Actimar - Python Documentation

Release svn1645

Actimar

September 10, 2009

CONTENTS

Contenu

CHAPTER

ONE

LES TUTORIELS

Chacun des tutoriels existe sous la forme d'un script en python pouvant être excécuter tel quel, et le "listing" d'un script constitue le tutoriel correspondant, inclus tel quel dans le document. Certains des tutoriels aboutissent à la création de figure et/ou de fichiers divers. Les scripts se trouvent ici :

/home/raynaud/svn/doc/python/src

1.1 Outils de base

1.1.1 Outils variés

Voir le module misc.

Les chaines de caractères

```
# -*- coding: utf8 -*-
# Opérations
print 'aaa'+"bbb"
#   -> aaabbb
print '-'*50
#   -> --------------------------------------------------

# Substitutions
# - ordonné
print '%i' % 6
#   -> 6
print '%05i %f' % (5, 7)
#   -> 00005 7.000000
# - par mots clés
print '[%(toto)s] %(result)i !' % {'toto':'ok', 'result':20}
#   -> [ok] 20 !
# - par variables grace a vars()
adjectif = 'pratique'
print "C'est %(adjectif)s !"%vars()
#   -> C'est pratique !

# Divers
# - centrage
title = (' %s '%'centrage'.title()).center(40,'#')
print '#'*len(title)+'\n'+title+'\n'+'#'*len(title)
#  -> ########################################
#  -> ############### Centrage ###############
#  -> ########################################
# - tests
```

```
print 'haha'.startswith('h')
#  -> True
print '01'.isdigit()
#  -> True
# - split-join
print ' '.join('a-b-c d-e'.split('-'))
#  -> a b c d e
```

1.1.2 Les temps

La gestion des données temporelles.

Les bases

Les temps cdtime

Voir les time tips.

```
import cdtime

# Creer un objet 'comptime' : temps absolu
# - en specifiant tout (annee,mois,jour,heure,minute,seconde)
ctime = cdtime.comptime(2000,1,1,0,0,0)
# - ou seulement les premiers arguments
ctime = cdtime.comptime(2000)
# - a partir d'une chaines de caracteres
ctime = cdtime.s2c('2000-1-1 0')
# - on verifie
print ctime
#  -> 2000-1-1 0:0:0.0
# - on verifie des valeurs numeriques
print ctime.year,ctime.day
#  -> 2000 1

# Creer un objet 'reltime' : temps relatif
# - en specifiant explicitement (valeur, unites CF)
rtime = cdtime.reltime(50, 'years since 1950')
print '%s | %s | %s' %(rtime,rtime.value,rtime.units)
#  -> 50.000000 years since 1950 | 50.0 | years since 1950
# - a partir d'une chaines de caracteres et d'unites
rtime = cdtime.s2r('2000-1-1 0','years since 1950')
print rtime.value
#  -> 50.0

# Operations
# - soustraction/addition
print ctime.add(1,cdtime.Year),'|',rtime.add(-1,cdtime.Year)
#  -> 2001-1-1 0:0:0.0 | 49.00 years since 1950
# - conversions
rtime2 = ctime.torel('days since 2000')
ctime2 = rtime.tocomp().add(1,cdtime.Year)
# - comparaison
print rtime2 == rtime
#  -> True
print ctime2 <= ctime
#  -> False

# Verification des types
```

```
from actimar.misc.atime import is_comptime,is_reltime
print is_comptime(ctime),is_reltime(rtime)
```

Les différents types

```
import time,datetime,cdtime
from actimar.misc.atime import is_datetime,is_cdtime

# Le temps de base : time
# - heure locale
mytime =  time.localtime()
print mytime
#  -> (2007, 11, 12, 17, 5, 20, 0, 316, 0)
year = mytime[0]
# - chaine de caracteres
print time.asctime()
#  -> Mon Nov 12 17:08:13 2007

# Le temps manipulable (creation, transformation) : datetime
# !! ne fonctionne pas pour les dates avant 1900 !!
mytime = datetime.datetime(2000,10,1,2)
print mytime.day,mytime.second
#  -> 1 0
# - incrementer
mytime2 = mytime.add(datetime.timedelta(1,1)) # (day,second)
print mytime.day,mytime.second
#  -> 2 1

# Temps cdtime
#  ! Module de temps officiel d'actimar !
#  Voir le tutoriel (*@\ref{lst:misc.time.bases.cdtime}@*) pour plus d'infos
ctime = cdtime.comptime(2000,10)
print mytime.year,mytime.month
#  -> 2000 10

# Verification des types
print is_datetime(mytime),is_cdtime(mytime)
```

Outils Actimar

Le formatage en ascii

```
from actimar.misc.atime import strftime,strptime

# Lecture a partir d'une chaine de caracteres et d'un format
# (tappez strptime dans google)
mytime = strptime('1950-01-01 07:00:00','%Y-%m-%d %H:%M:%S')
# Verification partielle
print mytime.year,mytime.minute
# -> 1950 0

# On choisit la langue francaise
import locale
locale.setlocale(locale.LC_ALL,'fr_FR.utf8')

# Ecriture sous un autre format
```

```
print strftime('%e %B %Y a %Hh%M',mytime)
# ->  1 janvier 1950 a 07h00
```

Les unités de temps

```
# On definit des unites
units = 'hours since 2000-01-15 06:00'

# Le format est-il bon ?
from actimar.misc.atime import *
print are_good_units(units)
#  -> True

# Memes unites ?
print are_same_units('Hours since 2000-1-15 06', units)
#  -> True

# Changer les unites d'un axe de temps
from actimar.misc.axes import create_time
import numpy as N
taxis = create_time(N.arange(6.)*48, units)
# - avant changement
print taxis.units, taxis[0:2]
print taxis.asComponentTime()[0:2]
#  -> hours since 2000-01-15 06:00 [  0.  48.]
#  -> [2000-1-15 6:0:0.0, 2000-1-17 6:0:0.0]
# - changement
ch_units(taxis, 'days since 2000-1-15 06', copy=0)
# - apres changement
print taxis.units, taxis[0:2]
print taxis.asComponentTime()[0:2]
#  -> days since 2000-1-15 06 [ 0.  2.]
#  -> [2000-1-15 6:0:0.0, 2000-1-17 6:0:0.0]

# Le temps matplotlib
taxis_mpl = mpl(taxis)
print taxis_mpl[0], taxis_mpl.units
#  -> 730134.25 days since 0001

# Changer les unites de temps d'une variable
import MV2
var = MV2.array(MV2.arange(len(taxis)), dtype='f', axes=[taxis])
ch_units(var, 'hours since 2000-01-15 06')
print var.getTime()[0:2]
#  -> [  0.  48.]


# Changements de fuseau horaire
# - maintenant a l'heure UTC
t_utc = now(True)
print strftime('%H:%M', t_utc), t_utc.hour
#  -> 15:54 15
# - heure de paris
t_paris = utc_to_paris(t_utc)
print strftime('%H:%M', t_paris), t_paris.hour
#  -> 17:54 17
# - retour en UTC
print tz_to_tz(t_paris,'Europe/Paris','UTC').hour
#  -> 15
# - travail sur une chaine de caracteres !
```

```
print to_utc('2000-10-01 10:20', 'Europe/Paris')
#  -> '2000-10-1 8:20:0.0'
```

1.1.3 Les variables

Elles sont au coeur de toute l'arborescence des outils pythons à Actimar. Vous pouvez vous référer aux exemples sur site de CDAT, comme par exemple les base de cdms.

Les notions de bases

Les variables numériques

```
import numpy as N, MV2 as MV
MA = N.ma

# Creation d'un tableau numerique pur
# - plein de zeros (pareil avec ones)
z = N.zeros((3,4))
print z.shape
#  -> (3, 4)
# - avec des valeurs choisies
a = N.array([1.,2,3]) # un des element est reel, donc le tableau aussi
print a
#  -> [ 1., 2., 3.,]

# On cree un tableau maske
# - pas de masquage
a2 = MA.array(a)
# - on maske exactement les 2.
b = MA.masked_object(a,2.)
# - pareil mais avec une marge possible (voir help(masked_values))
print MA.masked_values(a,2.)
# > [1.0 ,-- ,3.0 ,]
# - autre selction
print MA.masked_outside(a,0.,2.)
#  -> [1.0 ,2.0 ,-- ,]

# Masks : 1 = valeur masquee !
# - standard
print b.mask()
#  -> [0,1,0,]
# - quand une variable n'a pas de valeur masquee
print a2.mask()
#  -> None
# - mais si on veut quand meme un tabeau
print MA.getmaskarray(a2)
#  -> [0,0,0,]

# Copy or not copy ?
#  Si on a copy=0, on fait seulement un "lien" des
#  valeurs numeriques =>
#   - ON UTILISE MOINS DE MEMOIRE
#   - attention aux suprises si le tableau d'origine est modifie
d = MA.masked_object(a,2.,copy=0)
a[0] = 10
print d
#  -> [10.0 ,-- ,3.0 ,]
```

```
# Une variable avec des axes
s# - creation avec lien
m = MV.array(b,copy=0,id='yoman')
# - modification des valeurs SANS CREER UN NOUVEL OBJECT !
m[:] = MA.masked_object(m,10.,copy=0)
print repr(m)
# > yoman
# > array(data =
# >  [ 2., 2., 3.,],
# >       mask =
# >  [1,1,0,],
# >      fill_value=[ 2.,])
print m
#  -> [-- ,-- ,3.0 ,]
m.info()
#  -> bla bla sur les attributes de la variable et ses axes
```

Axes et variables

```
import numpy as N, cdms2 as cdms

# Creation d'un axe de longitude
# - base
lon = cdms.createAxis([-5.,-4.,-3.],id='lon')
lon.long_name = 'Longitude'
lon.units = 'degree_east'
# - ajout de lon.axis='X' et lon.modulo = '360.'
lon.designateLongitude()

# Latitude
lat = cdms.createAxis([46.,47.,48.],id='lat')
lat.long_name = 'Latitude'
lat.units = 'degree_north'
lat.designateLatitude() # lat.axis = 'Y'

# Pronfondeur
depth = cdms.createAxis([-200.,-100.,-0.],id='depth')
depth.long_name = 'Depth'
depth.units = 'm'
depth.designateLevel() # depth.axis = 'Z'

# Temps
# - creation
time = cdms.createAxis([0.,1.,2.],id='time')
time.long_name = 'Time'
time.units = 'days since 2006-08-01'
time.designateTime(calendar=cdtime.DefaultCalendar) # time.axis = 'T'
# - verif
ctime = time.asComponentTime()
print ctime,ctime[1].day
#  -> [2006-8-1 0:0:0.0, 2006-8-2 0:0:0.0, 2006-8-3 0:0:0.0] 2
rtime = time.asRelativeTime()
print rtime,rtime[1].value
#  -> [0.00 days since 2006-08-01, 1.00 days since 2006-08-01,
#      2.00 days since 2006-08-01] 1.0


# Maintenant, on cree une variable avec ces axes
#- methode direct
temp1 = cdms.createVariable(N.ones((3,3,3,3)),typecode='f',id='temp',
```

```
        fill_value=1.e20,axes=[time,depth,lat,lon],copyaxes=0,
        attributes=dict(long_name='Temperature',units='degC'))
# - remarque
print cdms.createVariable is MV.array
#  -> True         (ce sont les meme fonctions !)
# - une methode indirecte
#   . initialisation
temp2 = MV.array(N.ones((3,3,3,3))).astype('f')
#   . attributs
temp2.id = 'temp'
temp2.long_name = 'Temperature'
temp2.units = 'degC'
temp2.set_fill_value(1.e20) # <=> temp2.setMissing(1.e20)
#   . axes
temp2.setAxisList([time,depth,lat,lon])
#   . ou par exemple individuellement
temp2.setAxis(1,depth)
```

Les outils actimar

Manipulation des axes

```
# Creation d'un axe de temps
from actimar.misc.axes import *
import numpy as N,cdms2 as cdms
time_axis = create_time(N.arange(10.),
        'days since 2006-10-01',long_name='Mon axe de temps')

# Creation des axes geographiques
# - longitude: on change l'id de 'lon' a 'longitude'
lon_axis = create_lon(N.arange(5)-5.,id='longitude')
# - latitude
lat_axis = create_lat(N.arange(10)*.5+44.)
# - profondeur
dep_axis = create_dep(N.arange(0,-400.,50),units='m')

# Un axe quelconque
bad_axis = cdms.createAxis([1],id='pipo')

# Verification des types d'axes avec 'axis_type'
# - affichage pour tous
print ' | '.join(['%s:%s'%(axis.id,axis_type(axis))
        for axis in time_axis,lon_axis,lat_axis,dep_axis,bad_axis])
#  -> time:t | other_time:t | longitude:x | lat:y | depth:z | pipo:-
# - verification ponctuele
print islon(lon_axis),islon(lat_axis),is_geo_axis(lat_axis)
#  -> True False False

# Reformattage des axes pour deviner identifier
#  ceux geographiques (lon,lat,dep,time)
# Le reformattage peut changer : id, units, long_name.
# - un axe pourri mais avec 'long_name' explitcite
time_axis2 = cdms.createAxis([6],id='other_time')
time_axis2.long_name = 'Time'
check_axis(time_axis2)
print istime(time_axis2) # en fait, 'check_axis' appelle 'istime'
#  -> True
# - une variable tiree d'un fichier (voir le tutoriel (*@\ref{lst:misc.io.netcdf}@*))
f = cdms.open('/home2/amzer/raynaud/misc/samples/mars3d.nc')
var = f('u',time=slice(0,1),lat=slice(0,10),lon=slice(0,10),squeeze=1)
```

```
f.close()
check_axes(var)
lat_axis = var.getAxis(0)
print lat_axis.axis, islat(lat_axis)
#  -> Y True
```

1.1.4 Lecture et écriture de fichiers

Voir le module `io`.

Les fichiers netcdf

Voir le tutoriel sur le site de CDAT.

```
# Ouverture
import cdms2 as cdms
f = cdms.open('/home2/amzer/raynaud/misc/samples/mars3d.nc')

# Attribut global
print f.title
#  -> MARS3D FORECAST

# Lister les variables
print f.variables.keys()
#  -> ['temp', 'uz', 'h0', 'xe', 'u', 'v', 'vz']

# Avoir des informations sur une variables sans la lire, via []
nt = f['temp'].shape[0]
print f['temp'].getTime().asComponentTime()[0:nt:nt-1]
#  -> [2008-1-7 0:0:0.0, 2008-1-9 23:0:0.0]

# Lire une selection de la variable
import cdtime
temp = f('temp', ('2008-1-7',cdtime.comptime(2008,1,7,12), 'cc'),
        z=(0., .5), lon=slice(5,6), lat=(47.49,47.53), squeeze=1)
print temp.shape
# -> (13, 6, 6)
# squeeze a supprime l'axes des longitudes de dim 1

# Fermer le fichier lu
f.close()

# Creer un nouveau fichier
f = cdms.open('misc.io.netcdf.nc', 'w') # ouverture en ecriture
f.write(temp) # ecriture d'une variable
f.history = 'Created with '+__file__.encode('utf8') # attribut global
f.close() # fermeture
```

Les fichiers ascii

Il y a des exemples plus complets ici, avec notamment des fichiers binaires. Voir notamment les fonctions : `numpy.loadtxt()`, `numpy.savetxt()`, `open()`.

```
# On recupere une variable cdms
import cdms2 as cdms
f = cdms.open('/home2/amzer/raynaud/misc/samples/mars3d.nc')
kwselect = dict(lon=slice(10, 11), lat=slice(10, 11), squeeze=1)
```

```
temp = f('temp', z=slice(-2, -1), **kwselect) # une serie
u =  f('u', **kwselect) # une autre serie
v =  f('v', **kwselect) # une autre serie
f.close()

# On recupre le temps
time =  temp.getTime() # axe
ctime = time.asComponentTime() # temps cdtime.comptime()

# Creation a la main au format : YYYY/MM/DDZHH:MM TEMP U V
from actimar.misc.atime import strftime, ch_units, strptime
f = open('misc.io.ascii.1.dat', 'w')
f.write('# Ligne de commentaire\n')
for it in xrange(len(temp)):
        t = strftime('%Y/%m/%dZ%H:%M',  ctime[it])
        f.write('%s %.4f %f %f\n' % (t, temp[it], u[it], v[it]))
f.close()

# Verification rapide (deux premieres lignes)
f = open('misc.io.ascii.1.dat')
print ''.join(f.readlines()[:3])
f.close()
#  -> # Ligne de commentaire
#  -> 2008/01/07Z00:00 13.2232 0.157425 0.395160
#  -> 2008/01/07Z01:00 13.2231 0.317030 0.386611

# Ecriture rapide via numpy
# - creation
import numpy as N
time_units = 'hours since %s'%ctime[0]
newtime = ch_units(time, time_units)[:]
data = N.array([newtime,temp.filled(999.),
        u.filled(999.), v.filled(999.)],copy=0)
f = open('misc.io.ascii.2.dat', 'w')
f.write('# Ligne de commentaire\n')
N.savetxt(f, data.transpose(), fmt='%.3f', delimiter='\t')
#   note : on peut donner f ou le nom du fichier a N.savetxt()
# - verification
f = open('misc.io.ascii.2.dat')
print ''.join(f.readlines()[:2])
f.close()
#  -> 0.000 13.223 0.157 0.395
#  -> 1.000 13.223 0.317 0.387

# Lecture avancee
# - convertisseur pour le temps
def convtime(s):
        return strptime(s, '%Y/%m/%dZ%H:%M').torel(time_units).value
# - chargement partiel
tt, uu, vv = N.loadtxt('misc.io.ascii.1.dat',  comments='#',
        usecols=[0, 2, 3], converters={0:convtime}, unpack=True)
# - verification
print tt[0], uu[0], vv[0]
#  -> 0.0 0.15742500000000001 0.39516000000000001
```

Ce tutoriel crée les fichiers ascii `misc.io.ascii.1.dat` et `misc.io.ascii.2.dat`.

Les fichiers xyz

Voir : XYZ

```
# Lecture d'un fichier xyz et plot
from actimar.misc.io import XYZ
xyz = XYZ('/home2/amzer/raynaud/misc/samples/manche.xyz')

# Verifs
print len(xyz)
#  -> 23538
print xyz[0:3]
#  -> [(-5.93,52.97,25.0),(-5.90,52.97,41.0),(-5.85,52.97,33.0)]

# Exclusion
xyz.exclude([[-6., 52.], [-5.5, 52], [-5.5, 52.5], [-6., 52.5]])

# Modification
print xyz.z().max()
#  -> 141.0
xyz *= 1.1
print xyz.z().max()
#  -> 155.1

# Plot
xyz.plot(title='Mnt Manche', units='m', subplot=211, show=False,
        figsize=(5.5, 5), left=.12, top=.96, right=1, bottom=.06)

# Zoom
xyz_zoom = xyz.clip((-6, 51.5, -4, 52.5), long_name='Zoom mnt Manche')
xyz_zoom.plot(subplot=212, savefigs='misc-io-xyz', size=10)

# Sauvegarde
xyz_zoom.save('zoom.xyz')

# Autres exemples d'initialisations
xyz2 = XYZ(xyz.xyz())
xyz3 = XYZ((xyz.x(), xyz.y(), xyz.z()), units='m')
xyz4 = XYZ(xyz, long_name='my XYZ')
```

Les formats fortran

On passe par Scientific.IO.FortranFormat.

```
from Scientific.IO.FortranFormat import FortranFormat,FortranLine

# Fichier ascii exemple
f = open('misc.io.fortran.dat', 'w')
f.write('   59999\n   68888\n')
f.close()

# Declaration du format
format = FortranFormat('2I4')

# Lecture
import numpy as N
f  = open('misc.io.fortran.dat')
data = N.zeros((2, 2), 'i')
for i, line in enumerate(f):
        data[i] = FortranLine(line[:-1], format)[:]
print data
#  -> [[   5 9999]
#  ->  [   6 8888]]
```

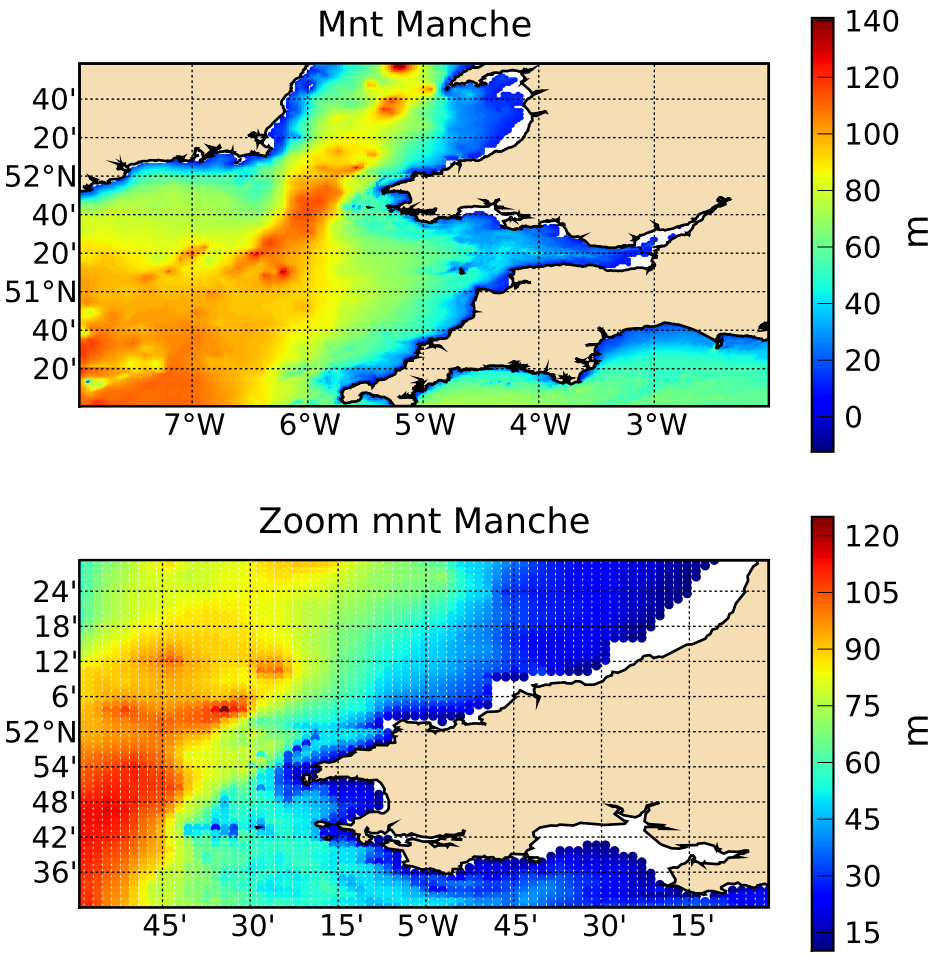


Figure 1.1: Un fichier ascii à trois colonnes (format .xyz) et un zoom sont tracés, puis réinitialisés part trois méthodes.

Ce tutoriel crée le fichier ascii `misc.io.fortran.dat`.

Les fichiers Excel

```
# -*- coding: utf8 -*-
# On initialise le document Excel
from pyExcelerator import *
w = Workbook()

# Définition des styles
from actimar.misc import xls_style
from copy import deepcopy as cp
style_header = xls_style(b=1, ha='center', bottom=2, top=2, c=2)
style_left = xls_style(ha='left', i=1)
style_num = xls_style(fmt='0.00')
style_bottom = xls_style( top=2)

# Première feuille
# - nom
wc = w.add_sheet('Courants')
# - données
data1 = {
        'Four':[1.15646541, 0.25641],
        'Pointe du Raz':[2.56421, 0.1556122],
}
# - taille
wc.col(0).width = int(wc.col(0).width * 1.5)
wc.col(1).width = int(wc.col(0).width * 0.7)
# - entêtes
for icol, title in enumerate(('','STD [m/s]', 'RMS [%]')):
        wc.write(0, icol, title, style_header)
# - colonne de gauche et données
for irow, title in enumerate(data1.keys()):
        # titre
        wc.write(irow+1, 0, title, style_left)
        # données
        for icol, value in enumerate(data1[title]):
                # RMS en %
                if icol == 1:
                        col_style = xls_style(cp(style_num), fmt='0.00%')
                else:
                        col_style = style_num
                # écriture
                wc.write(irow+1, icol+1, value, col_style)
# - bottom
for icol in xrange(3): wc.write(irow+2, icol, '', style_bottom)

# Deuxième feuille
# - nom
wn = w.add_sheet('Testouille')
# - gros titre
wn.write_merge(0, 1, 0, 5, u"Ça c'est un gros titre", xls_style(o=1, s=500))
# - hyperlien
uri = "http://relay.actimar.fr/~raynaud/pydoc"
urn = "Python Actimar"
wn.write_merge(3,3,1,10,Formula('HYPERLINK("%s";"%s")'%(uri, urn)),xls_style(c=4))
# - groupes de niveaux
wn.write(4, 1, 'level1')
wn.write(5, 1, 'level2')
wn.write(6, 1, 'level2')
wn.write(7, 1, 'level1')
wn.row(4).level = wn.row(7).level = 1
```

```
wn.row(5).level = wn.row(6).level = 2
# - dates
from datetime import datetime
wn.write(8, 0, 'Dates :')
wn.write(8, 1, datetime.now(), xls_style(fmt='h:mm:ss AM/PM'))
wn.write(8, 2, datetime.now(), xls_style(fmt='MMM-YY'))


# Écriture du document
w.save('misc.io.xls.xls')
print 'ok'
```

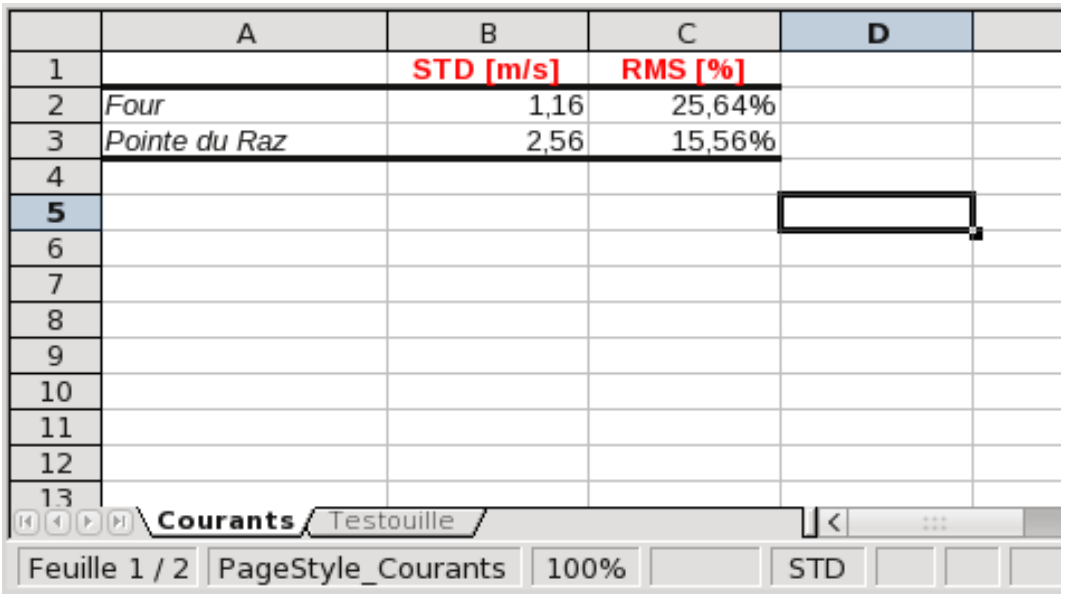


Figure 1.2: Première feuille Excel.

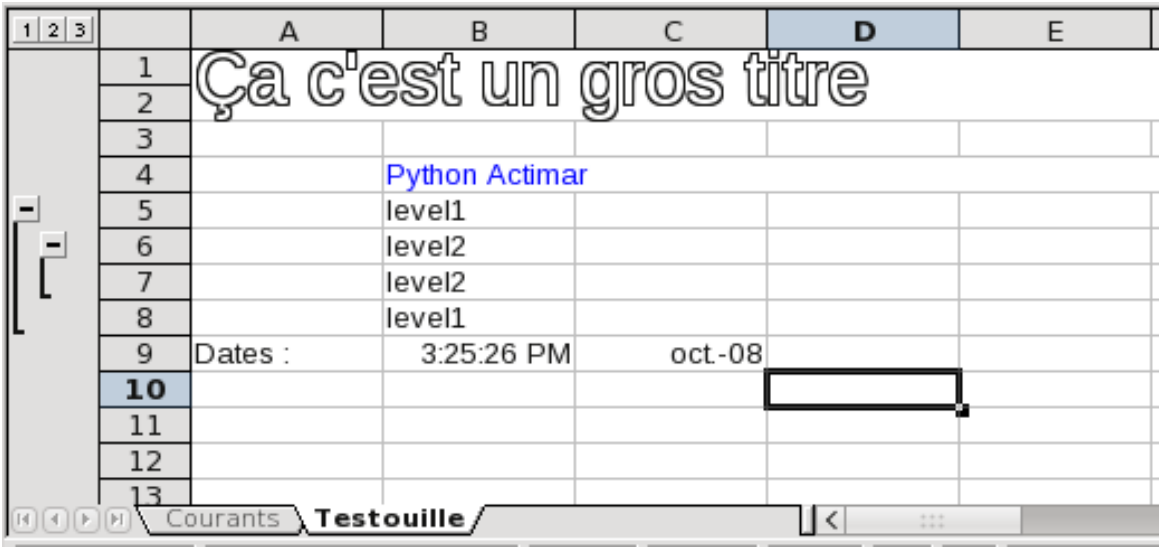


Figure 1.3: Deuxième feuille Excel.

Les fichiers de config .cfg ou .ini

On utilise le module `ConfigParser`.

Fichier de configuration initial :

```
# -*- coding: utf8 -*-
[DEFAULT]
# Valeurs accessible de toutes les sections
lat_min = 40.
lat_max = 50.
# Commentaire
# Autre forme de commentaire :
; lat_max = 51.
zone = Iroise

[sst]
# Nouvelle section
# - nom (utilisation de ':')
name: Sea surface temperature on %(zone)s
# - unites
units = C

[wind]
# - nom
name = Wind on %(zone)s
# Autre lat max
lat_max = 55.
# - unites
units = m/s
```

```
# -*- coding: utf8 -*-
# On charge le fichier
from ConfigParser import SafeConfigParser
config = SafeConfigParser()
config.read('misc.misc.config.in.ini')

# List des sections
print config.sections()

# On récupère les unités pour la SST
print config.get('sst', 'units')
#  -> m/s

# La latitude max de la section par défaut
print config.defaults()['lat_max']
#  -> 50.0
# Latitude max de la sst = celle par défaut
print config.getfloat('sst', 'lat_max')+1
#  -> 51.0

# Subsitutions
# - contenu substitué
print config.get('wind', 'name')
#  -> Wind on Iroise
# - contenu brut
print config.get('wind', 'name', raw=True)
#  -> Wind on %(zone)s
# - contenu substitué par la force
print config.get('wind', 'name',  vars=dict(zone='for Britanny'))
#  -> Wind on Britanny



# On vire une section
config.remove_section('wind')
```

```
# On en crée une autre
print config.has_section('sealevel')
#  -> False
config.add_section('sealevel')
config.set('sealevel','name', 'Sea level')
config.set('sealevel','units', 'm')
print config.has_option('sealevel', 'name')
#  -> True

# On sauvegarde
fc = open('misc.misc.config.out.ini', 'w')
config.write(fc)
fc.close()
```

Fichier de configuration final :

```
[DEFAULT]
lat_max = 50.
lat_min = 40.
zone = Iroise

[sst]
units = C
name = Sea surface temperature on %(zone)s

[sealevel]
units = m
name = Sea level
```

Les fichiers sinusX

Voir la fonction `snx()`.

```
# Init
from actimar.misc.io import write_snx
from _geoslib import Point, LineString, Polygon
import numpy as N
fbase = __file__[:-2]

# Points
fpoints = fbase+'points.snx'
# - tuples de points
write_snx([(0., 0., 0.), (1., 1., 1.)], fpoints)
# - marche aussi par groupe
write_snx([[(2., )*3, (3, )*3]], fpoints, mode='a') # append
# - depuis des tableaux
x = N.arange(4.)
y = N.arange(4.)
z = N.arange(4.)
bloc_xyz = N.array([x, y, z]).transpose() # bloc_xyz[i] = point i
write_snx(bloc_xyz, fpoints, mode='a')
# - deux blocs
bloc_xyz2 = bloc_xyz+10.
write_snx([bloc_xyz, bloc_xyz2], fpoints, type='points', mode='a')
#   note : il faut preciser le type dans ce cas
# - depuis des "Point" de _geoslib
write_snx([Point((0., 0.)), Point((100., 100.))], fpoints, mode='a', z = 99)

# Lignes
flines= fbase+'lines.snx'
```

```
# - blocs
write_snx(bloc_xyz, flines, type='lines') # on doit preciser le type
write_snx([bloc_xyz, bloc_xyz2], flines, mode='a')
# - "LineString" de _geoslib
write_snx(LineString(bloc_xyz[:, :2]+100.), flines, mode='a', z=999)
write_snx([LineString(bloc_xyz[:, :2]+100.), LineString(bloc_xyz2[:, 2]+100.)], flines, mode='a')

# Polygones : comme les lignes, mais fermes
fpolys= fbase+'polys.snx'
# - blocs
write_snx(bloc_xyz, fpolys, type='polygons') # on doit preciser le type
write_snx([bloc_xyz, bloc_xyz2], fpolys, mode='a', type='polygon')
# - on referme presque la ligne => detection AUTO comme polygone
bloc_xyz3 = N.concatenate((bloc_xyz, bloc_xyz[:1]+1.))
write_snx([bloc_xyz3], fpolys, mode='a')
# - "Polygon" de _geoslib
write_snx(Polygon(bloc_xyz3[:, :2]+100.), flines, mode='a')

# Via un descripteur de fichier
f = open(fbase+'mix.snx', 'w')
write_snx([bloc_xyz, bloc_xyz2], f, type='point', close=False)
write_snx([bloc_xyz, bloc_xyz2], f, type='line', close=False)
write_snx([bloc_xyz, bloc_xyz2], f, type='polygon', close=False)
f.close()

# Ecriture auto dans des fichiers separes
write_snx([bloc_xyz, bloc_xyz2], fbase+'split%i.snx', type='polygon', close=False)


print 'Done'
```

1.1.5 Les graphiques

Matplotlib

Pour les graphiques de base, le meilleur moyen de trouver ce que l'on cherche est d'aller voir la galerie de Matplotlib.

Cas simples

Tracé de cartes

Voir : `map()`.

```
# Lecture et masquage de la SST
import cdms2
f = cdms2.open('/home2/amzer/raynaud/misc/samples/mars3d.nc')
sst = f('temp', time=slice(0, 1),  z=slice(-1, None),
        lat=(47.8, 48.6), lon=(-5.2, -4.25),  squeeze=1)
f.close()
sst[:] = cdms2.MV.masked_object(sst, 999., copy=0)

# Trace de la carte
from actimar.misc.plot import map
map(sst,  title='SST en iroise', savefigs='misc-plot-basic-map',
        right=1., top=.9, figsize=(5.5, 4.5))
```

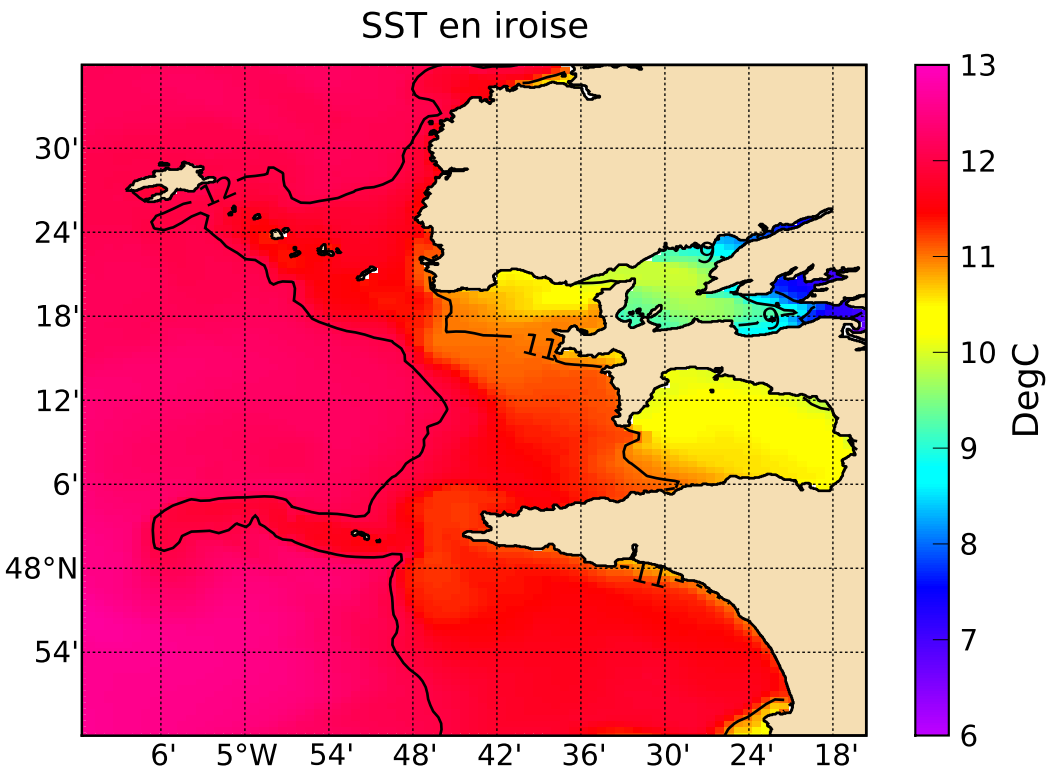


Figure 1.4: Une carte de base.

Tracé de vecteurs sur carte

Voir : `map()`.

```
# Lecture et masquage de la vitesse
import cdms2
zoom = dict(lat=(48.3, 48.55), lon=(-5.2, -4.8))
f = cdms2.open('/home2/amzer/raynaud/misc/samples/previcot.mars.r4.uv.nc')
u = f('u', **zoom)
v = f('v', **zoom)
f.close()
for var in u, v:
        var[:]*100.
        var[:] = cdms2.MV.masked_object(var, 0., copy=0)

# Trace des vecteurs
from actimar.misc.plot import map
map((u, v),  title='Vitesses en surface', show=False,
        quiver_alpha=.5, quiver_samp=10, quiver_width=0.009, quiverkey_value = 5., nofill=True,
        left=.08, right=1., top=.9, figsize=(6.6, 5.5), proj='merc', quiver_norm=2,
        savefigs=__file__, savefigs_pdf=True,)
```

Tracé de courbes

Voir : `curve()`.

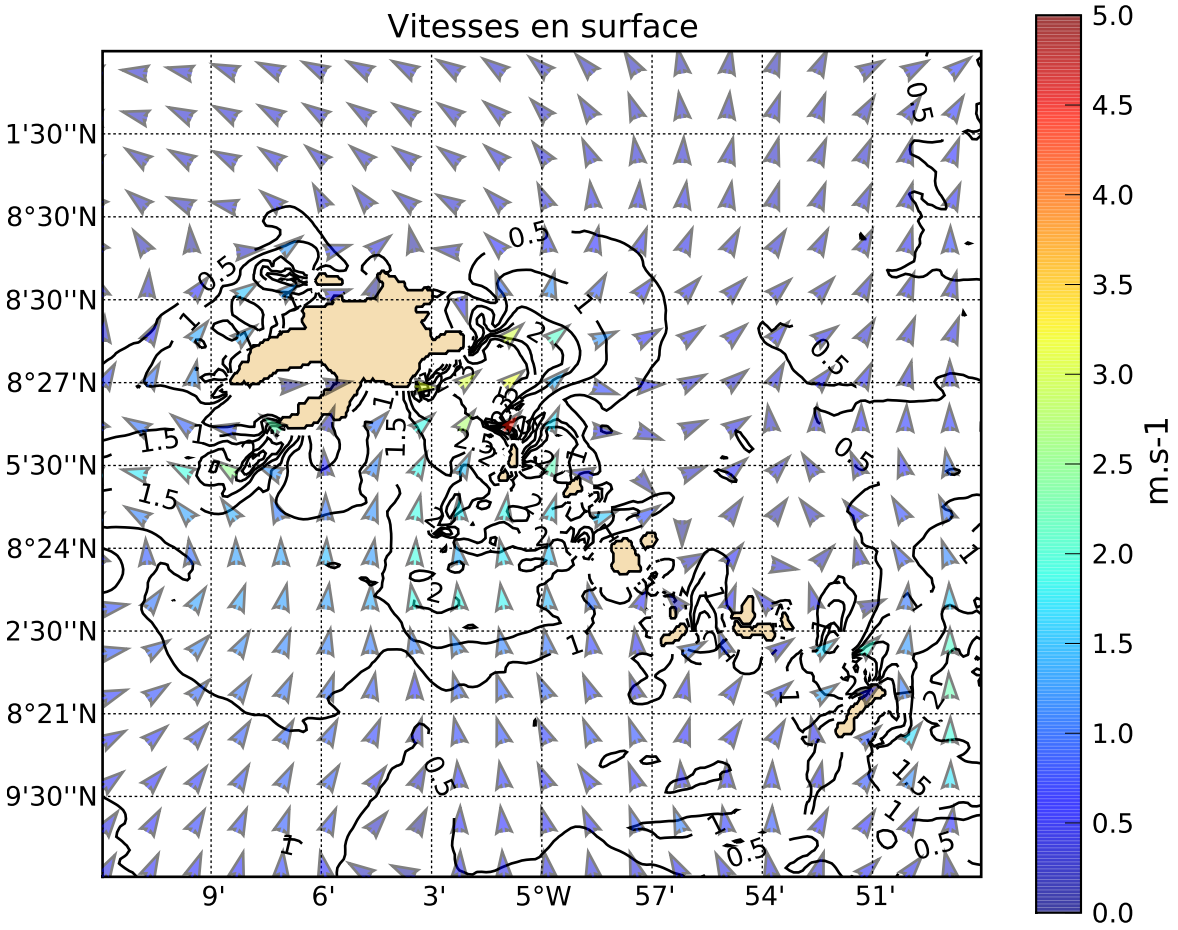


Figure 1.5: Un tracé de vecteurs vitesse sur une carte. Le module de est automatiquement tracé en fond mais peut être désactivé avec `nofill=True,contour=False`. Il est aussi possible de tracer un autre champ en fond avec par exemple `map((u,v,sst))`

```
# -*- coding: utf8 -*-
# Lecture et masquage de la SST
import cdms2
f = cdms2.open('/home2/amzer/raynaud/misc/samples/mars3d.nc')
sst = f('temp', z=slice(-1, None),
        lat=(48.1, 48.3), lon=(-5., -4.8),  squeeze=1)
f.close()
sst[:] = cdms2.MV.masked_object(sst, 999., copy=0)

# Trace de la moyenne spatiale
from actimar.misc.plot import curve
curve(sst, title='SST en moyenne spatiale',units = u'°C',
        along='t', subplot=211, show=False)

# Trace de la moyenne meridienne et temporelle
curve(sst, title=u'SST zonale', color='r',
        subplot=212,top=.9,hspace=.4,left=.15,bottom=.07,
        savefigs=__file__, savefigs_pdf=True)
```

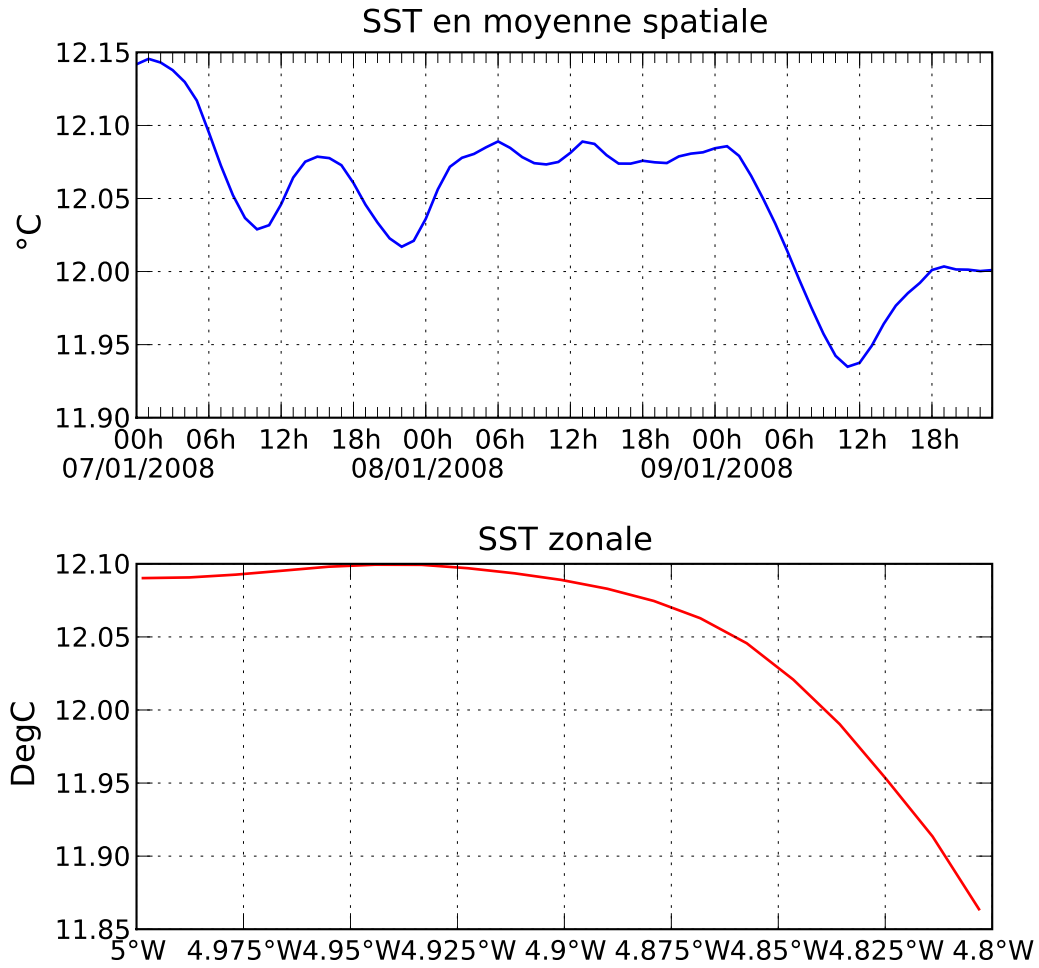


Figure 1.6: Quelques tracés basiques de courbes.

Stick plot

Voir : `stick()`.

```
# Lecture et masquage de la vitesse
import cdms2
f = cdms2.open('/home2/amzer/raynaud/misc/samples/mars3d.nc')
selector = dict(time=slice(0, 48),lat=slice(60, 61),
        lon=slice(60, 61), squeeze=1)
u = f('u', **selector)
v = f('v', **selector)
f.close()
for uv in u, v:
        uv[:] = cdms2.MV.masked_object(uv, 999., copy=0)

# Trace de la carte
from actimar.misc.plot import stick
stick(u, v,  title='Courants en Iroise', units='m/s',
        bottom=.2, top=.85, quiver_headwidth=2,
        quiverkey_value=.5, quiver_width=0.004, quiver_scale=5.,
        savefigs=__file__, figsize=(5.5,  3), savefigs_pdf=True)
```

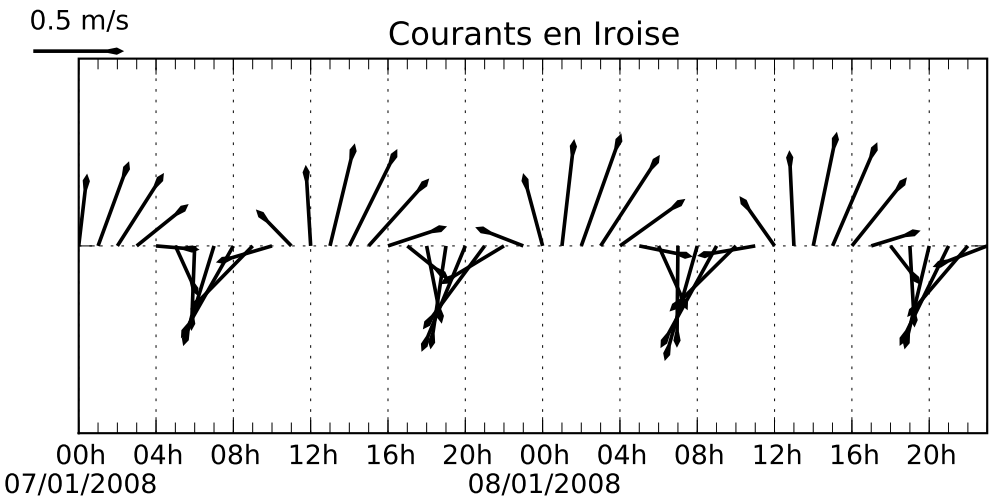


Figure 1.7: Graphique en “arrêtes de poisson”, dit *stick plot*.

Diagramme de Taylor

Pour comprendre ce qu’est un diagramme de Taylor, se référer à cette page.

