pyOASIS

P. Gambron, R. Ford, A. Piacentini, S. Valcke

Apr 29, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

pyOASIS is a Python wrapper for OASIS written using ctypes and ISO C bindings to Fortran. It provides an object-oriented interface to OASIS3-MCT. This allows users to write and couple models written in Python or to couple models written in Python with models written in Fortran.

It is part of the distribution of OASIS3-MCT. See https://portal.enes.org/oasis/download for more information about obtaining it.

CHAPTER

TWO

USING PYOASIS

2.1 Code structure

The source code of pyOASIS is in the directory `pyoasis/src`. First, the OASIS3-MCT Fortran code is wrapped in Fortran using ISO-C bindings. The corresponding source files are in the subdirectory `lib/cbindings/fortran_isoc`. The file names are the same as the corresponding ones in the original source code but ending in `_iso.F90`. Subsequently, the Fortran with ISO-C bindings is wrapped in C, see the source code is in `lib/cbindings/c_src`. The names of the files are the same as the corresponding Fortran ones, but ending in `_c.c`. Finally, the C is wrapped in Python in the directory `pyoasis/src`. A low-level wrapper is made using the same filenames as the Fortran ones but ending in `.py`. To access the python interface, one must write a `import pyoasis` in his/her python code.

Users can find examples on how to use the interface in python, C and Fortran in `pyoasis/examples`; to run all these examples at once, use `make test` in `oasis3-mct/pyoasis`. More advanced testing of the python wrapping functions can be done with `make wrappertest`; this will call the script `tests/run_pytest.sh` that invokes the standard `pytest` testing framework (that has to be installed).

2.2 Creating a component using MPI

In pyOASIS, components are instances of the **Component** class. To initialise a component, its name has to be supplied

```
import pyoasis
component_name = "component"
comp = pyoasis.Component(component_name)
```

It is also possible to provide an optional `coupling_flag` argument which defaults to "True", which means the component is coupled through OASIS3-MCT:

```
import pyoasis
component_name = "component"
coupling_flag = True
comp = pyoasis.Component(component_name, coupling_flag)
```

OASIS3-MCT couples models which communicate using MPI. By default, the **Component** class will set up MPI internally and provides methods to get access to information such as rank and number of processes in the local communicator gathering only the component processes

```
import pyoasis

comp = pyoasis.Component("component")

```

(continues on next page)

(continued from previous page)

```
print("Hello world from process " + str(comp.localcomm.rank)
      + " of " +  str(comp.localcomm.size))
```

If the user needs to specify that the global communicator gathering at start all components of the coupled model is different from the default MPI COMM WORLD, this can be passed to the **Component** class through the communicator optional argument. This should be created with `mpi4py`.

```
import pyoasis
from mpi4py import MPI
[...]
comm = my_global_comm

component_name = "component"
coupling_flag = True
comp = pyoasis.Component(component_name, coupling_flag, comm)
```

To create a coupling communicator for a subset of processes, one can use the method `create_couplcomm`, with a flag being `True` for all these processes:

```
coupled = True
if local_comm_size > 3:
  if local_comm_rank >= local_comm_size - 2:
      coupled = False
comp.create_couplcomm(coupled)
```

If such a communicator already exists in the code, it should simply be provided to OASIS3-MCT with the method `set_couplcomm`. Notice that the processes not involved in the coupling should still invoke this method providing the `MPI.COMM_NULL` communicator (predefined in `mpi4py`):

```
couplcomm = comp.localcomm.Split(icpl, local_comm_rank)
if icpl == 0:
    couplcomm = MPI.COMM_NULL
comp.set_couplcomm(couplcomm)
```

To set up an MPI intra-communicator or inter-communicator between the local component and another component, e.g. the component `receiver` in the example below, one can use `get_intracomm` or `get_intercomm`:

```
intracomm = comp.get_intracomm("receiver")
intercomm = comp.get_intercomm("receiver")
```

To set up an MPI intra-communicator among some of the coupled components, listed in the `comp_list` list, one can use:

```
intracomm, root_ranks = comp.get_multi_intracomm(comp_list)
```

where `root_ranks` is a dictionary (the keys are the elements of `comp_list`) providing the ranks of the component roots in the intra-communicator.

