Metview Macro First Tutorial

Meteorological Visualisation Section
Operations Department
ECMWF

12/03/2013

This tutorial was tested with Metview version 4.3.7
but should not work for all 4.3.x versions.

INTRODUCING METVIEW MACRO

Overview

A macro language was part of the first design specification of Metview and is designed to perform data manipulation and plotting from within the Metview system environment.

A language is the best "user interface" to describe very complex sequences of actions (particularly if the flow of action is conditional) and it also provides a common means to express the mathematical formulae used when performing data manipulations. The Metview macro language was designed to be :

- as easy to use as a script language (e.g. UNIX shell) - to be as simple as a shell implies that no variable declarations or program units should be required. This feature is achieved through the implementation of typeless variables, a benefit of object-oriented languages.
- as powerful as a modern computer language - to be as complex as a computer language implies support for variables, flow control, functions and procedures, I/O and error control.

The Metview macro language provides an easy, powerful and comprehensive way for a researcher to manipulate and display meteorological data. It extends the use of Metview into an operational environment as it enables a user to write complex scripts that may be run everyday at user defined times.

Furthermore, the seamless integration with user written FORTRAN and C programs, access to shell commands from within the macro code, possibility to set and use environment variables and the ability of macro programs to be run in batch (command line mode) are major features of Metview Macro which extend enormously its range of application.

This tutorial exercise provides a step by step introduction to the Metview Macro language, divided in the following main sections :

- A simple visualisation task introduces Macro's basic features
- The Macro's task control features are applied to this simple task
- Role and implementation of functions in Macro, including the integration of user's FOR-TRAN and C routines
- Further short examples involving all types of data that Macro can handle

The Metview Macro Icon

In Metview everything is an icon, and macro programs are represented in user workspaces by their own icon which looks like so:

Working with the macro language follows the usual Metview procedure:

- create a **Macro** icon, where suitable code is written and kept, which is saved in your work space
- carry out an action on the macro icon - depending on its code you may execute, visualise, save or drop the icon to obtain the desired output

Which action you choose to run the macro program depends on how you implement the macro return. This is a particular topic of the tutorial, since this provides a powerful means to control Metview tasks and allows the same macro program to be run in a variety of ways, producing a variety of outputs.

Creating a Macro Program

You can create a macro program by one of three procedures.

The first is ideal while creating your very first programs, since it shows clearly the relationship between icons and macro code:

- You may create (and possibly test) the icons required for the task - e.g. MARS icon(s) and contour icon(s), and drop them inside the macro editor window. This provides you with the macro language equivalents of the icons. You still have to structure these into a proper macro program and add relevant instructions (e.g. the one to plot the data). This procedure is very effective and helpful if you want to know how to specify a given definition (or part of it) in macro language code. Creating complex programs following this procedure may be quite cumbersome. Note that the resulting textual translation only specifies those parameters in the dropped icon which have been changed from their default values.

The second offers a quick way to turn interactive code into a macro program :

- Simply save a visualisation on screen as a macro icon. This gives you a program that can regenerate the visualisation but its scope is limited, the automatically generated code maybe cumbersome to understand and doesn't help you much when you need to carry out computations on existing data

The third is just hand coding:

- Create a macro program from scratch, type in the relevant code and / or cut and paste from other programs or available examples.

In this tutorial we will start with the first approach and then quickly move on to hand coding due to the limitations of icon-drop programming.

A BASIC MACRO PROGRAM

Introduction

The first steps in this tutorial cover a simple exercise based on differences between analysis and forecast fields. The tutorial materials consist of :

- A single GRIB file (**TUV_Data**) containing multi-level temperature and wind analysis fields for 01/03/2012 and multi-level temperature/wind 5 and 2 day forecast fields verifying in the same date
- Two **GRIB Filter** icons, one filtering only the temperature analysis data from the file (**Tan**), the other the temperature forecast data (**Tfc**)
- A **Simple Formula** icon (**fa_diff**) set up to return the difference between the temperature analysis and the forecast fields at each level
- Two **Contour** icons (**pos** and **neg**) providing contours suitable for temperature difference fields

To examine the data interactively you can visualise the **fa_diff** icon using both the **pos** and **neg** contour icons dropped together; you may also inspect and Examine any of the **Tan**, **Tfc** or **TUV_Data** icons as well.

Other icons are provided to help in the first stages of the tutorial : a **Display Window** icon **(dw)**, a **Geo View** icon (**ps_atlantic**) and a **Cross Section View** icon (**xs_euro**).

A final word on how to carry out the tutorial exercise :

The tutorial is organised as a work through number of sequential steps. Save the result of each step in a separate macro. Generally, to progress to the next, step duplicate the macro for the current step and edit this duplicate.

Step1 - From icons to macro code

The quick and easy way to start coding your interactive work in macro is to simply drop the prepared icons inside a Macro editor.

We know that visualising the **fa_diff** icon leads to a display window with a set of forecast-analysis difference fields. This icon uses other icons as input, but there is no need to worry about this :

Dropping an icon in a Macro editor will provide the macro code equivalent of the icon, and the macro code equivalent of the all the icons it uses as input.

Start by creating a Macro icon. Rename it to **step1**.

Edit the macro – if you took the icon from the drawer, there will be a header line - place the cursor after this line.

Double-click **step1**, then drag the **fa_diff** icon and drop it inside the macro editor window. The following piece of text is written to it (a dummy group and user id have been used) :

```
#Metview Macro

# Importing : /macro_tutorial/macro_tut1/TUV_Data

tuv_data =
read("/home/xy/xyz/metview/macro_tutorial/macro_tut1/TUV_Data")

# Importing : /macro_tutorial/macro_tut1/tfc

tfc = read(
        logstats    :    "",
        type    :    "fc",
        param    :    130,
        date    :    20120228,
        step    :    48,
        data    :    tuv_data
        )

# Importing : /macro_tutorial/macro_tut1/TUV_Data

tuv_data =
read("/home/xy/xyz//metview//macro_tutorial/macro_tut1/TUV_Data")

# Importing : /macro_tutorial/macro_tut1/tan

tan = read(
        logstats    :    "",
        type    :    "an",
        param    :    130,
        date    :    20120301,
        data    :    tuv_data
        )

# Importing : /macro_tutorial/macro_tut1/fa_diff

fa_diff = tfc - tan
```

Looking at the code above you can see how all the intervening icons have been translated into Metview macro. Because each is read in turn and the **TUV_Data** icon is used as input in both **Tan** and **Tfc** filter icons, the **TUV_Data** file is read twice. So, you can delete the second reading of the GRIB file. You can also remove the comments, or add your own if you so wish.

The resulting code reads the GRIB file, and then filters out the analysis and the forecast data, storing them in the variables `tan` and `tfc`. These variables hold the fields and are of type `fieldset`. This is also the type of the variables `tuv_data` and `fa_diff`.

Note that no further action has yet been specified. So if you were to save and run the above macro, nothing visible would happen, though the data would have been read and filtered. So, to make things more interesting, add the line of code below :

```
plot(fa_diff)
```

To run the macro use one of these two methods:

- Click the ‘play’ button (keyboard F9) from the Macro editor

- save the macro and choose **Execute** from the icon right-click menu

Either way, the result is a default display window with the difference fields displayed using the default contouring.

NOTE:

You can remove the *logstats* lines from your macro code if you wish – they do not do anything.

Only the non-default parameters in a dropped icon are written to the macro.

The GRIB file icon internal to the **GRIB Filter** icon was also translated into the macro. When you drop an icon inside a macro editor, all of its internal (input) icons also get translated to macro code.

Variables are typeless - they are simply created when something is assigned to them and take the type of whatever is assigned. The memory and disk space used by variables is released if you assign another item to the variable, or if the variable becomes out of scope (e.g. when the macro finishes).

Whatever you do not specify in macro is provided by the system defaults (e.g. display window and contours).

Comments in macro programs are anything preceded by the symbol #.

Strings can be inside double (“) or single (‘) quotes - both are equivalent.

Step 2 - Using contours and a display window

Duplicate the **step1** macro icon and rename the duplicate **step2**.

In the previous step we did not specify a suitable contour and we simply used the default projection (cylindrical) and geographical area (globe). In this step we will specify all these elements, still by means of icon drops.

First, ensure the cursor in the Macro editor is placed before the call to the `plot()` function as the icon code is inserted at the cursor location.

Second, drop the contour icons **pos** and **neg**. This adds the contour macro definitions to the program.

Third, drop the display window icon **dw** after the contour definitions. This **dw** icon uses the map view icon **ps_atlantic** as input, and this icon uses the Coastline icon **acoast** as input. As before, all these input embedded icons are automatically translated to macro.

