Objectives:

1) Simplify the calling instructions as much as possible
2) Speed-up the performances and minimize the memory requirement
3) IO format specifications must appear only in one unique module
4) Easy switch from one IO library to another
5) Improve the code readability as much as possible
6) Merge *histdef* and *histwrite* in one unique call
7) Temporal mean done in OPA, no more in IO library
8) Distribute the write instructions in each module, i.e. where the variables are available
9) On-line interpolation when reading (ORCA05 data forcing ORCA1/12)

➡ Creation of a new module: IOM (Input/Ouputs Manager)

IOM is able to work with 3 IO libraries:
- jpioipsl : IOIPSL (only its new module fliocom) for NetCDF files
- jpnf90 : Native F90 NetCDF library
- jprstdimg: Fortran Direct Access for "dimg like" restart file

Error handling in IOM: Following OPA philosophy, it has been decided that a detected error will not force the model to STOP but will only write an error message in "numout" (followed by a call flush) and add 1 to *nstop* variable. In addition, when an error is detected, we try to bypass all specific call to the I/O library in order to avoid as much as possible additional errors and stops that would occurs when accessing the data.

IOM modules

iom_def is used to define all parameters and shared variables among iom* modules.
iom is the main module that contains all the work that has to be done independently of the IO library (this module do not contain any "USE" of IO libraries). For example: size check of domains, files, variables… According to chosen IO library, iom will call the appropriate module to open, read, write… Iom is the only module that has to be used within the other parts of NEMO code.
iom_nf90, **iom_ioipsl** and **iom_rstdimg** are the 3 modules specific to the IO libraries. "USE" of IO libraries are located only in theses modules that perform open, read write… Introducing a new IO library will be done by :
- creating a new module similar to the 3 modules already existing
- adding the appropriate calls in iom.F90

No other modifications should be done within the code. In consequence, adding/changing IO library will not affect the way IO commands are written in the code.

IOM_OPEN : Subroutine, opens the file and get back a file identifier

IOM_OPEN(cdname, kiomid, *ldwrt, kdom, kiolib, ldstop*)
```
CHARACTER(len=*), INTENT(in   )           :: cdname ! File name
INTEGER         , INTENT(  out)           :: kiomid ! iom identifier of the opened file
LOGICAL         , INTENT(in   ), OPTIONAL :: ldwrt  ! open in write mode (default = .FALSE.)
INTEGER         , INTENT(in   ), OPTIONAL :: kdom   ! domain on which we write the file
                                                    ! (default = jpdom_local_noovlap)
INTEGER         , INTENT(in   ), OPTIONAL :: kiolib ! IO library used (default = jpnf90)
LOGICAL         , INTENT(in   ), OPTIONAL :: ldstop ! stop if open to read non-existing file
                                                    ! (default = .TRUE.)
```

could have been a function instead of a subroutine…

- IOM_OPEN opens the file in read-only(read-write) mode if ldwrt = F(T). By default ldwrt = F. In read-only mode, if the file does not exist, iom_open returns 0 in kiomid. In addition, if ldstop = T (default), iom_open will print a STOP message and add 1 to *nstop* variable. With ldstop = F, iom_open can be used to test if a file exists or not. In write mode, the file can be new or old.
- If needed, the file name is automatically completed by:
  - a suffix according to the value of kiolib (".dimg" if kiolib = jprstdimg and ".nc" in other cases)
  - the cpu number (starting at 1 if kiolib = jprstdimg and 0 in other cases). The maximum number of digits that can be used to write the cpu number is jpmax_digits.

