Metview Macro Model-OBS Differences Tutorial

Development Section

ECMWF

01/05/2014

This tutorial was tested with Metview version 4.4.6
but should work for all 4.3.x and 4.4.x versions.

Objective

In this tutorial we compute simple observations-analysis differences and plot them. The exercise covers the conversion of observation data in BUFR format into Metview's geopoint format, combining geopoints data with field data and plotting of scattered data. It then goes on to investigate how to import data stored in ASCII table files to achieve the same result.

The ingredients are :

- a BUFR file containing observations over Europe
- the ECMWF 2m T analysis field for the same day as the observations
- an ASCII file containing observations of 2m T at a number of European locations (used later in the tutorial)

We will take the observations and plot the differences to the analysis field with the following colour scheme:

- observations more than 1 degree warmer than analysis are plotted in red
- observations more than 1 degree colder than analysis are plotted in blue
- observations within 1 degree of the analysis values are plotted in grey

This threshold of 1 degree is a crude assessment and one should really use some typical value of uncertainty. The plotting can show coloured numerical values or symbols at the stations' locations.

Program Overview

The outline of the macro program will be :

- extract the BUFR file's 2m temperature values as geopoints format
- read the analysis GRIB file
- calculate difference between observation geopoints and analysis field and classify geopoints according to the magnitude of the difference
- define visual definitions for plotting
- plot the data

To start the exercise, create a new Macro, give it a name of your choice and code away - depending on your experience you may need/want to look at the information below.

Extracting Parameters from BUFR Data

The first task is to extract the 2 metre temperature from the BUFR file (if you examine the BUFR file, you will see that each message contains many parameters, and not all messages contain 2m temperature).

This can be done by means of the *Observation Filter icon*. The **Output** should be set to Geographical Points, and the **Parameter** should be set to 012004 (which is the default). You can confirm that this is 2 metre temperature by clicking the arrow next to the selection box.

Visualise the icon to confirm that it is returning what you want. The supplied *Symbol Plotting* icon *obs_auto_symb* provides a quick way to get an overview of the data.

The Observation Filter icon can now be dropped into the Macro Editor to generate the first piece of code. The `read()` command will accept a relative path if you prefer.

```
 # read the BUFR file and extract t2m

obs_area_bufr = read("obs_area.bufr")

t2_geo = obsfilter(
        output  :  "geopoints",
        data    :  obs_area_bufr
        )
```

Reading the Analysis Field

Handling analysis fields in GRIB format is simply a case of using the `read()` function to read its contents into a fieldset variable.

```
 # read the GRIB file of 2T analysis
t2_grib = read("t2_an.grib")
```

Deriving Analysis-Observations Difference

Once both input data are ingested, calculations are easy in Macro.

```
 # Compute the difference
diff = t2_geo - t2_grib
```

When you subtract a field (fieldset) from point data (geopoints), the result is a set of point data (geopoints) with the differences.

Plotting of Data

The easiest way to plot the data is to let the classification of the differences be carried out by the plotting specifications. The icon *diff_symb* shows one way to do this, and its equivalent macro code is shown here, including the `plot()` command:

```
 # Compute the difference
diff = t2_geo - t2_grib

 # plotting visdef
diff_symb = msymb(
    symbol_type                         : "marker",
    symbol_table_mode                   : "advanced",
    symbol_advanced_table_selection_type : "list",
    symbol_advanced_table_level_list    : [-1000,-1,1,1000],
    symbol_advanced_table_colour_method : "list",
    symbol_advanced_table_colour_list   : ["blue","grey","red"]
    )

 # plot the data
plot(diff, diff_symb)
```

The symbol visual definition assigns differently coloured markers according to the value of the point data. It uses "advanced table mode", allowing the levels to be defined in a similar way to those in the *Contour* icon. Three intervals are defined, each with a different colour assigned.

This method of classification is the simplest, and saves you from doing computations, but it does limit the ways you can classify your data.

Alternative Classification and Plotting of Data

Duplicate your macro, and start editing the copy. Delete the `msymb()` and `plot()` commands.

We will now separate the 'cold', 'similar' and 'warm' points *in the data*, resulting in three new data sets (which will also be in geopoints format).

Now we need to identify those observations that exceed the analysis value by more than 1K, those that are more than 1K smaller and those within this interval. We can use the `filter()` geopoints function :

```
 # Extract geopoints that are hotter by 1 deg or more
hotter = filter(diff, diff >= 1)

 # Extract geopoints that are colder by 1 deg or more
colder = filter(diff, diff <= -1)

 # Get geopoints that are within +/-1
exact  = filter(diff, (diff>-1)*(diff<1))
```

The `filter` function uses two geopoints: the first one is filtered by retaining the values where the second geopoints is non-zero. Here, the second geopoints results from logical operations involving the first geopoints itself - i.e. the operation `diff`

`>= 1` returns a geopoints with values of 1 where the condition is true and 0 otherwise.

