Meteorology and Python

desperately trying to forget technical details

Claude Gibert,
Europython 2011

synopticView

Background

- Meteorology - NWP – Numerical Weather Prediction
- ECMWF – European Centre for Medium-Range Weather Forecasts.
- GMAO – Global Modelling and Assimilation Office at NASA. Run a numerical forecast to calibrate satellite sensors.
- Atmospheric models running from initial conditions: the analysis – optimisation function – observations
- Forecast verification – statistics computed against the verifying analysis and observations
- Observation statistics – monitoring of observing systems

History

- It all started in 2004.

- Design a forecast verification package:

  1) offer an interface which doesn’t require programming, but can be used in batch processing
  2) retrieve raw data from the archive
  3) compute statistics
  4) store statistics
  5) extract and display statistics (plots)

Thoughts (1)

- offer an interface which doesn't require programming, but can be used in batch processing:
  - this called for a "descriptive" language which could have been anything from XML to a "home made" syntax
  - I have always been against proprietary languages which proliferated. Good examples of scientific software were grads, matlab, IDL, pvwave. Not a good idea.
  - There was an interface for accessing the archive which was request based:

```
retrieve,
    param  = T,
    level  = 500/850,
    type   = forecast,
    expver = 0001
```

Thoughts (2)

- retrieving raw data is difficult:
  - interface the multiple PB archive and handle its complexity
  - interface data decoding libraries (Fortran and C) to decode the data from the archive, two main formats
  - have sufficient knowledge of meteorological parameters to derive some of them from others which are in the archive.

- compute statistics: this is relatively trivial, but it is the stage preceding the computing which requires the most attention:
  - to pair and possibly aggregate data correctly
  - this means associating the right forecast with the right reference or the observations from a same observing system together.

Thoughts (3)

- store statistics. The example of the "retrieve" request shows few parameters, in fact there are probably 15 different ones for each statistic. This is a typical meteorological archive, the size of the metadata is about 20 times the size of the data.
- A SQL database was used although I have now been looking at Non-SQL databases. This is out of the scope of this presentation.

```
retrieve,
    param  = T,
    level  = 500/850,
    type   = forecast,
    expver = 0001
```

Thoughts (4)

- extract and display (plot) statistics. A flexible system should offer a decent choice of plots, but the user should have very good control over them:
  - choice of bars, curves etc…
  - way of grouping curves based on metadata
  - ways of organising plots
  - control over titles and legends

Decisions (1)

- this package was going to do what most people in the organisation already did in separate programs and languages (Fortran and C) for many different purposes:
  - access the archive
  - decode data
  - identify data
  - post-process data
  - pair data if applicable
  - compute
  - store / plot

- at best the code reuse was “copy-and-paste”

synopticView

Choices (1)

- it made sense to develop a framework first and then build the verification package with it:
  - hopefully if users invested in the framework, a common tool would simplify maintenance and development efforts
  - the ultimate goal of developing an application would be met
  - offers a platform for new developments
- the question was: in which language should this be done?

Choices (2)

- I needed a language which would be easy to interface with C, C++ and possibly Fortran. I also needed to glue different libraries together.

- Then I learnt Python in 1 day using “Dive into Python”, and actually tried it.

Purpose of the talk

- I am going to describe how the language for the user was developed to be both:
  - an efficient way of conveying information to the application, let’s be modest, a “fancy” argument system
  - a way of supporting the developer, by guaranteeing the validity of the input and configuration defaults.

- I would like to show how the definition of the concept of a directive can contribute to the creation of complex requests for a plotting system.

The language

- Using the Python interpreter for the interface was the best option to insert custom code. I went back to:

```
retrieve,
    param  = T,
    level  = 500/850,
    type   = forecast,
    expver = 0001
```

this could easily be mapped to:

```
retrieve(
    param  = 'T',
    level  = [500,850],
    type   = 'forecast',
    expver = '0001'
)
```

- this is called a **Directive**

synopticView

The directive

- Directives are basically Python dictionaries to which semantics are added, to help both the programmer and the user:
  - list of valid keywords (dictionary keys)

```
{
    "directive": "store",
    "keywords": {
         "name": {
         },
         "age": {
         },
         "nationality": {
         }
    }
}
```

