CDO’s python bindings

Ralf Müller

MPI Met Hamburg

October 17, 2018
What to expect

Overview:
- General features
- Installation
- What it’s not

Source Code Examples:
- Basics Usage
- Work with temporary files
- Parallelization with Python
- Integration with numpy/xarray/...
... is offered

- integration of CDO into python/ruby scripts like a native library
WHAT ... 

... is offered

- integration of CDO into python/ruby scripts like a native library
- keeps CDOs main feature: **operator chaining**
WHAT ...

... is offered

- integration of CDO into python/ruby scripts like a native library
- keeps CDOs main feature: **operator chaining**
- multiple types of return values:
  - output files, numpy arrays, masked arrays, XArray
  - netCDF4 or XDataset handles
  - strings for operators, which write to stdout
  - None on error (optional)
WHAT ... is offered

- integration of CDO into python/ruby scripts like a native library
- keeps CDOs main feature: **operator chaining**
- multiple types of return values:
  - output files, numpy arrays, masked arrays, XArray
  - netCDF4 or XDataset handles
  - strings for operators, which write to stdout
  - None on error (optional)
- access to all options
  - -f file format
  - -P OpenMP-threads
  - ...
- environment settings
- GPL-2 licensed like CDO itself
... to get it

- prebuild debian packages: python-cdo, python3-cdo
- installation via pip or conda (conda-forge)
- or spack (https://spack.io)
... to get it
- prebuild debian packages: python-cdo, python3-cdo
- installation via pip or conda (conda-forge)
- or spack (https://spack.io)

... to work with it
- IO: provide automatic tempfile handling
- IO: optional use of existing files if present
- interactive help
- use different CDO binaries for different tasks
cdo.{rb,py}

- is a *smart* caller of a CDO binary (with all the pros and cons)
- doesn’t need to be re-installed for a new CDO version
- isn’t a shared library, which keeps everything in memory
- doesn’t allow write access to files via the numpy or masked arrays

See MPI-MET ort github page:

https://code.zmaw.de/projects/cdo/wiki/Cdo{rbpy}
https://github.com/Try2Code/cdo-bindings

Ralf Müller (MPI Met Hamburg)
```python
from cdo import Cdo
import glob

cdo = Cdo()

# use a special binary
cdo.setCdo('/sw/rhel6-x64/cdo/cdo-1.9.5-gcc64/bin/cdo')

# concatenate list of files into a temp file with relative time axis
ofile = cdo.cat(input = glob.glob('*\.nc'), options = '-r')

# vertical interpolation
Temp3d = cdo.intlevel(100,200,500,1000, options = '-f grb',
                      input = ofile,
                      output = 'TempOnTargetLevels.grb')

# perform zonal mean after interpolation in nc4 classic format with 8 OpenMP threads
zonmeanFile = cdo.zonmean(input = '-remapbil,r1400x720 %s%(Temp3d),
                          options = '-P 8 -f nc4c')
```
Using tempfiles can become a problem

Tempfiles are usually removed at the end of a script. But in long-lasting or SIGKILLed interactive session (ipython/jupyter notebooks) with possibly many users per node the system tempdir can get filled up sooner or later. How to avoid a reboot?
Possible issues with tempfiles

Using tempfiles can become a problem

Tempfiles are usually removed at the end of a script. But in long-lasting or SIGKILLed interactive session (ipython/jupyter notebooks) with possibly many users per node the system tempdir can get filled up sooner or later. How to avoid a reboot?

Solution

Manual clean-up for all files created by cdo.py belonging to the current user

cdo.cleanTempDir()
Possible issues with tempfiles

Using tempfiles can become a problem

Tempfiles are usually removed at the end of a script. But in long-lasting or SIGKILLed interactive session (ipython/jupyter notebooks) with possibly many users per node the system tempdir can get filled up sooner or later. How to avoid a reboot?