  For example, if cdname is defined as "toto" and kiolib = jpnf90, iom_open will test the existence of toto.nc, toto_x.nc, toto_xx.nc, toto_xxx.nc … with x the cpu number. If cdname is defined as "toto.nc", iom_open is also able to look for toto_x.nc, toto_xx.nc, … It will not try "toto.nc.nc" but if kiolib = jprstdimg it will try "toto.nc.dimg", "toto.nc_x.dimg"… So it is safer to avoid the suffix of file name when calling iom_open.
- Note that the output kiomid is not the logical unit associated to the file. It is an identifier defined and used by iom to refer to the file. Therefore this identifier must be used only through iom subroutines. For example the command CLOSE(kiomid) will crash. Note that the maximum value of kiomid is fixed by the parameter jpmax_files.
- Each opened file is associated to a structure used as a file descriptor. This structure is the element kiomid of the array iom_file (defined in iom_def). There is its description:

```
TYPE, PUBLIC ::   file_descriptor
   CHARACTER(LEN=240)                        :: name   !: file name
   INTEGER                                   :: nfid   !: file identifier (0 if closed)
   INTEGER                                   :: iolib  !: library used to read the file
   INTEGER                                   :: nvars  !: number of identified varibles
   INTEGER                                   :: iduld  !: id of the unlimited dimension
   INTEGER                                   :: irec   !: writing record position
   CHARACTER(LEN=16), DIMENSION(jpmax_vars)  :: cn_var !: variables name
   INTEGER, DIMENSION(jpmax_vars)            :: nvid   !: variables Id
   INTEGER, DIMENSION(jpmax_vars)            :: ndims  !: nb of dimensions of each variables
   LOGICAL, DIMENSION(jpmax_vars)            :: luld   !: variables using unlimited dimension?
   INTEGER, DIMENSION(jpmax_dims,jpmax_vars) :: dimsz  !: dimensions size of variables
   REAL(kind=wp), DIMENSION(jpmax_vars)      :: scf    !: scale_factor of the variables
   REAL(kind=wp), DIMENSION(jpmax_vars)      :: ofs    !: add_offset of the variables
END TYPE file_descriptor
```

IOM_OPEN : Subroutine, close the file and free its Id

IOM_CLOSE(*kiomid*)

```
INTEGER, INTENT(in), OPTIONAL ::   kiomid   ! iom file identifier
```

Close the file and free the structure associated to kiomid by doing a simple:

```
          iom_file(kiomid)%nfid = 0
```

- if kiomid is not provided, iom_close close and free all opened files.
- if kiomid <= 0 nothing is done
- For jprstdimg files opened in read-write mode, the file header is written by iom_close just before closing the file.

IOM_GET : Subroutine, read 0/1/2/3D array, one time step at once

```
IOM_GET( kiomid, kdom, cdvar, pvar, ktime, kstart, kcount )
INTEGER,                      INTENT(in )          ::   kiomid  ! iom file identifier
INTEGER,                      INTENT(in )          ::   kdom    ! domain on which is written
                                                                ! the file
CHARACTER(len=*),             INTENT(in )          ::   cdvar   ! variable name
Through interfaces, pvar must suit one of the following definition
REAL(wp),                   , INTENT(out), OPTIONAL ::   pv_r0d  ! read field (0D case)
REAL(wp), DIMENSION(:)      , INTENT(out), OPTIONAL ::   pv_r1d  ! read field (1D case)
REAL(wp), DIMENSION(:,:)    , INTENT(out), OPTIONAL ::   pv_r2d  ! read field (2D case)
REAL(wp), DIMENSION(:,:,:)  , INTENT(out), OPTIONAL ::   pv_r3d  ! read field (3D case)
INTEGER                     , INTENT(in ), OPTIONAL ::   ktime   ! record number (default = 1)
INTEGER , DIMENSION(:)      , INTENT(in ), OPTIONAL ::   kstart  ! reading start position
INTEGER , DIMENSION(:)      , INTENT(in ), OPTIONAL ::   kcount  ! number of points to read
```

Note: online interpolations are not yet implemented.

Limitations and rules of iom_get:

To keep code readability and avoid writing an I/O module longer than OPA itself, we fixed some rules to limit the possible cases when reading data. Some of the points concern only NetCDF files. With jprstdimg library, files contain only one temporal record, they are therefore not concerned by all calendar stories!

