Proposal for a new internal observational data structure in the HIRLAM variational data assimilation code

Kristian S. Mogensen

June 2001

Abstract

This report discusses a proposal for changing the current internal observational data structure in the HIRLAM variational data assimilation code. The internal observational data structure is the way in which the information from observations is stored in memory during the variational assimilation runs.

The current observational data structure is reviewed and problems with the current code is identified as a starting point for the presentation.

The main idea is to store data in allocatable arrays contained in Fortran-90 modules. A detailed discussion of various possibilities using this idea is presented. The recommendation is to store similar information (*e.g.* station longitude) from all stations of the same observation type in one dimensional allocatable arrays contained in Fortran-90 modules. The implications of implementing the new proposal in the current code are discussed and a suggestion for a road map on how to proceed with the implementation is presented.

1 Introduction

At the HIRLAM all-staff meeting 2001 there was a discussion group concerning how to handle observational data in the future. The main point of discussion was if we should switch from the currently used central memory array (CMA) data format (White, 2000) originating from European Centre for Medium-Range Weather Forecasts (ECMWF) to the newly implemented (also by ECMWF) observation database (ODB) format (Saarinen, 1998). It was agreed that the current practice in the HIRLAM variational data assimilation system of using CMA both as an external data format to store data in files and as an internal data structure (in terms of an array containing the CMA data) would make the transition from CMA to ODB a large task. Even though the decision about which data format to use in the future was postponed, it was agreed that we should make the transition to another data format easier by changing the internal data structure. CMA should be kept as the external data format until we choose another file format for observations.

This report describes the ideas presented at the all-staff meeting for a new internal data structure in the HIRLAM variational assimilation code (denoted HIRVDA in the rest of the report) in some detail. It should also be seen as an attempt to document the ideas before the actual coding starts.

2 Observational information in variational assimilation

Let us first discuss what kind of information from observations we need to store when we do variational assimilation. Variational assimilation based on minimization of a cost function which can be written in the following form:

$$J = J_B + J_o = \frac{1}{2}\delta\mathbf{x}^T\mathbf{B}^{-1}\delta\mathbf{x} + \frac{1}{2}\left(H\mathbf{x^b} + \mathbf{H}\delta\mathbf{x} - \mathbf{y}\right)^T\mathbf{R}^{-1}\left(H\mathbf{x^b} + \mathbf{H}\delta\mathbf{x} - \mathbf{y}\right) \quad (1)$$

where $\mathbf{x^b}$ is the background state (the first guess), $\delta\mathbf{x} = \mathbf{x} - \mathbf{x^b}$ is the analysis increment in form of the difference between the model state ($\mathbf{x}$) and the background state, $\mathbf{B}$ is the background error covariance matrix, H is the observation operator, $\mathbf{H}$ is the linearized observation operator, $\mathbf{y}$ is the observation vector, $\mathbf{R}$ is the observation error covariance matrix, and $()^T$ denotes the adjoint. During the minimization we try to find the vector $\delta\mathbf{x}$ which gives the minimum value of the cost function. This vector is then added to the background state to give the result of the analysis.

To calculate the cost function we need the observed quantities ($\mathbf{y}$), the observation error covariances ($\mathbf{R}$) and the information from the observations needed to calculate the $\mathbf{H}\delta\mathbf{x}$ and $H\mathbf{x^b}$ terms such as position and station height.

Since observations are not perfect we also need to be able to distinguish which of our input observations we actually want to use in our $\mathbf{y}$ observation vector and which observations to disregard due to errors or too many observations within an area (redundancy). This process is called screening of observations which sets flags based on various checks.

In summary we need the following information to do the analyses: observed values, information about observation errors, station information and screening flags. In addition it might be beneficial to store information which is constant during the minimization, such as $H\mathbf{x^b}$.