Why dictionaries? Because they are part of the core of Python and they make it awesome.

```
observation = {
'directive': 'obsidentifier',
'keywords' : {
        'date': { 'type’: int },
        'domain_name': {
            'alias': ['domain'],
            'default_value': ['global’]
        },
        'variable': {
        },
        'level': {
            'alias': 'channel',
            'type’: float,
            'optional': True
        },
        'type': {
            'validate': ['ValidateChoice','ob','im'],
            'default_value': 'ob’,
            ‘unique’: True
        },
        ‘kt’: { ‘type’: int, },
        ‘kx’: { ‘type’: int, }
    }
}
```

synopticView

```
observation = {
'directive': 'obsidentifier',
'keywords' : {
        'date': { 'type': int },
        'domain_name': {
            'alias': ['domain'],
            'default_value': ['global']
        },
        'variable': {
        },
        'level': {
            'alias': 'channel',
            'type': float,
            'optional': True
        },
        'type': {
            'validate': ['ValidateChoice','ob','im'],
            'default_value': 'ob',
            'unique': True
        },
        'kt': { 'type': int, },
        'kx': { 'type': int, }
    }
}
```

```
class Observation(Directive):

    def __init__(self,*args,**kwargs):
        super(Observation,self).__init__(*args,**kwargs)
        self.checkLanguage()

    def languageReader(self):
        return DirectiveReader()
```

```
print Observation()
…
In directive observation:
Keyword variable is missing and is required
Keyword kt is missing and is required
Keyword date is missing and is required
Keyword kx is missing and is required
```

```
print Observation(
    variable = 'omf',
    kt = [4,5],
    kx = 220,
    type = ['ob'],
    date = 2011020312,
    domain = ['europe']
)
…
observation:
    domain_name = ['europe']
    variable = ['omf']
    kt = [4, 5]
    date = [2011020312]
    kx = [220]
    type = ob
…
type = 'wrong',
…
In directive observation:
Validation error for keyword type. The value: 'wrong'
     is not in the set: ob, im
```

synopticView

Argument checking

- As the directive system is defined here, it can be useful to format arguments to methods, either for the lifetime of the application or only at debug stage:

```
from checkargs import checkArgs

checkArgs.register('__call__',dict(
    directive = '__call__',
    keywords = dict(
    a = dict(unique = True),
    b = dict(alias = 'd', optional = True),
    c = dict(default_value = '12', type = int))
    )
)

class MyClass(object):

    @checkArgs
    def __call__(self,*args,**kwargs):
        return kwargs

print MyClass()(a = [12],d = 10)
```

```
{'a': 12, 'c': [12], 'b': [10]}
```

Directive

- Defining semantics for a directive is like working at class level when writing code. Just the same way, inheritance and specialisation are available for directive definition:

```
surface_observation = {
'directive': 'surface_observation',
‘inherit_from’: ‘observation’,
'keywords' : {
        ’level': { ’default_value’: 0 },
        ‘station_height’: { ‘unique’ : True }
    }
}
```

- Class inheritance is not required to mimic inheritance in directive definition.
- Multiple inheritance is supported.

synopticView

Directive: specialisation

```
observation = {
'directive': 'obsidentifier',
‘specialise_from’: {
   ‘type == “im”’: ‘impact’,
   ‘type == “ob”’: ‘rawobs’,
}
'keywords' : {
       'type': {
            'validate': ['ValidateChoice','ob','im'],
            'default_value': 'ob’,
            ‘unique’: True
        },
        etc…
    }
}
{
    "directive": "impact",
    "keywords" : {
        "variable": {
            "default_value": "xvec"
        }
    }
}
```

synopticView

Directive behaviour

- An object instance from a class inheriting from Directive can also inherit from other instances. This is some way of merging dictionaries with different flavours.

- child.inherit_from(parent): assign to child all keys from parent which:
  - are defined in its language and
  - are not defined in child or
  - are a default value in child and not in parent

- child.overwrite_from(parent):assign to child all keys from parent which:
  - are defined in its language and
  - are not defined in child or
  - are a default value in child

- recursively
- __setitem__ is overloaded to keep track of default values and language.

Plotting

- Plotting is not simple and it seems that it has always more or less been a “semi-manual” process. Even when the graphics software provides good support (e.g. matlab, matplolib) the user normally specifies manually:
  - the data for each curve,
  - the legends,
  - the title

- Most graphical packages do not provide sufficient support for automatic plotting, trying to figure out what to use, and are probably not sufficiently object orientated, for example:
  - xaxis, yaxis methods, plot_date etc…
  - there is always a way around but the good stuff is hardcoded

Plotting

- However, knowing the data to be plotted is normally the biggest problem. Observations have the following attributes:
  - kt
  - kx
  - level
  - domain_name
  - date
  - statistic