You should obtain (after cleaning comments and excluding previous code):

```
pos = mcont(
        contour_line_thickness       : 2,
        contour_line_colour          : "red",
        contour_highlight            : "off",
        contour_level_selection_type : "level_list",
        contour_max_level            : 10,
        contour_min_level            : 0.5,
        contour_level_list           : [0.5,1,2,4,10]
        )

neg = mcont(
        contour_line_thickness       : 2,
        contour_highlight            : "off",
        contour_level_selection_type : "level_list",
        contour_max_level            : -0.5,
        contour_min_level            : -10,
        contour_level_list           : [-10,-4,-2,-1,-0.5]
        )

acoast = mcoast(
        map_coastline_resolution        : "low",
        map_coastline_thickness         : 3,
        map_coastline_land_shade        : "on",
        map_coastline_land_shade_colour : "grey",
        map_coastline_sea_shade         : "on",
        map_coastline_sea_shade_colour  : "RGB(0.9,0.95,1)",
        map_grid_longitude_increment    : 10
        )

ps_atlantic = geoview(
        map_projection      : "polar_stereographic",
        map_area_definition : "corners",
        area                : [30,-25,50,65],
        coastlines          : acoast
        )

page = plot_page(
        view  : ps_atlantic
        )

dw = plot_superpage(
        custom_width    : 29.7,
        custom_height   : 21.0,
        pages           : page
        )
```

Now, simply add the variables `pos`, `neg` and `dw` to the `plot()` function:

```
plot (dw, fa_diff, pos, neg)
```

The above line can be understood as: plot in `dw`, the fieldset `fa_diff` with the contours `pos` and `neg`. Now run the macro.

NOTE:

The internal coastline icon was also imported even though it is placed in a hidden folder.

The `plot()` function can take a variable number of arguments but it is important that they are supplied in a well defined order :

- the first argument must be the display window variable (if you are using a non default one)
- the second argument (or first if you're happy with the default display window) must be the data to be plotted
- the remaining arguments following the data must be visual definitions (e.g. contours , wind arrows, ...) and you can have as many as you need.

- if plotting several data, each must be followed by the visual definition (e.g. contour) that applies to it. If using a single visual definition for all data items, specify all the variables first and then the visual definition.

We run the macro with the Execute action, but you can do it with other actions (or run modes), such as Visualise, Examine or Save - the outcome would have been the same, since we coded a specific action in the macro (plot), which is carried out irrespective of the run mode. Later we'll see how to make the macro respond differently to different run modes.

Step 3 - Add some control : Variables

DUPLICATE the **step2** macro icon and rename the duplicate **step3**.

So far the forecast step and the forecast and analysis dates are fixed and so is the plot geography. We now create variables to hold these values in order to introduce more flexibility in the macro. Note that Macro accepts both `"` and `'` as string delimiters.

Start by adding the following lines at the top of the macro:

```
par        = "t"
vf_date     = 2012-03-01
n_of_days = 5
the_area   = [30,-25,50,65] # S, W, N, E
```

and modify the data filtering as such :

```
tfc = read(
              type       : "fc",
              param      : par,
              date       : vf_date - n_of_days,
              step       : n_of_days * 24,
              data       : tuv_data
              )


tan = read(
              type       : "an",
              param      : par,
              date       : vf_date,
              data       : tuv_data
              )
```

Also modify the `ps_atlantic` definition as follows:

```
ps_atlantic = geoview(
        map_projection      : "polar_stereographic",
        map_area_definition : "corners",
        area                : the_area,
        coastlines          : acoast
        )
```

Clearly this changes nothing in the results as the program stands. However, the use of the variables allows us to view some different outputs with relatively little change. In addition to temperature, the source GRIB file contains U and V wind components. Also, the forecast step is not limited to 5 days; a 2-day forecast is available in the data file.

Try the following, in any order or combination:

- Change the variable `n_of_days` to 2.
- Change the variable `par` to `"u"` or `"v"` or `["u", "v"]`. Note that the contour definition used for the temperature fields may not be appropriate for wind components. The last option defines a **`list`** of parameters; Metview will filter out both the u and v wind components. Also note that when Metview has u and v wind components sequentially, it will display them as a single vector wind field.
- Change the area definition. This is a list, denoting the South, West, North and East limits of the area to be visualised.

In real life you are likely to be filtering data out of larger GRIB files or else retrieving data from a database. Under these circumstances you will have greater flexibility when changing parameters.

In any case it is good programming practice to keep input values which you may need to change, assigned to clearly named variables placed at the top of the macro - should you need to change any of those input values you don't need to search for their each and every occurrence in the macro code.

NOTE:

That numbers can specify dates in some Macro functions if they follow some pre defined formats (e.g. `YYYY-MM-DD`).

The way the macro language performs date manipulations - you can subtract or add any number (integer or real) from a date and the result will be another date (down to the second).

The use of lists - the plotting region was held in a variable of type **`list`** (`the_area`). The parameter variable (`par`) can be a list (["u", "v"]) or, alternatively, a string ("t"). For ease of editing, you could always specify a list, even if only filtering a single parameter, for example, ["t"].

Step 4 - Add some control : a function

Duplicate the **step3** macro icon and rename the duplicate **step4**.

The macro language supports functions like any other programming language. Here we provide a simple example where the difference between analysis and forecast is computed by a function.

We will prepare a function that takes as its arguments the name of a GRIB data file, a parameter code, a verification date and a forecast step in days and which returns the difference field.

To do this prepare a function declaration and rearrange the macro, *moving* the data reading and filtering inside the body of the function, as shown below. Things to watch out for are highlighted in bold.

```
function fc_an_diff (fname, par, vf_date, n_of_days)
```

```
        infile = read(fname)

        fc = read(
             type       :    "fc",
             param      :    par,
             date       :    vf_date - n_of_days,
             step       :    n_of_days*24,
             data       :    infile
             )

        an = read(
             type       :    "an",
             param      :    par,
             date       :    vf_date,
             step       :    0,
             data       :    infile
             )

        return fc-an

end fc_an_diff
```

Make sure that the three calls to **read** have been moved from the main body of the program - they should now only appear in the new function definition. Likewise, remove the `tfc-tan` calculation as it should also appear only in the function.

You can then call the new function. We also have to add a new variable to hold the file name from which we filter the data. To make this more general we also exemplify the usage of environment variables in macro :

```
home      = getenv("HOME")
path      = home & "/metview/macro_tutorial/macro_tut1/"
file_name = path & "TUV_Data"
par       = "t"
vf_date   = 2012-03-01
n_of_days = 5
the_area  = [30,-25,50,65] # S, W, N, E


 (...) # contouring and plot window

fa_diff = fc_an_diff(file_name, par, vf_date, n_of_days)
plot (dw, fa_diff, pos, neg)
```

At present, we are always using the contour visual definitions `pos`, and `neg`, even if plotting wind arrows. Metview knows not to apply a contour definition to a wind arrow plot, but we can do better than this. Drop the icon **Coloured Wind** into the macro editor just below the `pos`, and `neg` definitions, creating a new one called `coloured_wind`. Now add the following lines to select which visual definition we will use:

```
if (par = ['u', 'v']) then
    visdef = coloured_wind
else
    visdef = [pos, neg]
end if
```

Now we just have to revise the plot command to use this selected visual definition instead of always using `pos` and `neg`:

```
plot (dw, fa_diff, visdef)
```

NOTE:

The use of environment variables in macro code. Macro can read and set environment variables (through `getenv()` and `putenv()`) and this can be extremely useful to provide system input to the macro and to generalise your programs. A typical usage is to retrieve paths of storage locations specified as environmental variables (e.g. `$SCRATCH`).

The use of a full path to the data file. The `read()` function requires a full path if the macro is being run from within the macro editor; otherwise a relative path will be ok.

The string concatenation to form the full file name. Here we concatenated the path for the user home directory with the metview path and file name. A common usage is to derive a file name from, for example, verification date / forecast step / parameter specification by users who employ consistent file naming conventions.

The function code can be placed anywhere in the macro program. Functions can be stored in separate files, forming libraries available to one or all users.

No variable types are specified for the function arguments (but they could have been). You could have used the parameter number (130) rather than the letter ("t").

If You Have Extra Time ...

Print some statistics on the data. The following code will print information about the first difference field (level 850hPa):

```
print ('minvalue: ', minvalue(fa_diff[1]))
print ('maxvalue: ', maxvalue(fa_diff[1]))
print ('average:  ', average (fa_diff[1]))
```

Look at the output area in the Macro editor (or the Metview message window) to see this information. Can you find the page in the Metview Macro documentation which describes the list of available functions on fieldsets?

