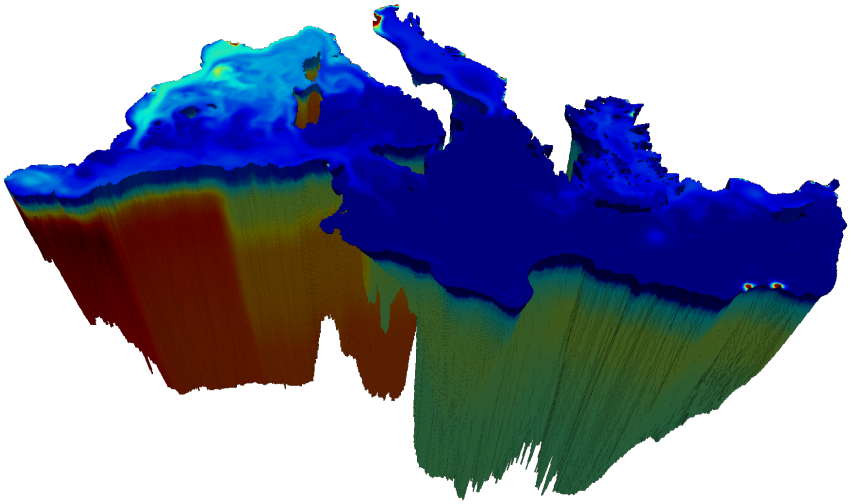
Master in High Performance Computing

High performance data analysis and visualization tools for the MedBFM physical-biogeochemical model



Supervisors:
Stefano Salon,
Giampiero Cossarini,
Giorgio Bolzon,
Paolo Lazzari

Candidate:
Cosimo Livi

3rd edition
2016–2017

Contents

1 Introduction

The MedBFM model system is developed and managed by OGS and operationally produces analysis, forecast and reanalysis of biogeochemical 3D fields for the Mediterranean Sea in the framework of the Mediterranean monitoring and forecasting center of the EU Copernicus Marine Environment Monitoring Services (CMEMS). The present post-processing scheme is based on an off-line suite of python scripts (aka "bit.sea") which is used to monitor the model output, to prepare the quality information documents targeted towards the users, and for scientific research. However, the inner complexity of the multivariate 4D data products (i.e., approximately 50 variables organized in 3D gridded fields evolving in time), the increase of the number of products (operational and derived), validation metrics, and users number, all combined with the continuos refining of spatial resolution, pose a series of challenges for the efficient management of the whole data stream analysis workflow and its performance.
Indeed, the usual approach to data analysis can easily become too complex for the generic user: the need to exploit a cluster for the analysis of large amount of data poses strong limits on the practical usability of standard analysis routines, as can be seen from skatch in Fig.1.1. The alternative approach proposed in this thesis work aims to develop an efficient and scalable tool that can directly access model's output (thus skipping the postprocessing phase), obtaining on-the-fly and on-demand results, while keeping a flexible and dynamic structure that also provides an intuitive graphical user interface (GUI), granting an easy access to the users.
This service may be able to run on a dedicated server for remote visualization, offering the possibility to interactively inquire datasets to a large number of users.
The natural environment for this kind of application is Paraview, since it is an open-source software with the capabilities to visualize and analyze large amount of data, both using interactive or batch/scripting methods.

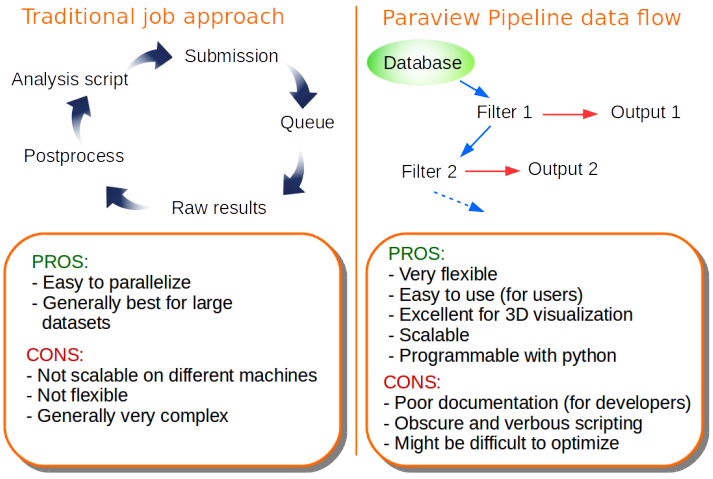


Figure 1.1: sketch of the comparison between the traditional approach to data analysis with the new scheme proposed in this work.

