Metview Macro Advanced GRIB Handling 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.

Introduction

This tutorial aims to introduce some advanced concepts in handling GRIB data. In particular, it will cover:

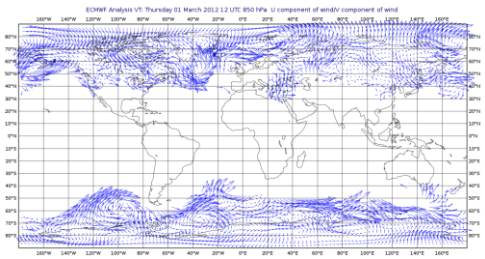
- masking fieldsets
- extracting point values
- extracting meta-data

Relevant reference information can be found here:

https://software.ecmwf.int/metview/Functions+and+Operators+on+Fieldsets

Masking one Field Based on Values in Another

For this exercise, we will take a wind field and mask out values which are above zero degrees Celsius according to a temperature field, keeping just the grid points where the temperature is below zero.



Fieldset masking operations depend primarily on two functions, `bitmap` and `nobitmap`. Their documentation is reproduced here:

```
fieldset bitmap (fieldset,number)
fieldset bitmap (fieldset,fieldset)
```

Returns a copy of the input fieldset (first argument) with zero or more of its values replaced with grib missing value indicators. If the second argument is a number, then any value equal to that number in the input fieldset is replaced with the missing value indicator. If the second argument is another fieldset with the same number of fields as the first fieldset, then the result takes the arrangement of missing values from the second fieldset. If the second argument is another fieldset with one field, the arrangement of missing values from that field are copied into all fields of the output fieldset. See also nobitmap .

```
fieldset nobitmap ( fieldset,number )
```

Returns a copy of the input fieldset (first argument) with all of its missing values replaced with the number specified by the second argument. See also bitmap.

To get you started, here is some code to read the temperature and wind fields from the given GRIB file. Note that this is very simplistic and normally you would properly filter the data, but this shows another way to select fields from a fieldset if you already know their ordering.

```
tuv_data = read('TUV_Data')
t  = tuv_data[1]    # select just the first field (Temperature)
uv = tuv_data[2,3]  # select the 2nd and 3rd  fields (U/V)
```

Remember that the data are stored in Kelvin, not Celsius, and that 0°C ~= 273.15°K.

Now have a go at using the `bitmap` function to perform the mask. If you would like to try this by yourself, you might find the following hints helpful.

Hints

Working backwards from what we want:

- we want to create a mask field which we can apply to the wind field using the `bitmap(fieldset, fieldset)` function.
- this mask field will contain missing values where the temperature field is above zero (we do not care what the other values in the mask field are – the `bitmap(fieldset, fieldset)` function will simply copy across the pattern of missing values)
- to create these missing values in the first place, we will use the `bitmap(fieldset, number)` function; for this to work, we will need to set all the points we wish to 'become missing' to one unique value
- logical operators such as <, > and = return a new fieldset where all the values are either 1 or 0, depending on whether they pass the test
- plot the results of intermediate steps to check that they are what you expect

Solution

As is often the case, the actual code to perform this task is much shorter than the explanation! Here is one solution with just 3 lines of code:

```
cold_mask = t < 273.15           # cold_mask contains just 1s and 0s
cold_mask = bitmap(cold_mask, 0) # turn 0s into missing values

uv_cold_only = bitmap(uv, cold_mask) # apply the temperature mask to
the u/v
```

An alternative, but equivalent, piece of logic could have been:

```
cold_mask = t >= 273.15          # cold_mask contains just 1s and 0s
cold_mask = bitmap(cold_mask, 1) # turn 1s into missing values
```

Notice something which went unsaid here: `uv` and therefore `uv_cold_only` are both fieldsets with 2 fields – the `bitmap` function applied the mask to both fields. If you look again at the documentation for this function, you will see that this is the expected behaviour.

A common use for masking operations is to use a land/sea mask to remove land or sea points from another field.

Extracting Point Values

Individual Points

The easiest way to find the value at a particular location in a GRIB field is to use the `nearest_gridpoint`, `nearest_gridpoint_info` and `interpolate` functions. These all take a fieldset as their first argument; their subsequent arguments provide the location(s) of the point(s) to be extracted and can be a geopoints variable, a list containing a latitude and a longitude, or else latitude and longitude provided as separate arguments. The `nearest_gridpoint` functions also accept as their second and third arguments vectors of latitudes and longitudes for more efficient extraction of multiple values.