Now all we need is to plot the data with suitable visual definitions – since we have three data variables, we will need a separate visdef for each (because each will be plotted in a different colour). See the supplied icon *diff_symb_red* for an example. The code could look like this:

```
red = msymb(
    symbol_type   : 'marker',
    symbol_colour : "red"
    )

blue  = msymb(
    symbol_type   : 'marker',
    symbol_colour : "blue"
    )

grey  = msymb(
    symbol_type   : 'marker',
    symbol_colour : "grey"
    )
```

Now use `plot()` (we could have also defined a suitable display window)

```
plot(hotter,red,colder,blue,exact,grey)
```

Notice that now there is no intelligence in the plotting definition – the three classifications have been performed on the data itself. With this method, the three derived data sets could be separately saved to file (`write()` command) or processed further in the macro.

ASCII Table Version of Task

We will now repeat the task, but with the observation data stored in an ASCII file instead of a BUFR file (much of the code can be copied and pasted). Create a new macro for this task.

You may like to use a *Table Visualiser* icon to directly plot the points from the CSV file for a quick visual inspection.

Conversion ASCII Table - Geopoints

Reading ASCII Table Files

New to Metview 4 is the ability to directly read ASCII table files. These are text files with one variable per column – CSV (comma separated values) is an example of this type of file and is a common export format from spreadsheet applications.

In Metview 3, such files had to be parsed line by line. If you have a text file which does is not readable by Metview 4's Table Reader, then you may still need to do this. See the main Metview documentation or an earlier version of this tutorial (available on request) for more information.

Look at the supplied file `t2_20120304_1400_1200.csv`. This is a standard CSV file, with a header row at the top, followed by one row per observation, one column per field.

```
Station,Lat,Lon,T2m
1,71.1,28.23,271.3
2,70.93,-8.67,274.7
. . .
```

To read it in Macro, we can first create a new *Table Reader* icon (**Filters drawer**) to help us. Edit it and drop the CSV file into the data field. Because this is a standard CSV file, the default settings are suitable for a standard CSV file, but you can see that there are enough options to cater for many different formats of ASCII table files.

Apply and close the editor; you cannot do anything with this icon apart from drop it into the Macro editor. Drop the edited Table Reader icon into your new macro. After renaming the given variable, we end up with something like:

```
 t2_csv = read_table(
        table_filename  :  "t2_20120304_1200.csv"
        )
```

We can confirm that this is a table variable with 4 columns:

```
print(type(t2_csv))
print(count(t2_csv))
```

Output:

```
table
4
```

All the data from the file is now stored in memory. If we had wanted to be a bit more selective, we could have told the *Table Reader* to only read columns 2, 3 and 4 (**Table Columns** parameter) since we don't care about the Station column.

We can now extract the columns we want and store their values in `vector` variables (a `vector` is simply an array of numbers):

```
lats = values(t2_csv, 'Lat')
lons = values(t2_csv, 'Lon')
vals = values(t2_csv, 'T2m')
```

We could also have specified the desired columns by index (starting at 1), for example: `lats = values(t2_csv, 2).`

In order for Metview to be able to compute the difference between this data and the analysis GRIB field, we must convert it into the geopoints format.

Creating a New Geopoints Variable

A set of related macro functions enables the creation of new, and modification of existing geopoints variables. We will not need them all, but you can read full details here:

`https://software.ecmwf.int/metview/Geopoints+Functions.`

The functions we will use are:

```
geopoints create_geo ( number, string )
```

Creates a new geopoints variable with the given number of points, all set to default values and coordinates. If saved, the geopoints file will be in the `traditional' 6-column format. If another format is desired, supply a string as the second parameter, possible values being 'polar_vector', 'xy_vector' and 'xyv'.

```
geopoints set_latitudes  ( geopoints, number or list )
geopoints set_longitudes ( geopoints, number or list )
geopoints set_values     ( geopoints, number or list )
```

Returns a geopoints variable with either its latitude, longitude or value component modified to be the values given in the second parameter.

The new code could look like this:

```
# create the geopoints variable we will populate
# - we could create a full, 6-column, geopoints variable,
#   which is the default if we do not supply a second
#   parameter to create_geo(). But here we do not require any
#   intelligent date-matching, so xyz is enough.
```

```
t2_geo = create_geo (count(lats),'xyv')


 # put the vectors of values into the geopoints variable.
 # - if we went for the 6-column type, then we would also call
 #   set_level(), set_date() and set_time().

t2_geo = set_longitudes(t2_geo, lons)
t2_geo = set_latitudes (t2_geo, lats)
t2_geo = set_values    (t2_geo, vals)
```

Putting it all Together

Now copy and paste code from your previous macro to read the GRIB field, compute the difference from `t2_geo` and plot it.

Further Work

We can print out some statistics on the difference values:

```
 # print out some statistics
print('Count: ', count(diff))
print('Min:   ', minvalue(diff))
print('Mean:  ', mean(diff))
print('Max:   ', maxvalue(diff))
```

We hard-coded the names of the data files that we read. Since the CSV filename contained elements such as the name of the parameter and the date and time, we could generalise our macro and put these into variables which we could then use to construct the name of any such file (assuming a consistent naming convention is used). Alternatively, the macro could do the reverse: take a filename as an input argument and parse it to obtain these details; a useful title could then be generated.

The code for the reading of the CSV file and conversion to geopoints could be made into a function.

If the generated geopoints variable is considered worth keeping, then it could be written to file using the `write()` command.

This macro might benefit from a simple user interface.