Voir : `taylor()`.

```
# Construction des jeux de donnees
import MV2, numpy as N
nt = 50
# - reference
ref = MV2.sin(MV2.arange(nt, dtype='f'))*10.
# - modele 1
model1 = ref+N.random.rand(nt)*10.
model1.long_name = 'Model 1'
# - modele 2
```

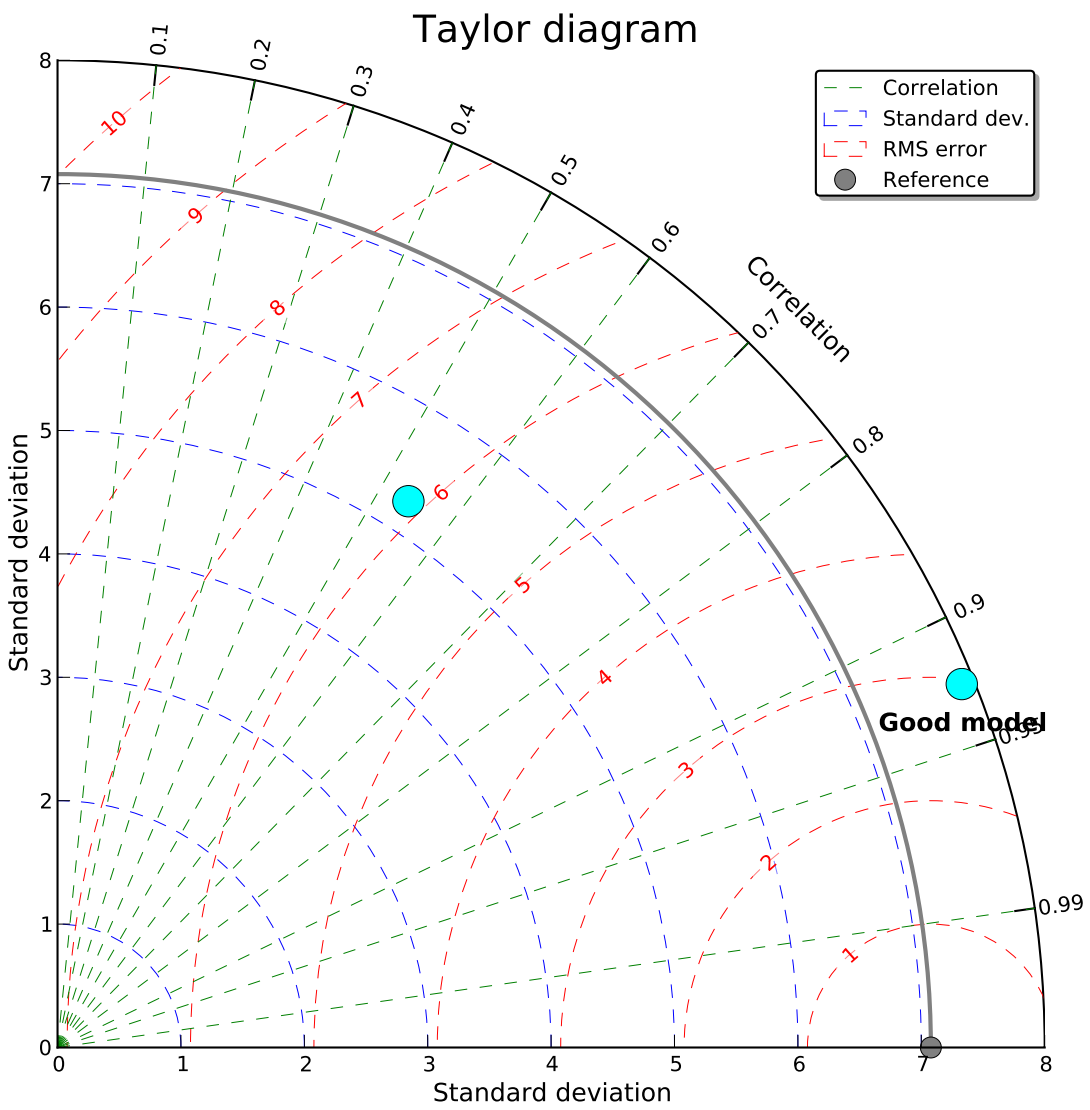


Figure 1.8: On compare ici deux jeux de données (modèle) à une référence (observations) à l'aide d'un diagramme de Taylor.

```
model2 = ref/2.+N.random.rand(nt)*15.
model2.long_name = 'Model 2'

# Plot
from actimar.misc.plot import taylor
taylor([model1, model2], ref, figsize=(8, 8), label_size='large', size=15,
        labels = ['Good model', None], colors='cyan', title_size=18,
        savefigs=__file__,  savefigs_pdf=True)
```

Cas complexes

Sorties météo

Voir : `curve()` `stick()` `bar()` `hldays()`.

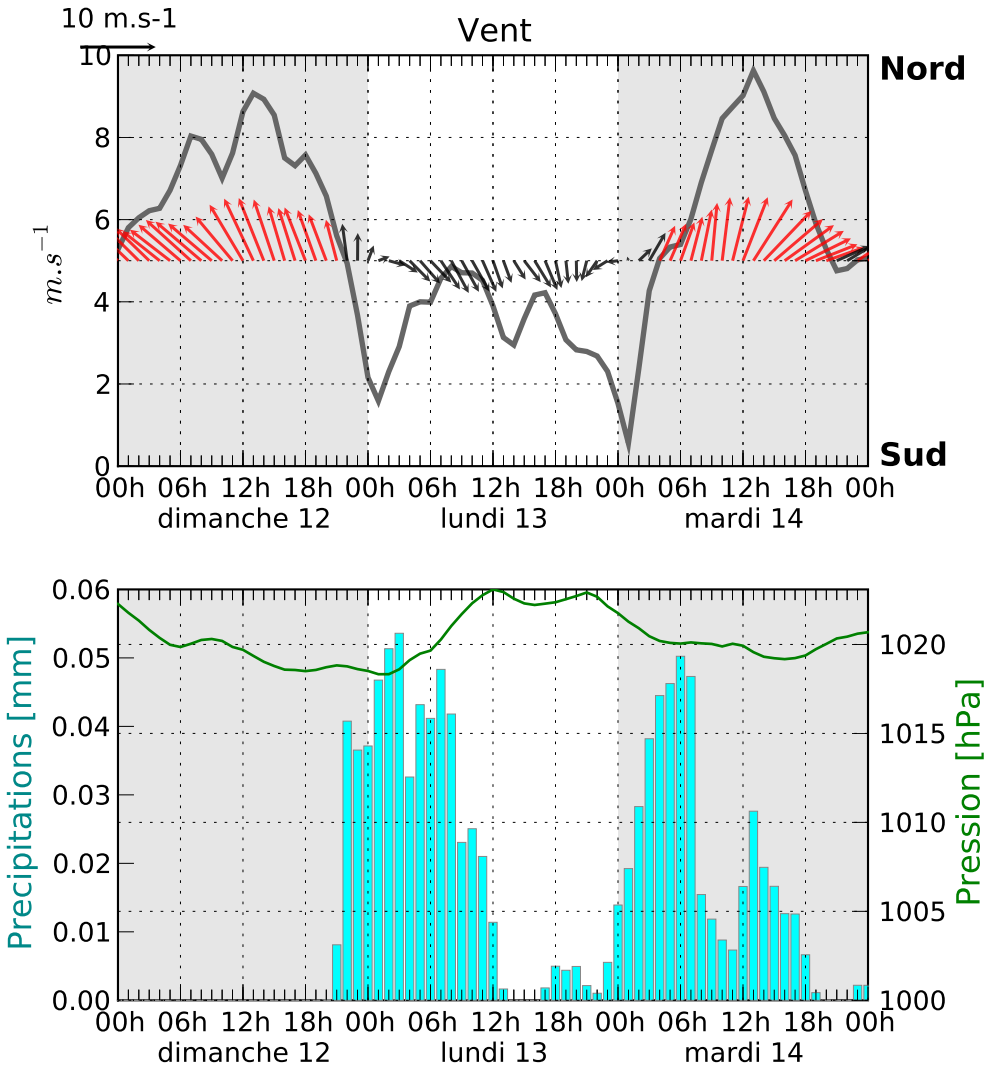


Figure 1.9: Quelques tracés 1D de quantités météorologiques.

```
# Lecture de la meteo
import cdms2, numpy as N, MV2
select = dict(time=('2008-10-12', '2008-10-15', 'cc'),
        lat=(48.1, 48.3), lon=(-4.9, -4.6))
f = cdms2.open('/home/raynaud/data/misc/samples/wrf.nc')
u3d = f('u10m', **select)
v3d = f('v10m', **select)
p3d = f('psfc', **select)
r3d = f('rain', **select)
f.close()


# Moyennes spatiale
from actimar.misc import cp_atts
for vn in 'u', 'v', 'p', 'r':
        # Moyenne
        orig = eval(vn+'3d')
        var = MV2.average(MV2.average(orig, axis=-1), axis=-1)
        # Attributs
        cp_atts(orig, var)
        exec vn+' = var'
nt = len(u)


# Plots
from actimar.misc.plot import curve, stick, bar, savefigs
import pylab as P
P.figure(figsize=(5.5, 6))
P.subplots_adjust(right=.85, hspace=.3, left=.14)
kwplot = dict(date_fmt=('day', {'special_fmt':'%A %e', 'phase':12}),
        show=False, nmax_tixks=-40)

# - module du vent
u.units = '$m.s^{-1}$'
m = MV2.sqrt(u**2+v**2)
m.units = u.units # mode latex pour les exposants
curve(m, dayhl=True, color='#666666',linewidth=2.,subplot=211,title=False,**kwplot)

# - vecteurs
P.twinx() # On travail sur le meme axe des X, mais pas des Y
colors = N.array(["#000000"]*nt) # noir a la base
colors[m.filled(0.)>5.] = '#ff0000'         # rouge si module > 4.
stick(u, v, color=colors.tolist(), xhide=True,
        quiver_headwidth=3,quiver_headlength=3,quiver_headaxislength=2,
        quiverkey_color='k', quiver_width=.004,alpha=.8, **kwplot)
fbox = P.gca().get_position(True) # positions pour ajouter Nord et Sud
P.figtext(fbox.xmax,fbox.ymin,' Sud',fontweight='heavy', size='12')
P.figtext(fbox.xmax,fbox.ymax,' Nord',va='top',fontweight='heavy', size='12')
P.title('Vent')

# - precipitations
dr = r.clone() # variation != accumulation
dr[:-1] = N.diff(r) ; dr[-1] = r[-1]-r[-2]
lr = bar(dr, width=.8, grid=False,  subplot=212,
        ylabel='Precipitations [%s]'%dr.units,
        title=False, dayhl=True, ylabel_color='#008888',zorder=100,
        color='#00ffff',linewidth=.2,edgecolor='#888888',**kwplot)

# - pression
P.twinx()
p[:] /= 100.
p.units = 'Pression [hPa]'
lp = curve(p, color='g', xhide=True,vminmax=1000, vmaxmin=1020,
        zorder=150, title=False, ylabel_color='g',**kwplot)

savefigs('test_bar', pdf=True)
```

Animation de courants

Voir : `map()`.

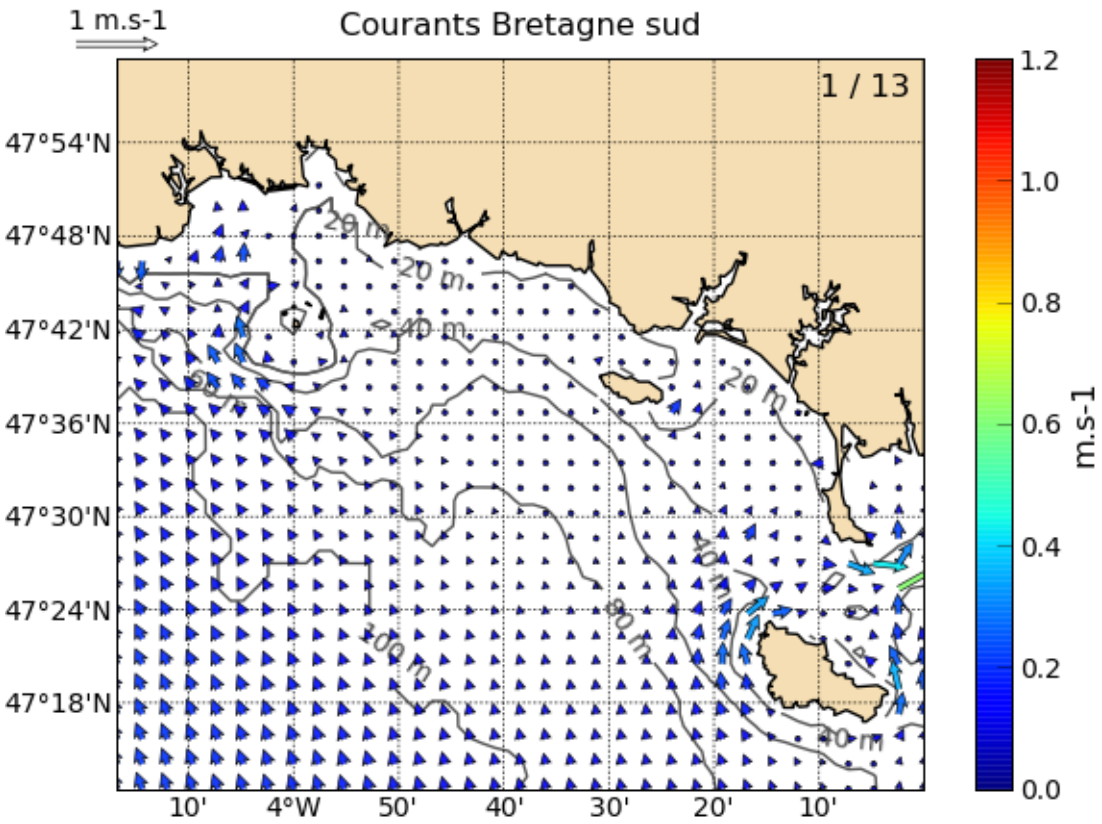


Figure 1.10: Animation de courants en gif animé.

```
# Lecture des donnnees
import cdms2, MV2
select=dict(lon=(-4.3, -3.), lat=(47.2, 48.), time=slice(9, 22))
f=cdms2.open('/home2/amzer/raynaud/misc/samples/mars2d.gascogne.nc')
u = f('u',**select)
v = f('v',**select)
h0=f('h0', **select)
f.close()
h0[:] = MV2.masked_values(h0, -999., copy=False)
for var in u, v:
        var[:] = MV2.masked_values(var, 0., copy=False)
mod = MV2.sqrt(u**2+v**2)

# Plots
from matplotlib import rc ; rc('font', size=10)
from actimar.misc.plot import map, savefigs, make_movie
from actimar.misc import auto_scale
import gc, pylab as P
m=None
levels = auto_scale(mod,nmax=10, vmin=0.)
nt = len(u)
for it in xrange(nt):
        print it
        # Bathymetrie
```

```
        m=map(h0, show=False, show=False, close=False, nofill=True,
                proj='merc', fmt='%i m', m=m, contour_colors='#555555', )
        # Courants
        map((u[it], v[it]), m=m, nofill=True,  quiverkey_value=1, quiver_scale=10,
                quiver_norm=3, contour=False, quiver_linewidth=0.5, quiver_alpha=.9,
                quiver_width=7.,  quiver_headwidth=2.5, quiver_headlength=2.5,
                quiver_headaxislength =2, show=False, levels=levels, right=1, quiver_samp=2,
                figsize=(6, 4.5), proj='merc', title='Courants Bretagne sud')
        # Indicateur de progression
        P.text(.98*m.xmax, .98*m.ymax, '%i / %i'%(it+1, nt), ha='right', va='top',
                zorder=200, size=12, family='courrier')
        # Savegardes
        P.savefig('quiver%02i.png'%it)
        if it==0: savefigs(__file__)
        P.close()
        gc.collect()

# Creation des animations
outbase=__file__[:-3].replace('.', '-')
# - gif anime
#make_movie('quiver*.png', outbase+'.gif')
# - video compatible windows
make_movie('quiver*.png', outbase+'.mpg', clean=True)
print 'Done'
```

1.1.6 La gestion des grilles

Voir le module `grid`.

Les polygones

Fait appel au module externe `_geoslib`.

```
# Importation des formes depuis la libraire de polygones
from _geoslib import Point, LineString, Polygon

# Polygone
import numpy as N
# - points sous la forme [[x1,y1],[x2,y2],...]
pp = N.array([[5., 5.], [25., 5.], [25., 25.], [15., 25.]])
# - transformation de numpy a Polygon
poly = Polygon(pp)
# - calcul de l'aire
print poly.area()
#  -> 300.0
# - plot grace a 'boundary'='get_coord()' (= pp)
import pylab as P
P.fill(poly.boundary[:, 0], poly.boundary[:, 1],
        facecolor=(.9, .9, .9))

# Points aleatoires et inclusion
# - coordonnees
xpts = N.random.random(50)*30
ypts = N.random.random(50)*30
# - transformation en Points
pts = []
for x, y in  zip(xpts, ypts):
        pts.append(Point((x, y)))
# - plot avec couleur differente si dans polygone
```

```
for pt in pts:
        # - utilisation de 'within()'
        color = ('r', 'g')[pt.within(poly)]
        # - plot grace a 'boundary'
        x, y = pt.boundary
        P.plot([x], [y], 'o'+color)

# Ligne de points et intersection
# - coordonnees
xy = N.array([[2., 7., 20., 26.], [2., 2., 28., 28.]])
# - LineString
line = LineString(xy.transpose())
# - plot
P.plot(xy[0], xy[1], '-r')
# - intersection (via 'intersects()') ?
print line.intersects(poly)
#  -> True
# - plot des intersections (via intersection)
for subline in line.intersection(poly): # = poly.intersection(line)
        xyl = subline.boundary
        P.plot(xyl[:, 0], xyl[:, 1], '-', color=(0, 1, 0))

# Autre polygone
# - creation et plot
xyp = N.array([[22., 3], [25., 7], [28, 3]])
triangle = Polygon(xyp)
P.fill(xyp[:, 0], xyp[:, 1], facecolor='b', alpha=.5)
# - intersections
if triangle.intersects(poly):
        for pol in triangle.intersection(poly):
                xyp = pol.boundary
                P.fill(xyp[:, 0], xyp[:, 1], facecolor='y', alpha=.5)
# Trace
from actimar.misc.plot import savefigs
P.axis([0, 30, 0, 30])
savefigs('misc-grid-polygons', pdf=True)
```

Les masques

Voir : masking

Masque d'après traît de côte

Voir : polygon_mask() map() add_key()

```
# Creation de la grille (15x15)
nx, ny = 15, 10
import numpy as N
from actimar.misc.axes import create_lon, create_lat
lon = create_lon(N.linspace(-5.2, -4.4, nx))
lat = create_lat(N.linspace(48, 48.6, ny))

# Parametres de plot
import pylab as P
import actimar.misc.color as C
P.figure(figsize=(5.5, 5))
P.subplots_adjust(top=.92,left=.03,bottom=.03,wspace=.05,hspace=.2)
kwplot = dict(colorbar=False,cmap=C.cmap_linear((.6, .8, 1), C.land), \
        contour=False,show=False,fillcontinents=False,
```

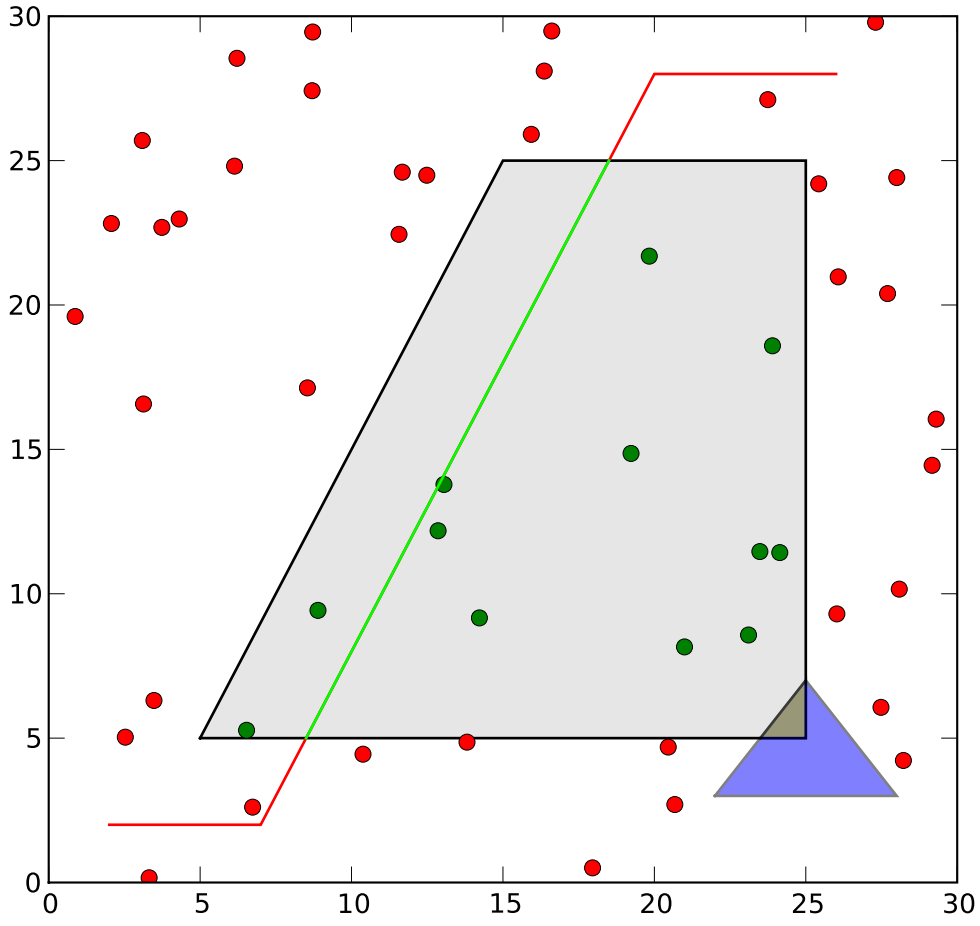


Figure 1.11: On se sert de la librarie de manipulation des formes (points, lignes, polygones) pour calculer des intersections, des aires, savoir quelle forme contient quel autre.

```
        drawmeridians=False, drawparallels=False)

# Creation du mask avec trait de cote fin ('f')
from actimar.misc.grid.masking import polygon_mask
from actimar.misc.plot import map
import MV2 as MV
# - mask si point central sur terre (mode = 0 = 'inside')
mask0 = polygon_mask((lon, lat), 'f', mode=0, ocean=False)
mask0 = MV.array(mask0, axes=[lat, lon]) # Forme cdms
mask0.long_name = "'inside'"
map(mask0, subplot=221, key=1, **kwplot)
# - mask si point central sur terre (mode = 1 = 'intersect')
seuils = .5
mask1 = polygon_mask((lon, lat), 'f', mode=1, thresholds=seuils, ocean=False)
mask1 = MV.array(mask1, axes=[lat, lon]) # Forme cdms
mask1.long_name = "'intersect' : seuil=%g" %seuils
map(mask1, subplot=222, key=2, **kwplot)
# - mask si point central sur terre (mode = 1 = 'intersect')
seuils = .3
mask1 = polygon_mask((lon, lat), 'f', mode=1, thresholds=seuils, ocean=False)
mask1 = MV.array(mask1, axes=[lat, lon]) # Forme cdms
mask1.long_name = "'intersect' : seuil=%g" %seuils
map(mask1, subplot=223, key=3, **kwplot)
# - mask si point central sur terre (mode = 1 = 'intersect')
seuils = (.3, .7)
mask1 = polygon_mask((lon, lat), 'f', mode=1, thresholds=seuils, ocean=False)
mask1 = MV.array(mask1, axes=[lat, lon]) # Forme cdms
mask1.long_name = "'intersect' : seuils=(%g/%g)" %seuils
map(mask1, subplot=224, key=4, **kwplot)

# Figure
from actimar.misc.plot import savefigs
savefigs('misc-grid-masking-coast')
```

Gestion des lacs dans un masque

Voir : GetLakes

```
# Inits
from actimar.misc.grid.masking import GetLakes
from actimar.misc.grid.misc import meshbounds
import actimar.misc.color as C
from actimar.misc.plot import savefigs
import numpy as N, pylab as P
P.figure(figsize=(5.5, 5))
P.subplots_adjust(bottom=.02, top=.92, left=.01, right=.98, hspace=.3)
cmap_mask = C.cmap_linear(((.6, .8, 1), C.land))
cmap_lakes = C.cmap_srs(('w', 'r', 'g', 'b'), stretch=0)

# Creer un mask avec des lacs
mask = N.ones((20, 30), '?') # Terre partout
mask[10:, 10:] = False  # Ocean
mask[10:15, 13:14] = True  # Ocean
mask[0, 0] = False      # Ptit lac
mask[5:15, 2:4] = False # Gros lac
xxb, yyb = meshbounds(N.arange(mask.shape[1]*1.), N.arange(mask.shape[0]*1.))
xlim = (xxb.min(), xxb.max()) ; ylim = (yyb.min(), yyb.max())
P.subplot(221)
P.pcolor(xxb, yyb, mask.astype('i'), cmap=cmap_mask)
P.xlim(xlim) ; P.ylim(ylim) ; P.xticks([])  ;P.yticks([])
```

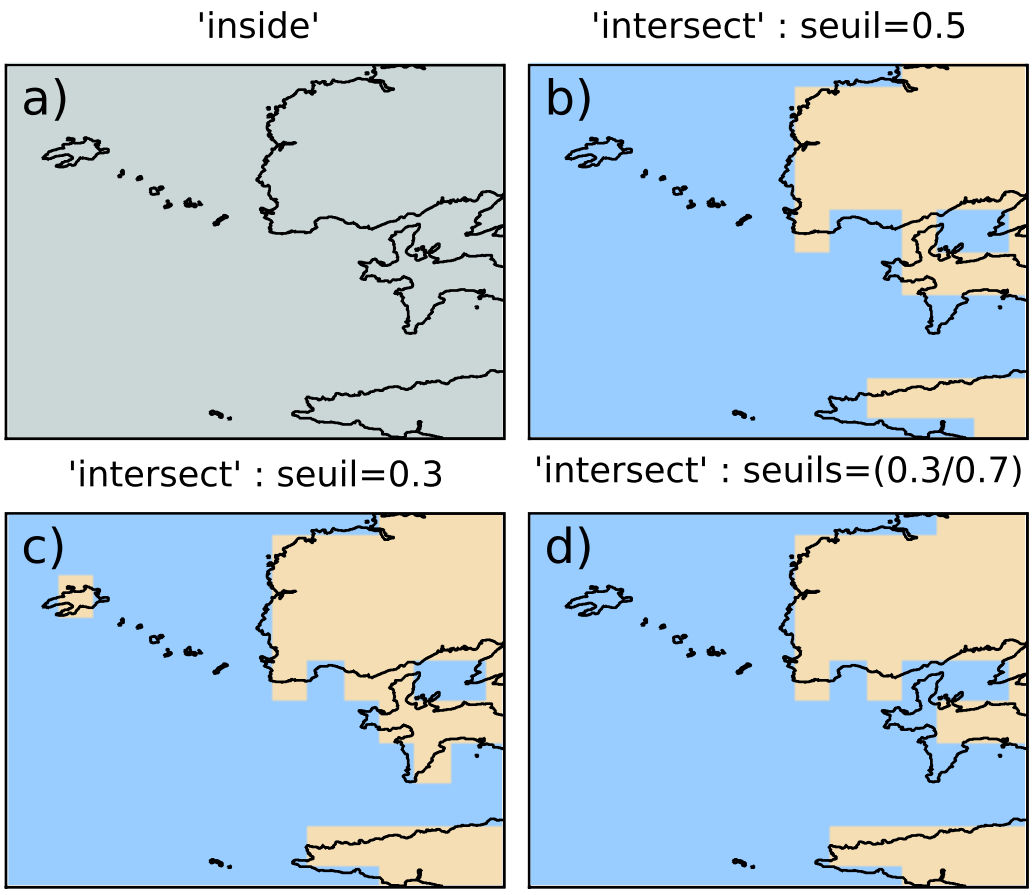


Figure 1.12: En a), une cellule est masquée si son centre est sur la terre. En b), la cellule n'est masquée que si plus de 50% se sa surface est sur de la terre. En c), comme en b) mais seulement 30% de la surface suffit à masquer la cellule. En d), comme en c) mais si la cellule a plusieurs fois de terre, il faut que la terre couvrant 70% de la cellule pour que celle-ci soit masquée.

```
P.title('Mask initial')

# Identification des lacs
lakes = GetLakes(mask)
P.subplot(222)
P.pcolor(xxb, yyb, lakes.lakes(), cmap=cmap_lakes)
P.xlim(xlim) ; P.ylim(ylim) ; P.xticks([])  ;P.yticks([])
P.title('Tous les lacs')

# Le deuxieme lac seulement
P.subplot(224)
P.pcolor(xxb, yyb, lakes.lakes(2), cmap=cmap_lakes, vmax=lakes.nlakes)
P.xlim(xlim) ; P.ylim(ylim) ; P.xticks([])  ;P.yticks([])
P.title('Deuxieme lac')

# Ocean
P.subplot(223)
P.pcolor(xxb, yyb, lakes.ocean(), cmap=cmap_mask)
P.xlim(xlim) ; P.ylim(ylim) ; P.xticks([])  ;P.yticks([])
P.title('Mask ocean')

# Save
savefigs('misc-grid-masking-lakes')
```

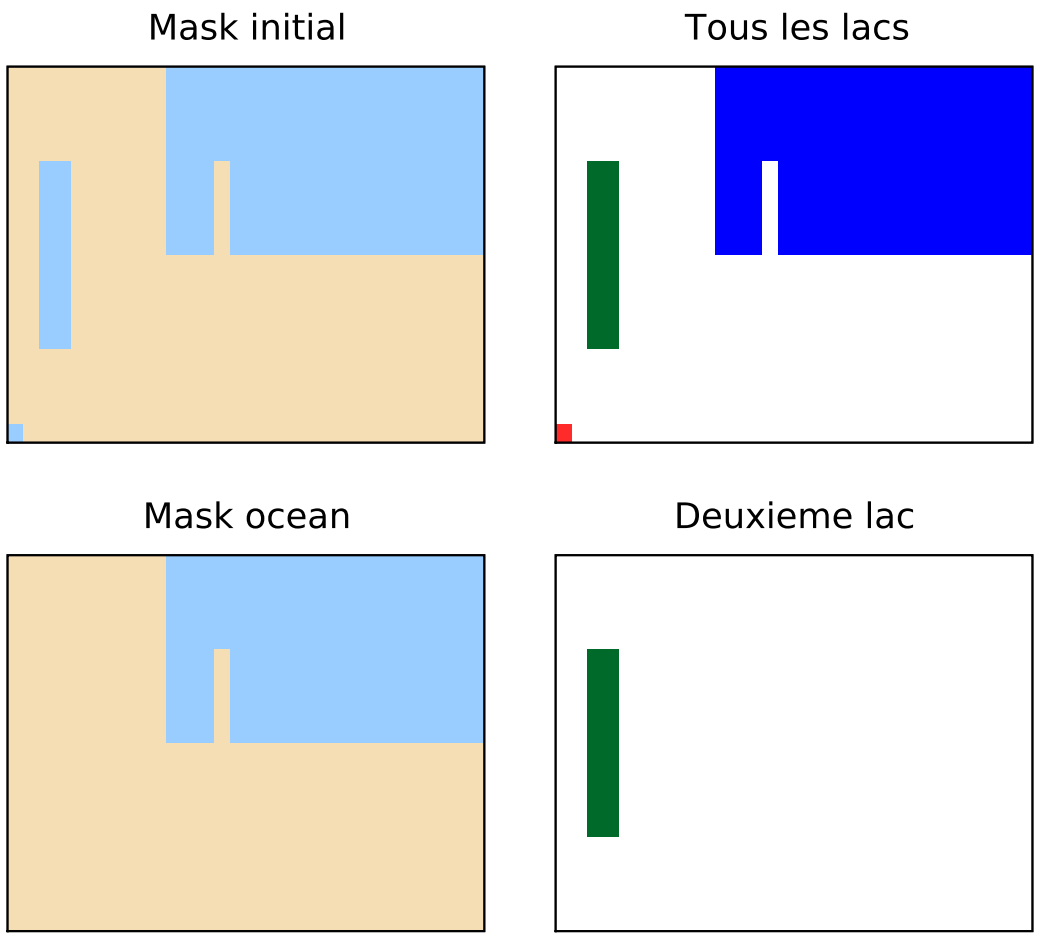


Figure 1.13: On détecte les différents lacs dans un masque. Les lacs sont d'abord distingués, puis le lac principal (océan) est tracé, puis le deuxième plus gros.

Regrillage 1D et 2D

Généralités importantes

Voir : `regridding`

Classes de regrillage On distingue deux grandes classes de regrillage :

L'interpolation Elle peut être typiquement par plus proche voisin, linéaire ou cubique, ou faire appel à des techniques plus sophistiquées en 2D. L'interpolation est à utiliser dans deux cas de figure :

Pour passer **d'une grille (ou axe en 1D) basse résolution vers une grille haute résolution**. Si elle est utilisée dans le sens opposé, elle peut être biaisée par de l'aliasing. Quand on n'a pas le choix.

Le remapping Cette technique considère les points de grille comme des cellules : pour chaque cellule cible, les cellules source recouvrant cette dernière sont moyénées avec une pondération proportionnelle aux aires de recouvrement. Elle est à utiliser pour passer **d'une grille (ou axe en 1D) haute résolution vers une grille basse résolution**.

Le tutoriel "*Remapping versus interpolation - 1D*" donne un aperçu du comportement des deux classes de regrillage.

Le problème des valeurs manquantes Ce problème est résolu en interpolant les masques eux-mêmes et en utilisant des méthodes d'ordre moins élévé. La méthode d'ordre le moins élévée, et donc celle de référence, est celle par plus proche voisin. Par exemple, près d'une valeur manquante, l'interpolation linéaire est approximée par l'interpolation par plus proche voisin. De même, celle cubique est approximée par celle linéaire, puis celle par plus proches voisins.

Dans le cas du remapping, les valeurs non masquées sont celles où le masque regrillé à une valeur par exemple inférieure à .5.

Regrillage 1D

Le regrillage 1D se fait l'un des axes d'une variable qui peut être multi-dimensionnelle. Par exemple, on peut chercher à regriller sur la verticale une variable 'tzyx'.

Le regrillage 1D supporte en outre ce que l'on pourrait appeler les "axes étendus" : un regrillage toujours 1D, mais variable suivant certains des autres axes. On utilise pour cela les mots clés `xmap` et `xmapper`.

Interpolation 1D Voir : `regrid1d()` `interp1d()` `cubic1d()` `hov()`.

```
# -*- coding: utf8 -*-
# Lecture du niveau de la mer sur 9 pas de temps à une latitude
import cdms2, MV2
f =cdms2.open('/home2/amzer/raynaud/misc/samples/previcot.mars.irhp.r3.xe.nc')
xe = f('xe', lat=(48.4, 48.41), squeeze=1, time=slice(0, 9), lon=(-5, -4.8))
f.close()
xe =  MV2.average(MV2.masked_values(xe, 999.), axis=1)
xe.long_name = 'Original'

# On crée un trou
xe[3:4, 20:30] = MV2.masked

# Nouvel axe temporel plus précis
from actimar.misc.axes import create_time
#old_time = xe.getTime()
old_time=create_time((xe.shape[0], ), 'hours since 2000')
```

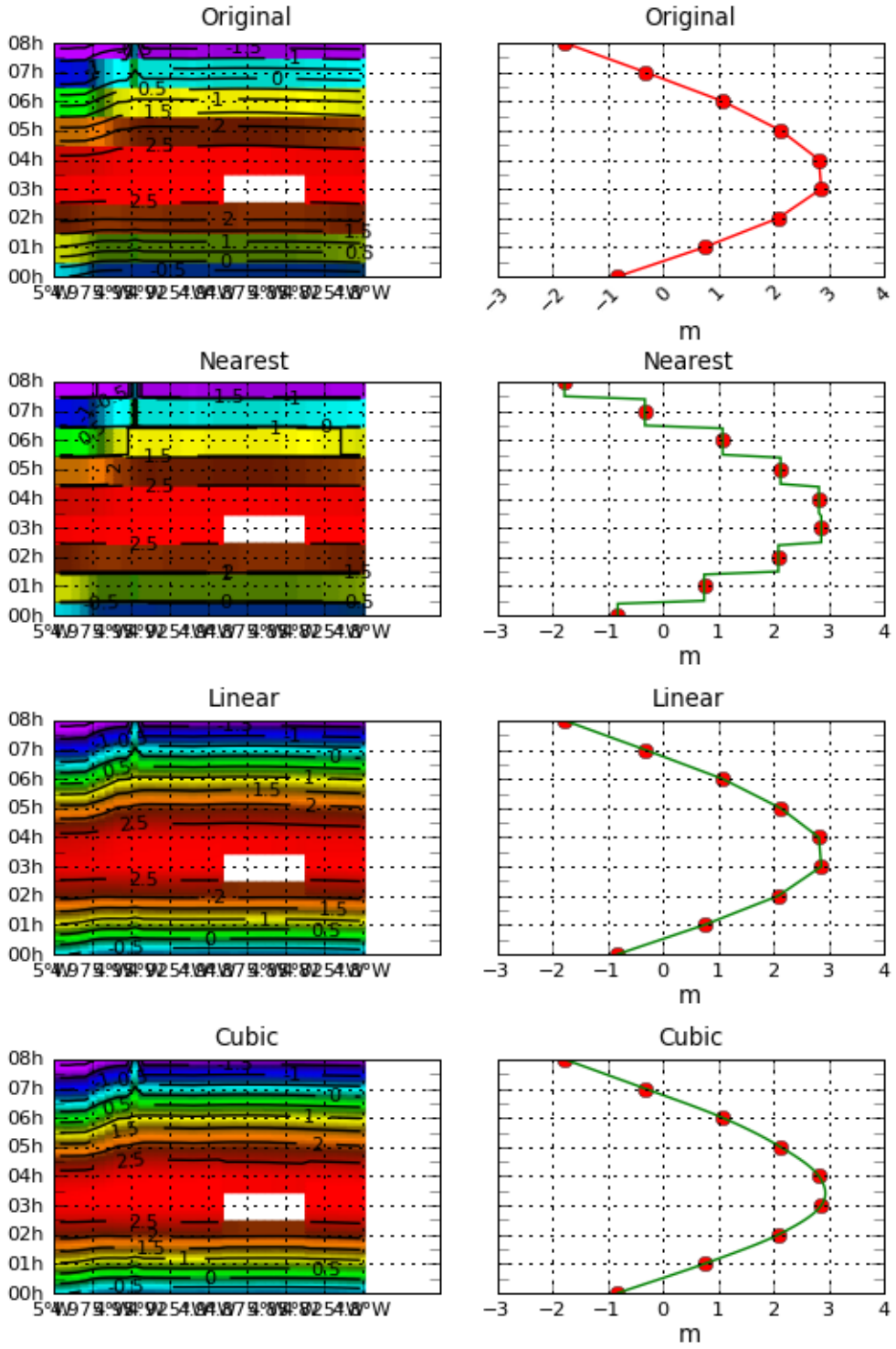


Figure 1.14: Un champ avec des valeurs manquantes sur un axes de temps basse résolution est interpolé vers un axe à plus haute résolution, par trois méthodes différentes.

```
xe.setAxis(0, old_time)
dt = (old_time[1]-old_time[0])/10.
new_time = create_time((old_time[0], old_time[-1]+dt, dt), old_time.units)

# Interpolation
from actimar.misc.grid.regridding import interp1d
# - nearest
xe_nea = interp1d(xe, new_time, method='nearest')
xe_nea.long_name = 'Nearest'
# - linear
xe_lin = interp1d(xe, new_time, method='linear')
xe_lin.long_name = 'Linear'
# - cubic
xe_cub = interp1d(xe, new_time, method='cubic')
xe_cub.long_name = 'Cubic'


# Plots
from matplotlib import rcParams ; rcParams['font.size'] = 8
import pylab as P
from actimar.misc.plot import hov, curve,  yhide, xscale, savefigs
from actimar.misc.color import cmap_jets
from genutil import minmax
vmin, vmax = minmax(xe, xe_lin)
kwplot = dict(vmin=vmin, vmax=vmax, show=False)
kwhov = dict(kwplot)
kwhov.update(cmap=cmap_jets(stretch=-.4), colorbar=False, xrotation=45.)
kwcurve = dict(kwplot)
kwcurve.update(vertical=True, linestyle='None', color='r')
# - original
hov(xe, subplot=421, top=.95, hspace=.45, figsize=(5.5, 8), bottom=.06, **kwhov)
axlims = P.axis()
curve(xe[:, 15], 'o', color='r', vertical=True, subplot=422,**kwhov)
xscale(1.1, keep_min=1)
yhide()
# - nearest
hov(xe_nea, subplot=423, **kwhov)
P.axis(axlims)
curve(xe[:, 15], 'o', subplot=424,**kwcurve)
curve(xe_nea[:, 15], vertical=True, **kwplot)
xscale(1.1, keep_min=1)
yhide()
# - linear
hov(xe_lin, subplot=425, **kwhov)
P.axis(axlims)
curve(xe[:, 15], 'o', subplot=426, **kwcurve)
curve(xe_lin[:, 15], vertical=True, **kwplot)
xscale(1.1, keep_min=1)
yhide()
# - cubic
hov(xe_cub, subplot=427,**kwhov)
P.axis(axlims)
curve(xe[:, 15], 'o', subplot=428, **kwcurve)
curve(xe_cub[:, 15], vertical=True, **kwplot)
xscale(1.1, keep_min=1)
yhide()
# - save
savefigs(__file__)
```

Remapping versus interpolation - 1D Voir : `regrid1d()` `interp1d()` `remap1d()`.

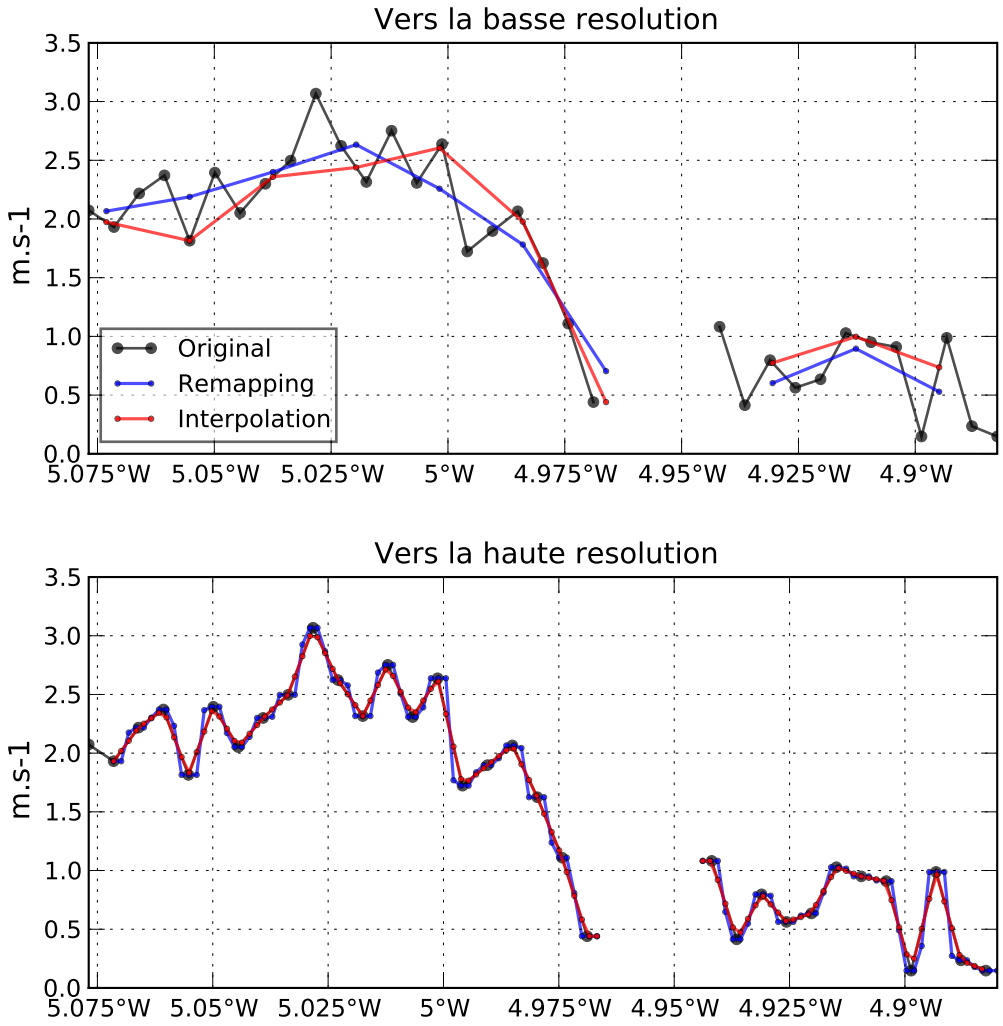


Figure 1.15: Un champ avec des valeurs manquantes sur un axes de temps basse résolution est interpolé vers un axe à plus haute résolution, par *remapping* et par interpolation linéaire.

```
# *-* coding: utf-8 *-*
# Lecture de la temperature
import cdms2, MV2, numpy as N
f =cdms2.open('/home2/amzer/raynaud/misc/samples/previcot.mars.r3.uv.nc')
v = f('v', lat=slice(251, 252), squeeze=1, lon=(-5.08, -4.88))
f.close()
v =  MV2.masked_values(v, 0.)
v.long_name = 'Original'

# On ajoute un peu de bruit
v[:] += N.random.random(len(v))

# Creation des deux axes de longitudes
from actimar.misc.axes import create_lon
lon = v.getLongitude().getValue()
dlon = N.diff(lon).mean()
# - basse résolution
lon_lr = create_lon((lon.min()+dlon*.7, lon.max()+dlon, dlon*3.3))
# - haute résolution
lon_hr = create_lon((lon.min()+dlon, lon.max()+dlon, dlon/3.3))
# - dict
lons = dict(haute=lon_hr, basse=lon_lr)

# Regrillages et plots
from matplotlib import rcParams ; rcParams['font.size'] = 9
from actimar.misc.grid.regridding import regrid1d
from actimar.misc.plot import curve,savefigs, yscale ; import pylab as P
P.figure(figsize=(5.5, 6))
kwplot = dict(show=False,vmin=v.min(),vmax=v.max(),alpha=.7)
for ilh,resdst  in enumerate(['basse', 'haute']):

        # Regrillage
        vlinear = regrid1d(v, lons[resdst], 'linear')
        vremap = regrid1d(v, lons[resdst], 'remap')

        # Plots
        P.subplot(2, 1, ilh+1)
        curve(v, 'o', markersize=4, color='k', label=u'Original', hspace=.3, **kwplot)
        curve(vremap, 'o', markersize=2, label=u'Remapping', linewidth=1.2, color='b', **kwplot)
        curve(vlinear, 'o', markersize=2, label=u'Interpolation', linewidth=1.2, color='r', **kwplot)
        yscale(1.1)
        P.title(u'Vers la %s resolution'%resdst)
        if not ilh: P.legend(loc='lower left').legendPatch.set_alpha(.6)

savefigs(__file__, pdf=True)
```

Regrillage 1D étendu Voir : `regrid1d()` `interp1d()` `cubic1d()` `hov()`, et le tutoriel "*Interpolation 1D*.

```
# -*- coding: utf-8 -*-
# Lecture de la temperature
import cdms2, MV2, numpy as N
f =cdms2.open('/home2/amzer/raynaud/misc/samples/mars3d.nc')
t = f('temp', lat=slice(97, 98), squeeze=1, lon=(-5., -4.43), time=slice(12, 13))
h0 = f('h0', lat=slice(97, 98), squeeze=1, lon=(-5., -4.43)).filled()
f.close()
t =  MV2.masked_values(t, 999.)
t.long_name = 'Original'
h0[h0<0.] = 0.

# Creation d'un axe de profondeur
```

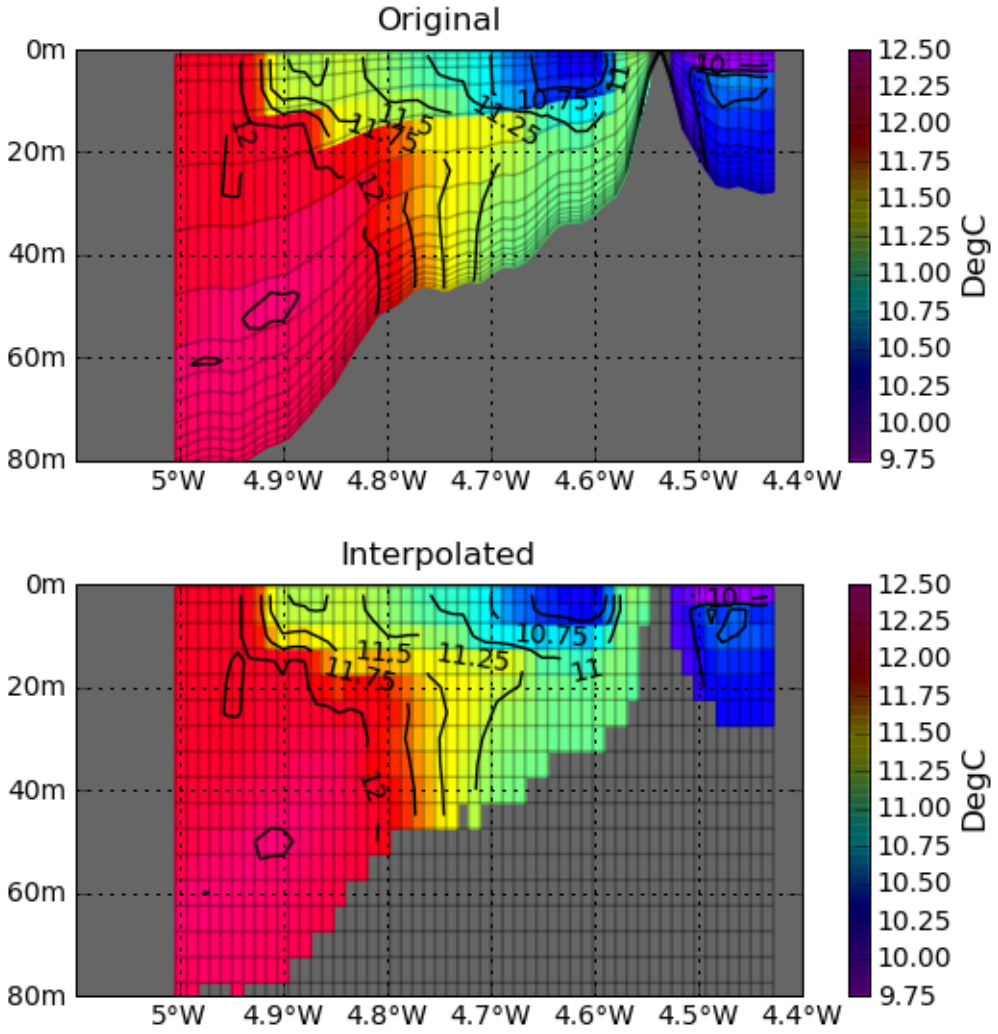


Figure 1.16: Interpolation linéaire sur selon un axe (vertical) dont les coordonnées varient selon 1 ou plusieurs des autres axes (zonal). On fait alors appel aux mots clés `xmap` désignant les axes sur lesquels les coordonnées varient, et `xmapper` pour spécifier ces coordonnées.

```
from actimar.misc.axes import create_dep
ddep = 5.
dep = create_dep((0., h0.max()+ddep, ddep))

# Creation de l'axe etendu (taille (ndep,nx))
depths = N.outer(t.getAxis(0)[:], h0)
dep[0] = depths[0].max()

# Regrillage lineaire
from actimar.misc.grid.regridding import regrid1d
tr = regrid1d(t, dep, 'linear', axis=0, xmap=-1, xmapper=depths.transpose())
tr.long_name = 'Interpolated'

# Plot
from actimar.misc.plot import section,savefigs,yscale, add_grid
import pylab as P
P.figure(figsize=(5.5, 6))
# - original
section(t, yaxis=depths, subplot=211, hspace=.3, show=False, ylim=(-80, 0))
add_grid((t.getLongitude(), -depths[:]), alpha=.3)
# - regridded
section(tr, subplot=212, show=False, ylim=(-80, 0))
add_grid((tr.getLongitude(), -dep[:]), alpha=.3)
savefigs(__file__)
```

Remappind conservatif 1D Le remapping conservatif est très similaire au remapping simple (`method="remap"`), et a pour but de conserver la somme du champ d'entrée vers le champ de sortie. Dans le cas conservatif, on n'effectue pas des moyennes mais des sommes.