Indeed, Paraview is built on excellent graphic libraries (vtk) and also offers the possibility to customize its features and filters through low-level python and C++ programming. Moreover, the typical pipeline structure offered by Paraview seems to perfectly satisfy the aforementioned needs of flexibility and dynamicity of the analysis tool we would like to produce.
One of the most important step for the new approach we had in mind was to find the best way for bridging the gap between the native data structures used by OGS and the vtk's requirements and formats for a proper visualization in Paraview. In order to facilitate this process, we developed a python module called `OGSParaviewSuite`, which is able to provide all the methods required for the conversion and the manipulation of data, as well as some high performance implementation for their analysis. as shown in Fig.1.2. The `OGSParaviewSuite` module essentially collects some features from the bit.sea, vtk and ctypes python modules and wraps them into simple classes and routines.

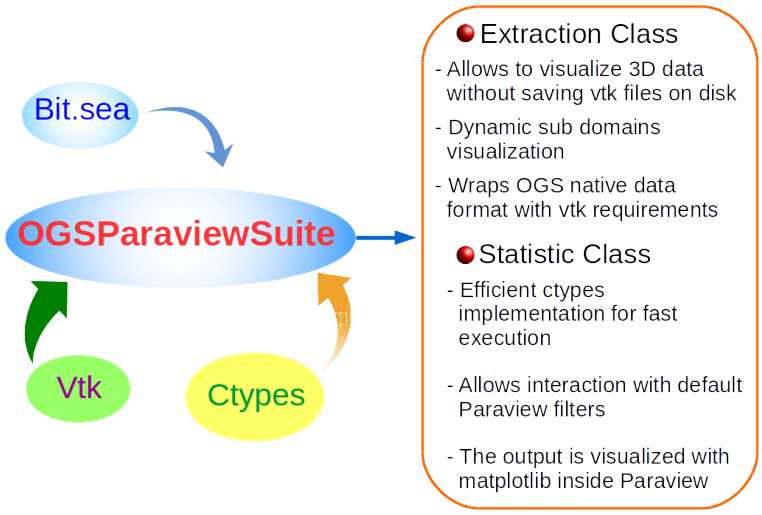


Figure 1.2: sketches of the classes provided by the `OGSParaviewSuite` module.

This tool has been the fundamental building block for the source scripts of custom Paraview's filters, greatly facilitating the, anyway complicated, developing process. A Paraview filter can be defined in general as a function that receives an input and produces an output; when several filters are chained togheter, then it is called a pipeline.
Therefore, one of the main scope of the project was to produce (with python) all the needed filters in order to dynamically fetch data from model's raw output, visualize and manipulate it, all without saving any extra file on disk.
This last aspect is of foundamental importance, since a long climatological simulation can easily produce 1TB of data in NetCDF format and the production of auxiliary files for visualization would be catastrophic in term of disk space and performances. Once

all the custom filters are produced, the last step of the work is to set-up a remote server with several services running on it. The server will be able to provide to the users a full access to the database and to the new developed tools, and this is accomplished exploiting the servers services offered by Paraview.
The thesis is organized as follows:

- Section 2 provides a detailed list of instructions for installing Paraview from source by using the provided installation scripts and for linking it with the tools developed in this work, on a machine mounting Ubuntu Linux distribution.
- Section 3 presents an overview of the developing process, starting from an explanation of the repository structure. The features introduced by the `OGSParaviewSuite` python module, as well as their benchmark, are then explained through some examples, with a short guide on how to use them for Paraview's filter programming. In the last part of this section the usage and capabilities of the developed filters are shown.
- Section 4 collects a full documentation of `OGSParaviewSuite` classes and methods, as well as the the documentation of the custom filters developed in this work. It can be considered as a stand-alone section that can be consulted without having read the whole thesis.

Note about text coloring and style: in the following of the work the `texttt` font for everything that is related to the unix's bash envirionment, while for quoting python code the python highlight will be used, so that the classes of a module will appear in blue, as `OGSvtk`, while methods are highlighted in red, as for example `GetVariable`.