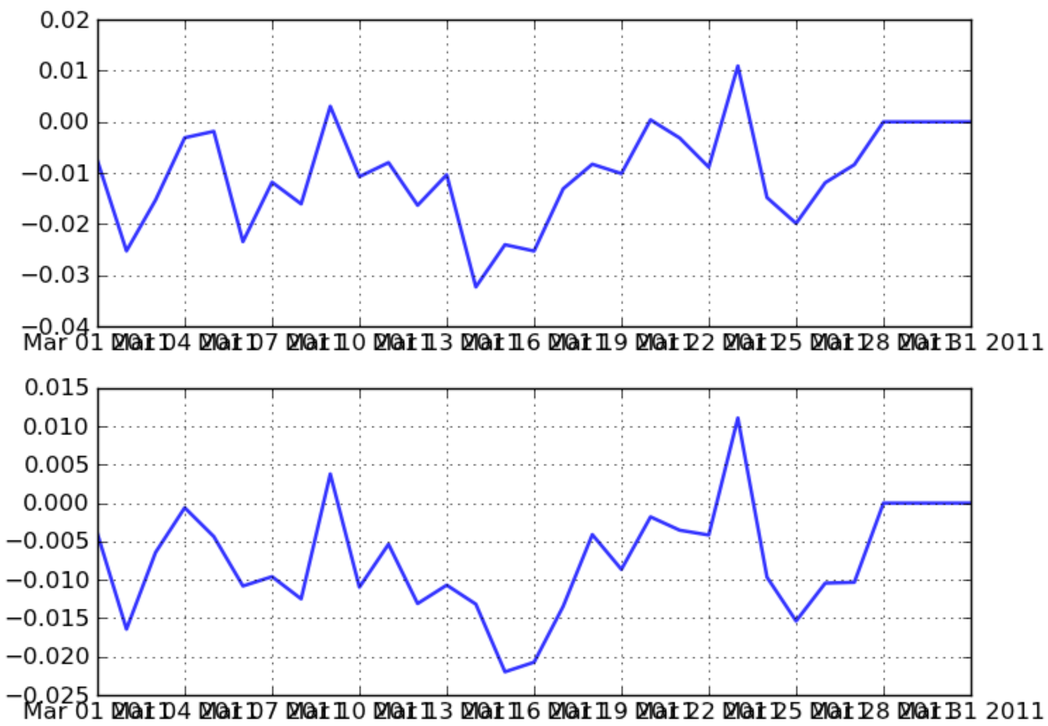
- How can we specify that we want n curves, each of them is one combination of kt, kx, level and we want a different domain for each plot?
- How can we specify that we want n curves, each of them is one combination of kt, kx, domain and we want a different level for each plot?

Document – Plot – Curve Model

- Each directive inherits from observation:
  - a document object contains:
    - attributes related to graphics
    - attributes related to layout and output
    - attributes related to the data being plotted
    - possibly title information
    - a list of plot objects
  - a plot object contains
    - attributes related to graphics
    - attributes related to the data being plotted
    - title information
    - a list of curve objects
  - a curve object contains:
    - attributes related to graphics
    - attributes related to the data being plotted
    - legend information

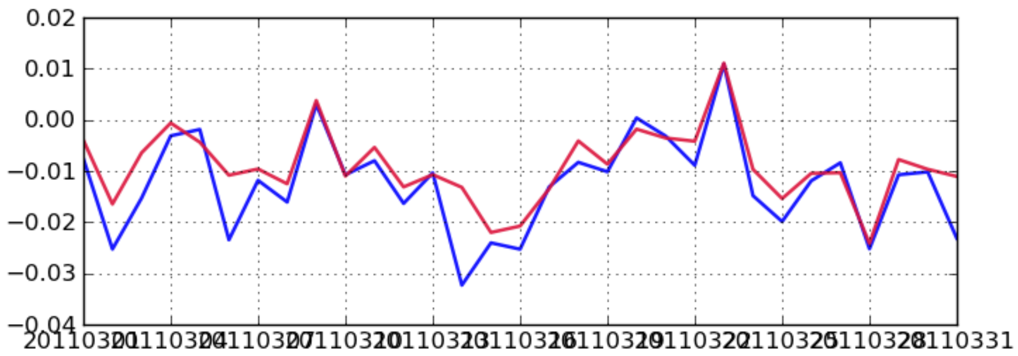
```
d = obsdocument(
    plot = [
        timeseriesplot(
            date = Dates(2011030100,2011033100,24),

            curve = line(
                kx = [120,220,221,132,229,232],
                kt = [4,5,11,44]
            )
        ),
    ],
    type = 'im',
    level = 1000,
    statistic = 'rate',
    domain = ['global','n.hem'],
    layout = [1,2]
}
```



