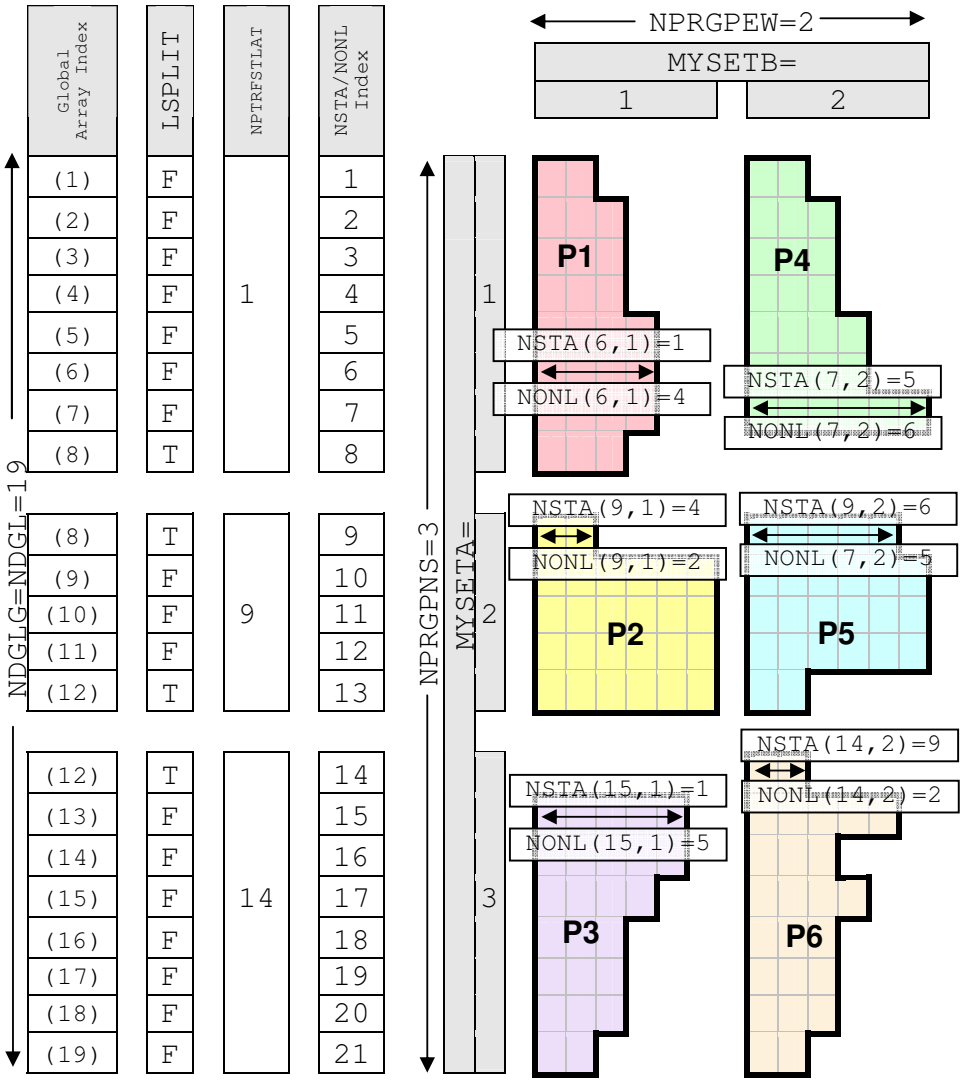


Figure 2.4 *NSTA and NONL in the Grid Point Decomposition.*

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 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
```

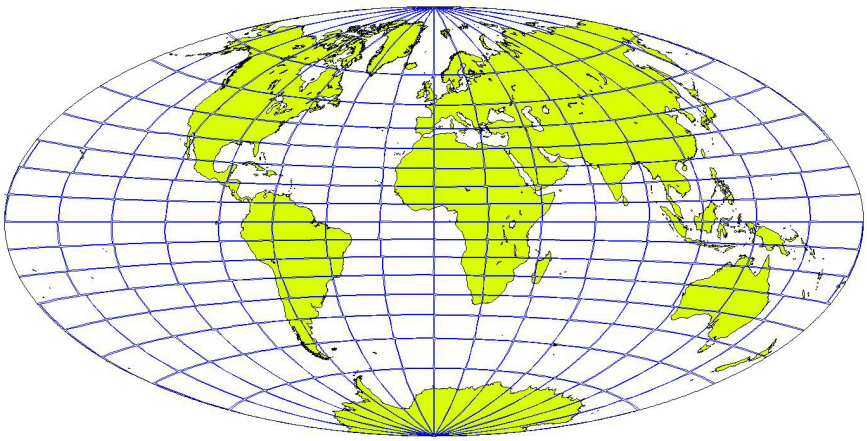


Figure 2.5 *2D partitioning, for 256 MPI tasks.*

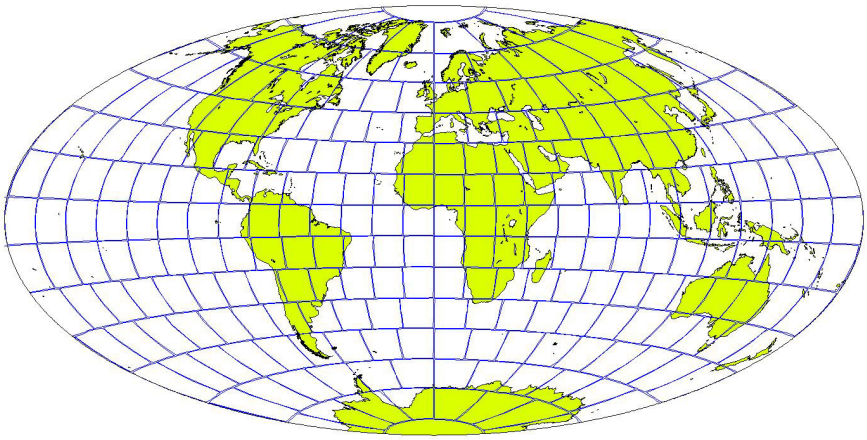


Figure 2.6 *EQ_REGIONS partitioning, for 256 MPI tasks.*

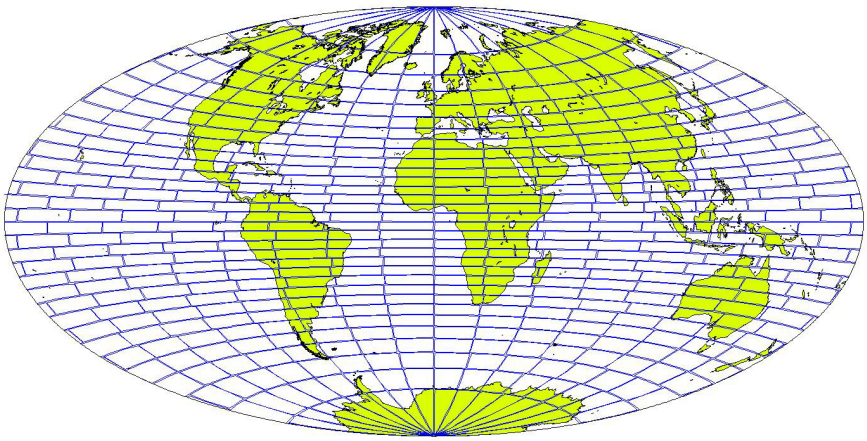


Figure 2.7 *2D partitioning, for 512 MPI tasks.*

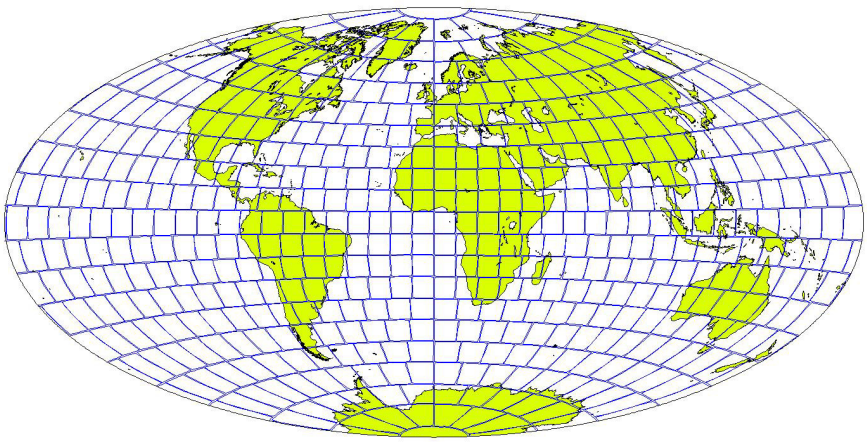


Figure 2.8 *EQ_REGIONS partitioning, for 512 MPI tasks.*

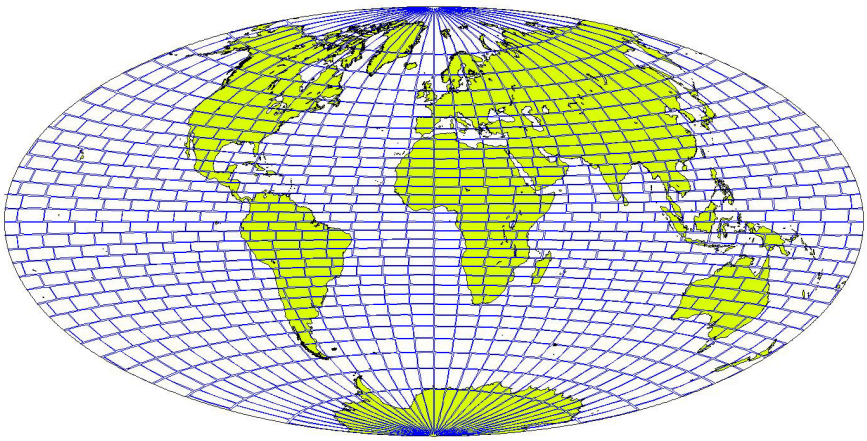


Figure 2.9 *2D partitioning, for 1024 MPI tasks.*

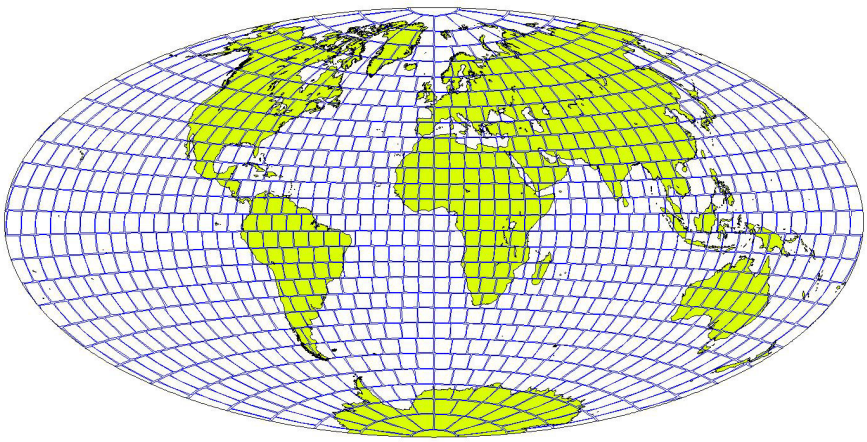


Figure 2.10 *EQ_REGIONS partitioning, for 1024 MPI tasks.*

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 using the reduced Gaussian grid, 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 is not the case for the cubic octahedral grid where the least density is at the mid-latitudes). 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.

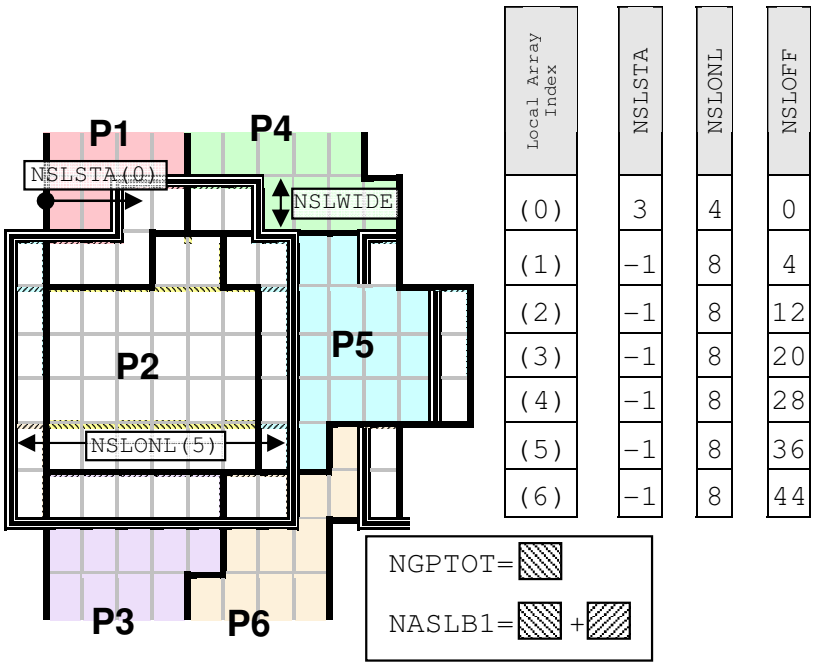


Figure 2.11 *The semi-Lagrangian Halo (For processor 2).*

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, 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.

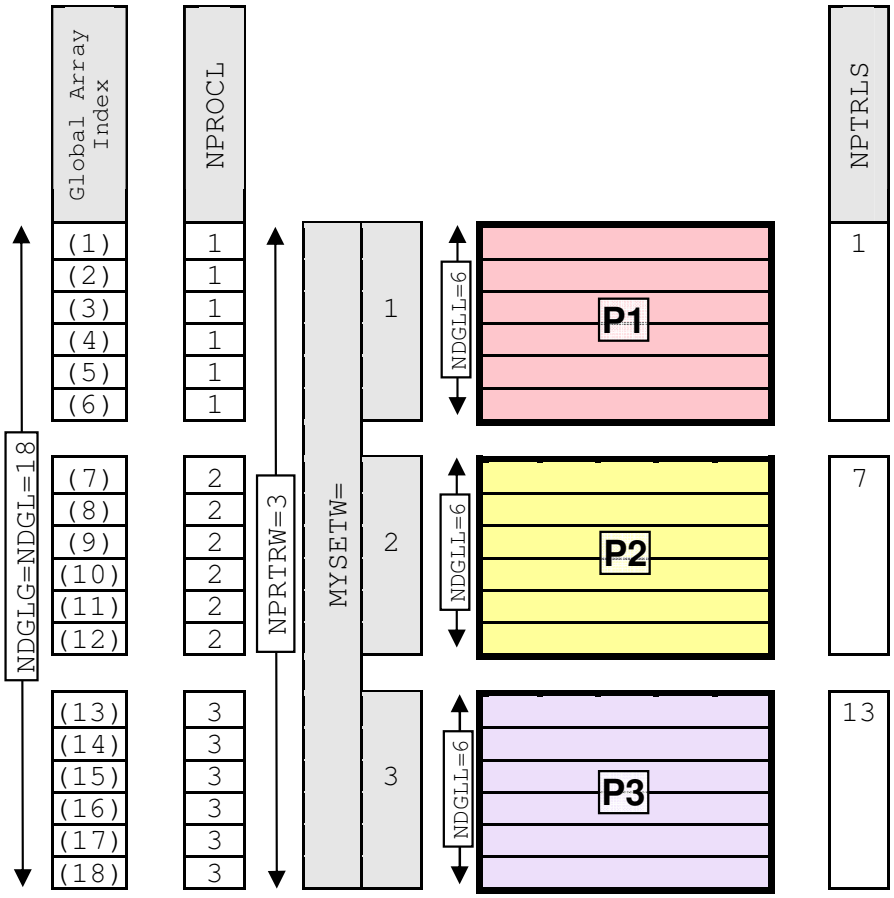


Figure 2.12 *Decomposition of zonal waves in Fourier space. Shown here is a single (first) vertical level.*

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.

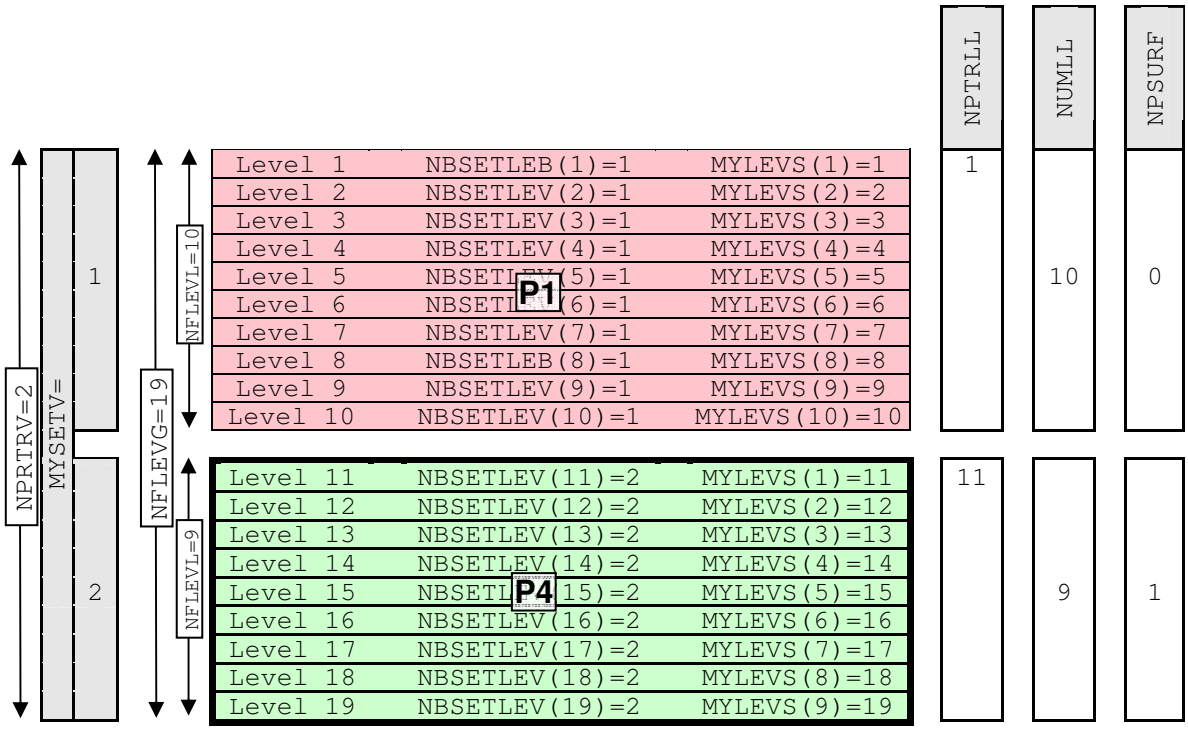


Figure 2.13 *Decomposition of vertical levels used in Fourier and spectral space. Shown here is a single "W" set.*

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)

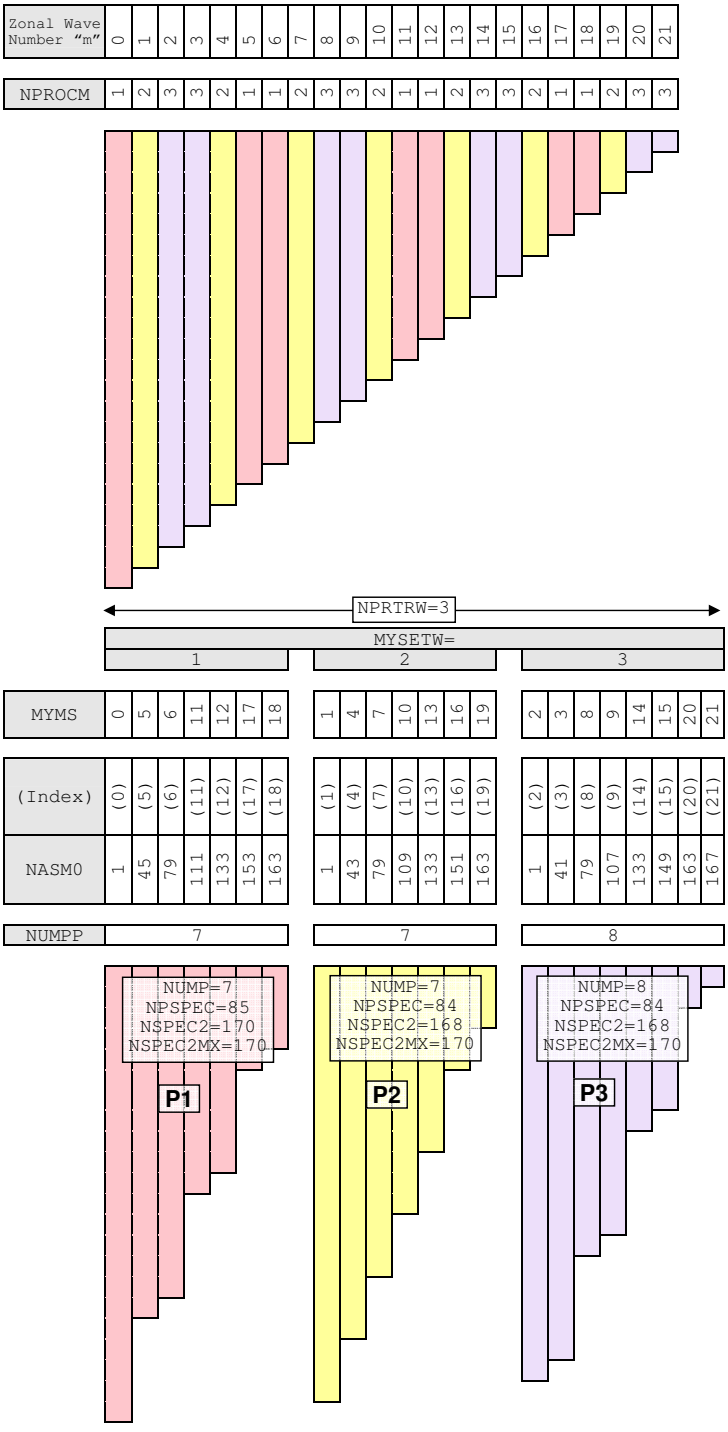


Figure 2.14 *Distribution of zonal wave numbers (T21 spectral triangle).*

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.

The variables used to describe this global data structure are shown in Table 2.6. The ordering of the global spectral fields is that used in the output GRIB.

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.

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

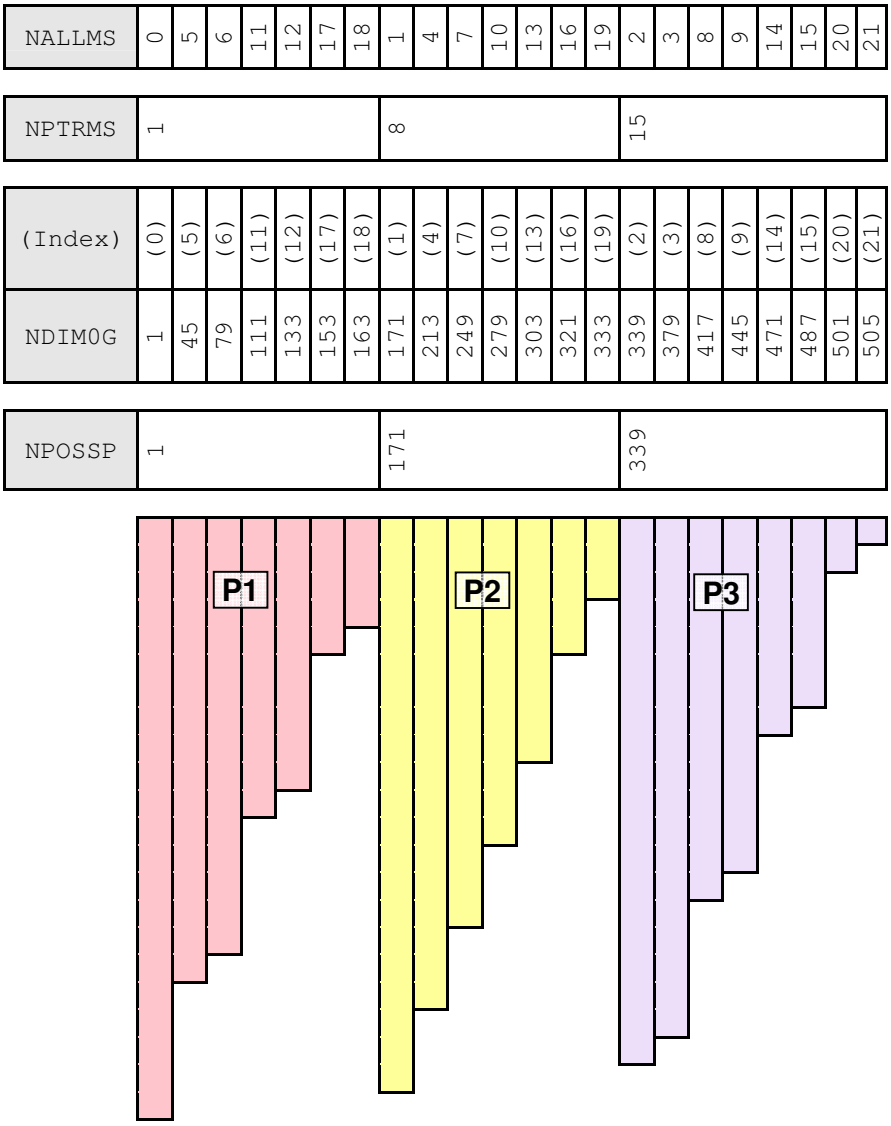


Figure 2.15 *Global representation of T21 spectral triangle.*

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.

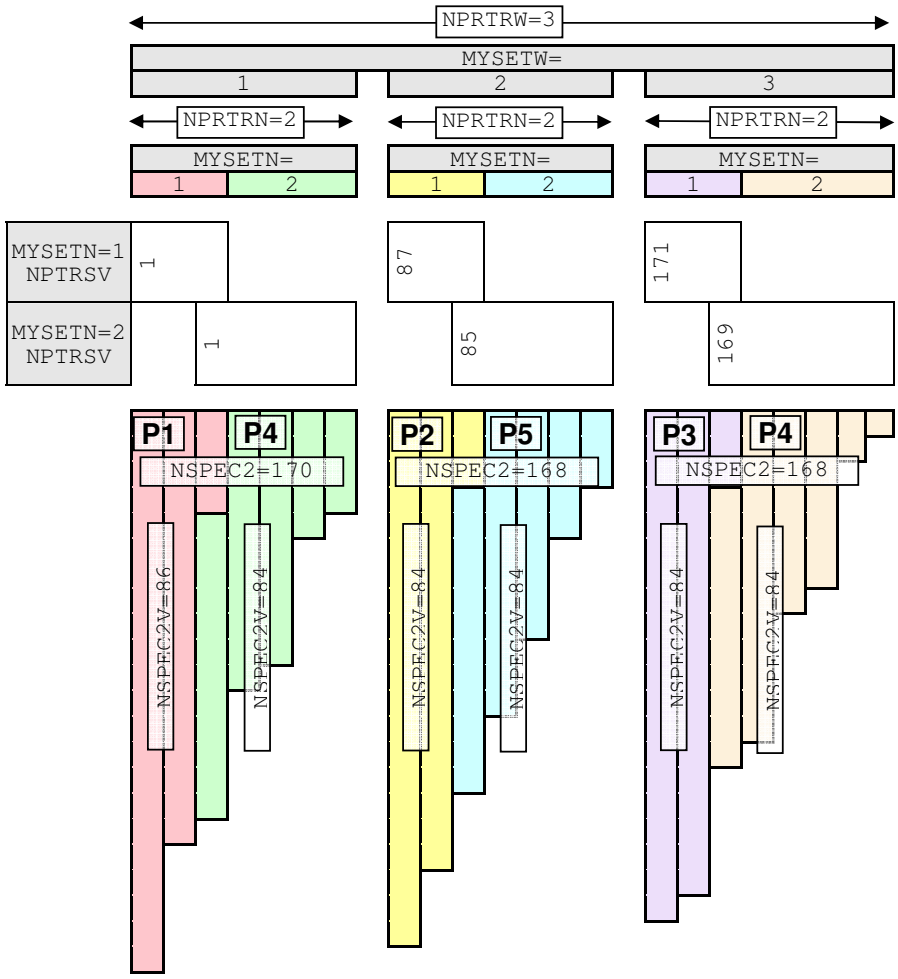


Figure 2.16 *Decomposition of spectral data for the semi implicit calculations.*

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.

Chapter 3

The Python data assimilation suite definition for ecFlow

Table of contents

3.1 INTRODUCTION

EcFlow is a package for running and monitoring complex suites of jobs across a range of machines. It has been written at ECMWF as a replacement for SMS. It submits jobs and receives signals from jobs when they change status or set events. It uses a suite definition file to define the relationships between jobs and the resources which they require, and is able to submit jobs dependent on triggers. This document describes how the Python language is used to create a suite definition file for the data assimilation.

The aim was to write a very general definition which could be shared as far as possible by both the Research Department (RD) and the Forecast Department (FD). RD data assimilation experiments should be able to run from any start date and time to any end date and time. They should be able to run at any pre-defined horizontal or vertical resolution, with appropriate resources for the chosen resolution. So, for example, very low resolution experiments should run quickly with minimum resources to facilitate fast debugging of new code. They should be able to run with any 4D-Var configuration, eg 6-hour 4D-Var,

12-hour 4D-Var, 24-hour 4D-Var. They should be able to run with any combination of streams, eg long-window 4D-Var, early delivery data assimilation, ensemble data assimilation. They should be configurable at run-time, using parameters defined by the RD experiment preparation package, prepIFS.

The operational suite runs one day at a time, to a schedule, in real-time. It runs a prescribed set of suites at a prescribed resolution and with a prescribed 4D-Var configuration. Results are required as fast as possible and machine resources are made available accordingly.

Python is a modular object-oriented language. A number of different modules have been created to contain sets of classes which describe different aspects of the data assimilation suite definition. For example, module `inc_obs.py` contains classes to define the data and methods for observation handling, while module `inc_libs.py` contains classes for building libraries and binaries on different platforms. Class methods are used to define code which can be shared by different parts of the suite definition, avoiding the need for duplicated code. Where different configurations require different code, child classes can be defined which inherit all the shared data and methods from the parent class and only need to redefine those parts which need to be done differently.

There are a number of fundamental differences in design between the old SMS suite definition and the new Python ecFlow definition and some of these are outlined below. The SMS definition was written in the special SMS definition language. The data assimilation suite definition started at the beginning and continued through to the end in a single file. Different streams, eg long-window 4D-Var and early delivery, had repeated blocks of code, rather than a single method which is called repeatedly, as is the case for the Python ecFlow definition. This means that when new code was added, the same code often had to be added in more than one place. Different configurations, such as the forecast sensitivity to observations, had completely separate definition files and the same changes had to be added there too.

In the SMS definition file, each stream was defined for one day at a time, with loops over the hours within the day. A significant proportion of the suite definition included if-blocks with one set of triggers for the first cycle in the day and another for later cycles. The entire day had to complete before the first cycle of the next day could start. If the last cycle contained a 10-day forecast, then the first cycle of the next day could not start until all 10 days had completed, even though the necessary first guess data was available once the forecast had passed the length of the 4D-Var window. Python has built-in methods for date and time manipulation. In the ecFlow definition, date-dependent triggers are calculated for each cycle and tasks can run as soon as all their data is available. Where possible, only genuine data triggers are used within the Python suite definition and courtesy triggers, to simplify the definition, are avoided.

3.2 ORGANISATION

3.2.1 Code location

The Python code is stored in a number of files in the ECMWF Research Department Perforce library, in directory `scripts/def` (which was also the directory used for storing the SMS suite definition files). It shares the same version control as the IFS source code. Changes to the Python definition can be tested in the prepIFS environment by supplying them on a Perforce branch, referenced by the prepIFS `BRANCH` or `SCRIPTS_BRANCH` parameter.

The top-level module for the data assimilation is file `an_ecf.py`, where '`an`' is the prepIFS experiment type for data assimilation. Similarly, the top-level module for the forecast sensitivity to observations configuration is file `fsobs_ecf.py`, which has '`fsobs`' as its prepIFS experiment type. File `an_ecf.py` is very short. Its contents will be described in more detail in the data flow section below, but mainly consist of a call to method `suite()` of `class AnalysisSuite()` in `module inc_an.py`.

File `inc_an.py` is a Python module which contains the definition of `class AnalysisSuite()`, together with its data and methods, and its child classes, such as `class FsobsSuite()`. Method `suite()` of `class AnalysisSuite()` defines the '`make`', '`obs`', '`main`', '`lag`' and '`wsjobs`' families, together with the looping structure over dates and times.

File `inc_fam.py` contains the detailed definition of the analysis and forecast families.

File `inc_obs.py` contains `class Observations()`, with all the data and methods for processing a run-time selection of observations.

File `inc_stream.py` contains a class for each stream type, e.g. `class StreamLWDA()` for long-window data assimilation and `class StreamELDA()` for ensemble data assimilation, together with the data and methods which enable different streams to be handled appropriately. It also contains `class StreamControl()`, which determines the run-time configuration of streams and cycle times.

File `inc_libs.py` contains `class Library()`, which has methods for compiling libraries and building binaries on different platforms.

File `aeolus.py` contains `class Aeolus()` for processing Aeolus data, with methods which do nothing unless switch `LAEOLUS` from module `parameters.py` has the value 'True'.

File `inc_common.py` contains classes and methods which should be able to be shared by suites other than data assimilation. `Class Resources()`, for example, defines resources required at submission time by a number of different parallel jobs.

File `ecf.py` contains extra layers on top of the original ecFlow python, providing added functionality.

An additional file, `parameters.py`, contains variables which define the run-time configuration of the data assimilation. For the Research Department setup, this file is constructed by prepIFS.

(i) Import statements

Python uses import statements to determine where to find any code or data which is not in the current module.