2 Setup

In this section the procedure for a proper installation of Paraview and the OGSParaviewSuite will be presented. At first the required steps for installing on the machine all the required software and libraries are presented in subsection 2.1, then subsection 2.2 hosts a detailed guide related to the installation of Paraview through the provided installation scripts, as well as the server setup. In subsection 2.3 the procedure related to a correct installation of the `OGSParaviewSuite` python model, developed in this thesis work, is explained.

2.1 Preparing the machine

The following guide is for a machine mounting Ubuntu 16.04, but the same procedure can be applied for CentOS7, by taking into account that packages in this case will have different names.
At first, install the basic software:

```
sudo apt-get install git python-dev python-pip libopenmpi-dev cmake
pip install numpy matplotlib mpi4py
```

Paraview is based on qt libraries and recents builds use the latest qt5 version. Backward compatibility with qt4 is anyway offered, which are usually easier to install. In order to have Paraview working with qt4 libraries the following packages needs to be installed:

```
sudo apt-get install qt4-dev-tools libqt4-core
```

If qt5 libraries are strictly needed one can choose between install qt5 from source or by adding a new repository:

```
sudo add-apt-repository ppa:beineri/opt-qt593-xenial
sudo apt-get update
sudo apt-get install qt59-meta-full
source /opt/qt59/bin/qt59-env.sh
```

with this option the libraries will be installed in `/opt/` folder and it may be more difficult to properly link them to Paraview. At this point clone the repository:

```
git clone https://gitlab.com/clivi/OGS_Paraview_module.git
```

For now let's focus only on the Paraview's installation scripts inside the `tools` folder, which brings us directly into the next part of the guide.

2.2 Installing Paraview

Two different ways of installing Paraview will be presented in this section: one will use proprietary OpenGL drivers for GPU rendering, while the other allows to render only by using CPUs thanks to the open source OSMesa drivers. The former is indicated when the machine in which Paraview is installed has access to a GPU and to a physical

or virtual screen, while the latter is useful in the cases of headless servers (i.e. servers without a display), or in general if GPUs are not available.
In this second case, OSMesa drivers will be installed for simulating OpenGL on CPUs and no display window will be opened by Paraview, so that it can be used only in a client-server configuration.

2.2.1 Using prorietary OpenGL

This is the simplest case as does not requires any additional software. The script `install_paraview.sh` will download, unpack, build ad install the desired configuration of Paraview using cmake.
Inside the script there are several important shell variables, like `VERS`, that must be an existing version of Paraview, and `INSTALL_PREFIX`, that defines the installation path.
Besides them, the most important section of the script is relative to the cmake configuration, below an example:

```
# Configure CMAKE
cmake ../ \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_INSTALL_PREFIX=$INSTALL_PREFIX \
  -DBUILD_SHARED_LIBS=ON \
  -DBUILD_TESTING=OFF \
  -DVTK_RENDERING_BACKEND=OpenGL \
  -DVTK_SMP_IMPLEMENTATION_TYPE=OpenMP \
  -DVTK_USE_X=ON \
  -DPARAVIEW_QT_VERSION=4 \
  -DPARAVIEW_BUILD_QT_GUI=ON \
  -DPARAVIEW_ENABLE_CATALYST=ON \
  -DPARAVIEW_BUILD_CATALYST_ADAPTORS=ON \
  -DPARAVIEW_ENABLE_PYTHON=ON \
  -DPYTHON_INCLUDE_DIR="/usr/include/python2.7" \
  -DPYTHON_LIBRARY="/usr/lib/x86_64-linux-gnu/libpython2.7.so" \
  -DPYTHON_EXECUTABLE="/usr/bin/python" \
  -DPARAVIEW_USE_MPI=ON \
  -DMPI_C_INCLUDE_PATH="/usr/include" \
  -DMPI_CXX_INCLUDE_PATH="/usr/include" \
  -DMPI_C_COMPILER="/usr/bin/mpicc" \
  -DMPI_CXX_COMPILER="/usr/bin/mpicxx" \
  -DMPIEXEC="/usr/bin/mpiexec" \
  -DPARAVIEW_INSTALL_DEVELOPMENT_FILES=ON \
  -DCMAKE_C_COMPILER=mpicc \
  -DCMAKE_CXX_COMPILER=mpicxx
```

The variable `PARAVIEW_QT_VERSION` can be set to 4 or 5, depending on the desired Qt version (usually 4 is more stable but deprecated). Also the variable `VTK_RENDERING_BACKEND` can be set to `OpenGL` or `OpenGl2`, depending on the available hardware and drivers (check the GPU's capabilities).
Note: it is very important that the full paths to python objects, MPI libraries and binaries are correct, remember to check them since they are usually different depending on the OS and architectures.