Cette méthode est utile pour regriller par exemple des précipitations dans le temps.

Voir : `regrid1d()`.

```
# -*- coding: utf-8 -*-
# Creation d'un jeu de precipitations horaires
import MV2, cdms2, numpy as N
from actimar.misc.axes import create_time
hours = create_time((12*60, 25.*60,  60), 'minutes since 2000')
precip = MV2.sin(N.arange(len(hours))*.2)*10
precip.setAxis(0, hours)

# Nouvel echantillonnage / 2h
hours2 = create_time((10, 30., 2), 'hours since 2000')

# Regrillage 1D conservatif
from actimar.misc.grid.regridding import regrid1d
precip2 = regrid1d(precip, hours2, 'conservative')

# Verifications
print 'Total precip.:'
print '- original =', precip.sum()
print '- remapped =', precip2.sum()
# > Total precip.:
# > - original = 89.957242832779755
# > - remapped = 89.957237



# Plots
import pylab as P
from actimar.misc.atime import mpl
```

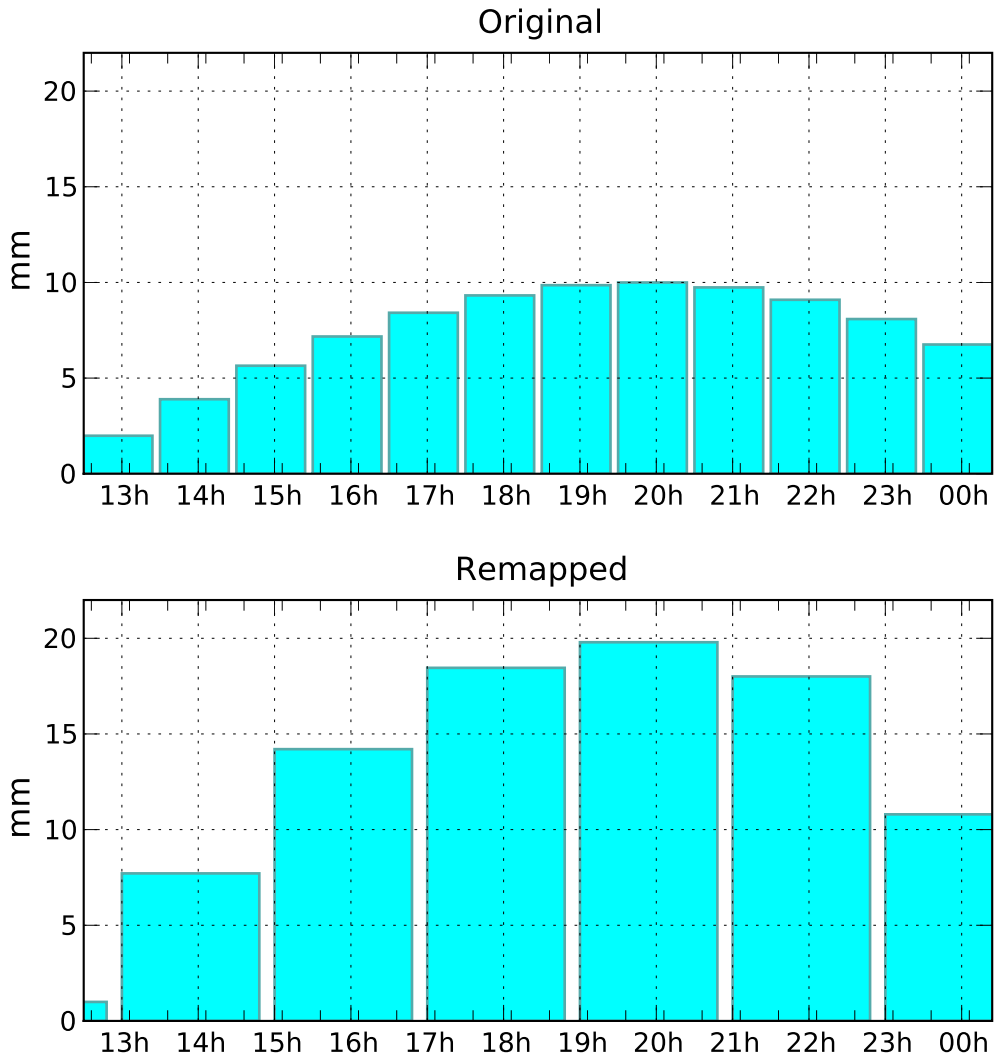


Figure 1.17: Des précipitations horaires sont réparties sur un axe de temps bi-horaire.

```
from actimar.misc.plot import xdate, savefigs
P.figure(figsize=(5.5, 6))
kwplot = dict(color='#00ffff',edgecolor='#55aaaa')
# - original
P.subplot(211)
dt = 1/24.
dates = mpl(hours)[:]-dt/2
P.bar(dates, precip.filled(0.), width=dt*.9, **kwplot)
xdate(fmt='%Hh', rotation=0)
P.ylabel('mm')
P.title('Original')
P.ylim(ymax=22)
axis = P.axis()
P.grid(True)
# - remapped
P.subplot(212)
dt2 = 2/24.
dates2 = mpl(hours2)[:]-dt2/2
P.bar(dates2, precip2.filled(0.), width=dt2*.9, **kwplot)
P.ylabel('mm')
P.title('Remapped')
xdate(fmt='%Hh', rotation=0)
P.axis(axis)
P.grid(True)
P.subplots_adjust(hspace=.3, bottom=.08)
savefigs(__file__, pdf=True)
```

Regrillage 2D

Interpolation vers grille régulière Voir `griddata()`.

```
# On construit un grille reguliere
import numpy as N
from actimar.misc.grid import meshbounds
xr = N.arange(20.)
yr = N.arange(10.)
xxr, yyr = N.meshgrid(xr, yr)
xrb, yrb = meshbounds(xr, yr)
zzr = (N.sin(xxr*N.pi/6)*N.sin(yyr*N.pi/6) + \
        N.exp(-((xxr-7.)**2+(yyr-7.)**2)/4.**2))*100.
zzr -= zzr.min()
vminmax=dict(vmin=zzr.min(), vmax=zzr.max())

# On construit un echantillon irregulier a partir du regulier
ij = (N.random.rand(50)*zzr.size).astype('i')
xi, yi, zi = xxr.flat[ij], yyr.flat[ij], zzr.flat[ij]

# Interpolation sur la grille reguliere
# - natgrid
from actimar.misc.grid.regridding import griddata
zirn = griddata(xi, yi, zi, (xr, yr), method='nat', sub=6)
# - krigeage
zirk = griddata(xi, yi, zi, (xr, yr), method='krig')

# Plot de verif
import pylab as P
from actimar.misc.plot import savefigs
P.figure(1, figsize=(6, 8))
P.subplots_adjust(hspace=.3, bottom=.05, top=.95, left=.06)
# - regulier
```

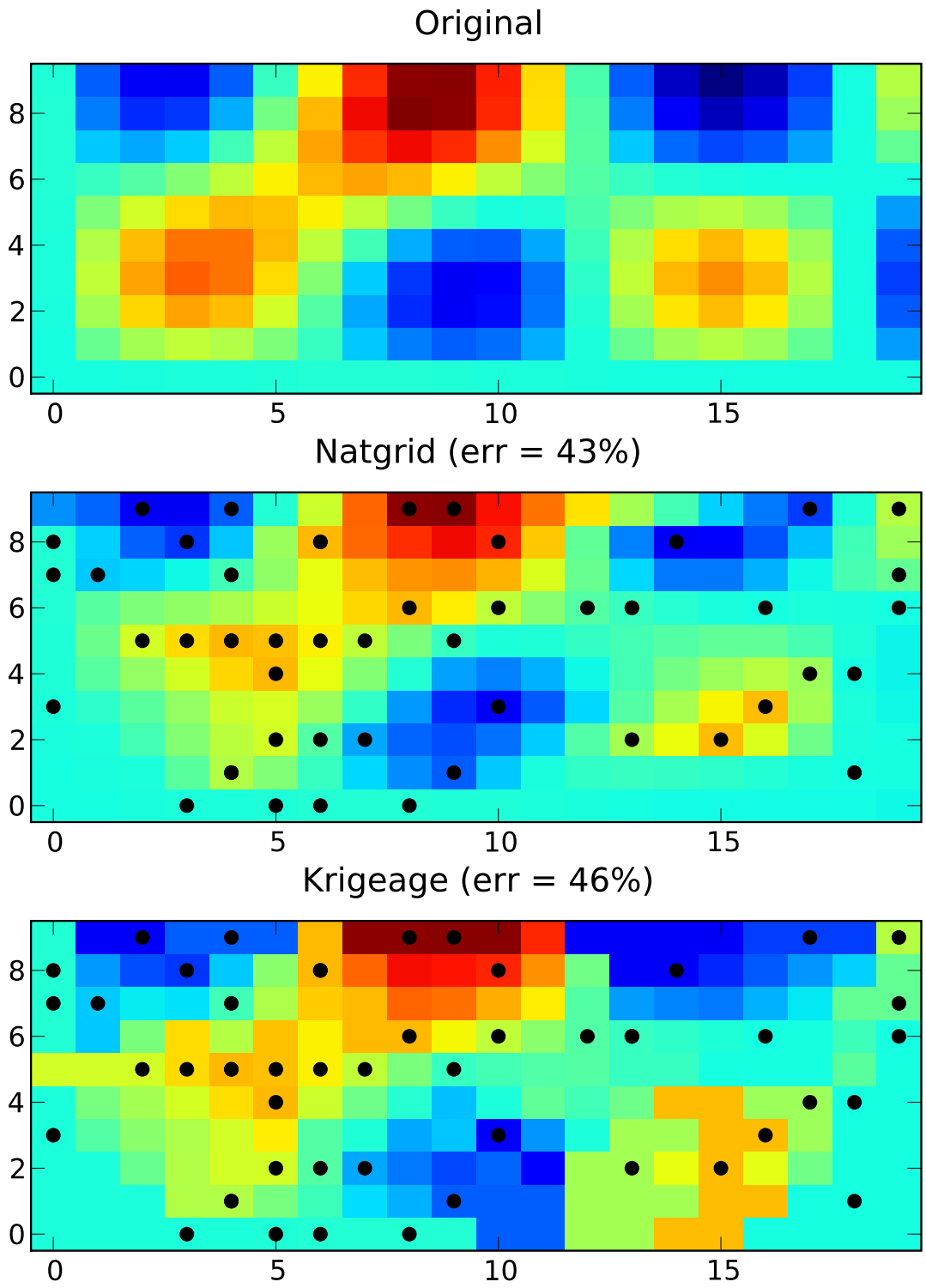


Figure 1.18: Des points aléatoires sont interpolés vers une grille régulière grâce à la libraire NatGrid (au milieu), et par krigeage (en bas).

```
P.subplot(311)
P.pcolor(xrb, yrb, zzr, **vminmax)
P.xlim(xrb.min(), xrb.max()) ; P.ylim(yrb.min(), yrb.max())
P.title('Original')
stdref = zzr.std()
# - irregulier via natgrid
P.subplot(312)
P.pcolor(xrb, yrb, zirn, **vminmax)
P.plot(xi, yi, 'ko')
P.xlim(xrb.min(), xrb.max()) ; P.ylim(yrb.min(), yrb.max())
P.title('Natgrid (err = %02i%%)'%((zzr-zirn).std()*100/stdref))
# - irregulier via krig
P.subplot(313)
P.pcolor(xrb, yrb, zirk, **vminmax)
P.plot(xi, yi, 'ko')
P.xlim(xrb.min(), xrb.max()) ; P.ylim(yrb.min(), yrb.max())
P.title('Krigeage (err = %02i%%)'%((zzr-zirk).std()*100/stdref))
savefigs('misc-grid-regridding-griddata')
P.show()
```

Interpolation entre deux grilles régulières Voir : "*Interpolation vers grille régulière*" `interp2d()` `regrid2d()` `add_grid()` `griddata()` `krigdata()`.

```
# -*- coding: utf8 -*-
# Variable d'entree
import numpy as N, cdms2 as cdms, MV2 as MV
from actimar.misc.grid import meshbounds
xi = N.arange(20.)
yi = N.arange(10.)
xxi, yyi = N.meshgrid(xi, yi)
xib, yib = meshbounds(xi, yi)
vari = (N.sin(xxi*N.pi/6)*N.sin(yyi*N.pi/6) +
        N.exp(-((xxi-7.)**2+(yyi-7.)**2)/4.**2))*100.
vminmax=dict(vmin=vari.min(), vmax=vari.max())
vari = cdms.createVariable(vari)
vari.setAxis(-2, cdms.createAxis(yi))
vari.setAxis(-1, cdms.createAxis(xi))
vari[3:4, 3:7] = MV.masked

# Grille de sortie
xo = cdms.createAxis(N.linspace(-3., 23., 70))
yo = cdms.createAxis(N.linspace(-3., 13., 40))

# Interpolation
from actimar.misc.grid.regridding import interp2d
# - bilinear
varob = interp2d(vari, (xo, yo), method='bilinear')
# - natgrid
varon = interp2d(vari, (xo, yo), method='nat', hor=1.)

# Plot
import pylab as P
from actimar.misc.plot import savefigs, add_grid
xob, yob = meshbounds(xo[:], yo[:])
lims = [xob.min(), xob.max(), yob.min(), yob.max()]
# -
P.figure(figsize=(5.5, 7))
P.subplots_adjust(bottom=.07, hspace=.35)
P.subplot(311)
P.pcolor(xib, yib, vari, **vminmax)
P.axis(lims)
```

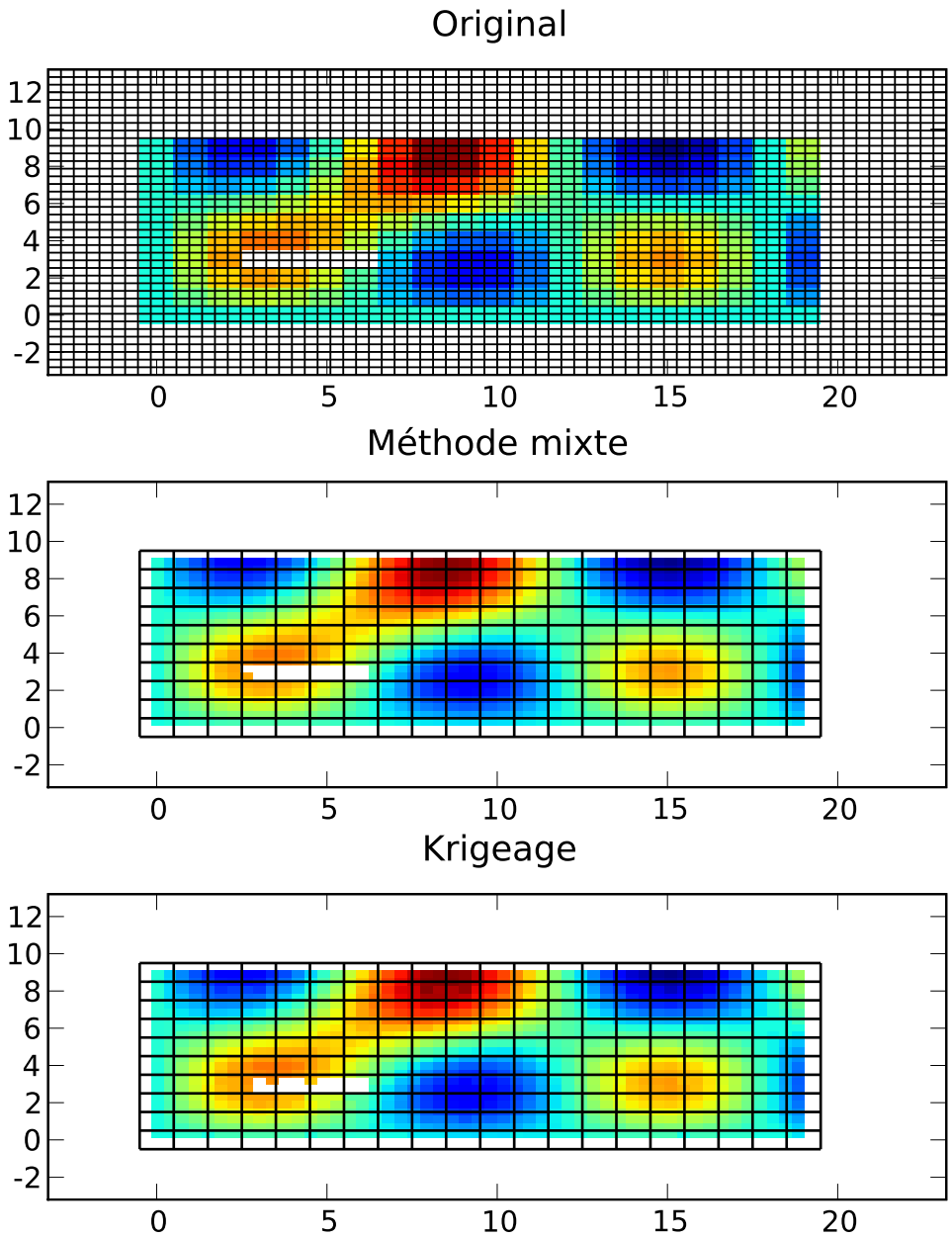


Figure 1.19: Un champ avec des valeurs manquantes sur une grille régulière basse résolution est interpolé vers une grille à plus haute résolution, par deux méthodes différentes.

```
add_grid((xo, yo), linewidth=.7, alpha=.4)
P.title('Original')
# -
P.subplot(312)
P.pcolor(xob, yob, varob, **vminmax)
P.axis(lims)
add_grid((xi, yi))
P.title(u'Bilinéaire')
# -
P.subplot(313)
P.pcolor(xob, yob, varon, **vminmax)
P.axis(lims)
add_grid((xi, yi))
P.title(u'Natgrid')
# -
savefigs(__file__)
P.show()
```

D'une grille curvilinéaire vers une grille rectangulaire Voir : "*Interpolation vers grille régulière*" regrid2d() add_grid()griddata() krigdata() scrip() SCRIP.

```
# -*- coding: utf8 -*-
from matplotlib import rc;rc('font', size=11)
import cdms2, MV2
zone = dict(yc=slice(0, 40), xc=slice(10, 40))
f = cdms2.open('/home/raynaud/data/misc/samples/swan.four.nc')
lon2d = f('longitude', **zone)
lat2d = f('latitude', **zone)
hs = MV2.masked_values(f('HS', time=slice(0, 1), squeeze=1, **zone), f['HS']._FillValue)
dlon = float(f.longitude_resolution)
dlat = float(f.latitude_resolution)
f.close()

# Affectation de la grille curvilineaire à la variable
from actimar.misc.grid import curv_grid, set_grid
cgrid = curv_grid(lon2d, lat2d)
hs = set_grid(hs, cgrid)

# Création de la nouvelle grille rectangulaire de résolution équivalente
from actimar.misc.axes import create_lon, create_lat
import numpy as N
lon1d = create_lon(N.arange(lon2d.min(), lon2d.max()+dlon/2., dlon))
lat1d = create_lat(N.arange(lat2d.min(), lat2d.max()+dlat/2., dlat))
rect_grid = cdms2.createRectGrid(lat1d, lon1d)

print 'Regrillage'
from actimar.misc.grid.regridding import regrid2d
print ' - par plus proche voisins'
hs_nearest = regrid2d(hs, rect_grid, method='nearest')
print ' - par SCRIP/conservative'
hs_scripcons = regrid2d(hs, rect_grid, method='conservative')
print ' - par SCRIP/bilineaire'
hs_scripbilin = regrid2d(hs, rect_grid, method='bilinear')
print ' - par krigeage'
hs_krig = regrid2d(hs, rect_grid, method='krig')
print ' - par natgrid'
hs_nat = regrid2d(hs, rect_grid, method='nat')

print 'Plots'
from matplotlib import rcParams ; rcParams['font.size'] = 10
import pylab as P
```

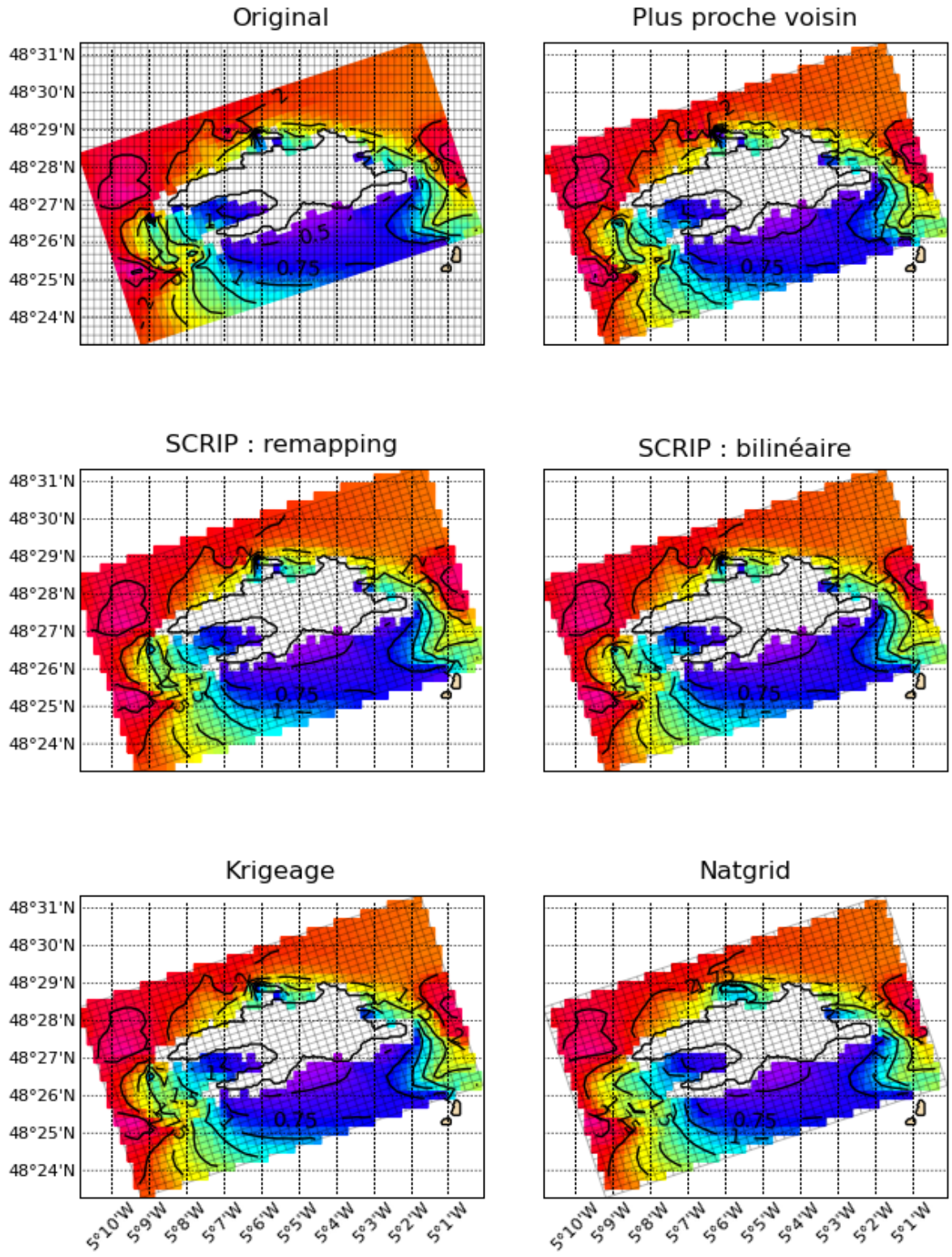


Figure 1.20: Regrillage d'une grille curvilinéaire (SWAN) vers une grille rectangulaire, par plusieurs méthodes.

```
from actimar.misc.plot import map, savefigs, xhide, yhide, add_grid
P.figure(figsize=(6.5, 9))
P.subplots_adjust(hspace=.22, bottom=.06, left=.09, right=.98, top=.95)
kwplot = dict(show=False, colorbar=False, vmin=hs.min(), vmax=hs.max(),
        drawparallels_size=8, drawmeridians_size=8, drawmeridians_rotation=45.)
m = map(hs, title='Original', subplot=321, xhide=1,**kwplot)
add_grid(rect_grid, lw=.7, alpha=.3)
map(hs_nearest, title='Plus proche voisin',yhide=1,xhide=1,subplot=322,m=m,**kwplot)
add_grid(cgrid, lw=.7, alpha=.3)
map(hs_scripcons, title='SCRIP : remapping', subplot=323, xhide=1,m=m, **kwplot)
add_grid(cgrid, lw=.7, alpha=.3)
map(hs_scripbilin, title=u'SCRIP : bilinéaire',subplot=324,xhide=1,yhide=1,m=m, **kwplot)
add_grid(cgrid, lw=.7, alpha=.3)
map(hs_krig, title='Krigeage', subplot=325, m=m, **kwplot)
add_grid(cgrid, lw=.7, alpha=.3)
map(hs_nat, title='Natgrid', subplot=326, yhide=1,m=m, **kwplot)
add_grid(cgrid, lw=.7, alpha=.3)
savefigs(__file__)
```

D'une grille curvilinéaire vers une grille curvilinéaire Voir : "*Interpolation vers grille régulière*" `regrid2d()` `add_grid()` `griddata()` `scrip()` `SCRIP`.

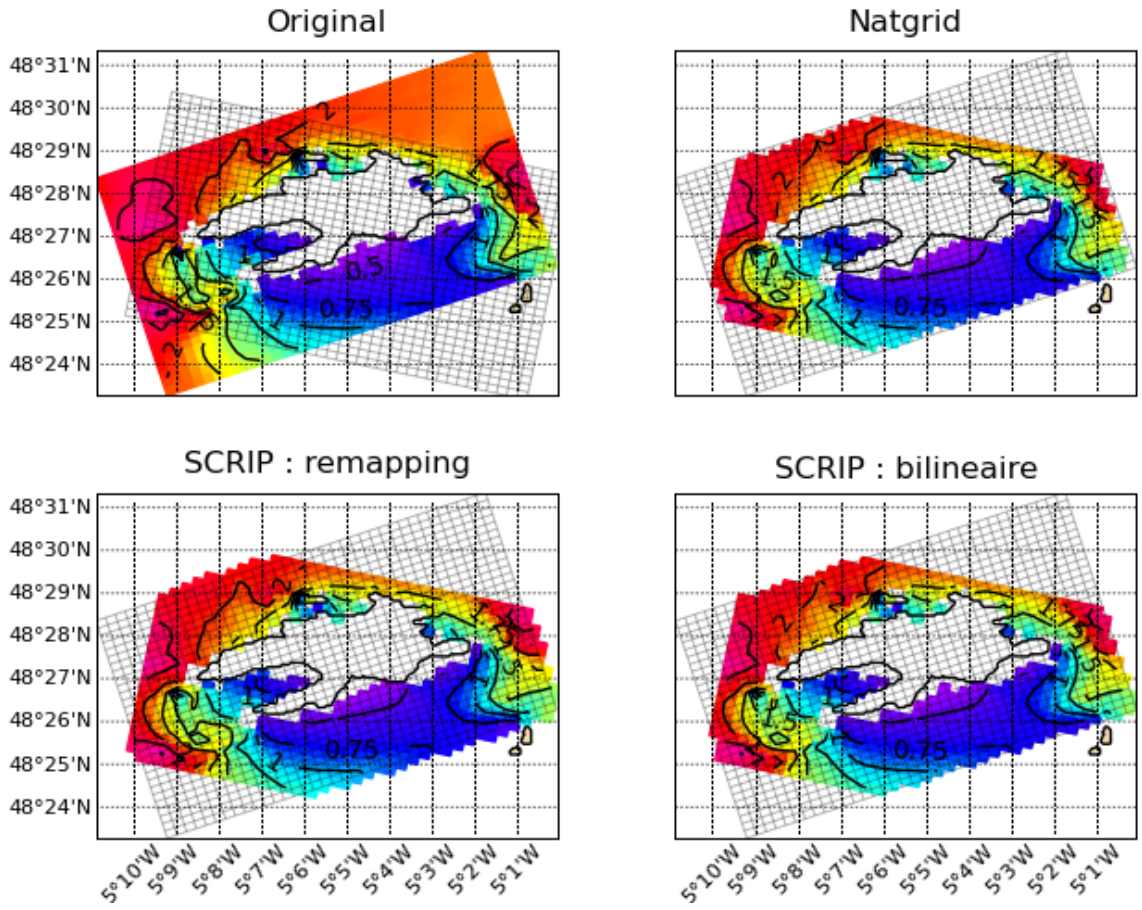


Figure 1.21: Regrillage d'une grille curvilinéaire (SWAN) vers une autre grille curvilinéaire, avec Natgrid et SCRIP.

```
# -*- coding: utf8 -*-
# Lecture des donnees
```

```
from matplotlib import rc;rc('font', size=11)
import cdms2, MV2
zone = dict(yc=slice(0, 40), xc=slice(10, 40))
f = cdms2.open('/home/raynaud/data/misc/samples/swan.four.nc')
lon2d = f('longitude', **zone)
lat2d = f('latitude', **zone)
hs = MV2.masked_values(f('HS', time=slice(0, 1), squeeze=1, **zone), f['HS']._FillValue)
dlon = float(f.longitude_resolution)
dlat = float(f.latitude_resolution)
f.close()

# Affectation de la grille curvilineaire à la variable
from actimar.misc.grid import curv_grid, set_grid
cgridi = curv_grid(lon2d, lat2d)
hs = set_grid(hs, cgridi)

# Rotation de la grid
from actimar.misc.grid import rotate_grid
import numpy as N
cgrido = rotate_grid(hs.getGrid(), -30)


print 'Regrillage'
from actimar.misc.grid.regridding import regrid2d
print ' - par natgrid'
hs_nat = regrid2d(hs, cgrido, 'nat')
print ' - par SCRIP/conservative'
hs_scripcons = regrid2d(hs, cgrido, 'conservative')
print ' - par SCRIP/bilineaire'
hs_scripbilin = regrid2d(hs, cgrido, 'bilinear')


print 'Plots'
from matplotlib import rcParams ; rcParams['font.size'] = 10
import pylab as P
from actimar.misc.plot import map, savefigs, xhide, yhide, add_grid
P.figure(figsize=(6.5, 5))
P.subplots_adjust(hspace=.28, bottom=.07, left=.08, right=.98)
kwplot = dict(show=False, colorbar=False, vmin=hs.min(), vmax=hs.max(),
        drawparallels_size=8, drawmeridians_size=8, drawmeridians_rotation=45.)
m = map(hs, title='Original', subplot=221, xhide=1,**kwplot)
add_grid(cgrido, lw=.7, alpha=.3)
map(hs_nat, title='Natgrid', subplot=222, xhide=1,yhide=1,m=m, **kwplot)
add_grid(cgridi, lw=.7, alpha=.3)
map(hs_scripcons, title='SCRIP : remapping', subplot=223, m=m, **kwplot)
add_grid(cgridi, lw=.7, alpha=.3)
map(hs_scripbilin, title=u'SCRIP : bilinéaire',subplot=224,yhide=1,m=m, **kwplot)
add_grid(cgridi, lw=.7, alpha=.3)
savefigs(__file__)
```

Interpolation de données grillées vers des positions aléatoires Cette routine est particulièrement adaptée pour passer d'une grille rectangulaire vers une grille non structurée.

Voir : `grid2xy()`.

```
# Lecture des donnnees
import cdms2, MV2, numpy as N
select=dict(lon=(-5.3, -4.72), lat=(47.9, 48.8), time=slice(0, 24))
f = cdms2.open('/home2/amzer/raynaud/misc/samples/mars2d.gascogne.nc')
v =  MV2.masked_values(f('v',**select), 0., copy=False)
f.close()
```

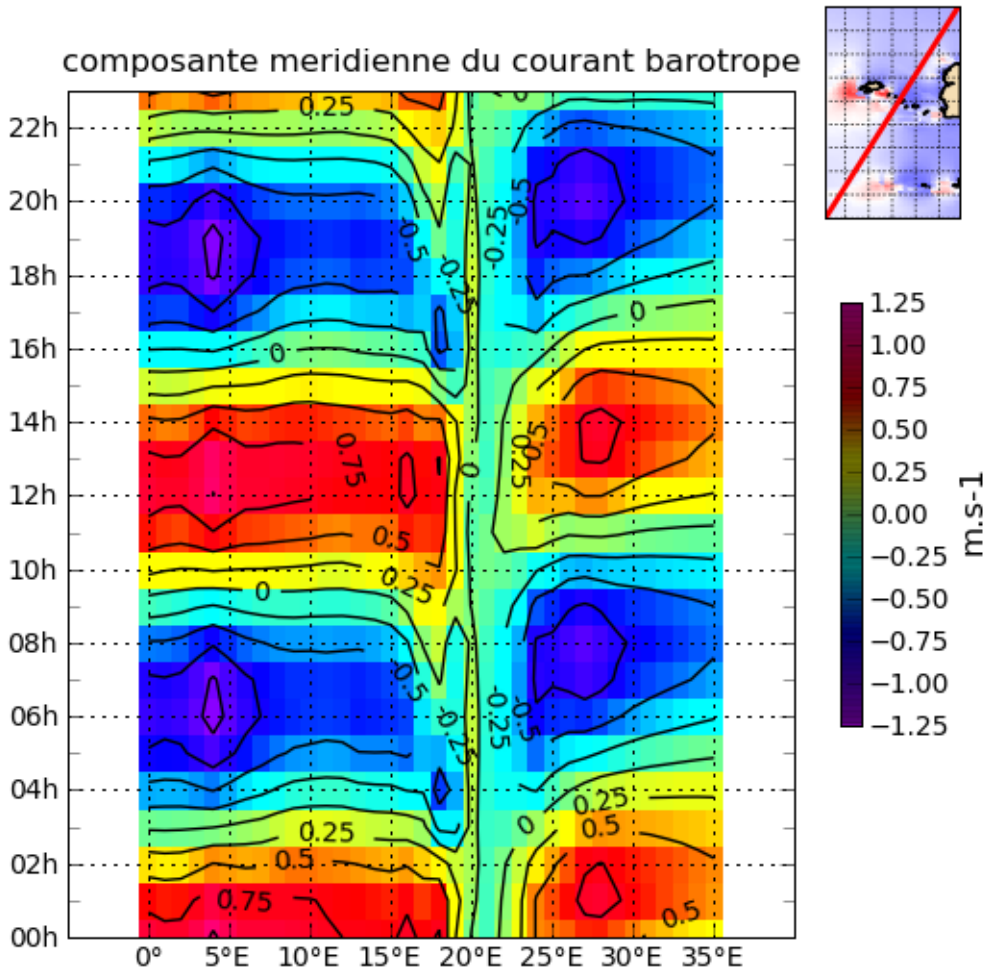


Figure 1.22: Interpolation de données grillées vers des positions aléatoires.

```
# Diagonale
lon = v.getLongitude()
lat = v.getLatitude()
nd = N.sqrt(len(lon)**2.+len(lat)**2)/2.
xo = N.linspace(lon[0], lon[-1], nd)
yo = N.linspace(lat[0], lat[-1], nd)

# Interpolation
from actimar.misc.grid.regridding import grid2xy
vo = grid2xy(v, xo, yo, method='bilinear')

# Plot
# - variable interpolee
from actimar.misc.plot import hov, savefigs, map
hov(vo, show=False,  top=.9, colorbar_shrink=.5)
# - carte + diagonal
import pylab as P
m = map(v[0],  xhide=True, yhide=True, contour=False, title=False, autoresize=0,
        colorbar=False, axes_rect=[.78, .78, .2, .2], cmap='cmap_bwr', show=False)
m.plot(xo, yo, 'r-', lw=2)
savefigs(__file__)
```

Interpolation entre des positions aléatoires Cette routine est particulièrement adaptée à l'interpolations entre des grilles non structurées.

Voir : `xy2xy()`.

```
# Donnees d'entree
import numpy as N
ni=100
xi = N.random.random(ni)-.5
yi = N.random.random(ni)-.5
zi = N.exp(-(xi**2+yi**2))#+N.random.random(ni)/10.
zi = N.ma.asarray(zi)
zi[(N.abs(yi)<.1)&(N.abs(xi)<.1)] = N.ma.masked

# Donnees de sortie
no = 40
xo = N.random.random(no)-.5
yo = N.random.random(no)-.5

# Regrillage
from actimar.misc.grid.regridding import xy2xy
zo = xy2xy(xi, yi, zi, xo, yo)

# Plots
import pylab as P
from actimar.misc.plot import savefigs
P.figure(figsize=(5, 7))
P.subplot(211)
P.title('Original')
P.scatter(xi, yi, c=zi, s=50, vmin=zi.min(), vmax=zi.max())
P.scatter(xi[zi.mask], yi[zi.mask], s=50, c='.5')
axlims = P.axis()
P.subplot(212)
P.title('Interplated')
P.scatter(xo, yo, c=zo, s=50, vmin=zi.min(), vmax=zi.max(), label='Data')
P.scatter(xo[zo.mask], yo[zo.mask], s=50, c='.5', label='Missing')
P.legend().legendPatch.set_alpha(.5)
P.axis(axlims)
savefigs(__file__, pdf=True)
```

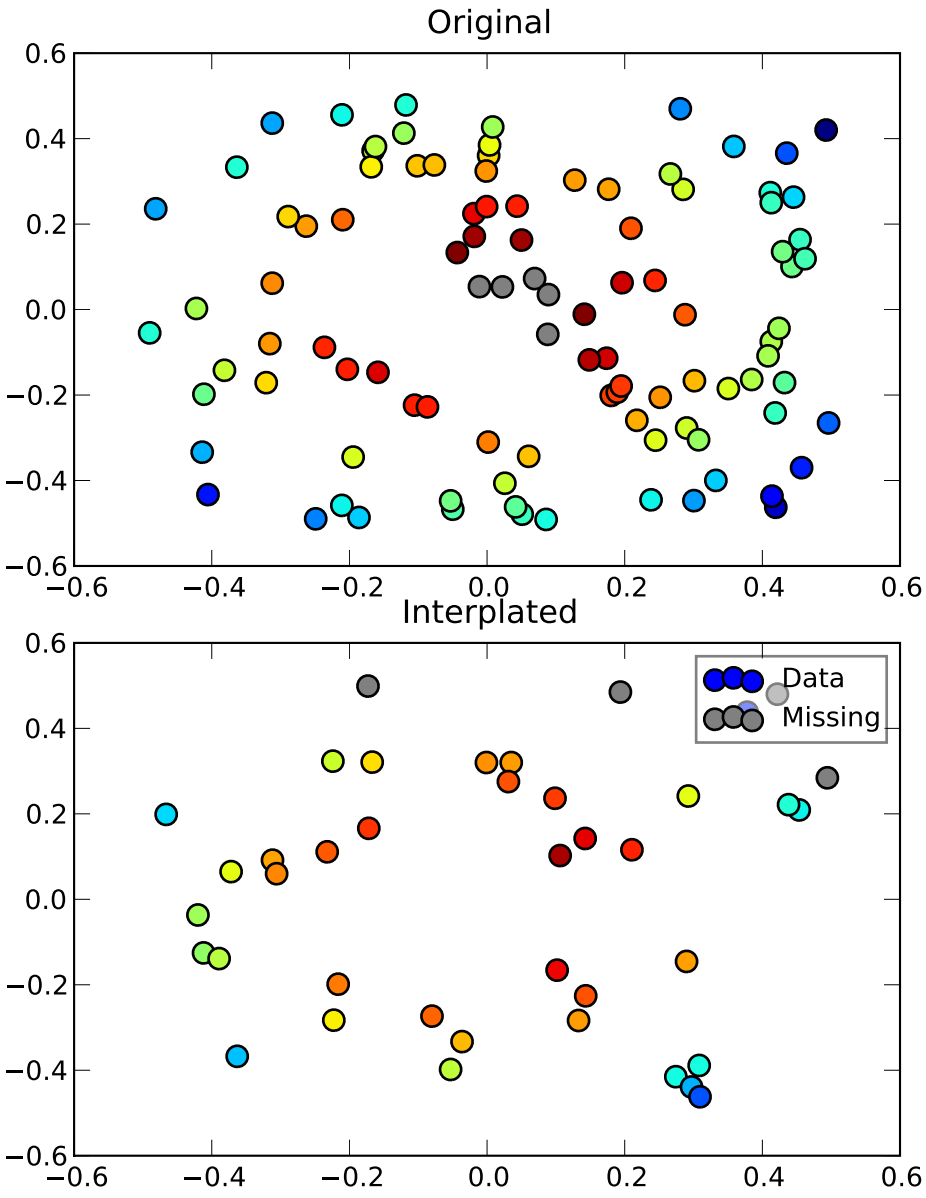


Figure 1.23: Interpolation de données entre des positions aléatoires.

Remplissage des valeurs manquantes

Remplissage 1D Voir : `fill1d()` `interp1d()` `fill2d()`.

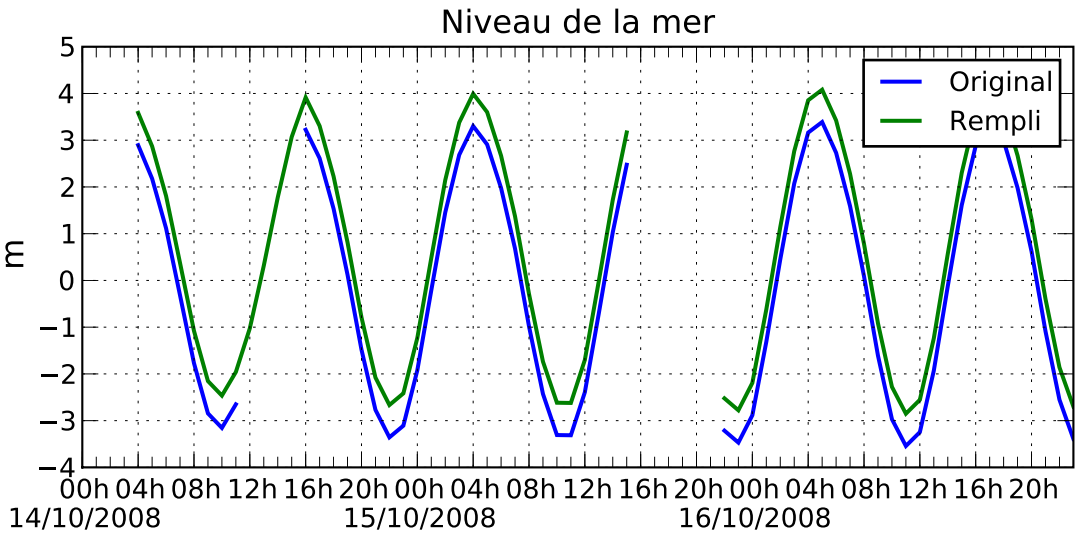


Figure 1.24: Replissage de valeurs manquantes par interpolation le long d'un axe.

```
# -*- coding: utf8 -*-
# Lecture du niveau de la mer horaire
import cdms2, MV2
f =cdms2.open('/home2/amzer/raynaud/misc/samples/previcot.mars.irhp.r3.xe.nc')
xe = f('xe', time=slice(72), lat=slice(50, 51), lon=slice(50, 51), squeeze=1)
f.close()
xe.long_name = 'Original'

# On crée des trous
# - petits
xe[:4] = MV2.masked
xe[12:16] = MV2.masked
# - gros
xe[40:46] = MV2.masked

# On rempli les petits trous (5 heures max) par interpolation cubique
from actimar.misc.grid.regridding import fill1d
import time ; t0=time.time()
xef = fill1d(xe, method='cubic', maxgap=5)
print time.time()-t0
xef.long_name = 'Rempli'
xef[:] += xef.max()/5. # on décalle pour les plots

# Plots
from actimar.misc.plot import curve, P, savefigs
curve(xe, show=False, linewidth=1.5, figsize=(6, 3), top=.88, bottom=.15)
curve(xef, show=False, linewidth=1.5, title='Niveau de la mer')
P.legend()
savefigs(__file__, pdf=True)
```

Remplissage 2D Voir : `fill2d()griddata()` `fill1d()`.