The current OASIS3-MCT internal debug level (`$NLOGPRT` value in the `namcouple`), can be retrieved as a property of a component, namely `comp.debug_level`, as in:

```
print("PyOasis debug level set to {}".format(comp.debug_level)
```

and can be changed by directly modifying, the `debug_level` property of the component:

```
comp.debug_level = 2
```

2.3 Creating a partition

The data can be partitioned in various ways. These correspond to the **SerialPartition**, **ApplePartition**, **BoxPartition**, **OrangePartition** and **PointsPartition** classes which are inherited from the **Partition** abstract class. For details on the different ways to describe the partitions, see OASIS3-MCT User Guide, section 2.2.3 and examples `1-serial`, `2-apple`, `3-box`, `4-orange`, `5-points` in `pyoasis/examples`.

The simplest situation is the serial partitioning where all the data is held by a single process and only the number of points has to be specified (see example `1-serial`)

```
n_points = 1600
serial_partition = pyoasis.SerialPartition(n_points)
```

In the case of the apple partitioning, each process contains a segment of a linear domain. To initialise such a partitioning, an offset has to be supplied for each rank as well as the number of data points that will be stored locally. The following example, close to example `2-apple`, if run with 4 processes, will produce 4 consecutive local segments containing 400 data points

```
component_name = "component"
comp = pyoasis.Component(component_name)
rank = comp.localcomm.rank
size = comp.localcomm.size
n_points = 1600
local_size = int(n_points/size)
offset = rank * local_size
partition = pyoasis.ApplePartition(offset, local_size)
```

When we use the box partitioning, a 2-dimensional domain is split into several rectangles. The global offset, local extents in the x and y directions and the global extent in the x direction have to be supplied to the constructor. The global offset is the index of the corner of the local rectangle. For example, we can split a 4x4 square domain into 4 2x2 parts with the following code that will have to be executed using 4 processes. The offset is computed from the global and local domain sizes as well as the rank. Another example of a box partition is available in example `3-box`

```
rank = comp.localcomm.rank
n_global_points_per_side = 4
n_partitions_per_side = 2
local_extent = n_global_points_per_side / n_partitions_per_side
i_partition_x = rank / n_partitions_per_side
i_partition_y = rank % n_partitions_per_side
global_offset =   i_partition_x * n_global_points_per_side * local_extent
                + i_partition_y * local_extent
global_extent_x = n_global_points_per_side
partition = pyoasis.BoxPartition(global_offset, local_extent, local_extent,
                                 global_extent_x)
```

The orange partitioning consists of several segments of a linear domain. As a consequence, a list of offsets and local sizes have to be provided. In this example, each process contains 2 consecutive segments of 2 points. (Another example with only one segment per process is available in example `4-orange`.)

```
rank = comp.localcomm.rank
n_segments_per_rank = 2
n_points_per_segment = 2
```

(continues on next page)

(continued from previous page)

```
offset_beginning = rank * n_segments_per_rank * n_points_per_segment
offset = [offset_beginning, offset_beginning + n_points_per_segment]
extents = [n_points_per_segment, n_points_per_segment]
partition = pyoasis.OrangePartition(offsets, extents)
```

The last type of partitioning is points, where we have to specify, in a list, the global indices of the points stored by the process. (Another example with only one segment per process is available in example `5-points`.)

```
global_indices=[0, 1, 2, 3]
partition = pyoasis.PointsPartition(global_indices)
```

2.4 Defining the coupling grids

The grid data files, containing the definition of the grids onto which the coupling data is defined, can be created by the user before the run or can be written directly at run time by the components, either by one component process to write the whole grid or by each process holding a part of a grid. Details about the grid definition can be found in section 2.2.4 of OASIS3-MCT User Guide. A full example of writing a grid in sequential and parallel models can be found in `examples/10-grid`.

To initialise a grid and write the grid longitudes and latitudes, one has to create an instance of the **Grid** class:

```
[...]
lon = np.array([-180. + comm_rank*nx_loc*dx + float(i)*dx +
             dx/2.0 for i in range(nx_loc)], dtype=np.float64)
lon = np.tile(lon, (ny_loc, 1)).T

lat = np.array([-90.0 + float(j)*dy + dy/2.0 for j in range(ny_loc)],
             dtype=np.float64)
lat = np.tile(lat, (nx_loc, 1))
grid = pyoasis.Grid('pyoa', nx_global, ny_global, lon, lat, partition)
```