```
import inc_an as ia
import parameters as ip
import ecf as ecflow

user = ip.OWNER
DEFS = ecflow.Defs()
ia.AnalysisSuite(DEFS).suite()
```

In the above example, the value of variable `OWNER` from `module parameters.py` is assigned to local variable '`user`'. The import statement

```
import parameters as ip
```

enables the shorthand version '`ip.OWNER`' to be used instead of '`parameters.OWNER`'. Similarly, the import statement

```
import inc_an as ia
```

enables the shorthand version '`ia.AnalysisSuite(DEFS).suite()`' to be used instead of '`inc_an.AnalysisSuite(DEFS).suite()`', meaning that `class AnalysisSuite()` and its method `suite()` can be found in module `inc_an.py`.

3.2.2 EcFlow syntax

This section describes the building blocks which are used to define the data assimilation suite. EcFlow has three kinds of nodes - suites, families and tasks. Tasks are submittable and correspond to individual jobs. Families are container nodes which can contain tasks or other families. The top-level node is a suite, which can contain families and/or tasks. Suites can be played into an ecFlow server. Figure 3.1 shows a screen-capture from ecFlowview (the graphical user interface for ecFlow), showing two RD ecFlow servers, called '`rdda1`' and '`rdda2`' (this is the same naming convention as was used for the old research department SMS servers). Below the ecFlow server is a suite for each user. Below the user's suite node is a family for each experiment. In the example shown in Figure 3.1, suite '`dah`', belonging to user '`dah`', has been played into server '`rdda1`'. Below suite '`dah`' are three families, containing experiments '`g7gt`', '`g93w`' and '`g8yc`'.

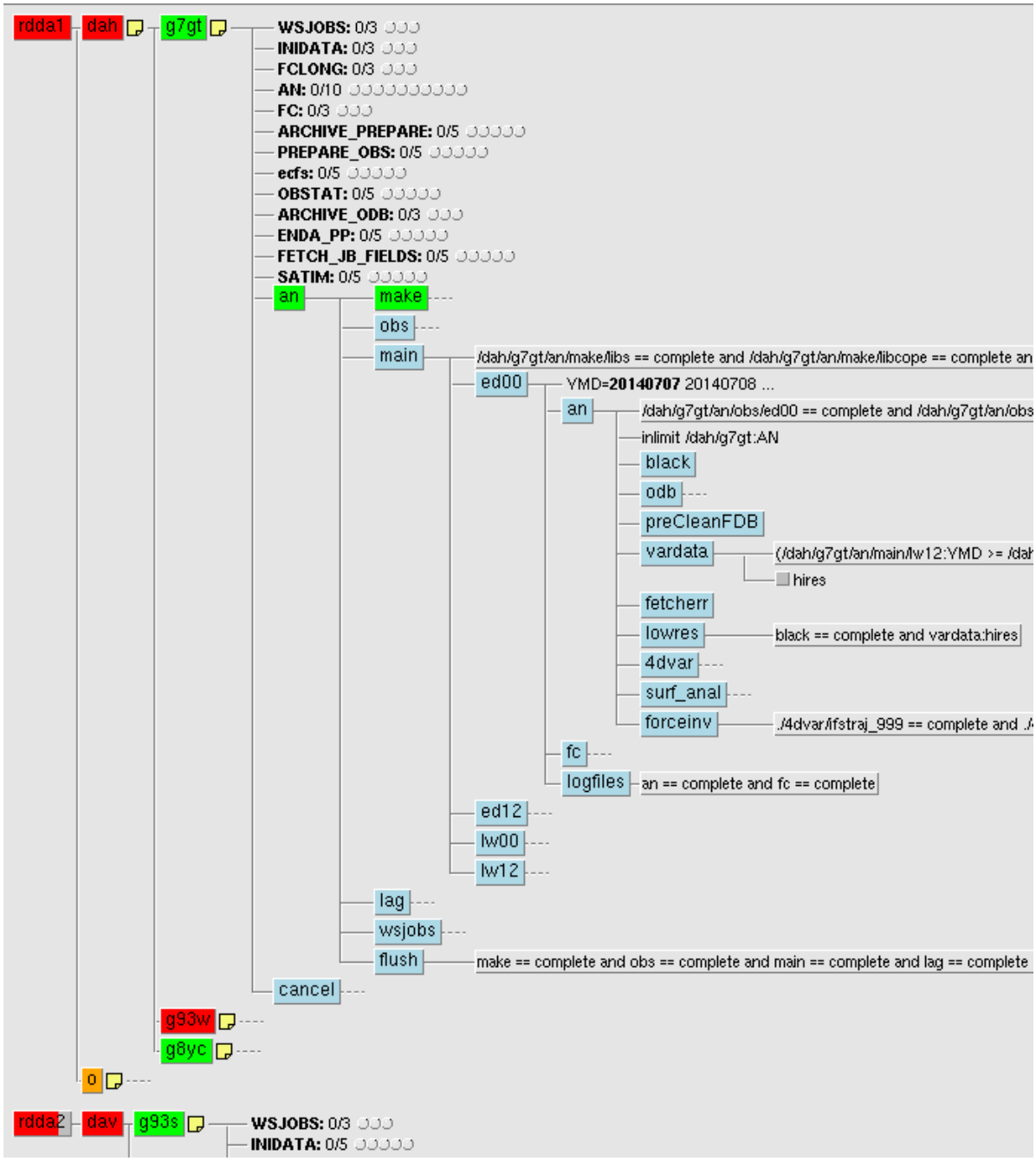


Figure 3.1 *EcFlowview display of 3 experiments belonging to user 'dah', running on ecFlow server 'rdda1'*

(i) Creating a suite

The following code creates an ecFlow definition file and adds a suite to it:

```
import ecf as ecflow

DEFS = ecflow.Defs()
s1 = DEFS.suite("my_suite")
```

The `import` statement defines where any code which is not in the current file can be found. The line

```
DEFS = ecflow.Defs()
```

states that an instance of `class Defs()` from module `ecf.py` is created and assigned to the local variable `DEFS`. The version of `class Defs()` in module `ecf.py` has enhanced functionality compared with the standard version of ecFlow.

The line

```
s1 = DEFS.suite("my_suite")
```

executes method `suite()` of `class Defs()` with the string "`my_suite`" as argument. This creates a suite called '`my_suite`', adds it to the ecFlow definition and assigns the node address of suite '`my_suite`' to local variable `s1`. '`my_suite`' is an instance of `class Suite()` from module `ecf.py`.

(ii) Adding a family

The following code adds a family called '`my_family`' to suite '`my_suite`':

```
import ecf as ecflow

DEFS = ecflow.Defs()
s1 = DEFS.suite("my_suite")
f1 = s1.family("my_family")
```

The line

```
f1 = s1.family("my_family")
```

executes method `family()` of `class Suite()` from module `ecf.py` with the string "`my_family`" as argument. This creates a family called '`my_family`', adds it to suite '`my_suite`' and assigns the node address of family '`my_family`' to local variable `f1`. '`my_family`' is an instance of `class Family()` from module `ecf.py`.

(iii) Adding a task

The following code adds 2 tasks to family '`my_family`' and 1 task to suite '`my_suite`':

```
import ecf as ecflow

DEFS = ecflow.Defs()
s1 = DEFS.suite("my_suite")
f1 = s1.family("my_family")
f1.task("task_1")
f1.task("task_2")
s1.task("task_3")
```

The line

```
f1.task("task_1")
```

executes method `task()` of `class Family()` from module `ecf.py` with the string "`task_1`" as argument. This creates a task called '`task_1`' and adds it to family '`my_family`'. '`task_1`' is an instance of `class`

Task() from module ecf.py. If the node address of task 'task_1' had been needed later in the suite definition, it could have been saved into a local variable, but in this example, this was not necessary.

The line

```
s1.task("task_3")
```

executes method task() of class Suite() from module ecf.py with the string "task_3" as argument. This creates a task called 'task_3' and adds it to suite 'my_suite'. It is not easy to see from this code that 'task_1' and 'task_2' belong to family 'my_family', while 'task_3' belongs to suite 'my_suite'.

(iv) Showing family blocks

Python has strict rules for indentation. Code in a new block must be indented by 4 spaces. Code in the same block must not be indented. In the example below, Python's 'with' notation is used to create a block which corresponds to the definition of the family and its contents.

```
import ecf as ecflow

DEFS = ecflow.Defs()
s1 = DEFS.suite("my_suite")

with s1.family("my_family") as f1:
    f1.task("task_1")
    f1.task("task_2")
s1.task("task_3")
```

With this notation, it can be seen much more easily that 'task_1' and 'task_2' belong to 'my_family', while 'task_3' belongs to 'my_suite'.

(v) Adding attributes to a node

Method add() can be used to add attributes such as variables, triggers and events to a node. The example below shows how a trigger and a list of variables are added to a task.

```
from ecf import Variables, Trigger, Meter, Label
import inc_common as ic
...
tsk = fan.task("ifstraj_999").add(
          Trigger("uptraj_%d == complete" % (mxup_traj - 1)),
          Variables(mode="traj",
                    uptraj=999,
                    NPES=annpes),
          ic.resources.priority(inc = inc_prior))
```

The import statement at the beginning of the example says that the class definitions for Variables(), Trigger(), etc can be found in module ecf.py. 'fan' is the node address for the analysis family. Method task() is called to create task 'ifstraj_999' and add it to the analysis family. Method add() is called to add a number of attributes to task 'ifstraj_999'.

The first attribute which is added is a trigger. Task 'ifstraj_999' can start when the family which contains the penultimate trajectory completes. In this example, 'mxup_traj' is a local variable which contains the maximum number of updates to the trajectory in the 4D-Var analysis, so (mxup_traj - 1) is the update number of the penultimate trajectory. The Python notation '%d' within the string "uptraj_%d == complete" means 'substitute into the string the integer value which follows the '%' sign after the string'. So if, for example, mxup_traj = 3, then the trigger would be "uptraj_2 == complete", where 'uptraj_2' is the name of the family which contains the penultimate 4D-Var trajectory.

The second attribute to be added is a list of variables. In this example, 3 variables are added - the string variable 'mode' with value "traj", the integer variable 'uptraj' with value 999 and the integer variable 'NPES' with the value contained in the local variable 'annpes'. In this example, the variables are added as

a comma-separated list of 'name= value' pairs. It is also possible to add variables as a dictionary. With the dictionary notation, the above example would look like:

```
Variables( { "mode":  "traj",
             "uptraj": 999,
             "NPES":   annpes } ),
```

If a variable is added to a task, or is inherited from a parent node, then within the task the value of the variable can be accessed as `%variable%`. So, for example, the same Python code can be shared by all members of an ensemble, with the actual ensemble member number determined by evaluating the ecFlow variable `MEMBER = %member%`.

The third attribute to be added in the example above is the result returned by method `priority()` of instance '`resources`' of `class Resources()` in module `inc_common.py`. This is a string of the form "`Variables(priority=value)`", where '`value`' is determined by method `priority()`.

(vi) Adding tasks and families recursively

As well as adding triggers, variables, events, meters, etc to a node, it is also possible to use the `add()` function to add tasks and families recursively. These tasks and families can in turn use the `add()` function to add attributes to themselves. In theory, any level of recursion is allowed, but in practice the code would soon become unreadable. An example of adding tasks recursively can be seen in method `build_bins()` of `class Library()` in module `inc_libs.py`.

```
from ecf import Variables, Trigger, Task

...
with node.family("bins") as fbins:
    fbins.add(
        Trigger(trigs),
        Task("odbtools"),
        Task("b2otools").add(
            Trigger("odbtools == complete"),
            Variables(MEM=ip.MEM_BIG_SERIAL)),
        Task("ifs").add(
            Trigger("b2otools == complete")),
```

Note the use of a capital letter in '`Task`'. It is an instance of `class Task()` from module `ecf.py`. As well as being a child class of `ecflow.Task()` and `ecf.Node()`, it is also a child class of `ecf.Attribute()`, which is what enables it to be added to a node with the `add()` function.

(vii) Trigger syntax

The arguments to `class Trigger()` from module `ecf.py` can be given in a number of different forms. Suppose we have a suite of the form:

```
suite my_suite
    family fam1
        task task1
        task task2
    family fam2
        task task3
        task task4
```

If `task4` has to wait for `task3` to complete before it can start, then its definition could be:

```
f2.task("task4").add(
    Trigger("task3 == complete"))
```

In this case, the relative path of `task3` is used. Alternatively, the absolute path of `task3` could be used:

```
f2.task("task4").add(
    Trigger("/my_suite/fam2/task3 == complete"))
```

If task4 must wait for both task3 and family fam1 to complete, then the relative path notation would be:

```
f2.task("task4").add(
    Trigger("task3 == complete and ../fam1 == complete"))
```

and the absolute path notation would be:

```
f2.task("task4").add(
    Trigger("/my_suite/fam2/task3 == complete and " \
            "/my_suite/fam1 == complete"))
```

The argument to the Trigger() class can also be a list of triggers. With the relative path notation, the above example would be:

```
f2.task("task4").add(
    Trigger(["task3", "../fam1"]))
```

The list is a comma-separated collection of strings inside square brackets, where the strings contain the names of the tasks or their relative paths or their absolute paths. If the status is not specified explicitly, then "complete" is the default value. Other possible values for the status are "submitted", "active", "suspended", "aborted", "queued" and "unknown".

It is also possible to use node addresses, rather than node names, in the arguments to the Trigger() class. If the suite definition had saved the task and family addresses in local variables, e.g.

```
f1 = s1.family("fam1")
f2 = s1.family("fam2")
t3 = f2.task("task3")
```

then the node addresses could be passed as arguments to the Trigger() class:

```
f2.task("task4").add(
    Trigger([f1,t3]))
```

Trigger lists can also contain events and meters.

(viii) Date-dependent triggers

The ECMWF research department data assimilation suite structure consists of a top-level family for the experiment and a family of type "an" which has a number of other families below it.

```
suite ${user}              # where ${user} = user id
    family ${exp}          # where ${exp} = experiment id
        family an
            family make    # prepare binaries and initial data
            family obs     # pre-fetch observations
            family main    # analysis and forecast computation
            family lag     # post-process data
            family wsjobs  # plotting etc on workstation
```

Beneath the 'obs','main', 'lag' and 'wsjobs' families is a common stream/date/time repeat structure and there is a standard form for date-dependent triggers between these families.

The 'obs' family for a given stream and hour runs ahead of the corresponding 'main' family by a pre-determined number of days, pre-fetching observations and other files from MARS and ECFS. The number of days by which it can run ahead is taken from variable 'ANALYSIS_LAG' from module parameters.py and stored as a node variable of the same name on the top-level experiment family node. If there is going to be a planned long outage of either MARS or ECFS, then the triggers on the 'obs' family can be over-ridden to pre-fetch extra days of data. This is done by increasing the value of variable 'ANALYSIS_LAG' on the

experiment family node. With all its input data available on supercomputer disk space, the computation-heavy '`main`' family can then continue to run, keeping the supercomputer busy.

The '`lag`' family post-processes and archives the data. If the data storage systems are unavailable, the '`main`' family can continue to run, storing its output data on supercomputer disk until the '`lag`' family is able to archive the data and free the disk space. The number of days by which the '`lag`' family can lag behind the '`main`' family for a given stream and hour is stored in variable '`ARCHIVE_LAG`' on the experiment family node. Increasing the value of this variable will increase the numbers of days that the '`main`' family can continue to run before it has to wait for the '`lag`' family to catch up (but beware - the greater the value of '`ARCHIVE_LAG`', the higher is the risk that the supercomputer disks could be filled by a rogue experiment).

The '`wsjobs`' family runs on the workstation, doing plotting and similar tasks, using data which has been prepared on the supercomputer by the '`lag`' family. Again, if there are delays on the workstation systems, the '`lag`' family can continue to run on independently.

Date repeats on the '`main`' and '`lag`' families for long-window 12-hour 4D-Var are structured as:

```
family main
    family lw00 YMD= startdate, ..., enddate
        family an
        family fc
    family lw12 YMD= startdate, ..., enddate
        family an
        family fc
family lag
    family lw00 YMD= startdate, ..., enddate
        family fb
        family archive
            ...
    family lw12 YMD= startdate, ..., enddate
        family fb
        family archive
            ...
```

This illustrates the date repeat structures below the '`main`' and '`lag`' families for long-window 12-hour 4D-Var data assimilation. In this example, there is a single stream-type, long-window 4D-Var, whose time families have the prefix "`lw`". For 12-hour 4D-Var, there are cycles at 00 UTC and 12 UTC, which have families '`lw00`' and '`lw12`' below the '`obs`', '`main`', '`lag`' and '`wsjobs`' families. Family '`lw00`', for example, is repeated with date variable `YMD`, of format `yyyymmdd`, running from '`startdate`' to '`enddate`'. Family '`fb`' in the '`lag`' family post-processes data produced by family '`an`' in the '`main`' family. But the trigger is not simply that 'family fb can run if family an is complete'. If the '`main`' family is already at least a day ahead, then it may be executing family '`an`', with the result that family '`an`' is not complete, but the data from the previous day is already available, so family '`fb`' can still run. For the '`lw00`' family in this example, the trigger (using absolute path addresses) becomes:

```
/${user}/${exp}/an/main/lw00/an == complete and \
/${user}/${exp}/an/main/lw00:YMD == /${user}/${exp}/an/lag/lw00:YMD or \
(/${user}/${exp}/an/main/lw00:YMD > /${user}/${exp}/an/lag/lw00:YMD)
```

If we parameterize this as:

```
ahead = /${user}/${exp}/an/main/lw00
behind = /${user}/${exp}/an/lag/lw00
```

then the trigger becomes:

```
${ahead}/an == complete and ${ahead}:YMD == ${behind}:YMD or \
(${ahead}:YMD > ${behind}:YMD)
```

The strings

```
"${ahead}", "${behind}" and
"and ${ahead}:YMD == ${behind}:YMD or (${ahead}:YMD > ${behind}:YMD)"
```

are returned by method `add_ymds()` from module `inc_common.py`. Triggers of this form are widely used between the '`obs`' and the '`main`' families, between the '`main`' and the '`lag`' families and between the '`lag`' and the '`wsjobs`' families.

3.3 DATA FLOW

3.3.1 Creating a data assimilation suite

(i) Create the ecFlow definition

File `an_ecf.py` is very short and contains the top-level definition of the data assimilation suite. It contains the following code:

```
import inc_an as ia
import parameters as ip
import ecf as ecflow
import os

DEFS = ecflow.Defs()
ia.AnalysisSuite(DEFS).suite()

print DEFS
```

The ecFlow suite definition, `DEFS`, is an instance of `class Defs()` from module `ecf.py` (which contains extra functionality on top of the standard ecFlow). It is populated with the data assimilation suite definition by calling method `suite()` of `class AnalysisSuite()` from module `inc_an.py`. The print statement generates a listing of the suite definition. In the prepIFS context, the print-out can be found in the output of task `exp_setup` below family `$exp_setup`. If there are any errors in the Python definition, task `exp_setup` will fail at this point with a meaningful error message.

(ii) Checking the suite definition

The following code fetches a number of environment variables, then checks that all the triggers in the suite definition can be resolved and that all the jobs can be created, i.e. the `.ecf` wrapper files and `include` files are available. If not, the experiment setup fails with a meaningful error message.

```
ECF_HOME = os.getenv("ECF_HOME")
ECF_OUT  = os.getenv("ECF_OUT")
host = os.getenv("ECF_NODE")
port = os.getenv("ECF_PORT")

job_ctrl = ecflow.JobCreationCtrl()
DEFS.check_job_creation(job_ctrl)
```

(iii) Create an ecFlow client and start the experiment

The following code creates an ecFlow client and plays the suite definition into the ecFlow server which is identified by the host and port variables.

```
clt = ecflow.Client(host, port)
path = "/" + ip.OWNER + "/" + ip.EXPVER
clt.sync_local()
clt.replace(path, DEFS, True, True)
```

Module `parameters.py`, generated by prepIFS to hold the experiment configuration, contains variables `OWNER = ${user}`, a string containing the user name of the owner of the experiment, and `EXPVER =`

${exp}, a string containing the experiment identifier. The server has a suite for each user, with below it a family for each experiment.

(iv) Creating other related configurations

Alternatively, several other closely-related configurations can also be set up by prepIFS. The Python code in Subsection 3.3.1 is modified as follows:

```
DEFS = ecflow.Defs()
if ip.LOBSTATFC:
    ia.ObstatFCSuite(DEFS).suite()
elif ip.LVAR_ONE_OBS:
    ia.AnalysisNoFcSuite(DEFS).suite()
else:
    ia.AnalysisSuite(DEFS).suite()
```

Class `ObstatFCSuite()` is a child of `class AnalysisSuite()` which contains the necessary changes to run the forecast departure from observations monitoring suite. It is invoked if variable `LOBSTATFC` in module `parameters.py`, which contains the prepIFS configuration variables, is '`True`'.

Class `AnalysisNoFcSuite()` is a child of `class AnalysisSuite()` where the '`main`' family has an '`an`' family for the analysis, but no '`fc`' family for the cycling forecast. This configuration is necessary for single observation experiments, which are invoked if variable `LVAR_ONE_OBS` in module `parameters.py` has the value '`True`'.

3.3.2 How it is called

As we saw in Subsection 3.3.1 above, the data assimilation suite definition is created by calling method `suite()` of `class AnalysisSuite()`.

```
import inc_an as ia
import ecf as ecflow
....
DEFS = ecflow.Defs()
ia.AnalysisSuite(DEFS).suite()
```

The statement '`ia.AnalysisSuite(DEFS).suite()`' creates an instance of `class AnalysisSuite()`, executes its constructor method and then executes its method `suite()`.

(i) Constructor method of class AnalysisSuite()

At the start of the definition of `class AnalysisSuite()` in module `inc_an.py` is its constructor method:

```
import inc_common as ic
import ecf
import inc_obs as io
import inc_stream as ist
import inc_fam as ifam
from ecf import Variables, Trigger, .., Task, Family, ..
 ...

class AnalysisSuite(ic.Seed):

    def __init__(self, defs=None, obs=None, lim=None, stype=None):
        """ constructor method, where:
           defs  = ecflow suite definition structure
           obs   = observations
           lim   = absolute path for limits
           stype = suite type
```

```
"""
...
# Check input parameters
self.check_input_parameters()

self.maker = ifam.MakeIDATA(plim)

if obs is None:
   obs = io.Observations()
self.observations = obs

self.streamControl = ist.StreamControl()
self.an = ifam.FamAnalysis(self.observations, plim)
self.fc = ifam.FamForecast(plim)
...
```

DEFS in the statement '`ia.AnalysisSuite(DEFS).suite()`' is an argument for the constructor method of `class AnalysisSuite()`.

Method `check_input_parameters()` of `class AnalysisSuite()` is called to check that module `parameters.py` contains a valid set of parameter values. If not, an exception is raised and the python terminates with a meaningful message explaining what is wrong with the input parameters.

An instance of `class MakeIDATA()` from module `inc_fam.py` is created and assigned to class variable `self.maker` of `class AnalysisSuite()`. This contains methods to compile the libraries, build binaries and prepare the initial data, as described in Subsection 3.3.2.(iv).

Because no observations were passed into the constructor method of `class AnalysisSuite()`, an instance of `class Observations()` from module `inc_obs.py` is created and assigned to class variable `self.observations` of `class AnalysisSuite()`. When the new instance is created, its constructor method is executed and a significant amount of work is done behind the scenes. This is described in more detail in Subsection 3.4.2.

An instance of `class StreamControl()` from module `inc_stream.py` is created and assigned to class variable `self.streamControl` of `class AnalysisSuite()`. Its constructor method, which is described in more detail in Subsection 3.5.8, works out the run-time configuration of streams and dates and cross-stream dependencies.

An instance of `class FamAnalysis()` from module `inc_fam.py` is created and assigned to class variable `self.an` of `class AnalysisSuite()`. Later on, in the method which constructs the '`main`' family, method `content()` of `class FamAnalysis()` is called to do the work of constructing the analysis family definition, and it is called as `self.an.content()`. This is described in more detail in Subsection 3.6.2.(ii).

Similarly, an instance of `class FamForecast()` from module `inc_fam.py` is created and assigned to class variable `self.fc` of `class AnalysisSuite()`. Later on, in the method which constructs the '`main`' family, method `content()` of `class FamForecast()` is called to do the work of constructing the forecast family definition, and it is called as `self.fc.content()`. This is described in more detail in Subsection 3.6.3.(ii).

Instances `self.longfc` of `class FamLongForecast()`, `self.fb` of `class FamFeedback()` and `self.ws` of `class FamWSJobs()`, all from module `inc_fam.py`, are also created.

(ii) Method check_input_parameters() of class AnalysisSuite()

Method `check_input_parameters()` of `class AnalysisSuite()`, called from the class constructor method, checks that a valid set of parameters has been supplied in module `parameters.py`. The prepIFS internal checks do most of the parameter checking. However, a few inconsistencies are not corrected by prepIFS. Rather than fail at a later point in the suite preparation in a way that is not obviously related to the incorrect parameter values, it is better to trap them as soon as possible and fail with a meaningful error message.

(iii) Method suite() of class AnalysisSuite()

Method `suite()` of `class AnalysisSuite()` calls a series of methods in order to construct the data assimilation suite definition.

```
import parameters as ip
...
    def suite(self):
        with self.defs.suite(ip.OWNER) as node:
            with node.family(ip.EXPVER) as fexpver:
                self.limits(fexpver)
                self.setup(fexpver)
                with fexpver.family(self.stype) as fan:
                    self.make(fan)
                    self.obs(fan)
                    self.main(fan)
                    self.lag(fan)
                    self.wsjobs(fan)
                    self._endFamily(fan)
                endTrig = self.stype + " == complete"
                self._finalize(fexpver, trig = endTrig)
```

First of all, method `suite()` of `class Defs()` from module `ecf.py` is called to create a suite with name `ip.OWNER` (which is a string containing the user id from module `parameters.py`), add the suite to the ecFlow definition in `self.defs` and return the node address of the suite in variable '`node`', as was described in Subsection 3.2.2.(i) above.

Next, method `family()` of `class Suite()` from module `ecf.py` is called to create a family with name `ip.EXPVER` (which is a string containing the experiment id from module `parameters.py`), add it to the suite and return the node address of the family in variable '`fexpver`', as was described in Subsection 3.2.2.(ii) above.

Methods `limits()` and `setup()` of `class AnalysisSuite()` are called to add experiment-wide limits and variables to the experiment family node. Next, method `family()` of `class Family()` from module `ecf.py` is called to add a family of name `self.stype` below the experiment family and return its node address in variable `fan`. Methods `make()`, `obs()`, `main()`, `lag()` and `wsjobs()` of `class AnalysisSuite()` are called to construct the corresponding families below family '`an`'. Method `_endFamily()` is then called to terminate family '`an`' and method `_finalize()` is called to terminate the experiment.

(iv) Method make() of class AnalysisSuite()

Method `make()` of `class AnalysisSuite()`, called from method `suite()`, prepares experiment file systems, compiles libraries, builds binaries and optionally prepares the initial data.

```
def make(self, node):

    with Family("make") as fmake:
        if node is not None:
            node.add(fmake)
        self.maker.content(fmake)
        self.maker.add_ymd_hour(fmake)
        self.main_trig += self.maker.main_trigs()
        return fmake
```

First of all, the lines

```
        with Family("make") as fmake:
            if node is not None:
                node.add(fmake)
```

demonstrate yet another way to create a family. An instance of `class Family()` from module `ecf.py` with name "`make`" is created and its address is returned in variable '`fmake`'. If method `make()` had been called with a valid node address in argument '`node`' (as is done in the RD configuration), then family '`make`' is added to the node. If method `make()` had been called with a null value for argument '`node`' (as is done in the FD configuration), then the returned variable '`fmake`' at the end of the routine is used to pass the address of family '`make`' back up to the calling routine.

The '`make`' family comes in 2 forms. All configurations need to compile any library changes if an input Perforce branch has been supplied, build binaries on the supercomputer and the workstation and set up the experiment file systems. Some configurations also need to prepare the input data for the first cycle. Configurations without a cycling forecast, such as '`fsobs`' (forecast sensitivity to observations) or '`obstatfc`' (forecast departure from observations monitoring suite), do not need to prepare initial data.

Method `content()` of `class Make()` from module `inc_fam.py` contains the code which is needed by all configurations. `Class MakeIDATA(Make)` from module `inc_fam.py` is a child of `class Make()`. Its method `content()` executes method `content()` of its parent class, then also prepares the initial data. As described in Subsection 3.3.2.(i) above, the constructor method for `class AnalysisSuite()` contains the line:

```
self.maker = ifam.MakeIDATA(plim)
```

while the constructor method for `class AnalysisNoFcSuite()` contains the line:

```
self.maker = ifam.Make(self.limits_)
```

So by storing the appropriate family instance in class variable `self.maker`, the same simple form of method `make()` of `class AnalysisSuite()` can be used for a number of different configurations. Method `content()` of `class Make()` is described in more detail in Subsection 3.6.5.

Method `add_ymd_hour()` of `class Make()` adds date and time variables `YMD` (with format '`yyyymmdd`') and `HOUR` (with format '`hh`') to the '`make`' family node and adds the '`logfiles_ecfs`' task to the end of the '`make`' family (see Subsection 3.3.2.(xiii)).

Certain triggers need to be passed between the '`make`' family and the '`main`' family. `self.main_trig` is a variable of `class AnalysisSuite()` in which the list of triggers for the '`main`' family is constructed. Other triggers need to be passed between the '`make`' family and the '`obs`' family. `self.maker.path_bins` is a variable of `class Make()` which contains the absolute path of the '`bins`' family, which needs to complete before the observation pre-processing computations can start.

(v) Date repeats

Looking at Figure 3.1, the screen-capture from ecFlowview of a data assimilation experiment, it can be seen that below the '`main`' family is a family for each hour of each stream. The experiment has early-delivery cycles at midnight and midday, with families '`ed00`' and '`ed12`', and 12-hour 4D-Var long-window data assimilation cycles, also at midnight and midday, with families '`lw00`' and '`lw12`'. The same structure exists below the '`obs`', '`lag`' and '`wsjobs`' families. Each of these stream/hour cycles has its own `YMD` date repeat from its own start date to its own end date. If an experiment starts at midday, then the start date for the 12 UTC cycles will be a day ahead of the start date for the 00 UTC cycles. Similarly, if the experiment ends at midnight, then the end date for the 00 UTC cycles will be a day later than the end date for the 12 UTC cycles.

The generic method `looper()` of `class AnalysisSuite()` is called from within methods `obs()`, `main()`, `lag()` and `wsjobs()` to loop over all stream/hour families. It is described in more detail in Subsection 3.3.2.(vii) below.

(vi) Method obs() of class AnalysisSuite()

Method `obs()` of `class AnalysisSuite()`, called from method `suite()`, controls the observation fetching and pre-processing.