Solution

Manual clean-up for all files created by cdo.py belonging to the current user

```python
cdo.cleanTempDir()
```

Solution

Use other tempdir like /dev/shm

```python
cdo = Cdo(tempdir='/dev/shm/{0}'.format(os.environ['USER']))
```
from cdo import Cdo
from multiprocessing import Pool

# define methods to use with the Pool
def cdozonmean(infile):
    ofile = cdo.zonmean(input=infile)

files = sorted([s for s in glob.glob(nicam_path+'/*sa_tppn.nc'))][0:20]

# create the Pool and a dict for collecting the results
pool, results = Pool(4), dict()

# fill and run the Pool, keep the connection of input and output
for file in files:
    results[file] = pool.apply_async(cdozonmean,(file,))

pool.close()
pool.join()

# retrieve the _real_ results from the Pool (i.e. filenames)
for k,v in results.items():
    results[k] = v.get()

cdo.cat(input = [results[x] for x in files], output = wrk_dir+'test.nc')
XArray/Numpy interaction

XArray

```python
# plotting with XArray
cdo.topo(returnXArray='topo').plot()

# IO with XDataset
dataSet = xarray.open_dataset(cdo.topo('global_0.1', options = '-f nc'))
dataSet['topo'] = 1.0 + np.abs(dataSet['topo'])
cdo.fldmin(input=dataSet,returnArray='topo').min() == ?
```

numpy/matplotlib-based plotting

```python
# or with masked arrays
from matplotlib import pylab
import numpy
oro = cdo.setrtomiss(-20000,0, input='sellonlatbox,-20,60,20,60 -topo', returnMaArray='topo')
pylab.imshow(numpy.flipud(oro))
pylab.show()
```
More Examples at github

Units test for all features available at Github

- numpy or masked arrays, XArray, XDataset, cdf handles ...

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
<th>return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>returnArray</td>
<td>varname</td>
<td>numpy array</td>
</tr>
<tr>
<td>returnMaArray</td>
<td>varname</td>
<td>numpy masked array</td>
</tr>
<tr>
<td>returnXarray</td>
<td>varname</td>
<td>XArray</td>
</tr>
<tr>
<td>returnXDataset</td>
<td>Bool</td>
<td>XDataset handle</td>
</tr>
<tr>
<td>returnCdf</td>
<td>Bool</td>
<td>netCDF4 file handle</td>
</tr>
</tbody>
</table>

- conditional output
- return None on error
- exception handling
- output operators

... test code is about 1.5 times the library code
thank you for your attention

???
```python
def __init__(self,
    returnCdf = False,
    returnNoneOnError = False,
    forceOutput = True,
    cdfMod = CDF_MOD_NETCDF4,
    env = os.environ,
    debug = False,
    tempdir = tempfile.gettempdir(),
    logging = False,
    logFile = StringIO()):

    # read path to CDO from the environment if given
    if 'CDO' in os.environ:
        self.CDO = os.environ['CDO']
    else:
        self.CDO = 'cdo'
```

# always return netCDF4 filehandles
# don't raise exception, return NoneOnError
# global switch for cond. output
# set the cdf module to be used
# environment for the object
# print commands, return codes, etc.
# location for temporary files
# log commands internally
Appendix: Pool.apply_async syntax explained

```python
from multiprocessing import Pool

def f(x, *args, **kwargs):
    print x, args, kwargs

args, kw = (1,2,3), {'cat': 'dog'}

print "# Normal call"
f(0, *args, **kw)

print "# Multicall"
P = Pool()
sol = [P.apply_async(f, (x,) + args, kw) for x in range(2)]
P.close()
P.join()

for s in sol: s.get()
```
```ruby
require 'parallel'
require 'cdo'

cdo = Cdo.new
files = Dir.glob("*.nc")

ofiles = Parallel.map(files,:in_processes => nWorkers).each { |file|
  basename = file[0..-(File.extname(file).size+1)]
  ofile = cdo.remap(targetGridFile,targetGridweightsFile,
      :input => file,
      :output => "remapped_#{basename}.nc")
}

# Merge all the results together

cdo.merge(:input => ofiles.join(" "),:output => 'mergedResults.nc')
```