3 Current internal data structure

The current internal data structure in the HIRVDA code reflects the external data handling, where WMO BUFR type data is converted to the CMA in a preprocessing step before the actual analysis. This step converts the BUFR reports into CMA reports where each report is a one-dimensional array of 64 bit floating-point data where all data in form of integers, floating-points and characters are converted into 64 bit floating-point data. In this conversion integers and characters are stored in the mantissa of the floating-point words. In the CMA output file data is stored report by report in the same order as the input BUFR file.

The handling of observations in the HIRVDA code reflects the fact that the code supports distributed memory computers as described by Gustafsson *et al.* (1999). During the start-up of the variational analysis, memory is allocated for the CMA data in an automatic array on all processing elements with a user-defined size defined in a Fortran namelist. The CMA datafile(s) is read and the reports (still in the CMA data format) distributed so each processing element has close to the same amount of observations of each observation type.

The first step in the variational code is to calculate the first guess minus observations $\left(H\mathbf{x^b} - \mathbf{y}\right)$ since this is a part of the cost function which is constant during minimization. This information is then stored in the CMA array.

The array of CMA data is also used during the screening of observations which results in the CMA array being updated with the screening decisions. After the screening the CMA array is compressed so only the information actually needed by the minimization is stored. All the information from the screening is stored in an additional automatic array of CMA data for later usage.

After the minimization the CMA data, updated with the difference between the analyzed state and the observation $\left(H\mathbf{x^b} + \mathbf{H}\delta\mathbf{x} - \mathbf{y}\right)$ and combined with the CMA data containing screening decisions, are written to a CMA file for additional observation usage diagnostics by external programs.

3.1 Observational data memory access patterns in the current code

In the current code observational data is typically accessed as in the following example taken from the vertical interpolation:

```
      do jh=1,totobs
        if( .not.heighco(jh) ) then
          do jv=1,nolv(jh)
            if( .not.passive(jh,jv) ) then
              jl = index_vint(jh,jv)
              ao_bg(jh,jv) =    a_bg(jh,jl) +
     x             wght_vint(jh,jv)*(a_bg(jh,jl+1)-a_bg(jh,jl))
            endif
          enddo
        endif
      enddo
```

where `ao_bg` is the field at the vertical levels of the observation and `a_bg` is the horizontally interpolated background field at model levels. This data in `ao_bg` is put back into the CMA array after the vertical interpolation. As can be seen from the example, it would be more efficient to store the data directly in the format in which the data is typically accessed and not in CMA. It is essential for performance to avoid having cycles of the following steps:

1. Extract the relevant quantities from the CMA array to additional arrays.
2. Make some calculations on these arrays.
3. Put the results back in the CMA array.

since there will be a lot of non-unit stride memory access during packing and unpacking of CMA data. Non-unit stride memory access may on cache-based systems cause performance problems because not all data loaded into cache are used. Ideally we should always access data element by element.

4 Requirements for new internal data structure in HIRVDA

In the design of a new internal data structure in HIRVDA we need to keep in mind the following:

Independence of external data storage. We want something which can be used both with the current input data in the form of CMA files and some future input data format.

Portability. The coding should be done in a way independent of machine architecture.

Expansion possibilities. If we for some reason need an extra quantity for each observation point it should be possible to implement. In the current CMA based set-up we are currently out of free space to store new information for the observation points.

Efficiency. We want unit stride where possible. This is especially important on cache based machines.

Ease to implementation. We cannot afford to suspend all other HIRVDA development while we implement the new structure.

Ease of expansion for new observation types. The ideas should be so clear that non computer experts can easily figure out how to add a new observation type.

Maintenance. We should be able to maintain the structure afterwards with a reasonable effort.

5 Proposal for an internal data structure in HIRVDA

The current HIRVDA set-up require a Fortran-90 compiler to compile the whole set-up. It is therefore natural to investigate whether there is any part of the Fortran-90 language which can be used to fulfill the requirements described above. By only using standard Fortran-90 constructions the code will be portable.