Add the possibility to plot onto a Mollweide-projected map. It should then be possible to switch between the two map types by changing a single line of code.

Place the code that selects the visual definition into a separate function. It should take as its parameter the name of the meteorological parameter to plot, and return the appropriate visual definition. Note that the visual definitions themselves (`cdiff` and `coloured_wind`) will have to be moved into this function so that it can 'see' them. If you want to return a 'null' visual definition if the supplied parameter is not one for which we have a visual definition, then you can return the empty definition, `()`.

Progress So Far

At this stage we have successfully implemented a number of Metview Macro features in our code. To recap let us show the full code developed so far. A few comments have been added for clarity:

```
#Metview Macro

# Program parameters
```

```
home      = getenv("HOME")
path      = home & "/metview/macro_tutorial/macro_tut1/"
file_name = path & "TUV_Data"
par       = "t"  #['u','v']
vf_date   = 2012-03-01
n_of_days = 5
the_area  = [30,-25,50,65] # S, W, N, E


 # Define some visual definitions

pos = mcont(
        contour_line_thickness       : 2,
        contour_line_colour          : "red",
        contour_highlight            : "off",
        contour_level_selection_type : "level_list",
        contour_max_level            : 10,
        contour_min_level            : 0.5,
        contour_level_list           : [0.5,1,2,4,10]
        )

neg = mcont(
        contour_line_thickness       : 2,
        contour_highlight            : "off",
        contour_level_selection_type : "level_list",
        contour_max_level            : -0.5,
        contour_min_level            : -10,
        contour_level_list           : [-10,-4,-2,-1,-0.5]
        )

coloured_wind = mwind(
        legend                                : "on",
        wind_arrow_legend_text                : "m/s",
        wind_advanced_method                  : "on",
        wind_advanced_colour_max_level_colour : "red",
        wind_advanced_colour_min_level_colour : "blue",
        wind_advanced_colour_direction        : "clockwise",
        wind_thinning_factor                  : 1
        )

acoast = mcoast(
        map_coastline_resolution        : "low",
        map_coastline_thickness         : 3,
        map_coastline_land_shade        : "on",
        map_coastline_land_shade_colour : "grey",
        map_grid_longitude_increment    : 10
        )

ps_atlantic = geoview(
        map_projection      : "polar_stereographic",
        map_area_definition : "corners",
        area                : the_area,
        coastlines          : acoast
        )


 # Define the display window

page = plot_page(
        view  : ps_atlantic
        )

dw = plot_superpage(
        custom_width    : 29.7,
        custom_height   : 21.0,
        pages           : page
        )
 # Check which visual definition to use: contour or wind arrows?

if (par = ['u', 'v']) then
    visdef = coloured_wind
else
    visdef = [pos, neg]
end if



 # Derive the difference field

fa_diff = fc_an_diff(file_name, par, vf_date, n_of_days)


 # Plot the result

plot (dw, fa_diff, visdef)


 # ===================================================================

 # A function to compute a difference field
 # Author, date, restrictions, argument types, etc,...

function fc_an_diff (fname, par, vf_date, n_of_days)

        infile = read(fname)

        fc = read(
             type      :    "fc",
             param     :    par,
             date      :    vf_date - n_of_days,
             step      :    n_of_days*24,
             data      :    infile
             )

        an = read(
             type      :    "an",
             param     :    par,
             date      :    vf_date,
             step      :    0,
             data      :    infile
             )

        return fc-an

end fc_an_diff
```

Clearly there are weaknesses in the current program, mainly:

- The macro program is only catering for on-screen visualisation. Other alternatives such as PostScript are available but there is a need to choose between them without editing the code
- Users need a way to pass arguments to the macro without having to edit the code each and every time.

The next section will show how to implement user control over macro program runs.

OTHER MACRO OUTCOMES

With Metview Macro you can specify a variety of outcomes for your task other than the on- screen visualisation we have been dealing with so far. You may also :

- return the output to the user (e.g. when the macro icon is dropped somewhere)
- save the output (whether GRIB, BUFR, text) to a file
- visualise the output on a medium other than the screen (PS, PNG, SVG or KML files)

Here we show simple examples of how to obtain these outcomes.

Step 5 - Macro Return

Duplicate the **step4** macro icon and rename the duplicate **step5**.

So far we have been executing the macro files that we have written. What happens if you drop this icon inside an existing display window? Nothing, because the macro did not explicitly provide any *return* information about what must be plotted.

To solve this, consider the entire macro as a function which can return something. We simply remove the `plot_superpage()` command and replace the `plot()` instruction with a `return` instruction:

```
 # the previous code
 (...)

fa_diff = fc_an_diff (file_name, par, vf_date, n_of_days)
return fa_diff
```

The macro can return more than a simple fieldset or item; returned items can be more complex objects, such as lists, which can hold a variety of different variable types:

```
 (...)
return [fa_diff, visdef]
```

This can be used to provide not only the data but also suitable visual definitions.

Running the macro by itself will no longer result in a plot because we have removed the `plot()` command.

Now, when you drop the macro icon in a display window, the macro output (in this case a difference field and a contour) is returned to Metview which will pass it to the visualisation module and hence the field will be visualised with the specified contour. If returning only the field, it will be visualised with the default contours; if returning a list of field plus contour it will use the returned contours.

Try dropping the macro in the visualised Geo View icon `Mollweide`; also try dropping it into the visualised cross-section view icon `xs_euro`. You can also try creating a display window with a frame for a cross-section and a frame for a geo view.

NOTE:

Macros that return fieldsets (or other data such as geopoints) are a powerful feature in that they can be used as input to another application, e.g. a **GRIB Filter**, a **Cross-Section**, etc.,. This is particularly relevant when you need to operate on a field which may be the result of a very complex calculation only feasible to carry out by means of Macro.

In Metview 3, the call to `plot_superpage()` would have to be commented out in order to avoid an empty Display Window being generated by the macro. In Metview 4, the Display Window will only be generated when the `plot()` command is called, so we can leave the code harmlessly in place.

Step 6 - Saving Output to a File

Duplicate the **step5** macro icon and rename the duplicate **step6**.

Apart from visualisation and returning we can use macros to derive data files. This may be your express purpose or you may need to do it if the derived data takes too long to calculate, and/or is too complex and/or is too large. In this situation you simply save the result of your calculations to a file.

In our tutorial example, all you need to do to save the difference field to a GRIB file is to replace the return instruction of the previous step by a suitable use of the function `write()` :

```
# previous code
(...)

# saving the resulting data
write("Diff.grib", fa_diff)
```

This creates the file in the same directory where you run the macro from. You may also use a file handler to do the same job in a different way (see NOTE to understand the difference):

```
# previous code
(...)

# saving the resulting data
fhan = file("Diff.grib")
write(fhan, fa_diff)
fhan = 0 # this closes the file
```

You do not need to specify what kind of data is being written. The macro will know what data type the variable holds and `write()` will create the right kind of file (e.g. a fieldset is written to a GRIB file). The file, if saved within the Metview environment (directory tree) will be assigned the icon of that file type.

NOTE:

When writing just once, it does not matter much whether you use a filename or a file handler. However, when writing multiple times to a file, the choice does matter:

- when using a filename, `write()` will always overwrite the given file with new data.
- when using a file handler, `write()` will overwrite an existing file, but append on subsequent calls using the same file handler.
- there exists a similar function, `append()` which will always append data to a file, even a pre-existing one; it does not matter here whether a filename or a file handler is used.
- there are other more subtle differences between using file names and file handlers - see the Metview User Guide.

In all cases you can specify a path to save the output in a place other than the current folder.

Step 7 - Other Visualisation Media (PS/PDF/PNG/SVG)

For this step we require again the geographic details and display window definition, so duplicate the **step4** macro icon and rename it **step7**. Note: we make a copy of **step4**!

So far we have been visualising on the screen. You can obtain a PS, PNG, PDF, SVG, etc. from the Display Window's **Export** button, but you may need to produce the output directly to one (or more) of these formats, either because that is the desired end product or because you run the macro in batch and hence cannot use an interactive visualisation.

The way to code for other output devices is to specify them with the `xxx_output()` functions and set the one(s) you want to use with the `setoutput()` function.