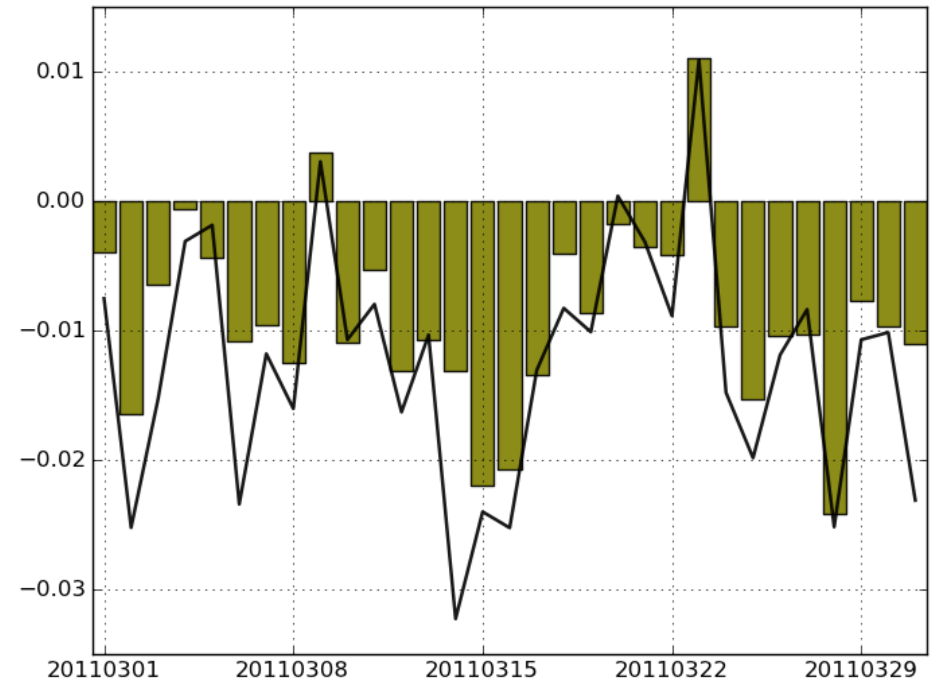
```
d = obsdocument(
    plot = [
        timeseriesplot(
            date = Dates(2011030100,2011033100,24),
            domain = ['global','n.hem'],
            curve = line(
                kx = [120,220,221,132,229,232],
                kt = [4,5,11,44]
            )
        ),
    ],
    type = 'im',
    level = 1000,
    statistic = 'rate',

    layout = [1,2]
}
```



synopticView

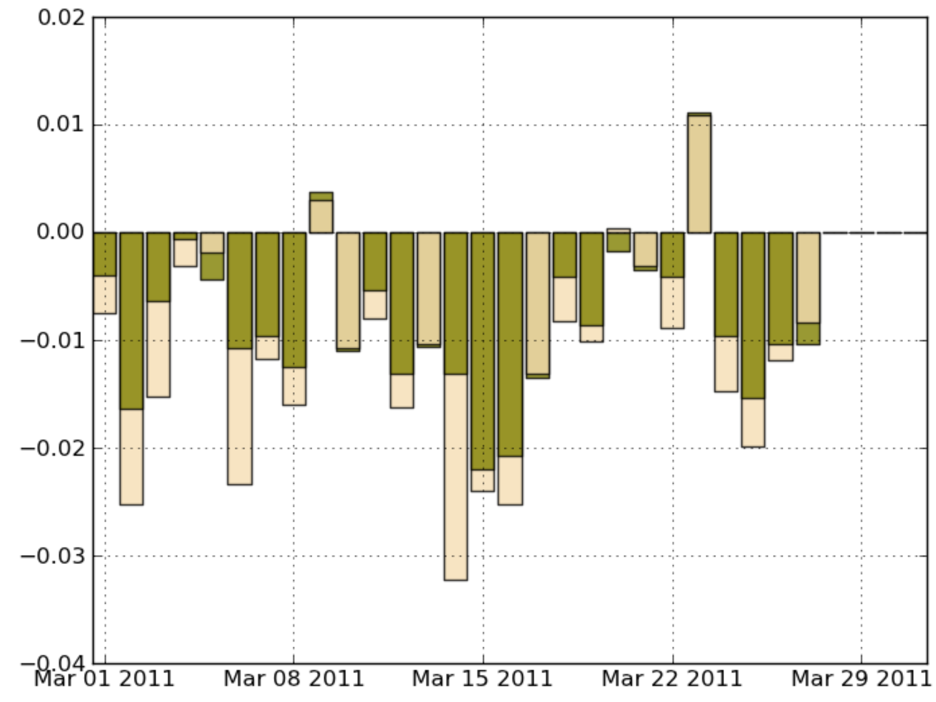
```
d = obsdocument(
    plot = [
        timeseriesplot(
            date = Dates(2011030100,2011033100,24),
            domain = ['s.hem','tropics'],
            kx = [120,220,221,132,229,232],
            kt = [4,5,11,44],
            curve = [
                line(
                    domain = 'global',
                    graphics = graphics(color = 'black')
                ),
                bar(  ← became bar
                    domain = 'n.hem',
                    graphics = graphics(c
                ),
            ],
            yaxis = axis(
                min = -0.035,
                max = 0.015
            ),
       ),
    ],
    type = 'im',
    level = 1000,
    statistic = 'rate',
)
```



synopticView

```
d = obsdocument(
    plot = [
        timeseriesplot(
            date = Dates(2011030100,2011033100,24),
            domain = ['global','n.hem'],
            curve = bar(           # <-- was line became bar
                kx = [120,220,221,132,229,232],
                kt = [4,5,11,44]
            )
        ),
    ],
    type = 'im',
    level = 1000,
    statistic = 'rate',
)
```