5.1 Discussion of different possibilities offered by Fortran-90 constructs

As a starting point we should realize two important points when handling observational data within an assimilation system:

1. the data is used many places in the code.
2. the number of observations is different from one assimilation cycle to the next.

A way to take into account these two points is to store the data in *allocatable arrays* in Fortran-90 *modules*. A Fortran-90 module can be thought of as global data much in the same way as a *common block* but with the very important difference that a module can contain dynamical allocatable data in the form of *pointers* and *allocatable arrays*.

If we choose the "allocatable arrays in modules" approach the next question is in what data types we store the data. There are (at least) two ways of doing it:

1. derived types.

2. standard types.

Fortran-90 has a feature called *derived types* which is similar to *structures* in C and *records* in Pascal. In a derived type we could for instance define a SYNOP type in a synopdef module in the following way:

```
module synopdef
type synop
! Header
integer :: synop_date             ! Obs. date
integer :: synop_time             ! Obs. time
real :: synop_lat                 ! Latitude
real :: synop_lon                 ! Longitude
...
! Observation
real :: synop_fispres             ! Pressure for fis obs
real :: synop_fisobs              ! Observed fis
real :: synop_fisobserr           ! Observed fis error
integer :: synop_fisanaflag       ! Fis analyse flag
integer :: synop_fisstaflag       ! Fis status flag
...
end type synop
end module synopdef
```

after which data can be defined as

```
module synopdata
use synopdef
!
! Bookkeeping
!
integer :: max_synop_number               ! Max no. of Synops
integer :: synop_number                   ! Current no. of Synops
integer,allocatable :: synop_winnumber(:) ! No. in each time slot
!
! Observations
!
type(synop),allocatable :: synops(:)
end module synopdata
```

and allocated with a single allocate statement: `allocate(synops(max_synop_number))` where `max_synop_number` is set by the user.

The above approach is elegant, but has two important drawbacks since

1. accessing *e.g.* the pressure of all geopotential observations from SYNOPS is non-unit stride.

2. the code has to be changed since we can no longer have subroutine calls of the following form:

```
call vint(fispres,...)
```

unless we reshape the data before the call (which is part of what we want to avoid) or change all subroutines to use the new derived types (which is a very large task).

If we only use allocatable arrays of standard types we need modules of the following type:

```
module synopdata
!
! Bookkeeping
!
integer :: max_synop_number                ! Max no. of Synops
integer :: synop_number                    ! Current no. of Synops
integer,allocatable :: synop_winnumber(:)  ! No. in each time slot
!
! Headers
!
integer,allocatable :: synop_date(:)       ! Obs. date
integer,allocatable :: synop_time(:)       ! Obs. time
real,allocatable :: synop_lat(:)           ! Latitude
real,allocatable :: synop_lon(:)           ! Longitude
...
!
! Observations
!
real,allocatable :: synop_fispres(:)       ! Pressure for fis obs
real,allocatable :: synop_fisobs(:)        ! Observed fis
real,allocatable :: synop_fisobserr(:)     ! Observed fis error
integer,allocatable :: synop_fisanaflag(:) ! Fis analyse flag
integer,allocatable :: synop_fisstaflag(:) ! Fis status flag
...
end module synopdata
```

where all data is stored in a collection of one-dimensional arrays. With this approach we get the unit-stride access to *e.g.* the pressure of all geopotential observations and we can keep the current subroutine calls. The major drawback is that we need to allocate all arrays separately.

Since the approach with allocatable arrays in modules is the easiest to implement and has unit stride access I *recommend* that we use this approach.

A small complication with both approaches is how to define screening flags information packed in integer values. I suggest that we keep the current conventions from CMA for the time being for how *e.g.* analysis flags are packed together. Maybe during the implementation we should investigate which flags we actually use in the variational code so we only allocate space for these flags and not for all flags available in CMA.

A quite appealing feature with the collection of arrays approach is that if we need an additional variable we just add an extra allocatable array to the given module and recompile (after making the code corrections for allocation and setting of values of course) after which we can start to use the data. Two very real examples are the logarithm of

the observed pressure (used in vertical interpolation) which is currently being calculated every time it is used and exact anemometer height for the so-called measured 10 meter winds.