1) iom_get read 1D, 2D and 3D arrays from data having or not a time axis
2) iom_get can read only one time step at once
3) iom_get do not read/use any calendar. The temporal record number to be read is specified by the user.
4) when reading a nD array, with n =1, 2, 3, the data must be stored in the file as a nD or (n+1)D array if the data have or not a time axis. ⇨ degenerated spatial dimensions with a length of 1 are not accepted. You can suppress them very easily with the nco command ncwa –a. However, for historical reasons (read the old coordinates and bathymetry files), we made an exception to this rule and we do accept to read a 2D array that have degenerated vertical and temporal dimensions.
5) the name of the spatial dimensions is not used. We assume that within the array describing the data, they are always defined (if existing) in the order x, y, z. The time dimension (if existing) is always the last one. Note that in the file itself, the dimensions can be defined in any order.
6) if the data has a time dimension, it must be the "unlimited" one otherwise it will be mix-up with a spatial dimension (problem for xyt / xyz arrays for example)
7) you must define kdom as jpdom_unknown and specify kstart and kcount (see below) when reading a 1D array or a 2D (3D) array that does not correspond to the full xy (xyz) domain.
8) What ever the format used in the file (R8 (double), R4 (float), I4 (int), I2 (short), I1 (byte)), the type of the output array is always REAL(wp). If the attributes add_offset and scale_factor are present they will be automatically used.
9) If kdom /= jpdom_unknown (see below) output array of iom_get has always the size jpi(j) along x(y) axis and jpk or jpkdta along z axis.

Note on point (4): The original idea was to make iom_get as user friendly as possible. Therefore we started to code all possible cases with/without degenerated dimensions. But we ended up with a version of iom_get that was several pages long and no more readable… That's why we decided to accept only a limited number of cases. Accepting only 3(+1)D arrays would have been nice because even vertical sections or a profiles have a position in the

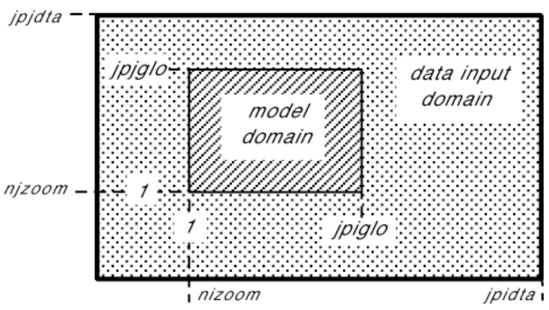
3 space directions. In addition tools like ncks keep degenerated dimension when extracting a hyperslab. However, it is very easy to remove degenerated dimensions with ncwa –a, but it was more difficult to add degenerated dimension into a file. That’s why we decided point (4).
Note on point (6): see http://nco.sourceforge.net/nco.html#dmn_rcd_mk
There is the nco commands to convert your time dimension called time_counter from a fixed dimension to a unlimited dimension.

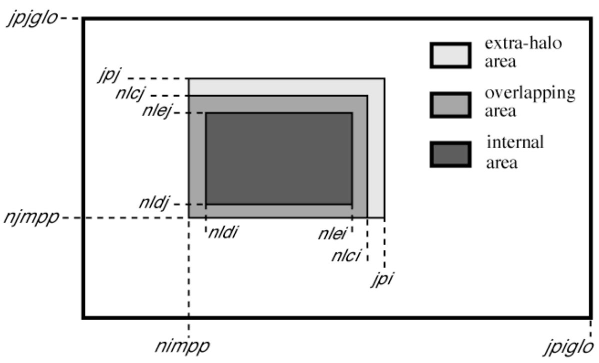
```
ncecat -O in.nc out.nc                          # Add degenerate record dimension named "record"
ncpdq -O -a time_counter,record out.nc out.nc   # Switch "record" and "time"
ncwa -O -a record out.nc out.nc                 # Average out degenerate "record"
```

Start and count definition in iom_get

By default (if kstart and kcount are not specified), iom_get will compute the starting point and the number of point to read in each direction. To do this, iom_get needs to know on which domain are written the data. There is the domain list and their definitions:

-	jpdom_data	➪ (1 :jpidta, 1 :jpjdta)
-	jpdom_global	➪ (1 :jpiglo, 1 :jpjglo)
-	jpdom_local	➪ One of the 3 following cases
-	jpdom_local_full	➪ (1 :jpi , 1 :jpi)
-	jpdom_local_noextra	➪ (1 :nlci , 1 :nlcj)
-	jpdom_local_noovlap	➪ (nldi:nlei ,nldj:nlej)
-	jpdom_unknown	➪ No dimension checking
-	jpdom_autoglo	➪ = jpdom_local if there is only figures between the last “_” and the last “.” found in the filename. = jpdom_global in other cases
-	jpdom_autodta	➪ = jpdom_local if there is only figures between the last “_” and the last “.” found in the filename. = jpdom_data in other cases