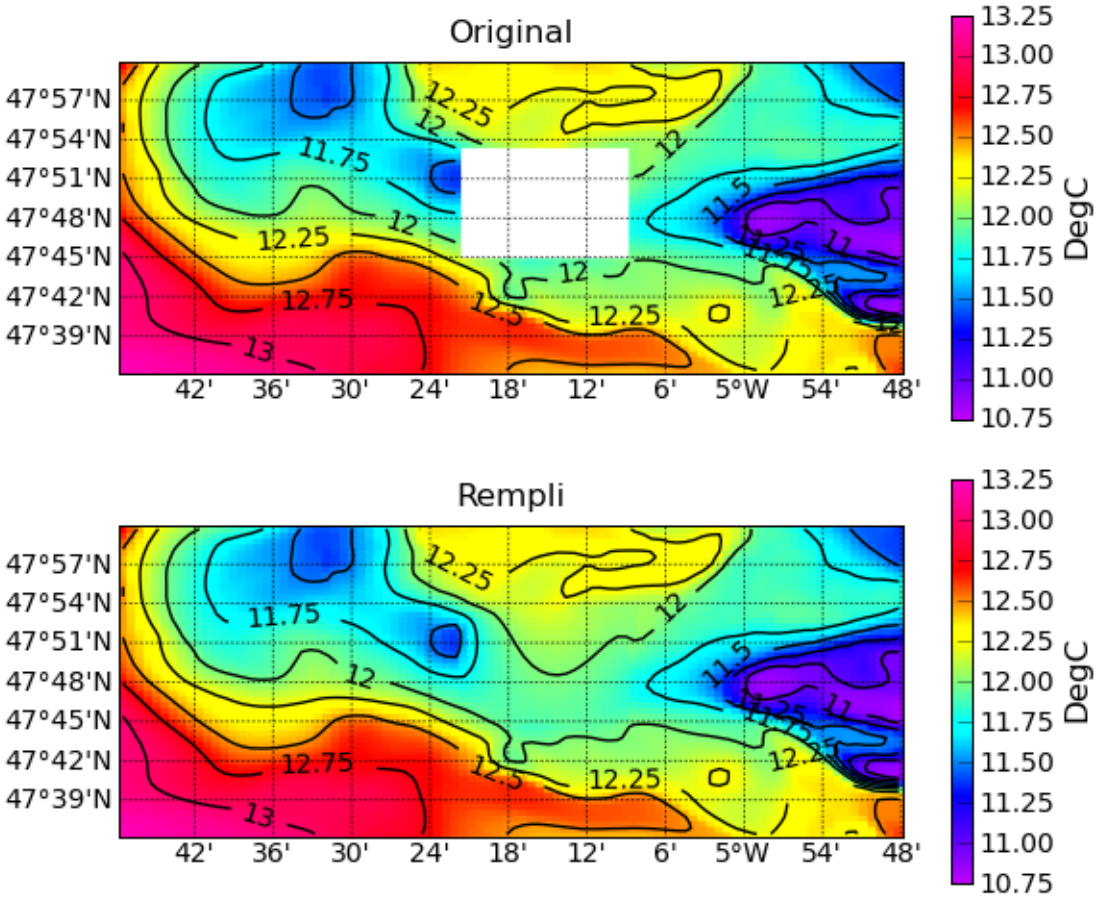


Figure 1.25: Replissage de valeurs manquantes par interpolation spatiale.

```
# -*- coding: utf8 -*-
# Lecture de la temperature
import cdms2, MV2
f = cdms2.open('/home2/amzer/raynaud/misc/samples/mars3d.nc')
temp = f('temp', time=slice(0, 1), z=slice(0, 1),
        lat=(47.6, 48), lon=(-5.8, -4.8), squeeze=1)
f.close()
temp.long_name = 'Original'

# On crée un trou
temp[20:40, 40:60] = MV2.masked

# On rempli des trous
from actimar.misc.grid.regridding import fill2d
tempf = fill2d(temp, method='nat')
tempf.long_name = 'Rempli'

# Plots
from actimar.misc.plot import map, P, savefigs
map(temp, show=False, subplot=211, left=.11, figsize=(6, 5.5))
map(tempf, show=False, subplot=212, savefigs=__file__)
```

1.2 Outils bathymétriques

Voir : bathy, shoreline

1.2.1 Les traîts de côte

Voir : shorelines

Traît de côte : lecture de shapefile polygones et tracés

Voir : Histolitt map()

```
# Creer une carte se limitant a Ouessant avec fond de mer
from actimar.misc.plot import map
from actimar.misc.color import ocean
m = map(lat=(48.41, 48.49), lon=(-5.15, -5), show=False,
        fillcontinents=False, drawcoastlines=False, figsize=(5.5, 4),
         bgcolor=ocean, left=.12, top=.9)

# Fichier au 1/25000eme avec selection de la zone de la carte
from actimar.bathy.shorelines import Histolitt
coast = Histolitt(m=m) # Chargement
coast.plot() # Trace

# On travail maintenant sur l'ile
# - creation d'un polygone (voir le tutoriel (*@\ref{lst:misc.grid.polygons}@*))
from _geoslib import Polygon
import numpy as N
select = Polygon(N.array([[-5.1, 48.41], [-5, 48.41], \
        [-5, 48.49], [-5.1, 48.49]]))
# - recuperation de l'ile
island = coast.greatest_polygon()
# - sauvegarde de l'ile complete dans un fichier ascii
f = N.savetxt('bathy.shorelines.dat', island.boundary)
# - boucle sur les intersections
```

```
import pylab as P
for poly in island.intersection(select):
        xx, yy = poly.boundary.transpose() # coordonnees
        P.fill(xx, yy, alpha=.5, facecolor='g', linewidth=0) # coloriage

# Fin du trace
from pylab import show, title
from actimar.misc.plot import savefigs
title('Trait de cote SHOM/IGN 1/25000')
savefigs('bathy-shorelines-readplot')
```

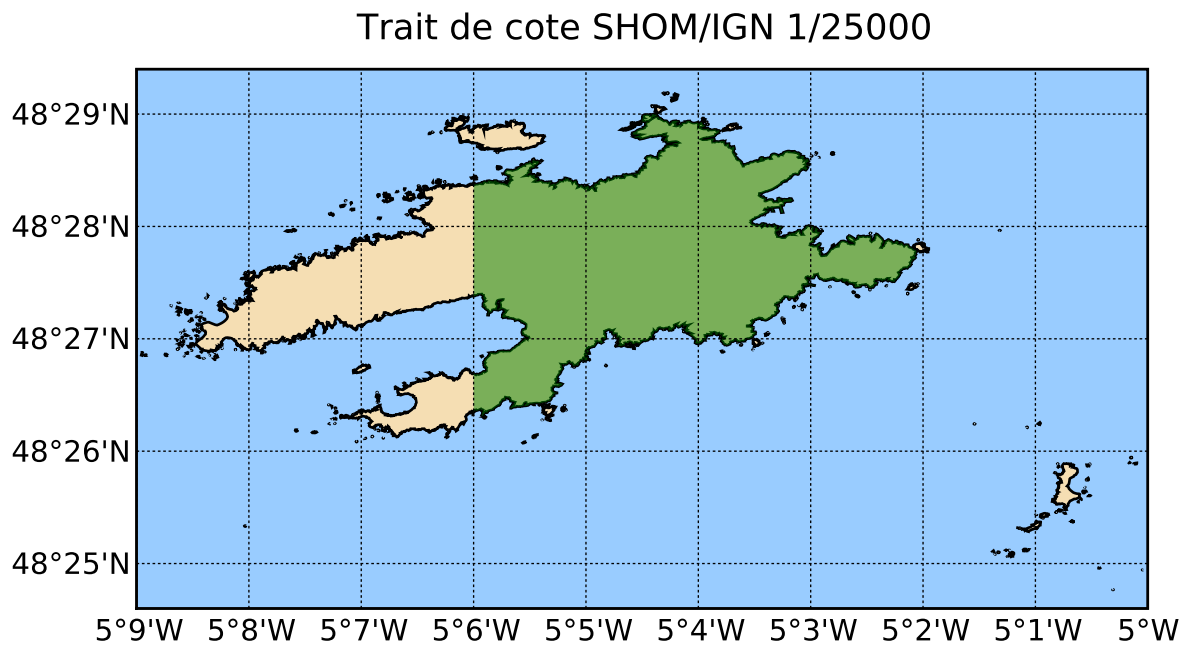


Figure 1.26: Un traît de côte au format shapefile est lu et représenté graphiquement. L'île est récupérée sous forme d'un polygone, puis une partie seulement est coloriée en rouge.

Comparaison de traîts de côte

Voir : GSHHS EUROSION Histolitt

```
# Lecture des divers traits de cote
from actimar.bathy.shorelines import *
zone = (-5.15, 48.42, -5.03, 48.49)
gmt = GSHHS(clip=zone)
euro = EUROSION(clip=zone)
thc = Histolitt(clip=zone)

# Trace
kwpt  = dict(fill=False, points=True, s=3.,alpha=.7, linewidth=0)
thc.plot(m=True,color='r',zorder=12,m_left=.1,
        label='Histolitt (SHOM/IGN)',m_figsize=(5.5, 6), **kwpt)
euro.plot(color='g',zorder=11,label='EUROSION', **kwpt)
gmt.plot(fill=False,color='k',linewidth=1.5, zorder=10,label='GSHHS')

# Fin de plot
from pylab import show, legend, title
from actimar.misc.plot import savefigs
title("Traits d'Ouessant")
```

```
#legend(loc='lower right')
savefigs('bathy-shorelines-compare')
```

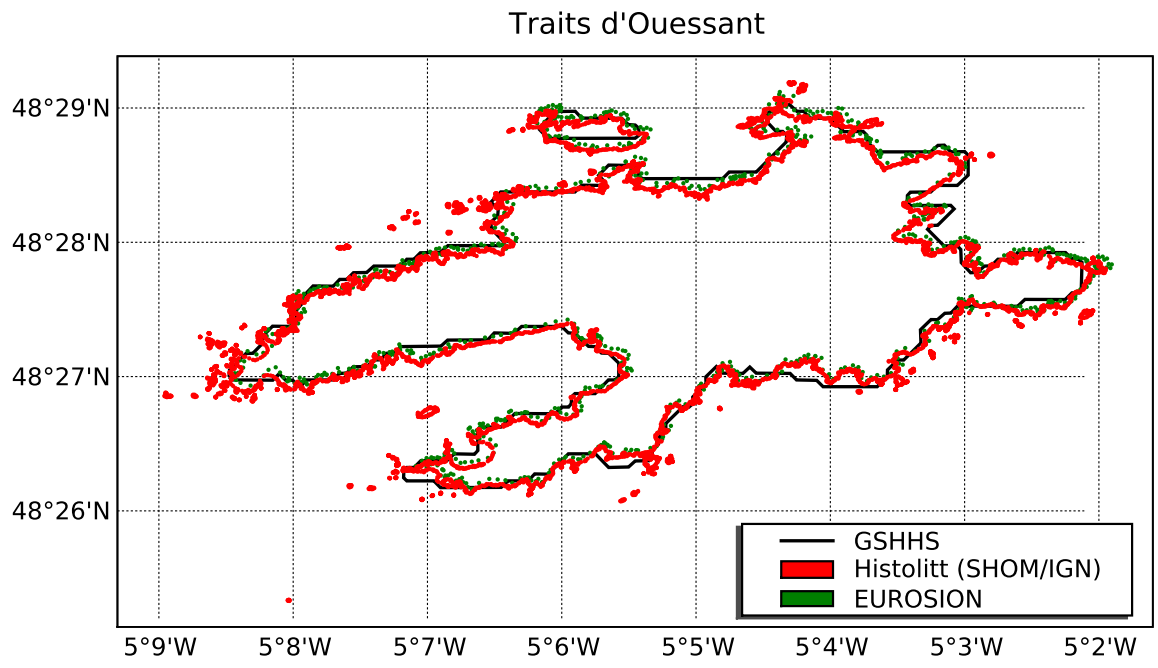


Figure 1.27: Tracé des différents trait de côte.

Niveau moyen sur trait de côte

Voir : `coastal_bathy()`

```
# -*- coding: utf8 -*-
# Definition de la region de travail Bretagne
zone = (-6.5, 47.2, -2, 49.05)

# Recuperation du trait Europeen
from actimar.bathy.shorelines import GSHHS
tc = GSHHS(input='h', clip=zone)

# Interpolation du niveau de la mer sur le trait
xyz = tc.bathy()

# Plot
xyz.plot(size=10, savefigs=__file__, show=True)
```

1.2.2 La bathymétrie

Voir : `bathy`

Bathymétries irrégulières XYZ

Les bases d'une bathy XYZ

Voir : `XYZBathy map()`

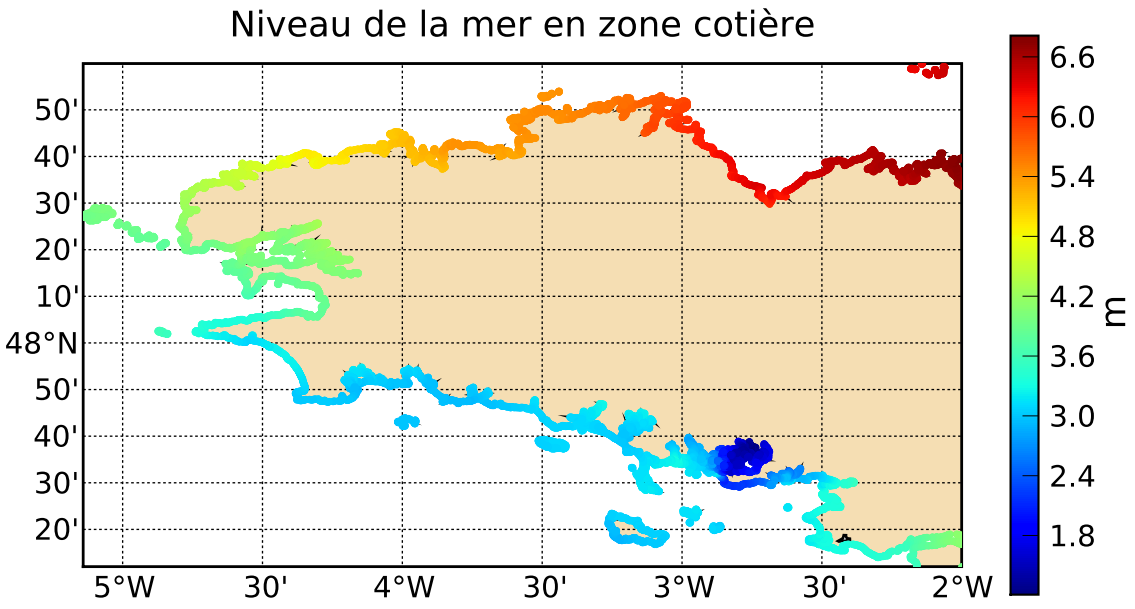


Figure 1.28: Le niveau moyen en différents ports de France est interpolé sur le traît de côte EUROSION.

```
# Modules
import numpy as N,  os
from actimar.bathy.bathy import XYZBathy
import pylab as P
from actimar.misc.grid.basemap import merc
from actimar.misc.grid import create_grid, resol
from actimar.misc.plot import map

# Creation d'une fausse bathymetrie xyz
n = 1000 ; xc = -5.1 ; yc = 48.2 ; xr = .4 ; yr = .15
noise = N.random.random(n)
a = N.random.random(n)*N.pi*2
r = N.random.random(n)
x = xc + r*xr*N.cos(a)
y = yc + r*yr*N.sin(a)
bathy = N.asarray([x, y, N.exp(-(x-xc)**2/xr**2-(y-yc)**2/yr**2)*30.+noise]).transpose()
fbathy = __file__[:-2]+'xyz'
N.savetxt(fbathy, bathy)

# Chargement dans XYZ avec sous echantillonage
xyz = XYZBathy(fbathy,  long_name='My XYZ', rsamp=0.01)
#  on aurait pu charger directement :
#  >>> xyz = XYZBathy(bathy)

# Ajout d'une zone de selection
#  -> triangulaire, par coordonnees [[x1,y1],...]
xyz.select([[-5.4, 48.1], [-4.8, 48.1], [-5.1, 48.5]])

# Ajout d'une zone d'exclusion
#  -> rectangulaire, par coins [xmin,ymin,xmax,ymax]
xyz.exclude([-5.2, 48., -5, 48.25])

# Infos
print xyz
```

```
# Recuperation des valeurs
x = xyz.x()
y = xyz.y()
z = xyz.z()

# Extensions
# - en tenant compte des zones de selection et exclusion
print  'Limites :', xyz.xmin(),  xyz.xmax(), xyz.ymin(),  xyz.ymax()
# - donnees brutes
print  'X min brut :', xyz.xmin(mask=False),   xyz.x(mask=False).min()

# Resolution moyenne
print 'Resolution geographique :', xyz.resol(deg=True)
print 'Resolution metrique :', xyz.resol()

# Definition auto d'une grille reguliere
grid_auto = xyz.grid()
print 'Resolution grille auto :', resol(grid_auto)

# Interpolation sur grille auto
print 'Interpolation auto'
gridded_auto = xyz.togrid()
#  equivalent a :
#  >>> gridded_auto = xyz.togrid(xyz.grid())

# Interpolation sur grille manuelle
print 'Interpolation et masquage manuels puis extraction'
# - defintion de la grille
grid_manual = create_grid((-5.3, -4.91, .01), (48.1, 48.41, .01))
# - interpolation
gridded_manual = xyz.togrid(grid_manual,  mask='i')
# Extraction d'une sous-zone (xmin,ymin...) avec marge
xyz_up = xyz.clip(zone=(None, None, None, 48.3), margin=2)
# - si None, valeurs limites internes (i.e xyz.xmin(), ...)
# - margin : marge relative en unite de resolution
#   -> ici : ymax = 48.3 + xyz.resol()[1]*2

# Sauvegarde
print 'Sauvegarde'
xyz_up.save(__file__[:-2]+'up.xyz')

# Plots
print 'Plots'
# - init
P.figure(figsize=(4.5, 8))
P.rc('font', size=8)
P.subplots_adjust(top=.95, hspace=.25, left=.1, bottom=.05, right=.98)
P.subplot(311)
m = map(lon=(xc-xr, xc+xr), lat=(yc-yr, yc+yr), proj='merc',
        autoresize=0, resolution='f', show=False, drawmeridians_rotation=45,
        ticklabel_size=9, xhide=True)
kwplot = dict(vmin=xyz.zmin(False), vmax=xyz.zmax(False),
        m=m, show=False, colorbar=False)
# - xyz
xyz.plot(size=10, mode='both', masked_alpha=.1, **kwplot)
# - interpole manuellement
kwplot.update(autoresize=0, drawmeridians_rotation=45, ticklabel_size=9)
P.subplot(312)
map(gridded_manual, xhide=True, title='Sur grille manuelle', **kwplot)
# - interpole auto
P.subplot(313)
map(gridded_auto, title='Sur grille auto', savefigs=__file__, **kwplot)
```

```
P.show()
```

Fusion de bathymétries XYZ

Voir : XYZBathy map()

```
# Creation de fausses bathymetries xyz
import numpy as N,  os
from actimar.bathy.bathy import XYZBathy,  XYZBathyMerger
# - fonction generatrice
def gene_bathy(xc, yc, xr, yr, n=500, amp=30.):
        noise = N.random.random(n)
        a = N.random.random(n)*N.pi*2
        r = N.random.random(n)
        x = xc + xr*r*N.cos(a)
        y = yc + yr*r*N.sin(a)
        return N.asarray([x, y, N.exp(-(x-xc)**2/xr**2-(y-yc)**2/yr**2)*amp+noise])
# - creations
xyz1 = XYZBathy(gene_bathy(-5, 48.3, .3, .15)) # top right
xyz2 = XYZBathy(gene_bathy(-5.45, 48.3, .2, .1,  amp=10)) # top left
xyz3 = XYZBathy(gene_bathy(-5.45, 48.1, .45, .25, n=1000, amp=20), transp=False) # bot left
xyz4 = XYZBathy(gene_bathy(-5., 48.1, .14, .08, n=300)) # bot right
fxyz5 = __file__[:-2]+'xyz5.xyz'
N.savetxt(fxyz5, gene_bathy(-5.15, 48.2, .2, .1, amp=15).transpose()) # center

# Fusion directe
xyz = xyz1 + xyz2 + xyz3 + xyz4 + fxyz5

# Plot
import pylab as P ; P.figure(figsize=(5., 8.5)) ; P.subplot(311)
P.rcParams['font.size'] = 9
P.subplots_adjust(bottom=.03, top=.97, hspace=.25)
kwplot = dict(show=False, colorbar=False,map_res=None, margin=0., map_autoresize=0,
        xmin=xyz.xmin(), xmax=xyz.xmax(), ymin=xyz.ymin(), ymax=xyz.ymax())
xyz.plot(title='Fusion directe', **kwplot)

# Utilisation d'un fusionneur
# - init
merger = XYZBathyMerger()
# - ajout de xyz direct
merger += xyz1
merger.append(xyz2)
merger += xyz3
merger += xyz4
# - ajout a partir d'un fichier
merger += fxyz5
# - suppression d'un xyz
merger -= xyz2
# - suppression du dernier dataset
del merger[-1]
# - changement d'autres attributs
for i, xyz in enumerate(merger):
        xyz.long_name = 'Niveau : %i'%i # Pour les legendes
        xyz.set_transp(False) # Opacite

# Plot du fusionneur
# - valeurs
P.subplot(312)
merger.plot(mode='value', title='Merger : valeurs', **kwplot)
# - cluster
```

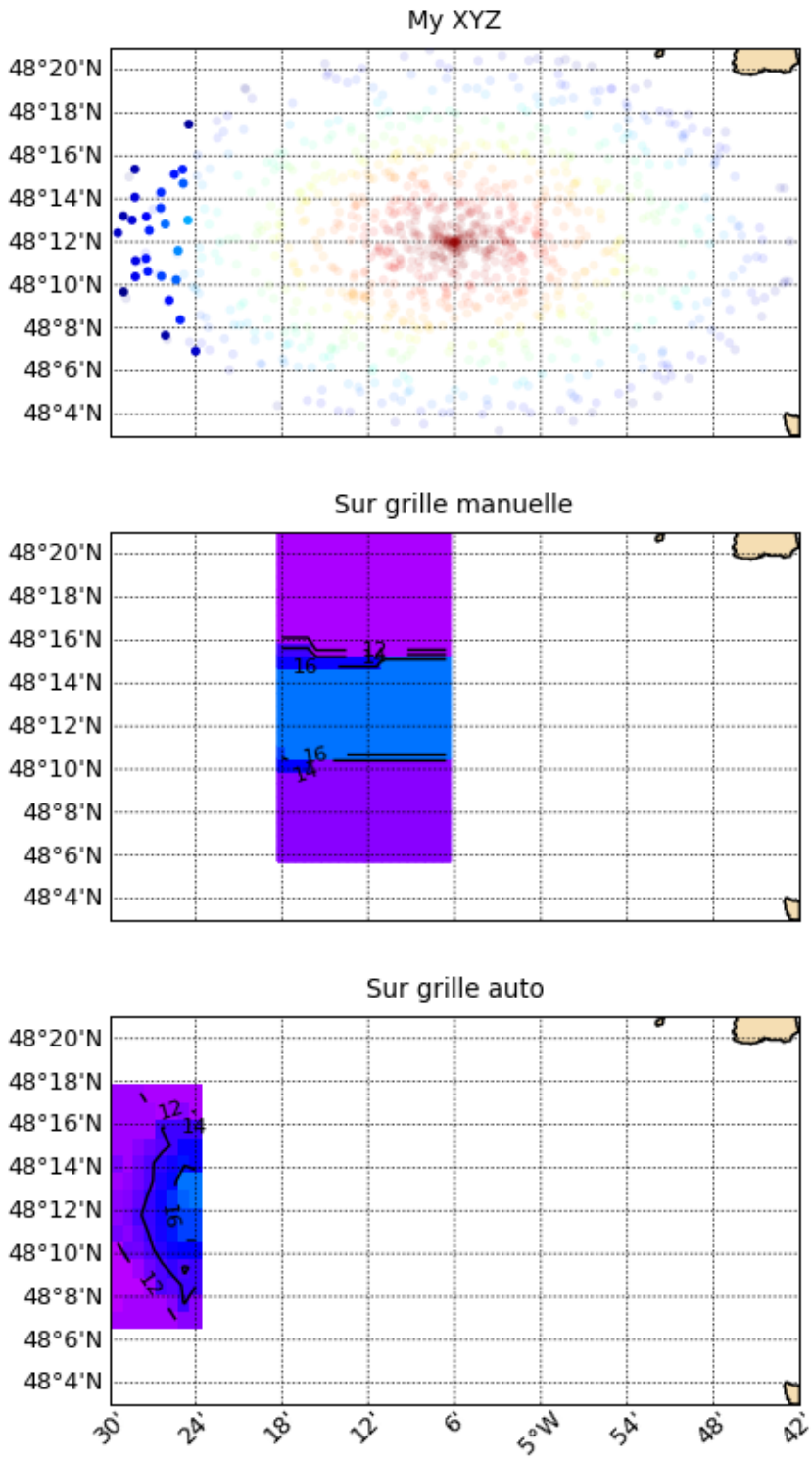


Figure 1.29: En haut, points bathymétriques éparses ayant subit un triangle de selection et un rectangle d'exclusion. Au milieu, les points sont interpolé/extrapolés sur une grille définie manuellement. En bas, de même mais la grille est entièrement définie automatiquement (extensions et résolution).

```
P.subplot(313)
merger.plot(mode='cluster', size=10, title='Merger : cluster', marker='o',
        legend_loc='upper left', savefigs=__file__, **kwplot)

# On recupere le tout dans un xyz
merged_xyz = merger.xyz(long_name='Merged')

# On peut donc le sauvegarder
merged_xyz.save(__file__[:-2]+'merged.xyz')
```

Ajout d'un traît de côte à une bathy XYZ

À venir...

Gestion d'une banque de bathymétries XYZ

Note: Dans l'exemple ci-dessous, le fichier de recensement des bathymétries est spécifié manuellement. Vous n'aurez normalement pas à le spécifier car il en existe un par défaut.

Voir : XYZBathyBank (et éventuellement XYZBathyBankClient et XYZBathyMixer).

```
# Creation de fausses bathymetries xyz
import numpy as N,  os
from actimar.bathy.bathy import XYZBathy,  XYZBathyBank
# - fonction generatrice
def gene_bathy(xc, yc, r, n):
        noise = N.random.random(n)
        a = N.random.random(n)*N.pi*2
        r = r*N.random.random(n)
        x = xc + r*N.cos(a)
        y = yc + r*N.sin(a)
        return N.asarray([x, y, N.exp(-(x-xc)**2/r**2-(y-yc)**2/r**2)*30.+noise]).transpose()
# - bathy sud
fsouth = __file__[:-2]+'bathy_south.xyz'
N.savetxt(fsouth, gene_bathy(-5.1, 48.1, .15, 200.))
# - bathy nord
fnorth = __file__[:-2]+'bathy_north.xyz'
N.savetxt(fnorth, gene_bathy(-5.1, 48.3, .15, 100.))
# - bathy large
flarge = __file__[:-2]+'bathy_large.xyz'
N.savetxt(flarge, gene_bathy(-5.2, 48., .4, 300.))

# On stocke dans une banque
# - from scratch
bank_file = __file__[:-2]+'bank.cfg'
if os.path.exists(bank_file): os.remove(bank_file)
# - init de la banque
bank = XYZBathyBank(bank_file)
# - ajout des fichiers bathy
bank.add(fsouth, id='south', long_name='South')
bank.add(fnorth, id='north', long_name='North')
bank.add(flarge) # id auto
#   you can also use: bank += flarge
# - on verifie ce qu'on a
print bank


# Petites modifs
# - id
```

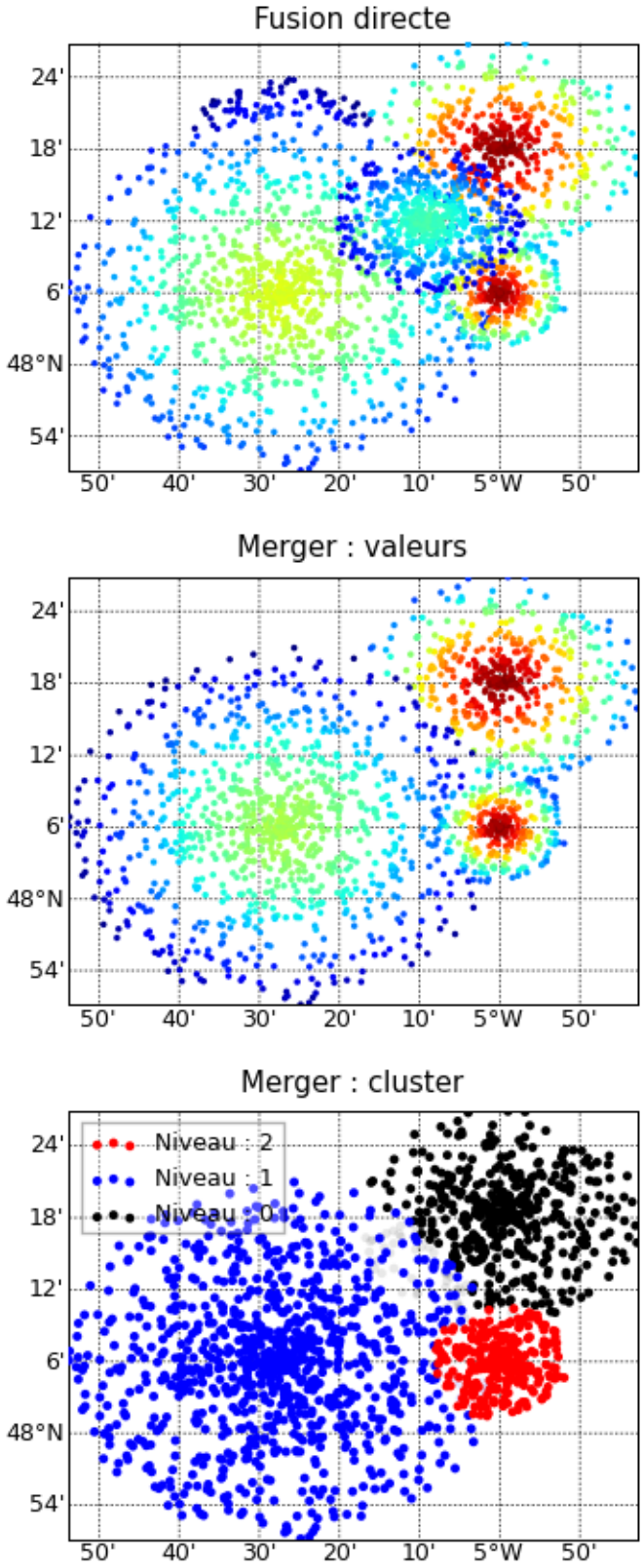


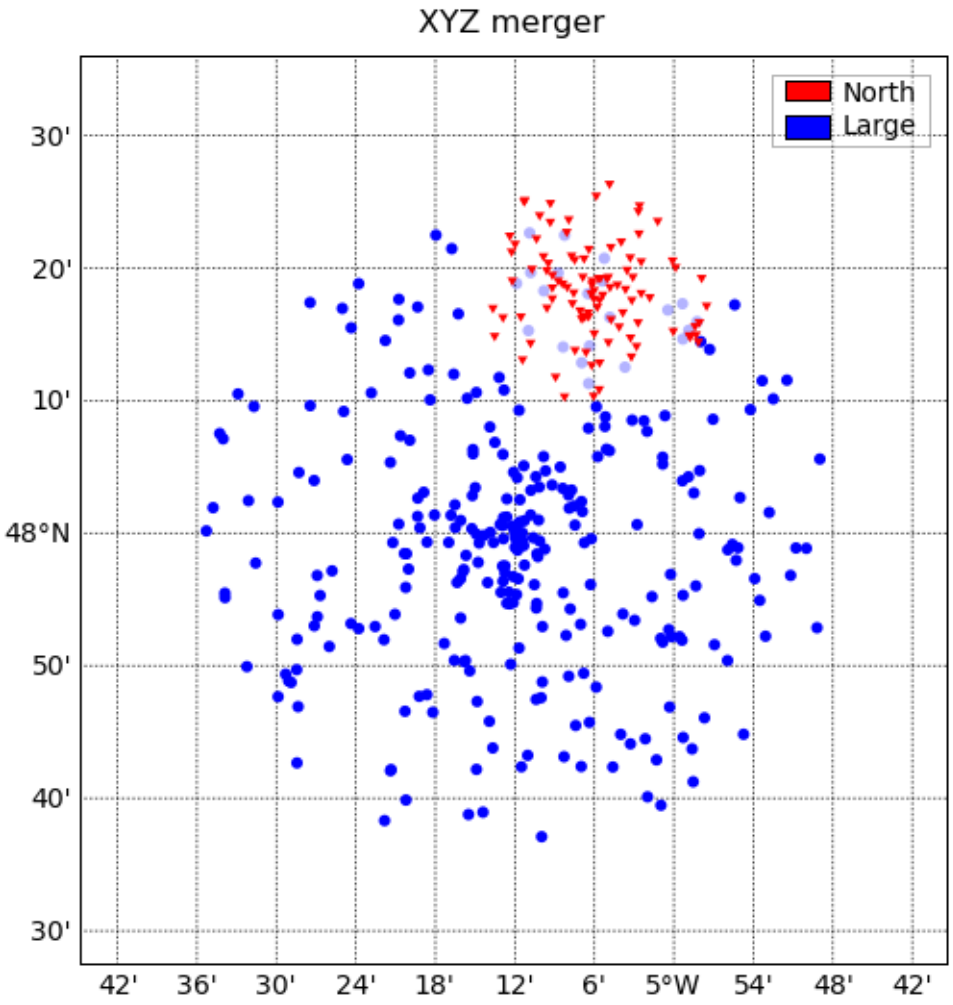
Figure 1.30: En haut, bathymétries diverses fusionnées directement par addition, avec celle en bas à gauche opaque. Au centre, trois bathymétries opaques distinguée dans le fusionneur, avec les points masquées partiellement transparents. En bas, de même mais avec les valeurs.

```
bank['bathy.bathy.xyz.bathybank.bathy_large'].id = 'large'
# - transparence
bank['north'].transp = False
# - long name
bank['large'].long_name = 'Large'

# Suppression d'une bathy
bank.remove('south')
#  on peut aussi utiliser:
#  >>> bank -= 'south'
#  >>> bank -= bsouth
#  >>> del bank['south']

# Chargement d'une bathy
bsouth = bank['north'].load()

# Plot de l'ensemble
bank.plot(size=15,  map_proj='cyl', map_res=None, savefigs=__file__,show=False)
```



Bathymétries grillées XYZ

Les bases d'une bathy grillée

Voir : GriddedBathy plot_bathy() GSHHS map() regrid2d() polygon_mask()

```
# -*- coding: utf8 -*-
# Récupération d'une bathy sous la forme d'une variable cdat
import cdms2
f = cdms2.open('/home2/amzer/pineau/BATHY/smith_sandwell_topo_v8_2.nc')
var = f('ROSE', lon=(-6.1, -3), lat=(47.8, 48.8))
f.close()

# Création de l'objet de bathy grillee
from actimar.bathy.bathy import GriddedBathy
bathy = GriddedBathy(var)

# On définit le traît de côte pour le masquage
bathy.set_shoreline('f')

# On récupère une version masquée
bathy_orig = bathy.bathy(mask=False)
bathy_masked = bathy.bathy(mask=True)


# Definition d'une grille zoom
from actimar.misc.grid import create_grid
new_grid = create_grid((-5.5, -4.6, 0.009), (48.25, 48.6, .006))

# On interpole la bathy masquée sur une autre grille
interp_bathy_masked = bathy.regrid(new_grid)

# On interpole et masque la bathy originale sur une autre grille
# - interpolation
interp_bathy_orig = bathy.regrid(new_grid, mask=False)
# - masquage
interp_bathy_orig_masked = GriddedBathy(interp_bathy_orig, shoreline='f').bathy()

# Plots
from matplotlib import rc ; rc('font', size=9);rc('axes', titlesize=9)
kwplot = dict(resolution=None, show=False, colorbar=False, contour=False,
        top=.97, hspace=.25, bottom=.03, right=.98,
        autoresize=False, vmax=bathy_orig.max(), vmin=bathy_orig.min())
# - colormap
from actimar.misc.color import auto_cmap_topo, land
kwplot['cmap'] = auto_cmap_topo(bathy_orig)
kwplot['bgcolor'] = land
# - directs
bathy.plot(title='Original', mask=False, figsize=(4.5, 8), subplot=411, key=1, **kwplot)
bathy.plot(title='Masked', mask=True, subplot=412, key=2, **kwplot)
# - indirect (bathy regrillées)
from actimar.bathy.bathy import plot_bathy
plot_bathy(interp_bathy_masked, title='Masked->Interpolated', subplot=413, key=3, **kwplot)
plot_bathy(interp_bathy_orig_masked, title='Interpolated->Masked', subplot=414, key=4, **kwplot)
# - sauvegarde
from actimar.misc.plot import savefigs
savefigs(__file__)
```

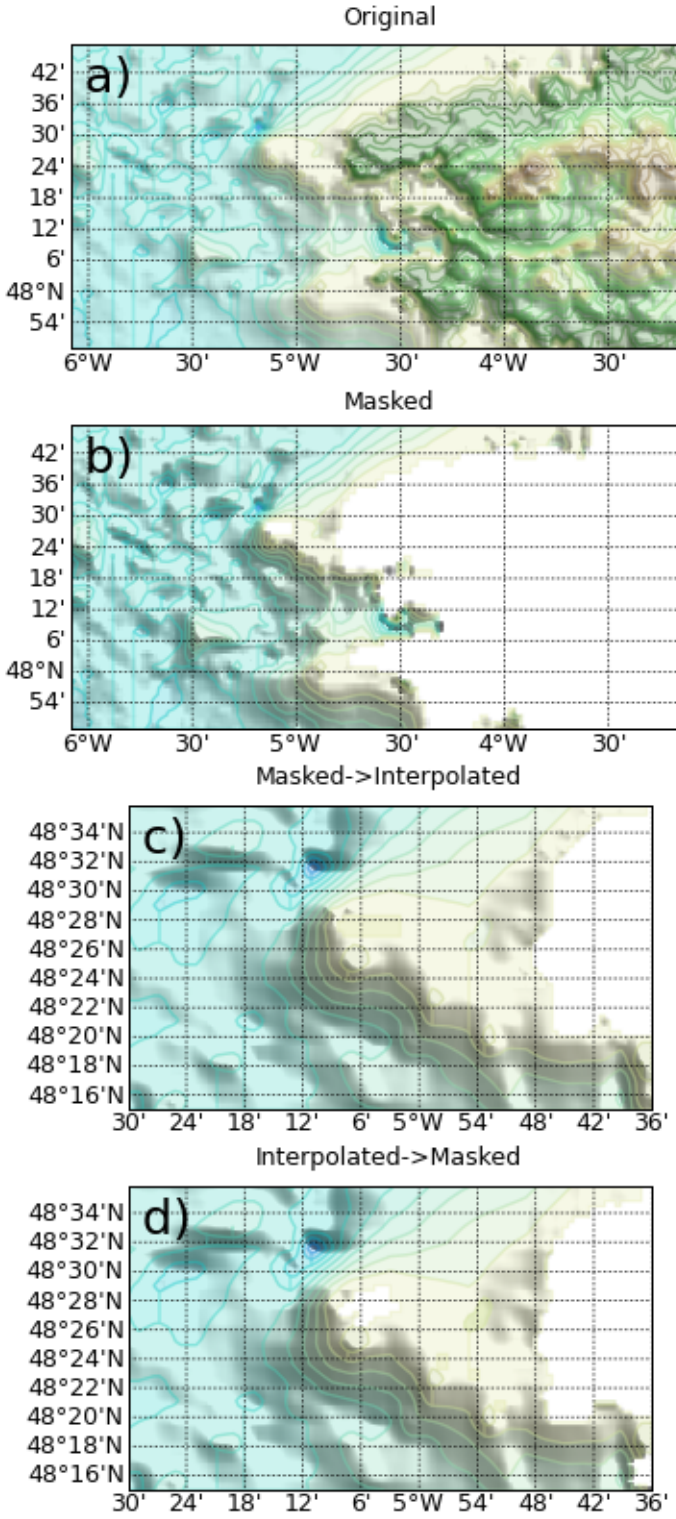


Figure 1.31: En **a)**, topographie original ; en **b)**, la même topographie, mais masquée sur la terre grâce au trait de côte GSHHS fin ; en **c)**, topographie masquée puis interpolée sur une grille plus fine ; en **d)**, la topographie non masquée est d'abord interpolée, puis elle est masquée à l'aide du traît de côte. Cette dernière méthode évite d'interpoler un masque grossier et est donc préférable.

Fusion de bathymétries grillées

Voir : `GriddedBathy` `GriddedMerger` `add_grid()`

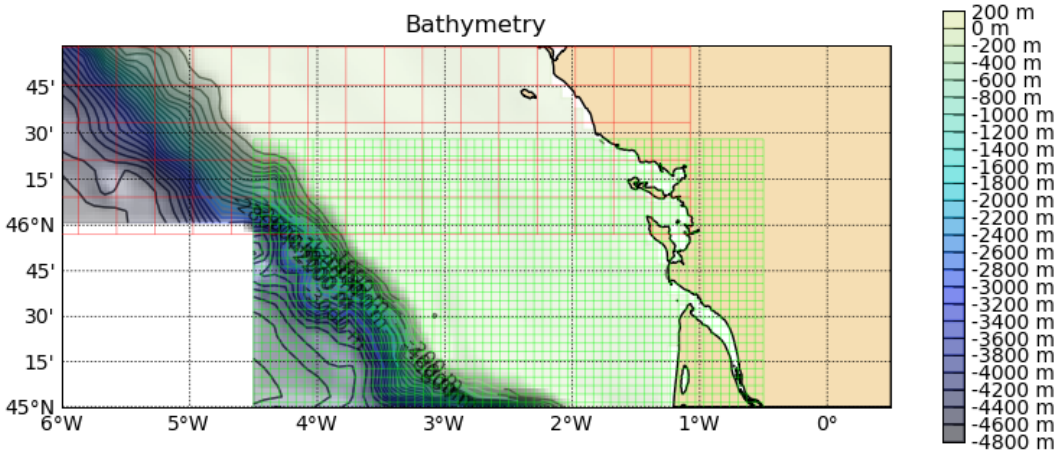


Figure 1.32: Bathmétrie grillée issue de la fusion de deux autres bathymétries à résolutions et emprises différentes.

```
# -*- coding: utf8 -*-
# Création de bathymétries fictives à partir de Smith and Sandwell
import cdms2
f = cdms2.open('/home2/amzer/pineau/BATHY/smith_sandwell_topo_v8_2.nc')
# - large
var_large = f('ROSE', lon=(-7, -1), lat=(46, 49))
# - petite
var_small = f('ROSE', lon=(-4.5, -.5), lat=(44.5, 46.5))
f.close()
# - regrillage de la large vers une grille moins fine
from actimar.misc.grid import regridding, resol, create_grid
xr, yr = resol(var_large.getGrid())
grid_large = create_grid((-7., -1, xr*4.5), (46., 49, yr*4.5))
var_large = regridding.regrid2d(var_large, grid_large)

# On ajoute un traît de côte pour le masquage
from actimar.bathy.bathy import GriddedBathy, GriddedBathyMerger
bathy_large = GriddedBathy(var_large, shoreline='i')
bathy_small = var_small

# Création de la grille finale de résolution intermédiaire
final_grid = create_grid((-6., .5, xr*2.5), (45, 47., yr*2.5))

# On crée maintenant le merger
merger = GriddedBathyMerger(final_grid)
# - ajout de la bathy basse resolution en premier (en dessous)
merger += bathy_large
# - puis ajout de celle haute résolution
merger += bathy_small

# On définit le traît de côte pour le masquage
merger.set_shoreline('h')

# Fusion vers la grille finale
bathy = merger.merge()
```

```
# Plot
merger.plot(show=False)
from actimar.misc.plot import savefigs, add_grid
kwgrid = dict(linewidth=.5, alpha=.5, samp=2)
add_grid(grid_large, color='r', **kwgrid)
add_grid(var_small.getGrid(), color='#00ff00', **kwgrid)
savefigs(__file__)
```

1.3 Outils liés à la marée

1.3.1 Infos sur les stations marégraphiques

Informations de base

Voir : StationInfo.

```
# -*- coding: utf8 -*-
# On cherche directement 'Bre' pour Brest
from actimar.tide import StationInfo
station = StationInfo('Bre')
# ->:
#Chargement de la station suivante :
#  Nom        : Ile de Brehat (Port-Clos)
#  Position   :  3.0°W / 48.9°N
#  Zone       : http://www.shom.fr/fr_page/fr_act_oceano/maree/zone6_9.htm
#  BM45       : 3.8
#  BM95       : 1.35
#  NM         : 5.89
#  PBM        : 0.1
#  PHM        : 11.6
#  PM45       : 8
#  PM95       : 10.4
#  ZERO_HYDRO : -5.4
#Definition des termes accessible avec .definitions()
# Loupé !
# On récupère la première station trouvé lors de l'initialisation
# Mais bon, on vérifie quand même
print station.attributes()
# -> ['igs', 'psmsl', 'uhslc', 'gloss', 'shom', 'legos', 'latitude',
#       'longitude', 'zone', 'phm', 'pm95', 'pm45', 'nm', 'bm45',
#        'bm95', 'pbm', 'zero_hydro']
print station.name, station.longitude
# -> Ile de Brehat (Port-Clos) -3.0

# On peut se servir de station pour continuer à chercher
# car le fichier est déjà chargé

# On affiche finalement toute les stations contenant 'bre'
print '-'*70
station.search('bre')
print '-'*70
# ->
#   Nom        : Ile de Brehat (Port-Clos)
#...
#   Nom        : Les Heaux-de-Brehat
 #...
#   Nom        : Brest

# Ok, on récupère Brest et uniquement Brest
```

```
# - en sélectionnant la station d'identifiant SHOM='Brest'
brest = station.find(shom='Brest', verbose=False)
print brest.name
# -> Brest
# - ou par sa position (station la plus proche)
brest = station.find(position=(4.5,48.4), verbose=False)
print brest.longitude, brest.latitude
# -> 2.36777777778 51.0347222222
```

... tide.station_info.mean_sea_level

1.3.2 Les filtres

Filtrage des surcotes/decotes

Voir : `demerliac()`, `godin()` et `generic()`.

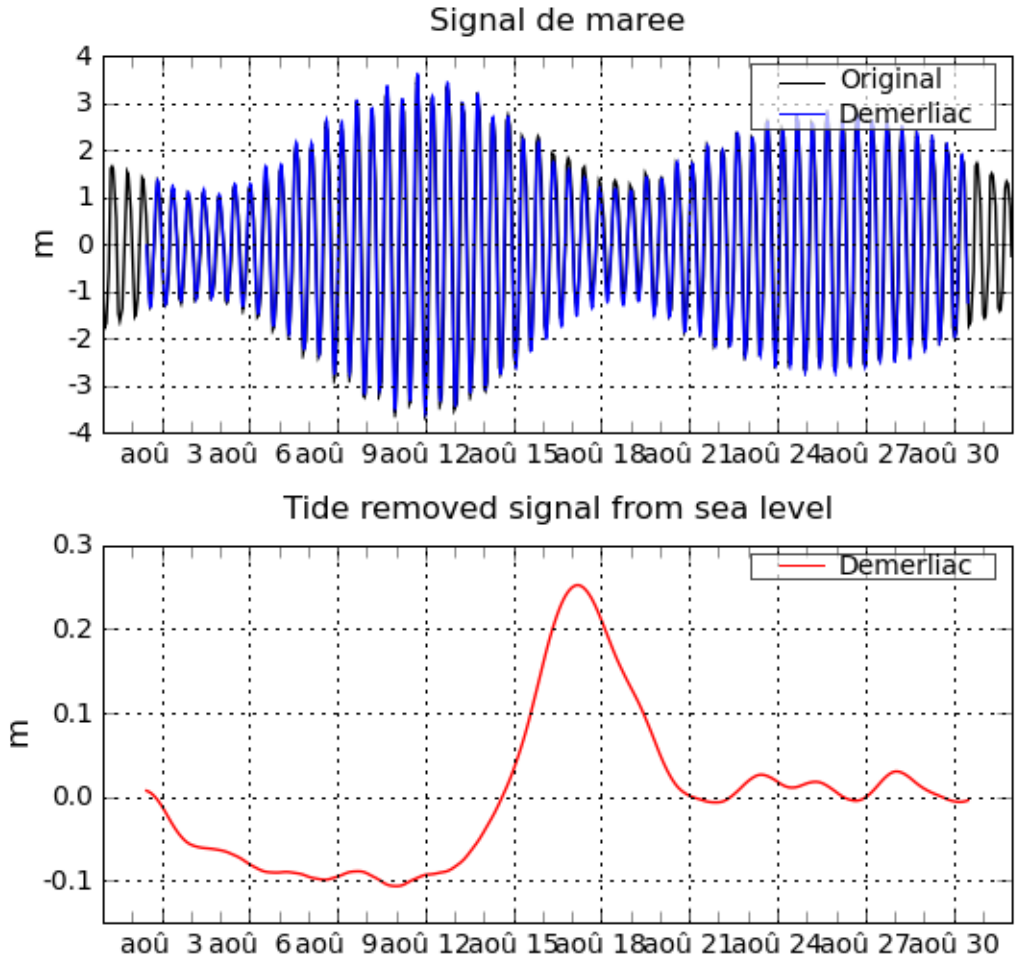


Figure 1.33: Filtre de Démerliac appliqué à un signal de marée.

```
# -*- coding: utf8 -*-
# Lecture d'une serie 1D de niveau de la mer
from actimar.tide.sonel_mareg import get_slv_shom
```

```
import MV2
sea_level = get_slv_shom('BREST', time_range=('2006-08', '2006-09', 'co'))
sea_level[:] -= MV2.average(sea_level)

# Filtrages
from actimar.tide.filters import demerliac, godin
cotes, tide = demerliac(sea_level, get_tide=True)

# Plots
from actimar.misc.plot import curve, savefigs
import pylab as P
kwplot = dict(date_fmt='%d/%m', show=False, date_rotation=0)
# - signal de maree
curve(sea_level, 'k', subplot=211, label='Original', **kwplot)
curve(tide, 'b', label='Demerliac', title='Signal de maree', **kwplot)
P.legend().legendPatch.set_alpha(.7)
# - surcotes/decotes
curve(cotes, 'r', subplot=212, hspace=.3, label='Demerliac', **kwplot)
P.legend().legendPatch.set_alpha(.7)
savefigs(__file__, savefigs_pdf=True)
```

Calcul des pleines mers et basses mers

Voir : `extrema()`, `filters`.