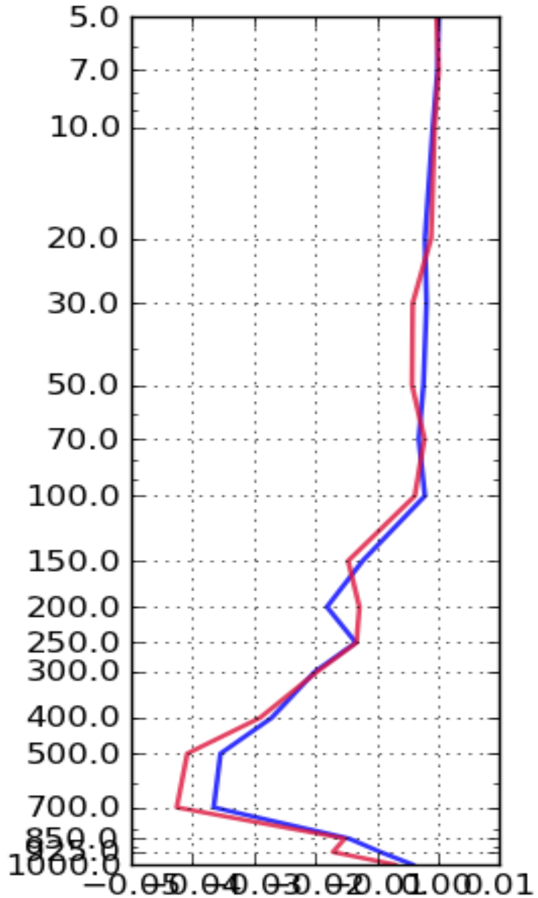
```
 "_combinable": {
      "default_value": [
            "domain_name",
            "statistic”,
             "level”
       ]
 },
 "_index": {
      "default_value": "date”
}
```



synopticView

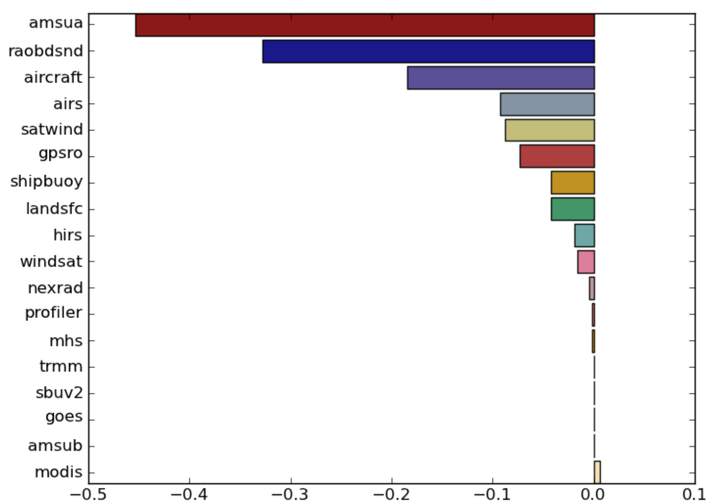
```
d = obsdocument(
    plot = [
        verticalxsection(
            statistic = 'rate',
            level = levels,
            domain = [‘n.hem’,’s.hem’]
            curve = [
                line(
                    kx = 220,
                    kt = [4,5],
                    date = Dates(2010090100,2010103100,24),
                 ),
             ],
        ),
    ],
    type = 'im',
    domain = 'global',
)

 "_combinable": {
     "default_value” [
         "domain_name",
         "statistic”
      ]
},
 "_index": {
        "default_value": ”level”
}
```