The new code follows after the set up of the input variables :

```
 # Program parameters

home      = getenv("HOME")
(...)


 # define some output media

outfile = path & 'diff'
to_psfile  = ps_output (output_name : outfile)
to_pngfile = png_output(output_name : outfile)
to_svgfile = svg_output(output_name : outfile)


 # set the required output medium to one of the defined above

setoutput(to_psfile)


 # Remainder of macro code (geography, display window, plot cmd,...)
 (...)
```

Now save and execute the macro. The result will be a PS file represented by a PS icon, or a series of PNG or SVG files represented by their respective icons. Alternatively, the plot may be visualised on-screen by removing or commenting out the `setoutput` command. Try changing the `setoutput` call in order to see each type of output. If you have need of more than one output format, you can pass them all to the `setoutput` command, for example these are equivalent:

```
setoutput(to_psfile, to_pngfile)    # passing two variables
setoutput([to_psfile, to_pngfile])  # passing a single list variable
```

NOTE:

A number of output device definitions are provided, (`to_psfile`, etc) one or more of which are explicitly selected with the `setoutput()` function. You needn't specify anything if you want your output on-screen, since this is the default output medium. There is no harm in keeping the definitions for output formats you will not use – for example, if you do not use `to_svgfile`, then defining it will have virtually no impact on performance.

For some output types, Metview automatically appends a number and an extension to the supplied filename. This is because these types cannot hold more than one 'page' per file.

*In Metview 3, the function `setoutput()` had to come before the setting up of the display window; the display window **must** be defined if you use `setoutput()`. This restriction is no longer true in Metview 4.*

If You Have Extra Time ...

There may be cases where an output filename should reflect the nature of the data. Add some code to generate a meaningful filename using the parameter name, the forecast step and the analysis date as elements. An example output filename for PostScript might be:

```
t_20120301_5_diff.ps
```

For the 'uv' case, you will have to translate the list ['u', 'v'] into a string. The following piece of code checks for the case where the parameter is a list and makes a single string out of its elements:

```
if (type (par) = 'list') then
    parameter_name = ""

    loop element in par
            parameter_name = parameter_name & element
    end loop
else
    parameter_name = par
end if
```

The function `yyyymmdd()` takes a date and creates an eight-digit number out of it. The Metview User Guide details many functions that can be used to produce different formats from a date. Numbers are automatically converted to string form when appended to a string.

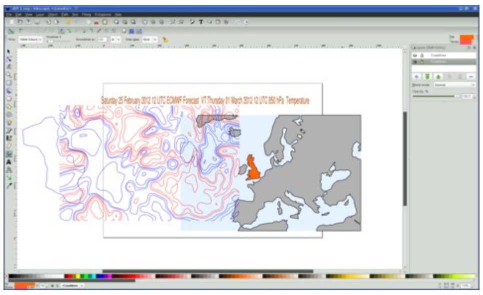
Putting everything together, a possible line of code to produce the filename could be:

```
outfile = path & parameter_name & '_'  & yyyymmdd(vf_date) &  '_'
```

```
                    & n_of_days & '_diff'
```

Add another line to the title, specifying the number of days that the forecast represents (held in variable `n_of_days`). To get the basis for the code, you can create a new Text Plotting icon, add a second line to it, then drop it into the Macro editor for further editing. Remember to add your title definition to the `plot()` command.

Plot to an SVG file. Then **examine** the resulting SVG icon to open it in the editing program Inkscape. After some **Ungroup** commands, you will be able to edit individual features in the file.


Saturday 25 February 2012 12 UTC ECMWF Forecast VT:Thursday 01 March 2012 12 UTC 850 hPa Temperature

MACRO RUN MODE CONTROL

Given the possible outcomes from Macro, it would be convenient to cater for all of them from within the same program; otherwise you will need to keep duplicated code. To do this, remember that you can run a macro program either :

- by choosing one of the following options from its icon menu:
    - **Execute**
    - **Visualise**
    - **Save**
    - **Examine**
- or by dropping the icon in a plot window or editor window icon field
- or by running in batch mode.

The way a macro is run is called the *macro run mode*. The four icon menu options above correspond to an identically named run mode. The last two options correspond to the run modes **Prepare** and **Batch** respectively. There is a also an **Edit** run mode, used for coding user interfaces (see below).

A macro can detect its own run mode and this provides the solution to our problem - we can assign specific actions and outcomes to each (or some) of the run modes including preventing the macro from running.

In our example we follow a very widespread usage of using run modes to specify different media for the macro output :

- plot to a display window if the run mode is Visualise
- print to a PNG file if the run mode is Execute
- print to a PS file if the run mode is Batch
- save the difference fields to a file if the run mode is Save

Step 8 - Macro Control Using runmode()

Duplicate the **step7** macro icon and rename the duplicate **step8**.

The way to code run mode dependent outcomes is by using the function `runmode()`. It returns a string with the run mode:

```
mode = runmode()
```

So it is enough to check this string and to code accordingly, either using `if/else` conditions or the `case/of` condition test.

To introduce the new functionality ***replace*** the existing unconditional call to `setoutput()` by the following lines of code :

```
# define four output media
(...)
```

```
 # check run mode
mode = runmode()


 # select outcome dependent on run-mode
if       (mode = "execute")   then setoutput(to_pngfile)
else if  (mode = "batch")     then setoutput(to_psfile)
else if  (mode = "visualise") then print('Plotting to screen')
else if  (mode = "prepare")   then print('Plotting to screen')
else     fail("Only execute, batch and visualise allowed")
end if

 # remaining code
 (...)
```

Now, depending on how you call the macro your output will be directed to different media. Choose different options from the icon's right-click menu to see what happens. Note that you can also simulate these actions from within the Macro editor (**Program | Run Options**). The 'prepare' run mode is the default one when you run your macro from the Macro editor. The 'batch' run mode will be explained in a later paragraph.

If you select an option not covered by the allowed run modes (e.g. **Save** or **Examine**), the macro will stop, turn red (failed run) and issue an error message - this behaviour is provided by the `fail()` function. A related function, `stop()`, will do the same but allow the macro to exit in the green state (successful run).

Note that you may have to delete the output files before running the macro in order to see that it has worked!

To run the macro in batch mode, you call Metview with the option `-b` followed by the macro name on the command line (assuming you are running from the same directory as the macro - otherwise you must provide a path to it). For example:

```
metview -b step8
```

or

```
metview4_new -b step8 [use this one for the ECMWF March 2013 course!]
```

The newlines within the conditional branching part of the code are down to personal preference. You could also have formatted the code as follows:

```
if (mode = "execute") then
    setoutput(to_pngfile)
else if (mode = "batch") then
    setoutput(to_psfile)
...
```

If You Have Extra Time ...

One important piece of functionality that we have omitted is the ability to save our derived data. Add some code to handle the **Save** run mode by saving the derived data in a file. There are three things you will have to do:

- at the end of the run mode checks, instead of
  ```
  else fail ( ... )
  ```
- we should not fail if the mode is "save":
  ```
  else if (mode <> "save")  then fail ( ... )
  ```
- we must again check the run mode, and if it is **Save** then write the data to a file (see Step 6); otherwise plot the data.

USER INPUT TO MACRO PROGRAMS

So far we have been developing examples of short illustrative macro programs, but one very desirable element is missing: the flexibility to run the same code for different parameters - e.g. you should be able to use the code of the above examples to plot differences for other parameters, other dates or forecast steps.

When users need to provide variable input to the macro this clearly shouldn't be done by editing the code every time. The main ways that a user can provide input to a macro (other than editing the code) are:

- command-line arguments (covered in Step 11)
- user interface (covered in Steps 9 and 10)
- environment variables (already used since we got `$HOME` in Step 4)

This section addresses how to provide input to a macro program via graphical user interfaces similar to those of icon editors, built by the user code. A user interface is defined by the input elements (things such as sliders and option buttons) and their layout. Macro implements such user interface input elements as functions you can call. You may also define default values, so the macro can run with some regularly-used set of values.

So all you have to do is to write the code for the user interface. Then the code has to retrieve the information you input in the user interface and pass it on to the part of the code that carries out the calculations.

There are two ways to implement a user interface :

- The first implementation of user input, demonstrated in step 9, is self-contained within a macro program. It uses the macro `dialog()` function.
- The second, demonstrated in step 10, uses a related Metview Module called **Macro Parameters** in conjunction with the **Macro** module itself.

Step 9 - User Interface 1 : dialog() Function

Duplicate the **step4** macro icon and rename the duplicate **step9**. Note: we make a copy of **step4**!

This implementation of a user interface to enter input is self contained within a macro program, using the macro function `dialog()`. The steps are as follows :

- Create a set of input elements to form a user interface. These are created via the input element functions.
- Pass the returns from the input element functions to the `dialog()` function. This creates the user interface and presents it to the user who may then modify the input values at will.
- `dialog()` extracts the input values which are then assigned to a definition variable to be used in the remainder of the macro.