```
def obs(self, node):
```

```
        """ Observation pre-processing stage, where:
            node = position in suite definition tree
        """
        # Loop over streams / times / dates
        fam = self.looper(node, "obs", self.obs_fam)
        fam.add( Trigger("make/setup == complete) )
        return fam
```

Method `looper()` of `class AnalysisSuite()` is called to create family "`obs`", then loop over all streams and hours, calling method `obs_fam()`, also from `class AnalysisSuite()`, to do the observation processing.

All configurations need to wait until task '`make/setup`' has run and set up the experiment filesystems.

(vii) Method looper() of class AnalysisSuite()

Method `looper()` is the generic method for looping over different streams, dates and hours, thereby avoiding the need to duplicate code for different configurations. The principle part of its body text is given below:

```
    def looper(self, node, name, call_name):
        """ Control looping over dates, times and streams, where
            node = position in suite definition tree
            name = family name, eg "obs", "main", etc
            call_name = method which is called for each
                        stream / time / date
        """
        with node.family(name) as x:
             # Loop over streams
             for streamClass in self.streamControl.streamClassList:
                 # Loop over hours
                 for fam in streamClass.famlist:
                     start = streamClass.startYMD[fam]
                     end = streamClass.endYMD[fam]
                     with x.family(fam) as xfam:
                         xfam.repeat("YMD", start, end)
                         streamClass.set_streamvars(fam)
                         call_name(xfam, fam, streamClass)
            return x
```

First of all, it creates a family called '`name`', where '`name`' is a string variable and is one of the method's arguments, e.g. "`obs`" or "`main`". Next comes a loop over streams. In Subsection 3.3.2.(i) above, it was described how the constructor method for `class AnalysisSuite()` created an instance of `class StreamControl()` and assigned it to the class variable `self.streamControl`. The constructor method for `class StreamControl()`, as described in more detail in Subsection 3.5.8.(i), uses the prepIFS switches from module `parameters.py` to build up a run-time list of streams in `self.streamControl.streamClassList`. It also constructs a list of stream / hour families for each stream in `self.streamControl.streamClassList.famlist`, together with a start date and an end date for each stream / hour family. Method `looper()` creates a family for each stream / hour, sets it to repeat from its own start date to its own end date, calls method `set_streamvars()` of the relevant `streamClass` to add stream-dependent variables to the stream / hour family node and then calls method '`call_name()`' with arguments the node address of the stream / hour family, the name of the stream / hour family and the `streamClass` instance. For the '`obs`' family, method '`call_name()`' is method '`obs_fam()`'.

(viii) Method obs_fam() of class AnalysisSuite()

Method `obs_fam()` of `class AnalysisSuite()` is called to do the observation processing for each stream, hour and date. In catchup mode for the RD configuration, it starts with a call to method

obs_realtimeTrigger() of class AnalysisSuite() to ensure that experiments which are running very close to real time do not fetch their observations until they have been archived by the operational suite. Method obs_fam() then calls methods of class Observations() from module inc_obs.py, as described in Section 3.4, to do the processing for the run-time selection of observations. A trigger is added to the stream / hour family to prevent it running more than 'ANALYSIS_LAG' days ahead of the corresponding 'main' family, as described in Subsection 3.2.2.(viii) above.

(ix) Method main() of class AnalysisSuite()

Method main() of class AnalysisSuite() has a similar structure to method obs(), as described in Subsection 3.3.2.(vi) above. It calls method looper(), which in turn calls method main_fam() to define the 'main' family for different streams, dates and hours. Method main_fam() calls methods of classes FamAnalysis() and FamForecast() from module inc_fam.py to define the analysis and forecast families. It also calls methods calc_eda_statistics() and calc_jb_statistics() of the appropriate stream class from module inc_stream.py to calculate the ensemble data assimilation statistics and JB statistics (or do nothing if it is not an ensemble stream).

(x) Method lag() of class AnalysisSuite()

Method lag() of class AnalysisSuite() has a similar structure to methods obs() and main(). It calls method looper(), which in turn calls method lag_fam() to define the 'lag' family for different streams, dates and hours. Method lag_fam() calls methods of class FamFeedback() from module inc_fam.py and methods from classes Observations() and AnalysisSuite() to define the archiving to MARS and ECFS and the obstat plotting.

(xi) Method lag_fam_arch() of class AnalysisSuite()

Method lag_fam_arch() of class AnalysisSuite() is called by method lag_fam(). It is triggered by the corresponding forecast family from the 'main' family being complete. It creates family 'archive'. Method archive_odb() from class Observations() in module inc_obs.py is called to archive the Observation Databases (ODBs) to MARS. Family 'archive_odb' is triggered by family '../fb/archive_prepare' being complete.

Method streamClass.err_save() from module inc_stream.py is called to archive Ensemble Data Assimilation (EDA) errors to MARS. Method err_save() is a dummy routine for all streams except streamELDA(). If switch 'LEDA_ERRORS_OUT' from module parameters.py has the value 'True', then task 'eda_err_save' is added, triggered by families 'enda_pp' and 'jb_stats' from the corresponding family 'an' in the 'main' family being complete.

Family 'archive_fields' is created. Method self.an.archiver() of class FamAnalysis() from module inc_fam.py is called to archive the analysis and short forecast fields to MARS. If there is a long forecast at the current cycle (with a length longer than the cycling forecast), then method self.fc.archiver() of class FamForecast() or method self.longfc.archiver() of class FamLongForecast() from module inc_fam.py is called to archive the long forecast fields to MARS.

(xii) Method wsjobs() of class AnalysisSuite()

Method wsjobs() of class AnalysisSuite() calls method looper() with argument method content() of class FamWSJobs() from module inc_fam.py to define the plotting and verification tasks on the workstation for each stream, date and hour.

(xiii) Logfiles

Logfiles are processed at the end of the 'make' family, and at the end of each stream / hour family for the 'obs', 'main', 'lag' and 'wsjobs' families. This is done from either task 'logfiles' (for the 'obs' and 'main' families) or task 'logfiles_ecfs'. Task 'logfiles' reorganises files on the supercomputer disks. Task 'logfiles_ecfs' also writes gzipped tar files to ECFS. The 'logfiles_ecfs' jobs have the attribute '# QSUB -EC ecfs_write = yes', which makes it possible for them to be held back if ECFS is not available.

On the supercomputer, logfiles are initially written to a log directory of the form:

```
$LOGDIR/$USER/$EXPVER/an/make        or
$LOGDIR/$USER/$EXPVER/an/[obs|main|lag]/$streamHour.
```

For the `make` family, the '`logfiles_ecfs`' task writes the gzipped tar file directly to:

```
ec:/$FSROOT/$EXPVER/log/an/make.$yyyymmdd.tar.gz
```

For the '`obs`', '`main`' and '`lag`' families, the '`logfiles`' or '`logfiles_ecfs`' task moves the contents of the log directory to:

```
$LOGDIR/$USER/$EXPVER/an/$yyyymmdd.$streamHour/[obs|main|lag]
```

For the '`lag`' family, the '`logfiles_ecfs`' task then makes a gzipped tar file from the contents of:

```
$LOGDIR/$USER/$EXPVER/an/$yyyymmdd.$streamHour
```

And writes it to

```
ec:/$FSROOT/$EXPVER/log/an/$yyyymmdd.$streamHour.tar.gz
```

Within the python definition, the appropriate logfiles task is added by a call to method `logTask()` from module `inc_fam.py`.

```
def logTask(trigs=None, top_trigs=None, ecfs=False):
    """ Add logfile tasks, where:
        trigs     = trigger for logfiles task
        top_trigs = trigger for calling location
        ecfs      = true if logfiles are written to ECFS
    """
```

Method `logTask()` returns the name of task '`logfiles`' or '`logfiles_ecfs`', which is triggered by argument '`trigs`'. If argument '`top_trigs`' is supplied, then it is added as a trigger to the calling family.

(xiv) Method _endFamily() of class AnalysisSuite()

Method `_endFamily()` of `class AnalysisSuite()` defines a trigger to mark when all the experiment's work has completed and calls task '`flush`' to flush the experiment's MARS data from cache.

(xv) Method _finalize() of class AnalysisSuite()

Method `_finalize()` of `class AnalysisSuite()` defines the '`cancel`' family. It contains task '`wipefdb`' to remove the fields data base from the supercomputer, task '`deletefws`' to remove the experiment's other filesystems from the supercomputer and task '`cancel`' to remove all the remaining experiment workspace and remove the experiment from the ecflow server. In research mode, the output from the cancel task can be found at `/vol/ifs_sms/cancel/${OWNER}_${EXPVER}_cancel...` (it cannot go to the usual experiment log directory, because this is deleted by the cancel task).

3.4 CLASS OBSERVATIONS()

3.4.1 Observation types

Module `inc_obs.py` starts with a set of lists which define all possible observation types:

```
# Conventional data
conventionalObs = ["conv"]

# Surface observations
surfaceObs = ["ims", "synrg"]

# Satellite-derived temperature and wind observations
satTempWindObs = ["satob", "gpsro", "scatt"]
```

```
# Microwave sounders
microwaveSounderObs = ["msu", "amsua", "amsua_allsky", "amsub",
                       "mwts", "mhs",  "mhs_allsky",  "mwhs", "atms",
                       "mwts2", "mwhs2", "atms_allsky"]
........
```

There are also lists of microwave imagers, infrared sounders, hyperspectral infrared sounders, radar observations, derived products and experimental observations. Lists of observations belonging to as-yet-undefined new observation categories could be added here.

The commonest way in which the suite definition is changed is the addition of a new satellite. This is straightforward with the Python definition. First, the name of the new observation type is added to the appropriate list. Then its BUFR subtype is added to dictionary '`bufr_subtypes`', which maps BUFR subtype numbers to observation names. Then its resource requirements for the BUFR to Observation Data Base (ODB) conversion task are added as a sub-dictionary to dictionary `obsdict{}` in method `_bufrToOdb()` of `class Observations()`:

```
obsdict = {
    "conv":        {"SUBTYPES_EXCLUDE":  "notconv",
                    "NPES":               2*ip.NPES_MKCMA},
    "ims":         {"SUBTYPES_INCLUDE":  "ims",
                    "THINFAC":            36},
    "satob":       {"SUBTYPES_INCLUDE":  "satobs",
                    "SENSOR":             -1,
                    "NPES":               2*ip.NPES_MKCMA},
     ..........
     }
```

A few methods in `class Observations()` need non-standard resources for certain observation types - it may help to search for the name of a similar observation type to find where such changes might be needed. A switch for the new observation type, of the form '`L`' followed by the name of the observation type in upper case and with possible values '`True`' and '`False`', should also be added to the set of prepIFS switches. Everything else needed for the processing of the new observation type should then be handled automatically, using the run-time list of observations generated by the constructor method for `class Observations()`.

3.4.2 Constructor method of class Observations()

Method `__init__()` of `class Observations` in module `_inc_obs.py` calls method `_obs_setlists()`, also of `class Observations()`, to construct run-dependent lists of observations. First of all, method `_obs_setlists()` concatenates all the lists of different observation types to make a master list of all possible observation types:

```
obsList = (conventionalObs + surfaceObs + satTempWindObs +
           microwaveSounderObs + microwaveImagerObs +
           infraredSounderObs + hyperSpectralInfraredObs +
           radarObs + derivedProductObs +
           experimentalObs)
```

Then it interrogates the observation switches supplied by prepIFS in module `parameters.py` to build up a run-time list of observation types in class variable `self.obsgrp_list` of `class Observations()`.

If the switch '`LMONITORING`' in module `parameters.py` is set to '`True`', then, for the long window data assimilation stream only, an extra screening step will be run after the main 4D-Var analysis to process the observation types listed in variable '`OBS_MONITORING`' from module `parameters.py`. In this case, a second list of observation types, `self.obs_reduced_list`, is constructed which consists of `self.obsgrp_list` with the observation types from '`OBS_MONITORING`' removed. It is this subset of observations which is processed by the main 4D-Var step and by all the other streams. If '`LMONITORING`' has the value '`False`',

then `self.obs_reduced_list` is an exact copy of `self.obsgrp_list`. With this mechanism, screening statistics can be generated for new observation types in the operational suite without slowing down the critical path through the analysis.

A run-time list of observation types which are processed by COPE is constructed in variable `self.obsgrp_cope_list`. A list of active microwave imagers is constructed in `self.mwimg_instr`. A list of active allsky ODB types is constructed in `self.obsgrp_allsky_list`. A run-time list of satellites which will be processed by the pre-screening is constructed in `self.sat_instr`.

It is these run-time lists of observations which will be used by the methods of `class Observations()` to control the observation processing.

3.4.3 Method obs_fetch() of class Observations()

Method `obs_fetch()` of `class Observations()` is called from method `obs_fam()` of `class AnalysisSuite()` to fetch the observations.

```
def obs_fetch(self, node, fam, streamClass, trig=None)
    """ Define tasks to fetch observations, where:
         node        = position in suite definition tree
         fam         = name of stream / hour family
         streamClass = class for current stream
         trig        = additional trigger for task ens_fetch_fields
    """
```

Class variable `streamClass.obsformatin` is inspected to see whether the observations should be fetched from the MARS BUFR or ODB archives. ODB format data is only used for the '`obstatfc`' configuration. Task '`fetchobs`' is used to fetch BUFR observations, while task '`fetchmarsodb`' fetches ODB data from MARS. In the operational configuration for family '`00lw`', task '`fetchobs_1`' is called to pre-fetch the observations for the following day, in order to minimise delays in the observation fetching process.

Method `obs_fetch()` calls task '`fetchmars`' to fetch field-format observations (e.g. sea surface temperature fields or ensemble data assimilation errors from a different experiment).

Method `streamClass.obs_fetch_fields()` is called to fetch the reference analysis fields for the ensemble data assimilation statistics calculations. For all streams except long-window ensemble data assimilation, `obs_fetch_fields()` is a dummy routine which does nothing. For the long-window ensemble data assimilation stream, method `obs_fetch_fields()` is a dummy routine unless switch '`LEDA_ERRORS_OUT`' from module `parameters.py` has the value '`True`', in which case it calls task '`ens_fetch_fields`'. Family '`make/bins`' must complete before task '`ens_fetch_fields`' runs.

Method `obs_fetch()` from module `aeolus.py` is called to fetch aeolus observations. This will be a dummy routine unless switch '`LAEOLUS`' from module `parameters.py` has the value '`True`'.

3.4.4 Method obs_prepare() of class Observations()

If the input observations are in BUFR format, method `obs_prepare()` of `class Observations()` is called from method `obs_fam()` of `class AnalysisSuite()` to pre-process the observations.

```
def obs_prepare(self, node, fam, streamClass, node=None, trig=None)
    """ Define tasks for observation pre-processing, where:
        fam         = name of stream / hour family
        streamClass = class for current stream
        node        = position in suite definition tree
        trig        = trigger for prepare\_obs family
    """
```

In the Research Department configuration, method `obs_prepare()` is called with the absolute path of the '`bins`' family in variable '`trig`'. At experiment start-up time, fetching the observations from MARS in task '`fetchobs`' can take a significant amount of time, and this can be overlapped with the compilation of libraries and building of binaries in the '`make`' family. The observation pre-processing, below family

'prepare_obs', needs the binaries to be available. In the operational configuration, the stream / hour family is triggered by the 'make' family being complete, so an additional trigger on the 'prepare_obs' family is unnecessary. The 'prepare_obs' family also has to wait for the 'fetchmars' task and either the 'fetchobs' task (for catch-up mode) or the 'get' family (for operational real-time mode) to be complete.

Method obs_prepare() creates family 'prepare_obs', then uses the run-time switches from module parameters.py to decide whether or not to add tasks such as 'preobs', 'pregeos', 'prescat', etc. If any microwave imager instruments are being used, corresponding tasks will be added below family 'premwimg'. If any other satellite instruments are being used, task 'pre1crad_prepare' and the corresponding 'pre1crad_${instr}' tasks will be added. If 'LFULL_IASI' has the value 'True', then family 'pre1crad_iasi' will be added, with tasks to split the IASI data into 16 different sub-sections which can be pre-processed in parallel.

3.4.5 Method makeodb() of class Observations()

Method makeodb() of class Observations() is called by method runner() of class FamAnalysis() from module inc_fam.py. Method runner() is called from method content() of class FamAnalysis(), which is in turn called from method main_fam() of class AnalysisSuite() from module inc_an.py.

Method makeodb() creates family 'makeodb' and adds task 'cleanodb'. If the input observations are in BUFR format, then method _bufrToOdb() of class Observations() is called to construct the ODBs. If the input observations are in ODB2 format, method odb2ToOdb1() of class Observations() is called to construct ODB1 format ODBs. Tasks 'mergeodb_2m', 'mergeodb_snow' and 'mergeodb_sekf' are then added to prepare the ODBs for the surface analysis.

3.4.6 Method _bufrToOdb() of class Observations()

Method _bufrToOdb() of class Observations() is called to construct ODBs from input BUFR data. Dictionary obsdict{} is defined, with a sub-dictionary of attributes for each observation type. Family 'bufr2odb' is created, with a default set of attributes added. A list of observation types which should be converted from BUFR to ODB format is constructed in local variable b2o_list. If switch 'LMONITORING' from module parameters.py has the value 'True', then for the long window data assimilation stream only, this is the full list of active observations, self.obsgrp_list. Otherwise, it contains the reduced list of observation types, self.obs_reduced_list. Radar altimeter data, observation type 'ralt', is already in ODB format, and simulated data, type 'simulobs', has a separate ODB creation task, so these two types are removed from list b2o_list (if they were initially present). A loop is then constructed over the observation types in b2o_list, creating tasks of the form 'b2o_${obstype}' and adding any attributes for the corresponding observation type from the dictionary obs_dict{}.

If surface pressure bias corrections are being calculated and the data assimilation is running in catchup-mode, then task 'b2o_conv' has to wait until task 'update_psbias' from the previous cycle has completed. The corresponding trigger is calculated by method getPsBiasTrig() of class Observations(). For real-time observations, the surface pressure bias correction is applied in task 'preobs'.

Finally, if the experiment is running with simulated observations, task 'simulobs2odb' is added with default status "suspended". The user is expected to edit this task and submit it manually.

3.4.7 Method cope() of class Observations()

Method cope() of class Observations() is called by method runner() of class FamAnalysis() if switch 'LCOPE' from module parameters.py has the value 'True'. Family 'cope' is created. Below this, a loop is constructed over the observation types in self.obsgrp_cope_list, adding tasks of the form 'cope_${obstype}' which are triggered by the corresponding task 'b2o_${obstype}' from family '../makeodb/bufr2odb'. Task 'cope_conv' has to wait for task 'vardata' in the 'an' family to complete, since this is where the radiosonde bias correction files are fetched. In the operational configuration, the last day of the month trigger 'ldotm' from the 'ref' family also has to be satisfied. The 'cope' family also contains family 'odb2odb1', with a loop over self.obsgrp_cope_list creating tasks of the form 'odb_${obstype}' which are triggered by the corresponding 'cope_${obstype}' tasks. Method

obs_prepare() from class Aeolus() in module aeolus.py is also called. This will be a dummy routine unless switch 'LAEOLUS' from module parameters.py has the value 'True'.

3.4.8 Method archive_prepare() of class Observations()

Method archive_prepare() of class Observations() is called by method runner() of class FamFeedBack() from module inc_fam.py. Method runner() is called from method content() of class FamFeedback(), which is in turn called from method lag_fam() of class AnalysisSuite() from module inc_an.py. Method archive_prepare() constructs tasks to convert the ODBs into a suitable form for archiving to MARS. First of all, family 'archive_prepare' is created. Its parent family 'fb' has to wait for the corresponding family 'an' in the 'main' family to complete. Below family 'archive_prepare' is a task of the form 'convert_${obstype}' for each of the observation types in list self.obs_reduced_list (or in list self.obsgrp_list for the long window data assimilation stream if 'LMONITORING' has the value 'True'). These tasks convert the ODBs to a suitable format for archiving to MARS.

3.4.9 Method archive_odb() of class Observations()

Method archive_odb() of class Observations() is called by method lag_fam_arch() of class AnalysisSuite() from module inc_an.py. It constructs tasks to archive the ODBs to MARS. Family 'archive_odb' is created, triggered by family '../fb/archive_prepare' being complete. For each of the observation types in list self.obs_reduced_list (or self.obs_list), a task of form 'archive_${obstype}' is created.

3.4.10 Method obstat() of class Observations()

Method obstat() of class Observations() is called by method runner() of class FamFeedBack() from module inc_fam.py. It constructs tasks to calculate the observation departure statistics. For ensemble data assimilation, obstat is only called for the control and the first 2 members and the latter have to wait until the processing for the control is complete.

First of all, family 'obstat_all' is created. Below this, family 'obstat' is created, triggered by family 'fb/archive_prepare' being complete. Method obstat_run() of class Observations() is called by method obstat() to generate the plotting tasks for the different observation types. Dictionary tuner{} is constructed to define the resources for the different observation types. Switch 'LOBSTAT_ALL' from module parameters.py determines whether all the observation departure statistics will be done in the single task 'obstat_calc' or in a task for each observation type. If 'LOBSTAT_ALL' has the value 'False', then for each of the observation types in list self.obs.reduced_list (or self.obs_list), a task of form 'obstat_${obstype}' is created.

Method obstat() then adds task 'obstat_merge' to family 'obstat_all', triggered by family 'obstat' being complete. If switch 'LGRIDSTAT' from module parameters.py has the value 'True', then method obstat_archive() from class Observations() is called.

3.4.11 Method satimsim() of class Observations()

If switch 'LSATIMSIM' from module parameters.py has the value 'True', then method satimsim() of class Observations() is called by method lag_fam() of class AnalysisSuite() from module inc_an.py to construct the tasks to do the satellite image simulation processing. These tasks use forecast data, which comes either from family 'fc', or from the separate family 'fclong', if a long forecast is required from an hour which is not the cycling forecast hour. First of all, family 'satim' is created. Parameters 'SATIMSTART', 'SATIMEND' and 'SATIMSTEP' from module parameters.py are used to construct a loop over time of families with names of the form 'satim_${hour}', where ${hour} runs from 'SATIMSTART' to 'SATIMEND' in steps of 'SATIMSTEP'. Processing at a given hour is triggered by the corresponding forecast having reached that step or having completed. Below family 'satim_${hour}' are families 'retr' to retrieve the forecast data, 'sat' to do the processing and 'satim_flush' to send the data to ECFS. Below family 'sat' is a loop over families of the form 'sat_${satid}', where ${satid} loops over a list of geostationary satellite identifiers. At the lowest level of each of these families is task 'satimsim', with variable 'SATIM_MODE' determining its behaviour.

3.4.12 Method obtime_run() of class Observations()

If switch 'WEBPLOTS' from module parameters.py has the value 'True' and it is not being called from an early delivery stream, then method obtime_run() of class Observations() is called from method content() of class FamWSJobs() in module inc_fam.py to generate the observation time series plots on the workstation. Family 'obtime' is created, triggered by the corresponding tasks 'obstat_all/obstat_merge' and 'biassave' from the 'lag' family. Below family 'obtime' is task 'get_obtime' to fetch the data. Only on the last cycle of each day, family 'plot' is created, triggered by task 'get_obtime'. Below family 'plot' is a task of form 'obtime_${obstype}' for each of the observation types in list self.obs.reduced_list (or self.obs_list). There are also time-series plots for the variable bias correction and the condition number of the minimization.

(i) Method fcSensObs() of class Observations()

Method fcSensObs() of class Observations() in module inc_obs.py creates family 'fc_sens_obs'. Below this are task 'fc_sens_prepare' and family 'fc_sens_save'. The latter contains tasks of the form 'fc_sens_save_${obstype}', for all ODB observation types. These tasks add the forecast sensitivity to observations information to the ODBs and prepare the ODBs for archiving to MARS.

3.5 STREAM-RELATED CLASSES

Data assimilation can be run with a number of different configurations, or 'streams'. Module inc_stream.py contains a class for each stream, with corresponding stream-specific variables and methods. It also contains class StreamControl(), which determines the run-time configuration of streams, dates and cross-stream dependencies.

3.5.1 Class StreamLWDA()

Class StreamLWDA(), which defines the long-window data assimilation stream, is the base class for all the data assimilation streams. In the operational suite, the long-window data assimilation stream runs with as late an observation cut-off as possible and is used to cycle the information forwards in time.

(i) Constructor method of class StreamLWDA()

At the start of module inc_stream.py is the line:

```
from datetime import datetime, timedelta
```

where 'datetime' is the standard Python module for manipulating dates and times. Method __init__() of class StreamLWDA() starts by using parameters 'INIBEGINDATE', 'INIBEGINHOUR', 'INIENDDATE' and 'LAST_HOUR' from module parameters.py to construct the datetime objects self.startTime and self.endTime, which define the start and end times of the long window data assimilation stream. Note that different streams may have different start and end times. If, for example, an experiment combines a long window data assimilation stream and a separate ensemble data assimilation stream to calculate flow-dependent background errors, then the ensemble will have to start a sufficient number of cycles in advance in order to be able to generate the background errors for the first long window cycle.

The constructor method then defines a number of window-related variables. For example, self.period_4d contains the length in hours of the 4D-Var period for the current stream. Again, it is possible for different streams to have different values for these variables. An experiment could, for example, combine 24-hour 4D-Var long window data assimilation with 6-hour 4D-Var early delivery data assimilation and 12-hour 4D-Var ensemble data assimilation. Class variable self.fchours defines the length in hours of the cycling forecast, set to the length of the 4D-Var window + 6 hours, in order to provide first guess values throughout the total extent of the following cycle.

Next, variables related to the name of the stream and the 4D-Var configuration are defined. The class variables self.lemiskf, self.lcalc_psbias, etc are assigned the values of the corresponding variables in module parameters.py. These are switches which define whether or not the emissivity Kalman filter, surface pressure bias correction, etc will be done at the end of the analysis. By contrast, in the constructor

method for `class StreamED()` (see Subsection 3.5.2.(ii) below), these class variables are set to '`False`', because the corresponding tasks are never executed for the 'early delivery' stream. Sets of ensemble data assimilation variables for the default configuration (`self.endanens = 0`, i.e. no ensemble) are defined. Similarly, sets of forecast obstat variables for the default configuration (`self.obstatfc = False`, i.e. forecast obstat switched off) are defined.

The line:

```
self.makeTime = self.startTime - timedelta(hours=self.period_4d)
```

defines the datetime object `self.makeTime` as the time of the '`make`' family, one cycle before the time of the first cycle. Use of Python's datetime objects and its `timedelta()` method makes manipulation of dates and times straightforward.

Values from module `parameters.py` are used to define a dictionary of forecast lengths in hours at each synoptic hour of the day. Empty dictionaries are defined for various triggers and forecast parameters. Empty dictionaries for the forecast sensitivity to observations configuration are also defined.

(ii) Method active() of class StreamLWDA()

Method `active()` of `class StreamLWDA()` uses the values of parameters from module `parameters.py` to determine whether the long window data assimilation stream will be active in the current run, returning a value of '`True`' or '`False`' as appropriate. For most streams, there is a switch which makes the stream active. Long window data assimilation will be active if none of the other streams are active, or if the experiment combines long window data assimilation and early delivery data assimilation.

(iii) Method set_famlist() of class StreamLWDA()

In Subsection 3.3.2.(v), it was described how, below the '`obs`', '`main`', '`lag`' and '`wsjobs`' families, there is a family for each hour of each stream. The example illustrated in Figure 3.1 has long-window 4D-Var families at 00 UTC and 12 UTC, with names '`lw00`' and '`lw12`', and early delivery families at the same hours, with names '`ed00`' and '`ed12`'. Method `set_famlist()` constructs a list of stream / hour family names for a given stream and stores it in the class variable `self.famlist`. First of all, it compares the dates of the start and end times of the stream. If they are the same, it is a single day experiment, with the first cycle at the hour of the start time and the last cycle at the hour of the end time. If the date of the end time is greater than the date of the start time, then all the cycle hours in a 24-hour period are calculated. Each stream / hour family name consists of the string `self.stream` (eg "`lw`" for long window data assimilation, or "`ed`" for early delivery), followed by a 2-digit hour (eg '`lw00`').

(iv) Method set_streamvars() of class StreamLWDA()

Method `set_streamvars()` defines the dictionary of stream-related variables which will be added to each stream / hour family.

(v) Method set_ymds() of class StreamLWDA()

For each of the stream / hour families in `self.famlist`, method `set_ymds()` constructs dictionary entries containing the start and end dates. Dictionaries `self.startymd[fam]` and `self.endymd[fam]` contain the dates as `datetime` objects, while `self.startYMD[fam]` and `self.endYMD[fam]` contain the dates as integers with `YYYYMMDD` format. If, for example, an experiment has its first cycle at midday, then the start date for the midnight cycle will be one day later than the start date for the midday cycle. Similarly, if the last cycle is at midnight, then the end date for the midnight cycle will be one day later than the end date for the midday cycle.

(vi) Method set_fclengths() of class StreamLWDA()

For each of the stream / hour families in `self.famlist`, method `set_fclengths()` determines how long (in hours) the forecast should be and whether or not a separate long forecast family is required. By default, forecasts are the cycling forecast length, `self.fchours`. A separate long forecast family will be required if a long forecast is requested at the analysis family time, but this is not the last synoptic time

in the 4D-Var window, i.e. `self.delta_fc > 0`. Consider, for example, 12-hour 4D-Var at 12 UTC, with the cycling forecast starting from the 18 UTC analysis. If a long forecast is required from 12 UTC, then a separate long forecast family will be required. But for 6-hour 4D-Var at 12 UTC, a separate long forecast family would not be required because the cycling forecast starts from the 12 UTC analysis.

(vii) Method set_fgParams() of class StreamLWDA()

For each of the stream / hour families in `self.famlist`, method `set_fgParams()` defines a set of parameters that will be needed to generate the first guess triggers.

```
self.reinitialize[fam] = "false"
self.fgStream[fam] = self.stream
self.deltaHourFg[fam] = ip.PERIOD_4D
```

`self.reinitialize[fam] = "false"` means that the first guess will be taken from the previous cycle's forecast of the current stream, `self.stream`. `self.deltaHourFg[fam]` defines the number of hours back to the start of the previous cycle's forecast.

(viii) Method set_prevParams() of class StreamLWDA()

Method `set_prevParams()` defines parameters related to the previous cycle of the current stream and of the stream from which the first guess will be taken. It is called from the constructor method of `class StreamControl()` (see Subsection 3.5.8.(i)) with argument '`allFams`', which contains a list of all the stream / hour families in the experiment. For most of the data assimilation, cycles at different hours of the same stream can run independently. However, at certain points, triggers to the previous cycle are required to ensure that the relevant tasks run in strict time order. The '`clean_cycle`' tasks from the '`lag`' family, for example, need to run in strict time sequence, as do the surface pressure bias correction calculations and the observation time sequence plotting.

For each of the stream / hour families in `self.famlist`, the name of the stream / hour family for the previous cycle is generated and then there is a check to see that the family exists, i.e. that it is contained in the list defined by input argument '`allFams`'. For a single cycle experiment, for example, there is no valid previous cycle. Next, there is a check to see if the previous cycle has the same date as the current cycle, or is one day earlier. Consider, for example, the first guess triggers for long window 12-hour 4D-Var. The midnight analysis takes its first guess from the previous day's midday forecast, so dictionary entry `prevYMD[fam]` is set to "`YMD-1`". If the date parameter '`YMD`' for the midday cycle is greater than or equal to the date parameter for the current cycle, then the previous day's forecast must have completed and the first guess data will be available, so dictionary entry `prevCond[fam]` is set to "`>=`". The midday analysis takes its first guess from the current day's midnight forecast. If the date parameter of the midnight cycle is greater than the date parameter for the midday cycle, then its first guess must be available. In this case, `prevYMD[fam] = "YMD"` and `prevCond[fam] = ">"`.

(ix) Method set_fgTriggers() of class StreamLWDA()

For each of the stream / hour families in `self.famlist`, method `set_fgTriggers()` defines the first guess triggers, using parameters defined in methods `set_fgParams()` and `set_prevParams()`. The first guess will be available if the cycle which generates it has already completed, or if the generating forecast is still running, but has already passed the steps needed by the next cycle's analysis. In order to ensure that the emissivity and VARBC bias correction files have been updated, the previous cycle's analysis must also have completed.

(x) Method set_fgErrTriggers() of class StreamLWDA()

Method `set_fgErrTriggers()` determines whether or not triggers will need to be generated for task '`fetcherr`', which fetches the first guess errors fields. If switch '`LOWNEDA`' in module `parameters.py` has the value '`True`', then the experiment is generating its own ensemble data assimilation first guess errors in a tightly coupled experiment, and task '`fetcherr`' must wait until the previous cycle of the ensemble data assimilation has completed. Otherwise, if switch '`LEDA_ERRORS_IN`' from module `parameters.py` has the value '`True`', then first guess errors will already have been fetched from MARS in task '`fetchmars`' and a

trigger is not required for task '`fetcherr`'. If '`LEDA_ERRORS_IN`' has the value '`False`', then randomization errors are being used. A trigger will need to be defined so that the '`fetcherr`' task waits for the first minimization of the previous cycle.

(xi) Method set_prevTriggers() of class StreamLWDA()

Triggers are generated to ensure that the observation time series plot tasks in family '`wsjobs`' and the '`clean_cycle`' tasks in the '`lag`' family are run in strict time sequence.

(xii) Method set_psbiasTriggers() of class StreamLWDA()

If an experiment is calculating its own surface pressure bias corrections (switch '`LCALC_PSBIAS`' from module `parameters.py` has the value '`True`'), then the next cycle's pre-processing of conventional observations must wait until the surface pressure bias correction computations have completed. For real-time observations, the relevant pre-processing is in task '`preobs`' of the '`obs`' family, while for catchup-mode observations, the corresponding pre-processing is in task '`makeodb/b2o_conv`' of the '`main`' family.

(xiii) Method set_obsRealTimeParams() of class StreamLWDA()

Experiments which are running very close to real time must not run their '`obs`' families until the corresponding operational suite tasks have run to archive the operational observations to MARS. A suite called '`o`' is added to each RD ecFlow server, holding triggers which are synchronised by the real operational suite.

(xiv) Method calc_eda_statistics() of class StreamLWDA()

Method `calc_eda_statistics()` of `class StreamLWDA()` is a dummy routine which does nothing.

(xv) Method calc_jb_statistics() of class StreamLWDA()

Method `calc_jb_statistics()` of `class StreamLWDA()` is a dummy routine which does nothing.

(xvi) Method err_save() of class StreamLWDA()

Method `err_save()` of `class StreamLWDA()` is a dummy routine which does nothing.

(xvii) Method obs_fetch_fields() of class StreamLWDA()

Method `obs_fetch_fields()` of `class StreamLWDA()` is a dummy routine which does nothing.

(xviii) Method fetch() of class StreamLWDA()

Method `fetch()` of `class StreamLWDA()` contains the code to generate the tasks to fetch the initial data in the '`make`' family. It needs to be defined here in module `inc_stream.py` because a different version, with additional looping over members, will be needed for ensemble data assimilation.

(xix) Method run_an() of class StreamLWDA()

Method `run_an()` of `class StreamLWDA()` defines the standard way to call the method which generates the analysis code. It needs to be defined here in module `inc_stream.py` because a different version, with additional looping over members, will be needed for ensemble data assimilation.

(xx) Method run_fc() of class StreamLWDA()

Method `run_fc()` of `class StreamLWDA()` defines the standard way to call the method which generates the forecast code. It needs to be defined here in module `inc_stream.py` because a different version, with additional looping over members, will be needed for ensemble data assimilation.

(xxi) Method get_path() of class StreamLWDA()

Method `get_path()` of `class StreamLWDA()` returns its calling argument unchanged.

3.5.2 Class StreamED(StreamLWDA)

`Class StreamED()`, which is a child of `class StreamLWDA()`, defines those variables and methods of the early delivery stream which differ from those for the long window data assimilation stream.

(i) Constructor method of class StreamED()

Method `__init__()` of class `StreamED()` starts by calling the constructor method of its parent class, `StreamLWDA()`:

```
super(streamED, self).__init()
```

The window-related variables are defined for 6-hour 4D-Var. There are two ways that an early delivery data assimilation cycle can define its first guess. If `self.reinitialize[fam]` has the value '`True`', then it takes its first guess from the forecast from the previous cycle of the reinitializing stream, usually a long window data assimilation. In the current configuration of the operational suite, for example, the early delivery data assimilation at midnight takes its first guess from the forecast from the long window data assimilation at midday on the previous day. If an early delivery data assimilation is also required at 06 UTC, then it will take the first guess from the forecast from the early delivery analysis at midnight. For the 06 UTC cycle, `self.reinitialize["ed06"]` has the value '`false`', meaning that it uses the forecast from its own previous analysis cycle as its first guess. Variable '`ED_PERIOD`' from module `parameters.py` defines the length in hours between early delivery cycles and this is stored in class variable `self.step`. Variable '`REINIPERIOD`' from module `parameters.py` defines the length in hours between cycles at which the early delivery analysis takes its first guess from the forecast from the analysis of its reinitializing stream. Variable '`REINIHOUR`' from module `parameters.py` defines the first hour in the day at which an early delivery data assimilation will take its first guess from the forecast from the analysis of its reinitializing stream.

Next, variables related to the name of the stream and the 4D-Var configuration are defined. Some calculations, such as surface pressure bias correction, radiosonde temperature and humidity bias correction, land emissivity Kalman filter computations and observation time-series plotting, are never done for the early delivery stream and are switched off here. Values from module `parameters.py` are used to define a dictionary of forecast lengths in hours at each synoptic hour of the day.

(ii) Method active() of class StreamED()

The early delivery stream will be active if variable '`IFSMODE`' from module `parameters.py` has the value "`early_delivery`" or if switch '`ED_ONLY`' from module `parameters.py` has the value '`True`'.

(iii) Method set_fgParams() of class StreamED()

For each of the stream / hour families in `self.famlist`, method `set_fgParams()` defines the corresponding entry in three dictionaries, `self.reinitialize{}`, `self.fgStream{}` and `self.deltaHourFg{}`. If early delivery analyses are being run more frequently than the reinitializing stream analyses ('`ED_PERIOD`' < '`PERIOD_4D`') and there is no corresponding analysis at this time for the reinitializing stream analysis, then the first guess parameters are defined to take the first guess from the forecast from the previous early delivery analysis, i.e.