To write the grid cell corner longitudes and latitudes, the `set_corners` method can be used

```
[...]
ncrn = 4
clo = pyoasis.asarray(np.zeros((nx_loc, ny_loc, ncrn), dtype=np.float64))
clo[:, :, 0] = lon[:, :] - dx/2.0
clo[:, :, 1] = lon[:, :] + dx/2.0
clo[:, :, 2] = clo[:, :, 1]
clo[:, :, 3] = clo[:, :, 0]
cla = pyoasis.asarray(np.zeros((nx_loc, ny_loc, ncrn), dtype=np.float64))
cla[:, :, 0] = lat[:, :] - dy/2.0
cla[:, :, 1] = cla[:, :, 0]
cla[:, :, 2] = lat[:, :] + dy/2.0
cla[:, :, 3] = cla[:, :, 2]

grid.set_corners(clo, cla)
```

To write the grid cell areas, the `set_area` method can be used :

```
[...]
area = np.zeros((nx_loc, ny_loc), dtype=np.float64)
area[:, :] = dp_conv * \
           np.abs(np.sin(cla[:, :, 2] * dp_conv) -
```

(continues on next page)

(continued from previous page)

```
                    np.sin(cla[:, :, 0] * dp_conv)) * \
            np.abs(clo[:, :, 1] - clo[:, :, 0])

grid.set_area(area)
```

To define the mask of the grid, the `set_mask` method can be used (here a mask where all points have zero value i.e. are valid). Notice the optional argument `companion` providing the name of the corresponding ocean grid from which the masks and fractions are obtained:

```
msk = np.zeros((nx_loc, ny_loc), dtype=np.int32)
grid.set_mask(msk, companion=None)
```

To define the grid cell water fraction, the `set_frac` method can be used:

```
frc = np.ones((nx_loc, ny_loc), dtype=np.float64)
frc = np.where(msk == 1, 0.0, 1.0)
grid.set_frac(frc, companion=None)
```

To define the grid cell angles, the `set_angle` method can be used:

```
angle = np.zeros((nx_loc, ny_loc), dtype=np.float64)
grid.set_angle(angle)
```

2.5 Declaring the coupling data

The coupling data is handled by the class **Var**. Its constructor requires its symbolic name, as it appears in the `namcouple` file, the partition and a flag indicating whether the data is incoming or outgoing. The latter is an enumerated type and can have the values `pyoasis.OasisParameters.OASIS_OUT` or `pyoasis.OasisParameters.OASIS_IN`. In the following example, we wish to send data to a process having the rank 1 and we use a partition that was previously created.:

```
data_name = "name"
variable = pyoasis.Var(data_name, partition,
                       pyoasis.OasisParameters.OASIS_OUT)
```

In the case of the receiving model, the code is:

```
data_name = "name"
variable = pyoasis.Var(data_name, partition,
                       pyoasis.OasisParameters.OASIS_IN)
```

The property `is_active` can be tested to check if the variable is activated in the `namcouple` configuring file:

```
variable2 = pyoasis.Var("NOTANAME", partition, OASIS.OUT)
if variable2.is_active:
     print("{} is active".format(variable2))
 else:
     print("{} is not active".format(variable2))
```

The coupling period(s) of the data, as defined in the `namcouple`, can be accessed with the property `cpl_freqs` and the number of coupling exchanges in which the data is involved by `len(cpl_freqs)`:

```
var_1 = pyoasis.Var("FRECVATM_1", partition, OASIS.IN)
print("Recv_one: coupling frequencies for {} = ".format(var_1.name),
var_1.cpl_freqs)
```

The method `put_inquire` of the variable tells what would happen to the corresponding data at that date below the corresponding send action. This maybe useful if, for example, the calculation of a coupling field is costly and if one wants to compute it only when it is really sent out. The different possible return codes are listed in section 2.2.9 of OASIS3-MCT User Guide.

```
for date in range(43200):
    if var_1.put_inquire(date) == OASIS.SENT:
        var_1.put(date, pyoasis.asarray([date], dtype=np.float64))
[...]
```

2.6 Ending the definition phase

We must end the definition of the component by calling the `enddef()` method.

```
comp.enddef()
```

However this must be done only once the partitioning and the variable data have been initalised.