The input elements that we will enable users of this macro to enter in the user interface will be :

- A GRIB icon storing the analysis and forecast data
- The verification date
- The forecast step
- The meteorological parameter
- The geographical area to plot

First, you will need to modify the function `fc_an_diff()` so that it accepts a data object instead of a file name as the first argument. Simply remove the `read()` instruction on the function's first line and replace its first argument by the variable `infile`. We need to do this as we will supply the data icon directly through the user interface.

Second, you can remove the hardcoded variables sitting at the top of the macro, as the user interface will take their place.

The following code shows the changes you need to make to the macro.

~~# Program parameters~~

~~home   = getenv("HOME")~~

```
(...)

 # define the user interface components
a = icon(
              name       : "input data",
              class      : "GRIB"
              )

b = any(
              name        : "verif date (yyyy-mm-dd)",
              default     : "2012-03-01",
              help        : "help_script",
              help_script_command : "echo Dates must be in YYYY-MM-DD
format"
              )

c = slider(
              name       : "days",
              min        : 1,
              max        : 10,
              default    : 5
              )

d = option_menu(
              name       : "parameter",
              values     : ["UV", "t"],
              default    : "t"
              )

e = any(
              name       : "area",
              help       : "help_input",
              input_type : "area",
              default    : [30,-25,50,65]    # S, W, N, E
              )


 # retrieve the input values
input = dialog([a, b, c, d, e])
```

```
if input <> nil then
        indata          = input["input data"]
        vf_date         = input["verif date (yyyy-mm-dd)"]
        n_of_days       = input["days"]
        par             = input["parameter"]
        the_area        = input["area"]

        # translate "UV" into a list, since that is what 'read'
 understands
        if (par = "UV") then
             par = ["u", "v"]
        end if

else
        fail("macro failed to get input elements")
end if

print (vf_date, "   ", n_of_days, "   ", par)


 # vis defs, geography and display window
 (...)

 # derive and plot the difference field
fa_diff = fc_an_diff(indata, par, vf_date, n_of_days)

plot(dw, fa_diff, visdef)
```

The first section of the code creates a user interface with five input elements: an icon field **Input Data** which accepts only GRIB data icons; a text field **Verif Date** with a default value of 2012-03-01 ('any' means the input text can be a number, a string, a date, a list, etc.) and a help button which executes a shell command in order to display a message; a slider Days with a minimum of 1, a maximum of 10 and a default value of 5; an option menu **Parameter** with two options t and UV; and a text field Area which includes a help button, enabling the user to easily select an area via a graphical user interface (see the `help` and `input_type` members).

In the second part of the code, a list containing these user interface elements is passed to `dialog()` which presents the user interface to the user and returns a definition variable called `input` which holds the input values. The input values are extracted from this definition and used in the remainder of the macro. Note that when working with U/V pairs, we need to supply the `read()` function with a list, not a string, hence there is some additional code to perform this conversion.

To run the macro, Execute or Visualise it: a simple user interface comes up ready for user input. This interface has input tools similar to those you can find in other icon editor windows.

Specifying values for **Verif Date**, **Days**, **Parameter** and **Area** is straightforward. The input file is specified simply by dropping the data icon in the icon field. Once you finish specifying the input parameters, click-left the **OK** button and the macro will run. In this way you can use the same macro to plot analysis vs forecast of any number of days of either T or U/V, for any date, in a chosen geographical area (within the limits of the data available in the file). You may wish to modify the macro so that the user can also select the U and V wind components separately - this is a one-line change to the definition of the **Parameter** input element. Note, however, that you may wish to provide a different contour definition for these parameters.

Step 10 - User Interface 2 : Macro Parameters

Duplicate the **step9** macro icon and rename the duplicate **step10**.

The main problem with using `dialog()` for managing user interfaces is that the user's preferences are not stored. If the defaults specified in the macro are not suitable, then either the macro must be changed or the user must adjust the input values each time they use the user interface.

An alternative way to provide user input to a macro program uses an auxiliary module to Macro, named **Macro Parameters**. Once it has been set up, you will interact with your macro *through* the **Macro Parameters** icon which can then store the values you have set in the user interface. If you edit the **Macro Parameters** icon, it will call the `edit` *handler* in your macro. Similarly, visualising the icon will call the `visualise` handler in your macro. Handlers are special functions which are alternative entry points to your macro.

The macro will implement a special handler called `edit`, called by the **Macro Parameters** icon when you edit the **Macro Parameters** icon. Within this handler, the required input elements (e.g. sliders, option buttons,...) are specified in exactly the same way as in the previous step and returned as a list of controls.

Because of this set-up, we will also prevent the macro icon from running by itself. This is achieved by checking the run mode at the start of the macro and preventing action for all except run mode **Edit**.

```
 # prevents macro from working except if with Parameters
mode = runmode()

if not(mode = "edit") then
        fail("Error : Must use Macro with Parameters")
end if

on edit
        # define input elements a to e as in previous step
        (...)
        return [a, b, c, d, e]
end edit
```

Now, the rest of the macro will implement the passing of the values in a definition using the same names. Here we restrict the operation to the **Visualise** run mode. All other run modes will cause the macro to fail. The rest of the code will be:

```
on visualise(input)
        indata          = input["input data"]
        vf_date         = input["verif date (yyyy-mm-dd)"]
        n_of_days       = input["days"]
        par             = input["parameter"]
        the_area        = input["area"]

        (...)

        fa_diff = fc_an_diff(indata, par, vf_date, n_of_days)

        plot(dw, fa_diff, visdef)
end visualise

# exit program if not visualised
on execute(input)
```

```
                fail("Error : Visualise only")
        end execute

        on save(input)
                fail("Error : Visualise only")
        end save

        on examine(input)
                fail("Error : Visualise only")
        end examine

        on prepare(input)
                fail("Error : Visualise only")
        end prepare
```

Note the code to trap any unwanted action or error in execution - the **Macro** program icon cannot be used except within a **Macro with Parameters** icon; this is done by only allowing the `edit` run mode. The **Macro with Parameters** icon can only be visualised, by explicitly stopping the macro for any other handler / run mode.

To operate, create a **Macro with Parameters** icon and Edit it. Drop the macro program icon inside its icon field. This builds a user interface in the icon editor - provide the input as in the previous step and save the icon. Rename it to **Step10.par** (or whatever you like).

To obtain your output simply visualise the **Macro Parameters** icon.

Apart from the obvious difference in the mode of operation, using **Macro Parameters** allows you to keep the values you have last entered rather than reverting to the defaults as is the case with the `dialog()` function. The idea is that you can have several of these **Macro Parameters** icons each with different sets of input elements.

If you want to see exactly what information a **Macro Parameters** icon stores, click on the icon at the top-left of its editor.

If You Have Extra Time ...

Add a control to the user interface to allow the selection of contouring styles for temperature difference fields. There is another contouring icon, `diff_shade`, which can be incorporated into the macro for this purpose.

Add a control to determine whether or not to apply land and sea shading to the coastline rendering. Note that you can use code similar to this in order to dynamically modify the coastlines:

```
        mycoast = (map_coastline_resolution : "low",
                   ...)  # this variable is a 'definition'

        coast_shade = (map_coastline_land_shade : "on",
                       ...)  # this variable is a 'definition'

        if (user_wants_coast_shading) then
            mycoast = (mycoast, coast_shade)  # merge the two definitions
        end if

        acoast = mcoast(mycoast) # create a coastline out of the definition
```

RUNNING MACRO IN BATCH MODE

So far the tasks carried out have been interactive - they require an operator (you) to be physically present and click mouse buttons for actions to take place. However, this may be limiting and eventually you will want to automate a task or set of tasks (e.g. routine plots of meteorological variables).

This may simply be a question of style (command line exclusivists), but more frequently it is a crucial requirement - either for repetitive tasks or for long tasks that need to be run overnight, for tasks embedded in shell scripts, etc.

Macro programs run in batch mode offer that possibility, particularly when coupled with shell scripts and scheduling procedures.

How to run in batch

Metview runs in batch mode when option `-b` is specified when you first call it. You can run Metview in batch mode while having a Metview interactive session up and running. To run a macro in batch mode simply specify its name on the command line, e.g.:

```
% metview -b macro_name
Or
% metview4_new -b macro_name [use this one for the
ECMWF March 2013 course!]
```

where `macro_name` is the name of the macro you want to run. You can specify a path to the macro if not running from its own directory.

When Metview is run in batch you cannot have on-screen visualisations. Hence, the result of the macro must be one of the following two:

- a plot in a PS, PNG, etc. file
- a data file (GRIB, geopoints, ASCII) with the results of the macro

Step 11 - User Input in Batch Mode

Duplicate the **step7** macro icon and rename the duplicate **step11**.