2.2.2 Using OSMesa

If Paraview is being installed on a server that neither has access to a display (physical or virtual) nor to a GPU, then OSMesa drivers must be used. In order to do so, at first `llvm` and then `mesa` must be installed (better from source), and this can be accomplished using the provided installation scripts.
Once `mesa` drivers are installed, just run the script `install_paraview_osmesa.sh`, which is essentially the same as the one presented in subsection 2.2.1, but now `osmesa` configuration variables appear in the cmake configuration:

```
# Configure CMAKE
cmake ../ \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_INSTALL_PREFIX=$INSTALL_PREFIX \
  -DBUILD_SHARED_LIBS=ON \
  -DBUILD_TESTING=OFF \
  -DVTK_RENDERING_BACKEND=OpenGL \
  -DVTK_SMP_IMPLEMENTATION_TYPE=OpenMP \
  -DVTK_USE_X=OFF \
  -DVTK_OPENGL_HAS_OSMESA=ON \
  -DVTK_USE_OFFSCREEN=OFF \
  -DOPENGL_INCLUDE_DIR=IGNORE \
  -DOPENGL_xmesa_INCLUDE_DIR=IGNORE \
  -DOPENGL_gl_LIBRARY=IGNORE \
  -DOSMESA_INCLUDE_DIR=${MESA_DIR}/include \
  -DOSMESA_LIBRARY=${MESA_DIR}/lib/libOSMesa.so \
  -DPARAVIEW_QT_VERSION=4 \
  -DPARAVIEW_BUILD_QT_GUI=OFF \
  -DPARAVIEW_ENABLE_CATALYST=ON \
  -DPARAVIEW_BUILD_CATALYST_ADAPTORS=ON \
  -DPARAVIEW_ENABLE_PYTHON=ON \
  -DPYTHON_INCLUDE_DIR=$PYTHON_DIR/include/python${PYTHON_VERS%.*} \
  -DPYTHON_LIBRARY=$PYTHON_DIR/lib/libpython${PYTHON_VERS%.*}.so \
  -DPYTHON_EXECUTABLE=$PYTHON_DIR/bin/python \
  -DPARAVIEW_USE_MPI=ON \
```

```
-DMPI_C_INCLUDE_PATH="$MPI_DIR/include" \
-DMPI_CXX_INCLUDE_PATH="$MPI_DIR/include" \
-DMPI_C_COMPILER="$MPI_DIR/bin/mpicc" \
-DMPI_CXX_COMPILER="$MPI_DIR/bin/mpicxx" \
-DMPIEXEC="$MPI_DIR/bin/mpiexec" \
-DPARAVIEW_INSTALL_DEVELOPMENT_FILES=ON \
-DCMAKE_C_COMPILER=mpicc \
-DCMAKE_CXX_COMPILER=mpicxx
```

So, now several new variables related to the `osmesa` build are present. It is important, as before, that all the full paths to libraries and binaries are correct.

2.2.3 Define the Environment

After a successfull installation of Paraview, a last step is required: define the environment. If just one version of Paraview is installed, the simplest way to proceed is to modify the `bashrc` file in the user's home; if one needs, indeed, to switch between different versions of the software, the best approach is to prepare some environment scripts that every time export the needed paths only in that shell.
What is required is to tell the system where to look for vtk libraries, python packages and binaries installed by Paraview, and this can be done by modifying three environment variables: `LD_LIBRARY_PATH, PATH` and `PYTHONPATH`, in the following way (assuming Paraview is installed into `/opt/5.4.0`):

```
PV_DIR="/opt/5.4.0"
export LD_LIBRARY_PATH=$PV_DIR/lib:$PV_DIR/lib/paraview-5.4/
       site-packages/vtk:$PV_DIR/lib/paraview-5.4:$PV_DIR/lib/
       paraview-5.4/site-pacakges/paraview/vtk:$LD_LIBRARY_PATH
export PYTHONPATH=$PYTHONPATH:$PV_DIR/lib/paraview-5.4/
       site-packages:$PV_DIR/lib/paraview-5.4/site-packages/vtk
export PATH=$PV_DIR/bin:$PATH
```

Now vtk libraries and python modules will be available also for the system, so that vtk can be used inside C codes and python scripts without being forced to use pvpython.