Note that :
- dta and glo domains differ only if you use the zoom functionality
- glo and “locals” domains differ only if you use MPI domain decomposition

If kdom /= jpdom_unknown, data are read only on the “local no overlap” domain: from nldi(j) to nlei(j) for the x(y) axis. As output of iom_get must have the size of the “local full” domain (jpi(j) for x(y) axis), we call lbc_lnk (with the optional parameter cd_mpp) to fill the gaps between “local no overlap” and “local full” domains. This will:
- make a communication to fill the overlapping areas: from 1 to nldi(j) and from nlei(j) to nlci(j)

- fill the extra-halo, from nlci(j) to jpi(j), with the values contained in the last line(column) of the “local no overlap” domain (nlej(i))

When using a very large number of cpus, reading only the inner part of the domain can significantly reduce the amount of read data (-50% for ORCA05 on 255 cpus!) and speed up I/O as communications between cpus are much faster than disk access.
Note that this call to lbc_lnk does not take care of the periodicity or the north fold boundary. The user must check himself that the input data is correct.

If kdom == jpdom_unknown and kstart and kcount are not specified, we read all the data along spatial directions.

If the user specifies kstart and kcount, he must define kdom as jpdom_unknown

Error checking in iom_get

iom_get check :
- if the variable cdname exists in the file
- that data has exactly n spatial dimensions when reading a nD array
- that we do not try to read more data than existing (start + count – 1 <= size of the data)
- the shape and size of the output array correspond to shape and size of the data to read

IOM_VARID : Function, returns the id of the variable cdvar

IOM_VARID (kiomid, cdvar, *kdimsz, ldstop*)
```
INTEGER               , INTENT(in   )           :: kiomid   ! file Identifier
CHARACTER(len=*)      , INTENT(in   )           :: cdvar    ! name of the variable
INTEGER, DIMENSION(:), INTENT(  out), OPTIONAL :: kdimsz   ! size of the dimensions
LOGICAL               , INTENT(in   ), OPTIONAL :: ldstop   ! stop if looking for non-existing
                                                            ! variable (default = .TRUE.)
```

Returns the id of the variable cdvar contained in the file having for identifier kiomid. If the variable was not found it returns -1 (and 0 if another error occurs). If ldstop = T (default), iom_varid will print a STOP message and add 1 to *nstop* variable. With ldstop = F, iom_varid can be used to test if a variable exists or not in a file.
Add the optional input kdimsz to get back the size of the variable dimensions. kdimsz must have the same number of elements than the number of dimensions of the variable.

IOM_GETTIME : Subroutine, read the time axis

IOM_GETTIME(kiomid, cdvar, ptime)
```
INTEGER               , INTENT(in)  ::   kiomid   ! Identifier of the file to be closed
CHARACTER(len=*)      , INTENT(in)  ::   cdvar    ! time axis name
REAL(wp), DIMENSION(:), INTENT(out) ::   ptime    ! the time axis
```

This routine works only with NetCDF files. Read the time axis called cdvar in the file having for identifier kiomid. The variable cdvar must have only 1 dimension and this dimension must be unlimited. All the elements of cdvar are read, ptime must thus have the same size as the array stored in cdvar.
Note that as iom_get can read only one time-step at once, therefore it cannot be used to read the time axis (except by doing a do loop).

IOM_RSTPUT : Subroutine, write 0/1/2/3D array into a restart file

IOM_RSTPUT(kt, kwrite, kiomid, cdvar, pvar, *ktype*)

```
INTEGER                         , INTENT(in)           ::   kt        ! ocean time-step
INTEGER                         , INTENT(in)           ::   kwrite    ! writing time-step
INTEGER                         , INTENT(in)           ::   kiomid    ! Identifier of the file
CHARACTER(len=*)                , INTENT(in)           ::   cdvar     ! variable name
```