If running macro in batch, it is frequently/usually required that you pass arguments to it. These have to be provided on the command line, since interactive functionality is not available. The general form of the macro run in batch is:

% metview -b macro_name $input_1$ $input_2$ ... $input_n$

where $input_i$ describes an arbitrary number of input arguments.

To run this macro in batch mode, we need to pass the variables at the top of the macro as command line arguments, e.g.:

```
% metview[1] -b step11 2012-03-01 5  t  TUV_Data
```

But the macro needs to retrieve the arguments from the command line. To do this we need to use the `arguments()` function. The `arguments()` function parses all the command line arguments that follow the macro name and returns them as a list, whose elements can then be addressed/retrieved individually.

It is also good practice to provide a check on the number of arguments. Further checks could be done on the type (number, string, date,...) of the input arguments using the `type()` function.

To implement command line argument retrieval, remove the hardcoded variables sitting at the top of the macro (apart from `the_area`) and replace with the following piece of code :

```
 # command line arguments retrieved and stored in list
input = arguments()

 # check number of args and extract from list
if count(input) <> 4 then
        fail("wrong number of args - needs YYYY-MM-DD STEP PARAM
FILE_NAME")
else
        vf_date    = input[1]
        n_of_days  = input[2]
        par        = input[3]

        home       = getenv("HOME")
        path       = home & "/metview/macro_tutorial/macro_tut1/"
        file_name  = path & input[4]
end if

print ('Date: '      & vf_date & ', Days: ' & n_of_days)
print ('Parameter: ' & par     & ', File: ' & file_name)

 # translate "UV" into a list, since that is what 'read' understands
if (par = "UV") then
    par = ["u", "v"]
end if
```

Since only PS output is to be allowed you can remove the other unused output definitions. The remainder of the macro would remain the same.

To run this macro, type the following on the command line (repeated from above):

```
% metview[1] -b step11 2012-03-01 5  t  TUV_Data
```

NOTE:

The macros run in batch are ideally suited to be incorporated into shell scripts. The above example is still specific in that it looks for the input file in a particular directory. This is the kind of parameter that could be read by the macro from an environment variable (e.g. `$DATADIR`) or passed on the command line to make it more general.

[1] Use `metview4_new` instead of `metview` for the ECMWF March training course

If You Have Extra Time ...

Adapt the macro to accept another input parameter which will be added as a third line of text to the title. On the command line, wrap the text in quotes if it contains any spaces - otherwise Metview will see it as multiple parameters.

USING FUNCTIONS IN MACRO

User written functions in macro can be organised in ways other than keeping them inside the macro program. This is particularly advantageous if the function performs a useful and likely to be repeated task.

Metview offers several ways to organise your code, which we will review briefly. We will take as a working example one of the programs resulting from the first part of the tutorial. We'll use that of Step 4 for simplicity, but this is applicable to any of the other programs. The preparatory steps are:

- Duplicate the **step4** macro and rename to **step12**
- Create a new Macro icon and bring up its editor
- Cut the function text from the **step12** macro and paste to the new macro
- Rename the new macro with the name of the function

You should end with two macros, **step12** with the main code, and **fc_an_diff** containing the function code.

Step 12a - Including a Function

The `include` command literally includes the text of any macro at the insertion point. In our current example to make the function available to the rest of the program, it is enough to add the line:

```
include "fc_an_diff"
```

anywhere in the macro.

The included macro is read at the point where the `include` instruction is found. You can specify absolute or relative path names (understood to be relative to the position of the macro being run, so be careful if you are running from within the macro editor):

```
include ".../uid/metview/mylib/fc_an_diff"
```

In this way you can place small libraries of functions in macro files, stored in a folder of your choice, ready for inclusion.

NOTE:

Inclusion does not require the included macro to be a self contained program or function, any partial fragment of code can be included.

Changes in a program which is included will affect all the macros that include it.

An **`include`** statement is interpreted before the rest of the macro is executed, including lines that precede the **`include`** statement. This means that you cannot, for instance, use a dynamically generated path to find the file to be included.

Step 12b - Function in a User/System Library

This is the method that can be really described as building a library of functions. Its principle is very simple - place macro functions in a particular folder which is searched by the function look- up procedure so they can be called from any macro program without the need for an `include` statement.

In your case, simply drag the **fc_an_diff** macro icon to the folder `~/metview/System/Macros`. From then on you can call this function from within any of your macros. This allows you to build your own personal function library. For a function to be available to all users, you need to place the macros with the functions in a system wide Macro folder - your local Metview developer/guru will be of help.

FORTRAN AND C IN METVIEW MACRO

The ability to embed FORTRAN and C programs within macros is a very powerful feature of the Metview macro language. It extends immensely the scope of the macro language and enables you to make efficient use of existing resources.

FORTRAN or C programs are used in tasks that cannot be achieved by means of a function or combination of functions of the macro language. This happens for example if the task requires calculations which are a function of gridpoint positions. Otherwise, it may be that you already have suitable FORTRAN or C code and the writing of the same task in macro language would simply consume precious time.

Currently the FORTRAN/C Metview macro interface is supported for input data of types GRIB, number, string and vector. BUFR, images and matrices are awaiting implementation. This exercise emphasises GRIB input.

Note that there are three interfaces available: the Macro/FORTRAN Interface (‘mfi’) and the Macro/C Interface (‘mci’) use GRIB_API for GRIB handling (compatible with GRIB editions 1 and 2). A ‘legacy’ interface which uses GRIBEX is available but is deprecated and will not be discussed here – please use one of the other interfaces.

This tutorial focuses on the ‘mfi’ interface, but equivalent example code is also provided for the C interface.

General Approach

The basic principle is that FORTRAN programs are treated and work as any other macro function. In fact, with the proper implementation method, a user cannot distinguish by looking at the calling macro code between a FORTRAN program and a macro function. There are two ways of using a FORTRAN program from a macro: **inlined** and **external**. When a FORTRAN program is inlined, its source code is written directly into the macro’s source file. This is the preferred way to use FORTRAN programs with macro, as the user does not need to separately compile the program - this is done automatically when the macro is run, using compiler settings that are consistent with the Metview installation. The other option is to compile the FORTRAN program into an external executable and reference this from the macro. This has the disadvantage of being less portable (the executable will probably not be portable across platforms) and the user also has to ensure that the compiler settings are correct, and that the correct libraries are linked.

The requirements on either side are:

Macro side - you need only to use the FORTRAN function, e.g ensuring its input data is available and correct, and its output used downstream. If using an externally compiled executable, there will be a single declaration to provide; if using inlined code, then the FORTRAN source will be written into the macro.

FORTRAN side - clearly you need to write the FORTRAN program; if creating an external executable, you will need to compile and debug / test it. To write the code, you have available a suite of FORTRAN routines which do the basics for you. These are known as *interface routines* and they carry out tasks such as:

- get the input arguments
- decode GRIB headers
- create empty outputs (e.g. fieldsets)
- save and set results

At its simplest, the FORTRAN program dealing with a GRIB file is composed of

- a section where input is read and output prepared
- a section (loop) where the fields are loaded, expanded, validated and the processing carried out (usually within a routine) and the result saved
- a section where output is set

If compiling the FORTRAN program externally, the executable has to be specified via an **`extern`** declaration in the macro code, or alternatively must be placed in a folder that Metview scans automatically during the function loading procedure (e.g. the **Macros** folder within the **System** folder).

Simple Example - Advection of a Scalar Field

The Metview macro program

This example demonstrates a simple task which requires you to derive a fieldset from some input fieldset using a FORTRAN program: obtaining the advection of a scalar field which requires a FORTRAN program to compute the gradient of the field.