```
self.reinitialize[fam] = "false"
self.fgStream[fam] = self.stream
self.deltaHourFg[fam] = self.period_4d
```

Otherwise, the first guess parameters are defined to take the first guess from the forecast from the previous analysis of the long window data assimilation stream:

```
self.reinitialize[fam] = "true"
self.fgStream[fam] = "lw"
self.deltaHourFg[fam] = ip.PERIOD_4D
```

(iv) Method reinihourOK() of class StreamED()

Method `reinihourOK()` checks that the value of variable '`REINIHOUR`' from module `parameters.py` has been correctly specified. If it is a single day experiment and '`REINIHOUR`' does not lie between the hours of the first and last cycles, then the value '`False`' is returned. Otherwise, the value '`True`' is returned.

(v) Method set_obsRealTimeParams() of class StreamED()

For experiments which are running very close to real time, the '`obs`' family must not run before the corresponding operational suite task has archived the observations to MARS. For the operational early delivery stream, date variable '`YMD`' for the 00 UTC cycle (and the 06 UTC cycle if it existed) is one day behind the data date. For RD experiments, the '`YMD`' date variable is always the same as the data date.

3.5.3 Class StreamELDA(StreamLWDA)

`Class StreamELDA()`, which is a child of `class StreamLWDA()`, defines those variables and methods of the ensemble data assimilation stream which differ from those for the long window data assimilation stream.

(i) Constructor method of class StreamELDA()

Method `__init__()` of `class StreamED()` starts by calling the constructor method of its parent class, `StreamLWDA()`:

```
super(streamELDA, self).__init()
```

If both ensemble data assimilation (EDA) and long window data assimilation are running in the same experiment, then the start time for the ensemble data assimilation has to be advanced by as many cycles as it takes to spin up the background error calculation for the first cycle of the long window data assimilation. Next, stream variables related to the name of the stream and the 4D-Var configuration are defined. Then a set of ensemble-related stream variables are defined. Method `_eda_families()` of `class StreamELDA()` is called to define the lists of EDA families.

(ii) Method active() of class StreamELDA()

The ensemble data assimilation stream will be active if the number of ensemble members, variable '`ENDANENS`' from module `parameters.py`, is greater than zero, as long as it is not an ensemble Kalman filter experiment (for which variable '`LENKF`' from module `parameters.py` has the value '`True`').

(iii) Method _eda_families() of class StreamELDA()

Method `_eda_families()` constructs lists of family names (preceded by a "/") for the ensemble. List `self.endaset[]` contains names of the form:

```
["/control", "/member001", "/member002", ...,
 "/member\${ip.ENDANENS}"]
```

List `self.jbset[]` contains lists of family names (preceded by a "/") for the jb calculation of the form:

```
["/jb_control", "/jb_member001", "/jb_member002", ...,
 "/jb_member${ip.N_BGMEMBERS}"]
```

(iv) Method ini_fetch() of class StreamELDA()

Method `ini_fetch()` of `class StreamELDA()` returns a list of family names (not preceded by a "/") for the initial data '`fetch`' families. If an ensemble is taking its initial data from another ensemble and the original ensemble has at least as many members as the new ensemble, then below the '`fetch`' family, there will be a family for the control and each member, fetching the initial data from the corresponding member of the original ensemble. If an ensemble is taking its initial data from an experiment which is not an ensemble, or the original ensemble had less members than the new ensemble, then there is a single '`control`' family below the '`fetch`' family. In this case, for each ensemble member, task '`inidata_serial`'

writes a copy of the control family's initial data to the fields data base, with the GRIB headers modified for the appropriate member number.

(v) Method set_fgParams() of class StreamELDA()

This method is the same as the corresponding method of its parent `class StreamLWDA()`, except that the number of hours back to the start of the previous cycle's forecast, `self.deltaHourFg[fam]`, is set to the value of variable 'EDA_PERIOD_4D' from module `parameters.py`.

(vi) Method set_fgTriggers() of class StreamELDA()

For each of the stream / hour families in `self.famlist`, for each member of the ensemble, the first guess comes from the forecast at the previous cycle of the same member. A generic trigger is made for all members, containing the string "memberxx". When the trigger is used, "memberxx" is replaced by the family name of the appropriate member. Task '`vardata`', which fetches the first guess, must also wait until the previous cycle's control family analysis is complete, in order to ensure that the emissivity and bias correction files have been updated.

Task '`ens_fetch_fields`' cannot run until the corresponding task at the previous cycle has completed. For each of the stream / hour families in `self.famlist`, dictionary entry `ensFetchTrigger[fam]` tests for the '`ens_fetch_field`' task at the previous cycle being complete.

If the experiment is calibrating its error fields against its own forecasts, task '`ens_fetch_fields`' must also wait until the forecast from the early delivery analysis has reached the end of the analysis window.

(vii) Method set_fgErrTriggers() of class StreamELDA()

For each of the stream / hour families in `self.famlist`, method `set_fgErrTriggers()` calculates the triggers for task '`fetcherr`'. Normally, whenever an ensemble data assimilation is running, the EDA errors will be calculated (parameter 'LEDA_ERRORS_OUT' from module `parameters.py` has the value '`True`'). If the experiment is using its own EDA errors as input ('`LEDA_ERRORS_IN' == True` and `EDAEXPVER == EXPVER`), then task '`fetcherr`' must wait until families '`enda_pp`' and '`jb_calc`' at the previous cycle have completed. If the experiment is taking the EDA errors from a different experiment as input (an unlikely combination), then they will already have been fetched in task '`fetchmars`' and task '`fetcherr`' will need no triggers. If randomization errors are used, then the radiation error code in task '`fetcherr`' must wait until task '`ens_stats_gather`' at the previous cycle has completed.

If EDA errors are not being calculated and randomization first guess errors are being used (`LEDA_ERRORS_OUT == False` and `LEDA_ERRORS_IN == False`), then the '`fetcherr`' task for each member must wait until the first minimization of the control member has completed at the previous cycle. If EDA errors are not being calculated and EDA errors from a different experiment are being read in task '`fetchmars`', then task '`fetcherr`' needs no triggers.

(viii) Method set_psbiasTriggers() of class StreamELDA()

For each of the stream / hour families in `self.famlist`, method `set_psbiasTriggers()` calculates triggers which depend on the surface pressure bias correction calculation. If switch 'LCALC_PSBIAS' from module `parameters.py` has the value '`True`', this is done in task '`update_psbias`' at the end of the control member's analysis at the previous cycle.

(ix) Method calc_eda_statistics() of class StreamELDA()

If switch 'LEDA_ERRORS_OUT' from module `parameters.py` has the value '`True`', then method `calc_eda_statistics()` of `class StreamELDA()` defines family '`enda_pp`' to calculate the ensemble data assimilation statistics. Family '`enda_pp`' is triggered by the forecast family '`fc`' being complete and contains family '`ens_stats_members`' and tasks '`ens_stats_gather`', '`ens_cal_rad`', '`ens_errors_rad`', '`ens_errors`' and, if variable 'LENS_CAL' from module `parameters.py` has the value '`True`', task '`ens_cal`'. Family '`ens_stats_members`' has a control family and a family for each member. These in turn contain task '`ens_stats_mem`', with variable 'MEMBER' containing the appropriate member number.

(x) Method calc_jb_statistics() of class StreamELDA()

If switch 'LEDA_ERRORS_OUT' from module parameters.py has the value 'True', then method calc_jb_statistics() of class StreamELDA() defines family 'jb_stats' to calculate the background error statistics. Family 'jb_stats' is triggered by the forecast family 'fc' being complete and contains families 'fetch_jb_fields' and 'jb_calc'. Family 'fetch_jb_fields' has a control family and ${N_BGMEMBERS} additional families, with variable 'N_BGMEMBERS' defined in module parameters.py. These in turn contain task 'fetch_jb_fields_mem', with variable 'MEMBER' containing the appropriate member number. Family 'jb_calc' is triggered by family 'fetch_jb_fields' being complete. For loop variable 'update' taking values from '0' to (${MXUP_RESOLJB} - 1), with variable 'MXUP_RESOLJB' defined in module parameters.py, family 'uptraj_${update}' is constructed, containing task 'ifsmin', with variable 'uptraj' containing the appropriate update number.

(xi) Method err_save() of class StreamELDA()

If switch 'LEDA_ERRORS_OUT' from module parameters.py has the value 'True', then method err_save() of class StreamELDA() adds task 'eda_err_save' to family 'archive' of the current stream and hour family below the 'lag' family. Task 'eda_err_save' archives the EDA errors to MARS and saves the EDA calibration coefficients and wavelet JBs to ECFS. Method err_save() is called by method lag_fam_arch() of class AnalysisSuite(), which is called in turn by method lag_fam(), also of class AnalysisSuite().

(xii) Method obs_fetch_fields() of class StreamELDA()

If switch 'LEDA_ERRORS_OUT' from module parameters.py has the value 'True', then method obs_fetch_fields() of class StreamELDA() adds task 'ens_fetch_fields' to the current stream and hour family below the 'obs' family. Task 'ens_fetch_fields' fetches the reference analysis which will be used to calibrate the ensemble data assimilation statistics from MARS. Method obs_fetch_fields() is called by method obs_fetch() of class Observations(), which is called in turn by method obs_fam() of class AnalysisSuite().

(xiii) Method fetch() of class StreamELDA()

Method fetch() of class StreamELDA() is called to fetch the initial data for the experiment. It is called by method _initial_data() of class MakeIDATA() from module inc_fam.py. First of all, method ini_fetch(), also of class StreamELDA(), is called to determine how many member families there will be below the 'fetch' family. If the experiment is taking its initial data from another ensemble with at least as many members as the current ensemble, then there will be one 'fetch' family member for each ensemble member. If the experiment is starting from a non-ensemble experiment, or from an ensemble with less members than the current ensemble, there will just be a single 'control' family which will modify the GRIB headers to generate ${ENDANENS} copies of the initial data, where the value of parameter 'ENDANENS' from module parameters.py is the number of ensemble members in the experiment. Method fetch() then loops over the appropriate number of input ensemble members, creating a family for each member and then calling method fetch() of its parent class StreamLWDA() to fetch the initial data, and if necessary, generate extra copies of it.

(xiv) Method run_an() of class StreamELDA()

Method run_an() of class StreamELDA() is called by method content() of classes FamAnalysis() and FamFeedback() from module inc_fam.py. It loops over the appropriate number of members, creating a family for the member and then calling method run_an() of its parent class StreamLWDA().

(xv) Method run_fc() of class StreamELDA()

Method run_fc() of class StreamELDA() is called by method content() of classes FamForecast() and FamLongForecast() and by method _initial_data() of class MakeIDATA(), all from module inc_fam.py. If long forecasts are requested for an ensemble, they will probably not be wanted for all the members. Variable 'ENDANENSFCLONG' from module parameters.py defines how many long forecasts will be required. Method run_fc() loops over the appropriate number of members, creating a family

for the member and then calling method `run_fc()` of its parent `class StreamLWDA()` and setting the forecast length parameter to be a long forecast or a cycling forecast, as appropriate.

(xvi) Method get_path() of class StreamELDA()

Method `get_path()` of `class StreamELDA()` adds the string "`/control`" to its calling argument. It is called by method `content()` of `class FamWSJobs()` from module `inc_fam.py`.

3.5.4 Class StreamSCDA(StreamLWDA)

`StreamSCDA()`, which is a child of `class StreamLWDA()`, defines those variables and methods of the short cut-off stream which differ from those for the long window data assimilation stream.

(i) Constructor method of class StreamSCDA()

The constructor method of `class StreamSCDA()` starts by calling the constructor method of its parent `class StreamLWDA()`. In RD mode, the stream / hour family names start with the string "`scda`", while in the operational suite they end with the string "`bc`". Window-related parameters are configured for 6-hour 4D-Var. The default set up, as in the operational suite, is to only run the short cut-off suite at 06 UTC and 18 UTC. Switches for bias correction calculation, emissivity calculations and monitoring are set to '`False`' by default.

(ii) Method active() of class StreamSCDA()

The short cut-off data assimilation stream will be active if the value of switch '`LSCDA`' from module `parameters.py` is '`True`'.

(iii) Method set_fgParams() of class StreamSCDA()

For each of the stream / hour families in `self.famlist`, method `set_fgParams()` defines the corresponding dictionary entries so as to take the first guess from the early delivery analysis from 6 hours earlier:

```
self.reinitialize[fam] = "true"
self.fgstream[fam] = "da"
self.deltaHourFg[fam] = self.period
```

(iv) Method set_obsRealTimeParams() of class StreamSCDA()

For the short cut-off stream, the date in variable '`YMD`' for the 00 UTC (if it existed) and 06 UTC operational cycles is one day behind the data date. For RD mode, the date in variable '`YMD`' is always the same as the data date. Method `set_obsRealTimeParams()` uses this information to define parameters which are needed in the generation of near real-time triggers for the '`obs`' family.

3.5.5 Class StreamMACC(StreamLWDA)

`StreamMACC()`, which is a child of `class StreamLWDA()`, defines those variables and methods of the MACC data assimilation stream which differ from those for the long window data assimilation stream.

(i) Constructor method of class StreamMACC()

The constructor method of `class StreamMACC()` starts by calling the constructor method of its parent `class StreamLWDA()`. The principal difference between the MACC configuration and the long window configuration is that the MACC configuration defines the main cycling stream as "DA" rather than "LWDA".

(ii) Method active() of class StreamMACC()

The MACC stream is active if the value of switch 'LMACC' from module `parameters.py` is '`True`' and it is not an ensemble data assimilation.

3.5.6 Class StreamMonitorOnly(StreamLWDA)

`StreamMonitorOnly()`, which is a child of `class StreamLWDA()`, defines those variables and methods of the 'monitor-only' data assimilation configuration which differ from those for the long window data assimilation stream. The 'monitor-only' configuration is used to investigate the quality of new observation types. The first guess is taken from another experiment which has already run at the resolution of the current experiment. There is no cycling forecast or surface analysis, no initial data needs to be prepared and only the screening is run for 4D-Var. This is sufficient to generate 'departure from first guess' statistics for the observations.

(i) Constructor method of class StreamMonitorOnly()

The constructor method of `class StreamMonitorOnly()` starts by calling the constructor method of its parent `class StreamLWDA()`. It defines class variables to switch off the surface analysis and the type-999 analysis (which generates '`type=an`' rather than '`type=4v`' analysis fields) and switch off bias correction and emissivity computations. The maximum number of updates to the trajectory, `self.mxup_traj`, is set to '0'. Class variable `self.monitor_only` is set to '`True`'.

(ii) Method active() of class StreamMonitorOnly()

The 'monitor only' stream is active if the value of switch '`MONITOR_ONLY`' from module `parameters.py` is '`True`', but the value of parameter '`LOBSTATFC`' must also be '`False`', i.e. it is not a 'forecast obstat' experiment.

(iii) Method set_streamvars() of class StreamMonitorOnly()

Method `set_streamvars()` of `class StreamMonitorOnly()` defines the variables which will be added to each stream / hour family. In particular, the value of parameter '`MXUP_TRAJ`' is set to '1', because otherwise certain parts of script '`ifstraj`' will not be executed.

(iv) Method set_fgParams() of class StreamMonitorOnly()

Method `set_fgParams()` of `class StreamMonitorOnly()` defines parameters to ensure that the first guess is taken from the experiment defined by the variables '`REINIEXPVER`', '`REINICLASS`' and '`REINISTREAM`' of module `parameters.py`.

(v) Method set_fgTriggers() of class StreamMonitorOnly()

Method `set_fgTriggers()` of `class StreamMonitorOnly()` sets the first guess triggers for each stream / hour family to '`None`', since the reinitialising experiment defined by parameters '`REINIEXPVER`', '`REINICLASS`' and '`REINISTREAM`' has already run and is already available.

(vi) Method set_fgErrTriggers() of class StreamMonitorOnly()

Method `set_fgErrTriggers()` of `class StreamMonitorOnly()` sets the first guess error triggers for each stream / hour family to '`None`', since the reinitialising experiment defined by parameters '`REINIEXPVER`', '`REINICLASS`' and '`REINISTREAM`' has already run and is already available.

(vii) Method run_fc() of class StreamMonitorOnly()

Method `run_fc()` of `class StreamMonitorOnly()` creates a dummy task which has been set to '`complete`' and which will be the contents of the forecast family. The 'monitor-only' configuration does not have a cycling forecast, but this dummy task means that all the triggers which refer to the forecast family being complete do not have to be redefined.

3.5.7 Class StreamObstatfc(StreamMonitorOnly)

`StreamObstatfc()`, which is a child of `class StreamMonitorOnly()`, defines those variables and methods of the 'forecast-obstat' data assimilation configuration which differ from those for the 'monitor-only' stream. The 'forecast-obstat' configuration calculates observation departure statistics for different

forecast ranges of an experiment which has already run. As in the 'monitor-only' configuration, there is no cycling forecast or surface analysis, no initial data needs to be prepared and only the screening is run for 4D-Var. Observations are taken from archived ODBs.

(i) Constructor method of class StreamObstatfc()

The constructor method of `class StreamObstatfc()` starts by calling the constructor method of its parent `class StreamMonitorOnly()`. It defines class variables to fetch the observations from archived ODB data rather than archived BUFR data and sets the configuration to 'obstatfc'.

```
self.obsformatin = "odb2"
self.obstatfc    = True
```

It calls method `_obstatfc_families()` of `class StreamObstatfc()` to define the families for the different obstatfc forecast time ranges.

(ii) Method active() of class StreamObstatfc()

The 'obstatfc' stream is active if the value of switch '`LOBSTATFC`' from module `parameters.py` is '`True`'.

(iii) Method set_fgParams() of class StreamObstatfc()

Unlike its parent `class StreamMonitorOnly()`, `class StreamObstatfc()` sets the 'reinitialize' switches back to 'false'.

(iv) Method _obstatfc_families() of class StreamObstatfc()

Method `_obstatfc_families()` of `class StreamObstatfc()` uses parameters '`FIRST_RANGE`', '`STEP_RANGE`' and '`LAST_RANGE`' from module `parameters.py` to define the list of range families that will be needed by the fcobstat configuration.

3.5.8 Class StreamControl()

The constructor method for `class StreamControl()` is called from the constructor method of `class AnalysisSuite()` (see Subsection 3.3.2.(i)). It is the engine which works out the run-time configuration of streams and dates and times and determines the cross-stream dependencies.

(i) Constructor method of class StreamControl()

Module `inc_stream.py` has a variable '`STREAMS`' which consists of a list of the names of all possible stream classes. The Research Department and Forecast Department configurations have different lists of streams. The constructor method of `class StreamControl()` starts by looping over all possible streams, adding those whose method `active()` returns the value '`True`' to the list of active streams, `self.streamClassList`.

Next, for each active stream, its method `set_famlist()` is called to construct a list of 'stream / hour' family names (see Subsection 3.5.1.(iii)) and these are added to '`allFams`', the list of all possible 'stream / hour' families.

For each active stream, its method `set_ymds()` is called to construct dictionaries, with entries for each stream / hour family, of start and end dates as both Python datetime objects and integer variables of the format '`YYYYMMDD`' (see Subsection 3.5.1.(v)).

For each active stream, its method `set_fclengths()` is called to construct dictionaries, with entries for each stream / hour family, of forecast lengths in hours and possibly also separate long forecast lengths (see Subsection 3.5.1.(vi)).

Before the cross-stream dependencies can be computed, it is necessary to complete the construction of the list '`allFams`'. Python does not permit triggers to non-existent families, so every family in a trigger has to be checked before it can be used.

For each active stream, its method `set_fgParams()` is called to construct dictionaries, with entries for each stream / hour family, of parameters which will be needed for the definition of the first guess triggers (see Subsection 3.5.1.(vii)).

For each active stream, its method `set_prevParams()` is called to construct dictionaries, with entries for each stream / hour family, of parameters related to the previous cycle of the current stream and of the stream from which the first guess will be taken (see Subsection 3.5.1.(viii)).

For each active stream, its method `set_fgTriggers()` is called to construct dictionaries, with entries for each stream / hour family, of first guess triggers (see Subsection 3.5.1.(ix)).

For each active stream, its method `set_fgErrTriggers()` is called to construct dictionaries, with entries for each stream / hour family, of first guess error triggers (see Subsection 3.5.1.(x)).

For each active stream, its method `set_psbiasTriggers()` is called to construct dictionaries, with entries for each stream / hour family, of the surface pressure bias correction triggers for the conventional observation preparation (see Subsection 3.5.1.(xii)).

For each active stream, its method `set_prevTriggers()` is called to construct dictionaries, with entries for each stream / hour family, of triggers to the previous cycle of the current stream which are needed by tasks which need to run in strict sequence order (e.g. the observation time series plotting and the clean-cycle tasks) (see Subsection 3.5.1.(xi)).

For each active stream, its method `set_obsRealTimeParams()` is called to construct dictionaries, with entries for each stream / hour family, of parameters related to triggers for the operational archiving of observations (see Subsection 3.5.1.(xiii)).

3.6 ANALYSIS AND FORECAST CLASSES

Module `inc_fam.py` contains the classes which define the detail of the analysis and the forecast.

3.6.1 Class BaseFam()

`Class BaseFam()` is a parent class to classes `FamAnalysis()`, `FamForecast()` and `FamWSJobs()`. It is used to store data and methods which need to be shared between these families. The dictionary `BaseFam.memo{}` is used to store triggers, node addresses, etc which can be defined at one point in the suite definition and used elsewhere. This is particularly helpful when the RD and FD suite definitions share low-level code but have different higher level calling structures.

Variable `BaseFam.limits_` defines the node address where experiment-wide limits are stored. The list `BaseFam.add_to_logs[]` holds a list of triggers which are to be added to the logfiles task. Method `BaseFam.logs_trigger()` returns the list of triggers which are stored in `BaseFam.add_to_logs[]`.

3.6.2 Class FamAnalysis()

`Class FamAnalysis()`, which is a child of `class BaseFam()`, contains the code which defines the analysis part of a data assimilation suite.

(i) Constructor method of class FamAnalysis()

The constructor method of `class FamAnalysis()` is called from within the constructor method of `class AnalysisSuite()` (see Subsection 3.3.2.(i) above):

```
self.an = ifam.FamAnalysis(self.observations, plim)
```

It passes the address of the experiment-wide limits to the constructor method of its parent `class BaseFam()` and stores the input instance of `class Observations()` in its own class variable, `self.observations`.

(ii) Method content() of class FamAnalysis()

Method `content()` of `class FamAnalysis()` is called for each stream / hour family from within method `main_fam()` of `class AnalysisSuite()` (see Subsection 3.3.2.(ix) above).

```
def content(self, node, fam, streamClass)
    """ Call analysis, where:
        node        = position in suite definition tree
        fam         = name of current stream / hour family
        streamClass = class for current stream
    """
```

Argument '`node`' points to family '`an`' of the current stream / hour family. Triggers are applied to family '`an`' to ensure that the observations are available and the '`lag`' family is not too far behind. For the '`forecast obstat`' configuration only, family '`an`' must also wait until the '`stage`' family at midnight has completed. Limit '`AN`', defined in method `limits()` of `class AnalysisSuite()` (see Subsection 3.3.2.(iii) above), restricts how many tasks below family '`an`' can run simultaneously.

A number of variables are added to family '`an`'. '`NPES_AN`', taken from module `parameters.py`, defines the number of parallel MPI processes for the 4D-Var tasks. '`NPOOLS`', also taken from module `parameters.py`, defines the number of ODB pools. '`REINITIALIZE`', taken from dictionary entry `streamClass.reinitialize[fam]`, defines whether or not the first guess will come from a cycling forecast of the current stream.

Method `runner()` of `class FamAnalysis()` contains the code which defines the analysis. However, the actual call from within method `content()` is:

```
streamClass.run_an(node, fam, self.runner)
```

Method `run_an()` is a wrapper layer which enables the same call to be used for a single analysis or an ensemble of analyses.

(iii) Method runner() of class FamAnalysis()

```
def runner(self, node, fam, member, streamClass)
    """ Define the contents of analysis family 'an', where:
        node        = position in suite definition tree
        fam         = name of current stream / hour family
        member      = ensemble member number
        streamClass = class for current stream
    """
```

For non-ensemble streams, argument '`member`' has the value '0' and argument '`node`' points to family '`an`'. For ensemble streams, argument '`node`' points to the ensemble family, e.g. '`control`', '`member001`', '`member002`', ..., which lies in turn below family '`an`'.

Method `runner()` starts by constructing lists '`an_trig`' and '`ltrig_list`', which hold lists of triggers for 4D_Var and the '`lowres`' task respectively.

Task '`black`', which processes the blacklist, is added to the tree. For an ensemble, only the control member processes the blacklist. For the other members of the ensemble, the '`lowres`' task and the '`4D_Var`' family must wait until the '`black`' task in the '`control`' family has completed. Its absolute path is stored in the dictionary entry `BaseFam.memo["black"]`, from where it can be accessed later in the definition to construct triggers for the blacklist task being complete.

Family '`odb`' is created. For non-ensemble streams or for the control member of an ensemble, method `makeodb()` of `class Observations()` is called to construct the ODBs. If switch '`LCOPE`' from module `parameters.py` has the value '`True`', and the input observations are in BUFR format (rather than coming from archived ODBs), then method `cope()` of `class Observations()` is called to do the COPE processing. Task '`restartodb`' is added to the tree with default status '`complete`'. If 4D-Var needs to be

rerun, then task 'restartodb' should be executed manually beforehand. For non-zero ensemble members, task 'copyodb' is added below family 'odb', triggered by family 'odb' of the control family being complete.

Method getFgTrigs() of class FamAnalysis() is called to fetch the first guess triggers for task 'vardata' and the first guess error triggers for task 'fetcherr'. For the 'forecast obstat' configuration, an extra loop over forecast range families is inserted at this point. These have to run in strict sequence order and include code to fetch and prepare the initial data. Except for the 'forecast obstat' and 'forecast sensitivity to observations' configurations, task 'preCleanFDB' is added to remove any 'type = an' data for the current cycle from the Fields Data Base (FDB). This will only be present if the cycle is being rerun. Over-writing data in the FDB does not always give the expected results, especially if the rewritten fields have a different length to the original fields. It is safer to remove them beforehand.

If switch 'LCHEM' from module parameters.py has the value 'True', then task 'prep_chem' is added to prepare the input data for the active chemistry configuration. It is also added to the list of triggers for task 'vardata'.

Task 'vardata', which fetches the first guess data, is added to the tree. Event 'hires' is added to its list of attributes. This is set when the data required by task 'lowres' is ready. The remaining processing in task 'vardata' can then continue in parallel with task 'lowres'. The list of attributes which are added to the node also includes the line:

```
ic.resources.joblim("vardata")
```

Method joblim() of instance 'resources' of class Resources() (or class ResourcesFsobs() for the 'forecast sensitivity to observations' configuration) from module inc_common.py is executed with argument "vardata". This returns a string of the form Variables(self.joblim["vardata"]). Dictionary self.joblim{} of class Resources() contains another dictionary for each entry which defines values for all the variables which will be required at job submission time. Dictionary self.joblim{} is initialised by the constructor method of class Resources(), taking values from module parameters.py. This is how resolution-dependent resources can be defined by prepIFS and passed into the python suite definition.

Except for the 'forecast obstat' configuration, task 'fetcherr', which fetches the first guess errors, is added to the tree. Its job submission variables are defined by the attributes returned by the method ic.resources.joblim("ens_errors").

Task 'lowres', which interpolates the first guess to the resolution of the 4D-Var inner loops, is added to the tree. Its job submission variables are defined by the attributes returned by the method ic.resources.joblim("lowres").

Method analysis() of class FamAnalysis() is called to construct family '4dvar', which contains the main 4D-Var computation. It returns value 'ready_4dvar', which contains the trigger which defines when the analysis data which will be needed by the following cycling forecast will be available. 'ready_4dvar' is stored in the dictionary entry Basefam.memo[''${member}/an''], where 'member' is the ensemble member number (or '0' for non-ensembles). It is picked up from here by the cycling forecast.

If the current stream has a surface analysis (the 'forecast sensitivity to observations' and the 'monitor-only' configurations, for example, do not have a surface analysis), then method _surfaceAnalysis() of class FamAnalysis() is called to construct family 'surf_anal'. It is triggered by family 'odb', task 'preCleanFDB' and the first trajectory of 4D-Var being complete.

If the current stream has a surface analysis, then task 'forceinv' is added to force the invariant fields back to climatology and 'forceinv' is added to the list of triggers in Basefam.memo[''${member}/an''], which will be picked up by the forecast family to determine when the analysis data will be available to start the next cycling forecast.

For non-ensemble streams or for the control member of an ensemble, method runner_zero() of class FamAnalysis() is called to construct the final section of the analysis family.

For the 'forecast obstat' configuration, task 'gather_screen' is added to the end of the analysis family.

(iv) Method getFgTrigs() of class FamAnalysis()

Method `getFgTrigs()` of `class FamAnalysis()` is called by method `runner()` of `class FamAnalysis()` to construct the first guess and first guess error triggers. First of all, the absolute path of the '`main`' family is determined and stored in local variable '`mainPath`'. The first guess and first guess error triggers for the current stream / hour family are stored in local variables '`var_trig`' and '`err_trig`' respectively. Both these variables contain the generic string "`<main>`". This is replaced by the actual value from variable '`mainPath`'. For ensembles, the first guess trigger will also contain the generic string "`/memberxx`". This is replaced by the name of the current ensemble member family, e.g. "`/control`", "`/member001`", etc.

(v) Method analysis() of class FamAnalysis()

Method `analysis()` of `class FamAnalysis()` is called by method `runner()` of `class FamAnalysis()` to construct the 4D-Var analysis.