The examples above are for SYNOP data and do not contain all the information of real SYNOP BUFR reports. In addition, other data types (such as radiosondes) with multi-level data in a single report need to be handled. These two issues will be discussed in the following.

5.2 Single-level data

If we assume that the modules need to contain all the information currently available in CMA (including additions from HIRLAM development work) we need a module for SYNOP data with a structure as presented in appendix **??**. Modules for other single-level data types such as SHIP, AIREP (include AMDAR and ACARS), DRIBU and SATOB can easily be constructed based on the same principles as the module for SYNOP data.

5.3 Multi-level data

For multi-level data such as radiosondes the storage of observational data is more complicated than for single-level data since the HIRVDA code uses all significant levels of radio-sonde reports. There are (at least) three ways of solving this problem. The first solution is to define data as two dimensional allocatable arrays in the following form (for temperatures in TEMP observations):

```
real,allocatable :: temp_Tobs(:,:)            ! T obs
```

and then allocate memory with `allocate(temp_Tobs(nobs,maxlvls))` where `nobs` is the number of observations and `maxlvls` is the maximum number of levels in any of the reports. Since all reports do not have the same number of levels this is obviously a memory inefficient way of doing this but the access of data is straight forward. A more memory efficient way is to define the data as one dimensional arrays and then allocate the sum of the number of levels in all reports' elements. It is then necessary to have an additional integer array to define where the data from each report starts. For some of the code it is not necessary to actually use this additional array as in the example from section 3.1 which can be rewritten as

```
      do jh=1,totobslvls
        if( .not.heighco(jh) ) then
          if( .not.passive(jh) ) then
            jl = index_vint(jh)
            ao_bg(jh) =    a_bg(jh,jl) +
     x           wght_vint(jh)*(a_bg(jh,jl+1)-a_bg(jh,jl))
          endif
        endif
      enddo
```

which gives unit stride in the access of `ao_bg`, `index_vint` and `wght_vint`. A third way would be to have an array of pointers. *This is not directly supported by Fortran-90* but can be implemented by defining a derived type containing a pointer (Metcalf and Reid,

1998). This will produce complicated code with non-unit stride memory access, so in my opinion we should disregard this idea.

The memory saving suggestion is only slightly more complicated than the two dimensional array version so I *recommend* that we implement the memory saving version. An example of a possible way of defining a Fortran-90 module for TEMP data is given in appendix **??**. This example can easily be modified to handle PILOT, SATEM and ATOVS data.

6 Work needed to implement the Fortran-90 modules approach

When implementing the above described new observational data structure in HIRVDA we should first carefully consider how we put data into the Fortran-90 modules and how to access data in the modules. After the full implementation the observational data stored in modules should be handled in the following steps:

1. Convert input observational data (in CMA now) into data in modules.
2. Make the following calculations on the data in the modules:
   (a) Set flags depending on screening decisions.
   (b) Use the data in the minimization.
3. Convert data from the modules to an output data format and write the converted data to a file. This file is used for observation usage statistics *etc* by external programs.

within the HIRVDA code. Step 2 is were the real conversion work is and will be discussed in more detail.

Currently the data from CMA is converted into local automatic arrays with a somewhat similar structure to this proposal by the subroutines `readobs_[conv,sat,rtm,rad].F` for some of the access to the data but a few subroutines access data directly from CMA. We could in principle change the `readobs_[conv,sat,rtm,rad].F` subroutines to access data in modules instead of CMA as a quick and dirty way of changing the data structure, but in my opinion we will then lose some of the advantages of the new data structure. The direct way of accessing the module data is by inserting `use <obstype>data` in the subroutines which make calculations on `<obstype>` data. This means that we will need to make a subroutine to calculate the gradient of the cost function for all observation types in contrast to the current code where we have a single subroutine for all conventional observations (but separate subroutines for ground-based GPS, radar and ATOVS data). Since the code for having a separate subroutine for each observation type will be more clear without many of the current `if` statements and easy to read compared to the current code I do not consider having more subroutines a problem. My *recommendation* is to use the module data directly.