2.7 Sending and receiving data

pyOASIS expects data to be provided as a **pyoasis.asarray** object:

```
field = pyoasis.asarray(range(n_points))
```

This is a Numpy array but ordered in the Fortran way. In C, multidimensional arrays store data in row-major order where contiguous elements are accessed by incrementing the rightmost index while varying the other indices will correspond to increasing strides in memory as we use indices further towards the left. By default, Numpy arrays use that ordering as well. Fortran, on the other hand, uses column-major order. In that case, contiguous elements are accessed by incrementing the leftmost index. **pyoasis.asarray** objects use the same ordering as Fortran. As a consequence, it is not necessary to transform data in order to use it in the OASIS3-MCT Fortran library.

The sending and receiving actions may be called by the component at each timestep. The date argument is automatically analysed and actions are actually performed only if date corresponds to a time for which it should be activated, given the period indicated by the user in the namcouple. See OASIS3-MCT User Guide section 2.2.7 for details.

The data is sent with the following function.

```
date = int(0)
variable.put(date, field)
```

Conversely, it is received with the function

```
variable.get(date, field)
```

This will fill the **pyoasis.asarray** object.

2.8 Termination

Finally, the coupling is terminated in the destructor of the component:

```
del comp
```

2.9 Exceptions and aborting

When an error occurs in OASIS3-MCT, the code coupler returns an error code and an **OasisException** is raised. In practice, OASIS3-MCT will internally handle the error, write an error message in its debug log files and to the screen, and abort before the exception is raised. It may also happen that the code aborts before the error message appears on the screen.

When an error is caught by the pyOASIS wrapper, such as an incorrect parameter or a wrong argument type, a **PyOasisException** is raised.

In the following example, where we attempt to initialise a component, a **PyOasisException** will be raised as the user supplies an empty name :

```
try:
    comp = pyoasis.Component("")
except (pyoasis.PyOasisException) as exception:
    pyoasis.pyoasis_abort(exception)
```

The function `pyoasis.pyoasis_abort` takes an exception as argument. It stops the execution of all the processes after having displayed an error message and written information in the log files about the error and the context in which it took place.

Another function is available, `pyoasis.oasis_abort`, for the cases where a voluntary abort is needed in the code where or not an exception has been raised. Its interface mimics the corresponding OASIS3-MCT functio `oasis_abort`.

CHAPTER

THREE

EXAMPLES

3.1 Serial partitions

The example in examples/1-serial/python, containing the full source code as well as the namcouple file and a script to run this example, consists in two models, one sending data to another. The sender and receiver start in the same way.

-Import pyOASIS and initialise the MPI communicator

```
import pyoasis
from mpi4py import MPI
comm = MPI.COMM_WORLD
```

-Initialisation of the component

```
comp = pyoasis.Component(component_name)
```

-Initialisation of the serial partition

```
partition = pyoasis.SerialPartition(n_points)
```

-From this point, the sender and the receiver start to differ. In the sender, the variable data is initialised by

```
variable = pyoasis.Var("FSENDOCN", partition,
                       pyoasis.OasisParameters.OASIS_OUT)
```

whereas, in the receiver, we have

```
variable = pyoasis.Var("FRECVATM", partition,
                       pyoasis.OasisParameters.OASIS_IN)
```

where the last flag is instead `pyoasis.OasisParameters.OASIS_IN` to indicate that, in this case, the data will be incoming.

In both scripts, the initialisation of the component ends by

```
comp.enddef()
```

In the sender, the data is subsequently transmitted by

```
time_in_the_model = int(0)
field = pyoasis.asarray(range(n_points))
variable.put(date, field)
```

while, in the receiver, it is recovered by

```
time_in_the_model = int(0)
field = pyoasis.asarray(numpy.zeros(n_points))
variable.get(date, field)
```

3.2 Apple and orange partitions

In the example in `examples/6-apple_and_orange/python`, both models run as several processes. The beginning of the code is identical to the previous example. We will highlight only the differences. In the sender, the data is split according to the apple partitioning

```
partition = pyoasis.ApplePartition(offset, local_size)
```

whereas the receiver uses the orange partitioning.

```
partition = pyoasis.OrangePartition(offsets, extents)
```

In both cases, the offsets and sizes of the local part of the data have to be specified. Each process subsequently transmits and receives its share of the data as previously. In the sender, we have

```
date = int(0)
field = pyoasis.asarray(numpy.zeros(local_size))
for i in range(local_size):
    field[i] = offset + i
variable.put(date, field)
```

while, in the receiver,

```
date = int(0)
field = pyoasis.asarray(numpy.zeros(extent))
variable.get(date, field)
```

3.3 Fortran and Python interoperability

In order to illustrate the possibility to couple models written in Python and in Fortran, the previous example is repeated in pyoasis/examples/8-interoperability/fortran_and_python but this time, the sender has been written in Fortran. The receiver is the same as above.