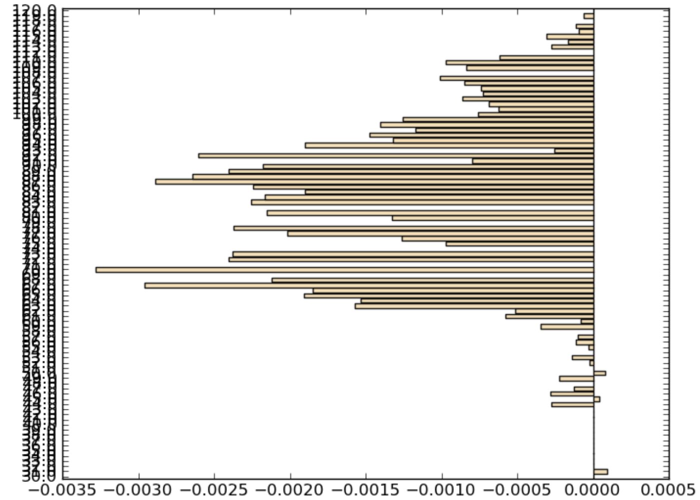


synopticView

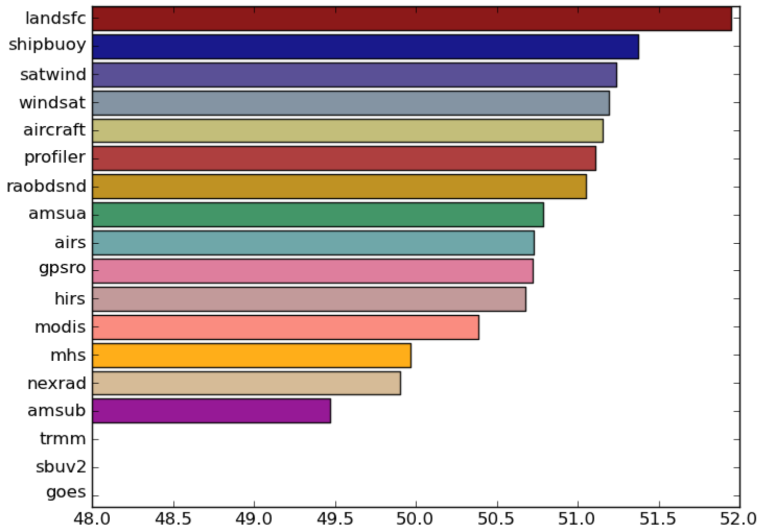
Gallery



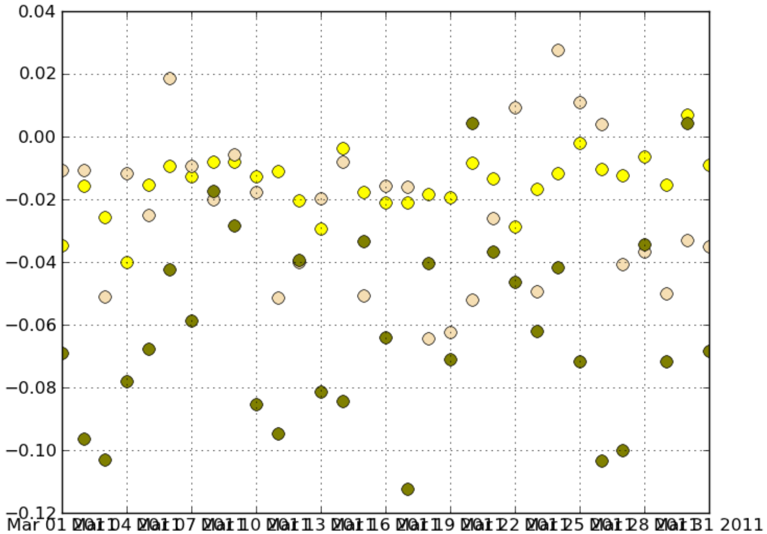
amsua
raobdsnd
aircraft
airs
satwind
gpsro
shipbuoy
landsfc
hirs
windsat
nexrad
profiler
mhs
trmm
sbuv2
goes
amsub
modis
−0.5 −0.4 −0.3 −0.2 −0.1 0.0 0.1