Through interfaces, pvar must suit one of the following definition

```
REAL(wp)                        , INTENT(in), OPTIONAL ::   pv_r0d    ! written 0d field
REAL(wp), DIMENSION(        jpk), INTENT(in), OPTIONAL ::   pv_r1d    ! written 1d field
REAL(wp), DIMENSION(jpi,jpj    ), INTENT(in), OPTIONAL ::   pv_r2d    ! written 2d field
REAL(wp), DIMENSION(jpi,jpj,jpk), INTENT(in), OPTIONAL ::   pv_r3d    ! written 3d field
INTEGER                         , INTENT(in), OPTIONAL ::   ktype     ! variable external type
```

This routine is a first step of writing data with iom. Restarts have been chosen because it is easier work! It is instantaneous variables with input files of fixed dimension (jpi,jpj,jpk). The choice of the variables to be written is driven by the model physics/configuration and user has only to control the output frequency.

- For NetCDF format, the file definition is done the first time this routine is called but writing of the data is done only when kt = kwrite. In a practical point of view, we call iom_rstput twice: (1) at kt = kwrite -1 to define the header of the NetCDF file and (2) at kt = kwrite to write the data itself. A test is done to know if the dimension and variables are already define or not. For exemple, if you can call iom_rstput at every time step, at kt = nit000 the file header is defined, at kt = kwrite the data are written and in between noting is done (except the test to know if the header definition has already been done or not). Note that it is possible to define and write the variables at the same time but performances will be very bad (as before with restput).
- Default variable external type is jp_r8 corresponding to REAL(8). For NetCDF format, others type are available: jp_r4, jp_i4, jp_i2 and jp_i1. Note that on vector computer jp_i2 and jp_i1 are not vectorised (but are smaller to write…).
- In order to keep only 3 spatial dimensions in the file, we do not accept 1/2/3d variables with other dimensions than jpk/(jpi,jpj)/(jpi,jpj,jpk). For example in the ice, the variable moment (created to speed up the old restput) with a third dimension of 35 has been split in 35 variables. zinfo 1D variables have been also split in several scalar variables.
- In NetCDF files, a time dimension with a size of 1 as been put in the restart file. It could be useful for some analyses or post-treatment (??). Its value is the time-step at which the file is written (is is not really usefull!).
- The domain on which the file is written is controlled by the input parameter kdom of iom_open.
- This routine is also used to write mesh files that contains instantaneous outputs with fixed size and no interaction from the user.
- In future this routine may change to handle obc restarts…

Annex 1: New headers for restart "dimg like" format (jprstdimg)

The data to be stored in the file header can easily exceed the record size (that corresponds to the horizontal domain) when a large number of cpu are used. To avoid the problems in a very large majority of cases, we decided to split the "header of these "dimg like files" in two records. The first and the last records. They contain:

```
WRITE( idrst, REC = 1, IOSTAT = ios, ERR = 987 )   &
           &   irecl8, inx, iny, inz, in0d, in1d, in2d, in3d, irhd,   &
           &   jpni, jpnj, jpnij, narea, jpiglo, jpjglo,   &
           &   nlcit, nlcjt, nldit, nldjt, nleit, nlejt, nimppt, njmppt
and
WRITE( idrst, REC = irhd, IOSTAT = ios, ERR = 987 )    &
```

```
           &   clna0d(1:in0d), zval0d(1:in0d), clna1d(1:in1d), zval1d(1:in1d),   &
           &   clna2d(1:in2d), zval2d(1:in2d), clna3d(1:in3d), zval3d(1:in3d)
```

With:

- irecl8 (integer(4)): record size
- inx,iny,inz (integer(4)): domain size in each direction
- in0d, in1d, in2d, in3d (integer(4)): the number of 0/1/2/3D variables
- irhd: the number of the last record that contains the second part of the header
- other domain and subdomain variables (same as before).
- clna0/1/2/3d (charactere(len=32)): the name of 0/1/2/3D variables
- zval0d (real(8)): the value of the 0D variables
- zval1/2/3d (real(8)): the record position to read the 1/2/3D variables

- to avoid singular case, if the file do not contain anay Nd variables (wth N=0/1/2/3), we force : inNd = 1 with clnaNd(1) = ‘noNd’ and zvalNd(1) = -1.0
- 1d arrays are stored in a record of the size of a 2d array.
- 3d arrays are stored on jpk consecutives records.