In a new macro, read any GRIB file and try to extract the values from some points. Here is an example:

```
tuv_data = read("TUV_Data")

lat = 50
lon = 10

p = nearest_gridpoint(tuv_data, lat, lon)
print(p)

n = interpolate(tuv_data, lat, lon)
print(n)

i = nearest_gridpoint_info(tuv_data, lat, lon)
print(i)
```

Each function will return a list of results, one for each field in the fieldset. You can see the the results of `p` and `n` are not the same; i gives more than just the raw values – it gives the coordinates of the nearest points so you can see that they are not exactly at the location we specified. The following output has been abridged by only considering the first 3 fields (`tuv_data = tuv_data[1,3]`).

```
p: [282.181091309,1.51431274414,-0.407638549805]
n: [281.653313531,1.36153496636,-0.740971883138]
```

```
i:
[(value:282.181,latitude:49.5,longitude:10.5),(value:1.51431,latitude:
49.5,longitude:10.5),(value:-0.407639,latitude:49.5,longitude:10.5)]
```

The type of `i` is 'list of definitions'. The following line of code shows how to access particular elements of it:

```
print('first latitude: ', i[1].latitude)
```

Lists of Points

If you have a geopoints file containing a set of locations for which you wish to extract the values from a GRIB field, the above functions will also accept your geopoints variable as their second argument, returning a new geopoints variable with its fields updated to contain the values from the GRIB field. Look at the supplied geopoints file `locations.gpt`; the following code will use it to extract values from a set of points:

```
locations = read("locations.gpt")
p = nearest_gridpoint(tuv_data, locations)
n = interpolate(tuv_data, locations)
```

If you set your macro to `return` one of these results, you can right-click **examine** the macro to see what is generated.

All Points

If you wish to extract all the data points from the GRIB into a text file, the easiest way is to use the GRIB to Geopoints icon and use it to generate the equivalent macro code. You can then use the `write` function to write it to a file. Note that this will only convert the first field in the fieldset – if you want to convert all the fields, you will need to write a loop. Try it, or look in the solution below.

```
tuv_data = read("TUV_Data")

 # write a single field
geo = grib_to_geo(data : tuv_data)
write (getenv('SCRATCH') & '/T.gpt', geo)

 # write all fields
for i = 1 to count(tuv_data) do
    geo = grib_to_geo(data : tuv_data[i])
    write (getenv('SCRATCH') & '/tuv_' & i & '.gpt', geo)
end for
```

You can also extract the latitudes, longitudes and values as `vector` variables like this:

```
tuv_data = read("TUV_Data")
data = tuv_data[1]  # look only at the first field

lats = latitudes(data)
lons = longitudes(data)
vals = values(data)
```

As well as performing mathematical operations on them, the values can also be written out to a text file in a format of your choosing. The syntax is the same as the `print` command, except you should open a file handle.

This line shows how to access the individual grid points:

```
i = 100
print('i=', i, ', lat=', lats[i], ', lon=', lons[i], ', val=',
vals[i])
```

Try to write a loop which writes all the data values to a file. Sample code is provided here:

```
 # write the whole lot out to a file
fh = file(getenv('SCRATCH') & '/field_points.txt')
for i = 1 to count(vals) do
    write(fh, 'index=', i, ', lat=', lats[i], ', lon=', lons[i], ',
val=', vals[i], newline)
end for
fh = 0 # close the file handle
```

Extracting Meta-Data from the GRIB Header

Metview Macro provides functions for accessing meta-data from GRIB headers using GRIB_API. The functions are prefixed with grib_get, followed by the internal data type that GRIB_API should use to retrieve the value (`_long`, `_double`, `_string`, `_long_array` and `_double_array`). The Metview documenation page referred to at the beginning gives details.

The following example shows a simple usage:

```
tuv_data = read("TUV_Data")
levs = grib_get_long(tuv_data, 'level')
print(levs)
```

Here, `levs` will be a list of numbers – one for each field in the fieldset. The output in this case will be:

```
[850,850,850,300,300,300,850,850,850,300,300,300,850,850,850,300,300,3
00]
```

Inspect any GRIB file with Metview's **examine** tool to see the available keys for that GRIB (in this example, '`level`' is the key), then write some code to extract this data. Note that not all keys are valid for all GRIB files!

The following code will find the indexes of all fields whose units are Kelvin:

```
units = grib_get_string(tuv_data, 'units')
k = find(units, 'K', 'all')
```

The `find` function (new in Metview 4) normally returns the index of the first occurrence of an item in a list; if a third argument, '`all`', is given, it will return a list of all occurrences.

This technique could be used to write a specialised GRIB filter. Try it – it just involves looping through the list of indexes returned by `find()`.

Consider the function `grib_get()`, which offers a more efficient way to retrieve multiple keys in a single function call (see the documentation).