2.2.4 Set up a Paraview Server

Once Paraview is successfully installed on both the server and the client machines, it is possible to work in client-server mode. This configuration has several advantages: at first, all the data can be kept only in the server machine, as well as the custom python modules and plugins. Moreover, since the server can be launched in parallel with `mpirun`, it allows to exploit a parallel implementation of a filter, if it has one. One last positive aspect is the possibility to split computation from rendering, so that if the server is headless or does not have a GPU, the rendering can be made by the client machine,

while the computation are still executed on the server side.
In order to establish a connection, these steps must be followed:

- Launch pvserver on the remote server. Once logged with ssh launch:

```
mpirun -np 12 pvserver --server-port=11111
```

- Open an ssh tunnel in the client machine:

```
ssh -L 11111:<hostname>:11111 -N user@hostname
```

- Launch Paraview on the client, click "Connect", then click "Add server", select a name, press "configure", choose start up as "Manual" and save.

At the end of this procedure, the client connects to the remote server and all the installed plugins will be available.

2.3 Installing OGSParaviewSuite

The installation of `OGSParaviewSuite` python module and the paraview custom filters is really straightforward:

- if it hadn't been already done, clone the repository:

```
git clone https://gitlab.com/clivi/OGS_Paraview_module.git
```

- add the folder `bit.sea` to the `PYTHONPATH`:

```
export PYTHONPATH=/path_to_bit.sea/bit.sea:$PYTHONPATH
```

  then enter into `CLIMATOLOGY` folder and lunch the script `update_clim_folder.sh`.

- copy all the `.xml` files from the `PW_filters` folder into: `paraview_install_dir/lib/paraview-5.4/plugins` (if the `plugins` folder is not present, create it).

- compile C++ library needed by `OGSParaviewSuite`: got to the folder `clib/src` inside the repository, edit the Makefile in order to set the correct include path of `CFLAG` variable to vtk's `include` directory (usually is `paraview_installation_dir/5.4.0/include/paraview-5.4`) and type `make`.

Now Paraview is linked with OGS's python modules and custom filters.

3 Implementation and Features

This section is intended as a full overview of the developing process, with step by step examples for introducing the reader to the new tools produced during this thesis work. The repository's structure is presented in subsection 3.1, with a detailed list and descriptions of the files contained in it. In subsection 3.2 the basic features offered by the implemented `OGSParaviewSuite` python module are explained and benchmarked, while subsection 3.3 will present how to exploit the module for developing Paraview's plugins. In the last part, embodied by subsection 3.4, the custom plugins developed in this work are explained and analyzed.

3.1 The Repository

In this subsection the content of the git repository that stores all the work will be explained in detail, as well as the data structure of the MEDBFM model.
The MEDBFM output comes in NetCDF4 files and three different resolution can be selected:

- Low: each variable's file occupies about 11 MB.
- Mid: each variable's file occupies about 50 MB
- High: each variable's file occupies about 200 MB

for each resolution, a file called `meshmask.nc` stores the information about the domain subdivision into the different bathymetric levels (coast, open sea and everywhere) and different sub-basins (which can be intended as the different regions of the Mediterrean sea). Moreover, the model's grid uses Latitude and Longitude for $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ directions, so that a proper projection to a plane and, thus, a conversion to meters must be performed in order to obtain a rectilinear grid.
The connection among the aforementioned structure with the vtk requirements for visualization inside Paraview is handled by the `OGSParaviewSuite` python module, which relies on the files contained in the repository.
The repository contains the following folders:

`MESHMASK:` all the files required for domain subdivision and conversion to meters are stored into the `MESHMASK` folder, where three subfolders (one for each resolution) can be found. Each of them contains the files required by the `OGSParaviewSuite`'s routines for handling the grid creation and domains subdivision. The latter is obtained by adding two mask variables to the grid, one for the coast and the other for the sub-basins, that will be used for dynamically select the desired region of the sea. Two scripts are also provided in order to re-obtain all these required files starting only by the `meshmask.nc` file: `LonLattoMeters.py` and `mask_extractor.py`. Both can be executed using the following syntax:

```
python LonLattoMeters.py -i$PWD -r "desired resolution"
```

`CLIMATOLOGY`: the folder contains the files required for a dynamic evaluation of the climatology, and a bash script `update_clim_folder.sh` used for update a file inside the `bit.sea` folder with the information required to find the aforementioned climatology files. The script is supposed to being launched inside the `CLIMATOLOGY` folder immediately after the download.

`bit.sea`: all the custom python modules, like `OGSParaviewSuite`, are stored in this folder, thus it is mandatory to add it to the `PYTHONPATH` (see Sec.2.3).

`Documentation`: the folder that hosts the present thesis as documentation, as well as the source `tex` file, and a simple presentation about the project.

`PW_filters`: inside this folder all the source python scripts used for the custom Paraview's plugins are stored, as well as the actual plugins (the xml files) and a python script for the conversion from python to xml.

`clib`: hosts the source code and the Makefile for compiling the C++ library needed by the `OGSParaviewSuite` module. In order to compile it, just change the include path of the `CFLAGS` variable so that it points to the correct path of vtk's include folder (see Sec.2.3).

`macros`: a macro for plotting with matplotlib inside Paraview are present. This macro is a python script that can be loaded directly inside Paraview using the "Load macros" function.

`tools`: the folder that contains the installation scripts for Paraview and OSMesa. Their usage is explained in Sec.2.2.

3.2 OGSparaviewSuite: applications and benchmarks

The `OGSParaviewSuite` python module has been developed with the aim to connect the OGS' data structure, including the resolutions and the domain subdivision, with vtk libraries, in order to obtain an easy-to-use interface for producing satisfying 3D visualization and statistic analysis. Indeed, the final purpose was to be able to properly exploit the dynamic nature of a Paraview pipeline, giving to the users the possibility to quickly select the desired region of the sea to visualize and, eventually, being able to perform on it a complete and on-the-fly statistic analysis, all with very simple operations through the Paraview's interface.
This section will mainly focus on the attempt to give to the reader the key aspects of this module and introduce him to the basic usage of the different classes and methods through some examples; for an extensive description of all the functions provided, refer to the documentation on Sec.4.
Three classes are defined inside the module:

- OGSvtk: provides the routines needed for on-the-fly conversion of model's output NetCDF files in a vtk visualizable object.

- LoadNetCDF: allows to access the variables stored in NetCDF files and convert them into numpy arrays
- OGSStatistic: provides the routines necessary to perform statistic operations on the desired sea region. It exploits a custom C++ library and wraps it to the python code using Ctypes.

3.2.1 OGSvtk

This class essentially offers the instruments for an on-the-fly conversion from raw model's output to a visualizable vtk object, which can be visualized without saving any additional file on disk. This is a mandatory feature, since a long simulation output can easily reach the TeraByte threshold, so that it would be really problematic if auxiliary vtk files would be required for visualization.
Another important feature this class introduces is the possibility to add a mask for coasts, sub-basins and depth subdivisions, so that any desired region can be selected for insvestigation.
Let's see in detail how this class works, starting from an example:

```
# Load module
from OGSParaviewSuite import *
# Initialize class
OGS = OGSvtk('/home/user/MESHMASK')
OGS.SetResolution('mid')
# Create vtkRectilinearGrid
rg = OGS.CreateRectilinearGrid()
OGS.AddMask(rg)
# Load data from NetCDF
data = OGS.LoadNetCDF.GetVariable('input_file','N3n')
# Add data to vtkRectilinearGrid
rg.GetCellData().AddArray(OGS.createVTKscaf("N3n",data))
# Get the output
self.GetOutput().ShallowCopy(rg)  # if inside Paraview's
                                  # programmable filter
```