You could assume that you would have the FORTRAN program doing what you want and for the time being concentrate on the writing of the macro program itself. Assume, therefore that you will have a FORTRAN program called `gradientb` which returns the gradient of a fieldset in its two components. Once you have this, it is a trivial task to compute the advection of the scalar quantity for which you calculated the gradient. The following macro computes the advection of specific humidity `q` at 700 hPa. Create a new **Macro** icon and rename it **q_advection**. Copy the following code into it. Alternatively, copy the pre-prepared **q_advection** file from the Solutions folder and study it.

```
 # set the area we wish to retrieve data from
 #          N,   W,   S,   E

area_xx = [70, -45, 10, 85]


 # Retrieve the specific humidity
q = retrieve(
              date  :    -1,
              param :    "q",
              level :    700,
              grid  :    [1.5,1.5]
              )

 # Get the u and v components of the wind
u = retrieve(
              date  :    -1,
              param :    "u",
              level :    700,
              area  :    area_xx,
              grid  :    [1.5,1.5]
              )
v = retrieve(
              date  :    -1,
              param :    "v",
              level :    700,
              area  :    area_xx,
              grid  :    [1.5,1.5]
              )


 # Compute the gradient of Q
q = gradientb(q)

 # Extract the area we are calculating on
q = read ( area : area_xx, data : q)


 # Compute the advection of Q
a = q[1]*u + q[2]*v
a = -a * (10 ^ 8) # units will be 10e-8 (kg/kg)/sec


 # Plot positive advection in blue, negative in red
contour_common = (
              contour_level_selection_type    :    "interval",
              contour_interval                :    3,
              contour_label                   :    "on",
              contour_label_height            :    0.25,
              contour_highlight               :    "off",
              contour_hilo                    :    "on",
              contour_hilo_type               :    "number",
              contour_hilo_format             :    "F5.1",
              contour_hilo_height             :    0.3
              )

cont_n = mcont(
              contour_common,
              contour_max_level       : -0.0001,
              contour_line_colour     : "red",
              contour_label_colour    : "red",
              contour_lo_colour       : "red"
              )

cont_p = mcont(
              contour_common,
              contour_min_level       : 0.0001,
              contour_line_colour     : "blue",
              contour_label_colour    : "blue",
              contour_hi_colour       : "blue"
              )

 # A plot window
acoast = mcoast(
              map_coastline_resolution        : "low",
                    map_grid_longitude_increment    : 10,
                    map_coastline_land_shade        : "on",
                    map_coastline_land_shade_colour : "cream"
                    )

ps_atlantic = mapview(
                    map_projection  :    "polar_stereographic",
                    area            :    [30,-25,50,65],
                    coastlines      :    acoast
                    )

page = plot_page(
                    view            :    ps_atlantic
                    )

dw = plot_superpage(
                    custom_width    :    29.7,
                    custom_height   :    21,
                    pages           :    page
                    )


 # Now plot the result

plot(dw,a,cont_p,cont_n)
```

The code above is straightforward. The only question remaining is the function `gradientb()`.

The FORTRAN program

`gradientb()` is a FORTRAN program which you write and, in this case, embed in the macro code. The program is structured according to the sections outlined above, containing:

- a section where input is read (`mfi_get_fieldset`) and output prepared (`mfi_new_fieldset`)
- a loop (on the number of fields in the argument fieldset) where the fields are loaded (`mfi_load_one_grib`), expanded (`grib_get_real8_array`, a GRIB_API function), validated and processed (user routines) and the result stored (`grib_set_real8_array`) and saved (`mfi_save_grib`)
- a section where output is set (`mfi_return_fieldset`)

Note the interface routines, all prefixed by `mfi`. Most of this `FORTRAN` code is standard to access and process a `GRIB` fieldset. The user only has to define the routines `VALID()` and `GRAD()`. The first checks whether the properties of the input match the requirements and the second derives the actual gradient field.

`GRAD()` takes the input fieldset, calculates its gradient in its two components. These are saved separately and coded as wind components, so each of these can be accessed separately in the macro for the calculation of the advection.

We will inline this program and so its compilation and linking will be taken care of automatically without us even being aware of it.

The FORTRAN code is listed below, and may be copied into an empty **Macro** icon named gradientb :

```
!
!  "GRADIENTB" COMPUTES THE GRADIENT OF A FIELD
!
!  THIS PROGRAM IS A MODIFIED VERSION OF THE FORMER
!  "GRADIENT" TO TAKE INTO ACCOUNT THE UNITS.
!
!  THE UNITS ARE IN THE INTERNATIONAL SYSTEM
!
!  GRIBEX version:    October,  1996
!  GRIB_API version:  March,    2010
!  MFI version:       November, 2010

  PROGRAM GRADIENTB
  USE grib_api
  IMPLICIT NONE

  INTEGER  fieldset_in, fieldset_out, icnt
  INTEGER  grib_id, isize, istatus, i
  INTEGER  byte_size
  REAL*8,  ALLOCATABLE :: grib_in(:)
  REAL*8,  ALLOCATABLE :: grib_out_u(:)
  REAL*8,  ALLOCATABLE :: grib_out_v(:)

                               !-- GET FIRST ARGUMENT AS A FIELDSET.
                               !-- icnt IS THE NUMBER OF FIELDS
  CALL mfi_get_fieldset( fieldset_in, icnt )

                               !-- CREATE A NEW OUTPUT FIELDSET
  CALL mfi_new_fieldset( fieldset_out )

                               !-- LOOP ON FIELDS
  DO i=1, icnt
                               !-- GET NEXT FIELD FROM INPUT FIELDSET
     CALL mfi_load_one_grib( FIELDSET_IN, grib_id )

                               !-- ALLOCATE ARRAYS, GET FIELD VALUES
     CALL grib_get_size( grib_id, 'values', isize )
     ALLOCATE( grib_in(isize), grib_out_u(isize), grib_out_v(isize) )

     CALL grib_get_real8_array( grib_id, 'values', grib_in, istatus )

                               !-- VALIDATE AND DERIVE OUTPUT
     CALL valid( grib_id )
     CALL grad( grib_in, grib_out_u, grib_out_v )

                               !-- SET OUTPUT AS U COMPONENT OF WIND
     CALL grib_set_real8_array( grib_id, 'values', grib_out_u,
istatus )
     CALL grib_set_int( grib_id, 'paramId', 131 )

                               !-- ADD IT TO THE OUTPUT FIELDSET
     CALL mfi_save_grib( fieldset_out, grib_id )

                               !-- SET OUTPUT AS V COMPONENT OF WIND
     CALL grib_set_real8_array( grib_id, 'values', grib_out_v,
istatus )
     CALL grib_set_int( grib_id, 'paramId', 132 )

                               !-- ADD IT TO THE OUTPUT FIELDSET
     CALL mfi_save_grib( fieldset_out, grib_id )
```

```
                                  !-- RELEASE MEMORY
      CALL grib_release( grib_id )
      DEALLOCATE( grib_in, grib_out_u, grib_out_v )

   END DO
                                  !-- RETURN THE RESULT
   CALL mfi_return_fieldset( fieldset_out )

   STOP
 END


 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 !--
 !--  USER ROUTINE TO CHECK VALIDITY OF INPUT FIELD
 !--  VALID FOR A GLOBAL FIELD, LAT/LONG, 1.5 DEG GRID
 !--

 SUBROUTINE valid( grib_id )
   USE grib_api
   INTEGER grib_id
   INTEGER ivalue
   REAL*8  rvalue

   CALL grib_get_int(grib_id, 'dataRepresentationType', ivalue)
   IF( ivalue .NE. 0 ) CALL mfi_fail("GRID not lat/lon")

   CALL grib_get_real8(grib_id, 'iDirectionIncrementInDegrees',
 rvalue)
   IF( rvalue .NE. 1.5 ) CALL mfi_fail("GRID not 1.5/1.5")

   CALL grib_get_real8(grib_id, 'jDirectionIncrementInDegrees',
 rvalue)
   IF( rvalue .NE. 1.5 ) CALL mfi_fail("GRID not 1.5/1.5")

   CALL grib_get_int( grib_id, 'Ni', ivalue )
   IF( ivalue .NE. 240 ) CALL mfi_fail("GRID not global")

   CALL grib_get_int( grib_id, 'Nj', ivalue )
   IF( ivalue .NE. 121 ) CALL mfi_fail("GRID not global")

   RETURN
 END


 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 !--
 !--  DERIVE GRADIENT OF INPUT FIELD F (BENITO ELVIRA, IM)
 !--  FA = HORIZONTAL GRADIENT, FB = VERTICAL GRADIENT
 !--

 SUBROUTINE GRAD (F, FA, FB)

    !-- DIMENSIONS CORRESPONDING TO 1.5 x 1.5 GRID
   DIMENSION F(240,121), FA(240,121), FB(240,121)

   PI = ACOS(-1.0)
   RT = 6371000.0
   CB = (RT*1.5*PI)/180.0
                                       !-- COMPUTE HORIZONTAL GRADIENT
   DO I = 1, 121

      C = COS( (90.0-I*1.5 + 1.5)*PI/180.0 )
      FA(1,i) = (F(2,i)-F(240,i)) / (2.0*C*CB)
      FA(240,i) = (F(1,i)-F(239,i)) / (2.0*C*CB)

      DO J = 2, 239
```

```
               FA(j,i) = (F(j+1,i)-F(j-1,i)) / (2.0*C*CB)
            END DO

         END DO
                                                          !-- COMPUTE VERTICAL GRADIENT
         DO I = 1, 240

            FB(i,1) = 0
            FB(i,121) = 0
            DO J = 2, 120
               FB(i,j) = (F(i,j+1)-F(i,j-1)) / (-2.0*CB)
            END DO

         END DO

         RETURN
       END
```

Embedding the FORTRAN program

If we wished to embed the FORTRAN program directly into our macro `q_advection`, then we would need only type/paste the code into the macro source file and surround it with the following lines:

```
extern gradientb(f:fieldset) "fortran90" inline
... (FORTRAN code here)
end inline
```

Note that we could have used `"fortran"` here if our code was FORTRAN 77-specific.