```
def analysis(self, node, streamClass, member, an_trig)
    """ Define the 4D-Var analysis, where:
        node        = position in suite definition tree
        streamClass = class for current stream
        member      = ensemble member number
        an_trig     = trigger for family '4dvar'
    """
```

First of all, family '`4dvar`' is created. Its triggers are defined by the method's input parameter '`an_trig`' and the job submission variables of its child tasks are defined by the attributes returned by the method `ic.resources.joblim("an")`. The first task in family '`4dvar`', the first trajectory, will run with normal priority. The second task, the first minimization, runs with (normal + 20) priority and the priority for all subsequent tasks is incremented by 1 for each task. The main 4D-Var calculation needs a lot of temporary disk space, which is shared by the different tasks of family '`4dvar`', but which can be released when the family completes successfully. Accordingly, once the first task of family '`4dvar`' has started, the aim is to run the remaining tasks as quickly as possible and then free up their resources, rather than start the '`4dvar`' family for a different experiment.

The number of updates of the trajectory is taken from class variable '`streamClass.mxup_traj`' and stored in local variable '`mxup_traj`'. This is how, within the same experiment, the early delivery analysis can be run with a different number of updates to the trajectory compared with the cycling long window data assimilation. There is then a loop over updates of the trajectory, creating families with names of the form '`uptraj_${update}`', where `${update}` is the trajectory update number, taking values from '0' to '`mxup_traj`' (or '0' to '`mxup_traj - 1`' for the 'forecast sensitivity to observations' configuration). The different updates to the trajectory are triggered to run in strict sequence order.

All trajectory update families contain task '`ifstraj`'. The final update to the trajectory cannot run until the surface analysis has completed. Event '`finalwave`' is added to the list of attributes of task '`ifstraj`' in the final trajectory.

Except for the final trajectory update family, task '`ifstsave`' is added to save the high resolution trajectory data, triggered by task '`ifstraj`' being complete.

Except for the final trajectory update family, task '`ifsmin`' is added to do the minimization. It is triggered by task '`ifstraj`' being complete.

If, as well as writing '`type=4v`' analyses to the fields data base, the current stream also generates '`type=an`' analyses at synoptic hours (the 'forecast sensitivity to observations' and the 'monitor-only' configurations, for example, do not do this), then method `analysis_999()` of `class FamAnalysis()` is called to generate the '`type=an`' analyses.

(vi) Method analysis_999() of class FamAnalysis()

Method `analysis_999()` of `class FamAnalysis()` is called by method `analysis()` of `class FamAnalysis()` to generate the '`type=an`' analyses at synoptic hours and write them to the Fields Data

Base (FDB). Task '`ifstraj_999`' is added to family '`4dvar`', triggered by the penultimate trajectory update family being complete. The value of the trajectory update number is set to '999' and the value of variable '`NPES`' is set to the value of parameter '`NPES_999`' from module `parameters.py`.

Task '`restart_999`' is also added to family '`4dvar`', with default status '`complete`'. It can be run manually if task '`ifstraj_999`' fails due to an instability in the tangent-linear model. In this case, '`type=4v`' fields from the final trajectory can be extracted from the FDB, their GRIB headers can be modified to produce '`type=an`' fields, which can then be written to the FDB. It is triggered by event '`finalwave`' of the final trajectory, which is set when all the '`type=4v`' data has been written to the FDB.

(vii) Method _surfaceAnalysis() of class FamAnalysis()

Method `_surfaceAnalysis()` of `class FamAnalysis()` is called by method `runner()` of `class FamAnalysis()` to construct the surface analysis. First of all, family '`surf_anal`' is created. Default values for the job submission variables of its child tasks are defined by the attributes returned by the method `ic.resources.joblim("surf")`. Node variable '`FSFAMILY`' is set to "`surf_anal`", used in the definition of the directory path of the workspace for the surface analysis.

Task '`t2ana`' is added to family '`surf_anal`' to do the 2 metre temperature analysis. Variable '`mode`' is added to its attributes with value "`t2m`".

Task '`rh2ana`' is added to family '`surf_anal`' to do the 2 metre relative humidity analysis. It is triggered by task `t2ana`' being complete. Variable '`mode`' is added to its attributes with value "`rh2m`".

Task '`snow`' is added to family '`surf_anal`' to do the snow analysis. It is triggered by task '`rh2ana`' being complete. Variable '`mode`' is added to its attributes with value "`snow`" and the job submission variables returned by method `ic.resources.joblim("snow")` are added to its list of attributes.

Task '`sst`' is added to family '`surf_anal`' to do the sea surface temperature analysis.

If switch '`LSEKF`' from module `parameters.py` has the value '`True`', tasks are added for the surface extended Kalman filter. If switch '`LUSE_SMOS`' from module `parameters.py` has the value '`True`' and this is the control member of an ensemble, or it is not an ensemble, then task '`presatsekf`' is added to pre-process the SMOS data. Job submission variables returned by method `ic.resources.joblim("presatsekf")` are added to its list of attributes. Task '`sekf`' is added to do the surface extended Kalman filter. It is triggered by the blacklist task and the 2 metre relative humidity analysis being complete and, if it exists, task '`presatsekf`' being complete. Job submission variables returned by method `ic.resources.joblim("sekf")` are added to its list of attributes.

Task '`slwet`' is added to family '`surf_anal`' to do the soil moisture analysis. It is triggered the sea surface temperature and snow analyses being complete and, if it exists, the surface extended Kalman filter being complete. Variable '`MEM`' is added to define extra memory needed by this large serial job.

(viii) Method runner_zero() of class FamAnalysis()

Method `runner_zero()` of `class FamAnalysis()` is called by method `runner()` of `class FamAnalysis()` to define the final section of the analysis family. For ensemble data assimilation, it is only called for the control member.

```
def runner_zero(self, fan, streamClass, ready_4dvar)
    """ Define the final section of the analysis, where:
        fan         = node address of family 'an'
        streamClass = class for current stream
        ready_4dvar = event trigger which is set when the
                      final trajectory data is available
    """
```

If class variable `streamClass.lemiskf` has the value '`True`' (see Subsection 3.5.1.(i) above), then task '`emiskf`' is added to family '`an`' to compute the land microwave emissivity Kalman filter.

If class variable `streamClass.lcalc_psbias` has the value '`True`', then task '`update_psbias`' is added to family '`an`' to compute the surface pressure bias correction, triggered by the final trajectory data being available.

If class variable `streamClass.lcalc_rstrhbias` has the value '`True`', then task '`update_rstrhbias`' is added to family '`an`' to compute the radiosonde temperature and relative humidity bias correction, triggered by the final trajectory data being available.

If class variable `streamClass.lmonitoring` has the value '`True`', then task '`monitoring`' is added to family '`an`' to monitor passive data, triggered by family "`4dvar`" being complete. Job submission variables returned by method `ic.resources.joblim("an")` are added to its list of attributes.

Method `l2c()` from module `aeolus.py` is called to do the Aeolus Level 2C processing. This will be a dummy routine unless switch `LAEOLUS` from module `parameters.py` has the value '`True`'.

(ix) Method archiver() of class FamAnalysis()

Method `archiver()` of `class FamAnalysis()` is called by method `lag_fam_arch()` of `class AnalysisSuite()` (see Subsection 3.3.2.(xi) above) to archive the analysis data and the cycling forecast data from the Fields Data Base (FDB) into the Mars archive. It is also called by method `archive()` of `class MakeIDATA()` to archive the initial data (see Subsection 3.3.2.(iv)). Method `archiver()` adds tasks '`anml`' for the model level data, '`anpl`' for the pressure level data, '`ansfc`' for the surface data and optionally '`anwave`' for the wave data and '`anil`' for data on isentropic surfaces. Each task is triggered by event '`fdb`' of the previous task (if there is one).

3.6.3 Class FamForecast()

`Class FamForecast()`, which is a child of `class BaseFam()`, contains the code which defines the forecast part of a data assimilation suite.

(i) Constructor method of class FamForecast()

The constructor method of `class FamForecast()` is called from within the constructor method of `class AnalysisSuite()` (see Subsection 3.3.2.(i) above):

```
self.fc = ifam.FamForecast(plim)
```

It passes the address of the experiment-wide limits to the constructor method of its parent `class BaseFam()`.

(ii) Method content() of class FamForecast()

Method `content()` of `class FamForecast()` is called for each stream / hour family from within `method main_fam()` of `class AnalysisSuite()` (see Subsection 3.3.2.(ix) above).

```
def content(self, node, fam, streamClass)
    """ Define resources and run forecast, where:
        node        = position in suite definition tree
        fam         = name of current stream / hour family
        streamClass = class for current stream
    """
```

Argument '`node`' points to family '`fc`' of the current stream / hour family.

First of all, the length in hours of the forecast, defined by `streamClass.fclength[fam]`, is compared with the length in hours of a cycling forecast, `streamClass.fchours`. If the forecast is longer than a cycling forecast, then local variable '`longfc`' is set to '`True`' and the forecast is assigned resources appropriate for a long forecast, rather than a short cycling forecast. Class variable `streamClass.fclength[fam]` was calculated by method `streamClass.set_fclengths()` as described in Subsection 3.5.1.(vi) above. Class variable `streamClass.fchours` was set by the constructor method of the relevant streamClass.

Limit 'FC' (or limit 'FCLONG' for a long forecast), as defined in method limits() of class AnalysisSuite() (see Subsection 3.3.2.(iii) above), restricts how many tasks below family 'fc' may run simultaneously.

Variables 'HOUR', set to streamClass.fchr[fam] and 'DELTA_DAY', set to streamClass.fcDeltaDay[fam], are added as attributes to family 'fc'. These 2 class variables were also set by method streamClass.set_fclengths(). Variable 'FSFAMILY' is set to "fc", to be used in the definition of the directory path of the workspace for the forecast family.

Method runner() of class FamForecast() contains the code which defines the forecast family. However, the actual call from within method content() is:

```
streamClass.run_fc(node, streamClass.fclength[fam], self.runner, longfc=longfc)
```

Method run_fc() is a wrapper layer which enables the same call to be used for a single forecast or an ensemble of forecasts.

Method run_fc() is also called from within method _initial_data() of class MakeIDATA() from module inc_fam.py to do the short cycling forecast step while preparing the initial data at the start of the data assimilation (see Subsection 3.3.2.(iv) above).

(iii) Method runner() of class FamForecast()

```
def runner(self, node, fclength, member, streamClass, longfc = False)
    """ Fetch the initial data and run a forecast, where:
        node        = position in suite definition tree
        fclength    = length in hours of forecast
        member      = ensemble member number
        streamClass = class for current stream
        longfc      = True for long forecast
    """
```

For non-ensemble streams, argument 'member' has the value '0' and argument 'node' points to family 'fc'. For ensemble streams, argument 'node' points to the ensemble family, e.g. 'control', 'member001', 'member002', ..., which lies in turn below family 'fc'.

Method runner() starts by adding variable 'FCLENGTH' to the input argument 'node'. For an ensemble, only the first 'ENDANENSFCLONG' members (where the value of 'ENDANENSFCLONG' is taken from module parameters.py) have long forecasts. The remaining members of the ensemble have short cycling forecasts. This logic is executed by method run_fc() of the relevant streamClass, as described in Subsection 3.5.3.(xv) above.

If method runner() is called from within the 'make' family, or within the 'initial' family of the 'forecast obstat' configuration, then no triggers need to be added to the input argument 'node'. If it is called from within a cycling data assimilation, then the trigger which determines when the analysis data is available to be used as initial data for the forecast has been stored in Basefam.memo["${member}/an"] (see Subsection 3.6.2.(iii) above).

If switch 'LCHEM' from module parameters.py has the value 'True', then task 'prep_chem' is added to prepare the input data for the active chemistry configuration. It is also added to the list of triggers for task 'getfcdata'.

Task 'getfcdata', which fetches the input analysis data, is added to the tree. Then task 'getpersSST', which fetches the persistence sea surface temperature data, is added, triggered by task 'getfcdata' being complete. Its job submission variables are defined by the attributes returned by the method ic.resources.joblim("persSST").

Finally, task 'model' is added to run the forecast, triggered by task 'getpersSST' being complete. If it is a long forecast, its job submission variables are defined by the attributes returned by the method ic.resources.joblim("fclong"). For a short cycling forecast, the job submission variables are defined

by the attributes returned by the method `ic.resources.joblim("fc")`. A meter is added to the task, with name "`step`" and values running from '-1' to '`fclength`'.

(iv) Method archiver() of class FamFamForecast()

Method `archiver()` of `class FamForecast()` is called by method `lag_fam_arch()` of `class AnalysisSuite()` (see Subsection 3.3.2.(xi) above) to archive long forecast data from the Fields Data Base (FDB) into the Mars archive. The short cycling forecast data is archived by method `archiver()` of `class FamAnalysis()`, see Subsection 3.6.2.(ix) above. Method `archiver()` optionally adds task '`ml`' for the model level data if switch '`LRMLP`' from module `parameters.py` has the value '`True`'; task '`pl`' for the pressure level data if switch '`LRPLP`' from module `parameters.py` has the value '`True`'; task '`sfc`' for the surface data if switch '`LRSUP`' from module `parameters.py` has the value '`True`'; task '`pt`' for the data on potential temperature levels if switch '`LRPTP`' from module `parameters.py` has the value '`True`'; task '`pv`' for the data on potential vorticity levels if switch '`LRPVP`' from module `parameters.py` has the value '`True`' and task '`hl`' for the data on height levels if switch '`LRHLP`' from module `parameters.py` has the value '`True`'. Each task is triggered by event '`fdb`' of the previous task (if there is one).

3.6.4 Class FamLongForecast(FamForecast)

`Class FamLongForecast()`, which is a child of `class FamForecast()`, contains the code to define a separate long forecast family.

(i) Constructor method of class FamForecast()

The constructor method of `class FamLongForecast()` is called from within the constructor method of `class AnalysisSuite()` (see Subsection 3.3.2.(i) above):

```
self.longfc = ifam.FamLongForecast(plim)
```

It calls the constructor method of its parent `class FamForecast()`.

(ii) Method content() of class FamLongForecast()

Method `content()` of `class FamLongForecast()` is called from within method `main_fam()` of `class AnalysisSuite()` if `streamClass.longfclength[fam] > 0`, where '`fam`' is the name of the stream / hour family. A separate long forecast family will be required if a long forecast is requested at a synoptic hour which is not the hour within the 4D-Var window of the cycling forecast. In this case, the stream / hour family will contain family '`an`' for the analysis, family '`fc`' for the short cycling forecast and family '`longfc`' for the separate long forecast.

Limit '`FCLONG`', as defined in method `limits()` of `class AnalysisSuite()` (see Subsection 3.3.2.(iii) above), restricts how many tasks below family '`longfc`' may run simultaneously.

Method `runner()` of parent `class FamForecast()` contains the code which defines the forecast family. The call from within method `content()` of `class FamLongForecast()` is:

```
streamClass.run_fc(node, streamClass.longfclength[fam], self.runner, longfc=True)
```

Method `run_fc()` is a wrapper layer which enables the same call to be used for a single forecast or an ensemble of forecasts.

3.6.5 Class Make()

`Class Make()`, which is a child of `class FamAnalysis()`, contains the code which defines the start-up part of a data assimilation suite. It defines tasks to prepare experiment file systems, compile libraries and build binaries. It has a child class, `class MakeIDATA(Make)` (see Subsection 3.6.6), which does all of the above, but also prepares the initial data.

(i) Constructor method of class Make()

The constructor method of `class Make()` is called from within the constructor method of `class AnalysisNoFcSuite()`, or from within the constructor method of `class MakeIDATA()`, which in turn is called from within the constructor method of `class AnalysisSuite()` (see Subsection 3.3.2.(i) and Subsection 3.3.2.(iv) above):

```
self.maker = ifam.Make(self.limits_)
```

It starts by calling the constructor method of its parent `class FamAnalysis()`. It then creates an instance of `class Library()` from module `inc_libs.py` and stores it in its class variable `self.libraries`.

```
self.libraries = il.Library()
```

The '`make`' family needs an associated stream, for example for defining the path of the work directories or for defining the kind of initial data which will need to be prepared. For most configurations, this will be the 'long window data assimilation' stream, `StreamLWDA()` from module `inc_stream.py`. The exceptions are MACC experiments, which need `StreamMACC()`, and ensemble data assimilation experiments, which need `StreamELDA()`.

```
self.makeStream = ist.StreamLWDA()
```

The time of the '`make`' family needs to be set to one cycle before the first cycle time. This is taken from the stream family's variable '`makeTime`' and stored in the `Make()` class variable `self.makeFamilyTime`. For an experiment which combined both ensemble data assimilation to generate first-guess errors and deterministic data assimilation, the ensemble data assimilation would need to start a sufficient number of cycles in advance to be able to generate the first guess errors for the first deterministic cycle, so its '`make`' family would have an earlier time than the time of the deterministic assimilation's '`make`' family.

```
self.makeFamilyTime = self.makeStream.makeTime
```

An empty list is set up in class variable '`self.main_trig`' to hold a list of triggers from the '`make`' family which must be satisfied before the '`main`' family can run.

(ii) Method content() of class Make()

Method `content()` of `class Make()` is called by method `make()` of `class AnalysisSuite()` (see Subsection 3.3.2.(iv) above). It starts by calling method `set_streamvars()` of `class self.makeStream` to add stream-related variables to the '`make`' family node. It then calls method `_tsetup()` of `class Make()` to setup up the experiment's filesystems on the workstation and the supercomputer.

Method `make_libs()` of `class Library()` is called to build the libraries on the supercomputer.

Method `get_aeolus()` from module `aeolus.py` is called to fetch the aeolus perforce libraries and scripts if switch '`LAEOLUS`' from module `parameters.py` has the value '`True`'. Otherwise, it is a dummy routine which does nothing.

Tasks '`links`', '`rsbias_datalinks`' and '`datalinks`' are added to the '`make`' family to set up symbolic links to various data files.

If switch '`LCOPE`' from module `parameters.py` has the value '`True`', then task '`libcope`' is added to family '`make`' to prepare the COPE libraries on the supercomputer.

If switch '`LOOPS`' from module `parameters.py` has the value '`True`', then method `make_oopsLibs()` of `class Library()` is called to build the OOPS libraries on the supercomputer.

Method `build_bins()` of `class Library()` is called to build binaries on the supercomputer.

If switch '`BUILDWS`' from module `parameters.py` has the value '`True`', then method `ws_build()` of `class Library()` is called to prepare the libraries and build the binaries on the workstation.

If switch '`WAVE`' from module `parameters.py` has the value '`True`', then task '`wconst`' is added to family '`make`' to prepare the wave analysis constant files.

The list of triggers from the 'make' family to the 'main' family which is stored in self.main_trig is added to at various steps of method content(). Similarly, the list of triggers to the 'logfiles' task, which is stored in self.add_to_logs, is also added to at various steps of method content().

3.6.6 Class MakeIDATA(Make)

Class MakeIDATA(), which is a child of class Make(), contains the code to prepare the initial data for a data assimilation suite.

(i) Constructor method of class Make()

The constructor method of class MakeIDATA() is called from within the constructor method of class AnalysisSuite() (see Subsection 3.3.2.(i) and Subsection 3.3.2.(iv) above):

```
self.maker = ifam.MakeIDATA(plim)
```

It calls the constructor method of its parent class Make().

(ii) Method content() of class MakeIDATA()

Method content() of class MakeIDATA() is called by method make() of class AnalysisSuite() (see Subsection 3.3.2.(iv) above). It starts by calling method content() of its parent class Make(). If switch 'MAKE_IDATA' from module parameters.py has the value 'False', the code exits from method content() at this point without preparing the initial data. Otherwise, a list of triggers is constructed in variable 'trig_list' of tasks which must complete before the initial data can be prepared.

For long window 4D-Var, if there are overlapping windows, then the initial data will need to be prepared at more than one time-step. With no overlap, only a single set of initial data is required. A loop is set up over initial data steps, creating families with names of the form 'inifam_1', 'inifam_2', etc as required. Method _initial_data() of class MakeIDATA() is called from within the loop to prepare the initial data at a single time-step.

If an experiment contains both ensemble data assimilation and deterministic data assimilation, then separate initial data families are also required for the ensemble data assimilation.

(iii) Method _initial_data() of class MakeIDATA()

Method _initial_data() of class MakeIDATA() is called by method content() of class MakeIDATA().

```
def _initial_data(self, node, makeStream)
    """ Fetch initial data and run short forecast, where:
        node       = position in suite definition tree
        makeStream = class for make family stream
    """
```

First of all, family 'initial', or 'initial_eda' for an ensemble, is created. Method fetch() of class makeStream is called to create family 'fetch' and prepare the initial data. If 'makeStream' is an instance of class StreamLWDA(), then method 'fetch' gets a single set of initial data, prepares it at the experiment resolution and sets the GRIB headers appropriately (see Subsection 3.5.1.(xviii)). If 'makeStream' is an instance of class StreamELDA(), then below family 'fetch' is a family for each input member, preparing the corresponding initial data (see Subsection 3.5.3.(xiii)).

Family 'fc' is created and method run_fc() of class makeStream is called to run a short cycling forecast. If 'makeStream' is an instance of class StreamLWDA(), then method 'run_fc' fetches the initial data and runs a single short forecast (see Subsection 3.5.1.(xx)). If 'makeStream' is an instance of class StreamELDA(), then below family 'fc' is a family for each ensemble member, running the corresponding short forecast (see Subsection 3.5.3.(xv)).

Method archive() of class MakeIDATA() is then called to archive the initial data to MARS.

3.7 LAG AND WSJOBS FAMILY CLASSES

Module `inc_fam.py` also contains the classes which define the detail of the processing which is done after the analysis and forecast families have completed. The '`lag`' family contains tasks which run on the supercomputer, archiving data to MARS and ECFS and calculating observation departure statistics. The '`wsjobs`' family contains tasks which run on the workstation, producing plots and updating web pages.

3.7.1 Class FamFeedBack(FamAnalysis)

`Class FamFeedback()` is a child of `class FamAnalysis()`. Its constructor method is called from within the constructor method of `class AnalysisSuite()` (see Subsection 3.3.2.(i) above).

```
self.fb = ifam.FamFeedBack(self.observations, plim)
```

It calls the constructor method of its parent class, `FamAnalysis()` (see Subsection 3.6.2.(i) above).

(i) Method content() of class FamFeedback()

Method `content()` of `class FamFeedBack()` is called for each stream / hour family from within method `lag_fam()` of `class AnalysisSuite()` (see Subsection 3.3.2.(x) above).

```
def content(self, node, fam, streamClass)
    """ Define contents of family 'fb', where:
        node        = position in suite definition tree
        fam         = name of current stream / hour family
        streamClass = class for current stream
    """
```

Argument '`node`' points to family '`fb`' of the current stream / hour family. Triggers are applied to family '`fb`' to ensure that family '`an`' of the corresponding stream / hour family in the '`main`' family has completed.

Method `runner()` of `class FamFeedback()` contains the code which defines the content of family '`fb`'. However, the actual call from within method content() is:

```
streamClass.run_an(node, fam, runner=self.runner, kind="obs")
```

Method `run_an()` is the wrapper layer which enables the same call to be used for a single analysis or an ensemble of analyses. Argument '`kind`' determines the number of members for an ensemble, which will be the value of parameter '`ENDANENSODB`' from module `parameters.py`.

(ii) Method runner() of class FamFeedBack()

```
def runner(self, node, fam, member, streamClass)
    """ Define the contents of analysis family 'an', where:
        node        = position in suite definition tree
        fam         = name of current stream / hour family
        member      = ensemble member number
        streamClass = class for current stream
    """
```

For non-ensemble streams, argument '`member`' has the value '0' and argument '`node`' points to family '`fb`'. For ensemble streams, argument '`node`' points to the ensemble family, e.g. '`control`', '`member001`', '`member002`', ..., which lies in turn below family '`fb`'.

Method `runner()` starts by adding task '`fdbksave`' to save files to ECFS. It is constrained by limit '`ecfs`', as defined in method `limits()` of `class AnalysisSuite()` (see Subsection 3.3.2.(iii) above), which restricts how many tasks may write to ECFS simultaneously. If the ECFS service is unavailable, this limit can be set to '0' at the top experiment family node. Its job submission priority is defined by the attributes returned by the method `ic.resources.priority("store")`. Historically, jobs which fetched from or stored to ECFS or MARS were run at a lower priority. This is not working well on the Cray, so fetch and store priority has been reset to normal priority.

Task 'biassave', which saves bias files to ECFS, is added and method archive_prepare() of class Observations() is called to convert the ODBs into a form which is suitable for archiving to MARS (see Subsection 3.4.8 above) (these 2 steps are not needed for non-zero ensemble members).

Method obstat() of class Observations() is called to prepare the observation departure statistics. For ensembles, this is only done for the control and up to 2 members.

If switch 'WAVE' from module parameters.py has the value 'True', task 'wavesave' is added to save wave files to ECFS.

If switch 'LENKF' from module parameters.py has the value 'True', task 'enkf_ecfs' is added to save ensemble Kalman filter files to ECFS.

Method archive() from module aeolus.py is called to archive aeolus data. This will be a dummy routine unless switch LAEOLUS from module parameters.py has the value "True".

3.7.2 Class FamWSJobs(BaseFam)

Class FamWSJobs() is a child of class BaseFam(). It contains the definition of family 'wsjobs', which holds the jobs which run on the workstation. Its constructor method is called from within the constructor method of class AnalysisSuite() (see Subsection 3.3.2.(i) above).

```
self.ws = ifam.FamWSJobs(self.observations, plim)
```

It passes the address of the experiment-wide limits to the constructor method of its parent class BaseFam() and stores the input instance of class Observations() in its own class variable, self.observations.

(i) Method content() of class FamWSJobs()

Method content() of class FamWSJobs() is called for each stream / hour family from within method wsjobs() of class AnalysisSuite() (see Subsection 3.3.2.(xii) above).

```
def content(self, node, fam, streamClass)
    """ Define workstation jobs, where:
        node        = position in suite definition tree
        fam         = name of current stream / hour family
        streamClass = class for current stream
    """
```

Argument 'node' points to the current stream / hour family below family 'wsjobs'. It is triggered by the corresponding stream / hour family in the 'lag' family being complete. Family 'buildws' in the 'make' family, which prepares libraries and builds binaries on the workstation, must also have completed successfully. Method onws() of class Resources() from module inc_common.py adds attributes to family 'node' which define resources required for running jobs on the workstation. It defines appropriate values for variables 'ECF_JOB_CMD', 'ECF_KILL_CMD', 'QUEUE' and 'LOGDIR'. The length of the forecast is added as variable 'FCLENGTH' to family 'node'. This has the value of streamClass.fclength[fam] unless there is a separate long forecast at this cycle, in which case 'FCLENGTH' has the value of streamClass.longfclength[fam].

If switch 'PLOT_INC' from module parameters.py has the value 'True' (for field increment plots) or switch 'PLOT_RMS' from module parameters.py has the value 'True' (for observation rms plots) or streamClass.obtime has the value 'True' (for observation time series plots), then family 'plot' is created. The 'plot' families for a given stream have to run in strict time sequence order. The corresponding trigger is taken from streamClass.WsTrigger[fam] and added to the 'plot' family. If analysis increment plots are requested, then task 'pincr' is added to family 'plot', triggered by the corresponding family 'an' from the 'main' family being complete.

If observation rms plots are requested, then task 'pobstat' is added to family 'plot', triggered by the corresponding task 'obstat_merge' from the 'lag' family being complete. By default, variable 'PLOT_RMS' has the value 'False', because obstat statistics produced by the offline package 'dobstat' from a number

of cycles are more useful. Plotting a single cycle at this point is useful for testing new versions of the graphics software. The resulting plot files are left on disk for 1 day and then deleted.

If observation time series plots are requested, method `obtime_run()` of `class Observations()` is called to add family '`obtime`' and its member tasks, as described in Section 4.12 above.

If switch '`SCOR`' from module `parameters.py` has the value '`True`' and there is a long forecast for the current stream / hour family and the cycle time is midnight or midday, then family '`verify`' is created, containing tasks '`verify`' and '`wamverify`', to calculate the forecast verify statistics. Family '`verify`' is triggered by the corresponding '`archive`' family from the '`lag`' family being complete.

If observation time series plots are requested and this is the last cycle in the day, then family '`web`' with task '`web_update`' is added, triggered by family '`plot`' being complete.

Task '`clean_ws`' cleans up the workstation disk space, triggered by the '`plot`' and '`verify`' families being complete, if they exist for this cycle. Finally, task '`logfiles_ecfs`' produces tar files from the logfiles and writes them to ECFS.

3.8 CLASS LIBRARY()

Module `inc_libs` contains `class Library()`, which has methods for compiling libraries and building binaries on different platforms.

3.8.1 Method make_libs() of class Library()

For the supercomputer, method `make_libs()` of `class Library()` is called from method `content()` of `class Make()` (see Subsection 3.6.5.(ii)). For the workstation libraries, method `make_libs()` is called from method `ws_build()` of `class Library()`, which is in turn called from method `content()` of `class Make()`. It starts by creating family '`libs`'. Parameter '`BRANCH`' from module `parameters.py` contains the name of the Perforce branch with library changes for the experiment. If '`BRANCH`' has zero length, then no changes have been supplied, family '`libs`' is set to complete and the standard release libraries will be used.

Family '`libs`' contains tasks '`p4setup_ifs`', to set up the perforce client, and '`p4diff_ifs`', to list changes on the perforce branch (if there is one). List '`liblist`' is then constructed, containing a list of all the libraries which might need to be compiled on the given platform. The list for the supercomputer is significantly longer than the list for the workstation. Family '`compile`' is added to family '`libs`'. It contains a family for each library in '`liblist`', set to '`complete`' by default, containing tasks to compile libraries, possibly compile modules and possibly (as for the '`odb`'), compile more complicated structures. If a Perforce branch has been supplied, then task '`p4setup_ifs`' will interrogate the branch and find which libraries have been changed and need to be compiled. It will then change the status of the corresponding '`compile`' families from '`complete`' to '`queued`'.

3.8.2 Method build_bins() of class Library()

Method `build_bins()` of `class Library()` is called from method `content()` of `class Make()` (see Subsection 3.6.5.(ii)) to build the binaries on the supercomputer. It starts by creating family '`bins`'. All the child tasks below family '`bins`' are added in a single step as arguments to its `add()` function. Some build tasks are triggered by earlier tasks being complete, some have extra memory requirements, some are executed conditionally.

3.8.3 Method ws_build() of class Library()

If switch '`BUILDWS`' from module `parameters.py` has the value '`True`', then method `ws_build()` of `class Library()` is called from method `content()` of `class Make()` (see Subsection 3.6.5.(ii)) to compile any libraries which have been modified by a supplied Perforce branch, link files and build the binaries on the workstation. It starts by creating family '`buildws`'. Method `onws()` of instance '`resources`' of `class Resources()` from module `inc_common.py` defines the variables which need to be set in order to run the tasks below family '`buildws`' on the workstation rather than the supercomputer. Family

'buildws' is triggered by family 'libs' on the supercomputer being complete. Method make_libs() of class Library() is called with argument 'wsmode = True' to compile any libraries which have been modified (see Subsection 3.8.1 above). Task 'linksws' is added. Then family 'binsws' is added, with a much reduced list of tasks (compared with the supercomputer) to build the binaries which are needed on the workstation. Finally, task 'remove_wsclient' is added below family 'buildws', triggered by family 'binsws' being complete, to remove the Perforce client from the workstation.

3.8.4 Method make_oopsLibs() of class Library()

If switch 'LOOPS' from module parameters.py has the value 'True', then method make_oopsLibs() of class Library() is called from method content() of class Make() (see Subsection 3.6.5.(ii)) to compile on the supercomputer any OOPS libraries which have been modified by a supplied Perforce branch. It creates family 'oops' with child tasks 'oopslib' and 'oopsifslib'.

3.9 CLASS AEOLUS()

Module aeolus.py contains class Aeolus(), which has methods for processing Aeolus data, and class Empty(), a child of class Aeolus(), which has a set of dummy methods. If switch 'LAEOLUS' from module parameters.py has the value 'True', then item 'inst' from module aeolus.py is an instance of class Aeolus(). If not, then 'inst' is an instance of class Empty(). Throughout the Python definition, whenever Aeolus processing is required, it is referred to as aeolus.inst.method(), which will result in genuine Aeolus processing if 'LAEOLUS == True' and a dummy call otherwise.

3.9.1 Method get_aeolus() of class Aeolus()

Method get_aeolus() of class Aeolus() fetches additional code which is needed by the aeolus processing. It is called from method content() of class Make() (see Subsection 3.6.5.(ii)).

3.9.2 Method libs() of class Aeolus()

Method libs() of class Aeolus() builds the binaries which are needed by the aeolus processing. It is called from method build_bins() of class Library() (see Subsection 3.8.2).

3.9.3 Method obs_fetch() of class Aeolus()

If switch 'LAEOLP' from module parameters.py has the value 'True', then method obs_fetch() of class Aeolus() fetches the observations and data files which will be needed for the Level 2B processing of the aeolus data. It is called from method obs_fetch() of class Observations() (see Subsection 3.4.3).

3.9.4 Method obs_prepare() of class Aeolus()

Method obs_prepare() of class Aeolus() is called from method cope() of class Observations() (see Subsection 3.4.7). It creates family 'aeolus_process' and adds task 'odb2odb1_aeolus_l2b' to convert the data from ODB2 to ODB1 format. If switch 'LAEOLP' from module parameters.py has the value 'True', then method aeolus_L2B() of class Aeolus() is called to do the Level 2B processing.

3.9.5 Method aeolus_L2B() of class Aeolus()

Method aeolus_L2B() of class Aeolus() is called from method obs_prepare() of class Aeolus() (see Subsection 3.9.4). It does the Level 2B processing of the aeolus data.

3.9.6 Method l2c() of class Aeolus()

If switch 'LAEOLP' from module parameters.py has the value 'True', then method l2c() of class Aeolus() does the Level 2C processing of the aeolus data after the 4D-Var analysis has completed. It is called from method runner_zero() of class FamAnalysis() (see Subsection 3.6.2.(viii)).

3.9.7 Method archive() of class Aeolus()

If switch 'LAEOLP' from module `parameters.py` has the value 'True', then method `archive()` of `class Aeolus()` adds task '`aeolus_archive`', triggered by task '`fdbksave`', to archive the aeolus data to ECFS. It is called from method `runner()` of `class FamFeedback()` (see Subsection 3.7.1.(ii)).

3.10 THE FORECAST SENSITIVITY TO OBSERVATIONS SUITE

The object-oriented nature of the Python language makes it relatively straightforward to use the data assimilation suite definition as a starting point for a number of other related suite definitions. These include the Ensemble Kalman Filter suite and the Forecast Sensitivity to Observations (fsobs) suite. This chapter describes what steps were needed to derive the fsobs suite definition.

3.10.1 fsobs_ecf.py

For the data assimilation (see Subsection 3.7.1.(ii) above), the top level of the suite definition is in file `an_ecf.py`, which contains the lines:

```
import inc_an as ia
...
DEFS = ecflow.Defs()
ia.AnalysisSuite(DEFS).suite()
```

For the fsobs definition, the corresponding lines in file `fsobs_ecf.py` are:

```
DEFS = ecflow.Defs()
ia.FsobsSuite(DEFS).suite()
```

3.10.2 Class FsobsSuite(AnalysisNoFcSuite)

`Class FsobsSuite()` in module `inc_an.py` is a child of `class AnalysisNoFcSuite()` (see Subsection 3.3.1.(iv)), which in turn is a child of `class AnalysisSuite()`. `Class AnalysisNoFcSuite()` has an analysis family, '`an`' below its '`main`' family, but no cycling forecast family '`fc`'. Also, its '`make`' family does not have a section for making the initial data. `Class FsobsSuite()` inherits all the data and methods of its parent classes. Only those methods which are different for the 'fsobs' configuration need to be supplied.

(i) Constructor method of Class FsobsSuite()

The constructor method of `class FsobsSuite()` starts by setting the suite type, `self.stype`, to "`fsobs`" and then calling the constructor method of its parent `class AnalysisNoFcSuite()`. Then it adds the lines:

```
self.makeIDATA = ifam.MakeIDATAFsobs(self.limits_)
self.fb = ifam.FamFeedBackFsobs(self.observations, self.limits_)
```

(ii) Method suite() of Class FsobsSuite()

Method `suite()` of `class FsobsSuite()` closely resembles method `suite()` of `class AnalysisSuite()` (see Subsection 3.3.2.(iii)), but it needs two more steps at each cycle to prepare the initial data and calculate the sensitivity gradient and it does not have plot jobs running on the workstations.