The sender consists in an analogous sequence using the OASIS3-MCT Fortran API

-Initialisation of the component

```
call oasis_init_comp(comp_id, comp_name, kinfo)
```

-Initialisation of the apple partition with the relevant offset and local size

```
part_params=[1, offset, local_size]
call oasis_def_partition(part_id, part_params, kinfo)
```

-Creation of the variable data, a bundle with two fields

```
var_nodims=[1, 2]
call oasis_def_var(var_id, var_name, part_id, var_nodims, OASIS_OUT,
                   [1], OASIS_REAL, kinfo)
```

-End of the definition of the component

```
call oasis_enddef(kinfo)
```

-Transmission of the local part of the data to the other component

```
call oasis_put(var_id, date, bundle, kinfo)
```

-End of the coupling

```
call oasis_terminate(kinfo)
```

CHAPTER

FOUR

API REFERENCE

CHAPTER

FIVE

CORRESPONDENCE WITH THE OASIS3-MCT FORTRAN API

These tables show the correspondence between the functions described in the OASIS3-MCT user guide and their analogue in pyoasis. All the functions have been implemented with their full interface except `put` which, in the case of pyoasis, accepts only a single field.

5.1 Component

OASIS3-MCT	pyoasis.Component.
oasis_init_comp(comp_id, comp_name, ierror, coupled, comm_world)	**__init__(name, coupled=True,** communicator=None)
oasis_terminate(ierror)	__del__()
oasis_create_couplcomm(icpl, localcomm, couplcomm, kinfo)	create_couplcomm(coupled)
oasis_set_couplcomm(couplcomm, kinfo)	set_couplcomm(couplcomm)
oasis_get_multi_intracomm(newcomm, cdnam, root_ranks, kinfo)	get_multi_intracomm(complist)
oasis_get_intracomm(newcomm, cdnam, kinfo)	get_intracomm(compname)
oasis_get_intercomm(newcomm, cdnam, kinfo)	get_intercomm(compname)
oasis_enddef(ierror)	enddef()
oasis_get_local_comm(local_comm, ierror)	localcomm
oasis_get_debug(debugvalue)	debug_level
oasis_set_debug(debugvalue)	debug_level

5.2 Partition

OASIS3-MCT	pyoasis.
oasis_def_partition(ilpart_id, igparal, ierror, isize, name)	**SerialPartition(size,** global_size=-1, name="") **ApplePartition(offset, size,** global_size=-1, name="") **BoxPartition(global_offset,** local_extent_x, local_extent_y, global_extent_x, global_size=-1, name="") **OrangePartition(offsets, extents,** global_size=-1, name="") **PointsPartition(global_indices,** global_size=-1, name="")

5.3 Var

OASIS3-MCT	pyoasis.var.
oasis_def_var(var_id, name, il_part_id, var_nodims, kinout, var_actual_shape, vartype, ierror)	**__init__(name, partition, inout,** bundle_size=1)
oasis_put(varid, date, fld1, info, fld2, fld3, fld4, fld5, write_restart)	**put(date, field,** write_restart=False)
oasis_get(varid, date, fld, info)	get(date, field)
oasis_put_inquire(varid, date, kinfo)	put_inquire(date)
oasis_get_ncpl(varid, ncpl, kinfo)	len(cpl_freqs)
oasis_get_freqs(varid, mop, ncpl, cplfreqs, kinfo)	cpl_freqs

5.4 Grid

OASIS3-MCT	pyoasis.grid.
oasis_start_grids_writing(flag) oasis_write_grid(cgrid, nx_global, ny_global, lon, lat, il_partid)	**__init__(cgrid, nx_global,** ny_global, lon, lat, partition=None)
oasis_write_corner(cgrid, nx_global, ny_global, nc, clon, clat, il_partid)	set_corners(clo, cla)
oasis_write_area(cgrid, nx_global, ny_global, area, il_partid)	set_area(area)
oasis_write_mask(cgrid, nx_glo, ny_glo, mask, part_id, companion)	**set_mask(mask,** companion=None)
oasis_write_frac(cgrid, nx_glo, ny_glo, frac, part_id, companion)	**set_frac(frac,** companion=None)
oasis_write_angle(cgrid, nx_global, ny_global, angle, il_partid)	set_angle(angle)
oasis_terminate_grids_writing()	write()