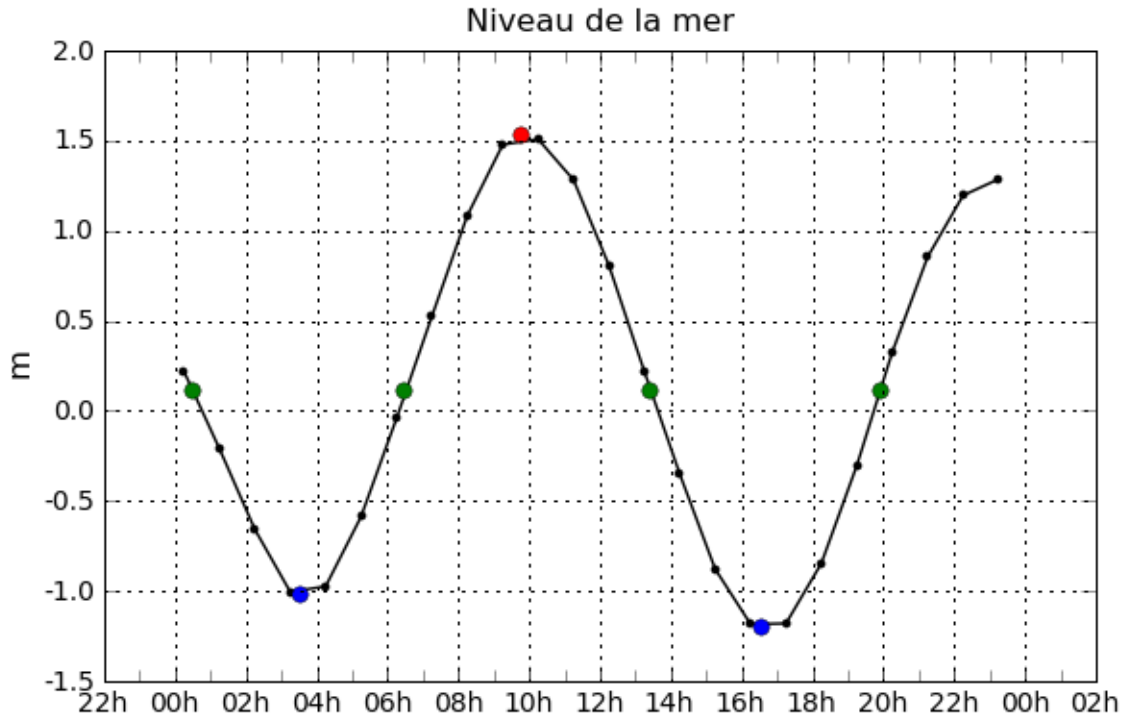


Figure 1.34: Estimation des pleines mers et basses mers par splines sur une série 1D de niveau de la mer.

```
# -*- coding: utf8 -*-
# Lecture d'une serie 1D de niveau de la mer du modele
import cdms2
f = cdms2.open('/home2/amzer/raynaud/misc/samples/tide.sealevel.BREST.mars.nc')
sea_level = f('sea_level', ('2006-10-01', '2006-10-02'))[1::4] # Toutes les heures
```

```
f.close()

# Recuperation des pleines mers, basses mers et zeros
from actimar.tide.filters import extrema, zeros
bm, pm = extrema(sea_level, spline=True, reference='mean')
zz = zeros(sea_level, reference='mean')

# Plots
from actimar.misc.plot import curve
curve(sea_level, 'ko', markersize=3, figsize=(6, 4), show=False)
curve(zz, 'go', linewidth=0, show=False, xstrict=False)
curve(pm, 'ro', linewidth=0, show=False, xstrict=False)
curve(bm, 'bo', linewidth=0, xstrict=False, title="Niveau de la mer",
        savefigs=__file__, savefigs_pdf=True)
```

Outil marégraphique “tout en un”

Voir : `Marigraph2`, `demerliac`, `godin()` et `generic()`.

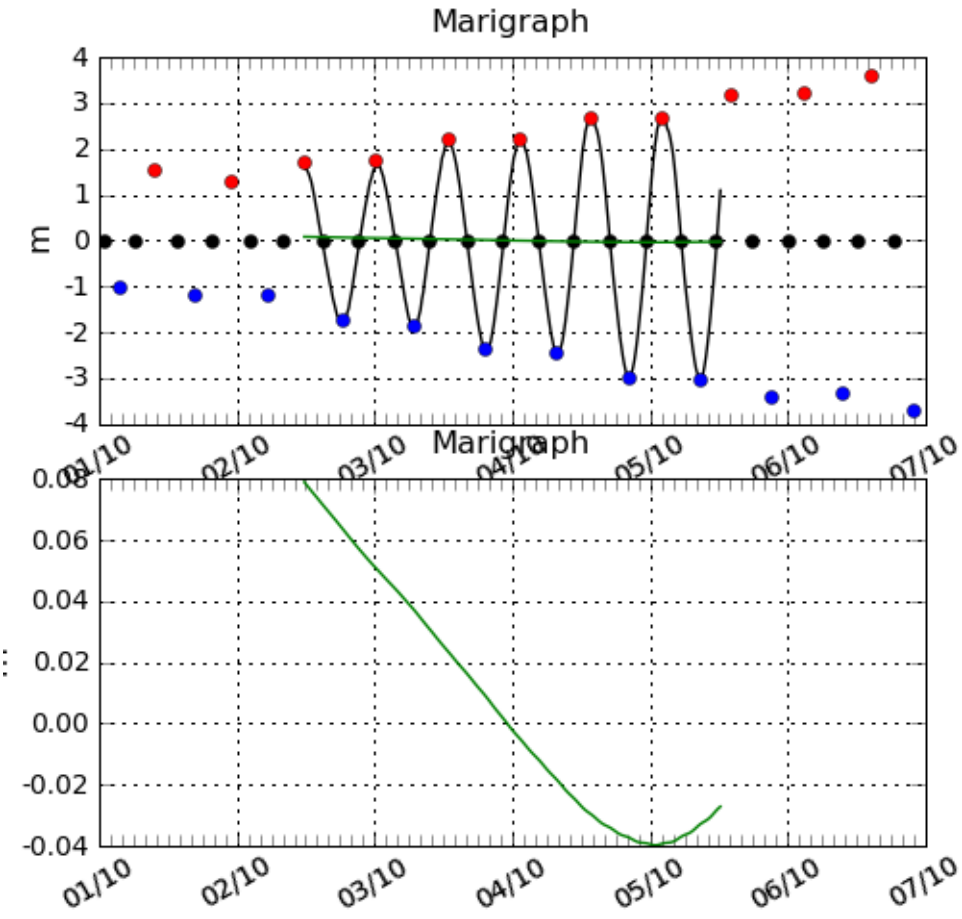


Figure 1.35: Outil de filtrage marégraphique.

```
# -*- coding: utf8 -*-
# Lecture du niveau de la mer
import cdms2
f = cdms2.open('/home2/amzer/raynaud/misc/samples/tide.sealevel.BREST.mars.nc')
```

```
sea_level = f('sea_level', time=('2006-10', '2006-10-07'))
f.close()

# Initialisation de l'objet marégraphique
from actimar.tide import Marigraph2
mg = Marigraph2(sea_level, verbose=True)

# Extraction du signal de maree
tide = mg.tide(tide_filter='demerliac')

# On peut aussi specifier le filtre avec :
mg.set_tide_filter('demerliac') # On peut aussi choisir 'godin'

# Calcul surcotes/decotes
cotes = mg.cotes()

# Calcul des pleines et basses mers
ref = 'mean'
highs = mg.highs(ref=ref)
lows = mg.lows(ref=ref)
zeros = mg.zeros(ref=ref)

# On plot
import pylab as P
P.figure(figsize=(5, 5))
kwplot = dict(date_fmt='%d/%m', date_locator='day', show=False, )
# - tout sauf le signal d'origine
P.subplot(211)
mg.plot(orig=False, tide_color='k', **kwplot)
# - seules les surcotes decotes
P.subplot(212)
mg.plot('cotes', savefigs=__file__, savefigs_pdf=True, **kwplot)
```

CHAPTER

TWO

DÉTAIL DES MODULES

2.1 actimar.misc — Outils génériques

2.1.1 actimar.misc.misc– Divers

Misc tools

Note: You can import it directly for example like this:

```
>>> from actimar.misc import auto_scale
```

ismasked(*arr*)

bound_ops(*bounds*)

Get operators that must be used for checking inclusion within boundaries. Returned operators (ops) must be used in the following way to return True if value is inside bounds:

```
>>> ops[0](value,lower_bound)
>>> ops[1](value,upper_bound)
```

- **bounds**: Boundary closing indicator (like 'co')

Return: A 2-element tuple of operators taken within (operator.ge,operator.gt,operator.le,operator.lt).

auto_scale(*data=None, nmax=None, vmin=None, vmax=None, separators=None, fractions=False, symetric=False, **kwargs*)

Computes levels according to a dataset and its range of values. Locators are on a 10-base. Different scaling can be used with this version.

- **data**: The dataset
- *vmax*: Replaces max(data)
- *vmin*: // min(data)
- *symetric*: min_value and max_value are made symetric with respect to zero [default: False]
- *nmax maximal*: Number of levels
- *separators*: Subdivision for irregular levels
- *fractions*: Consider separators as fractions (]0.,1.[)
- *steps*: Base 10 steps for finding levels [default: [1,2,2.5,5,10]]
- *geo*: Treat levels as geographical degrees [default: False]
- Other parameters are given to MaxNLocator

Return: List of levels

See Also:

basic_auto_scale() geo_scale()

basic_auto_scale(*vmin, vmax, nmax=7, steps=, [1, 2, 2.5, 5, 10], geo=False, minutes=False, **kwargs*)
Computes levels according to a dataset and its range of values. Locators are on a 10-base.

- **vmin/vmax**: Find levels around this range.
- *nmax*: Maximal number of steps.
- *steps*: Base 10 steps for finding levels [default: [1,2,2.5,5,10]]
- *geo*: Assume longitude or latitude degrees [default: False].
- *minutes*: If geo, find suitable levels to match nice minutes locations when locations have floating values (like 1.2) [default: False].

See Also:

auto_scale() geo_scale()

geo_scale(**args, **kwargs*)
auto_scale() with geo=True

See Also:

basic_auto_scale() auto_scale()

get_atts(*var, id=True, **kwargs*)
Get all attributes from a variable

- **var**: Target variable
- *id*: Get also id [default: False]
- Other keywords are set as attributes

See Also:

cp_atts() set_atts() check_def_atts()

cp_atts(*var1, var2, **kwargs*)
Copy all atributes from one variable to another

- **var1/var2**: Copy var1 attributes to var2
- *id*: Also copy id [default: False]
- Other keywords are set as attributes

See Also:

get_atts() set_atts() check_def_atts()

set_atts(*var, atts=None, **kwargs*)
Set attributes

- **var**: Change attributes of var
- *atts*: A dictionary of attributes
- Other keywords are set as attributes

See Also:

cp_atts() get_atts() check_def_atts()

check_def_atts(*obj, **defaults*)
Check defaults attributes and set them if empty

See Also:

get_atts() cp_atts() set_atts()

iterable(*obj*)
Check if an object is iterable or not.

- **obj**: Object to check

Return: True/False

isnumber(*var*)

rm_html_tags(*str*)
Remove html tags from a string

- **str**: A string

Return: Cleaned string

Example:

```
>>> rm_html_tags('<title>My title</title>')
My title
```

deg2str(**args, **kwargs*)

lonlab(*longitudes, **kwargs*)
Return nice longitude labels

- **longitudes**: Value of longitudes
- *decimal*: Use decimal representation [default: True]
- *fmt*: Format for decimal representation [default: '%4.1f']
- *no_seconds*: Do not add seconds to degrees representation [default: False]
- *tex*: Use Tex degree symbol [default: False]
- *no_symbol*: Don't use degree symbol [default: False]
- *no_zeros*: Don't insert zeros when integer < 10 [default: False]
- *auto*: If True, find the ticks according to the range of values [default: False]
- *auto_minutes*: Automatically suppress degrees if value is not exactly a degree (just display minutes and seconds) else display degree [default: False]

latlab(*latitudes, **kwargs*)
Return nice latitude labels

- **latitudes**: Value of latitudes
- *decimal*: Use decimal representation [default: True]
- *fmt*: Format for decimal representation [default: '%4.1f']
- *no_seconds*: Do not add seconds to degrees representation [default: False]
- *tex*: Use Tex degree symbol [default: False]
- *no_symbol*: Don't use degree symbol [default: False]
- *no_zeros*: Don't insert zeros when integer < 10 [default: False]
- *auto*: If True, find the ticks according to the range of values [default: False]
- *auto_minutes*: Automatically suppress degrees if value is not exactly a degree (just display minutes and seconds) else display degree [default: False]

deplab(*depths, fmt='%gm', auto=False*)
Return well formatted depth labels

- **depths**: Numerical depths
- *fmt*: Numeric format of the string (including units) [default: '%gm']
- *auto*: If True, find the ticks according to the range of depths [default: False]

deg_from_dec(*dec*)
Convert from decimal degrees to degrees,minutes,seconds

kwfilter(**args, **kwargs*)
Alias for `dict_filter()`

dict_filter(*kwargs, filters, defaults=None, copy=False, short=False, keep=False, **kwadd*)
Filter out kwargs (typically extra calling keywords)

- **kwargs**: Dictionnary to filter.
- **filters**: Single or list of prefixes.
- *defaults*: dictionnary of default values for output fictionnary.
- *copy*: Simply copy items, do not remove them from kwargs.
- *short*: Allow prefixes to not end with "_".
- *keep*: Keep prefix filter in output keys.

Example

```
>>> kwargs = {'basemap':'f', 'basemap_fillcontinents':True, 'quiet':False,'basemap_plot':False}
>>> print kwfilter(kwargs,'basemap', defaults=dict(drawcoastlines=True,plot=True),good=True)
{'plot': False, 'fillcontinents': True, 'good': True, 'basemap': 'f', 'drawcoastlines': True}
>>> print kwargs
{'quiet': False}
```

dict_aliases(*kwargs, aliases*)
Remove duplicate entries in a dictionnary according to a list of aliases. The first aliases has priority over the last ones.

mask_nan(*input*)
Mask NaN from a variable

- **input**: Array (numpy, numpy.ma or MV2)

Note: If input is pure numpy, it is converted to numpy.ma

Example:

```
>>> import numpy as N
>>> from actimar.misc import mask_nan
>>> var = N.array([1,N.nan])
>>> print mask_nan(var)
[1.0 --]
```

write_ascii_time1d(*var, file, fmt='%g'*)
Write an ascii file in the following format: YYY/MM/DD HH:MN:SS DATA where DATA is one column.

- **var**: a cdms variable WITH A TIME AXIS
- **file**: output file name
- *fmt*: format of DATA [default: '%g']

xls_style(*style=None, b=None, i=None, u=None, c=None, o=None, bd=None, fmt=None, va=None, ha=None, f=None, s=None, n=None, **kwargs*)
Excel style sheet for pyExcelerator cell objects

- *style*: Style object
- *b*: Bold [True/False]
- *i*: italic[True/False]
- *u*: Underline [True/False]
- *c*: Colour index [from 0 to 82!, with 0=black, 1=white, r=red, 3=green, 4=blue, 5=yellow, 6=magenta, 7=cyan]
- *o*: Outline [True/False]
- *bd*: Borders [True/int/False/dict(top=int...)]
- *fmt*: Format ['general','0.0',...]
- *va*: Vertical alignment ['top','bottom','center','justified']
- *ha*: Horizontal alignment ['left','right','center','justified']

class `FileTree`(*input_dir, relative=False, scan=True, **kwargs*)

Bases: `object`

Build a file tree

- **input_dir**: Input directory.
- *patterns*: A string (or a list of strings) indicating which REGULAR EXPRESSION patterns files must match (using glob.glob) [default: '.*']
- *exclude*: A string (or a list of strings) indicating which REGULAR EXPRESSION patterns files must not match (using re.search) [default: ['CVS/','.svn/']]
- *include*: A string (or a list of strings) indicating which REGULAR EXPRESSION patterns files must match if they are excluded by 'exclude' patterns (using re.search) [default: None]
- *relative*: Return file names relative to input_dir [default: False]

`file_list`(***kwargs*)

`scan`(***kwargs*)

Recursive scan to list files

`set_exclude`(*value, **kwargs*)

`set_include`(*value, **kwargs*)

`set_patterns`(*value, **kwargs*)

`geodir`(*direction, from_north=True*)

Return a direction in degrees in the form 'WNW'

`main_geodir`(*directions, amp=None, num=False, res=22.5, getamp=False, **kwargs*)

Return the dominant direction from a set of directions in degrees

`intersect`(*seg1, seg2, length=False*)

Intersection of two segments

class `Att`()

Bases: `dict`

Class to create a dictionnary and access and set keys as attributes.

You initialize and manage it as using `dict` .

Example

```
>>> dd = Att(toto=3)
>>> print dd['toto']
3
>>> print dd.toto
3
>>> dd.toto = 5
>>> print dd['toto']
5
```

`clear`()

D.clear() -> None. Remove all items from D.

`copy`()

D.copy() -> a shallow copy of D

static `fromkeys`()

dict.fromkeys(S[,v]) -> New dict with keys from S and values equal to v. v defaults to None.

`get`()

D.get(k[,d]) -> D[k] if k in D, else d. d defaults to None.

`has_key`()

D.has_key(k) -> True if D has a key k, else False

`items`()

D.items() -> list of D's (key, value) pairs, as 2-tuples

iteritems()
D.iteritems() -> an iterator over the (key, value) items of D

iterkeys()
D.iterkeys() -> an iterator over the keys of D

itervalues()
D.itervalues() -> an iterator over the values of D

keys()
D.keys() -> list of D's keys

pop()
D.pop(k[,d]) -> v, remove specified key and return the corresponding value If key is not found, d is returned if given, otherwise KeyError is raised

popitem()
D.popitem() -> (k, v), remove and return some (key, value) pair as a 2-tuple; but raise KeyError if D is empty

setdefault()
D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

update()
D.update(E, **F) -> None. Update D from E and F: for k in E: D[k] = E[k] (if E has keys else: for (k, v) in E: D[k] = v) then: for k in F: D[k] = F[k]

values()
D.values() -> list of D's values

broadcast(*set, n, mode='last', **kwargs*)
Broadcast `set` to the specified length `n`

- **set**: A single element or a sequence.
- **n**: Final requested length.
- *mode*: Filling mode.
  - `"last"`: Use the last element.
  - `"first"`: Use the first element.
  - `"cycle"`: Cycle through set.
  - if `fillvalue` is passed as a keyword, it is used to fill.

Example

```
>>> broadcast([2,4], 5)
>>> broadcast(5, 4)
>>> broadcast((2,3), 3)
>>> broadcast((2,3), 3, mode='first')
>>> broadcast([2,3], 3, mode='value', fillvalue=999)
```

makeiter(*var*)
Make var iterable as a list if not ietrable or a string

get_svn_revision(*path, max=False*)
Get the revision number of a path

Adapted from `numpy.disutils.misc_util.Configuration`

dirsize(*folder, units='b'*)
Get the size of a directory

folder: path of the directory *units*:

- `"b"`: bytes
- `"k"`: Kb
- `"m"`: Mb
- `"g"`: Gb

- "t": Tb

Cfg2Att(*cfg*)
Convert a ConfigObj object to an arborescence of Att objects

2.1.2 actimar.misc.axes — Gestion des axes

Generic tools dealing with information about longitude, latitude, depth and time axes

See Also:

Tutorials: *Manipulation des axes*

isaxis(*axis*)

islon(*axis*)
Check if a axis is longitude

islat(*axis*)
Check if a axis is latitude

islev(*axis*)
Check if a axis is levels

isdep(*axis*)
Check if a axis is levels

istime(*axis*)
Check if a axis is time

check_axes(*var*)
Check the format of all axes of a cdms variable

is_geo_axis(*axis*)
Return True if axis is time, level, lat or lon

check_axis(*axis*)
Check the format an axis

axis_type(*axis*)
Return the axis type as a signle letter (CDAT standards): -, t, z, y, or x

Example

```
>>> axis_type(create_time((5,),'days since 2000'))
't'
```

check_id(*axis, **kwargs*)
Verify that an axis has a suitable id (not like 'axis_3' but 'lon')

create(*values, atype='-', **atts*)
Quickly create an axis

- **values**: Numerical values.
- *atype*: axis type within 'x','y','z','t','-' [default: '-']
- Other keywords are passed as attributes to the axis.

create_time(*values, units=None, **atts*)
Create a time axis

- **values**: Valeurs numeriques ou liste de chaines de characteres, ou cdtime, ou datetime.
- *units*: Unites de type 'days since 2000-01-01'.
- Other keywords are passed as attributes to the axis.

```
>>> from actimar.misc.atime import create_time
>>> from datetime import datetime
>>> import cdtime
>>> taxis = create_time([1,2],units='months since 2000',long_name='My time axis')
>>> taxis = create_time(taxis)
>>> create_time([datetime(2000,1,1),'2000-2-1'],units='months since 2000')
>>> create_time([cdtime.reltime(1,'months since 2000'),cdtime.comptime(2000,1)])
```

create_lon(*values, **atts*)
Creation d'un axe de longitude

- *values*: valeurs numeriques
- Keywords are passed as attributes to the axis.

create_lat(*values, **atts*)
Creation d'un axe de latitude

- *values*: valeurs numeriques
- Keywords are passed as attributes to the axis.

create_dep(*values, **atts*)
Creation d'un axe de pronfondeur

- *values*: valeurs numeriques
- Keywords are passed as attributes to the axis.

create_depth(*values, **atts*)
Creation d'un axe de pronfondeur

- *values*: valeurs numeriques
- Keywords are passed as attributes to the axis.

2.1.3 actimar.misc.atime — Gestion du temps

Time utilities

now(*utc=False*)
Return current comptime

- *utc*: Return UTC time [default: False]

add(*mytime, amount, units*)
Add value to time

- **mytime**: Target time
- **amount**: Value to add
- **units**: Units

Example:

```
>>> add('2008-02', 3, 'days')
'2008-2-4 0:0:0.0'
```

axis_add(*oldaxis, amount, units=None, copy=0*)
Add value to time axis

- **mytime**: Target time
- **amount**: Value to add
- *units*: Units. If None, only add numeric value [default: None]

Example

```
>>> import actimar.misc as M
>>> myaxis = M.axes.create_time((5.,),units='days since 2000')
>>> M.atime.axis_add(myaxis, 10)
>>> print myaxis.asComponentTime()[0]
2000-1-11 0:0:0.0
>>> print M.atime.axis_add(myaxis, 1, 'month', copy=1).asComponentTime()[0]
2000-2-11 0:0:0.0
```

mpl(*mytimes, add_offset=-1, copy=1, **kwargs*)
Convert to matplotlib units

Keywords are passed to ch_units()

are_same_units(*units1, units2*)
Compare time units

```
>>> from actimar.misc.atime import are_same_units
>>> are_same_units('days since 1900-1', 'days since 1900-01-01 00:00:0.0')
True
```

are_good_units(*units*)
Check that units are well formatted

Example

```
>>> from actimar.misc.atime import are_good_units
>>> are_good_units('months since 2000')
True
>>> are_good_units('months since 2000-01-01 10h20')
False
```

ch_units(*mytimes, newunits, units=None, relative=False, add_offset=0, force=False, copy=False*)
Change units of a CDAT time axis or a list (or single element) or cdtime times

- **mytimes**: CDAT axis time object or a CDAT variable with a time axis
- **newunits**: New time units
- *units*: Input units [default: None]
- *relative:* If True, return relative times [default: False]
- *copy*: Create a new axis instead of updating the current one [default: False]

Return New time axis, or relatives times.

Example

```
>>> mytimes = ch_units(time_axis,relative=False,value=False,copy=True)
```

comptime(*this_time*)
Force this time to be a component time from cdtime

- **this_time**: string like '1997-01-01 00:00:00' or shorter, or a component or relative time

Example

```
>>> from datetime import datetime ; import cdtime
>>> from actimar.misc.atime import comptime
>>> comptime(datetime(2000,1,1))
2000-1-1 0:0:0.0
>>> comptime(cdtime.reltime(10,'days since 2008')).day
```

```
10
>>> comptime('1900-01-01').year
1900
```

See Also:

reltime() datetime()

reltime(*this_time, units*)
Convert to cdtime.reltime() format

See Also:

comptime() datetime()

datetime(*mytime*)
Convert to datetime.datetime() format

See Also:

comptime() reltime()

is_cdtime(*mytime*)
Check if a mytime is a cdat time (from cdtime)

Equivalent to:

```
is_reltime(mytime) or is_comptime()
```

- **mytime**: object to check

```
>>> import cdtime
>>> from datetime import datetime
>>> from actimar.misc.atime import is_cdtime
>>> is_cdtime(cdtime.reltime(2,'days since 2000'))
True
>>> is_cdtime(cdtime.comptime(2000,2))
True
>>> is_cdtime(datetime(2000,2,1)
False
```

See Also:

is_comptime()is_reltime() is_time() is_datetime()

is_reltime(*mytime*)
Check if a time is a cdat reltime (from cdtime)

- **mytime**: object to check

See Also:

is_comptime() is_reltime() is_cdtime() is_time()

is_comptime(*mytime*)
Check if a time is a cdat comptime (from cdtime)

- **mytime**: object to check

See Also:

is_datetime() is_reltime() is_cdtime() is_time()

is_datetime(*mytime*)
Check if a time is a datetime time

- **mytime**: object to check

See Also:

is_comptime() is_reltime() is_cdtime() is_time()

check_range(*this_time, time_range*)
Check wether a time is before, within or after a range

- **this_time**: time to check (string or cdat time)
- **time_range**: 2(or 3)-element range (strings or cdat times) like ('1975',1980-10-01','co)

```
>>> from actimar.misc.atime import comptime, reltime, check_range
>>> check_range('2000-12', ('2000-11', '2001'))
0
>>> check_range(comptime(2000), (comptime(2000), ('2000','20001','oc')))
-1
```

Returns -1 is before, 0 if within, 1 if after

is_in_range(*this_time, time_range*)
Check if a time is in specified closed/open range

- **this_time**: time to check (string or cdat time)
- **time_range**: 2(or 3)-element range (strings or cdat times) like ('1975',1980-10-01','co)

```
>>> is_in_range('2000-12', ('2000-11', '2001'))
True
```

See Also:

check_range()

num_to_ascii(*yyyy=1, mm=1, dd=1, hh=0, mn=0, ss=0*)
Convert from [yyyy,mm,dd,hh,mn,ss] or component time or relative time to 'yyyy-mm-dd hh:mn:ss'

- *yyyy*: int year OR component OR relative time (cdms) [default: 1]
- *mm*: month [default: 1]
- *dd*: day [default: 1]
- *hh*: hour [default: 0]
- *mn*: minute [default: 0]
- *ss*: second [default: 0]

```
>>> num_to_ascii(month=2)
0001-02-01 00:00:00
>>> num_to_ascii(comptime(2000,10))
2000-10-01 00:00:00
```

class Gaps(*var, tolerance=0.10000000000000001, verbose=True, dt=None*)
Bases: cdms2.tvariable.TransientVariable

Find time gaps in a variable

A gap is defined as a missing time step or data. With this class, you can:

- get the gaps as a variable (the object itself)
- where 1 refers to the start of gap and -1 to the end
- print them (show())
- plot them (plot())
- save them in a netcdf or an ascii file (save())

Parameters:

- **var**: A cdms variable with a cdms time axis
- *dt*: Time step [default: minimal time step]
- *tolerance*: There is a gap when dt varies of more than tolerance*min(dt) [default: 0.1]

•*keyparam*: verbose Verbose mode

```
>>> import actimar.misc as M
>>> import MV2
>>> time = M.axes.create_time([1,2,4,5],units='days since 2000')
>>> var = MV.arange(
```

plot(*show=True, savefig=None, figsize=(8, 2.0), title=None, color='red', subplots_adjust={'top': 0.80000000000000004, 'left': 0.050000000000000003, 'bottom': 0.55000000000000004}, show_time_range=True, **kwargs*)

Plot the gaps

•*color*: Color of gaps [default: 'red']

•*figure*: Show results on a figure if there are gaps [default: True]

•*show*: Show the figure plotted if gaps are found [default: True]

•*title*: Use this title for the figure

save(*file*)

Save gaps to a netcdf or an ascii file

If the file name does not end with 'cdf' or 'nc', gaps are printed to an ascii file with the save format as displayed by (show())

•**file**: Netcdf file name

show(***kwargs*)

Print out gaps

Parameters are passed to col_printer()

unit_type(*units, string_type=False, s=True*)

Returns a type of units in a suitable form with checkings.

•**units**: A valid string or cdtime type of units.

•*string_type*: Returns a string type instead of a cdtime type [default: False]

Return: A valid string or cdtime type of units

Example:

```
>>> unit_type('minutes')
>>> unit_type(cdtime.Minutes,True)
```

get_dt(*axis, units=None*)

Returns the time steps of an axis. Value is computed according to the first two steps, and returned in original or specified units.

•**axis**: A time axis.

•*units*: Another valid unit type (cdtime or string) [default: None]

Return: Time step

Example:

```
>>> get_dt(var.getTime()
>>> get_dt(var.getTime(), cdtime.Months)
>>> get_dt(var.getTime(), 'months')
```

See Also:

unit_type()

compress(*data*)

Compress along time

plot_dt(*file, time_axis=None, nice=False*)

reduce(*data, comp=True, fast=True*)

Reduce a variable in time by performing time average on time step that have the same time or time bounds.

- **data**: A cdms variable with first axis supposed to be time.
- *comp*: Call to `compress()` before reducing [default: True]
- *fast*: Convert to pure numpy before processing, then convert back to cdms variable [default: True]

Return: The new variable on its new time axis.

yearly(*data, **kwargs*)
Convert to yearly means

- **data**: A cdms variable.

Return: A cdms variable on a hourly time axis

monthly(*data, **kwargs*)
Convert to monthly means

- **data**: A cdms variable.

Return: A cdms variable on a hourly time axis

hourly(*data, frequency=24, **kwargs*)
Convert to hourly means

- **data**: A cdms variable.
- *frequency* Used when different from hourly is requested [default: 24]

Return: A cdms variable on a hourly time axis

daily(*data, frequency=1, **kwargs*)
Convert to daily means

- **data**: A cdms variable.
- *frequency*: Daily frequency of averages, with 1 = daily, and 24 = hourly.

Return: A cdms variable on a daily time axis

hourly_exact(*data, time_units=None, maxgap=None, ctlims=None*)
Linearly interpolate data at exact beginning of hours

- **data**: Cdms variable.
- *time_units*: Change time units.
- *maxgap*: Maximal gap in hours to enable interpolation [default: None].

trend(*var*)
Get linear trend

detrend(*var*)
Linear detrend

strftime(*fmt, mytime=None*)
Convert current time, datetime, cdtime or string time to strftime

- **fmt**: Time format
- *mytime*: If None, takes current time [default: None]

See Also:

`datetime.datetime.strftime()`

strptime(*mytime, fmt*)
Parse a string according to a format to retreive a component time

- **fmt**: format like %Y-%m-%d.
- **mytime**: string date.

tz_to_tz(*mytime, old_tz, new_tz*)
Convert time from one time zone to another one

from_utc(*mytime, new_tz*)

to_utc(*mytime, old_tz*)

paris_to_utc(*mytime*)

class DateSorter(*pattern=None, basename=True*)
Bases: `object`

Sort a list of date string, optionally using a date pattern

Example:

```
>>> from actimar.misc.atime import DateSorter
>>> dates = ['annee 2006', 'annee 2002']
>>> ds = DateSorter('annee %Y')
>>> dates.sort(ds)
>>> print dates
['2002', '2006']
```

class SpecialDateFormatter(*level, fmt=None, special_fmt=None, join=None, phase=None, **kwargs*)
Bases: `matplotlib.dates.DateFormatter`

Special formatter for dates Example: ['00h 01/10/2000' '02h' .... '23h' '00h 01/10/2000'] to mark days and keep hours Here the 'phase' is 0 (00h) and the level is 3 (='day')

interp(*vari, outtimes, squeeze=1, **kwargs*)
Linear interpolation in time

- **vari****: A cdms variable with a time axis
- **outtimes**: A time axis, or a list (or single element) of date strings, comptime, reltime, datetime times.
- *squeeze*: Remove uneeded output dimensions [default: 1]
- all other keywords are passed to `interp1d()`.

is_time(*mytime*)
Check if mytime is a CDAT time or a datetime.datetime()

Equivalent to:

```
is_cdtime(mytime) or is_datetime(mytime)
```

See Also:

`is_datetime()` `is_reltime()` `is_cdtime()` `is_time()`

selector(*arg0, arg1=None, bounds=None, round=False, utc=True*)
Time selector formatter that returns start date and end date as component times

Usage:

```
>>> from actimar.misc.atime import selector as time_selector
>>> time_selector('2006','2007') # between two dates
>>> time_selector(comptime(1950)) # from a date to now
>>> time_selector(1,'month','co') # from now into the past
```

round_date(*mydate, round_type*)
Round a date to year, month, day, hour, minute or second

class Intervals(*time_range, interval, reverse=False, roundto=None, bounds=True*)
Iterator on intervals

Example:

```
>>> from actimar.misc.atime import Intervals
>>> for itv in Intervals(('2000','2001','co'),(1,'month')): print itv
>>> for itv in Intervals(('2000','2001','co'),12): print itv
```

next()

round(*mydate*)

utc_to_paris(*mytime*)

ascii_to_num(*str*)
Convert from 'yyyy-mm-dd hh:mn:ss' to [yyyy,mm,dd,hh,mn,ss]

```
>>> ascii_to_num('2000-01')
2000, 1, 1, 0, 0, 0
```

lindates(*first, last, incr, units*)
Create a list of linearly incrementing dates

interp_old(*var, outtimes, squeeze=1*)
Linear interpolation in time

- **var****: A cdms variable with a time axis
- **outtimes**: A time axis, or a list (or single element) of date strings, comptime, reltime, datetime times.
- *queeze*: Remove uneeded output dimensions [default: 1]

2.1.4 actimar.misc.plot — Graphiques

Generic plots using Matplotlib and taking advantage of CDAT

See: *Les graphiques*.

curve(*var, parg=None, along=None, vertical=None, label=None, nosingle=False, nodate=False, shadow=False, shadow_alpha=0.29999999999999999, shadow_att=0.29999999999999999, shadow_offset=0.040000000000000001, zorder=None, axis_setup=True, profile=None, units=None, **kwargs*)
Quick plot an 1D cdms variable as curve

- **var**: A 1D cdms variable

Curve:

- *along*: A letter within 'x','y','z','t' to select the axis ; None select the LAST AXIS (if 'tzyx', it plots along 'x') ; it reduces dimensions by averaging along successive axes [default: None]
- *color*: Color of the line .
- *linewidth*: Width of the line.
- *linestyle*: Style of the line.
- *label*: Specify the label to shown in legend. It defaults to the long name of the variable. If False, _nolegend_ is set.
- *units*: Label of var's axis. It defaults to the units of the variable. If False, no label is drawn.
- *nosingle*: Does not plot a point when enclosed by missing data [default: False]

Axis decoration:

- *[x/y]title*: Main label of the axis.
- *[x/y]label*: Same as x/ytitle.
- *[x/y]hide]*: Hide x/ytick labels.
- *[x/y]rotation*: Rotation of x/ytick labels (except for time).
- *[x/y]strict*: Strict axis limit.

- *[x/y]lim*: Override axis limit.
- *[x/y]min/max*: Override axis limit by x/ylim.
- *[x/y]minmax/maxmin*: Set maximal value of x/ymin and minimal value or x/ymax.
- *[x/y]nmax*: Max number of ticks.
- *x/yfmt*: Numeric format for x/y axis if not of time type.
- *date_rotation*: Rotation of time ticklabels.
- *date_locator*: Major locator for dates.
- *date_minor_locator*: Minor locator for dates.
- *date_nominor*: Suppress minor ticks for dates.
- *date_nmax_ticks*: Max number of ticks for dates
- *nodate*: Time axis must not be formatted as a time axis [default: False]

Misc settings:

- *figure*: Figure number.
- *figsize*: Initialize the figure with this size.
- *subplots_adjust*: Dictionary sent to `subplots_adjust()`. You can also use keyparams 'left', 'right', 'top', 'bottom', 'wspace', 'hspace' !
- *sa*: Alias for subplots_adjust.
- *bgcolor*: Background axis color.
- *axes_rect*: [left, bottom, width, height] in normalized (0,1) units to create axes using `axes()`.
- *axes_<keyword>*: <keyword> is passed to `axes()`.
- *title*: Title of the figure [defaults to var.long_name or '']
- *grid*: Plot the grid [default: True]
- *dayhl*: Add day highlithing [default: False]
- *figtext*: figtext Add text at a specified position on the figure. Example: figtext=[0,0,'text'] add a 'text' at the lower left corner, or simply figtext='text'.
- *anchor*: Anchor of the axes (useful when resizing) in ['C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'].
- *logo*: Add a logo to the figure [default: False]. logo can be a file name.
- *show*: Display the figure [default: True]
- *savefig*: Save the figure to this file.
- *savefigs*: Save the figure into multiple formats using `savefigs()` and 'savefigs' as the prefix to the files.
- *autoresize*: Auto resize the figure according axes (1 or True), axes+margins (2). If 0 or False, not resized [default: False=2].
- *key*: Add a key (like 'a)') to the axes using add_key is different from None [default: None]
- *close*: Close the figure at the end [default: False]
- *title_<keyword>*: <keyword> is passed to `title()`
- *key_<keyword>*: <keyword> is passed to `add_key()`
- *logo_<keyword>*: <keyword> is passed to `add_logo()`
- *figtext_<keyword>*: <keyword> is passed to `figtext()`
- *savefig_<keyword>*: <keyword> is passed to `savefig()`
- *savefigs_<keyword>*: <keyword> is passed to `savefigs()`
- *grid_<keyword>*: <keyword> is passed to `grid()`

All other keywords are passed to the plot function (like color, linewidth, etc).

map(*var=None, fill='pcolor', contour=True, overlay=False, fullscreen=False, lon_min=None, lon_max=None, lat_min=None, lat_max=None, lon_center=None, lat_center=None, lon=None, lat=None, lat_ts=None, m=None, minutes=True, drawnothing=False, maponly=False, drawrivers=False, fillcontinents=True, resolution='auto', meridional_labels=True, zonal_labels=True, land_color=None, getmap=True, ticklabel_size=None, getcb=False, projection='cyl', refine=0, drawcoastlines=True, drawmapboundary=True, meridians=None, parallels=None, nocache=False, no_seconds=False, cache_dir=None, **kwargs*)

Plot a map with contours and colorbar

- *var*: A 2D (or more) cdms variable. If None, plot only the map [default: None].
- *xaxis*: Use this X axis.
- *yaxis*: Use this Y axis.
- *tadd*: Add value to time (like tadd=1 or tadd=(1,'day'))
- *tadd*: Make a copy of the axis before any tadd [default: True]
- *tlocal*: Convert current UTC time axes to local time [default: False]

Map parameters:

- *m*: Plot on an old map.
- *lon_min*: Min longitude of the map.
- *lon_max*: Max longitude of the map.
- *lat_min*: Min latitude of the map.
- *lat_max*: Max latitude of the map.
- *lon_center*: Longitude of the center of the map.
- *lat_center*: Latitude of the center of the map.
- *oversamp*: Oversample in both direction using bilinear interpolation, if oversamp > 1 [default: 0]
- *projection*: Projection type. Special value is 'rgf93' [default: 'cyl']
- *drawparallels*: Draw parallels and its labels [default: True]
- *parallels*: Specify parallels manually.
- *drawmeridians*: Draw meridians and its labels [default: True]
- *meridians*: Specify meridians manually.
- *zonal_labels*: Display zonal labels [default: True]
- *meridional_labels*: Display meridional labels [default: True]
- *resolution*: of the coast line [default: 'i']
- *fillcontinents*: Fill the continents [default: True]
- *drawrivers*: Draw rivers [default: False]
- *drawcoastlines*: Draw coast lines [default: True]
- *drawmapboundary*: Draw map boundaries [default: True]
- *ticklabel_size*: Change size of tick labels.
- *basemap_<keyword>*: <keyword> is passed to `Basemap`
- *fillcontinents_<keyword>*: <keyword> is passed to `Basemap.fillcontinents()`
- *drawrivers_<keyword>*: <keyword> is passed to `Basemap.drawrivers()`
- *drawcoastlines_<keyword>*: <keyword> is passed to `Basemap.drawcoastlines()`
- *drawmapboundary_<keyword>*: <keyword> is passed to `Basemap.drawmapboundary()`
- *drawparallels_<keyword>*: <keyword> is passed to `Basemap.drawparallels()`
- *drawmeridians_<keyword>*: <keyword> is passed to `Basemap.drawmeridians()`
- *drawparallels_gs_<keyword>*: <keyword> is passed to geo_scale for finding `Basemap.parallels()`
- *drawmeridians_gs_<keyword>*: <keyword> is passed to geo_scale for finding `Basemap.meridians()`

Contours:

- *fill*: Fill method.
  - "nofill" or "no" or 0 or False: No fill
  - "pcolor" or 1: Use pcolor() (default)
  - "contourf" or "contour" or 2: Use contourf()
  - "imshow" or 3: Use imshow() (your grid must be rectangular and regular!)
- *contour*: Add line contours [default: True]
- *nomissing*: Fill missing value using inter/extrapolation [default: False]
- *cmap*: Colormap or a colormap name. If a string, it is passed to get_cmap(), except if it starts with 'cmap_' where it the corresponding function from actimar.misc.plot.color is used instead.
- *alpha*: Tranparency of filled (and pcolor) contours [default: 1.]
- *shading*: For pcolor [default: 'flat']
- *linewidths*: Line width of contours.
- *fmt*: Numeric format if contour labels [default: '%g']
- *clabel_fontsize*: Font size of contour labels.
- *clabel_hide*: Hide contour labels [default: False]
- *pcolor_<keyword>*: <keyword> is passed to pcolor()
- *contourf_<keyword>*: <keyword> is passed to contourf()
- *contour_<keyword>*: <keyword> is passed to:func:' ~matplotlib.pyplot.contour'
- *clabel_<keyword>*: <keyword> is passed to clabel()

Arrows:

- *quiver_polar*: (u,v) is treated as (rho,theta) [default: False]
- *quiver_norm*:
  - If 0, length of arrows is linearly dependent on modulus [default].
  - If 1, length is constant.
  - if 2, length is constant but color depends on modulus. In this case, you can pass a colormap using the keyword quiver_cmap=... to adjust the color of your arrows.
  - if 3, combination of 0 and 2 (size and color vary)
- *quiver_mask_func*: Function to mask data with modulus as first arguments [default: None]
- *quiver_mask_args*: Mandatory arguments to quiver_mask_func [default: []]
- *quiver_mask_kwargs*: Optional arguments to quiver_mask_func [default: {}]
- *quiver_samp*: Horizontal sampling of arrows (in both directions) [default: 1]
- *quiver_xsamp*: Sampling along X [default: quiver_samp]
- *quiver_ysamp*: Sampling along Y [default: quiver_samp]
- *quiver_res*: Horizontal resolution of arrows (in both directions) for undersampling [default: None]
- *quiver_xres*: Same along X [default: quiver_res]
- *quiver_yres*: Same along Y [default: quiver_res]
- *quiver_relres*: Relative resolution (in both directions). If > 0, = mean(res)*relres. If < -1, =''min(res)*abs(relres)''. If < 0 and > -1, =max(res)*abs(relres) [default: None]se
- *quiver_xrelres*: Same along X [default: quiver_relres]
- *quiver_yrelres*: Same along Y [default: quiver_relres]
- *quiver_usemapres*: For maps, map coordinates (in meters) are used to compute resolution, else degrees are used [default: quiver_relres]
- *quiver_georotate*: Rotate arrow from geo to map coordinate. Useful if coordinate are in degrees. Do not useful if coordinate in meters for example [default: False]
- *quiver_color*: Color of arrows [default: 'w']

- *quiver_linewidth*: Edge width [default: 1]
- *quiver_edgecolor*: Edge color [default: `'k'`]
- *quiver_shadow*: Add a shadow to arrow [default: False]
- *quiver_shadow_alpha:* Alpha for shadow [default: 0.5]
- *quiver_scaling*: Scale Y component of arrows with respect to X, for quiver. If True, scaling is guessed from geo axes (map, sections). It can be an array to be applied directly [default: True]
- *quiver_scaling_lat*: Latitude for depth/lon scaling [default: 48.]
- *quiver_<keyword>*: <keyword> is passed to `quiver()`
- *quiverkey*: Plot the key [default: True]

Quiver key:

- *quiverkey_pos*: Position of key for arrow [default: (0.0,1.02)]
- *quiverkey_text*: Text or format with variables 'value' and 'units' [default: '%(value)g %(units)s'].
- *quiverkey_value*: Numeric value for key (used by text) [default: None = deduced from cdms var]
- *quiverkey_units*: Units for key (used by text) [default: None = deduced from cdms var]
- *quiverkey_<keyword>*: <keyword> is passed to `quiverkey()`

Data levels:

- *levels*: Force the use of these levels for contours.
- *anomaly*: Levels must be symetric about zero (useful for anomly plots) and color map is misc.plot.cmap_bwre() [default: None]. If None, is is turned to True if max is near -min (20%)
- *nmax*: Max number of levels (see misc.auto_scale) [default: 10]
- *vmin*: Force min value for pcolor.
- *vmax*: Force max value for pcolor.

Axis decoration:

- *[x/y]title*: Main label of the axis.
- *[x/y]label*: Same as x/ytitle.
- *[x/y]hide]*: Hide x/ytick labels.
- *[x/y]rotation*: Rotation of x/ytick labels (except for time).
- *[x/y]strict*: Strict axis limit.
- *[x/y]lim*: Override axis limit.
- *[x/y]min/max*: Override axis limit by x/ylim.
- *[x/y]minmax/maxmin*: Set maximal value of x/ymin and minimal value or x/ymax.
- *[x/y]nmax*: Max number of ticks.
- *x/yfmt*: Numeric format for x/y axis if not of time type.
- *date_rotation*: Rotation of time ticklabels.
- *date_locator*: Major locator for dates.
- *date_minor_locator*: Minor locator for dates.
- *date_nominor*: Suppress minor ticks for dates.
- *date_nmax_ticks*: Max number of ticks for dates
- *nodate*: Time axis must not be formatted as a time axis [default: False]

Colorbar:

- *colorbar*: Plot the colorbar [default: True]
- *colorbar_horizontal*: Colorbar is horizontal [defaults: False]

- *colorbar_position*: To change the default position. Position is relative to the SPACE LEFT in the form (center,width) with values in [0,1] [defaults: False]
- *colorbar_visible*: Colorbar is visible [defaults: True]
- *colorbar_<keyword>*: <keyword> is passed to `colorbar()`
- *units*: Indicate these units on along the colorbar, else it guessed from the variable or suppressed if value is False [default: None]

Misc settings:

- *figure*: Figure number.
- *figsize*: Initialize the figure with this size.
- *subplots_adjust*: Dictionary sent to `subplots_adjust()`. You can also use keyparams 'left', 'right', 'top', 'bottom', 'wspace', 'hspace' !
- *sa*: Alias for subplots_adjust.
- *bgcolor*: Background axis color.
- *axes_rect*: [left, bottom, width, height] in normalized (0,1) units to create axes using `axes()`.
- *axes_<keyword>*: <keyword> is passed to `axes()`.
- *title*: Title of the figure [defaults to var.long_name or '']
- *grid*: Plot the grid [default: True]
- *dayhl*: Add day highlithing [default: False]
- *figtext*: figtext Add text at a specified position on the figure. Example: figtext=[0,0,'text'] add a 'text' at the lower left corner, or simply figtext='text'.
- *anchor*: Anchor of the axes (useful when resizing) in ['C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'].
- *logo*: Add a logo to the figure [default: False]. logo can be a file name.
- *show*: Display the figure [default: True]
- *savefig*: Save the figure to this file.
- *savefigs*: Save the figure into multiple formats using `savefigs()` and 'savefigs' as the prefix to the files.
- *autoresize*: Auto resize the figure according axes (1 or True), axes+margins (2). If 0 or False, not resized [default: False=2].
- *key*: Add a key (like 'a)') to the axes using add_key is different from None [default: None]
- *close*: Close the figure at the end [default: False]
- *title_<keyword>*: <keyword> is passed to `title()`
- *key_<keyword>*: <keyword> is passed to `add_key()`
- *logo_<keyword>*: <keyword> is passed to `add_logo()`
- *figtext_<keyword>*: <keyword> is passed to `figtext()`
- *savefig_<keyword>*: <keyword> is passed to `savefig()`
- *savefigs_<keyword>*: <keyword> is passed to `savefigs()`
- *grid_<keyword>*: <keyword> is passed to `grid()`

Output:

- *getmap*: If True, returns the map [default:: True]
- *getcb*: If True, returns the feed object for colorbar [default: False]

`section`(*var, zonal=None, fill='pcolor', contour=True, getcb=False, bgcolor='#666666', **kwargs*)

Plot a geographic section: contours of lat (or lon) along depth.

Section:

- **var**: A 2D (or more) variable. If more than 2D, it is averaged over the first dimensions.