```
import parameters as ip
...

    def suite(self):
        with self.defs.suite(ip.OWNER) as node:
            with node.family(ip.EXPVER) as fexpver:
            self.limits(fexpver)
            self.setup(fexpver)
```

```
        with fexpver.family(self.stype) as fan:
            self.make(fan)
            self.inicond(fan)
            self.sgr(fan)
            self.obs(fan)
            self.main(fan)
            self.lag(fan)
            self._endFamily(fan)
        endTrig = self.stype + " == complete"
        self._finalize(fexpver, trig = endTrig)
```

(iii) Method inicond() of class FsobsSuite()

Method `inicond()` of `class FsobsSuite()`, called from method `suite()`, closely resembles methods `obs()`, `main()` and `lag()` of `class AnalysisSuite()` (see Subsection 3.3.2.(vi)). It prepares the fsobs experiment's initial data from the reference experiment data.

```
def inicond(self, node):
    """ Prepare initial data for fsobs calculations, where:
        node = position in suite definition tree
    """
    # Loop over streams / times / dates
    fam = self.looper(node, "inicond", self.inicond_fam)

    trigs = [self.maker.path_bins]
    if ip.WAVE:
        trigs.append("make/wconst")
    fam.add(Trigger(trigs))
    return fam
```

Method `looper()` of `class AnalysisSuite()` is called to create family "`inicond`", then loop over all streams and hours, calling method inicond_fam() from `class FsobsSuite()` to prepare the initial data.

(iv) Method inicond_fam() of class FsobsSuite()

Method `inicond_fam()` of `class FsobsSuite()`, called from method `looper()` of `class AnalysisSuite()`, prepares the initial data for a given stream, hour and date. First of all, family '`adj`' is created to prepare the adjoint sensitivity data at `${SENSDELTA_HOUR}` hours before the cycle time, where the value of `SENSDELTA_HOUR` is taken from module `parameters.py`. Then method `_initial_data()` of `class MakeIDATAFsobs()` from module `inc_fam.py` is called to prepare the initial data for the current stream / hour / date. The '`inicond`' family is triggered to not run more than `${ANALYSIS_LAG}` days ahead of the corresponding '`main`' family, where the value of '`ANALYSIS_LAG`' is taken from module `parameters.py`.

(v) Method sgr() of class FsobsSuite()

Method `sgr()` of `class FsobsSuite()`, called from method `suite()`, has a similar structure to method `inicond()`. It calls method `looper()` of `class AnalysisSuite()` to prepare the sensitivity gradient data for all streams, dates and hours.

(vi) Method sgr_fam() of class FsobsSuite()

Method `sgr_fam()` of `class FsobsSuite()`, called from method `looper()` of `class AnalysisSuite()`, prepares the sensitivity gradient data for a given stream, hour and date. Family '`sgr`' is created, containing tasks '`J1`' and '`J1back`'.

(vii) Method lag_fam_arch() of class FsobsSuite()

Method `lag_fam_arch()` of `class FsobsSuite()` resembles method `lag_farm_arch()` of `class AnalysisSuite()` (see Subsection 3.3.2.(xi)). Instead of archiving forecast data, task '`sg`' is called to archive sensitivity gradient fields, and the ODB archiving is triggered by task '`fs_sens_obs`' being complete.

(viii) Method _endFamily() of class FsobsSuite()

Method `_endFamily()` of `class FsobsSuite()` resembles method `_endFamily()` of `class AnalysisSuite()` (see Subsection 3.3.2.(xiv)), but the '`flush`' task is triggered by a different list of families, reflecting the different contents of method `suite()`.

3.10.3 Class StreamFSOBS(StreamLWDA)

`Class StreamFSOBS()` in module `inc_stream.py` is a child of `class StreamLWDA()` (see Subsection 3.5.1).

(i) Constructor method of class StreamFSOBS()

The constructor method of `class FsobsSuite()` starts by calling the constructor method of its parent `class StreamLWDA()`. It defines class variables to switch off the surface analysis and the type-999 analysis (which generates '`type=an`' rather than '`type=4v`' analysis fields) and switch off bias correction, emissivity and monitoring computations.

(ii) Method active() of class StreamFSOBS()

The fsobs stream is active if the value of switch '`LFSOBS`' from module `parameters.py` is '`True`'.

(iii) Method set_fgTriggers() of class StreamFSOBS()

For `class StreamLWDA()`, the first guess comes from the short cycling forecast at the previous cycle time. For `class StreamFSOBS()`, the first guess comes from the short forecast in family '`inicond`' at the current cycle and the sensitivity gradient data for the current cycle also needs to be available.

3.10.4 Class MakeIDATAFsobs(MakeIDATA)

`Class MakeIDATAFsobs()` in module `inc_fam.py` is a child of `class MakeIDATA()` (see Subsection 3.6.6).

(i) Method archive() of class MakeIDATAFsobs()

Method `archive()` of `class MakeIDATAFsobs()` resembles method `archive()` of its parent `class MakeIDATA()` but the date and time variables '`YMD`' and '`HOUR`' are inherited from the stream / hour family, as set by method `looper()` for family '`inicond`', rather than being picked up from `self.makeStream.makeTime`, as in `class MakeIDATA()`.

3.10.5 Class FamFeedbackFsobs(FamFeedback)

`Class FamFeedbackFsobs()` in module `inc_fam.py` is a child of `class FamFeedback()` (see Subsection 3.7.1).

(i) Method runner() of class FamFeedbackFsobs()

Method `runner()` of `class FamFeedbackFsobs()` is much simpler than method `runner()` of its parent class. It just contains a call to method `fcSensObs()` of `class Observations()` to add the forecast sensitivity information to the ODBs before archiving them to MARS.

3.11 ACKNOWLEDGEMENTS

The work described here was done in close cooperation with Axel Bonet-Cassagneau of the Forecast Department. Jan Haseler would like to thank the MACC project for funding the project.

The appendicies on `ecf.py`, `inc_common.py` and class diagrams have been supplied by Axel Bonet-Cassagneau.

APPENDIX 3.A. MODULE ECF.PY

Module `ecf.py` adds an extra layer on top of the native ecFlow API. The aim is partly to facilitate suite design and development, and partly to be able to make changes to ecFlow without having to wait until a new release can be installed. It uses polymorphism, so that all suite components (nodes and attributes) can be added in a similar way. It facilitates good coding practice, reducing the need for temporary variables, lambda functions and list comprehensions.

Variables are provided to disable loading Triggers, Limits or Events in test mode. Similarly, 'Late' attributes that may be useful for operational suites, but not for Research Department experiments, can be disabled. The module also contains a test suite, which provides an example of how to write a suite definition and which can also be used for unit testing.

3.A.1 Class Root()

This is the abstract class from which classes `Suite()`, `Family()` and `Task()` derive.

Method add() of class Root()

Method `add()` of `class Root()` loops over all arguments, which may be lists, tuples or items derived from `class Attribute()`, and attaches them with a call to method `add_to()`.

Other methods of class Root()

`Class Root()` also defines methods `defstatus()`, `fullname()`, `inlimit()`, `limit()` and `repeat()`.

3.A.2 Class Node(Root)

`Class Node()` is a child of `class Root()`. It is the parent of classes `Family()` and `Task()`.

Methods of class Node()

`Class Node()` defines methods `add_limits()`, `complete()`, `complete_and()`, `complete_or()`, `cron()`, `event()`, `label()`, `meter()`, `time()`, `today()`, `trigger()`, `trigger_and()`, `trigger_or()` and `up()`.

3.A.3 Class Attribute()

`Class Attribute()` is the generic class for items which can be attached to a node.

Method add_to() of class Attribute()

Each class derived from `class Attribute()` has to overwrite this method in order to define the way it is attached to its parent node. Method `add_to()` contains native ecFlow API methods such as `add_limit()`, `add_inlimit()`, `add_repeat()`, etc.

Derived classes

Classes `Family()` and `Task()` derive with multiple inheritance from classes `Node()`, `Attribute()` and `ecflow.Family()` and `ecflow.Task()` respectively.

Other classes which derive from `class Attribute()` include:

- `Variable()` and `VVariables()` (which accepts a list, dictionary, strings, or 'argument=value' pairs for variable definition),
- `Label()`, `Event()`, `Meter()`, `Limit()`, `InLimit()`
- `Trigger()`, `Complete()`, `Time()`, `Today()`, `Date()`, `Day()`, `Cron()`
- `Clock()`, `Autocancel()`, `Late()`

Classes which derive in turn from the above include:

- `TriggerAnd()`, `TriggerOr()`, `Complete()`, `CompleteAnd()`, `CompleteOr()`

- `Defcomplete()`, `DefcompleteIf()`

Class TriggerAlways(Trigger)

`Class TriggerAlways()`, which is a child of `class Trigger()`, defines a trigger which will always be satisfied when it is added to a node.

Class TriggerImpossible(Trigger)

`Class TriggerImpossible()`, which is a child of `class Trigger()`, defines a trigger which will never be satisfied when it is added to a node.

3.A.4 Class State()

`Class State()` is a class which is used to define objects which can be used to test for the status of nodes.

Objects of class State()

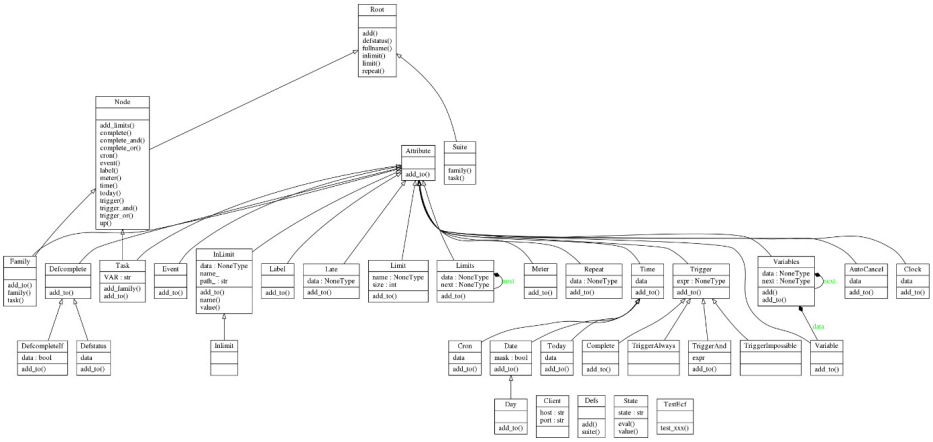
`SUBMITTED`, `ACTIVE`, `SUSPENDED`, `ABORTED`, `QUEUED`, `COMPLETE`, `UNKNOWN` are created as instances of `class State()`. Associated with the definition of class methods `__eq__`, `__ne__`, `__and__`, `__or__` for both `State()` and `Root()` classes, they allow the syntax `"Trigger(item == COMPLETE)"` to be used in the suite definition, with '`item`' being a Task or a Family. Provided that '`item`' is already attached to its parent family, the full (absolute) path is then added in the trigger expression, with no further intervention from the suite designer. This may prevent inconsistencies between hard-coded trigger expressions and the need to change the location of the item inside the suite.

3.A.5 Method If(test=, then=, otow=)

Method `If()` of module `ecf.py` returns the '`then`' clause if '`test`' is satisfied. Otherwise it returns the '`otow`' clause. It provides the ability to load in a suite item, or list of items, according to a test condition. Both branches are created but only one is added to the parent. This makes it possible to detect problems with the 'dead' branch at an earlier stage. (The test could be `VERSION=="0001"`, and it is now possible to identify broken triggers, before testing the operational suite.)

3.A.6 Class diagram for module ecf.py

Figure **??** below shows a class diagram for module `ecf.py`. Each class is represented by a rectangular box with three compartments. The top compartment contains the class name. The middle compartment contains the names of the class variables. The bottom compartment contains the names of the class methods. The boxes of child classes are linked with arrows to the boxes of their parent classes.



Class Diagram for module ecf.py.

APPENDIX 3.B. MODULE INC_COMMON.PY

Module `inc_common.py` originated from the operational emos environment. Its purpose is to provide system related settings and common entities that can be used across multiple suites.

3.B.1 Method find()

Once a suite has been created, method `find()` of module `inc_common.py` can be used to attach (or delete) attributes (e.g. Trigger, Complete) to nodes with a specific name, or for which the path (`ECF_NAME`) matches a regular expression.

3.B.2 Method add_ymds()

Method `add_ymds()` generates a concise and consistent notation for triggers which are YMD dependent (e.g. families `lag` and `pop`). This is described in more detail in Subsection 3.2.2.(viii) above.

3.B.3 Class Resources()

`Class Resources()` contains methods to define resources required by tasks which run on the supercomputer (method `onsc()`) or on a workstation (method `onws()`); methods to set the account or the priority, and methods to define specific variables for jobs or families (method `joblim()`).

3.B.4 Class Seed()

`Class Seed()` is a skeleton class for suite definition.

3.B.5 Classes Walk() and Traverse()

These classes and their methods can be used to print a suite, to filter out part of it, or to design "watch dog" utilities that check the status of tasks or the values of variables.

3.B.6 Method expand_dict()

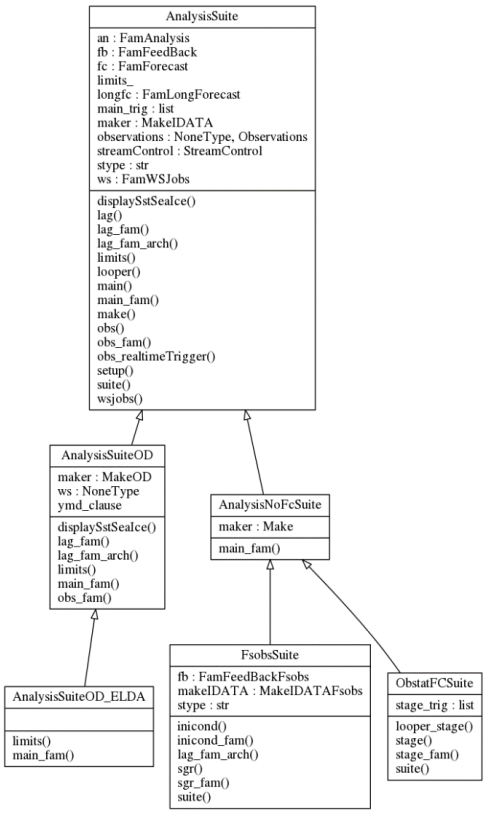
Method `expand_dict()` can be used to expand a suite definition which has been defined as a python dictionary.

For the emos environment, module `inc_common.py` is referenced via module `inc_emos.py`. This contains code specifically for emos configurations, which would not be useful for research suites.

APPENDIX 3.C. CLASS DIAGRAMS

3.C.1 Module inc_an.py()

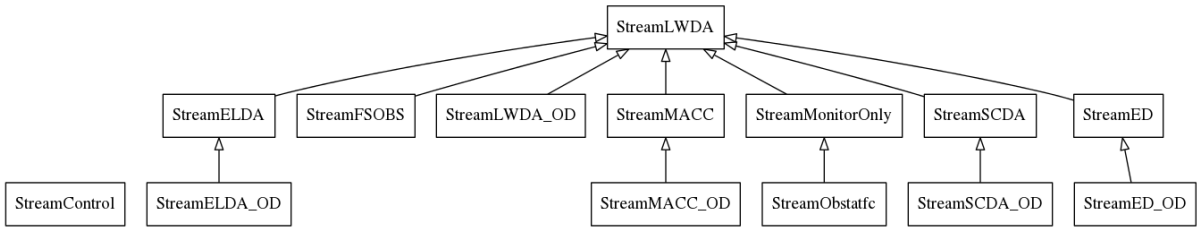
`Class AnalysisSuite()` is the main class of interest. `AnalysisSuiteOD()` is present in the same package, making it possible to appreciate the differences with the operational structure, or to run non regression tests.



Class Diagram for module inc_an.py.

3.C.2 Module inc_stream.py()

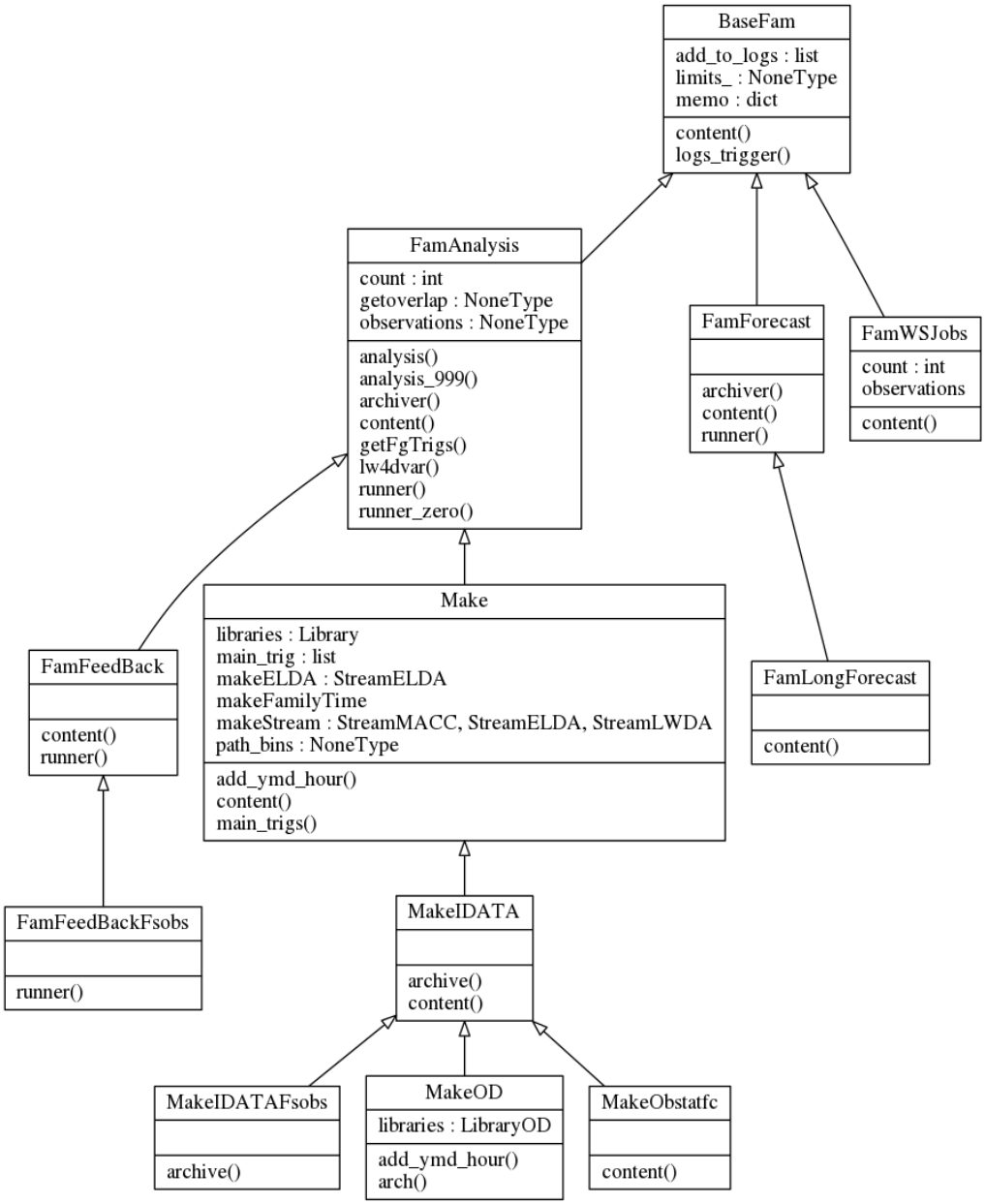
Most classes derive from `class StreamLWDA()`. FD specificities are handled with dedicated derived classes.



Class Diagram for module inc_stream.py.

3.C.3 Module inc_fam.py()

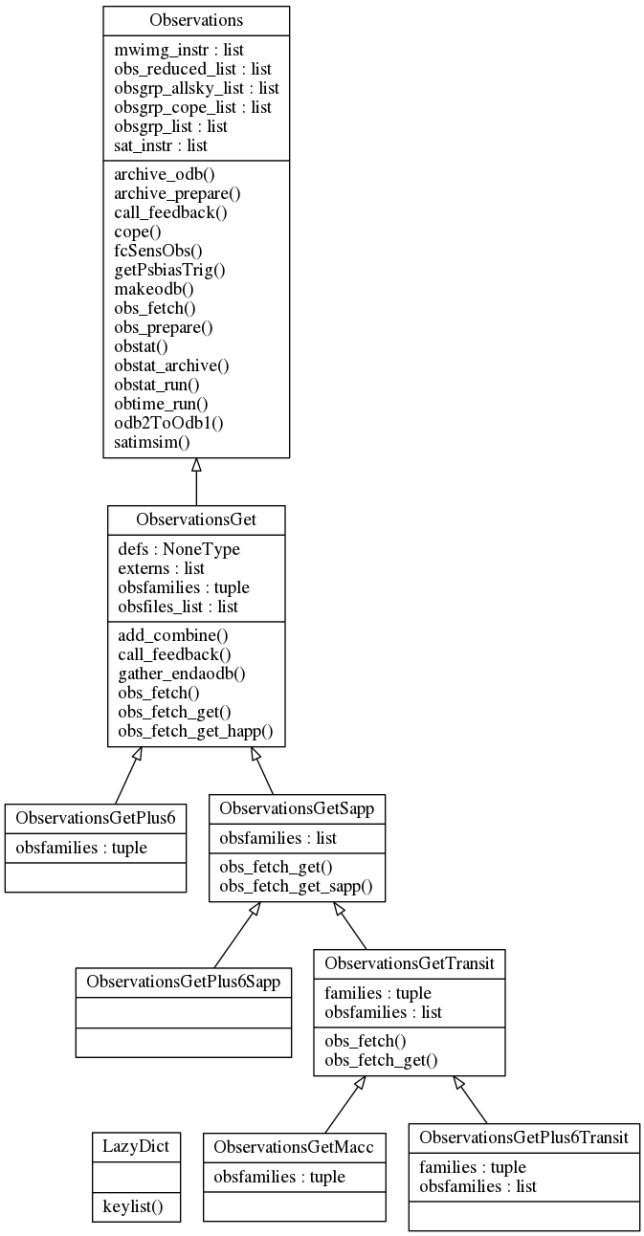
A common pattern could be factorised into `class BaseFam()`. Classes `FamAnalysis()` and `FamForecast()` can then derive from it. Classes `FamFeedback()`, `Make()`, `FamLongForecast()` are specialisations.



Class Diagram for module inc_fam.py.

3.C.4 Module inc_obs.py()

One of the most important classes is `Observations()`, which contains tasks '`fetchobs`' and '`fetchmars`'. Class `ObservationsGet()` is derived from it to introduce the "`get`" tasks needed in real-time mode. For some streams, '`Plus6`' and '`Minus6`' variants are needed. Class derivation simplified the transition to the SAPP system. A dedicated class was used to manage the transition, with both HAPP and SAPP "`get`" tasks being added in the same obs family.



Class Diagram for module inc_obs.py.

Appendix A
Structure, data flow and standards

Table of contents

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 1.1.

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 2.2.

Table 1.1 *Command line options.*

Option	Namelist Variable	Description
-c	NCONF	Job Configuration (see Table 4.11 on page 105)
-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 2.2 *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 2.2 *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 2.2 *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 2.2 *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

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
/
```

3.1 Index of namelists

Tables 3.3 to 3.10 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 3.3 *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 3.4 *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 3.5 *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 3.6 *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 3.7 *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 3.8 *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 3.9 *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	SUMP0

Table 3.10 *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

4 NCONF: IFS CONFIGURATION PARAMETER

The NCONF parameter is supplied by the namelist namct0 (see Table 3.3 on page 100), or this value can be overridden on the command line (using the "-n" option, see Table 1.1 on page 96). NCONF controls the function of any single execution of the IFS. Table 4.11 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 4.11 *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

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 6.16 on page 111). GMV variables are also contained in this file.

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

Table 5.12 *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 5.13 *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

continued on next page . . .

Table 5.13 *ICMGGxxidINIT: Surface fields on model Gaussian grid. (continued . . .)*

ECMWF GRIB code	IFS variable (GRIB name)	Description
129	NGRB<Z>	Geopotential (at the surface orography) *(If not provided in a spectral field)*
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

ICMGG*xxid*INIUA Contains upper air fields in grid point space.

All fields in this file have NFLEV levels.

Table 5.14 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 5.14 *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

ICMCL*xxid*INIT 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.).

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 5.15 *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

[1] LMCCIEC, defined in namelist NAMMCC (see Table 3.3 on page 100) is set to .TRUE. (the default value) if the climate fields are to be interpolated in time.

6 GMV AND GFL STRUCTURES IMPLEMENTATION AND USAGE

6.1 GMV structure

6.1.1 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.

6.1.2 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.

6.2 GFL structure

6.2.1 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

6.2.2 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 6.16) and pointers (Table 6.17). 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 6.16 *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

6.2.3 Adding a new attribute

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

Table 6.17 *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

6.2.4 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.

6.2.5 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).

7 OBJECT-DRIVEN FIELD API IN PREPARATION FOR ATLAS

The key strategic aim of the introduction of an object-based field API is to introduce a "Separation of Concerns" that allows performance engineering research (focused on accelerator architectures) to happen with little impact on the scientific day-to-day development. A first step towards this goal is the introduction of an abstraction layer for data layout in memory, as the "best-known" memory placement and layout may differ on alternative architectures. The Atlas library has been chosen to perform this role within the IFS software stack, as it encapsulates parallel domain decomposition and grid layouts, and thus necessarily has close control over memory allocation and placement.

Before introducing Atlas into the core sections of the IFS, it is important to ensure that existing data handling and performance optimisations can be replicated within an Atlas-based framework. One of the key realisations, however, is that up to this point all field arrays are allocated and managed manually as raw Fortran arrays, while Atlas represents individual fields as objects that provide a library-based API. Moreover, a specific block-allocation is used for all core and temporary fields throughout the IFS that complements the thread-parallel loop-blocking scheme (NPROMA) that is used to optimise grid-point computations for CPUs, which needs to be replicated under any new scheme in order to match current performance. To enable a gradual migration of individual IFScomponents an object-oriented API that is compatible with the current array management and can replicate current behaviour is being introduced in cycle 47.

7.1 Design overview

Before introducing explicit Atlas field objects as primary variable data managers, the internal data and type layout of the IFS needs to be restructured. The aim is to incrementally introduce an object-driven type hierarchy and API that cleanly exposes access to data and meta-data associated with variables and

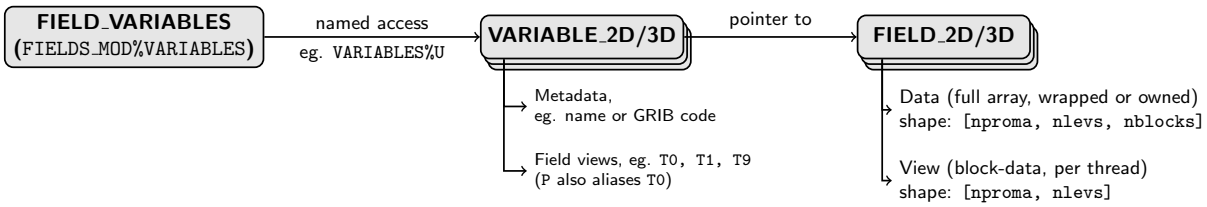


Figure 7.1 .

fields, while being compatible with the majority of the scientific code in the IFS. For this purpose two new derived types (classes) will be introduced to enscapulate the concepts of `VARIABLE` and `FIELD` objects that provide access to data pointers and associated meta-data. The objects mimic actual Atlas objects, as only a very limited set of capabilities is required from them, so that the compile-time dependency on Atlas remains optional.

In addition to separating `FIELD` objects from the arrays they encapsulate, the concept of a `VARIABLE` will be introduced to highlight the fact that several fields may share meta-data (eg. the values of a variables at the current, previous or next timestep). These associations are already in use throughout the IFS, but mapping is based on implicit naming assumptions, rather than strict object encapsulation. As shown in Figure 7.1, the new API will include explicit `VARIABLE` objects that primarily encapsulate meta-data pertaining to scientific and numerical concepts (eg. if that variable has multiple time levels or gets advected, etc.), while a `FIELD` objects manages the storage of field data in memory.

7.2 Data views: Array accesses in thread-parallel regions

One of the key performance optimisations in the IFS is the memory-blocked data layout of fields and its according thread-parallel loop-blocking strategy for grid-point computations, where individual threads work exclusively on `NPROMA`-sized sub-arrays (data views). This concept of a "data view" is already in use in the "driver" routines that define thread-parallel regions through the implicit variable scoping mechanisms used for persistent state variables (`GMV` and `GFL` base arrays), as well as the setup of per-thread temporary variables. One goal of the new object-based API is to unify the necessary boilerplate code to set up such data views (pointers) under a common API that will be provided by `FIELD` storage objects, so that variable scoping and memory placement may be managed explicitly without affecting their use in scientific kernel routines. The new API being introduced into `EC_PHYS` will therefore wrap existing state field arrays and manually manage temporaries, so that data access semantics are aligned throughout the scientific code, while backward compatibility with other components is maintained.