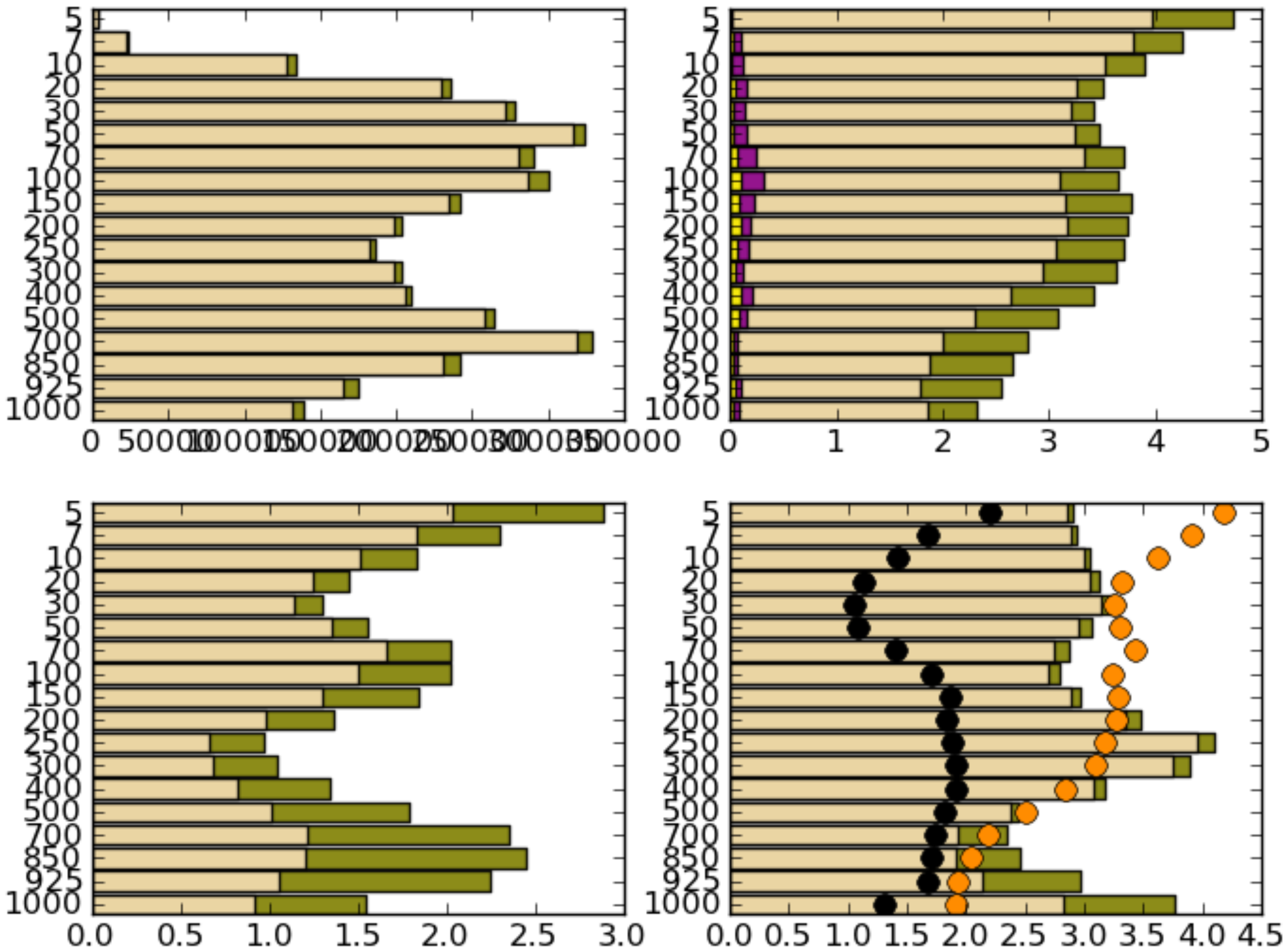
−0.0035 −0.0030 −0.0025 −0.0020 −0.0015 −0.0010 −0.0005 0.0000 0.0005



landsfc
shipbuoy
satwind
windsat
aircraft
profiler
raobdsnd
amsua
airs
gpsro
hirs
modis
mhs
nexrad
amsub
trmm
sbuv2
goes
48.0 48.5 49.0 49.5 50.0 50.5 51.0 51.5 52.0