- *xaxis*: Use this X axis.
- *yaxis*: Use this Y axis.
- *tadd*: Add value to time (like tadd=1 or tadd=(1,'day'))
- *tadd*: Make a copy of the axis before any tadd [default: True]
- *tlocal*: Convert current UTC time axes to local time [default: False]
- *zonal*: Force a zonal section plot if True, a meridional if False, else guess it [defaults to zonal]

Contours:

- *fill*: Fill method.
  - "nofill" or "no" or 0 or False: No fill
  - "pcolor" or 1: Use pcolor() (default)
  - "contourf" or "contour" or 2: Use contourf()
  - "imshow" or 3: Use imshow() (your grid must be rectangular and regular!)
- *contour*: Add line contours [default: True]
- *nomissing*: Fill missing value using inter/extrapolation [default: False]
- *cmap*: Colormap or a colormap name. If a string, it is passed to get_cmap(), except if it starts with 'cmap_' where it the corresponding function from actimar.misc.plot.color is used instead.
- *alpha*: Tranparency of filled (and pcolor) contours [default: 1.]
- *shading*: For pcolor [default: 'flat']
- *linewidths*: Line width of contours.
- *fmt*: Numeric format if contour labels [default: '%g']
- *clabel_fontsize*: Font size of contour labels.
- *clabel_hide*: Hide contour labels [default: False]
- *pcolor_<keyword>*: <keyword> is passed to pcolor()
- *contourf_<keyword>*: <keyword> is passed to contourf()
- *contour_<keyword>*: <keyword> is passed to:func:' ~matplotlib.pyplot.contour'
- *clabel_<keyword>*: <keyword> is passed to clabel()

Arrows:

- *quiver_polar*: (u,v) is treated as (rho,theta) [default: False]
- *quiver_norm*:
  - If 0, length of arrows is linearly dependent on modulus [default].
  - If 1, length is constant.
  - if 2, length is constant but color depends on modulus. In this case, you can pass a colormap using the keyword quiver_cmap=... to adjust the color of your arrows.
  - if 3, combination of 0 and 2 (size and color vary)
- *quiver_mask_func*: Function to mask data with modulus as first arguments [default: None]
- *quiver_mask_args*: Mandatory arguments to quiver_mask_func [default: []]
- *quiver_mask_kwargs*: Optional arguments to quiver_mask_func [default: {}]
- *quiver_samp*: Horizontal sampling of arrows (in both directions) [default: 1]
- *quiver_xsamp*: Sampling along X [default: quiver_samp]
- *quiver_ysamp*: Sampling along Y [default: quiver_samp]
- *quiver_res*: Horizontal resolution of arrows (in both directions) for undersampling [default: None]
- *quiver_xres*: Same along X [default: quiver_res]
- *quiver_yres*: Same along Y [default: quiver_res]
- *quiver_relres*: Relative resolution (in both directions). If > 0, = mean(res)*relres. If < -1, =''min(res)*abs(relres)''. If < 0 and > -1, =max(res)*abs(relres) [default: None]se

- *quiver_xrelres*: Same along X [default: quiver_relres]
- *quiver_yrelres*: Same along Y [default: quiver_relres]
- *quiver_usemapres*: For maps, map coordinates (in meters) are used to compute resolution, else degrees are used [default: quiver_relres]
- *quiver_georotate*: Rotate arrow from geo to map coordinate. Useful if coordinate are in degrees. Do not useful if coordinate in meters for example [default: False]
- *quiver_color*: Color of arrows [default: `'w'`]
- *quiver_linewidth*: Edge width [default: 1]
- *quiver_edgecolor*: Edge color [default: `'k'`]
- *quiver_shadow*: Add a shadow to arrow [default: False]
- *quiver_shadow_alpha:* Alpha for shadow [default: 0.5]
- *quiver_scaling*: Scale Y component of arrows with respect to X, for quiver. If True, scaling is guessed from geo axes (map, sections). It can be an array to be applied directly [default: True]
- *quiver_scaling_lat*: Latitude for depth/lon scaling [default: 48.]
- *quiver_<keyword>*: <keyword> is passed to `quiver()`
- *quiverkey*: Plot the key [default: True]

Quiver key:

- *quiverkey_pos*: Position of key for arrow [default: (0.0,1.02)]
- *quiverkey_text*: Text or format with variables 'value' and 'units' [default: '%(value)g %(units)s'].
- *quiverkey_value*: Numeric value for key (used by text) [default: None = deduced from cdms var]
- *quiverkey_units*: Units for key (used by text) [default: None = deduced from cdms var]
- *quiverkey_<keyword>*: <keyword> is passed to `quiverkey()`

Data levels:

- *levels*: Force the use of these levels for contours.
- *anomaly*: Levels must be symetric about zero (useful for anomly plots) and color map is misc.plot.cmap_bwre() [default: None]. If None, is is turned to True if max is near -min (20%)
- *nmax*: Max number of levels (see misc.auto_scale) [default: 10]
- *vmin*: Force min value for pcolor.
- *vmax*: Force max value for pcolor.

Axis decoration:

- *[x/y]title*: Main label of the axis.
- *[x/y]label*: Same as x/ytitle.
- *[x/y]hide]*: Hide x/ytick labels.
- *[x/y]rotation*: Rotation of x/ytick labels (except for time).
- *[x/y]strict*: Strict axis limit.
- *[x/y]lim*: Override axis limit.
- *[x/y]min/max*: Override axis limit by x/ylim.
- *[x/y]minmax/maxmin*: Set maximal value of x/ymin and minimal value or x/ymax.
- *[x/y]nmax*: Max number of ticks.
- *x/yfmt*: Numeric format for x/y axis if not of time type.
- *date_rotation*: Rotation of time ticklabels.
- *date_locator*: Major locator for dates.
- *date_minor_locator*: Minor locator for dates.
- *date_nominor*: Suppress minor ticks for dates.

- *date_nmax_ticks*: Max number of ticks for dates
- *nodate*: Time axis must not be formatted as a time axis [default: False]

Colorbar:

- *colorbar*: Plot the colorbar [default: True]
- *colorbar_horizontal*: Colorbar is horizontal [defaults: False]
- *colorbar_position*: To change the default position. Position is relative to the SPACE LEFT in the form (center,width) with values in [0,1] [defaults: False]
- *colorbar_visible*: Colorbar is visible [defaults: True]
- *colorbar_<keyword>*: <keyword> is passed to `colorbar()`
- *units*: Indicate these units on along the colorbar, else it guessed from the variable or suppressed if value is False [default: None]

Misc settings:

- *figure*: Figure number.
- *figsize*: Initialize the figure with this size.
- *subplots_adjust*: Dictionary sent to `subplots_adjust()`. You can also use keyparams 'left', 'right', 'top', 'bottom', 'wspace', 'hspace' !
- *sa*: Alias for subplots_adjust.
- *bgcolor*: Background axis color.
- *axes_rect*: [left, bottom, width, height] in normalized (0,1) units to create axes using `axes()`.
- *axes_<keyword>*: <keyword> is passed to `axes()`.
- *title*: Title of the figure [defaults to var.long_name or '']
- *grid*: Plot the grid [default: True]
- *dayhl*: Add day highlithing [default: False]
- *figtext*: figtext Add text at a specified position on the figure. Example: figtext=[0,0,'text'] add a 'text' at the lower left corner, or simply figtext='text'.
- *anchor*: Anchor of the axes (useful when resizing) in ['C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'].
- *logo*: Add a logo to the figure [default: False]. logo can be a file name.
- *show*: Display the figure [default: True]
- *savefig*: Save the figure to this file.
- *savefigs*: Save the figure into multiple formats using `savefigs()` and 'savefigs' as the prefix to the files.
- *autoresize*: Auto resize the figure according axes (1 or True), axes+margins (2). If 0 or False, not resized [default: False=2].
- *key*: Add a key (like 'a)') to the axes using add_key is different from None [default: None]
- *close*: Close the figure at the end [default: False]
- *title_<keyword>*: <keyword> is passed to `title()`
- *key_<keyword>*: <keyword> is passed to `add_key()`
- *logo_<keyword>*: <keyword> is passed to `add_logo()`
- *figtext_<keyword>*: <keyword> is passed to `figtext()`
- *savefig_<keyword>*: <keyword> is passed to `savefig()`
- *savefigs_<keyword>*: <keyword> is passed to `savefigs()`
- *grid_<keyword>*: <keyword> is passed to `grid()`
- *getcb*: If True, returns the feed object for colorbar [default: False]

hov(*var, fill='pcolor', contour=True, time_vertical=None, getcb=False, **kwargs*)
Plot a hovmoller diagram: contours where one of the dimensions is time.

- **var**: A 2D (or more) variable. If more than 2D, it is averaged over the first dimensions.

Hovmoller:

- *xaxis*: Use this X axis.
- *yaxis*: Use this Y axis.
- *time_vertical*: Y axis is time.

Contours:

- *fill*: Fill method.
  - "nofill" or "no" or 0 or False: No fill
  - "pcolor" or 1: Use pcolor() (default)
  - "contourf" or "contour" or 2: Use contourf()
  - "imshow" or 3: Use imshow() (your grid must be rectangular and regular!)
- *contour*: Add line contours [default: True]
- *nomissing*: Fill missing value using inter/extrapolation [default: False]
- *cmap*: Colormap or a colormap name. If a string, it is passed to get_cmap(), except if it starts with 'cmap_' where it the corresponding function from actimar.misc.plot.color is used instead.
- *alpha*: Tranparency of filled (and pcolor) contours [default: 1.]
- *shading*: For pcolor [default: 'flat']
- *linewidths*: Line width of contours.
- *fmt*: Numeric format if contour labels [default: '%g']
- *clabel_fontsize*: Font size of contour labels.
- *clabel_hide*: Hide contour labels [default: False]
- *pcolor_<keyword>*: <keyword> is passed to pcolor()
- *contourf_<keyword>*: <keyword> is passed to contourf()
- *contour_<keyword>*: <keyword> is passed to:func:' ~matplotlib.pyplot.contour'
- *clabel_<keyword>*: <keyword> is passed to clabel()

Arrows:

- *quiver_polar*: (u,v) is treated as (rho,theta) [default: False]
- *quiver_norm*:
  - If 0, length of arrows is linearly dependent on modulus [default].
  - If 1, length is constant.
  - if 2, length is constant but color depends on modulus. In this case, you can pass a colormap using the keyword quiver_cmap=... to adjust the color of your arrows.
  - if 3, combination of 0 and 2 (size and color vary)
- *quiver_mask_func*: Function to mask data with modulus as first arguments [default: None]
- *quiver_mask_args*: Mandatory arguments to quiver_mask_func [default: []]
- *quiver_mask_kwargs*: Optional arguments to quiver_mask_func [default: {}]
- *quiver_samp*: Horizontal sampling of arrows (in both directions) [default: 1]
- *quiver_xsamp*: Sampling along X [default: quiver_samp]
- *quiver_ysamp*: Sampling along Y [default: quiver_samp]
- *quiver_res*: Horizontal resolution of arrows (in both directions) for undersampling [default: None]
- *quiver_xres*: Same along X [default: quiver_res]
- *quiver_yres*: Same along Y [default: quiver_res]

- *quiver_relres*: Relative resolution (in both directions). If > 0, = `mean(res)*relres`. If < -1, =‘‘min(res)*abs(relres)‘‘. If < 0 and > -1, =max(res)*abs(relres) [default: None]se
- *quiver_xrelres*: Same along X [default: quiver_relres]
- *quiver_yrelres*: Same along Y [default: quiver_relres]
- *quiver_usemapres*: For maps, map coordinates (in meters) are used to compute resolution, else degrees are used [default: quiver_relres]
- *quiver_georotate*: Rotate arrow from geo to map coordinate. Useful if coordinate are in degrees. Do not useful if coordinate in meters for example [default: False]
- *quiver_color*: Color of arrows [default: `'w'`]
- *quiver_linewidth*: Edge width [default: 1]
- *quiver_edgecolor*: Edge color [default: `'k'`]
- *quiver_shadow*: Add a shadow to arrow [default: False]
- *quiver_shadow_alpha:* Alpha for shadow [default: 0.5]
- *quiver_scaling*: Scale Y component of arrows with respect to X, for quiver. If True, scaling is guessed from geo axes (map, sections). It can be an array to be applied directly [default: True]
- *quiver_scaling_lat*: Latitude for depth/lon scaling [default: 48.]
- *quiver_<keyword>*: <keyword> is passed to `quiver()`
- *quiverkey*: Plot the key [default: True]

Quiver key:

- *quiverkey_pos*: Position of key for arrow [default: (0.0,1.02)]
- *quiverkey_text*: Text or format with variables ‘value’ and ‘units’ [default: ‘%(value)g %(units)s’].
- *quiverkey_value*: Numeric value for key (used by text) [default: None = deduced from cdms var]
- *quiverkey_units*: Units for key (used by text) [default: None = deduced from cdms var]
- *quiverkey_<keyword>*: <keyword> is passed to `quiverkey()`

Data levels:

- *levels*: Force the use of these levels for contours.
- *anomaly*: Levels must be symetric about zero (useful for anomly plots) and color map is misc.plot.cmap_bwre() [default: None]. If None, is is turned to True if max is near -min (20%)
- *nmax*: Max number of levels (see misc.auto_scale) [default: 10]
- *vmin*: Force min value for pcolor.
- *vmax*: Force max value for pcolor.

Axis decoration:

- *[x/y]title*: Main label of the axis.
- *[x/y]label*: Same as x/ytitle.
- *[x/y]hide]*: Hide x/ytick labels.
- *[x/y]rotation*: Rotation of x/ytick labels (except for time).
- *[x/y]strict*: Strict axis limit.
- *[x/y]lim*: Override axis limit.
- *[x/y]min/max*: Override axis limit by x/ylim.
- *[x/y]minmax/maxmin*: Set maximal value of x/ymin and minimal value or x/ymax.
- *[x/y]nmax*: Max number of ticks.
- *x/yfmt*: Numeric format for x/y axis if not of time type.
- *date_rotation*: Rotation of time ticklabels.
- *date_locator*: Major locator for dates.

- *date_minor_locator*: Minor locator for dates.
- *date_nominor*: Suppress minor ticks for dates.
- *date_nmax_ticks*: Max number of ticks for dates
- *nodate*: Time axis must not be formatted as a time axis [default: False]

Colorbar:

- *colorbar*: Plot the colorbar [default: True]
- *colorbar_horizontal*: Colorbar is horizontal [defaults: False]
- *colorbar_position*: To change the default position. Position is relative to the SPACE LEFT in the form (center,width) with values in [0,1] [defaults: False]
- *colorbar_visible*: Colorbar is visible [defaults: True]
- *colorbar_<keyword>*: <keyword> is passed to `colorbar()`
- *units*: Indicate these units on along the colorbar, else it guessed from the variable or suppressed if value is False [default: None]

Misc settings:

- *figure*: Figure number.
- *figsize*: Initialize the figure with this size.
- *subplots_adjust*: Dictionary sent to `subplots_adjust()`. You can also use keyparams 'left', 'right', 'top', 'bottom', 'wspace', 'hspace' !
- *sa*: Alias for subplots_adjust.
- *bgcolor*: Background axis color.
- *axes_rect*: [left, bottom, width, height] in normalized (0,1) units to create axes using `axes()`.
- *axes_<keyword>*: <keyword> is passed to `axes()`.
- *title*: Title of the figure [defaults to var.long_name or '']
- *grid*: Plot the grid [default: True]
- *dayhl*: Add day highlithing [default: False]
- *figtext*: figtext Add text at a specified position on the figure. Example: figtext=[0,0,'text'] add a 'text' at the lower left corner, or simply figtext='text'.
- *anchor*: Anchor of the axes (useful when resizing) in ['C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'].
- *logo*: Add a logo to the figure [default: False]. logo can be a file name.
- *show*: Display the figure [default: True]
- *savefig*: Save the figure to this file.
- *savefigs*: Save the figure into multiple formats using `savefigs()` and 'savefigs' as the prefix to the files.
- *autoresize*: Auto resize the figure according axes (1 or True), axes+margins (2). If 0 or False, not resized [default: False=2].
- *key*: Add a key (like 'a)') to the axes using add_key is different from None [default: None]
- *close*: Close the figure at the end [default: False]
- *title_<keyword>*: <keyword> is passed to `title()`
- *key_<keyword>*: <keyword> is passed to `add_key()`
- *logo_<keyword>*: <keyword> is passed to `add_logo()`
- *figtext_<keyword>*: <keyword> is passed to `figtext()`
- *savefig_<keyword>*: <keyword> is passed to `savefig()`
- *savefigs_<keyword>*: <keyword> is passed to `savefigs()`
- *grid_<keyword>*: <keyword> is passed to `grid()`

- *getcb*: If True, returns the feed object for colorbar [default: False]

vector_line(*uu, vv, polar=False, degrees=False, ycenter=None, color='k', alpha=1, quiverkey=True, units=None, cmap=None, nodate=False, **kwargs*)

Plot a line of vectors

- **uu**: X coordinates of vectors OR length if polar=True
- **vv**: Y coordinates of vectors OR angle if polar=True

Vectors:

- *xaxis*: Use this X axis.
- *polar*: Consider input as polar instead of cartesian coordinates [default: False]
- *degrees*: If polar=True, angle (vv) is in degrees instead of radians [default: False]
- *color*: Color of arrows [default: 'k']
- *color*: Alpha transparency of arrows [default: 1]
- *quiver_<keyword>*: <keyword> is passed to `quiver()`

Quiver key:

- *quiverkey_pos*: Position of key for arrow [default: (0.0,1.02)]
- *quiverkey_text*: Text or format with variables 'value' and 'units' [default: '%(value)g %(units)s'].
- *quiverkey_value*: Numeric value for key (used by text) [default: None = deduced from cdms var]
- *quiverkey_units*: Units for key (used by text) [default: None = deduced from cdms var]
- *quiverkey_<keyword>*: <keyword> is passed to `quiverkey()`

Axis decoration:

- *[x/y]title*: Main label of the axis.
- *[x/y]label*: Same as x/ytitle.
- *[x/y]hide]*: Hide x/ytick labels.
- *[x/y]rotation*: Rotation of x/ytick labels (except for time).
- *[x/y]strict*: Strict axis limit.
- *[x/y]lim*: Override axis limit.
- *[x/y]min/max*: Override axis limit by x/ylim.
- *[x/y]minmax/maxmin*: Set maximal value of x/ymin and minimal value or x/ymax.
- *[x/y]nmax*: Max number of ticks.
- *x/yfmt*: Numeric format for x/y axis if not of time type.
- *date_rotation*: Rotation of time ticklabels.
- *date_locator*: Major locator for dates.
- *date_minor_locator*: Minor locator for dates.
- *date_nominor*: Suppress minor ticks for dates.
- *date_nmax_ticks*: Max number of ticks for dates
- *nodate*: Time axis must not be formatted as a time axis [default: False]

Misc settings:

- *figure*: Figure number.
- *figsize*: Initialize the figure with this size.
- *subplots_adjust*: Dictionary sent to `subplots_adjust()`. You can also use keyparams 'left', 'right', 'top', 'bottom', 'wspace', 'hspace' !
- *sa*: Alias for subplots_adjust.
- *bgcolor*: Background axis color.

- *axes_rect*: [left, bottom, width, height] in normalized (0,1) units to create axes using `axes()`.
- *axes_<keyword>*: <keyword> is passed to `axes()`.
- *title*: Title of the figure [defaults to var.long_name or ‘’]
- *grid*: Plot the grid [default: True]
- *dayhl*: Add day highlithing [default: False]
- *figtext*: figtext Add text at a specified position on the figure. Example: figtext=[0,0,’text’] add a ‘text’ at the lower left corner, or simply figtext=’text’.
- *anchor*: Anchor of the axes (useful when resizing) in [’C’, ‘SW’, ‘S’, ‘SE’, ‘E’, ‘NE’, ‘N’, ‘NW’, ‘W’].
- *logo*: Add a logo to the figure [default: False]. logo can be a file name.
- *show*: Display the figure [default: True]
- *savefig*: Save the figure to this file.
- *savefigs*: Save the figure into multiple formats using `savefigs()` and ‘savefigs’ as the prefix to the files.
- *autoresize*: Auto resize the figure according axes (1 or True), axes+margins (2). If 0 or False, not resized [default: False=2].
- *key*: Add a key (like ‘a)’) to the axes using add_key is different from None [default: None]
- *close*: Close the figure at the end [default: False]
- *title_<keyword>*: <keyword> is passed to `title()`
- *key_<keyword>*: <keyword> is passed to `add_key()`
- *logo_<keyword>*: <keyword> is passed to `add_logo()`
- *figtext_<keyword>*: <keyword> is passed to `figtext()`
- *savefig_<keyword>*: <keyword> is passed to `savefig()`
- *savefigs_<keyword>*: <keyword> is passed to `savefigs()`
- *grid_<keyword>*: <keyword> is passed to `grid()`

All other keyparam are passed to `quiver()`

ellipsis(*xpos, ypos, eAXIS, eaxis=None, rotation=0, sign=0, np=100, gobj=None, scale=1.0, fill=False, sign_usearrow=True, axes=True, sign_usecolor=False, sign_usewidth=False, units=None, key=True, key_align=’top’, key_orientation=’tangential’, key_position=1.1499999999999999, key_format=’%(value)g %(units)s’, **kwargs*)

Create a dicretized ellipsis

- **xpos**: X position of the ellipsis.
- **ypos**: Y //
- **eAXIS**: Length of grand axis
- *eaxis*: Length of little axis [default: eAXIS]
- *rotation*: Rotation angle in degrees of grand axis from X axis [default: 0.]
- *sign*: Indicate a direction of evolution along the ellipsis using a variable line width and/or color [default: 0]. If 0, nothing is done. If positive or negative, it defines the direction relative the trigonometric convention.
- *sign_usearrow*: Use an arrow to indicate sign [default: True]
- *sign_usewidth*: Use width variation to indicate sign [default: False]
- *sign_usecolor*: Use color variation (from grey to default color) to indicate sign [default: False]
- *scale*: Scale factor to apply to grand and little axes [default: 1.]
- *np*: Number of discretized segments [default: 100]
- *gobj*: Graphical object on which to plot the ellipsis (can be a Basemap object) [default: ~matplotlib.pyplot.gca()]. If map object, eAXIS and eaxis are supposed to be in kilometers.

add_colorbar(*var=None, sm=None, horizontal=False, position=None, cmap=None, drawedges=False, cax=None, fig=None, figsize=None, frameon=None, levels=None, barwidth=None, barmargin=0.10000000000000001, norm=None, **kwargs*)
Standalone colorbar

scolorbar(**args, **kwargs*)
Shortcut to `add_colorbar()`

rotate_tick_labels(*angle, vertical=0, *args, **kwargs*)
Rotate labels along an axis

- **angle**: Angle in degrees OR False.
- *vertical*: On vertical axis if not 0 [default: 0]
- Other arguments are passed to `setp()`

rotate_xlabels(*angle, *args, **kwargs*)
Shortcut to `xrotate()`

rotate_ylabels(*angle, *args, **kwargs*)
Shortcut to `yrotate()`

scale_xlim(**args, **kwargs*)
Shortcut to `xscale()`

scale_ylim(**args, **kwargs*)
Shortcut to `yscale()`

xhide(*choice=True, **kwargs*)
Hide or not an axis

- **choice**: What to do [default: True]

yhide(*choice=True, **kwargs*)
Hide or not an axis

- **choice**: What to do [default: True]

xdate(*rotation=45.0, **kwargs*)
Consider x axis as dates

- *tz*: Time zone.
- *auto*: Auto Scaling [default: True]
- *rotation*: Rotation angle of tick labels. If None, automatic [default: None]
- *fmt*: Date format.
- *locator*: Major locator. Can be within ['year','month','Weekday','day','hour','minute','second'] or be like matplotlib.dates.MonthLocator().
- *minor_locator*: Minor locator.
- *nominor*: Do not try to add minor ticks [default: False]
- *nmax_ticks*: Maximal number of ticks
- *locator_<keyword>*: <keyword> is passed to locator if locator is a string. If locator = 'month', locator = MonthLocator(locator_<keyword>=<value>).
- *minor_locator_<keyword>*: Same with minor_locator.

ydate(*rotation=0.0, **kwargs*)
Consider y axis as dates

- *tz*: Time zone.
- *auto*: Auto Scaling [default: True]
- *rotation*: Rotation angle of tick labels. If None, automatic [default: None]
- *fmt*: Date format.

- *locator*: Major locator. Can be within ['year','month','Weekday','day','hour','minute','second'] or be like matplotlib.dates.MonthLocator().
- *minor_locator*: Minor locator.
- *nominor*: Do not try to add minor ticks [default: False]
- *nmax_ticks*: Maximal number of ticks
- *locator_<keyword>*: <keyword> is passed to locator if locator is a string. If locator = 'month', locator = MonthLocator(locator_<keyword>=<value>).
- *minor_locator_<keyword>*: Same with minor_locator.

add_logo(*file='logo_actimar.gif', axes=None, loc='upper left', scale=None, alpha=1*)
Add a logo to the figure

- *file*: File of the image [default: 'logo_actimar.gif']
- *axes*: Axes of the image (overwrites loc and scale keywords).
- *loc*: Position of the image (use words in 'lower', 'upper', 'left', 'right', 'center') [default: 'upper left']
- *scale*: Scale the image. By default, it is auto scale so that the logo is never greater than 1/4 of the width ir the height of the figure.
- *alpha*: Alpha transparency [default: 1]

wedge(*direction, width=18, fig=None, axes=None, figsize=(4, 4), shadow=False, shadow_alpha=0.5, circle=False, strength=None, strength_cmap=None, from_north=False, center=True, center_radius=0.040000000000000001, scale=1.0, **kwargs*)
Plot a wedge

- **direction**: Direction of the wedge in degrees from north
- *width*: Base width of the wedge in degrees [default: 20]
- *shadow*: Add a shadow [default: False]
- *shadow_alpha*: Alpha transparency of the shadow [default: .5]
- *circle*: Add a circle arround the wedge [default: True]
- *shadow_<keyword>*: <keyword> is passed to `matplotlib.patches.Shadow` class.
- *circle_<keyword>*: <keyword> is passed to `matplotlib.patches.Circle` class.

add_time_mark(*when, xlim=None, ylim=None, shadow=False, shadow_alpha=0.29999999999999999, shadow_offset=0.029999999999999999, label='Current time', **kwargs*)
Plot a time mark along a time axis

xi2a(*inch, data=None*)
Convert from inch to X axis coordinates

yi2a(*inch, data=None*)
Convert from inch to Y axis coordinates

yi2f(*inch*)
Convert from inch to relative height of figure

xi2f(*inch*)
Convert from inch to relative width of figure

gobjs(**args, **kwargs*)
Store and retreive graphic objects of the current axes

add_key(*key, pos=1, fmt='%s)', ax=None, **kwargs*)
Add a key (like "a)") to current axes

- **key**: A string or a integer. If a integer is given, it converted to letter (1->'a').
- *fmt*: Format the string [default: '%s)']
- *pos*: Position with 1='top left', 2='top right', 3='bottom right', 4='bottom left'. If negative, push it outside axis box.

- *ax*: Axes to use [default: `gca()`]

colorbar(*pp=None, vars=None, drawedges=False, levels=None, colorbar_horizontal=False, colorbar=True, colorbar_visible=True, colorbar_position=None, units=None, standalone=False, cax=None, extend='neither', **kwargs*)

Colorbar:

- *colorbar*: Plot the colorbar [default: True]
- *colorbar_horizontal*: Colorbar is horizontal [defaults: False]
- *colorbar_position*: To change the default position. Position is relative to the SPACE LEFT in the form (center,width) with values in [0,1] [defaults: False]
- *colorbar_visible*: Colorbar is visible [defaults: True]
- *colorbar_<keyword>*: <keyword> is passed to `colorbar()`
- *units*: Indicate these units on along the colorbar, else it guessed from the variable or suppressed if value is False [default: None]

hldays(*color=(0.90000000000000002, 0.90000000000000002, 0.90000000000000002), y=False, ax=None, zmin=None, zmax=None, tmin=None, tmax=None, **kwargs*)

Highlight different days with a different background color

- *color*: Background color [default: (.95,.95,.95)]
- *y*: Work on Y axis [default: False]
- *ax*: Axes to work on [default: `gca()`]
- Other keyparam are passed to `fill()`.

stick(*uu, vv, polar=False, degrees=False, ycenter=None, color='k', alpha=1, quiverkey=True, units=None, cmap=None, nodate=False, **kwargs*)

Plot a line of vectors

- **uu**: X coordinates of vectors OR length if polar=True
- **vv**: Y coordinates of vectors OR angle if polar=True

Vectors:

- *xaxis*: Use this X axis.
- *polar*: Consider input as polar instead of cartesian coordinates [default: False]
- *degrees*: If polar=True, angle (vv) is in degrees instead of radians [default: False]
- *color*: Color of arrows [default: 'k']
- *color*: Alpha transparency of arrows [default: 1]
- *quiver_<keyword>*: <keyword> is passed to `quiver()`

Quiver key:

- *quiverkey_pos*: Position of key for arrow [default: (0.0,1.02)]
- *quiverkey_text*: Text or format with variables 'value' and 'units' [default: '%(value)g %(units)s'].
- *quiverkey_value*: Numeric value for key (used by text) [default: None = deduced from cdms var]
- *quiverkey_units*: Units for key (used by text) [default: None = deduced from cdms var]
- *quiverkey_<keyword>*: <keyword> is passed to `quiverkey()`

Axis decoration:

- *[x/y]title*: Main label of the axis.
- *[x/y]label*: Same as x/ytitle.
- *[x/y]hide]*: Hide x/ytick labels.
- *[x/y]rotation*: Rotation of x/ytick labels (except for time).
- *[x/y]strict*: Strict axis limit.

- *[x/y]lim*: Override axis limit.
- *[x/y]min/max*: Override axis limit by x/ylim.
- *[x/y]minmax/maxmin*: Set maximal value of x/ymin and minimal value or x/ymax.
- *[x/y]nmax*: Max number of ticks.
- *x/yfmt*: Numeric format for x/y axis if not of time type.
- *date_rotation*: Rotation of time ticklabels.
- *date_locator*: Major locator for dates.
- *date_minor_locator*: Minor locator for dates.
- *date_nominor*: Suppress minor ticks for dates.
- *date_nmax_ticks*: Max number of ticks for dates
- *nodate*: Time axis must not be formatted as a time axis [default: False]

Misc settings:

- *figure*: Figure number.
- *figsize*: Initialize the figure with this size.
- *subplots_adjust*: Dictionary sent to `subplots_adjust()`. You can also use keyparams 'left', 'right', 'top', 'bottom', 'wspace', 'hspace' !
- *sa*: Alias for subplots_adjust.
- *bgcolor*: Background axis color.
- *axes_rect*: [left, bottom, width, height] in normalized (0,1) units to create axes using `axes()`.
- *axes_<keyword>*: <keyword> is passed to `axes()`.
- *title*: Title of the figure [defaults to var.long_name or '']
- *grid*: Plot the grid [default: True]
- *dayhl*: Add day highlithing [default: False]
- *figtext*: figtext Add text at a specified position on the figure. Example: figtext=[0,0,'text'] add a 'text' at the lower left corner, or simply figtext='text'.
- *anchor*: Anchor of the axes (useful when resizing) in ['C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'].
- *logo*: Add a logo to the figure [default: False]. logo can be a file name.
- *show*: Display the figure [default: True]
- *savefig*: Save the figure to this file.
- *savefigs*: Save the figure into multiple formats using `savefigs()` and 'savefigs' as the prefix to the files.
- *autoresize*: Auto resize the figure according axes (1 or True), axes+margins (2). If 0 or False, not resized [default: False=2].
- *key*: Add a key (like 'a)') to the axes using add_key is different from None [default: None]
- *close*: Close the figure at the end [default: False]
- *title_<keyword>*: <keyword> is passed to `title()`
- *key_<keyword>*: <keyword> is passed to `add_key()`
- *logo_<keyword>*: <keyword> is passed to `add_logo()`
- *figtext_<keyword>*: <keyword> is passed to `figtext()`
- *savefig_<keyword>*: <keyword> is passed to `savefig()`
- *savefigs_<keyword>*: <keyword> is passed to `savefigs()`
- *grid_<keyword>*: <keyword> is passed to `grid()`

All other keyparam are passed to `quiver()`

xscale(**args, **kwargs*)
Scale xlim using factor and auto_scale

- **factor**: Relative factor with 1. meaning no change
- *keep_min/keep_max*: Kee min/max during scaling [default: False]

Return: New xlim()

yscale(**args, **kwargs*)
Scale ylim using factor and auto_scale

- **factor**: Relative factor with 1. meaning no change
- *keep_min/keep_max*: Kee min/max during scaling [default: False]

Return: New ylim()

xrotate(*angle, *args, **kwargs*)
Rotate xticklabels

See: `rotate_tick_labels()`

yrotate(*angle, *args, **kwargs*)
Rotate yticklabels

See: `rotate_tick_labels()`

add_grid(*gg, color='k', borders=True, centers=False, m=None, linecolor=None, linewidth=1, alpha=0.5, linestyle='solid', zorder=100, marker='o', markersize=4, facecolor=None, edgecolor=None, label_borders='Grid cells', label_centers='Grid points', samp=1, **kwargs*)
Add a a 1D or 2D grid to a plot

- **gg**: A cdms grid OR a (lon,lat) tuple of axes OR a cdms variable with a grid
- *borders*: Display cell borders
- *centers*: Display cell centers
- *samp*: Undersampling rate
- *m*: A basemap instance to plot on
- *color*: Default color (or *c*)
- *linecolor*: Color of the lines (or *lc*) [defaults to *color*]
- *linestyle*: Line style (or *ls*)
- *edgecolor*: Edge color of marker (or *ec*) [defaults to *color*]
- *facecolor*: Face color of marker (or *fc*) [defaults to *color*]
- *marker*: Type of marker
- *markersize*: Size of markers (or *s*)
- *alpha*: Alpha transparency (or *a*)
- *zorder*: Order of the grid in the stack (100 should be above all objects)

savefigs(*basename, png=True, pdf=False, verbose=True, dpi=(80, 100), nodots=True, **kwargs*)
Save a figure in multiple formats (optimized for sphinx doc generator)

- **basename**: File name without suffix
- *png*: If True, save to png format
- *pdf*: If True, save to pdf format
- *verbose*: If True, print file names that are created
- Other keywords are passed to `savefig()`

> **Warning:** Dots ('.') in figure name are converted to dashes ('-') for compatilibity reasons with latex. It optimized for the sphinx doc generator and may NOT BE SUITABLE FOR YOU. In this case, use `savefig()` instead.

bar(*var, width=1.0, align='center', along=None, vertical=False, offset=None, label=None, nodate=False, axis_setup=True, profile=False, **kwargs*)
Quick plot an 1D cdms variable as bar plot

- **var**: A 1D cdms variable

Bar:

- *along*: A letter within 'x','y','z','t' to select the axis ; None select the LAST AXIS (if 'tzyx', it plots along 'x') ; it reduces dimensions by averaging along successive axes [default: None]
- *color*: Color of the bars .
- *edgecolor*: Color of the bar contours.
- *linewidth*: Width of the bar contours.
- *linestyle*: Style of the bar contours.
- *label*: Specify the label to shown in legend. It defaults to the long name of the variable. If False, _nolegend_ is set.
- *units*: Label of var's axis. It defaults to the units of the variable. If False, no label is drawn.

$decorate_axis

$_start_plot_ $_end_plot_

All other keywords are passed to the plot function (like color, linewidth, etc).

decorate_axis(*axis=None, vertical=0, date_rotation=None, date_fmt=None, date_locator=None, date_minor_locator=None, date_nominor=False, nodate=False, values=None, ax=None, **kwargs*)
Axis decoration:

- *[x/y]title*: Main label of the axis.
- *[x/y]label*: Same as x/ytitle.
- *[x/y]hide]*: Hide x/ytick labels.
- *[x/y]rotation*: Rotation of x/ytick labels (except for time).
- *[x/y]strict*: Strict axis limit.
- *[x/y]lim*: Override axis limit.
- *[x/y]min/max*: Override axis limit by x/ylim.
- *[x/y]minmax/maxmin*: Set maximal value of x/ymin and minimal value or x/ymax.
- *[x/y]nmax*: Max number of ticks.
- *x/yfmt*: Numeric format for x/y axis if not of time type.
- *date_rotation*: Rotation of time ticklabels.
- *date_locator*: Major locator for dates.
- *date_minor_locator*: Minor locator for dates.
- *date_nominor*: Suppress minor ticks for dates.
- *date_nmax_ticks*: Max number of ticks for dates
- *nodate*: Time axis must not be formatted as a time axis [default: False]

make_movie(*fig_pattern, outfile, delay=1, clean=False, verbose=False, windows=True*)
Make a movie from a series of figures

- **fig_pattern** Unix-like file pattern to select png figures.
- **outfile**: Output file.
- *delay**: Delay in seconds between two frames.
- *windows*: When output is a movie, choose basic mpeg for windows instead of mpeg4 codec.
- *clean*: If `True`, remove figures once the movies os created.

Usage

```
>>> make_movie('/home/toto/fig*.png', 'movie.mpg', delay=.5, clean=True)
```

taylor(*datasets, ref, labels=False, colors=None, units=None, **kwargs*)
Plot a Taylor diagram after computing statistics

- **datasets**: One or several arrays that must be evaluated against the reference. Best is to provide `cdms2` variables with an appropriate *long_name* attribute to set labels automatically.
- **ref**: Reference array or the same shape as all `datasets` arrays.
- *labels*: Label of the points. Obviously, if there are several points, there must have several labels. If `None`, it tries to get it from the *long_name* attribute. If `False`, it doesn't display it.
- *reflabel*: Name of the reference label. It defaults to the *long_name* attribute or to `"Reference"`.
- *size*: Size(s) of the markers in points.
- *colors*: Color of the points. It can be a list to set a different color for each pair.
- *label_colors*: Color of the labels. If `"same"`, color is taken from `colors`.
- *label_<keyword>*: <`keyword`> is passed to `text()` for plotting the labels.
- *symbols*: Symbols for the points (default to `"o"`).
- *point_<keyword>*: <`keyword`> is passed to `plot()` for plotting the points.
- *reflabel*: Label of the reference.
- *refcolor*: Color of the reference.
- *title*: A string for the general title.
- *savefig*: Save figure to `savefig`.
- *savefig_<keyword>*: <`keyword`> is used for saving figure.
- *savefigs*: Save figure to `savefig` using `savefigs()`.
- *savefigs_<keyword>*: <`keyword`> is used for saving figure using `savefigs()`.

Example:

```
>>> taylor(var_model, var_obs, title='Single point', reflabel='Observations')
>>> taylor(([var_model1,var_model2], var_ref)
```

See Also:

"*Diagramme de Taylor*".

vtaylor(*data, ref=None, ref_color='red', color=None, label=None, title=None, xoffset=5, yoffset=-2, show=False, stdmax=None, savefig=None, **kwargs*)
Plot a Taylor diagram using `vcs` module from CDAT

- **data**: A single or several points where a point is a pair of (std, corr).
- *ref*: Reference value (standard deviation of observations).
- *color*: Color of the points. It can be a list to set a different color for each pair.
- *label*: Label of the points. Obviously, if there are several points, there must have several labels.
- *x|yoffset*: Offset in font size unit of label position.
- *title*: A string or `False`.
- *savefig*: Save figure to `savefig`.
- *savefig_<keyword>*: `keyword` is used for saving figure.
- Other keywords are passed to `plot()`.

Example:

```
>>> taylor([21.,.83], ref=30., title='Single point')
>>> taylor(([[[21.,.83], [33., .7]], ref=28., color=['blue', 242], label=['Point 1', 'Point2'])
```

> **Warning:** This function uses the `vcs` module for plotting. Therefore, yan can't alter the plot with `matplotlib` functions.

get_cls(*n, colors=, ['k', 'b', 'r', 'c', 'm', 'y'], linestyles=, ['-', '–', '-.', ':']*)
Get a list of string argument for `plot()` to specify the color and the linestyle

- **n**: Length of the list
- *colors*: Colors on which to cycle
- *linestyles*: Linestyles on which to cycle

dtaylor(*stats, stdref, labels=None, **kwargs*)
Directly plot a Taylor diagram

- **stats**: Statistics is the form `[std, corr]` or `[[std1,corr1],[std2,corr2],...]`
- **stdref**: Standard deviation of reference
- *labels*: Label of the points. Obviously, if there are several points, there must have several labels. If `None`, it tries to get it from the *long_name* attribute. If `False`, it doesn't display it.
- *reflabel*: Name of the reference label. It defaults to `"Reference"`.

$_taylor_

2.1.5 `actimar.misc.color` — Couleurs et palettes

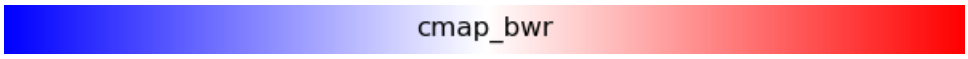
Variables and utilities about colors and color maps

cmap_custom(*colors, name='mycmap', ncol=256, ranged=False, **kwargs*)
Quick colormap creation

- **colors**: Like ((color1, position1),(color2, position2), etc...) or dict(red=((pos1,r1a,r1b), (pos2,r2a,r2b)),etc...)

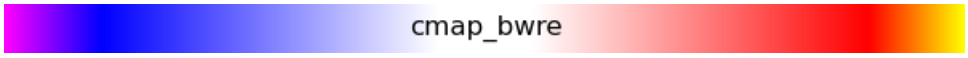
cmap_bwr(*wpos=0.5, wcol='w'*)
Blue->white->red colormap

- *white*: relative position of white color in the map [default: 0.5]



cmap_bwre(*wpos=0.5, gap=0.10000000000000001, wcol='w'*)
Returns a violet->blue->white->red colormap->yellow

- *white*: relative position of white color in the map [default: 0.5]
- *gap*: Relative width of the pure central white gap [default: 0.1]

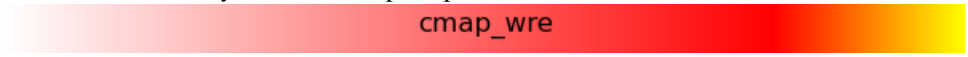


cmap_br(*sep=0.5*)
Blue->red colormap

- *sep*: relative position blue/red transition the map [default: 0.5]

cmap_wr()
White->red colormap

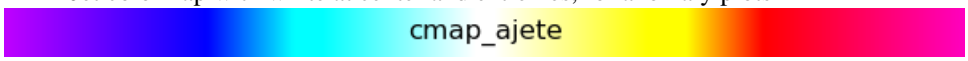
cmap_wre()
White->red->yellow colormap for positive extremes



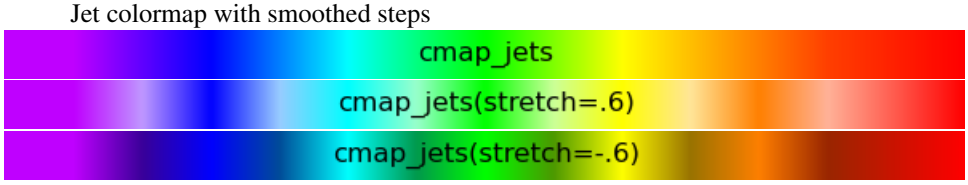
cmap_bathy(*start=0.0, stop=1.0*)
Colormap for bathymetry maps

cmap_jete()
Jet colormap with extremes

cmap_ajete(*w=0.02*)
Jet colormap with white at center and extremes, for anomaly plots



cmap_jets(***kwargs*)
Jet colormap with smoothed steps

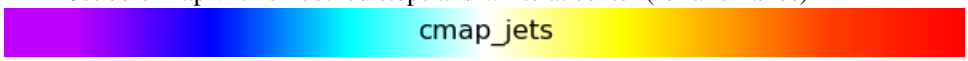


See Also:

cmap_smoothed_regular_steps()

cmap_wjets(*wcol=(1, 1, 1), **kwargs*)
White jet colormap with smoothed steps

cmap_ajets(***kwargs*)
Jet colormap with smoothed steps and white at center (for anomalies)



cmap_smoothed_steps(*colors, stretch=-0.59999999999999998, name='cmap_css'*)
Smoothed steps

- *colors*: Central positions for each colors [(r1,g1,b1,pos1),...]

See Also:

cmap_steps() cmap_regular_steps() cmap_smoothed_regular_steps()

cmap_smoothed_regular_steps(*colors, steptype='center', **kwargs*)
Smoothed regular steps

- **colors**: [(r1,g1,b1),...]
- other keywords are passed to cmap_smoothed_steps()

See Also:

cmap_steps() cmap_smoothed_steps() cmap_regular_steps() (cmap_regular_jets() for an example)

cmap_ss(**args, **kwargs*)
Shortcut to cmap_smoothed_steps()

cmap_steps(*cols, stretch=-0.59999999999999998, name='cmap_steps'*)
Colormap by steps

- **cols**: [(col1,pos1),(col2,pos2),...]
- *stretch*: Color darkening (<0) or whitening within steps [default: -.6]

See Also:

cmap_regular_steps() cmap_smoothed_steps() cmap_smoothed_regular_steps()

cmap_regular_steps(*colors, **kwargs*)
Colormap by regular steps

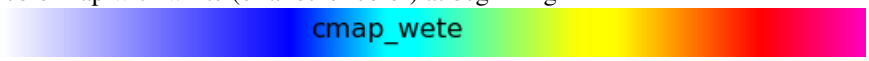
- **cols**: [col1,col2,...]
- *stretch*: Color darkening (<0) or whitening within steps [default: -.6]
- *steptype*: 'center', 'stair' or 'bounds' [default: 'center']
- other keywords are passed to `cmap_steps()`

See Also:

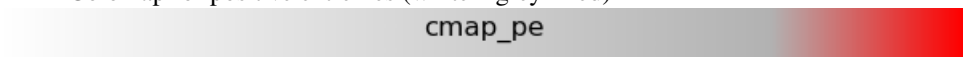
`cmap_steps()` `cmap_smoothed_steps()` `cmap_smoothed_regular_steps()`

cmap_rs(**args, **kwargs*)
Shortcut to `cmap_regular_steps()`

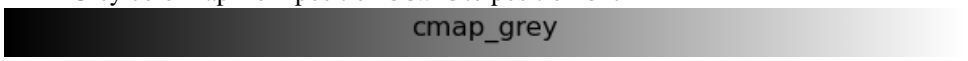
cmap_wjet(*first_color=(1, 1, 1)*)
jet colormap with white (or another color) at beginning



cmap_pe(*red=0.80000000000000004*)
Colomap for positive extremes (white->grey->red)



cmap_grey(*start=0, end=1.0*)
Grey colormap from position `start` to position `end`



show_cmap(*cmap*)
Alias for `plot_cmap()`

cmaps_mpl()
List available Matplotlib colormaps

See Also:

`get_cmap()`

cmaps_gmt()
List available GMT colormaps

cmap_gmt(*name*)
Get a colormap from GMT

cmaps_act()
List available Actimar colormaps

See Also:

`get_cmap()`

print_cmaps_gmt()
List available gmt colormaps

darken(*c, f*)
Darken a color 'c' by a factor 'f' (max when f=1)

whiten(*c, f*)
Whiten a color 'c' by a factor 'f' (max when f=1)

to_shadow(*c, att=0.29999999999999999*)
Return the shadow color of a color

- **c**: Color.
- *att*: Attenuation factor [default: .3]

class StepsNorm(*levels, log=False, **kwargs*)
Bases: matplotlib.colors.Normalize

Normalize a given value to the 0-1 range on a stepped linear or log scale

autoscale(*A*)
Set *vmin*, *vmax* to min, max of *A*.

autoscale_None(*A*)
autoscale only None-valued vmin or vmax

inverse(*value*)

scaled()
return true if vmin and vmax set

RGB
Returns an *RGB* tuple of three floats from 0-1.

arg can be an *RGB* or *RGBA* sequence or a string in any of several forms:

1. a letter from the set 'rgbcmykw'
2. a hex color string, like '#00FFFF'
3. a standard name, like 'aqua'
4. a float, like '0.4', indicating gray on a 0-1 scale

if *arg* is *RGBA*, the *A* will simply be discarded.

cmap_srs(**args, **kwargs*)
Shortcut to cmap_smoothed_regular_steps()

cmap_rs(**args, **kwargs*)
Shortcut to cmap_regular_steps()

cmap_linear(*c, **kwargs*)
Basic linear map between colors

plot_cmap(*cmap, ax=None, figsize=(5, 0.25), show=True, aspect=0.050000000000000003, title=None, sa={'top': 1, 'right': 1, 'bottom': 0.0, 'left': 0.0}, savefig=None, savefigs=None, **kwargs*)
Display a colormap

plot_cmaps(*cmaps=None, figsize=None, show=True, savefig=None, nrow=40, savefigs=None, **kwargs*)
Display a list of or all colormaps

get_cmap(*cmap*)
A simple way to get an Actimar, GMT or MPL cmap

Example

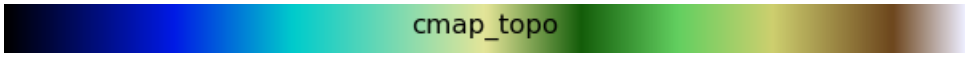
```
>>> get_cmap('jet')          # Matplotlib
>>> pylab.jet()              # Matplotlib
>>> pylab.get_cmap('jet')    # Matplotlib
>>> get_cmap('cmap_grey')    # Actimar
>>> cmap_grey(start=.2)      # Actimar
>>> get_cmap('gmt_gebco')    # GMT
>>> cmap_gmt('gebco')        # GMT
```

See Also:

cmap_gmt()

cmap_land(*start=0.0, stop=1.0*)
Colormap for land maps

cmap_topo(*start=0.0, stop=1.0, id='cmap_topo', over='k', under='#eeeeff', bad='0.5'*)
Colormap for bathy+land maps



See Also:

cmap_bathy() cmap_land()

auto_cmap_topo(*varminmax=(0.0, 1.0), **kwargs*)

Adjusted cmap_topo() colormap so that altitude 0 fall on center of full colormap

- **varminmax**: Define the min and max of topo ; it can be either :
  - a topo variable (array) from which min and max are computed
  - a tuple of (min, max) fo topo

2.1.6 actimar.misc.io — Lecture/ecriture de fichiers de divers types

Outils pour les entrees/sorties

class col_printer(*columns, headsep=True, align='left', left=None, right=None, top=None, bottom=None, frame=None, colsep=' ', print_header=True, file=None*)

Class to print formatted columns with header and frame

Note: Obviously, you must use monospaced fonts

Example:

```
>>> p = col_printer(['Year',6,'%i'],['Value',7,'%4.1'], headsep='=',align='center',frame=True)
>>> p(2000,25.9)
>>> p(2001,23.7)
...
>>> del p # prints the footer (and closes the file in file mode)
```

bottom()

Print bottom frame and close file if needed

close()

Close the file (if file mode)

header()

Print header (stop + header text + header separator)

headsep()

Print header separator line

top()

Print top frame

read_nc_forecast(*var_name, sort=None, file_pattern='*.nc', directories='.', date_pattern=None, lon=':', lat=':', time=':', z=':', daily_slice=None, quiet=False, day=0, verbose=False, new_grid=None, first=False, regular=True, hsamp=1, daily_frequency=None, missing_value=None, daily_average=False, day_bounds=None*)

Read a variable from MARS Previd1 files

- **var_name**: Name of the variable
- *day*: Select the current day if 0, else one of the following days. Must between -1 and 1 [default: 0]
- *suffix*: Suffix (like "pipo.<suff>.nc")
- *lon/lat/z/time*: Range selectors
- *day_bounds*: Extend/restrict the selected day range. For example, use ((6,cdtime.Hour),(-6,cdtime.Hour)) to restrict day from 6am to 17am included. If day is extended, it implies an time_average=True.
- *daily_average*: Perform an average over the selcted day.
- *daily_frequency*: Intraday averages at this frequency (for example, 2 means 12h averages). It create proper time bounds.
- *first*: Read only the first file found [default: False]. Becomes True if time is a slice.