5.5 Utilities

OASIS3-MCT	pyoasis.
oasis_abort(compid, routinename, abortmessage, rcode)	**oasis_abort(component_id,** routine, message, filename, line, error) pyoasis_abort(exception)

CHAPTER

SIX

INSTALLATION

6.1 Under GNU/Linux

6.1.1 Prerequisites

- A Fortran and C compiler suite
- An MPI library
- NetCDF 4
- Python 3 with mpi4py, numpy and netCDF4
- **Extra optional packages for plotting (as in examples `11-test-interpolation` and `12-grid-functions`)**
  - matplotlib
  - GEOS (Geometry Engine, Open Source): package libgeos-dev under Debian or Ubuntu, geos-devel under Fedora
  - proj: package libproj-dev under Debian or Ubuntu, proj-devel under Fedora
- **Optional packages for generating the documentation:**
  - sphinx
  - Tex Live
- **Optional package for automated testing:**
  - pytest

6.1.2 OASIS3-MCT Installation

- Obtain OASIS3-MCT (refer to OASIS3-MCT User Guide for details).
- Change directory to ${OASIS_ROOT}/util/make_dir.
- Create your own make.inc file based on make.intel_davinci, make.gfortran_openmpi_linux or make.bindings.
- Build and install OASIS3-MCT and pyOASIS:

```
make -f TopMakefile realclean
make -f TopMakefile pyoasis
```

- Append the lines displayed at the end of the compilation to your .bashrc file or, alternatively, before using pyOASIS, source the script mentioned there.

6.1.3 Virtual Python environment

- Create a virtual environment (set `VENVDIR` as a directory of your choice containing the virtual environment):

```
export VENVDIR=~/INSTALL/PY_ENV/PyO
python3 -m venv ${VENVDIR}
source ${VENVDIR}/bin/activate
```

- Install packages:

```
pip install --upgrade pip
pip install mpi4py
pip install numpy
pip install netcdf4
```

6.1.4 Extra software

- For applications using Cartopy plots, as in the examples `11-test-interpolation` and `12-grid-functions`:

```
pip install scipy
pip install matplotlib
pip uninstall shapely
pip install shapely --no-binary shapely
pip install cartopy
```

- Optional package for performances optimisation:

```
pip install pykdtree
```

6.2 Under macOS

6.2.1 Prerequisites

Refer to the GNU/Linux section.

If using Brew (tested with gnu up to version 10.2.0):

```
brew install gcc
brew install openblas
brew install openmpi
```

For the optional packages:

```
brew install geos
brew install proj
```

6.2.2 OASIS3-MCT Installation

Same as under GNU/Linux. See previous section.

6.2.3 Virtual Python environment

- Create a virtual environment:

  Same as under GNU/Linux, see previous section.

- Install packages:

```
pip install mpi4py
pip uninstall numpy
pip cache remove numpy
OPENBLAS="$(brew --prefix openblas)" pip install --global-option=build-ext numpy
pip install netcdf4
```

6.2.4 Extra software

- For applications using Cartopy plots, as in the examples `11-test-interpolation` and `12-grid-functions`:

```
pip uninstall scipy
pip cache remove scipy
pip install --global-option=build-ext scipy
pip uninstall shapely
pip install shapely --no-binary shapely
pip install matplotlib
pip install cartopy
```

- Optional package for performances optimisation:

```
pip install pykdtree
```

6.2.5 Set up the environment

At the end of your .bashrc,

```
export TMPDIR=/tmp
export OMPI_MCA_mca_base_env_list=LD_LIBRARY_PATH=${PYOASIS_ROOT}/lib
```

6.3 Documentation

If pyOASIS is modified, this document can be regenerated, using Sphinx, by typing the following command in the directory ${OASIS_ROOT}/pyoasis:

```
make doc
```

CHAPTER

SEVEN

ACKNOWLEDGMENTS

This work has been financed by the IS-ENES3 project which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 824084.

CHAPTER

EIGHT

INDEX AND SEARCH

- genindex
- search