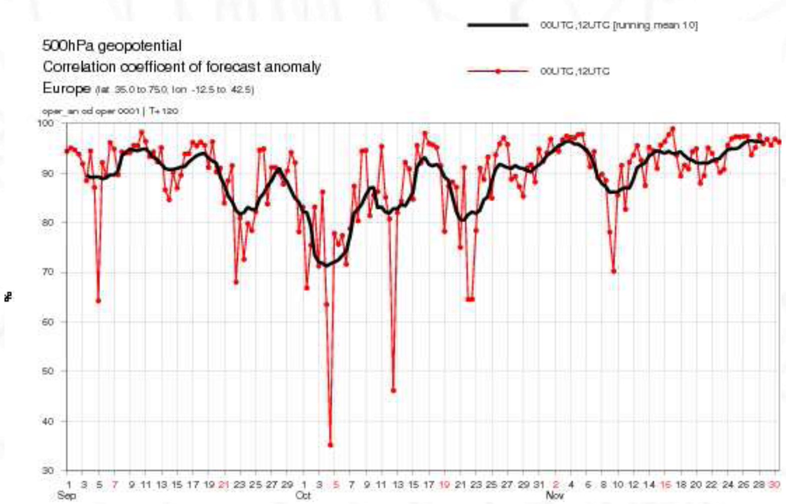


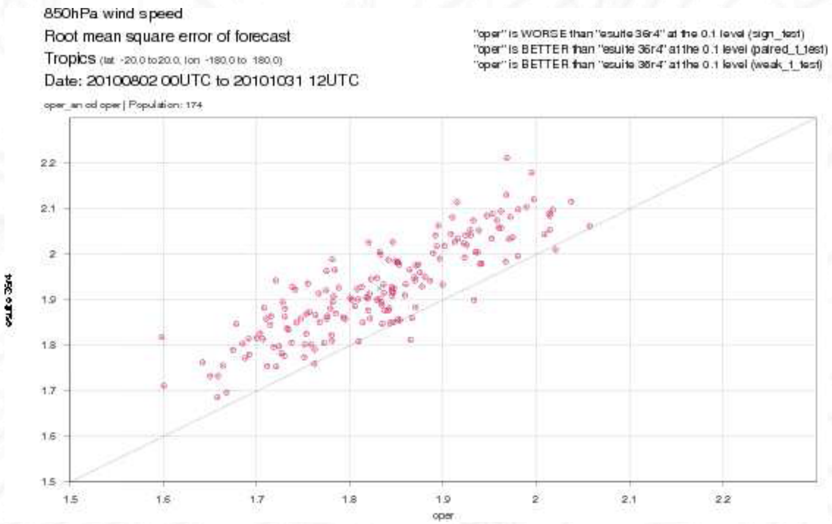
0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
Mar 01 2011 Mar 04 2011 Mar 07 2011 Mar 10 2011 Mar 13 2011 Mar 16 2011 Mar 19 2011 Mar 22 2011 Mar 25 2011 Mar 28 2011 Mar 31 2011

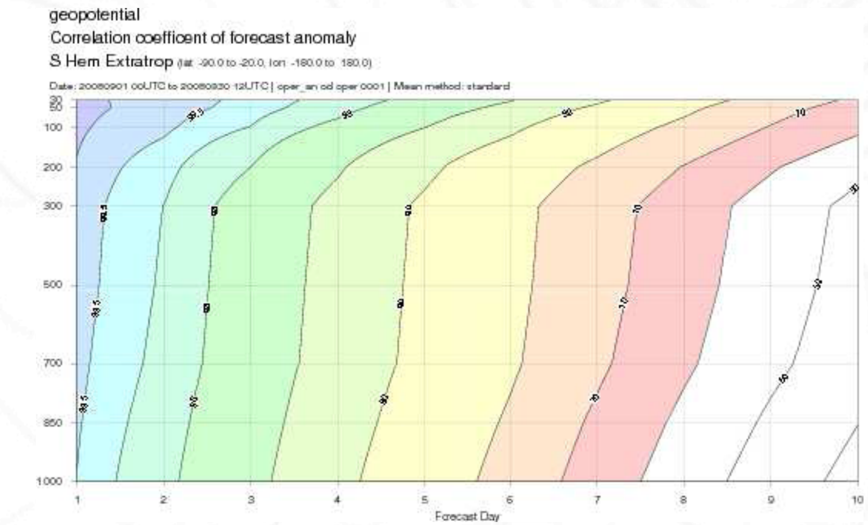


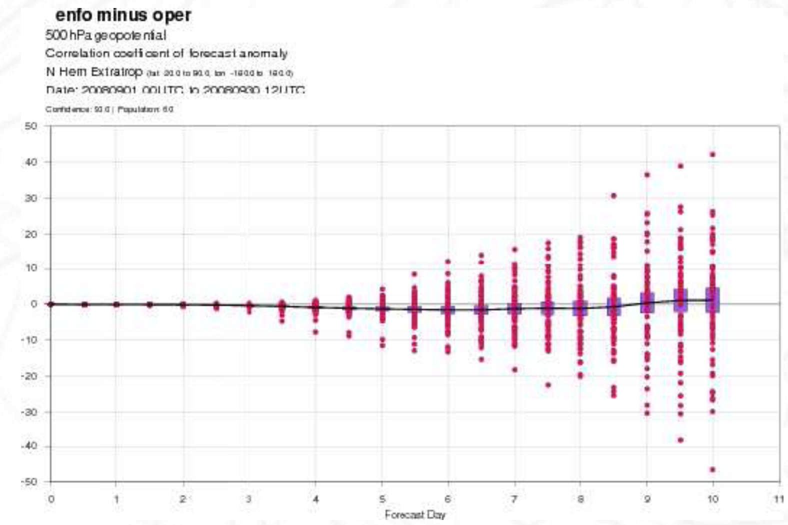
synopticView

Gallery









synopticView

Conclusion

- The directive helps the user in specifying values, a ‘help’ key can be added in the definition file. The syntax is quite simple and no programming is required. However, for some keywords, callables can be specified.

- The directive helps the programmer in the sense that it guarantees that the values of a dictionary are formatted according to specifications (lower case, lists etc…)

Score computation