An intermediate step in the conversion could be to keep using CMA in the screening and then convert the output of the screening in form of CMA data into module data and use the module data in the minimization. When the use of module data in the

minimization works correctly (which can be defined as giving the exact same results as for the original code) we can change the screening code to use modules as well.

For the minimization we can start using module data for one observation type at a time if we keep the current data in the form of CMA available. This will require more storage compared to the final version with no CMA data but can be used to keep the code running at all times. In the development phase we can have a logical variable for each observation type which controls whether the observational data should be taken from CMA or from the modules. When we are satisfied with the module set-up we should remove the code using CMA.

In the current code the CMA data used in the minimization is a compressed part of the full CMA input data with only the data not flagged by the screening. In my opinion we can probably afford not to use compression when we use modules but we might have to reconsider this point depending on the performance of the code. Effect similar to CMA compression can be obtained with more arrays in the module containing only the information needed for the minimization from non-flagged observations. Since we do not need copies of the full sets of data, only a very limited amount of extra storage is required. My *recommendation* is to start without compression but reconsider the decision later.

We can summarize the work needed to be done in the following steps:

1. Define modules for each observation type.
2. Write code to allocate the module data.
3. Write code to convert input data from CMA to module data for each observation type.
4. Write diagnostic code to print out data from the modules. This is mainly for helping the debugging and development of the code.
5. Write code to convert module data back to CMA needed for write-out of observation statistics.
6. Write code to calculate the gradient of the cost function for each observation type using the data in the modules.
7. Update the screening code to use modules.

Steps 1 to 6 can be done in parallel for each observation type. Presently, these steps have been implemented for SYNOP data without much difficulty. Very preliminary tests with the updated version showed a decrease in the computing time from 53 seconds to 42 seconds on a single CPU Linux PC for the time spent in the calculation of the contribution of the SYNOP data to the gradient of the cost function during a single low resolution run with all conventional observations. Since the computer time is short in both the run using data from CMA and the run using data from modules the time saved is not very significant but it shows that the new proposal is more efficient than the original code, as expected.

7 Conclusion

The proposal presented for a new internal observational data structure has some clear advantages compared to the current structure based on CMA. The most important advantages are the dynamical memory management, the unit stride access to data, greater flexibility when adding extra information needed by the assimilation code and independence of external data file format. The latter will make a replacement of the current CMA data file format easy.

The major disadvantage is the fact that we need to spend time to recode the access to observational data in the HIRVDA code, but you cannot make improvements without making changes.

8 Acknowledgements

The input from the participants in the observation handling discussion group at the HIRLAM all-staff meeting 2001 are gratefully acknowledged. I will also like to thank Maryanne Kmit for comments on the manuscript.

References

Gustafsson, Nils, Hörnquist, Sara, Lindskog, Magnus, Berre, Loik, Navascués, Beatriz, Thorsteinsson, Sigurdur, Huang, Xiang-Yu, Mogensen, Kristian S., and Rantakokko, Jarmo. 1999. *Three-dimensional variational data assimilation for a high resolution limited area model (HIRLAM).* Technical Report 40. Hirlam 4 project, c/o Met Éireann, Glasnevin Hill, Dublin 9, Ireland.

Metcalf, Michael, and Reid, John. 1998. *Fortran 90/95 explained.* Oxford: Oxford University Press.

Saarinen, Sami. 1998. A distributed database for observation handling. *In: ECMWF workshop, Towards Teracomputing - The Use of Parallel Processors in Meteorology, Reading, 16-20 November 1998.*

White, Peter W. 2000. *IFS documentation. Part I: Observation processing. CY21R4.* available from *http://www.ecmwf.int/research/ifsdocs/.*