- *new_grid*: Regrid on this rectangular grid
- *quiet*: Quiet mode
- *verbose*: More verbose
- *regular*: Fill gaps with missing value to have a regular time axis

class `CachedRecord`(*buoy, time_range, **kwargs*)

Abstract class for managing cached records

** It cannot be used by itself **

The following class variables must be defined:

- _cache_file: cache file name
- _time_range: ('2000-12-01 12:00','2005-10-01','co') OR (1, 'day', 'cc')
- _var_info: (('lon','Longitude', 'degrees east', None),('hs','Wave Height', 'm', (0., 20)),)
- _station_info: (['Start Bay', '2007-12-25'],['Long Bay', '2004-05-18'])
- _dt: (1800, cdtime.Seconds)
- _time_units: 'seconds since 2000-01-01'

The following methods must be defined:

- _load_from_source_:

`check_cache`(*time_range=None*)

Update the cache

`get`(*var_name, time_range=None*)

`load`(*time_range*)

Check if time_range is included in in-memory variables, else, check cache and load from it

`plot`(*var_name, time_range=None, **kwargs*)

`save`(*file_name, mode='w', warn=True*)

`show_variables`()

Print available variables

class `Shapes`(*input, m=None, inverse=False, clip=True, shapetype=None, min_area=None, sort=True, reverse=True, samp=1*)

A class to read shapefiles and return GEOS objects Inspired from basemap.readshapefile

Here are the conversion rules from shapefile to GEOS objects :

- Points and multipoints are interpreted as Points.
- Polylines are interpreted as LineString.
- Polygons are interpreted as Polygons

Parameters:

- **input**: Refers to a shapefile or is a shapes isntance ; if a shapefile, it assumes that <input>.shp contains points, multipoints, lines or polygons, and that <input>.dbf contains their attributes.
- *m*: A Basemap instance for converting shapefile coordinates to geos polygons and for plotting [default: None].
- *inverse*: Inverset the conversion with m [default: False].
- *clip*: If in the form `(xmin,ymin,xmax,ymax)`, clips to this box ; if a polygon like argument, it clips to this polygon (see `polygons()` for arguments). If simply `True` and *m* is present, it clips to the bounds of *m*.
- *min_area*: Minimal area to keep a polygon [default: None]
- *shapetype*: If 0, it must only deal with points ; if 1, only polylines ; if 2, only polygons (conversion 1<->2 is automatic) [default: None].

`clip`(*zone, **kwargs*)
Clip to zone

`get_data`(*key=None*)
Get the numeric version of the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

`get_map`()
Return the associated basemap instance if set

`get_points`(*key=None, split=True*)
Get all the points from all the shapes as a tuple (x,y)

`get_projs`()
Get the list on projections (proj instance, inverse flag)

`get_shapes`(*key=None*)
Get the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

`get_type`()
Return the type of shapes

- 0 = Points,
- 1 = LineStrings = PolyLines
- 2 = Polygons

`plot`(*select=None, ax=None, fill=None, fillcolor=None, points=False, m=None, show=False, **kwargs*)
Plot shapes

- *select*: argument for selecting shapes in the list [defaul: None].
- *fill*: Force filling (True/False), else guessed from shpe type, ie filling for polygons only [default: None]
- *ax*: Axes instance [default: None]
- Other keyparams are used for plotting (lines or polygons)

`project`(*proj, inverse=False*)
Project shapes using proj

- **proj**: A Basemap instance or pure projection instance (from Basemap)
- *inverse*: Inverse projection [default: False]

`resol`(*deg=True*)
Compute the mean "resolution" of the shapes based on the first shape

- **deg**:
  - if `False`: return a resolution in meters has a the median distance between points
  - if `True`: return the median distance between points as a resolution in degrees `(xres,yres)`

`sort`(*reverse=True*)
Sort shapes according to their surface

- *reverse*: If True, greater polygons are first [default: True]

`sorted`()

`xy`(*key=None*)
Shortcut to `get_points(split=false)`

class `XYZ`(*xyz, m=None, units=None, long_name=None, transp=True, trans=False, magnet=0, rsamp=0, id=None, **kwargs*)
Class to manipulate xyz data (randomly spaced)

- **xyz**: It can be either
  - a .xyz ascii file, or a netcdf/grd file with variables `x`, `y` and `z`,
  - a (x,y,z) tuple,
  - a (3, npts) array,
  - another XYZ instance.

- *long_name*: Long name
- *units* Units
- *tranform*: It can be either
  - a factor applied to z at initialisation
  - a fonction that takes z as the only argument to filter its data.
- ***exc*: Polygons to exclude data (see `exclude()`).** Several polygons must be passed as a tuple (poly1, poly2, ...).
- ***sel*: Polygons to select data (see `select()`).** Several polygons must be passed as a tuple (poly1, poly2, ...).
- load_<keywords>: keywords are passed to `numpy.loadtxt()`
- Other keywords are set as atrributes.

Slicing:

- len(obj): number of xyz data
- obj[1:3]: [(x0,y0,z0),(x1,y1,z1)]

Operations :

```
>>> xyz += 2
>>> xyz3 = xyz1 + xyz2/2. # concatenation
```

clip(*zone=None, margin=None, inverse=False, mask=False, id=None, **kwargs*)

Geographical selection of part of the data

- *zone*: (xmin,ymin,xmax,ymax) or a float/int a complex polygon (see `polygons()`).
- *margin*: Margin around `zone` relative to the resolution (see `resol()`)
- *inverse*: Inverse the selection.
- *mask*: `zone` must be interpreted as a mask

consolidate()

Apply radius undersampling and all exclusions and selections to data and reset them

contains(*x, y*)

Check if one or several points are within a the convex hull

- **x,y**: X,Y positions as floats or lists or an `numpy` arrays.

copy()

Deep copy

exclude(**zones*)

Add one or more zones where data are not used.

A zone can be :

- an argument to `polygons()` to get a `_geoslib.Polygon` instance,
- another :class:XYZ‘ instance from which the convex hull (see `hull()`) is used as a delimiting area

Usage

```
>>> xyz.exclude([[-8,43],[-5.5,43],[-6,45.]],[[-10,45],[-7,47],[-10,49.]])
>>> xyz.exclude(polygon1,polygon2)
>>> xyz.exclude(xyz1,[-5,42,-3,48.])
```

See Also:

`select()` `exclusions()`

exclusions()

Get all exclusion polygons as a tuple

get_magnet()

Get the magnet integer attribute

Note: Useful only for mixing `XYZ` instances

get_res(*deg=False, auto=None*)
Get the mean X and Y resolutions in meters or degrees

get_rsamp()
Get the radius sampling

get_transp()
Get the transparency boolean attribute

Note: Useful only for mixing XYZ instances

grid(*res=None, xmin=None, xmax=None, ymin=None, ymax=None, relres=0.5, degres=False, id='xyz_grid'*)
Rectangular grid based on x/y positions and resolution

- *res*: Resolution. It can be:
  - a float where then xres=yres=res
  - a tuple as (xres,yres)
  - else it is guessed using get_res() (and maybe resol())‘ and multiplied by relres
- *relres*: Relative resolution factor applied to res when resolution is guessed (res=None)
- *degres*: When res is explicitly given, it interpreted as degrees is degres is True.
- *xmin,xmax,ymin,ymax*: Bounds of the grid. If not specified, bounds of the dataset are used (see xmin(), etc).

Note: Resolutions are adjusted when they are not mutiple of grid extensions (slightly decreased). Therefore, extensions of the grid are always preserved.

See Also:

resol(), togrid()

hull(*out='xy', mask=True*)
Return the convex hull

Returns Depends on out

- "xy": (xhull, yhull)
- "ind": indices of points
- "poly": _geoslib.Polygon instance

interp(*xyo, xyz=False, ext=True, nl=True, **kwargs*)
Interpolate to (xo,yo) positions using nat.Natgrid

- **xo**: Output X
- **yo**: Output Y
- *xyz*: If True, return a XYZ instance instead of a numpy array
- *ext*: Allow extrapolation
- *nl*: Nonlinear interpolation

Returns An XYZ instance

mask()
Get the current mask due to exclusion and selection polygons

See Also:

exclude() select()

plot(*size=5.0, color=None, alpha=1.0, masked_alpha=0.29999999999999999, masked_size=None, linewidth=0.0, show=True, savefig=None, savefigs=None, m=None, colorbar=True, title=None, units=None, cmap=None, mode='valid', zorder=100, masked_zorder=50, margin=2, xmin=None, xmax=None, ymin=None, ymax=None, xres=None, yres=None, **kwargs*)
Scatter plot of bathymetry

reset_exclusions()
Remove all exclusions

reset_rsamp()
Reset rsamp without affecting data

reset_selections()
Remove all selections

resol(*convex_hull_method='delaunay', exc=, [], deg=False*)
Return the mean resolution.
Algorithm: Median distances between facets of triangles

Returns (xres,yres)

save(*xyzfile, **kwargs*)
Save to a xyz file

- **xyzfile**: Output file name
- Other keywords are passed to `numpy.savetxt()`

select(**zones*)
Add one or more zone (polygons) where only these data are used
A zone is an argument to `polygons()` to get a `_geoslib.Polygon` instance.

Usage

```
>>> xyz.select([[-8,43],[-5.5,43],[-6,45.]],[[-10,45],[-7,47],[-10,49.]])
>>> xyz.select(polygon1,polygon2)
```

See Also:
`exclude()` `selections()`

selections()
Get all selection polygons as a tuple

set_magnet(*magnet*)
Set the magnet integet attribute. If set to `0`, no magnet effect.
Note: Useful only for mixing `XYZ` instances

set_res(*xres, yres=None*)
Set the resolution of the dataset
If `yres` is not, it is set to `xres`. When a value is negative, it is supposed to be in meters (not in degrees)

set_rsamp(*rsamp*)
Set the radius sampling If set to `0`, no sampling.

set_transp(*transp*)
Set the transparency boolean attribute
Note: Useful only for mixing `XYZ` instances

shadows()
Get the polygons defining the 'shadow' of this dataset.
It consists of a tuple of two elements:

- the convex hull as a polygon,
- a list of exclusion polygons that intersect the convex hull.

Therefore, a point in the shadow must be inside the convex hull polygon, and outside the exclusion polygons.

Returns (hull_poly, [exclusion_poly1,...])

tocfg(*cfg, section, param=None*)
Dump one or all parameters as options to a cfg section

- **cfg**: ConfigParser object
- **section**: Section of cfg
- *param*: A single or a list of parameter names

togrid(*grid=None, mask=False, cgrid=False, **kwargs*)
Interpolate to a regular grid

- **grid**: The output grid. It can be either:
  - a (x,y) tuple or a grid or a cdms variable with a grid,
  - `None`, thus guessed using `grid()`
- **mask**: It can be either:
  - `None`, `False` or `MV2.nomask`: no masking

– an array: this mask array is directly applied
– a Shapes instance (or ShoreLine) or a single char GSHHS resolution (and optionally 's' for Histolitt)
– a callable fonction so that `mask = thisfunc(mask, **kwmask)`
– a float: data with this value are masked

- *mask_<param>*: <param> is passed to `polygon_mask()` for evaluation of mask thanks to the polygons.
- *grid_<param>*: <param> is passed to `grid()`.
- *cgrid*: If `True`, returns bathy at U- and V-points, else at T-points
- Other keyparam are passed to `griddata()` for regridding.

Return: `(Zx,Zy)` OR Z depending on cgrid.

x(*mask=True*)
Get valid X positions

xmax(*mask=True*)

xmin(*mask=True*)

xy(*mask=True*)
Return coordinates as a (2, npts) array

- `xy()[0]`: X
- `xy()[1]`: Y

xyz(*mask=True*)
Return coordinates and data as a (3, npts) array

- `xy()[0]`: X
- `xy()[1]`: Y
- `xy()[2]`: Z

y(*mask=True*)
Get valid Y positions

ymax(*mask=True*)

ymin(*mask=True*)

z(*mask=True*)
Get valid Z values

zmax(*mask=True*)

zmin(*mask=True*)

zone(*poly=False, mask=True*)
Get xmin,ymin,xmax,ymax

- *poly*: if True, return zone as a Polygon instance

read_adcp(*file, time_units=None, no_mask=True, valid_range=None, select=None*)
Read an ADCP ascii file and return a list of cdms variables.

Header : 6 lines with last line giving name of variables. Then, for example: 2006 07 12 18 44 0.0 0.1474 0.1111 0.2222 0.3333

- **file**: File name
- *time_units*: Use these units, else use minutes since the begining.

Return: List of cdms variables

class **XYZMerger**(**datasets, **kwargs*)
Mix different bathymetries

append(*d*)
Append a dataset to the merger

clean()
Remove all current dataset

copy()

ids()

merge(***kwargs*)
Shortcut to xyz()

plot(*color=None, marker=None, mode='cluster', title='XYZ merger', show=True, colorbar=True, savefig=None, savefigs=None, legend=True, xmin=None, xmax=None, ymin=None, ymax=None, margin=5, xres=None, yres=None, **kwargs*)

- *alpha*: Alpha transparency:
  - applied to **all** points if mode="cluster"
  - applied to **hidden** points if mode="data"
- *mode*: Display mode:
  - **"cluster": Points from different datasets have different colors and markers,** and hidden points are transparent.
  - **"data": Points have the same marker, colors depends on Z value and hidden** points are masked.
- *marker*: Define a single or several markers to be used.
- *legend*: Show a legend if mode="cluster".
- *title*: Title of the plot.
- *m*: Basemap instance.
- *m_margin*: Margin for m, relative to the mean resolution (see XYZ.resol())
- *m_<keywords>*: Keywords are passed to map().
- Extra keywords are passed to XYZ.plot().

remove(*d*)
Remove a dataset from the merger

togrid(**args, **kwargs*)
Interpolate merged bathymetries to a grid

tolist()
Return the merger as a list of datasets

xyz(*mask=True, **kwargs*)
Merge current dataset

write_snx(*objects, snxfile, type='auto', mode='w', z=99, close=True*)
Write points, lines or polygons in a sinusX file

class **Logger**(*name, logfile=None, maxlogsize=0, maxbackup=0, cfmt='%(name)s, [%(levelname)-8s], %(message)s', ffmt='%(asctime)s: %(name)s, [%(levelname)-8s], %(message)s', asctime='%Y-%m-%d %H:%M', level='debug', colors=True, full_line=False*)
Class for logging facilities when subclassing. Logging is sent to console and optionally log file

- **name**: Name of the logger
- **logfile**: Log file
- *maxlogsize*: Maximal size of log file before rotating it
- *maxbackup*: Maximal number of rotated files
- *sfmt*: Format of log messages in log file
- *cfmt*: Format of log message in console
- *asctime*: Time format
- *level*: Initialize logging level (see set_loglevel())
- *colors*: Use colors when formatting terminal messages?
- *full_line*: Colorize full line or just level name?

See Also:

logging module

critical(*text*)
Send a critical message

debug(*text*)
Send a debug message

error(*text*)
Send an error message

get_loglevel()
Get the log level as integer

info(*text*)
Send a info message

set_loglevel(*level*)
Set the log level (DEBUG, INFO, WARNING, ERROR, CRITICAL)

warning(*text*)
Send a warning message

class **TermColors**()

disable()

format(*text, color='NORMAL'*)
Format a string for its color printing in a terminal

- **text**: simple string message
- *color*: color or debug level

2.1.7 actimar.misc.filters — Filtres numériques 1D/2D

Various filters

generic2d(*data, weights, fast=False, fill_value=None, min_valid=0*)
Generic 2D filter

- **data**: 2D variable.
- **weights**: Weights of the filter as 2D array of odd sizes.

shapiro2d(*data, **kwargs*)
Shapiro (121) 2D filter

- **data**: Data array
- Keywords are passed to generic2d()

gaussian2d(*data, nxw, nyw=None, **kwargs*)
Gaussian 2D filter

- **data**: Data array
- **nxw**: Size of gaussian weights array along X (and Y if nyw not given)
- *nyw*: Size of gaussian weights array along Y [default: nxw]
- Other keywords are passed to generic2d()

deriv(*data, axis=0, fast=True, fill_value=None, physical=True, lat=None*)
Derivative along a given axis

- **data**: Data array (converted to MV array if needed)
- *axis*: Axis on which the derivative is performed [default: 0]
- *fast*: Filled masked array before derivating, so use Numeric which is faster than MA or MV [*WARNING* default: True]
- *physical*: Try physical derivative, taking axis units into account [default: True]

- *lat*: Latitude for geographical deriviative to convert positions in degrees to meters

deriv2d(*data, direction=None, **kwargs*)
Derivative in a 2D space

- **data**: 2D variable
- *direction*: If not None, derivative is computed in this direction, else the module is returned [default: None]
- Other keywords are passed to deriv()

norm_atan(*var, stretch=1.0*)
Normalize using arctan (arctan(strecth*var/std(var))

- *stretch*: If stretch close to 1, saturates values [default: 1]

Return: Value in [-1,1]

running_average(*x, l, d=0, w=None, keep_mask=True*)
Perform a running average on an masked array. Average is linearly reduced near bounds, so that the input and output have the same size.

- **x**: Masked array
- **l**: Window size
- *d*: Dimension over which the average is performed (0)
- *w*: Weights (1...)
- *keep_mask*: Apply mask from x to output array

Example:

```
>>> running_average(x, l, d = 0, w = None, keep_mask = 1)
```

Returns Average array (same size as x)

2.1.8 actimar.misc.grid — Travail sur les grilles

actimar.misc.grid.misc — Divers

Grid utilities.

It deals with bounds, areas, interpolations...

See: *La gestion des grilles*.

isgrid(*gg, curv=None*)
Check if gg is a grid

- **gg**: A cdms grid
- *curv*: If True, restrict to curvilinear grids

get_resolution(*mygrid, lon_range=None, lat_range=None*)
Get the mean resolution of a grid

get_distances(*xxa, yya, xxb, yyb, geo=False*)
Find the distances between a series of points

- **xxa**: X coordinate of the first series
- **yya**: Y //
- **xxb**: X coordinate of the second series
- **yyb**: Y //
- *geo*: Suppose X is longitude and Y is latitude [default: False]

Return: Distances as an (nb,na) array.

get_closest(*xx, yy, xp, yp, geo=False, mask=None*)
Find the closest unmasked point on a grid and return indices

- **xx**: 1D or 2D X axis
- **yy**: 1D or 2D Y axis
- **zz**: masked array
- **xp**: X position of the point (float)
- **yp**: Y //
- *geo*: if True, force to treat the grid as geographical [default: False, unless detected as True].

Returns (i,j) 2-element tuple of indices along y and x

get_geo_area(*grid, mask=None*)
Compute cell areas on the a regular geographical grid.

- **grid** The grid
- *mask*: Force the use of this mask

Returns 2D array of areas in m^2

bounds2d(**xyaxes*)
Compute bounds on a rectangular grid.

- **xyaxes**: 2D arrays(ny,nx) (or 2x1D arrays)

Example

```
>>> xx_bounds,yy_bounds = bounds2d(xx,yy)
```

Returns xx(ny,nx,4),...

bounds1d(*xx*)
Compute bounds on a linear sequence or axis. It is based on genGenericBounds of CDAT.

get_axis(*gg, iaxis=0*)
A robust way to get an axis from a variable

- **var**: The variable or grid
- *iaxis*: The axis number [default: 0]

Returns The requested axis

get_grid(*gg, geo=True, intercept=True*)
Get a cdms grid from gg

- **g**: A cdms variable with a grid OR cdms grid OR a tuple like (xx,yy) where xx and yy are numpy arrays or cdms axes
- *geo*: Output grid must have longitude and latitude
- *intercept*: Raise an error in case of problem

get_grid_axes(*gg, raw=False*)
Get the (lat,lon) axes from a grid

- **gg**: A cdms grid or a tuple or axes
- *raw*: If True, return raw axes which are different from real axes with curvilinear grids

grid2d(**args, **kwargs*)
Alias for curv_grid()

num2axes2d(*xnumaxis, ynumaxis, xatts=None, yatts=None, xbounds2d=None, ybounds2d=None*)
Convert numerical 2D to complete 2D axes

- **xnumaxis**: 2D arrays of x positions [x,y]
- **ynumaxis**: 2D arrays of y positions [x,y]
- *xatts*: Attributes for output 2D X axis
- *yatts*: Attributes for output 2D Y axis
- *xbounds*: Bounds of output 2D X axis
- *ybounds*: Bounds of output 2D Y axis

Returns [xaxis2d,yaxis2d]

var2d(*var, xaxis=None, yaxis=None, xatts=None, yatts=None, gid=None, **kwargs*)
Create 2D cdms variable with on a proper 2d curvilinear grid

- **var**: Numeric or formatted X axis
- *xaxis*: Numeric or formatted X axis. Mandatory if var is not a cdms variable!
- *yaxis*: Numeric or formatted Y axis. Mandatory if var is not a cdms variable!
- *atts*: Attributes of the variable .
- *xatts*: Attributes of X axis.
- *yatts*: Attributes of Y axis.
- *gid*: Id of the grid.
- All other keywords are passed to cdms.createVariable()

Return: A cdms variable

axis1d_from_bounds(*axis1d, atts=None, numeric=False*)
Create a numeric of formatted 1D axis from bounds

- **axis1d**: Input 1d axis from which we get bounds
- *numeric*: Return a simple numeric array
- *atts*: Attributes for outputs axis

get_xy(*gg, m=False, mesh=None, num=False, **kwargs*)
Get axes from gg

- **gg**: (xx,yy) or grid (1D or 2D) or cdms variable.
- *m*: If True or basemap instance, convert to meters. If None, check if lon and lat axes to force conversion. [default: False]

deg2xy(*lon, lat, m=None, inverse=False, proj='merc', **kwargs*)
Convert from degrees to map (m) coordinates using map projection, and reverse

- **lon**: Longitudes in degrees
- **lat**: Latitude in degrees
- *m*: Basemap object for projection. If False, returns (lon,lat). If None, a new instance is created.
- *inverse*: Inverse transform (from meters to degrees)

set_grid(*var, gg, axes=True*)
Set a grid to a variable

- **var**: A cdms variable with at least 2 dimensions
- **gg**: A cdms grid or tuple or axes

- *axes*: Set grid raw axes to the variable (gg.getAxis(i)) which are different from real axes with curvilinear grids

check_xy_shape(*xx, yy, mesh=None*)
Check that xx and yy have the same shape

- **xx**: X positions (in meters or degrees).
- **yy**: Y positions (in meters or degrees).
- *mesh*: Return a 2D axes if True, or if None and xx and yy have not the same shape [default: None]

axes2d(*xaxis, yaxis, bounds=False, numeric=False, xatts=None, yatts=None, xbounds2d=None, ybounds2d=None*)
Create 2D numerical of complete axes

- **xaxis**: 1D or 2D X axis
- **xaxis**: 1D or 2D Y axis
- *bounds*: Return extended axes positioned on bounds (useful for pcolor) [default: False]
- *numeric*: Only return numerical values instead of complete axes [default: True]
- *xbounds2d*: 2D bounds of input xaxis
- *ybounds2d*: 2D bounds of input yaxis

Return: xaxis2d,yaxis2d

meshgrid(*x, y, copy=1*)
Convert pure numeric x/y axes to 2d arrays of same shape. It works like `numpy.meshgrid()` but is more flexible.

- **x**: 1D or 2D array
- **x**: 2D or 2D array

Return xx(ny,nx),yy(ny,nx)

meshbounds(**args*)
A shortcut to `meshcells()`

meshweights(*x, y=None, proj=None, axis=-1*)
Return a 1D or 2D array corresponding the cell weights

- **x**: 1D or 2D array
- **y**: 1D or 2D array
- *proj*: Geographic projection before computing weights (for 2D case only with both x and y)
  - `True`: use default mercator projection (see `merc()`),
  - else, directly apply projection.

Returns ww(nx) OR ww(ny,nx)

t2uvgrids(*gg, getu=True, getv=True, mask=None*)
Convert a (C) grid at T-points to a grid at U- and/or V- points

- **gg**: A (x,y) or a cdms grid or a cdms variable with a grid.
- *getu*: Get the grid at U-points [default: True]
- *getv*: Get the grid at V-points [default: True]
- *mask*: If False, do not try to guess masks ; if None, try to get them from original grid by conversion to U and V points with t2uvmask() [default: None]

Return: ugrid,vgrid OR ugrid OR vgrid depending on getu and getv

meshcells(*x, y=None*)
Return a 1D or 2D array corresponding the cell corners

- **x**: 1D or 2D array
- **y**: 1D or 2D array

Returns xxb(nx+1) OR xxb(ny+1,nx+1),yyb(ny+1,nx+1)

cells2grid(*xxb, yyb*)
Convert a grid of cells (grid of corners, like results from `meshcells()`) to a grid (grid of centers)

- **xxb**: X of corners (ny+1,nx+1)
- **yyb**: Y of corners (ny+1,nx+1)

Returns xx(ny,nx),yy(ny,nx)

curv_grid(*xaxis, yaxis, xatts=None, yatts=None, id=None, mask=None*)
Create a curvilinear 2D grid from 1D or 2D axes

- **xaxis**: Numeric or formatted X axis
- **yaxis**: Numeric or formatted Y axis
- *xatts*: Attributes of X axis
- *yatts*: Attributes of Y axis
- *id*: Id of the grid

bounds2mesh(*xb, yb=None*)
Convert 2D 4-corners cell bounds arrays (results from `bounds2d()`) to 2D arrays

- **xb**: array(ny,nx,4)
- **yb**: array(ny,nx,4)

Return xxb(ny+1,nx+1),yyb(ny+1,nx+1)

resol(*axis, averaged=True*)
Get the resolution of an axis or a grid

- **axis**: it can be either
  - a 1D axis or array
  - a grid of 1D or 2D axes or tuple of (lon,lat)
- *averaged*: Return an averaged resolution (do not use it the grid highly anisotropic!!)

create_grid(*lon, lat, mask=None, lonatts={}, latatts={}, curv=None, **kwargs*)
Create a cdms rectangular or curvilinear grid from axes

- **lon**: Array or axis of longitudes or any argument passed to `create_lon()`.
- **lat**: Array or axis of latitudes or any argument passed to `create_lat()`.
- *mask*: Grid mask.
- *(lon/lat)atts*: Attributes to set for axes.
- *curv*:
  - `None`: If one axis is 2D, the grid will be curvilinear.
  - `True|False`: Force the grid to be or not to be curvilinear.

Return A `cdms2` grid object.
Example

```
>>> create_grid([1.,3.,5.,7], numpy.arange(45., 60., .5))
>>> create_grid((.1, 8., 1.), (45., 60., .5))
```

See Also:

`create_lon()` `create_lat()` `get_grid()` `set_grid()`

rotate_grid(*ggi, angle, pivot='center', **kwargs*)
Rotate a grid

- **ggi**: Cdms grid ou (lon,lat) or variable.
- **angle**: Angle in degrees.
- *pivot*: it can be either
  - A tuple of (lon, lat)
  - A string that specify the vertical and horizontal position. Use the following keys : `top`, `bottom`,''left'',''right'',''center''.
- Other keywords are passed to `curv_grid()`

Returns A curvilinear cdms grid.

Example

```
>>> mygrid = rotate_grid((lon,lat), 60., 'top right')
>>> mygrid2 = rotate_grid(mygrid, 60., (-5.,45))
```

isregular(*axis, tol=0.050000000000000003, iaxis=None, dx=None*)
Check is an 1S or 2D axis is regular

- **axis**: A `cdms2` axis or `numpy` array
- *tol*: Relative tolerance
- *iaxis*: On which direction to operate for 2D axis

monotonic(*xx, xref=None*)
Make monotonic an array of longitudes

actimar.misc.grid.regridding — Regrillage

Regridding utilities

See Also:

Tutorials: *Regrillage 1D et 2D*

fill1d(*vari, axis=0, method='linear', maxgap=0*)
Fill missing values of a 1D array using interpolation.

- **vari**: Input `cdms2` variable
- *axis*: Axis number on which filling is performed
- *method*: Interpolation method (see `interp1d()`)
- *maxgap*: Maximal size of filled gaps (in steps)

Example

```
>>> fill1d(vari, axis=2, method='cubic', maxgap=5)
```

Return Filled `cdms2` variable similar to input one

regular(*vi, dx=None, verbose=True, auto_bounds=False*)
Fill a variable with missing values when step of first axis is increasing

- **vi**: Input array on almost regular axis

- *dx*: Force grid step to this. Else, auto evaluated.

regular_fill1d(*var, k=1, dx=None*)
Combination: fill1d(regular_fill)) (with their parameter)

remap1d(*vari, axo, conserv=False, **kwargs*)
Remapping along an axis

- **vari**: Input cdms array
- **axo**: Output cdms axis
- *axis*: Axis on wich to operate
- *conservative*: If True, remapping is conservative
- Other keywords are passed to `regrid1d()`

Note: This is an wrapper to `regrid1d()` using `remap` or `conservative` as a default method. See its help for more information.

spline_interp1d(*old_var, new_axis, check_missing=True, k=3, **kwargs*)
Backward compatibility function

See `regrid1d()`

refine(*vari, factor, geo=True, smoothcoast=False, noaxes=False*)
Refine a variable on a grid by a factor

- **vari**: 1D or 2D variable.
- **factor**: Refinement factor > 1

class **GridData**(*xi, yi, ggo, nl=False, ext=False, geo=None, method='nat', sub=30, compress=False, **kwargs*)
2D interpolator from a randomly spaced sample on a regular grid

Possible algorithms:

- Natural Neighbor using nat.Natgrid:
  - http://www.ncarg.ucar.edu//ngmath/natgrid/nnhome.html
  - http://www.cisl.ucar.edu/zine/98/spring/text/3.natgrid.html
  - http://dilbert.engr.ucdavis.edu/~suku/nem/nem_intro/node3.html
  - http://www.ems-i.com/gmshelp/Interpolation/Interpolation_Schemes/Natural_Neighbor_Interpolation.htm
- 2D splines using css.css.Cssgrid

Parameters
- **xi**: Input 1D X positions.
- **yi**: Input 1D Y positions.
- **ggo**: Output grid. Can either (xo,yo), a cdms grid or a cdms variable with a grid.
- *method*: Interpolator type, either 'nat' (Natural Neighbors) or 'css' (='splines' using splines) [default: 'nat']
- *nl*: Nonlinear interpolator (usually gives better results) [default: False]
- *ext*: Extrapolate value outsite convex hull [**'nat' only**, default: False]
- *mask*: Mask to apply to output data [default: None]
- *compress*: If `True`, interolate only unmasked data, and thus does not try guess the best mask (that's more efficient but very bad if data are masked!).
- *sub*: Size of blocks for subblocking [**"nat" only**]
- Other keywords are set as attribute to the interpolator instance ; to get the list of parameters:

```
>>> import nat ; nat.printParameterTable()
>>> import css ; css.printParameterTable()
```

Example

```
>>> r = GridData(gridi, grido, method='nat', hor=.2, ext=False)
>>> varo1 = r(vari1)
>>> varo2 = r(vari2)
```

regrid(*zi, missing_value=None, **kwargs*)
Interpolate zi on output grid

- **zi**: At least a 1D array.

rgrd(*zi, missing_value=None, **kwargs*)
Interpolate zi on output grid

- **zi**: At least a 1D array.

griddata(*xi, yi, zi, ggo, method='krig', cgrid=False, **kwargs*)
Interpolation in one single shot using GridData

- **xi**: 1D input x coordinates.
- **yi**: 1D input y coordinates (same length as xi).
- **zi**: 1D input values (same length as xi).
- *method*: Method of interpolation, within ('nat', 'css', 'krig') [default: 'krig']
- *cgrid*: Output on a C-grid at U- and V-points deduced from ggo [default: False]

See: `GridData` and `krigdata()`

krigdata(*xi, yi, zi, ggo, mask=None, geo=None, compress=False, missing_value=None, **kwargs*)
Interpolator using basic krigging

- **xi**: Input 1D X positions.
- **yi**: Input 1D Y positions.
- **ggo**: Output grid. Can either (xo,yo), a cdms grid or a cdms variable with a grid.
- *mask*: Mask to apply to output data [default: None]

fill2d(*var, xx=None, yy=None, mask2d=None, copy=True, **kwargs*)
Fill missing value of 2D variable using inter/extrapolation

- **var**: A cdms 2D variable.
- *xx/yy*: Substitutes for axis coordinates [default: None]
- Other keywords are passe to `griddata()`

regrid2d(*vari, ggo, method='auto', mask_thres=0.5, ext=False, use_scrip=None, bilinear_masking='dstwgt', ext_masking='poly', mixt_fill=True, check_mask=True, **kwargs*)
Regrid a variable from a regular grid to another

- **vari**: Variable on regular grid
- **ggo**: Tuple of (x,y) or a cdms grid or a cdms variable with a grid
- **method**: One of:
  - `"auto"`: method guessed according to resolution of input and output grid (see `regrid_method()`)
  - `"nearest"`: nearest neighbour
  - `"linear"` or `"bilinear"`: bilinear interpolation (low res. to high res.)
  - `"dstwgt"` : distance weighting (low res. to high res.)
  - `"remap"` : weighted remapping based on areas of cells (high res. to low res.)
  - `"bining"` : simple averaging using bining (very high res. to low res.)
  - `"nat"` : Natgrid interpolation (low res. to high res.) (see `GridData` for more info)
  - `"krig"` : Fake kriging interpolation (low res. to high res.) (see `krigdata()` for more info)
- *ext*: Perform extrapolation when possible
- *use_scrip*: Use `SCRIP` interpolator when possible

- *bilinear_masking*: the way to handle interpolation near masked values
  - "nearest": brut masking using nearest neighbor
  - "dstwgt" : distance weight data are used where interpolated mask is lower mask_thres
- *mask_thres*: Threshold for masking points for some methods:
  - method="bilinear" and bilinear_masking="dstwght"
  - method="remap" or method="bining"
- *ext_masking*: Manual masking method when ext=False (when needed) with methods ["krig",] (see grid_envelop_mask()) if input grid is not rectangular
  - "poly": use the polygon defined by the input grid envelopp and check if output points are inside
  - "nearest": use hack with nearest 2d interpolation
- Other keywords are passed to special interpolation functions depending on method and choices :
  - SCRIP when SCRIP interpolator is used
  - krigdata() when "nat" or "krig" method is used

Examples

```
>>> regrid2d(var, (lon, lat), method='bilinear', bilinear_masking='nearest')
>>> regrid2d(var, grid, method='remap', mask_thres=.8)
>>> regrid2d(var, grid, method='nat', hor=.2)
```

class SCRIP(*ggi, ggo, check_mask=True, ext=False, **kwargs*)

Regridding using SCRIP regridder

See cdms help and http://climate.lanl.gov/Software/SCRIP

areas(*igrid*)

Get overlapping cell areas (conservative only)

- **igrid** :
  - 0: Source grid
  - 1: Destination grid

exe()

SCRIP executable

fractions(*igrid*)

Get overlapping fractions

- **igrid** :
  - 0: Source grid
  - 1: Destination grid

grid(*igrid=0*)

Get one grid

mask(*igrid=0, ongrid=False*)

Get one mask

method()

Get the regridding method

r(*togrid=1*)

Shortcut to regridder()

regrid(*vari*)

Regrid input variable from grid0 to grid1

regridder(*togrid=1*)

Built regridder instance

rgrd(*vari*)

Regrid input variable from grid0 to grid1

workdir()
Workdir

scrip(*vari, ggo, **kwargs*)
Regridding between curviliear grids using SCRIP

regrid1d(*vari, axo, method='auto', axis=None, xmap=None, xmapper=None*)
Interpolation along one axis

- **vari**: Input cdms array.
- **axo**: Output cdms axis.
- **method**:
  - "nearest": Nearest neighbor
  - "linear": Linear interpolation
  - "cubic": Cubic interpolation
  - "remap": Remapping (cell averaging)
  - "conserv": Conservative remapping (like remap but with integral preserved)
- *axis*: Axis (int) on wich to operate. If not specified, it is guessed from the input and output axis types, or set to 0.
- **xmap**: Integer or tuple that specify on which axes input axis is varying.
- **xmapper**: Array that specify values of input axis along axes specified by xmap. It is an array of size (...,len(var.getAxis(xmap[-2])), len(var.getAxis(xmap[-1])), len(var.getAxis(axis))].

interp1d(*vari, axo, method='linear', **kwargs*)
Linear or cubic interpolation along an axes

- **vari**: Input cdms array
- **axo**: Output cdms axis
- *axis*: Axis on wich to operate
- Other keywords are passed to regrid1d()

Note: This is an wrapper to regrid1d() using linear as a default method. See its help for more information.

nearest1d(*vari, axo, **kwargs*)
Interpolation along an axes

- **vari**: Input cdms array
- **axo**: Output cdms axis
- *axis*: Axis on wich to operate
- Other keywords are passed to regrid1d()

Note: This is an wrapper to regrid1d() using nearest as a default method. See its help for more information.

cubic1d(*vari, axo, **kwargs*)
Cubic interpolation along an axes

- **vari**: Input cdms array
- **axo**: Output cdms axis
- *axis*: Axis on wich to operate
- Other keywords are passed to regrid1d()

Note: This is an wrapper to regrid1d() using cubic as a default method. See its help for more information.

xy2grid(**args, **kwargs*)
Alias for griddata()

See Also:

GridData krigdata() xy2grid()

grid2xy(*vari, xo, yo, method='bilinear', outaxis=None*)
Interpolate gridded data to ramdom positions

- **vari**: Input cdms variable on a grid
- **xo**: Output longitudes
- **yo**: Output latitudes
- *method*: Interpolation method
  - –nearest: Nearest neighbor
  - –bilinear: Linear interpolation
  - –nat: Natgrid interpolation
- *outaxis**: Output spatial axis

fill1d(*vari, axis=0, method='linear', maxgap=0*)
Fill missing values of a 1D array using interpolation.

- **vari**: Input cdms2 variable
- *axis*: Axis number on which filling is performed
- *method*: Interpolation method (see interp1d())
- *maxgap*: Maximal size of filled gaps (in steps)

Example

```
>>> fill1d(vari, axis=2, method='cubic', maxgap=5)
```

Return Filled cdms2 variable similar to input one

class **GriddedMerger**(*grid, id=None, long_name=None, units=None*)

add(**args, **kwargs*)
Alias for append()

append(*var, method='auto', **kwargs*)
Append a bathymetry to the top of the merger

get_grid()
Get the grid for merging

get_lat()
Get the loatitudes of the grid

get_lon()
Get the longitudes of the grid

insert(*idx, var*)

merge(*res_ratio=0.5, **kwargs*)
Merge all the variables on to a grid

- **grid**: Out put grid or axes.
- *res_ratio*: Resolution ratio for choosing between remaping or bilinear interpolation (see: regrid_method()).
- *regrid_<kwparam>*: <*kwparam*> is passed to regrid2d() for interpolation.

plot(***kwargs*)
Merge and plot

remove(*var*)

set_grid(*grid*)
Set the grid for merging

regrid_method(*gridi, grido, ratio=0.5*)
Guess the best regridding method for passing from `gridi` to `grido`

If `resolution(grido) <= ratio*resolution(gridi)`, `method="remap"` else `method="interp"`

- **gridi**: Input 1d or 2D grid, or 1D axis.
- **grido**: Output 1d or 2D grid, or 1D axis.
- *ratio*: Limit ratio of output grid resolution to input grid resolution.

..note:

```
The resolution of the grids is check in their attributes "_xres" and "_yres"
before before trying to compute them.
```

Returns 'remap' OR 'interp'

remap2d(*vari, ggo, **kwargs*)
Shortcut to `regrid2d()` call with `method="remap"`

interp2d(*vari, ggo, method='interp', **kwargs*)
Shortcut to `regrid2d()` call with `method="interp"` by default

xy2xy(*xi, yi, zi, xo, yo, nl=False, geo=True, **kwargs*)
Interpolation between to unstructured grids using `Natgrid`

- **xi?yi**: 1D input positions
- **zi**: atleast-1D input values
- **xo,yo**: 1D output positions
- *nl*: Non linear interpolation using natural neighbours
- *geo*: convert positions to meters using mercator projection

actimar.misc.grid.masking — Masquage

Utilities to deal with masks and selections

See Also:

Tutorials: *Les masques*, *Les polygones*

get_coast(*mask, land=True, b=True, borders=True, corners=True*)
Get a mask integer of ocean coastal points from a 2D mask

- **mask**: Input mask with 0 on water and 1 on land.
- *land*: If True, coast is on land [default: True]
- *corners*: If True, consider borders as coastal points [default: True]
- *borders:* If True, consider corners as coastal points [default: True]

At each point, return 0 if not coast, else an interger ranging from 1 to (2**8-1) to describe the coast point.
128 4 64 8 + 2 16 1 32

get_coastal_indices(*mask, coast=None, **kwargs*)
Get indices of ocean coastal points from a 2D mask

- **mask**: Input mask with 0 on water and 1 on land.
- *boundary*: If True, consider outside boundary as land [default: True]

class GetLakes(*mask, nmaxcells=None*)

Bases: object

Find lakes in a 2d mask where 0 is water and 1 is land

Example:

```
>>> from actimar.misc.grid import GetLakes
>>> import numpy as N
>>> from pylab import pcolor,show,title
>>> # Build the mask
>>> mask=N.ones((500,500))
>>> mask[3:10,100:102]=0
>>> mask[103:110,200:210]=0
>>> mask[203:210,200:220]=0
>>> # Find the lakes
>>> lakes = GetLakes(mask,nmaxcells=80)
>>> print 'Number of lakes:', len(lakes.indices())
Number of lakes: 2
>>> pcolor(lakes.mask(),shading=False)
>>> title('Lakes')
>>> show()
```

indices(*nmaxcells=None, **kwargs*)

Return a list of lake coordinates

lakes(*id=None, nmaxcells=None, **kwargs*)

Return an array of same size as masks, but with points in lakes set to its lake id, and others to set to zero

Parameters

- *id*: Select lake #<id>
- *nmaxcells*: Do not consider lakes with number of points greater than nmaxcells

mask(*id=None, nmaxcells=None, **kwargs*)

Returns a boolean land/sea mask from lakes() (land is True)

Parameters

- *id*: Select lake #<id>
- *nmaxcells*: Do not consider lakes with number of points greater than nmaxcells

Example

```
>>> mask_lake2 = GetLakes(mask).mask(2)
```

ncells()

Number of cell point for each lake

ocean(***kwargs*)

Get the biggest lake or its mask (integer type)

Parameters

- *mask*: If True, return the mask, not the lake [default: True]
- Other keywords are passed to `scan()`

plot(***kwargs*)

Display the lakes

scan(***kwargs*)

Find all lakes

polygon_mask(*gg, polys, mode='intersect', thresholds=, [0.5, 0.75], ocean=False, fractions=0, yclean=True, premask=None*)

Create a mask on a regular or curvilinear grid according to a polygon list

- **gg**: A cdms grid or variable or a tuple of (xx,yy).
- **polys**: A list of polygons or GMT resolutions or Shapes instance like shorelines.

- *mode*: Way to decide if a grid point is masked. Possible values are:
  - `intersect`, 1, `area` (default): Masked if land area fraction is > `thresholds[0]`. If more than one intersections, leand area fraction must be > `thresholds[1]` to prevent masking straits.
  - else: Masked if grid point inside polygon.
- *thresholds*: See `intersect` mode [default: [.5, .75]]
- *ocean*: If True, keep only the ocean (= biggest lake) [default: True]

masked_polygon(*vv, polys, copy=0, **kwargs*)
Mask a variable according to a polygon list

- **vv**: The variable
- **polys**: Polygons (or something like that)
- *copy*: Copy data [default: 0]
- Other keywords are passed to `MV.masked_where()`

polygons(*polys, proj=None, clip=None, shapetype=2, **kwargs*)
Return a list of Polygon instances

- **polys**: A tuple or list of polygon proxies (see examples).
- *shapetype*: 1 = Polygons, 2=Polylines(=LineStrings) [default: 2]
- *proj*: Geographical projection to convert positions.

Example :

```
>>> from actimar.misc.grid.masking import polygons
>>> import numpy as N
>>> X = [0,1,1,0]
>>> Y = N.array([0,0,1,1])
>>> polygons( ([X,Y],) )
>>> polygons( (zip(X,Y), [X,Y], N.array([X,Y]) )
>>> polygons( (polygons(([X,Y],), polygons(([X,Y],)))
>>> polygons( ([min(X),min(Y),max(X),max(Y)],) )
```

d2m(*x, y*)

polygon_select(*xx, yy, polys, vv=None, mask=False*)
Select unstructered points that are inside polygons

- **xx**: Positions along X.
- **yy**: Positions along Y.
- **polys**: Polygons.
- *vv*: Values at theses positions.
- *mask*: If not False, return:
  - the mask if vv is not set,
  - a masked version of vv if vv is set.

envelop(**args, **kwargs*)
Shortcut to `convex_hull()`

check_poly_islands(*mask, polys, offsetmin=0.84999999999999998, offsetmax=1.5, dcell=2*)
Check that islands as greater as a cell are taken into account

- **mask**: The initial mask. A cdms variable with X (longitude) and Y (latitude), or a tuple (lon, lat, mask).
- **polys**: Coastal polygons.
- *offset*: Islands whose area is > 1-offset and < 1+offset % of the mean cell area are checked [default: .95]
- *dcell*: dx and dy relative extension from the center of the cell in resolution units.

Example:

```
>>> from actimar.misc.grid.masks import check_poly_islands
>>> import MV as cdms ; from actimar.misc.axes import create_lon,create_lat
>>> mask = MV.array(mask)
>>> mask.setAxis TO BE FINISHED
```

check_poly_straits(*mask, polys, dcell=2, threshold=0.75*)
Check that straits are opened by counting the number of polygon intersection in each cell and the area of water

- **mask**: The initial mask. A cdms variable with X (longitude) and Y (latitude).
- **polys**: Coastal polygons.
- *dcell*: dx and dy relative extension from the center of the cell in resolution units.
- *threshold*: Maximal fraction of cell area that must be covered of land (> .5) [default: .25]

t2uvmasks(*tmask, getu=True, getv=True*)
Compoute masks at U and V points from a mask at T points on a C grid (True is land)

- **tmask**: A 2D mask.
- *getu*: Get the mask at U-points
- *getv*: Get the mask at V-points

Return umask,vmask OR umask OR vmask depending on getu and getv.

mask2d(*mask, method='or'*)
Convert a 3(or more)-D mask to 2D by performing compression on first axes

Note: Mask is False on ocean

- **mask**: At least 2D mask
- *method*: Compression method
  - `"or"`: Only one point must be unmask to get the compressed dimension unmasked
  - `"and"`: All points must be unmask to get the compressed dimension unmasked

grid_envelop(*gg, centers=False, poly=True*)
Return a polygon that encloses a grid

- **gg**: A cdms grid or a tuple of (lat,lon)
- *centers*:
  - `True`: The polygon is at the cell center.
  - `False`: The polygon is at the cell corners.
- *poly*:
  - `True`: Return as Polygon instance.
  - `False`: Return two 1D arrays `xpts,ypts`

convex_hull(*xy, poly=False, method='delaunay'*)
Get the envelop of cloud of points

- **xy**: (x,y) or array of size (2,nxy)
- *poly*:
  - `True`: Return as Polygon instance.
  - `False`: Return two 1D arrays `xpts,ypts`
  - *method*:
    - `"angles"`: Recursive scan of angles between points.
    - `"delaunay"`: Use Delaunlay triangulation.

uniq(*data*)
Remove duplicates in data

Example

```
>>> import numpy as N
>>> a = N.arange(20.).reshape((10,2))
>>> a[5] = a[1]
>>> b = uniq(a)
```

rsamp(*x, y, r, z=None, rmean=0.69999999999999996, proj=False, rblock=3, getmask=False*)
Undersample data points using a radius of proximity

- **x**: 1D x array.
- **y**: 1D Y array.
- **r**: Radius of proximity.
- *z*: 1D Z array.
- *proj*: Geographic projection instance to convert coordinates.
- *rmean*: Radius of averaging relative to `r` (`0`<`rmean`<1)
- *rblock*: Size of blocks relative to `r` for computation by blocks.

zcompress(*z, *xy, **kwargs*)
Compress 1D arrays according to the mask of the first one

- **z**: Reference (masked) array
- *xy*: Additional arrays to be compressed
- *numpify*: Force convertion to numpy array

Example

```
>>> z, x, y = zcompress(z, x, y, numpify=True)
```

actimar.misc.grid.basemap — Dérivés de mpl_toolkits.basemap

Utilities derived from mpl_toolkits.basemap

gshhs_autores(*lon_min, lon_max, lat_min, lat_max*)
Guess best resolution from lon/lat bounds

cached_map(*m=None, mapdir=None, verbose=False, **kwargs*)
Check if we have a cached map

- *m*: A Basemap instance [Default: None]
- *mapdir*: Where are stored the cached maps. If `None`, `matplotlib.get_configdir()` is used as a parent directory, which is the matplotlib configuration directory (`~/.matplotlib` undex linux), and `basemap/cached_maps` as the subdirectory.