7.2.1 Access to persistent state variables

As shown in Figure 7.1 the new API consists of a derived type `FIELD_VARIABLES` that provides access to objects of type `VARIABLE_2D` or `VARIABLE_3D`. Each variable object provides multiple data view pointers to individual block sub-arrays for different time levels, each of which is backed by a storage object of type `FIELD_2D` or `FIELD_3D`. An example of accessing the corresponding data arrays that is equivalent to the example in Section 6 on page 109 is shown below.

```
YDVARS%U%T9(JL,JK)    = REPS1*YDVARS%U%T9(JL,JK)    + &
                      & ZREST*YDVARS%U%T0(JL,JK)
YDVARS%V%T9(JL,JK)    = REPS1*YDVARS%V%T9(JL,JK)    + &
                      & ZREST*YDVARS%V%T0(JL,JK)
YDVARS%DIV%T9(JL,JK)  = REPS1*YDVARS%DIV%T9(JL,JK)  + &
                      & ZREST*YDVARS%DIV%T0(JL,JK)
```

7.2.2 Access to temporary variables

The temporary variables used throughout the ECMWF physics routines are grouped in a set of derived types and explicitly allocated/deallocated per block by each thread. In cycle 47 the explicit pointer

management has been replaced by data views extracted from associated FIELD objects that are created before entering the thread-parallel region in EC_PHYS_DRV. Prior to cycle 47 these derived types were stored in ifs/module/yomphyder.F90, while most of them are now defined in files of the format ifs/module/ecphys_<type_name>.F90 due to the additional boilerplate code required to populate the relevant pointers (data views) from the associated storage objects. Thus, their use in scientific subroutines remains unchanged.

7.2.3 Access to surface variables

In addition to the GMV and GFL variables, surface variables have also been wrapped for cycle 47, despite their access semantics remaining unchanged. The only exception to this are the time step updates that are now executed via the SURF_AND_MORE_TYPE that provides the type-bound utility routines SET9TO0(), SET1TO9, SET1TO0, SET0TO1 and PHTFILT. A secondary access API that honours the existing variable groupings is planned to be introduce in future cycles.

7.3 Changes to development workflow

It is important to note that, in order to ensure a smooth transition to the new Atlas-enabled object-driven data structures, a small overhead in common practices is unavoidable during the migration phase. While a large number of array accesses throughout the ECMWF physics routines have been converted to the new API, a number of variables are still accessed via indirect indexing into the GFL base array. Since the new data structures wrap existing memory in GFL GFL arrays, both approaches are fully compatible.

The creation of the new data structures to wrap existing field arrays is largely automated through the use of the FYPP preprocessor. The definition of the persistent core variables is defined in a configuration file ifs/module/field_config.yaml that defines the individual fields to be wrapped using the new data types. A similar data structure exists for surface fields in ifs/module/surface_fields_config.yaml that also defines the particular variables groupings. Therefore, the addition of a new variable to the IFS requires:

(i) Add the new component to the GFL data structure as described in Section 6 on page 109
(ii) Add a variable descriptor line of the form - {name: FOO, comment: My new variable} to the corresponding sub-group in either field_config.yaml or surface_fields_config.yaml.

In order to add local temporary variables for use inside the OpenMP thread-parallel region in EC_PHYS or CALLPAR, the a new array pointer, as well as a pointer to a FIELD_2D/2D object should be added to one of the derived types in ifs/module/ecphys_<typename>.F90. The boilerplate code to create the new storage object and set the data view pointer needs to be added to the respective routines <GROUP_TYPE>_INIT, <GROUP_TYPE>_UPDATE_VIEW and <GROUP_TYPE>_FINAL. The creation of a 3-dimensional temporary field, for example, will be:

```
! Temporary FIELD object creation in <GROUP_TYPE>_INIT()
SELF%F_FOO  => CREATE_TEMPORARY(FOO',␣GEOM=REGISTRY%GEOM)

!␣Data␣view␣update␣per␣block␣in␣<GROUP_TYPE>_UPDATE_VIEW(BLOCK_INDEX)
SELF%FOO␣=>␣SELF%F_FOO%GET_VIEW(BLOCK_INDEX)

!␣FIELD␣finalisation␣in␣<GROUP_TYPE>_FINAL()
CALL␣DELETE_TEMPORARY(SELF%F_FOO)
```

Appendix B
Message Passing Library (MPL)

Table of contents

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.

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

3 MPL_ALLGATHERV

Concatenate data from all processes

Purpose

Gathers data from all processes and distributes the resulting data to all processes in the group.
NOTE: The MPL_ALLGATHER routine is a dummy routine so this routine should be used where an ALLGATHER is required.
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_ALLGATHERV (sendbuf,recvbuf, KRECVCOUNTS)
```
Alternatives are: PSENDBUF=buffer, PRECVBUF=buffer, KSENDBUF=ibuffer or KRECVBUF=ibuffer

Input Arguments

Required

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

KRECVCOUNTS
: Array of counts stating how many elements will be received from each processor.

Optional

KRECVDISPL
: Array of displacements specifying where each message from the other processes should go in the receive array.

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

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

PRECVBUF
: Buffer to receive the message
(may be type (REAL*4), REAL*8 or INTEGER).

Optional

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

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

4 MPL_ALLREDUCE

Perform a reduction on data from all processes, with all processes receiving the result.

Purpose

Performs an operation on data from all processes and distributes the result to all processes in the group.
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_ALLREDUCE (sendbuf, oper)
```
Alternatives are: PSENDBUF=buffer or KSENDBUF=ibuffer

Input Arguments

Required

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

OPER
: Operation to be performed on the data, may be the string 'SUM', 'MIN', 'MAX, 'IEOR', or 'XOR'.

Optional

LDREPROD
: Logical specifying whether the reduction should be reproducible.

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_ALLGATHERV aborts when an error is detected.

5 MPL_ALLTOALLV

Sends differing data from all processes to all other processes.

Purpose

All processes send a block of data, with all processes receiving a different chunk of that data from all other processes.
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_ALLTOALLV (sendbuf, oper)
```
Alternatives are: PSENDBUF=buffer or KSENDBUF=ibuffer

Input Arguments

Required

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

KRECVCOUNTS
: Array of counts stating how many elements will be received from each processor.

KSENDCOUNTS
: Array of counts stating how many elements will be sent by each processor.

Optional

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

KRECVDISPL
: Array of displacements specifying where each message from the other processes should go in the receive array.

KSENDDISPL
: Array of displacements specifying where each message from the other processes should be send from in the send array.

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

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

Output Arguments

Required

PRECVBUF
: Buffer to receive the message
(may be type (REAL*4), REAL*8 or INTEGER).

Optional

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

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

6 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.

7 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.

8 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.

9 MPL_BYTES

Function that returns the number of bytes in the datatype of the argument.

Purpose

A function to query the number of bytes of the argument's datatype.

Interface

`CALL MPL_BYTES (var)`
Alternatives are: `KVAR=ivar`, or `PVAR=var`

Input Arguments

Required

`PBUF`

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

Optional

None

Output Arguments

Required

None

Optional

None

10 MPL_CLOSE

Close an MPIIO file

Purpose

Close a file that has been opened for MPIIO.

Interface

```
CALL MPL_CLOSE (handle,error)
```

Input Arguments

Required

KFPTR
: File handle of type INTEGER.

Optional

None

Output Arguments

Required

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

Optional

None

11 MPL_COMM_CREATE

Create a new communicator

Purpose

Create a new communicator and set as default

Interface

```
CALL MPL_COMM_CREATE
```

DEFERRED IMPLEMENTATION

12 MPL_COMM_FREE

Free a communicator

Purpose

Free an MPI communicator

Interface

```
CALL MPL_COMM_FREE (comm,error)
```

Input Arguments

Required

KCOMM
: Handle of communicator to be freed.

Optional

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

Output Arguments

Required

KERR
: Return error code
If not supplied, MPL_CLOSE aborts when an error is detected.

Optional

None

13 MPL_COMM_SPLIT

Split a communicator

Purpose

Split an MPI communicator

Interface

```
CALL MPL_COMM_SPLIT (oldcomm, color, key, newcomm, error)
```

Input Arguments

Required

KCOMM
: Handle of communicator to be split.

KCOLOR
: Processes with the same colour will be in the same new communicator. So the number of unique communicators will be given by the number of unique values of KCOLOR.

KKEY
: The processes in each new communicator are ordered by rank according to their value of KKEY.

Optional

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

Output Arguments

Required

KNEWCOMM
: Return new communicator handle.

KERR
: Return error code
If not supplied, MPL_CLOSE aborts when an error is detected.

Optional

None

14 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.

15 MPL_GATHERV

Concatenate data from all processes to a single process

Purpose

Gathers data from all processes to a single process in the group.
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_GATHERV (sendbuf,recvbuf, KRECVCOUNTS)
```
Alternatives are: PSENDBUF=buffer, PRECVBUF=buffer, KSENDBUF=ibuffer or KRECVBUF=ibuffer

Input Arguments

Required

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

KRECVCOUNTS
Array of counts stating how many elements will be received from each processor.

Optional

KROOT
Array of displacements specifying where each message from the other processes should go in the receive array.

KRECVDISPL
Array of displacements specifying where each message from the other processes should go in the receive array.

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

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

PRECVBUF
Buffer to receive the message, required on KROOT
(may be type (REAL*4), REAL*8 or INTEGER).

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

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

16 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.

17 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

18 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

19 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

20 MPL_OPEN

Open an MPIIO file

Purpose

Open a file for MPIIO.

Interface

```
CALL MPL_OPEN (handle,type,name,error)
```

Input Arguments

Required

KTYPE
: File open mode, 1 for read only, or 2 for create and write. Argument type INTEGER.

KNAME
: File handle of type INTEGER.

Optional

None

Output Arguments

Required

KFPTR
: File handle of type INTEGER.

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

Optional

None

21 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.

22 MPL_READ

Read an MPIIO file

Purpose

Read a file using MPIIO.

Interface

```
CALL MPL_READ (handle,op,buf,len,req,error)
```

Input Arguments

Required

KFPTR
: File handle of type INTEGER.

KOP
: Argument type INTEGER.

KLEN
: Length of data to be read of type INTEGER.

Optional

None

Output Arguments

Required

KBUF
: Buffer of data that has been read, (may be type REAL*8 or INTEGER)

KREQ
: Communication request identifier for when IO type is non-blocking.

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

Optional

None

23 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.

24 MPL_SCATTERV

Scatter data to all processes

Purpose

Scatter data from a specific processor.
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_SCATTERV (recvbuf, KROOT, sendbuf)
```
Alternatives are: PSENDBUF=buffer, PRECVBUF=buffer, KSENDBUF=ibuffer or KRECVBUF=ibuffer

Input Arguments

Required

KROOT
: Root process that will send the data.

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

Optional

KSENDCOUNTS
: Array of counts stating how many elements will be sent by the root processor.

KSENDDISPL
: Array of displacements giving the offsets between each data chunk sent by the root processor.

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

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

PRECVBUF
: Buffer to receive the message
(may be type (REAL*4), REAL*8 or INTEGER).

Optional

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

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

25 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.

26 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.

27 MPL_WAITANY

Waits for completion of any of the messages from an array of requests

Purpose

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

Interface

`CALL MPL_WAITANY(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

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.

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.

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.

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_TRANS0), 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.

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.

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

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.

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

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.

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

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.

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.

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.

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").

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.

2 BASIC USAGE

2.1 Getting started

2.1.1 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).

2.1.2 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 NAMCTO NAMDIM NAMDYN NAMPARO NAMPAR1 NAMOPH NAMFA NAMCT1.

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

2.1.3 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 : NAMCTO

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 : NAMCTO

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 : NAMCTO

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 : NAMCT0

LFPOS :

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

Scope : Boolean.

Default value : .FALSE.

Namelist location : NAMCT0

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 : NAMCT0

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.

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.

2.2.1 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 6.1 on page 198.

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 6.1.6.1.1 on page 199.

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 6.1.6.1.2 on page 200.

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 6.1.6.1.3 on page 201.

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 6.1.6.1.4 on page 203.

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.

2.2.2 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 2.1 on page 167.

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 2.2 on page 168;
elseif CFPFMT='LELAM' then refer to Table 2.3 on page 169.

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 2.2 on page 168;
elseif CFPFMT='LELAM' then refer to Table 2.3 on page 169.

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 2.3 on page 169.

Table 2.1 *Default values for NLAT and NLON according to CFPFMT and LELAM.*

(NLAT,NLON) CFPFMT	'GAUSS'	LELAM	'LALON'
LELAM			
.FALSE.	(NDGLG,NDLON)	See Table 2.3	See Table 2.2
.TRUE.	(32,64)	(NFPGUX,NFPLUX)	See Table 2.2

Table 2.2 *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 2.3 *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		

2.2.3 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 4.4 on page 193 —) 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 4.4 on page 193 —).

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 2.3 on page 169.

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.

2.3 Output files handling

2.3.1 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].

2.3.2 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.

2.3.3 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 2.4 on page 172.

Table 2.4 *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

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.

3.1 Scientific options

3.1.1 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 3.3.3.3.1 on page 182.

3.1.2 Tuning of the spectral filters

Several fields can be smoothed via tunable filters activated in spectral space (refer to Section 9 on page 207 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.

3.1.3 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 8 on page 206 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 3.5 on page 175 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 3.5 *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

3.1.4 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

3.1.5 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
```

3.1.6 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.

3.1.7 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.

3.1.8 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

3.1.9 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 $\delta(\ln p)$. 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

3.2 Optimizing the performance

NPROMA :

Definition : working length of the model data rows. Refer to Section 10.2 on page 209 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 10.2 on page 209 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 10.2 on page 209 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 : NAMPAR0

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 : NAMPAR0

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 : NAMPAR0

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 : NAMPAR0

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 10.1 on page 209 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

3.3 Output fields conditioning

3.3.1 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 6.1 on page 198 for upper air fields, and to Section 6.1.6.1.1 on page 199 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

3.3.2 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 6.1.6.1.2 on page 200. 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 6.1 on page 198 for upper air fields, and to Section 6.1.6.1.1 on page 199 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

3.3.3 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 4 on page 189).

Namelist location : NAMFPC

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 7 on page 204 shows an example of selection file.

3.5 Miscellaneous

3.5.1 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 6.1.6.1.2 on page 200.

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 6.1 on page 198 for upper air fields, and to Section 6.1.6.1.1 on page 199 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

3.5.2 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 : NAMCT0

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

4 THE FAMILY OF CONFIGURATIONS 927

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 4.1 on page 189, gobbleenv below,

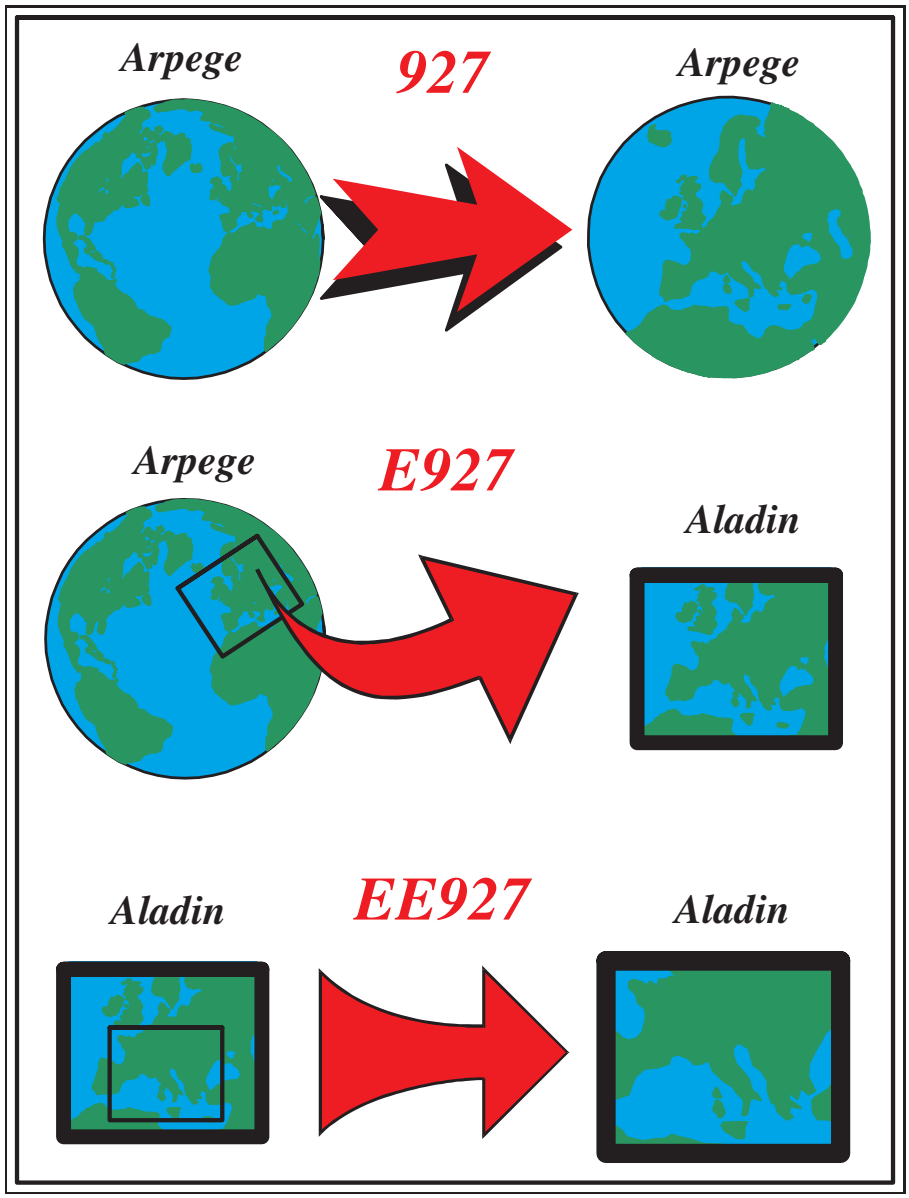


Figure 4.1 *The configuration 927, E927 and EE927.*

- 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).

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.

4.3 Namelists parameters

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

```
&NAMCT0
   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)='SURFZ0.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)='SURFGZ0.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
```

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.

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.

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.

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 $\times$ 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.)

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.

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 6.1.6.1.2 on page 200.

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 6.1.6.1.2 on page 200.

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 6.1 on page 198.

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 6.1 on page 198.

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 6.1 on page 198.

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 6.1.6.1.1 on page 199.

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 6.1.6.1.1 on page 199.

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 6.1.6.1.1 on page 199.

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.

6 FIELD DESCRIPTORS

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

6.1.1 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_HTOB 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_PTOB 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
```

6.1.2 *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   = SURFZO.FOIS.G       NBSR   = 24
Surface soil wetness ................. CNSSW  = SURFRESERV.EAU      NBSSW  = 24
Surface temperature .................. CNST   = SURFTEMPERATURE     NBST   = 24
Thermal roughness length (times g) ... CNZOH  = SURFGZO.THERM       NBZOH  = 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)
```

6.1.3 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)

6.1.4 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 = SOOOFL.CT CONVEC   NBXCVS = 24
Contribution of Convection to Q ...... CNXCVQ = SOOOFL.Q  CONVEC   NBXCVQ = 24
Contribution of Convection to U ...... CNXCVU = SOOOFL.U  CONVEC   NBXCVU = 24
Contribution of Convection to V ...... CNXCVV = SOOOFL.V  CONVEC   NBXCVV = 24
Contribution of Gravity Wave Drag to U CNXGDU = SOOOFL.U ONDG.OR   NBXGDU = 24
Contribution of Gravity Wave Drag to V CNXGDV = SOOOFL.V ONDG.OR   NBXGDV = 24
Contribution of Turbulence to Cp.T ... CNXTUS = SOOOFL.CT TURBUL   NBXTUS = 24
Contribution of Turbulence to Q ...... CNXTUQ = SOOOFL.Q  TURBUL   NBXTUQ = 24
Contribution of Turbulence to U ...... CNXTUU = SOOOFL.U  TURBUL   NBXTUU = 24
Contribution of Turbulence to V ...... CNXTUV = SOOOFL.V  TURBUL   NBXTUV = 24
Convective Cloud Cover ............... CNXCCC = SURFNEBUL.CONVEC   NBXCCC = 24
Convective precipitation ............. CNXCP  = SOOOPLUIE CONVEC   NBXCP  = 24
Convective Snow Fall ................. CNXCSF = SOOONEIGE 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 = SOOOPLUIE STRATI   NBXLSP = 24
Large Scale Snow fall ................ CNXLSS = SOOONEIGE 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 = SOOORAYT.SOLAIRE   NBXSSR = 24
Surface Thermal radiation ............ CNXSTR = SOOORAYT.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
```

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
```

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).

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 9.2 on page 207 illustrates this spectral filter. gobbleenv The next figure illustrates this spectral filter:

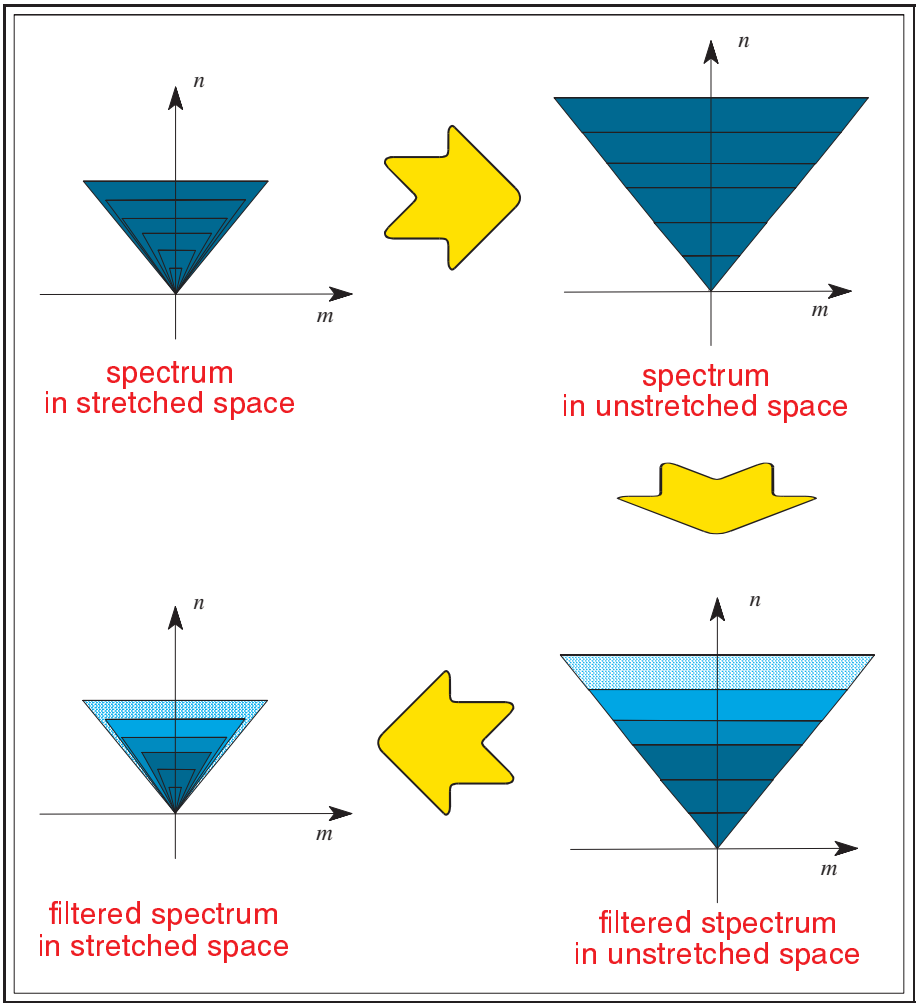


Figure 9.2 *Illustration of the spectral filter for derivatives in* ARPEGE/IFS.

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 9.3 on page 208 illustrates this spectral filter. gobbleenv The next figure illustrates this spectral filter:

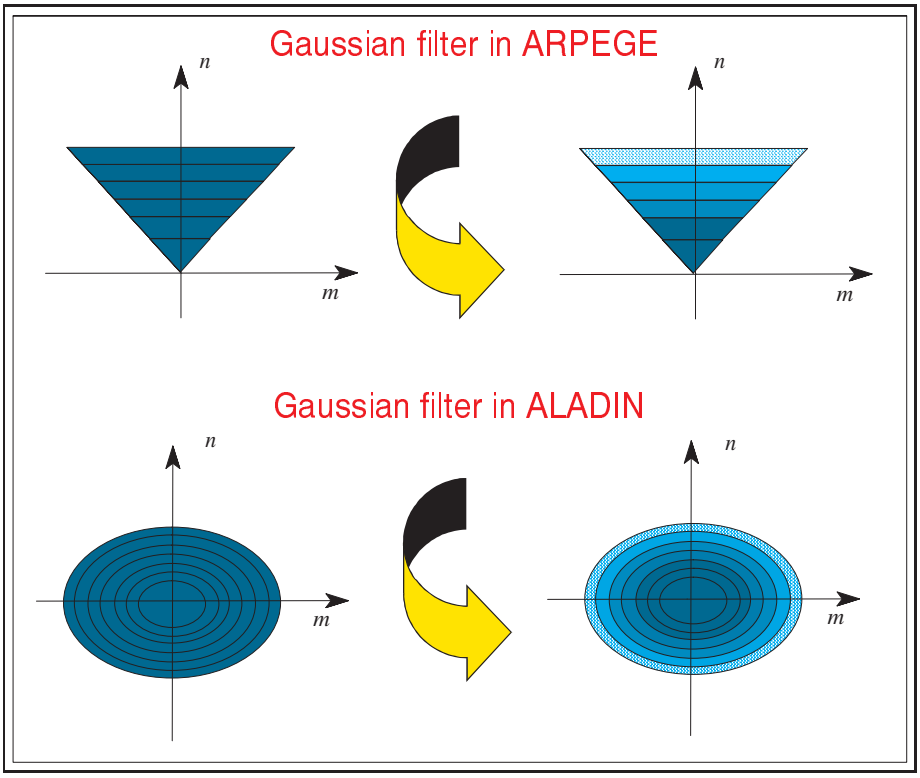


Figure 9.3 *Illustration of the Gaussian spectral filter.*

10 OPTIMIZATION OF THE PERFORMANCE

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.

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

1 FOUNDER PRINCIPLES

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).

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).
- 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

[1] This is less and less true.

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.

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.

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.

2 GENERAL CONCEPTION

2.1 Architecture

This section will describe the main subroutines involved in FULLPOS and how they are articulated between one another.

2.1.1 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 2.1 on page 214), though all the subroutines are not used. The next list describes the purpose of each control subroutine:

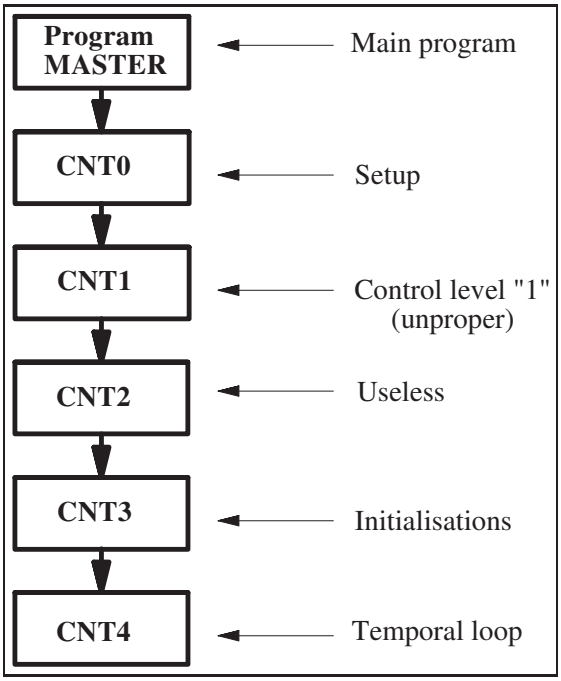


Figure 2.1 *General mechanism of the post-processing inside the model.*

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 2.2 on page 216). 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.

2.1.2 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.

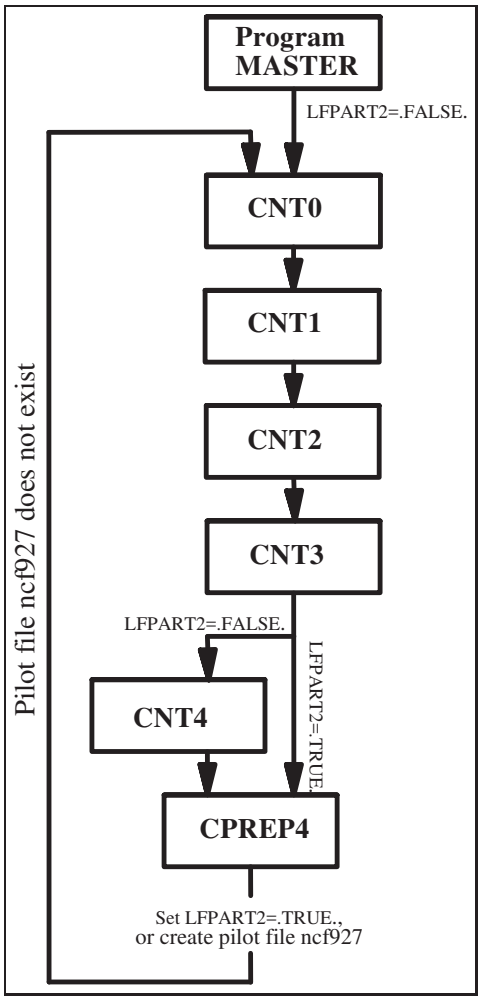


Figure 2.2 *General mechanism of the so-called configuration (e)(e)927.*

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 2.3 on page 217 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.

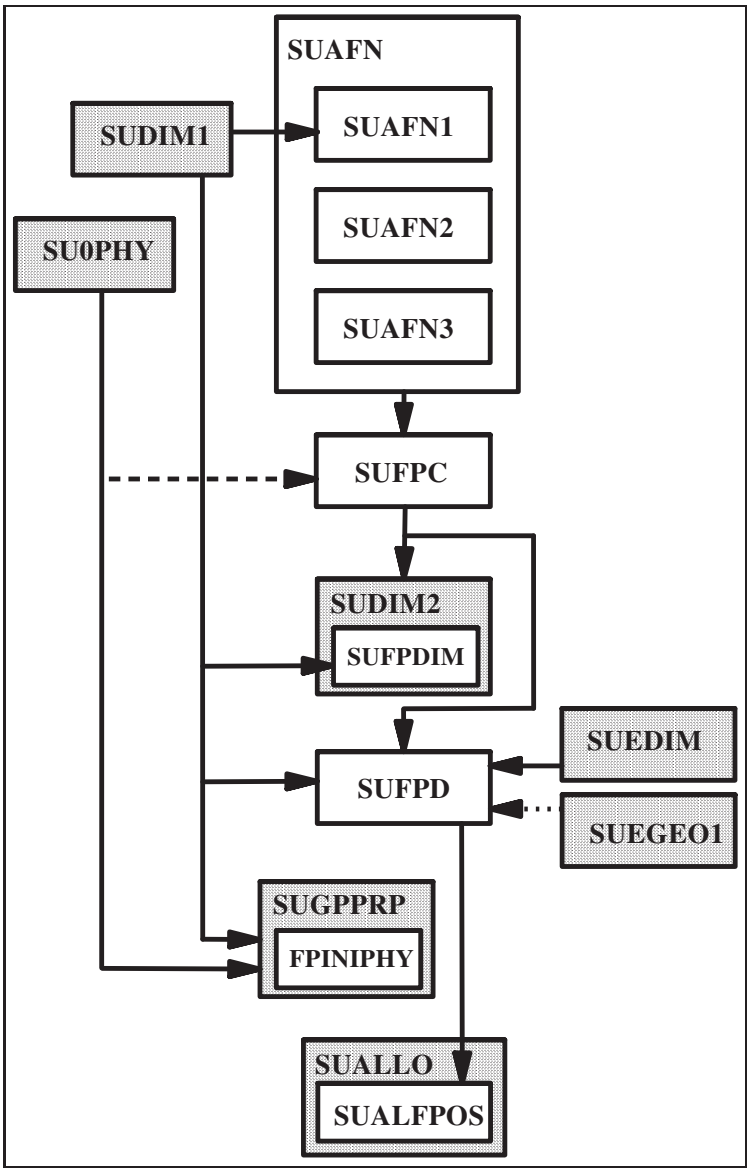


Figure 2.3 *Setup, first part: SU0YOMA. The greyish areas correspond to model subroutines.*

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 2.4 on page 219 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!