We would now have one source file containing both the macro code and the FORTRAN code. This is fine if we will not wish to reuse the FORTRAN program for another project.

However, we may wish to make this FORTRAN program available to other macros. In this case, it can be placed in its own file, along with the **`extern`** ... **`inline`** header/footer lines surrounding it as shown above. It can now be included by a macro in the same way as any standard macro function, using the **`include`** directive or placing it in a user or system library, as discussed in Steps 12 a, b and c. In the example given in the Solutions folder, we inline the FORTRAN source code in a separate file (`gradientb`) and **`include`** it in our main macro, using the following line at the top of the macro:

```
include "gradientb"
```

Using an externally compiled FORTRAN program

If you are interested in compiling your FORTRAN program separately and using the executable as a macro function, you should consult the Metview User Guide Vol II, section “Using FORTRAN In Macro“.

To use an externally compiled FORTRAN executable, you need to decide whether to declare the function explicitly or have the macro load the function automatically.

To declare the function explicitly, use the **`extern`** keyword. For this, introduce the following line at the top of the body of the macro program:

```
        extern gradientb(f:fieldset) "gradientb"
```

To have the macro program load the routine automatically you can place it in a user or system library, as discussed in Steps 12 a, b and c. Once this is done, the `FORTRAN` function is used exactly like any other macro function, with no difference in the syntax.

Note

In order for Metview Macro to determine whether a file in a Macro folder is a FORTRAN executable or a macro, it checks the return value of the UNIX `file` command on the file. If it includes the word 'executable', then it is assumed to be a FORTRAN program; otherwise, it is assumed to be a Metview macro.

Equivalent C Program

The following code shows the C equivalent of the above FORTRAN program, also using GRIB_API to handle the data. See the Metview User Guide for more details of the C interface.

```
extern gradientb(f:fieldset) "C" inline

 /*
    "gradientb" - computes the gradient of a field.
 */


#include <stdio.h>
#include <string.h>
#include <math.h>
#include "macro_api.h"



 /*
     check_data_ok - checks whether the data is in a format we can
 work with - returns 1 if it is, 0 otherwise
 */

int check_data_ok (grib_handle *gh)
{
    char grid_type [32];
    int  len = sizeof(grid_type);

    grib_get_string (gh, "typeOfGrid", grid_type, &len);

    if (strcmp (grid_type, "regular_ll"))
    {
        printf ("Data is in wrong grid type (%s) - it should be
'regular_ll'\n", grid_type);
        return 0;
    }


     /* if we got to here, then it was all good */

    return 1;
}
```

```
void compute_gradient (grib_handle *gh, double *vals_out_u, double
*vals_out_v)
{
    #define idx(X,Y) ((Y)*x_num + (X))

    double pi = acos(-1.0);
    double rt = 6371000.0;
    double cb = (rt * 1.5 * pi) / 180.0;
    long x_num, y_num;
    double x_inc, y_inc;
    double *vals;
    int ret, len;
    int i, j;

    if (!check_data_ok(gh))
    {
        mci_fail("Data not ok for this function - aborting.");
    }


    GRIB_CHECK(grib_get_long   (gh, "numberOfPointsAlongAParallel",
&x_num), 0);
    GRIB_CHECK(grib_get_long   (gh, "numberOfPointsAlongAMeridian",
&y_num), 0);
    GRIB_CHECK(grib_get_double (gh, "iDirectionIncrement", &x_inc),
0);
    GRIB_CHECK(grib_get_double (gh, "jDirectionIncrement", &y_inc),
0);

    x_inc /= 1000.0; /* increments are stored in millidegrees */
    y_inc /= 1000.0; /* increments are stored in millidegrees */


    /* check that the data is global */

    if ((x_num * x_inc != 360.0) || ((y_num-1) * y_inc != 180.0))
    {
        printf ("Data is not global (%d x %f, %d x %f)\n", x_num,
x_inc, y_num, y_inc);
        mci_fail("Data not ok for this function - aborting.");
    }



    len = x_num * y_num;

    vals = (double *) malloc (len * sizeof(double));

    printf ("getting %d elements...\n", len);
    ret = grib_get_double_array(gh,"values",vals, &len);
    printf ("got %d elements...\n", len);

    if (ret == GRIB_SUCCESS)
    {
         /* COMPUTE HORIZONTAL GRADIENT */

        for (i = 0; i < y_num; i++)
        {
            double c = cos((90.0 - (i * y_inc) + x_inc) * pi/180.0);

            if ((fabs(c) < 0.00001))
            {
                c = 0.00001;
            }

            vals_out_u[idx(0,   i)] = (vals[idx(1, i)] -
vals[idx(239, i)]) / (2.0 * c * cb);
```

```
            vals_out_u[idx(239, i)] = (vals[idx(0, i)] -
vals[idx(238, i)]) / (2.0 * c * cb);

            for (j = 1; j < x_num-1; j++)
            {
                vals_out_u[idx(j,i)] = (vals[idx(j+1,i)] -
vals[idx(j- 1,i)]) / (2.0 * c * cb);
            }
        }


        /* COMPUTE VERTICAL GRADIENT */

        for (i = 0; i < x_num; i++)
        {
            vals_out_v[idx(i,0)]   = 0;
            vals_out_v[idx(i,120)] = 0;

            for (j = 1; j < y_num-1; j++)
            {
                vals_out_v[idx(i,j)] = (vals[idx(i,j+1)] -
vals[idx(i,j-1)]) / (-2.0 * cb);
            }
        }
    }

    else
    {
      printf(">>> ERROR: grib_get_double_array returned %d\n", ret);
    }
}



int main()
{
    grib_handle* gh_a = NULL;
    grib_handle* gh_u = NULL;
    grib_handle* gh_v = NULL;
    void*   grib_id = NULL;

    int   num_fields   = 0;    /*-- field count --*/
    int    ret_a   = 0;    /*-- function return value --*/
    int    ret_b   = 0;    /*-- function return value --*/
    int    i;
    size_t len_a   = 0;    /*-- number of grid point values --*/
    size_t len_b   = 0;    /*-- number of grid point values --*/
    void*   grib_out_id = NULL; /*-- return GRIB id ptr, fieldset
handle */
    double *outvals_u = NULL;     /*-- array for grid point values */
    double *outvals_v = NULL;     /*-- array for grid point values */


     /* load the input grib fields */

    grib_id = mci_get_grib_id_ptr (&num_fields);


     /* loop on the input fields */

    for (i = 0; i < num_fields; i++)
    {
        /* get arrays of values from the fields */

        gh_a = mci_load_one_grib   (grib_id);

        grib_get_size(gh_a,"values",&len_a);
        printf("GRID size: %d\n",len_a);
        grib_out_id = mci_new_grib_id_ptr();  /* create return GRIB
id ptr */


        /* allocate memory for the result arrays */

        outvals_u = (double *) malloc (len_a * sizeof (double));
        outvals_v = (double *) malloc (len_a * sizeof (double));


        /* compute the gradient fields */

        compute_gradient (gh_a, outvals_u, outvals_v);


        /* create new fields using these results */

        gh_u = grib_handle_clone (gh_a);
        gh_v = grib_handle_clone (gh_a);


        /* set their values using the arrays we've computed */

        grib_set_double_array(gh_u, "values", outvals_u, len_a);
        grib_set_double_array(gh_v, "values", outvals_v, len_a);


        /* set their parameters to U and V */

        grib_set_long (gh_u, "indicatorOfParameter", 131);
        grib_set_long (gh_v, "indicatorOfParameter", 132);


        /* save them as the result of this inline program */

        mci_save_grib   (grib_out_id, gh_u);
        mci_save_grib   (grib_out_id, gh_v);
        mci_return_grib (grib_out_id);

        if (outvals_u != NULL) free(outvals_u);
        if (outvals_v != NULL) free(outvals_v);

        grib_handle_delete(gh_a);
        grib_handle_delete(gh_u);
        grib_handle_delete(gh_v);
    }

    free(grib_id);

    return 0;
}
end inline
```