Example

```
>>> m = cached_map(lon_min=-5, lon_max=6, lat_min=40, lat_max=50, projection='lcc', resolution='f')
>>> m = cached_map(m) # Does only caching of map
```

cache_map(*m, mapdir=None*)
Cache a map if still not cached

get_map(*gg, proj=None, res=None, auto=False, **kwargs*)
Get a suitable map for converting degrees to meters

- **gg**: cdms grid or variable, or (xx,yy).

- *res*: Resolution [default: None]
- *proj*: Projection [default: None->'merc']
- *auto*: If True, get geo specs according to grid. If False, whole earth. If None, auto = res is None [default: False]

class GSHHS_BM(*input=None, clip=None, sort=True, reverse=True, **kwargs*)

Bases: actimar.misc.io.Shapes

Shoreline from USGS using Basemap

Initialized with a valid Basemap instance with resolution not equal to None, or thanks to arguments passed to actimar.misc.plot.map()

- *m*: Basemap instance [default: None]

clip(*zone, **kwargs*)

Clip to zone

get_data(*key=None*)

Get the numeric version of the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_map()

Return the associated basemap instance if set

get_points(*key=None, split=True*)

Get all the points from all the shapes as a tuple (x,y)

get_projs()

Get the list on projections (proj instance, inverse flag)

get_shapes(*key=None*)

Get the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_type()

Return the type of shapes

- 0 = Points,
- 1 = LineStrings = PolyLines
- 2 = Polygons

plot(*select=None, ax=None, fill=None, fillcolor=None, points=False, m=None, show=False, **kwargs*)

Plot shapes

- *select*: argument for selecting shapes in the list [defaul: None].
- *fill*: Force filling (True/False), else guessed from shpe type, ie filling for polygons only [default: None]
- *ax*: Axes instance [default: None]
- Other keyparams are used for plotting (lines or polygons)

project(*proj, inverse=False*)

Project shapes using proj

- **proj**: A Basemap instance or pure projection instance (from Basemap)
- *inverse*: Inverse projection [default: False]

resol(*deg=True*)

Compute the mean "resolution" of the shapes based on the first shape

- **deg**:
  - if `False`: return a resolution in meters has a the median distance between points
  - if `True`: return the median distance between points as a resolution in degrees `(xres,yres)`

sort(*reverse=True*)

Sort shapes according to their surface

- *reverse*: If True, greater polygons are first [default: True]

sorted()

xy(*key=None*)
Shortcut to `get_points(split=false)`

merc(*lon=None, lat=None, **kwargs*)
Mercator map

- Extra keywords are passed to `mpl_toolkits.basemap.Basemap`
- *lon*: Longitudes to define `llcrnrlon` and `urcrnrlon`
- *lat*: Latitudes to define `lat_ts`, `llcrnrlat` and `urcrnrlat`

clean_cache(*mapdir, maxsize=10485760.0*)

2.1.9 actimar.misc.phys — Utilitaires de physique

actimar.misc.phys.constants — Constantes

Physical constants

actimar.misc.phys.units — Unités

Unit conversions

deg2dms(*deg*)
Convert from degrees to degrees/minutes/seconds

- **deg**: degrees

See Also:

`dms2deg()`

deg2m(*degrees, lat=None*)
Convert a distance in degrees to meters

- **degrees**: Distance in degrees
- *lat*: optional latitude, defaults to 0.

Return: Distance in meters

See Also:

`m2deg()`

dms2deg(*d, m=0, s=0*)
Convert from degrees/minutes/seconds to degrees

- **d**: degrees
- *m*: minutes [default: 0]
- *s*: seconds [default: 0]

See Also:

`deg2dms()`

kt2ms(*nd*)
Convert nds to m/s

m2deg(*meters, lat=None*)
Convert a distance in meters to degrees

- **meters**: Distance in meters
- *lat*: optional latitude, defaults to 0.

Return: Distance in degrees

See Also:

deg2m()

ms2bf(*ms*)
Convert from m/s to Beauforts (wind)

- **ms**: Wind speed in m/s.

See Also:

http://fr.wikipedia.org/wiki/%C3%89chelle_de_Beaufort

ms2kt(*ms*)
Convert m/s to nds

ms2nd(*ms*)
Convert m/s to nds

nd2ms(*nd*)
Convert nds to m/s

2.2 actimar.bathy — Bathymétrie et trait de côte

2.2.1 actimar.bathy.bathy — Bathymétries

Get or display the Gebco bathymetry

list_bathy()
List of available bathymetries

class Bathy(*lon=None, lat=None, name='etopo2', **kwargs*)
Bases: object

func(*lon=None, lat=None, sampling=None, fill=None, mask_function='greater', mask_value=0.0*)
Select a zoom and set the current variable

- *lon*: Longitude range [default: None]
- *lat*: Latitude range [default: None]
- *sampling*: Sampling in both directions [default: 1]
- *fill*: If True, fill positive values with 0. [default: True]
- *mask_function*: To mask points. Should be 'greater' or 'less'. If empty or None, no mask is applied [default: 'greater']
- *mask_value*: Value used with mask_function [default: 0.]

lat()
Latitude axis

lon()
Longitude axis

plot(*lon=None, lat=None, **kwargs*)
Plot the current bathymetry

- *lon*: Longitude range.
- *lat*: Latitude range.

See Also:

plot_bathy()

show(*lon=None, lat=None, **kwargs*)
Plot the current bathymetry

- *lon*: Longitude range.
- *lat*: Latitude range.

See Also:

plot_bathy()

write_ascii(*file, fmt='%10f %10f %10f'*)

Write the bathymetry as an ascii file in the format : x y z

- **file**: Output file name
- *fmt*: Format of lines [default: '%10f %10f %10f']

write_netcdf(*file*)

Write the bathymetry as netcdf file

- **file**: Output netcdf file name

zoom(*lon=None, lat=None, sampling=None, fill=None, mask_function='greater', mask_value=0.0*)

Select a zoom and set the current variable

- *lon*: Longitude range [default: None]
- *lat*: Latitude range [default: None]
- *sampling*: Sampling in both directions [default: 1]
- *fill*: If True, fill positive values with 0. [default: True]
- *mask_function*: To mask points. Should be 'greater' or 'less'. If empty or None, no mask is applied [default: 'greater']
- *mask_value*: Value used with mask_function [default: 0.]

plot_bathy(*bathy, shadow=True, contour=True, shadow_stretch=1.0, shadow_shapiro=False, show=True, shadow_alpha=1.0, shadow_black=0.29999999999999999, white_deep=False, nmax=30, m=None, alpha=1.0, **kwargs*)

Plot a bathymetry

- *lon*: Longitude range.
- *lat*: Latitude range.
- *show*:Display the figure [default: True]
- *pcolor*: Use pcolor instead of contour [default: False]
- *contour*: Add line contours [default: True]
- *shadow*:Plot south-west shadows instead of filled contours.
- *nmax*: Max number of levels for contours [default: 30]
- *white_deep*: Deep contours are white [default: False]
- All other keyword are passed to map()

fmt_bathy(*bathy, id=None, long_name=None*)

Format a bathymetry variable

class **XYZBathy**(*xyz, check_sign=False, *args, **kwargs*)

Bases: actimar.misc.io.XYZ

For random bathymetries, like mnt files

- **xyz**: A (x,y,z) tuple or a xyz file.
- *long_name*: Long name (like a title) for this bathymetry.

Note: Bathymetry is supposed to be positive over water.

Usage

```
>>> xzy1 = XZY('bathy1.xzy', long_name='My bathy')          # Load
>>> xyz1.select([-6,45,-4,48])
>>> xyz1 += XZY('bathy2.xzy')
>>> bathy = xyz1.togrid(mygrid)
```

See Also:

XYZ shoreline MeanSeaLevel

bathy(**args, **kwargs*)
Alias for `togrid()`

clip(*zone=None, margin=None, inverse=False, mask=False, id=None, **kwargs*)
Geographical selection of part of the data

- *zone*: (xmin,ymin,xmax,ymax) or a float/int a complex polygon (see `polygons()`).
- *margin*: Margin around `zone` relative to the resolution (see `resol()`)
- *inverse*: Inverse the selection.
- *mask*: `zone` must be interpreted as a mask

consolidate()
Apply radius undersampling and all exclusions and selections to data and reset them

contains(*x, y*)
Check if one or several points are within a the convex hull

- **x,y**: X,Y positions as floats or lists or an `numpy` arrays.

copy()
Deep copy

exclude(**zones*)
Add one or more zones where data are not used.

A zone can be :

- an argument to `polygons()` to get a `_geoslib.Polygon` instance,
- another :class:XYZ‘ instance from which the convex hull (see `hull()`) is used as a delimiting area

Usage

```
>>> xyz.exclude([[-8,43],[-5.5,43],[-6,45.]],[[-10,45],[-7,47],[-10,49.]])
>>> xyz.exclude(polygon1,polygon2)
>>> xyz.exclude(xyz1,[-5,42,-3,48.])
```

See Also:
`select()` `exclusions()`

exclusions()
Get all exclusion polygons as a tuple

get_magnet()
Get the magnet integer attribute
Note: Useful only for mixing `XYZ` instances

get_res(*deg=False, auto=None*)
Get the mean X and Y resolutions in meters or degrees

get_rsamp()
Get the radius sampling

get_transp()
Get the transparency boolean attribute
Note: Useful only for mixing `XYZ` instances

grid(*res=None, xmin=None, xmax=None, ymin=None, ymax=None, relres=0.5, degres=False, id='xyz_grid'*)
Rectangular grid based on x/y positions and resolution

- *res*: Resolution. It can be:
  - –a float where then `xres=yres=res`
  - –a tuple as `(xres,yres)`
  - –else it is guessed using `get_res()` (and maybe `resol()`)‘ and multiplied by `relres`
- *relres*: Relative resolution factor applied to `res` when resolution is guessed (`res=None`)
- *degres*: When `res` is explicitly given, it interpreted as degrees is `degres` is True.
- *xmin,xmax,ymin,ymax*: Bounds of the grid. If not specified, bounds of the dataset are used (see `xmin()`, etc).

Note: Resolutions are adjusted when they are not mutiple of grid extensions (slightly decreased). Therefore, extensions of the grid are always preserved.

See Also:

`resol()`, `togrid()`

hull(*out='xy', mask=True*)
Return the convex hull

Returns Depends on `out`

- `"xy"`: (xhull, yhull)
- `"ind"`: indices of points
- `"poly"`: `_geoslib.Polygon` instance

interp(*xyo, xyz=False, ext=True, nl=True, **kwargs*)
Interpolate to (xo,yo) positions using `nat.Natgrid`

- **xo**: Output X
- **yo**: Output Y
- *xyz*: If True, return a `XYZ` instance instead of a `numpy` array
- *ext*: Allow extrapolation
- *nl*: Nonlinear interpolation

Returns An XYZ instance

mask()
Get the current mask due to exclusion and selection polygons

See Also:

`exclude()` `select()`

plot(*m=True, **kwargs*)
Scatter plot of bathymetry

See Also:

For value of default parameters: `plot()`

reset_exclusions()
Remove all exclusions

reset_rsamp()
Reset rsamp without affecting data

reset_selections()
Remove all selections

resol(*convex_hull_method='delaunay', exc=, [], deg=False*)
Return the mean resolution.

Algorithm: Median distances between facets of triangles

Returns (xres,yres)

save(*xyzfile, **kwargs*)
Save to a xyz file

- **xyzfile**: Output file name
- Other keywords are passed to `numpy.savetxt()`

select(**zones*)
Add one or more zone (polygons) where only these data are used

A zone is an argument to `polygons()` to get a `_geoslib.Polygon` instance.

Usage

```
>>> xyz.select([[-8,43],[-5.5,43],[-6,45.]],[[-10,45],[-7,47],[-10,49.]])
>>> xyz.select(polygon1,polygon2)
```

See Also:

`exclude()` `selections()`

selections()
Get all selection polygons as a tuple

set_magnet(*magnet*)
Set the magnet integet attribute. If set to `0`, no magnet effect.

Note: Useful only for mixing `XYZ` instances

set_res(*xres, yres=None*)
Set the resolution of the dataset

If `yres` is not, it is set to `xres`. When a value is negative, it is supposed to be in meters (not in degrees)

set_rsamp(*rsamp*)
Set the radius sampling If set to `0`, no sampling.

set_transp(*transp*)
Set the transparency boolean attribute

Note: Useful only for mixing `XYZ` instances

shadows()
Get the polygons defining the 'shadow' of this dataset.

It consists of a tuple of two elements:

- the convex hull as a polygon,
- a list of exclusion polygons that intersect the convex hull.

Therefore, a point in the shadow must be inside the convex hull polygon, and outside the exclusion polygons.

Returns (hull_poly, [exclusion_poly1,...])

tocfg(*cfg, section, param=None*)
Dump one or all parameters as options to a cfg section

- **cfg**: ConfigParser object
- **section**: Section of cfg
- *param*: A single or a list of parameter names

togrid(*grid=None, mask=None, cgrid=False, proj=False, **kwargs*)
Interpolate to a regular grid

- **grid**: The output grid. It can be either:
  - a (x,y) tuple or a grid or a cdms variable with a grid,
  - `None`, thus guessed using `grid()`
- **mask**: It can be either:
  - `None`, `False` or `MV2.nomask`: no masking
  - an array: this mask array is directly applied
  - a `Shapes` instance (or `ShoreLine`) or a single char GSHHS resolution (and optionally 's' for Histolitt)
  - a callable fonction so that `mask = thisfunc(mask, **kwmask)`
  - a float: data with this value are masked
- *mask_<param>*: <param> is passed to `polygon_mask()` for evaluation of mask thanks to the polygons.
- *grid_<param>*: <param> is passed to `grid()`.
- *cgrid*: If `True`, returns bathy at U- and V-points, else at T-points
- Other keyparam are passed to `griddata()` for regridding.

Return: `(Zx,Zy)` OR `Z` depending on cgrid.

x(*mask=True*)
Get valid X positions

xmax(*mask=True*)

xmin(*mask=True*)

xy(*mask=True*)
Return coordinates as a (2, npts) array

- xy()[0]: X
- xy()[1]: Y

xyz(*mask=True*)
Return coordinates and data as a (3, npts) array

- xy()[0]: X
- xy()[1]: Y
- xy()[2]: Z

y(*mask=True*)
Get valid Y positions

ymax(*mask=True*)

ymin(*mask=True*)

z(*mask=True*)
Get valid Z values

zmax(*mask=True*)

zmin(*mask=True*)

zone(*poly=False, mask=True*)
Get xmin,ymin,xmax,ymax

- *poly*: if True, return zone as a Polygon instance

class XYZBathyBankClient(*bank, id, **kwargs*)
Bases: object

An single element (bathy infos) of the XYZBathyBank

- **bank**: A bank instance.
- **id**:Id of this bathy.
- Other keywords are set as attributes (like long_name, etc).

Usage

Let xyzbc being an element of the bank.

```
>>> print xyzbc
>>> xyzbc.long_name = 'New long name'
>>> xyzbc.xmax = -2.
>>> xyzbc.transp = False
>>> xyzbc.id = 'iroise'              # Rename in the bank
>>> xyzbc.xyzfile = 'newfile.mnt'    # xmin, xmax, ymin, ymax updated
>>> xyz = xyzbc.load()
```

load(*zone=None, margin=None*)
Load the bathymetry as a XYZBathy instance

See Also:

clip()

plot(*id, **kwargs*)
Load and plot this bathy

class XYZBathyBank(*cfgfile='/home2/amzer/raynaud/misc/bathy/XYZBathyBank.cfg'*)
Bases: object

Bank of XYZ bathymetry infos

See Also:

XYZBathyBankClient

Usage

```
>>> xyzb = XYZBathyBank()
```

add(*xyzfile, id=None, force=False, **kwargs*)
Add a bathy to the bank

- **xyzfile**: xyz file.
- ***id*: Id of the bank. If None, it is guessed from the file name:** `xyzfile="/path/toto.xyz" -> id="toto"`

copy(*id1, id2*)
Copy bathy infos to another

get_order(*order*)
Get a list of ids that define the order

ids()
Return the list of bathy ids

list()
Get all clients of the bank as a list

load(*id*)
Load a bathymetry as a `XYZBathy` instance

merger(***kwargs*)
Return a `XYZBathyMerger` instance using the current bank
Keywords are passed to `select()`

plot(***kwargs*)
Plot current bathies using a `XYZBathyMerger` instance

remove(*id*)
Remove a bathy from the bank
id: A bathy id or `XYZBathyBankClient` instance.

save(*cfgfile=None*)

select(*xmin=None, xmax=None, ymin=None, ymax=None, load=True, margin=None, ordered=None*)
Return a list of bathymetries relevant to the specified region

- *margin*: Margin for loading xyz bathies:
  - if None, the whole bathy is loaded from the bank,
  - else it should be a value relative to the approximative resolution (see `XYZBathy.resol()`)

set_order(*order*)
Set a list of ids to define the order

update_order()

class XYZBathyMerger(*xmin=None, xmax=None, ymin=None, ymax=None, long_name=None*)
Bases: `actimar.misc.io.XYZMerger`

Mix different bathymetries

append(*d*)
Append a dataset to the merger

bathy(*grid=None*)
Get a gridded bathymetry

clean()
Remove all current dataset

copy()

from_bank(*margin=5*)
Append all bathymetries from the bank that cover the grid

- *margin*: Relative margin for clipping datasets from the bank (see `clip()` and `resol()`)

All parameters are passed to `XYZBathyBank.select()`

ids()

merge(***kwargs*)
Shortcut to `xyz()`

plot(*color=None, marker=None, mode='cluster', title='XYZ merger', m=True, show=True, colorbar=True, savefig=None, savefigs=None, **kwargs*)

- *alpha*: Alpha transparency:
  - applied to **all** points if `mode="cluster"`
  - applied to **hidden** points if `mode="data"`
- *mode*: Display mode:
  - **`"cluster"`: Points from different datasets have different colors and markers,** and hidden points are transparent.
  - **`"data"`: Points have the same marker, colors depends on Z value and hidden** points are masked.
- *marker*: Define a single or several markers to be used.
- *legend*: Show a legend if `mode="cluster"`.
- *title*: Title of the plot.
- *m*: `Basemap` instance.
- *m_margin*: Margin for `m`, relative to the mean resolution (see `XYZ.resol()`)
- *m_<keywords>*: Keywords are passed to `map()`.
- Extra keywords are passed to `XYZ.plot()`.

remove(*d*)
Remove a dataset from the merger

togrid(**args, **kwargs*)
Interpolate merged bathymetries to a grid

tolist()
Return the merger as a list of datasets

xyz(*mask=True, **kwargs*)
Merge current dataset

class **GriddedBathy**(*var, shoreline=None*)
Bases: `actimar.bathy.bathy._GriddedBathyMasked_`

Interface to gridded bathymetries

Usage

```
>>> b = GriddedBathy(var)
>>> b.apply_mask('f')
>>> b.plot_bathy()
>>> b.save('bathy.nc')
>>> newvar = b.bathy()
```

bathy(*mask=True, id=None, long_name=None*)
Get the bathymetry variable

- *mask*: If True and if a shoreline is set (with `set_shoreline()`), apply a mask due to this shoreline.

get_grid()

get_lat()

get_lon()

get_res()

get_shoreline_mask()
Get the shoreline mask

masked_bathy(*id=None, long_name=None*)
Get a masked version of the bathymetry

plot(*mask=True, id=None, long_name=None, **kwargs*)
Plot using `plot_bathy()`

regrid(*grido, method='auto', mask=True, id=None, long_name=None, **kwargs*)
Regrid bathy to another grid

reset_shoreline()
Remove the shoreline

save(*ncfile, mask=True, **kwargs*)
Save bathy to a netcdf file

- **ncfile**: Output netcdf file name
- Other keywords are passed to `bathy()`

set_shoreline(*shoreline, margin=2*)
Set the shoreline that to create a mask

- **shoreline**: `ShoreLine` instance or argument to `get_shoreline()`

Note: The mask is applied only if `masked_bathy()` is called.

class GriddedBathyMerger(*grid, shoreline=None, id=None, long_name=None*)
Bases: `actimar.misc.grid.regridding.GriddedMerger`, `actimar.bathy.bathy.GriddedBathy`

Merger of gridded variables

Usage

```
>>> merger = GriddedMerger(mygrid, long_name='My bathy')
>>> merger += etopo2(lon=(-10,0),lat=(42,50))
>>> merger += my_bathy
>>> merger.set_shoreline('i')
>>> merged_bathy = merger.bathy()
```

add(**args, **kwargs*)
Alias for `append()`

append(*var, method='auto', **kwargs*)
Append a bathymetry to the top of the merger

bathy(**args, **kwargs*)
Shortcut to `merge()`

get_grid()
Get the grid for merging

get_lat()
Get the loatitudes of the grid

get_lon()
Get the longitudes of the grid

get_res()

get_shoreline_mask()
Get the shoreline mask

insert(*idx, var*)

masked_bathy(*id=None, long_name=None*)
Get a masked version of the bathymetry

merge(*res_ratio=0.5, mask=True, id=None, long_name=None*)
Merge all variable onto the grid and apply final mask

plot(***kwargs*)
Plot merged bathy

See Also:

`GriddedBathy.plot()`

regrid(*grido, method='auto', mask=True, id=None, long_name=None, **kwargs*)
Regrid bathy to another grid

remove(*var*)

reset_shoreline()
Remove the shoreline

save(*ncfile, mask=True, **kwargs*)
Save bathy to a netcdf file

- **ncfile**: Output netcdf file name
- Other keywords are passed to `bathy()`

set_grid(*grid*)
Set the grid for merging

set_shoreline(*shoreline, margin=2*)
Set the shoreline that to create a mask

- **shoreline**: `ShoreLine` instance or argument to `get_shoreline()`

Note: The mask is applied only if `masked_bathy()` is called.

2.2.2 actimar.bathy.shorelines — Traits de côte

Utilities to handle shorelines

class Histolitt(*input=None, *args, **kwargs*)
Bases: `actimar.bathy.shorelines.ShoreLine`

Shoreline of France from SHOM/IGN at 1/25000from shapefile of Polygons covering metropolitan France

bathy(*factor=None, ext=True, nl=True, xyz=True, **kwargs*)
Get bathymetry at coast from mean sea level at stations

- *factor*: Multiplicative correction factor
- *xyz*: If True, return a `XYZ` instance instead of a `numpy` array
- *ext*: Allow extrapolation
- *nl*: Nonlinear interpolation

Return A `XYZ` instance of sea level

clip(*zone, **kwargs*)
Clip to zone

get_data(*key=None*)
Get the numeric version of the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_factor()
Get factor to apply to the sea level on the shoreline to convert to bathymetic 'depth'

get_map()
Return the associated basemap instance if set

get_points(*key=None, split=True*)
Get all the points from all the shapes as a tuple (x,y)

get_projs()
Get the list on projections (proj instance, inverse flag)

get_shapes(*key=None*)
Get the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_type()
Return the type of shapes

- 0 = Points,
- 1 = LineStrings = PolyLines
- 2 = Polygons

greatest_polygon()
Get the greatest polygon (only works with polygons!)

plot(*select=None, ax=None, fill=None, fillcolor=None, points=False, m=None, show=False, **kwargs*)
Plot shapes

- *select*: argument for selecting shapes in the list [defaul: None].
- *fill*: Force filling (True/False), else guessed from shpe type, ie filling for polygons only [default: None]
- *ax*: Axes instance [default: None]
- Other keyparams are used for plotting (lines or polygons)

project(*proj, inverse=False*)
Project shapes using proj

- **proj**: A Basemap instance or pure projection instance (from Basemap)
- *inverse*: Inverse projection [default: False]

resol(*deg=True*)
Compute the mean "resolution" of the shapes based on the first shape

- **deg**:
  - if `False`: return a resolution in meters has a the median distance between points
  - if `True`: return the median distance between points as a resolution in degrees `(xres,yres)`

set_factor(*factor*)
Set factor to apply to the sea level on the shoreline to convert to bathymetic 'depth'

sort(*reverse=True*)
Sort shapes according to their surface

- *reverse*: If True, greater polygons are first [default: True]

sorted()

xy(*key=None*)
Shortcut to `get_points(split=false)`

xyz(**args, **kwargs*)
Shortcut to `bathy()`

class EUROSION(*input=None, *args, **kwargs*)
Bases: `actimar.bathy.shorelines.ShoreLine`

Shoreline of Europe at 1/100000 from shapefile of LineStrings .. warning:

```
Must not be used for masking, only for coastal interpolations
```

bathy(*factor=None, ext=True, nl=True, xyz=True, **kwargs*)
Get bathymetry at coast from mean sea level at stations

- *factor*: Multiplicative correction factor
- *xyz*: If True, return a `XYZ` instance instead of a `numpy` array
- *ext*: Allow extrapolation
- *nl*: Nonlinear interpolation

Return A `XYZ` instance of sea level

clip(*zone, **kwargs*)
Clip to zone

get_data(*key=None*)
Get the numeric version of the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_factor()
Get factor to apply to the sea level on the shoreline to convert to bathymetic 'depth'

get_map()
Return the associated basemap instance if set

get_points(*key=None, split=True*)
Get all the points from all the shapes as a tuple (x,y)

get_projs()
Get the list on projections (proj instance, inverse flag)

get_shapes(*key=None*)
Get the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_type()
Return the type of shapes

- 0 = Points,
- 1 = LineStrings = PolyLines
- 2 = Polygons

greatest_polygon()
Get the greatest polygon (only works with polygons!)

plot(*select=None, ax=None, fill=None, fillcolor=None, points=False, m=None, show=False, **kwargs*)
Plot shapes

- *select*: argument for selecting shapes in the list [defaul: None].
- *fill*: Force filling (True/False), else guessed from shpe type, ie filling for polygons only [default: None]
- *ax*: Axes instance [default: None]
- Other keyparams are used for plotting (lines or polygons)

project(*proj, inverse=False*)
Project shapes using proj

- **proj**: A Basemap instance or pure projection instance (from Basemap)
- *inverse*: Inverse projection [default: False]

resol(*deg=True*)
Compute the mean "resolution" of the shapes based on the first shape

- **deg**:
  - if `False`: return a resolution in meters has a the median distance between points
  - if `True`: return the median distance between points as a resolution in degrees `(xres,yres)`

set_factor(*factor*)
Set factor to apply to the sea level on the shoreline to convert to bathymetic 'depth'

sort(*reverse=True*)
Sort shapes according to their surface

- *reverse*: If True, greater polygons are first [default: True]

sorted()

xy(*key=None*)
Shortcut to `get_points(split=false)`

xyz(**args, **kwargs*)
Shortcut to `bathy()`

class GSHHS_SF(*input=None, *args, **kwargs*)
Bases: `actimar.bathy.shorelines.ShoreLine`

Fine world shoreline from USGS shapefile .. warning:

```
HUGE! Please use :class:'GSHHS' instead
```

bathy(*factor=None, ext=True, nl=True, xyz=True, **kwargs*)
Get bathymetry at coast from mean sea level at stations

- *factor*: Multiplicative correction factor
- *xyz*: If True, return a `XYZ` instance instead of a `numpy` array

- *ext*: Allow extrapolation
- *nl*: Nonlinear interpolation

Return A XYZ instance of sea level

clip(*zone, **kwargs*)
Clip to zone

get_data(*key=None*)
Get the numeric version of the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_factor()
Get factor to apply to the sea level on the shoreline to convert to bathymetic 'depth'

get_map()
Return the associated basemap instance if set

get_points(*key=None, split=True*)
Get all the points from all the shapes as a tuple (x,y)

get_projs()
Get the list on projections (proj instance, inverse flag)

get_shapes(*key=None*)
Get the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_type()
Return the type of shapes

- 0 = Points,
- 1 = LineStrings = PolyLines
- 2 = Polygons

greatest_polygon()
Get the greatest polygon (only works with polygons!)

plot(*select=None, ax=None, fill=None, fillcolor=None, points=False, m=None, show=False, **kwargs*)
Plot shapes

- *select*: argument for selecting shapes in the list [defaul: None].
- *fill*: Force filling (True/False), else guessed from shpe type, ie filling for polygons only [default: None]
- *ax*: Axes instance [default: None]
- Other keyparams are used for plotting (lines or polygons)

project(*proj, inverse=False*)
Project shapes using proj

- **proj**: A Basemap instance or pure projection instance (from Basemap)
- *inverse*: Inverse projection [default: False]

resol(*deg=True*)
Compute the mean "resolution" of the shapes based on the first shape

- **deg**:
  - if `False`: return a resolution in meters has a the median distance between points
  - if `True`: return the median distance between points as a resolution in degrees `(xres,yres)`

set_factor(*factor*)
Set factor to apply to the sea level on the shoreline to convert to bathymetic 'depth'

sort(*reverse=True*)
Sort shapes according to their surface

- *reverse*: If True, greater polygons are first [default: True]

sorted()

xy(*key=None*)
Shortcut to `get_points(split=false)`

xyz(**args, **kwargs*)
Shortcut to `bathy()`

class GSHHS(*input=None, clip=None, sort=True, reverse=True, **kwargs*)
Bases: `actimar.misc.grid.basemap.GSHHS_BM`, `actimar.bathy.shorelines._CoastalBathy_`

bathy(*factor=None, ext=True, nl=True, xyz=True, **kwargs*)
Get bathymetry at coast from mean sea level at stations

- *factor*: Multiplicative correction factor
- *xyz*: If True, return a `XYZ` instance instead of a `numpy` array
- *ext*: Allow extrapolation
- *nl*: Nonlinear interpolation

Return A `XYZ` instance of sea level

clip(*zone, **kwargs*)
Clip to zone

get_data(*key=None*)
Get the numeric version of the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_factor()
Get factor to apply to the sea level on the shoreline to convert to bathymetic 'depth'

get_map()
Return the associated basemap instance if set

get_points(*key=None, split=True*)
Get all the points from all the shapes as a tuple (x,y)

get_projs()
Get the list on projections (proj instance, inverse flag)

get_shapes(*key=None*)
Get the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_type()
Return the type of shapes

- 0 = Points,
- 1 = LineStrings = PolyLines
- 2 = Polygons

plot(*select=None, ax=None, fill=None, fillcolor=None, points=False, m=None, show=False, **kwargs*)
Plot shapes

- *select*: argument for selecting shapes in the list [defaul: None].
- *fill*: Force filling (True/False), else guessed from shpe type, ie filling for polygons only [default: None]
- *ax*: Axes instance [default: None]
- Other keyparams are used for plotting (lines or polygons)

project(*proj, inverse=False*)
Project shapes using proj

- **proj**: A Basemap instance or pure projection instance (from Basemap)
- *inverse*: Inverse projection [default: False]

resol(*deg=True*)
Compute the mean "resolution" of the shapes based on the first shape

- **deg**:
  - if `False`: return a resolution in meters has a the median distance between points

–if `True`: return the median distance between points as a resolution in degrees `(xres,yres)`

set_factor(*factor*)
Set factor to apply to the sea level on the shoreline to convert to bathymetic 'depth'

sort(*reverse=True*)
Sort shapes according to their surface

- *reverse*: If True, greater polygons are first [default: True]

sorted()

xy(*key=None*)
Shortcut to `get_points(split=false)`

xyz(**args, **kwargs*)
Shortcut to `bathy()`

class GSHHS_BM(*input=None, clip=None, sort=True, reverse=True, **kwargs*)
Bases: `actimar.misc.io.Shapes`

Shoreline from USGS using Basemap

Initialized with a valid Basemap instance with resolution not equal to None, or thanks to arguments passed to `actimar.misc.plot.map()`

- *m*: Basemap instance [default: None]

clip(*zone, **kwargs*)
Clip to zone

get_data(*key=None*)
Get the numeric version of the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_map()
Return the associated basemap instance if set

get_points(*key=None, split=True*)
Get all the points from all the shapes as a tuple (x,y)

get_projs()
Get the list on projections (proj instance, inverse flag)

get_shapes(*key=None*)
Get the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_type()
Return the type of shapes

- 0 = Points,
- 1 = LineStrings = PolyLines
- 2 = Polygons

plot(*select=None, ax=None, fill=None, fillcolor=None, points=False, m=None, show=False, **kwargs*)
Plot shapes

- *select*: argument for selecting shapes in the list [defaul: None].
- *fill*: Force filling (True/False), else guessed from shpe type, ie filling for polygons only [default: None]
- *ax*: Axes instance [default: None]
- Other keyparams are used for plotting (lines or polygons)

project(*proj, inverse=False*)
Project shapes using proj

- **proj**: A Basemap instance or pure projection instance (from Basemap)
- *inverse*: Inverse projection [default: False]

resol(*deg=True*)
Compute the mean "resolution" of the shapes based on the first shape

- **deg**:
  - if `False`: return a resolution in meters has a the median distance between points
  - if `True`: return the median distance between points as a resolution in degrees `(xres,yres)`

sort(*reverse=True*)
Sort shapes according to their surface

- *reverse*: If True, greater polygons are first [default: True]

sorted()

xy(*key=None*)
Shortcut to `get_points(split=false)`

class ShoreLine(*input, m=None, inverse=False, clip=True, shapetype=None, min_area=None, sort=True, reverse=True, samp=1*)
Bases: `actimar.misc.io.Shapes`, `actimar.bathy.shorelines._CoastalBathy_`

Version of Shapes dedicated to shorelines

bathy(*factor=None, ext=True, nl=True, xyz=True, **kwargs*)
Get bathymetry at coast from mean sea level at stations

- *factor*: Multiplicative correction factor
- *xyz*: If True, return a `XYZ` instance instead of a `numpy` array
- *ext*: Allow extrapolation
- *nl*: Nonlinear interpolation

Return A `XYZ` instance of sea level

clip(*zone, **kwargs*)
Clip to zone

get_data(*key=None*)
Get the numeric version of the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_factor()
Get factor to apply to the sea level on the shoreline to convert to bathymetic 'depth'

get_map()
Return the associated basemap instance if set

get_points(*key=None, split=True*)
Get all the points from all the shapes as a tuple (x,y)

get_projs()
Get the list on projections (proj instance, inverse flag)

get_shapes(*key=None*)
Get the list of geos objects (polygons, etc)

- *key*: A slice selector applied to the list.

get_type()
Return the type of shapes

- 0 = Points,
- 1 = LineStrings = PolyLines
- 2 = Polygons

greatest_polygon()
Get the greatest polygon (only works with polygons!)

plot(*select=None, ax=None, fill=None, fillcolor=None, points=False, m=None, show=False, **kwargs*)
Plot shapes

- *select*: argument for selecting shapes in the list [defaul: None].
- *fill*: Force filling (True/False), else guessed from shpe type, ie filling for polygons only [default: None]
- *ax*: Axes instance [default: None]

- Other keyparams are used for plotting (lines or polygons)

project(*proj, inverse=False*)
Project shapes using proj

- **proj**: A Basemap instance or pure projection instance (from Basemap)
- *inverse*: Inverse projection [default: False]

resol(*deg=True*)
Compute the mean "resolution" of the shapes based on the first shape

- **deg**:
  - if `False`: return a resolution in meters has a the median distance between points
  - if `True`: return the median distance between points as a resolution in degrees `(xres,yres)`

set_factor(*factor*)
Set factor to apply to the sea level on the shoreline to convert to bathymetic 'depth'

sort(*reverse=True*)
Sort shapes according to their surface

- *reverse*: If True, greater polygons are first [default: True]

sorted()

xy(*key=None*)
Shortcut to `get_points(split=false)`

xyz(**args, **kwargs*)
Shortcut to `bathy()`

get_best(*lon, lat, m=None, margin=0.10000000000000001*)
Get the best shoreline for my lons/lats set

get_shoreline(*arg, *args, **kwargs*)
Get a valid ShoreLine instance according to arg.

- **arg**: it can be either:
  - A :class'~actimar.misc.io.Shapes' instance
  - A string within the folowing lists:
    - for GSHHS resolutions: ['f', 'h', 'i', 'l', 'c']
    - for other shorelines: ['Histolitt', 'EUROSION', 'GSHHS_SF', 'GSHHS', 'GSHHS_BM']

2.3 actimar.buoy — Bouées

Class getting data from buoys

Each buy type has a dedicated class to retreive its data.

class CANDHIS(*buoy_id, time=None, verbose=True*)
Bases: `actimar.buoy._Buoy_`

CANDHIS buoy data retreiver

- *buoy_id*: Buoy ID. Use `candhis_list()` to list available buoys ID.

Example

```
>>> mybuoy = buoy.ndbc('62163',('2008-08-06','2008-08-09'))
>>> mybuoy.show_variables()
>>> mybuoy.plot('baro')
>>> mybuoy.get('baro').info()
```

get(*var_name, time_range=None*)
Get a variable

- **var_name**: Name of the variable

Example:

```
>>> myvar = mybuoy.get('baro')
```

Return: A 1D cdms variable

See: plot(), show_variables()

plot(*var_name, time_range=None, **kwargs*)

Plot a variable

- **var_name**: Name of the variable
- *time*: Plot only within this time range (like ('2007-01-01','2007-02-01','co')
- *show*: Show the figure [default: None]
- All other keywords are passed to actimar.misc.plot.curve()

Example:

```
>>> myvar = mybuoy.plot('baro')
```

See: get(), show_variables()

save(*file_name, mode='w', warn=True*)

Save to file

show_variables()

Print available variables

class ChannelCoast(*buoy, time_range, cache_file=None, **kwargs*)

Bases: actimar.buoy._Buoy_

ChannelCoast buoy data retreiver

- *buoy_id*: Buoy ID. Use channelcoast_list() to list available buoys ID.

Example:

```
>>> mybuoy = buoy.ndbc('62163',('2008-08-06','2008-08-09'))
>>> mybuoy.show_variables()
>>> mybuoy.plot('hmax')
>>> mybuoy.get('hmax').info()
```

check_cache(*time_range=None*)

Update the cache

get(*var_name, time_range=None*)

Get a variable

load(*time_range*)

Check if time_range is included in in-memory variables, else, check cache and load from it

plot(*var_name, time_range=None, **kwargs*)

Plot a variable

save(*file_name=None, *args, **kwargs*)

Save to cache

show_variables()

Print available variables

class NDBC(*buoy_id, time=None*)

Bases: actimar.buoy._Buoy_

NDBC buoy data retreiver

- *buoy_id*: Buoy ID. Use ndbc_list() to list available buoys ID.

Example:

```
>>> mybuoy = buoy.ndbc('62163',('2008-08-06','2008-08-09'))
>>> mybuoy.show_variables()
>>> mybuoy.plot('baro')
>>> mybuoy.get('baro').info()
```

get(*var_name, time_range=None*)
Get a variable

- **var_name**: Name of the variable

Example:

```
>>> myvar = mybuoy.get('baro')
```

Return: A 1D cdms variable

See: `plot()`, `show_variables()`

plot(*var_name, time_range=None, **kwargs*)
Plot a variable

- **var_name**: Name of the variable
- *time*: Plot only within this time range (like ('2007-01-01','2007-02-01','co')
- *show*: Show the figure [default: None]
- All other keywords are passed to `actimar.misc.plot.curve()`

Example:

```
>>> myvar = mybuoy.plot('baro')
```

See: `get()`, `show_variables()`

save(*file_name, mode='w', warn=True*)
Save to file

show_variables()
Print available variables

candhis_list()
Print available `CANDHIS` buoys

channelcoast_list(*pattern=None*)
Print available ChannelCoast buoys

- *keyparam*: List only buoy matching this RE pattern [default: None]

channelcoast_update_all(*pattern=None, timestamp=False, **kwargs*)
Update cache for all `ChannelCoast` buoys

ndbc_list()
print available `NDBC` buoys

2.4 actimar.tide — Marée

2.4.1 actimar.tide.station_info — Station et ports

2.4.2 :mod:‘actimar.tide.shom — Données et formats du SHOM

2.4.3 actimar.tide.sonel_mareg — Réseau Sonel

2.4.4 actimar.tide.filters — Les filtres

2.4.5 actimar.tide.marigraph — Outil marégraphique

2.5 actimar.meteo — Météo

2.5.1 actimar.meteo.metar — Outils METAR (données, stations)

class **Metar**(*station=None*)
Bases: object

classe definissant l'objet metar

getData(*stationID*)

myprint()

readData(*filename*)

writeData(*path='.'*)

class **MetarStation**(*arg1=None, arg2=None, **kwargs*)
Bases: object

dumpKML(*file_name, *args, **kwargs*)
Dump selected stations stations to a .kml file

@param file_name KML file name All other arguments and keywords are used for searching

dumpKMZ(**args, **kwargs*)
Dump selected stations stations to a .kmz file

@param file_name KMZ file name All other arguments and keywords are used for searching

findClosest(*lon, lat, verbose=True, radius=None*)
Search for the closest station

@param lon Longitude in degrees @param lat Latitude in degrees @keyparam verbose Display distance to the station @keyparam radius Max distance in km to select several stations

findIATA(*id*)
Search using IATA id

findICAO(*icao*)
Search using ICAO id

findInRange(*lonmin, lonmax, latmin, latmax, verbose=True*)
Search for all the stations in the specified area

@param lonmin, lonmax range of longitude (in degrees) @param latmin, latmax range of latitude (in degrees) @keyparam verbose Display the list of stations

findStationName(*station_name*)
Search using regexp matching on station name

findSynop(*id*)
Search using Synop id

class **Taf**(*station=None*)
Bases: object

getData(*stationID*)

dewPoint(*temperature, humidity*)

humidity(*temperature, dewPoint*)
Calculate humidity from temperature and dewpoint This is only an approximation, there is no exact formula, this one here is called Magnus-Formula http://www.faqs.org/faqs/meteorology/temp-dewpoint/

sunSetRise(*date=None, latitude=None, longitude=None, sunRise=True*)
Calculates sunrise and sunset for a location

windChill(*temperature, speed*)
Calculate windchill from temperature and windspeed (enhanced formula) http://www.nws.noaa.gov/om/windchill/

2.5.2 actimar.meteo.wunderground — Données de wunderground

Get daily, weekly and monthly data from wunderground

class WUnderground(*station_search, start_date, end_date=None, verbose=False*)
Bases: object

Class to retreive METAR hisorical data from htt://www.wunderground.com

@usage: >>> # Load data >>> from actimar.meteo import WUnderground ; import cdms2 as cdms >>> brest = WUnderground('Brest','2006-05','2007-06-15') >>> # Show information >>> print brest >>> brest.plot('temp_mean') >>> # Save mean temperature >>> f = cdms.open('temp_mean.nc') >>> f.write(brest('temp_mean') >>> f.write()

get(*var_name*)

plot(*var_name, **kwargs*)
Plot the time series of a variable

All keywords are passed to misc.plot.curve()

variables()
Return the list of available variables

2.6 mars — Outils pour MARS

2.6.1 actimar.mars.ranks — Gestion des rangs

Get grid information

class Rank(*rank=None, m=None, name=None, **kwargs*)
Bases: object

Create one rank

Example:

```
>>> rang = Rank(lat_min=47, lon_max=-4.)
>>> rang['lat_max'] = 49.
>>> rang.lon_min = -7.
>>> print rang
etc...
```

getGrid(*pt='t', ext=False*)
Get the grid at point <pt>

- *pt*: Point where to get the grid (within 't', 'u', 'v') [default: 't']
- *ext*: See getLongitude() and getLatitude() [default: False]

getLatitude(*pt='t'*)
Get the latitude axis

- *pt*: If pt is 't' or 'u', return at latitude at T-points, else returns a V-points [default: 't']

getLongitude(*pt='t'*)
Get the longitude axis

- *pt*: If pt is 't' or 'v', return at longitude at T-points, else returns a U-points [default: 't']

get_name(*parent=None, details=False*)

isinside(*rank*)
Check if current rank is inside specified rank

- **rank**: A Rank object.

isset()
To check if this Rank is correctly set to be used

show(*alpha=1, color='r', frame_width=2, show=True, xlim=None, ylim=None, m=None, figure=None, title=None, margin=0.050000000000000003, bgmode=True, **kwargs*)
Display the rank domain

- *xlim*: Limits of the plot along X [defaults from grid]
- *ylim*: Limits of the plot along Y [defaults from grid]
- *figure*: Force to plot on this figure [default: None]
- *alpha*: Transparency [default: 1]
- *color*: Color of grid and frame [default: 'r']
- *frame_width*: Line width of the frame [default: 2]
- Other keywords are passed to misc.plot.map()

class Ranks(*input=None, title=None, graphic_specs=, [('#888888', 0.80000000000000004), ('#bb4444', 0.90000000000000002), ('#44bb44', 0.40000000000000002), ('b', 0.40000000000000002), ('#EE82EE', 0.40000000000000002), ('#40E0D0', 0.40000000000000002), ('#FFA500', 0.40000000000000002), ('#FFFF00', 0.40000000000000002), ('#00008B', 0.40000000000000002), ('#A52A2A', 0.40000000000000002), ('#000000', 0.40000000000000002), ('#0000FF', 0.40000000000000002), ('#008000', 0.40000000000000002), ('#FF0000', 0.40000000000000002)]*)
Bases: `object`

Get information about ranks

Example:

```
>>> # Load module
>>> from actimar.mars.ranks import Ranks
>>> # Load file and create object
>>> previd1 = Ranks('/path/cool/entrees/head.cool','Mon config de MARS')
>>> # Show information about all ranks
>>> print previd1.nranks
>>> print previd1
>>> previd1.show()
>>> # Show  information about one rank
>>> print previd1[3]
>>> print 'Longitude max =',previd1[3].lon_max
>>> print previd1[3].keys
>>> help(previd1[3].show)
>>> print previd1[3].show([1,2,3]textcolor='b',gridcolor='k',resolution='c')
```

add_rank(*rank, update=True*)
Add (insert) a rank to the current configuration

del_rank(*irank*)
Delete a rank - **irank**: rank number

load(*file*)
Load configuration from a file

save(*headfile, suffix=None*)
Write configuration to a file

- **headfile**: Configuration file.
- *suffix*: If given, it is appent to the file name (<headfile>.<suffix>), else it is guessed from file name.

show(*ranks=None, show=True, xlim=None, ylim=None, colors=None, alphas=None, res='auto', margin=0.050000000000000003, legend_loc='upper left', details=False, legend=True, **kwargs*)
Show a map of the ranks

- *ranks*: Ranks to show. It can be an index, or a external Rank, or a list of them. If None, all internal ranks are shown [default: None]
- *xlim*: Limits of the plot along X [defaults from grid]
- *ylim*: Limits of the plot along Y [defaults from grid]
- *legend_loc*: Location of the legend (see pylab.legend()) [default: 'upper left']
- *show*: Show the figure [default: True]
- *legend*: Show the lgend or not [default: True]
- *legend_<keyword>*: <keyword> is passed to pylab.figlegend.
- All other keywords are passed to <ranks_obj>[i].show()

update()
Sort ranks and refine nesting

class PreviD1()
Bases: actimar.mars.ranks.Ranks

Rangs du demonstrateur 1 de Previmer

add_rank(*rank, update=True*)
Add (insert) a rank to the current configuration

del_rank(*irank*)
Delete a rank - **irank**: rank number

load(*file*)
Load configuration from a file

save(*headfile, suffix=None*)
Write configuration to a file

- **headfile**: Configuration file.
- *suffix*: If given, it is appent to the file name (<headfile>.<suffix>), else it is guessed from file name.

show(*ranks=None, show=True, xlim=None, ylim=None, colors=None, alphas=None, res='auto', margin=0.050000000000000003, legend_loc='upper left', details=False, legend=True, **kwargs*)
Show a map of the ranks

- *ranks*: Ranks to show. It can be an index, or a external Rank, or a list of them. If None, all internal ranks are shown [default: None]
- *xlim*: Limits of the plot along X [defaults from grid]
- *ylim*: Limits of the plot along Y [defaults from grid]
- *legend_loc*: Location of the legend (see pylab.legend()) [default: 'upper left']
- *show*: Show the figure [default: True]
- *legend*: Show the lgend or not [default: True]
- *legend_<keyword>*: <keyword> is passed to pylab.figlegend.
- All other keywords are passed to <ranks_obj>[i].show()

update()
Sort ranks and refine nesting

2.6.2 actimar.misc.axes — Génération de configurations

config(*config, suffixe*)

def_config()

2.6.3 actimar.mars.read — Lecture de fichiers

Tools for reading MARS outputs

previd1(*var_name, rank=None, mars2d=False, suffix=None, caparmor=True, **kwargs*)
Read a variable from MARS Previd1 files

- **var_name**: Name of the variable
- *day*: Select the current day if 0, else one of the following days. Must between -1 and 1 [default: 0]
- *suffix*: Suffix (like "pipo.<suff>.nc")
- *lon/lat/z/time*: Range selectors
- *day_bounds*: Extend/restrict the selected day range. For example, use ((6,cdtime.Hour),(-6,cdtime.Hour)) to restrict day from 6am to 17am included. If day is extended, it implies an time_average=True.
- *daily_average*: Perform an average over the selcted day.
- *daily_frequency*: Intraday averages at this frequency (for example, 2 means 12h averages). It create proper time bounds.
- *first*: Read only the first file found [default: False]. Becomes True if time is a slice.
- *new_grid*: Regrid on this rectangular grid
- *quiet*: Quiet mode
- *verbose*: More verbose
- *regular*: Fill gaps with missing value to have a regular time axis
- *mars2d*: Search in MARS2D instead of MARS3D files [default: False]
- *rank*: Model rank [default: highest]
- *suffix*: Suffix (like "pipo.<suff>.nc")
- *caparmor*: Use caparmor outputs (instead of nymphea) [default: True]

MODULE INDEX

A

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z