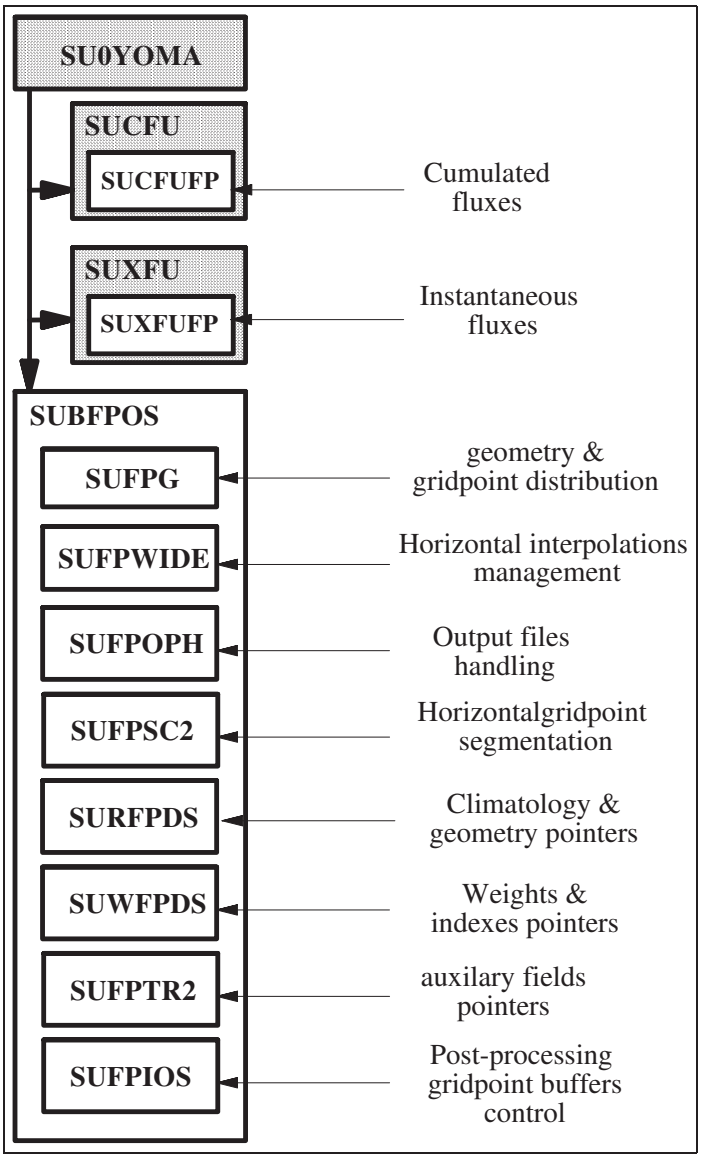


Figure 2.4 *Setup, second part: SUOYOMB. The greyish areas correspond to model subroutines.*

- 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.

2.1.3 Initializations

Figure 2.5 on page 220 shows the scheme of this subroutine.

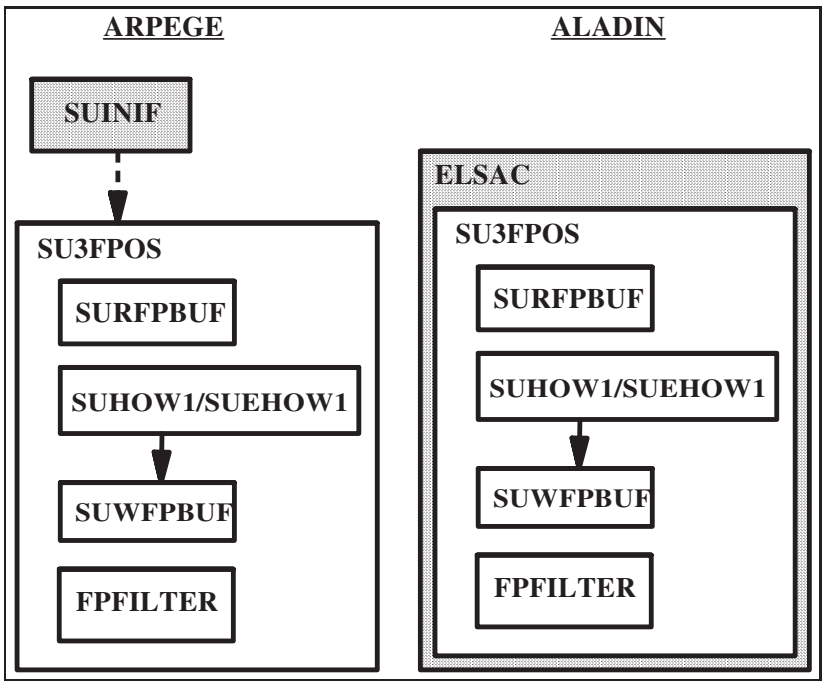


Figure 2.5 *Initializations: CNT3. The greyish areas correspond to model subroutines.*

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?

2.1.4 Temporal loop

Figure 2.6 on page 221 shows the scheme of this subroutine.

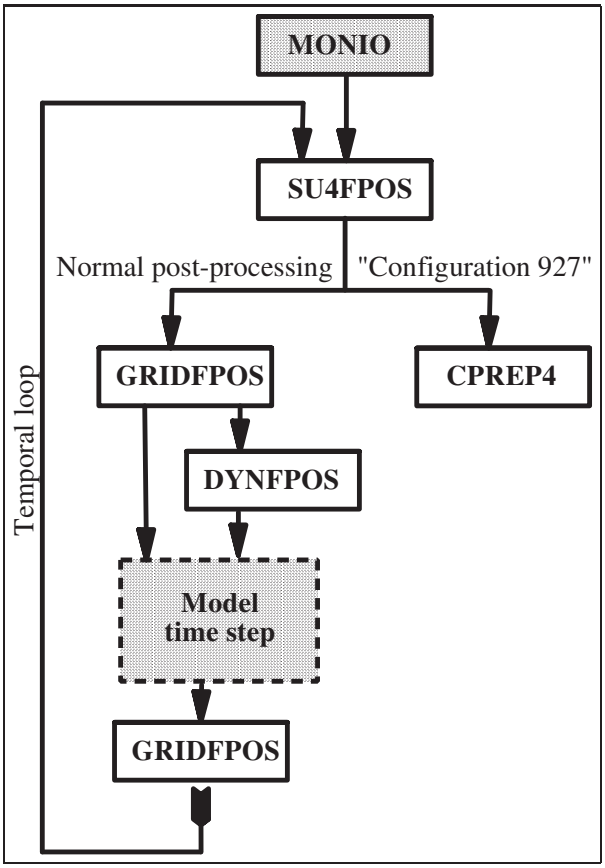


Figure 2.6 *Temporal loop: CNT4. The greyish areas correspond to model subroutines.*

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.

2.1.5 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 2.7 on page 223 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).

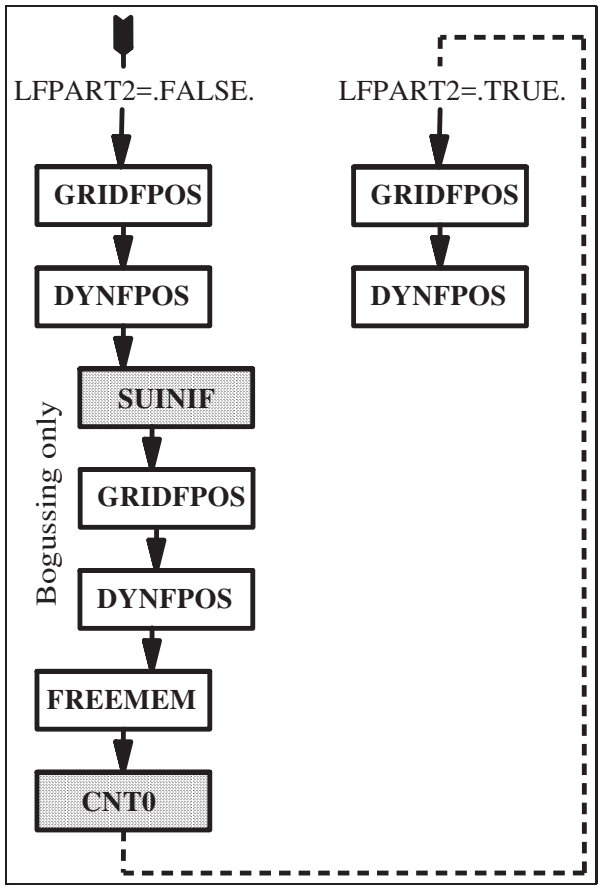


Figure 2.7 *Control of the configurations 927: CPREP4. The greyish areas correspond to model subroutines.*

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).

2.1.6 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 2.8 on page 224 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.

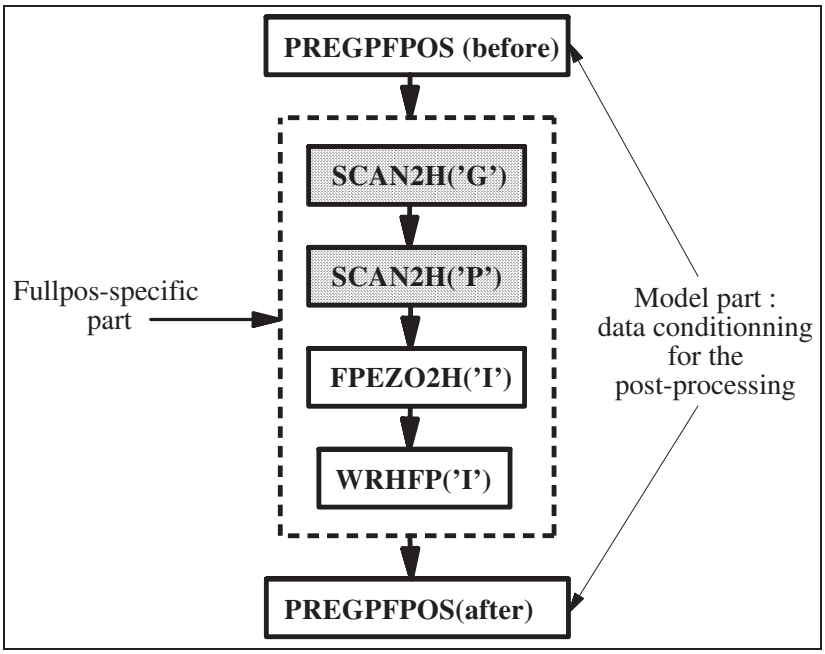


Figure 2.8 *Physical fields post-processing: GRIDFPOS. The greyish areas correspond to model subroutines.*

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.

2.1.7 Dynamic and physico-dynamic post-processing

Figure 2.9 on page 225 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:

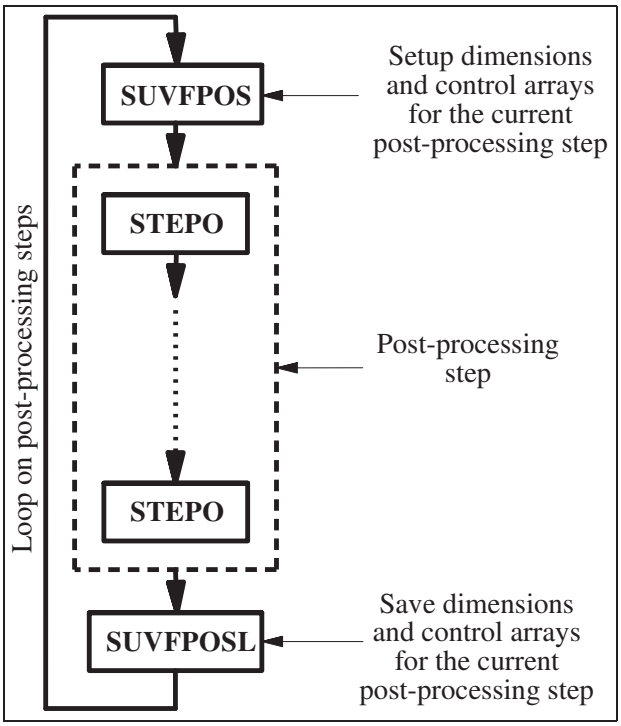


Figure 2.9 *Dynamical and physico-dynamical post-processing: DYNFPOS.*

(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 2.1 on page 227 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 2.2 on page 228. 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 2.10 on page 229 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 2.1 *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.

2.1.8 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 2.2 *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 2.11 on page 230 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 2.12 on page 231, contains the following elements:

SC2RDGFP : To extract data from a "fullpos buffer". Three of them are used here:

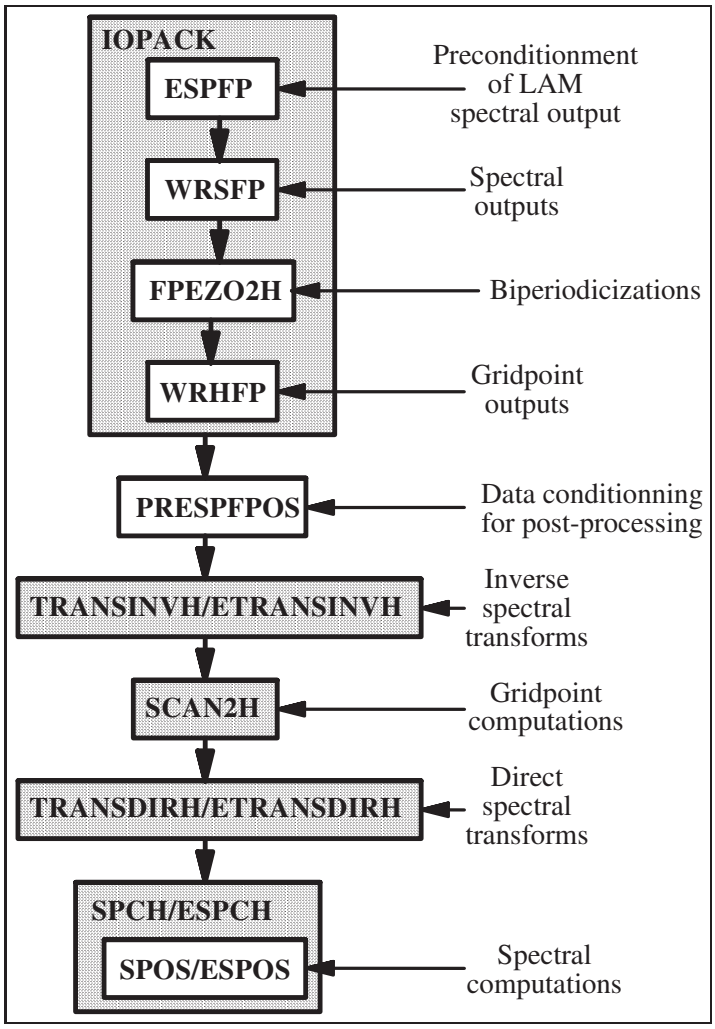


Figure 2.10 *Elements of STEPO used for the post-processing. The greyish areas correspond to model subroutines.*

(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.

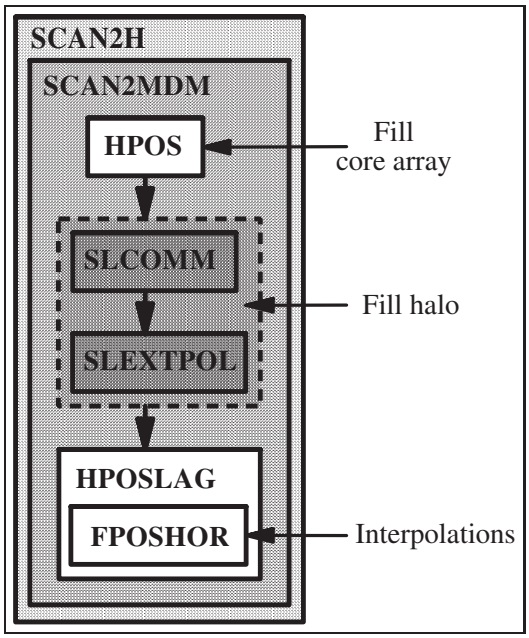


Figure 2.11 *General mechanism of horizontal interpolations. The greyish areas correspond to model subroutines.*

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 2.13 on page 231, 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 2.14 on page 232, 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

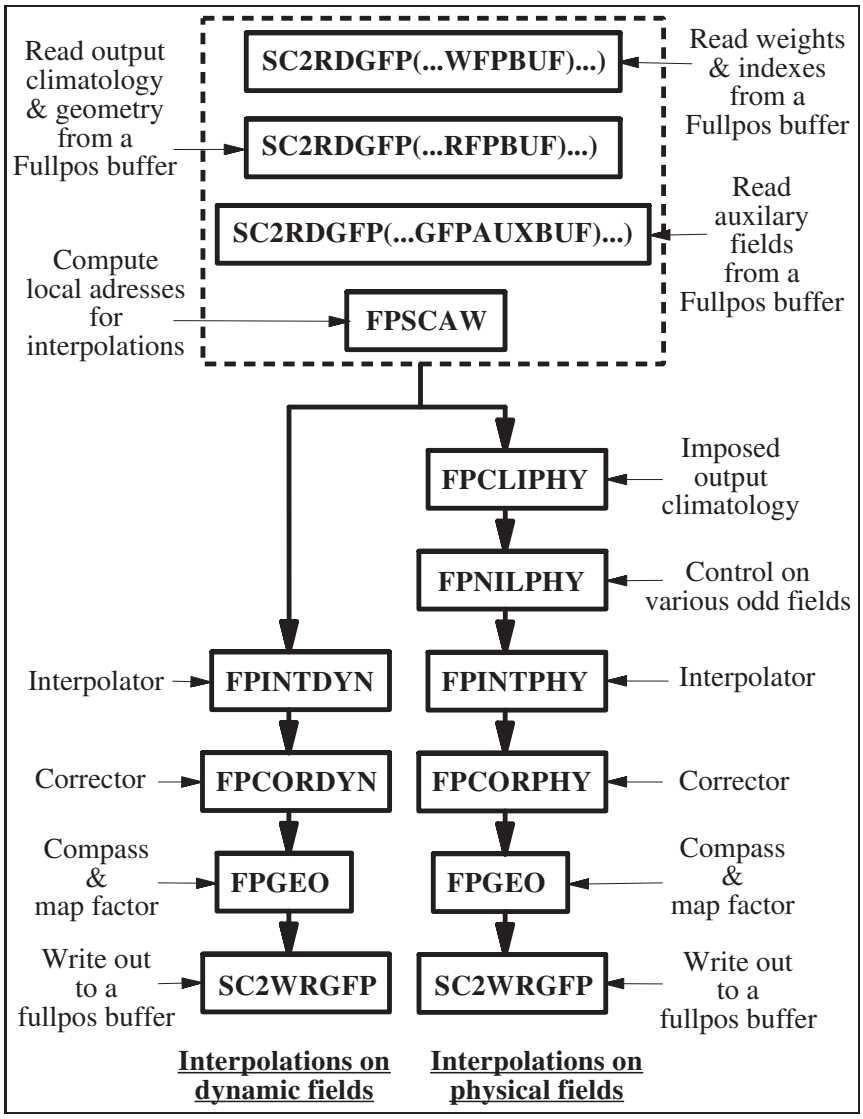


Figure 2.12 *Horizontal interpolations management: FPOSHOR.*

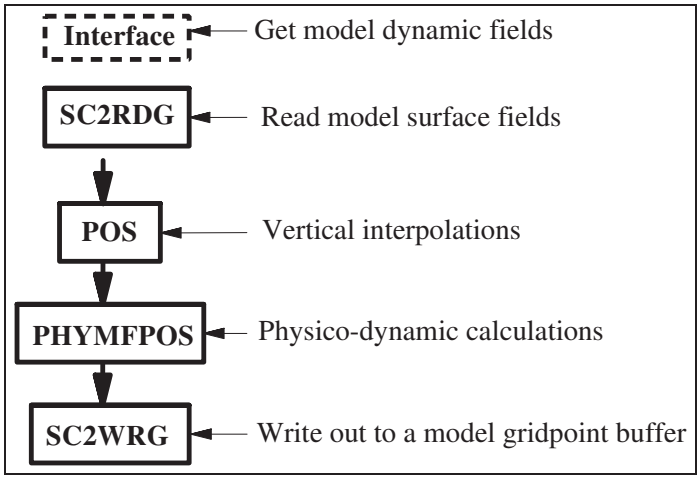


Figure 2.13 *Vertical interpolations and physico-dynamic calculations on input grid: VPOS.*

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.

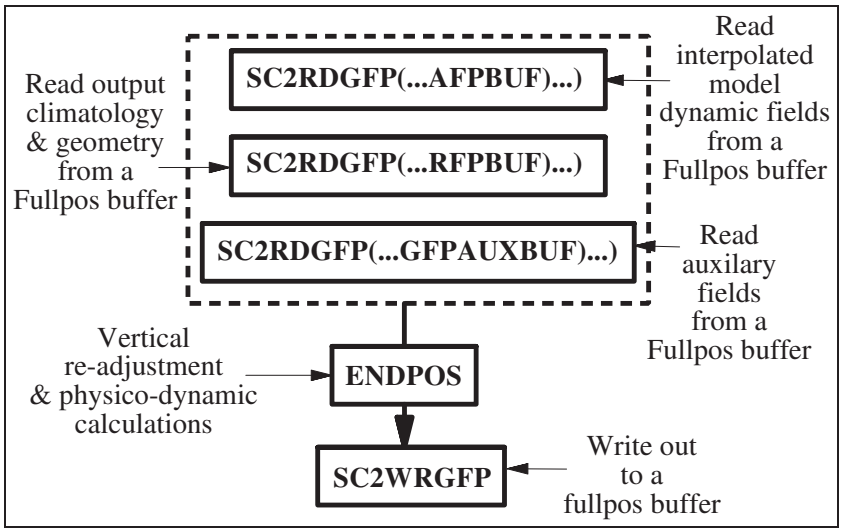


Figure 2.14 *Re-adjustments and physico-dynamic calculations on output grids: ENDVPOS.*

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 2.15 on page 233 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.

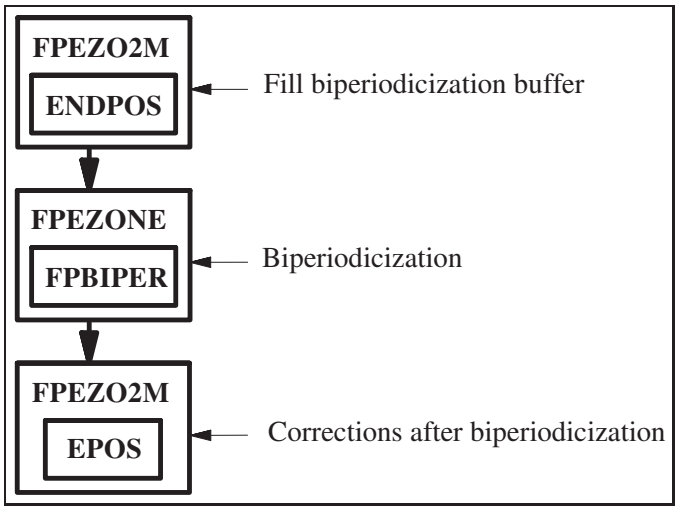


Figure 2.15 *Biperiodicization: FPEZO2H.*

2.2 Data flow

This section will describe the principal data arrays used in FULLPOS.

2.2.1 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 2.16 on page 234.)

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.

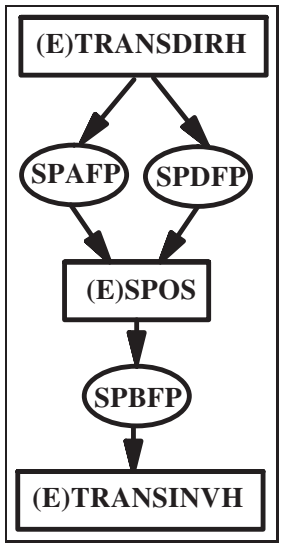


Figure 2.16 *Spectral post-processing data flow.*

2.2.2 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 2.17 on page 235 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.

2.2.3 Fullpos buffers

By "Fullpos buffers" we shall point out the gridpoint buffers specially designed for the post-processing output.

How are they shaped?

Figure 2.18 on page 236 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.

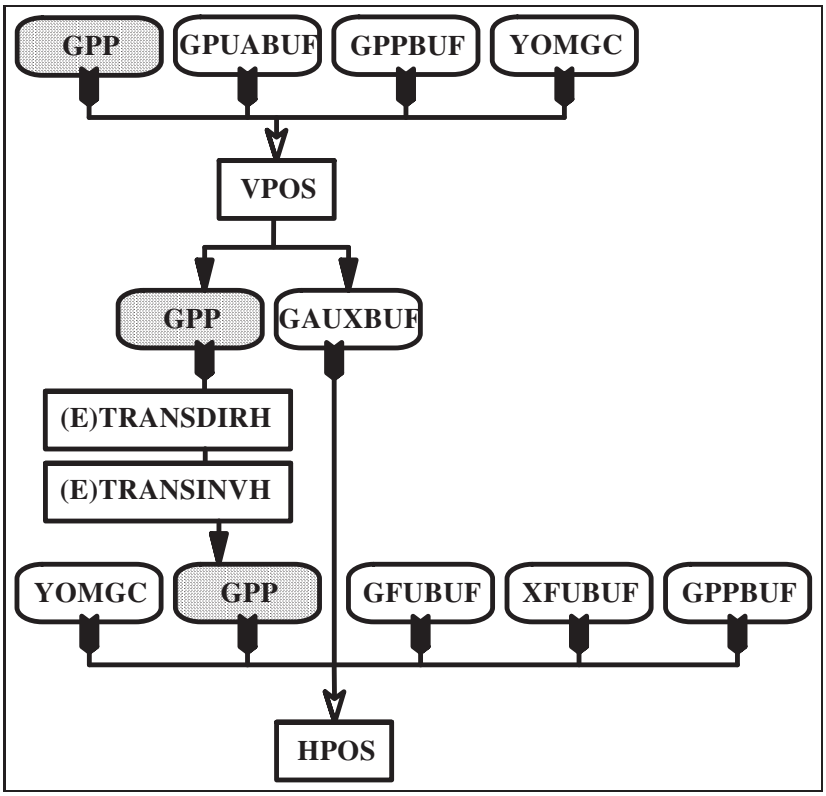


Figure 2.17 *Gridpoint buffers interactions.*

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 2.19 on page 236.

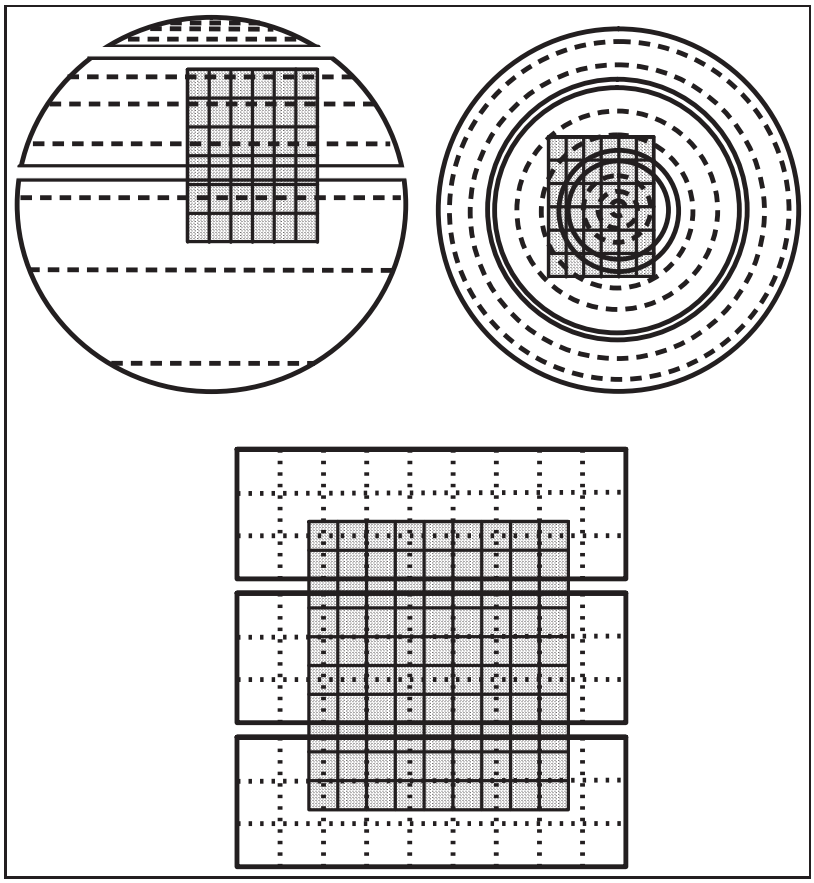


Figure 2.18 *Gridpoints repartitions between model and output grids: the greyish areas corresponds to an output grid; the background areas corresponds to the input model, with a physical separation between processors. Top left represents a typical coupling from a stretched grid, Top right a typical coupling from a rotated grid, bottom a typical nesting.*

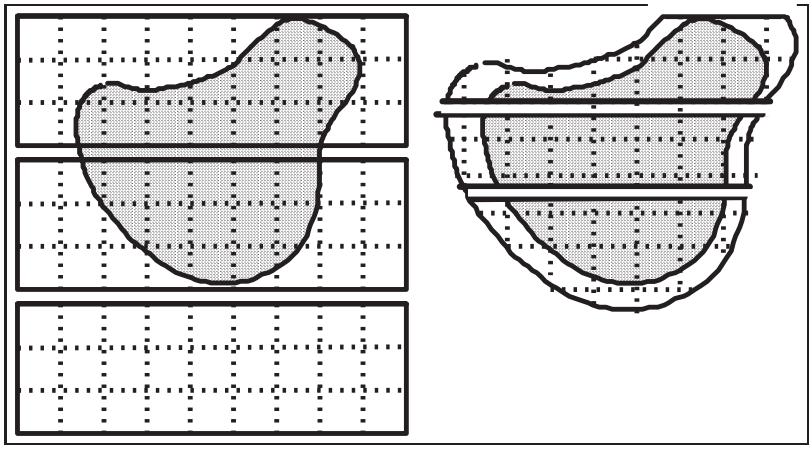


Figure 2.19 *"Next generation" fullpos horizontal interpolator: the cores and halos are re-built to fit the post-processing distribution.*

As shown in Figure 2.20 on page 237, 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:

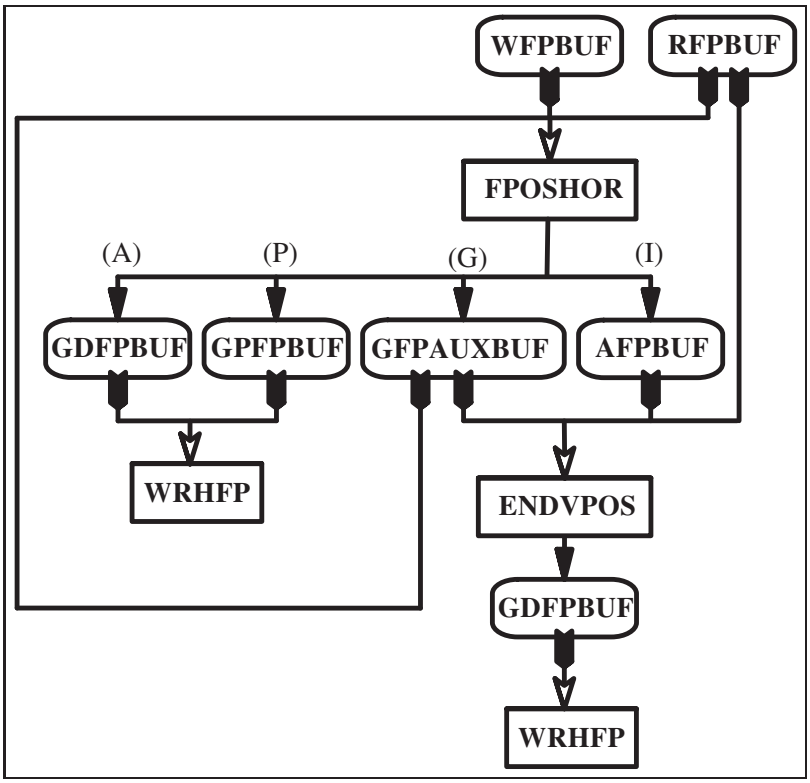


Figure 2.20 *Fullpos buffers interactions in the horizontal interpolations.*

`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 2.21 on page 238, 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`).

2.3 Monitoring

This section will describe how the post-processed fields are monitored through the software.

2.3.1 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:

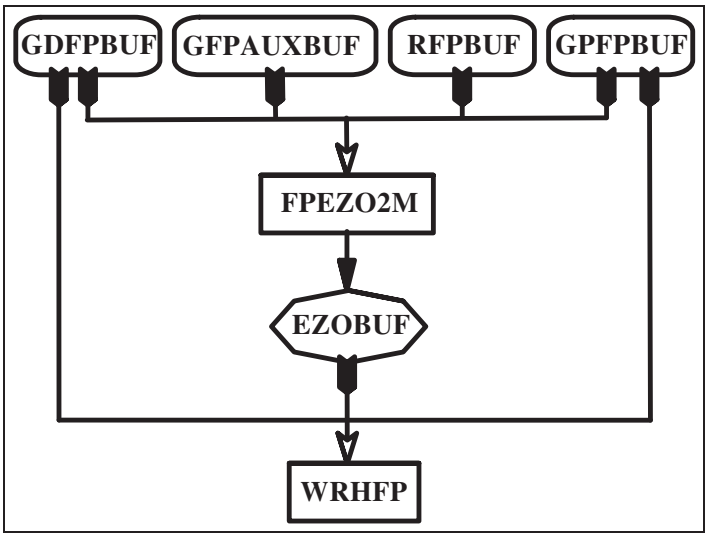


Figure 2.21 *Fullpos buffers interactions in the biperiodicizations.*

- 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.

2.3.2 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.

References