At first the module is imported (this requires to have added `bit.sea` folder to the `PYTHONPATH`, see Sec.2.3), and then the class is initialized by giving as input the full path to the `MESHMASK` folder.
In the next step the resolution is set, possible arguments of `SetResolution` are `low, mid` and `high`.
At this point we are ready to create the vtk rectilinear grid `rg`: the `CreateRectilinearGrid` method returns a vtkRectilinearGrid variable (see vtk documentation) that acts as a container for both the raw NetCDF4 variables, and the mask variables, but also contains some metadata informations (like the path to `MESHMASK` and the selected resolution) which can be accessed by the subsequent filters.
Variables can now be added to the grid, starting from the masks: the `AddMask` method allows to select which mask shall be added to the grid; an example of coast and sub-

basins masks is presented in Fig.3.1: essentially a different integer value is assigned to each region of the sea, so that with a simple threshold filter acting on the right mask variable one can select any desired region.

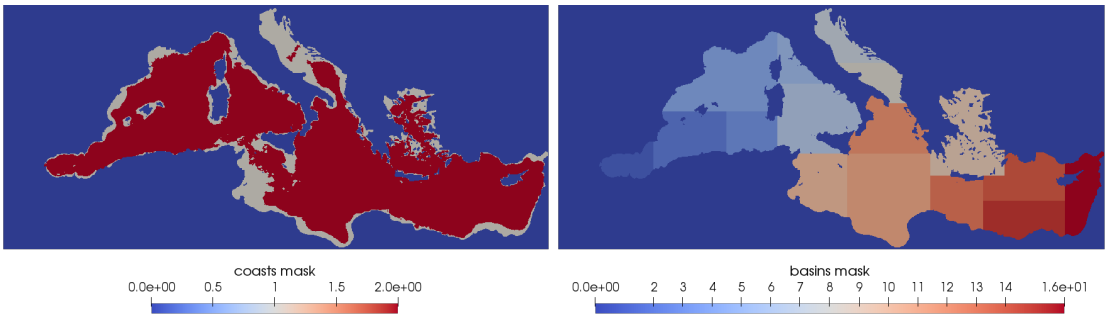


Figure 3.1: Left: top view of coast mask variable, highlighting open sea regions (red) and coasts region (white). Right: top view of basins mask variable. Every sub-basins corresponds to a different integer value, from 1 to 16.

The last steps are about adding an actual variable to the grid: at first the variable must be loaded in a numpy array through the method `GetVariable`. The loaded array must be converted to a vtk float array before it can be added to the grid, and this is done by the `createVTKscaf` method. An example of how a variable is visualized is shown in Fig.3.2, where only the domain's cells related to the sea are kept.

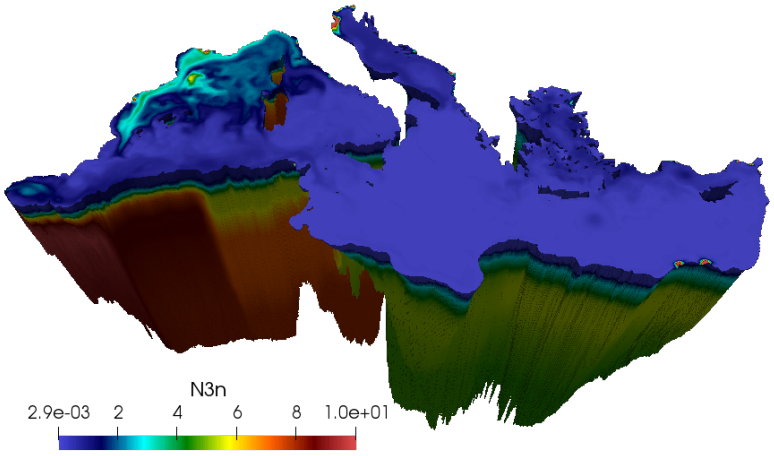


Figure 3.2: Example of the visualization of a Nitrate variable ("N3n") in $mmol/m^3$, after a threshold filter for removing the land cells from the grid.

Once one or more biogeochemichal variables are added, the grid is then ready to be sent to the output. Inside a Paraview's programmable filter this can be done by the

`self.GetOutput()` call, while optionally one can decide to dump the result to a vtk file with the `WriteGridToVtk` method.
It is useful to measure the time required to fulfill the whole sequence presented in the example, and also the actual memory (RAM) consumption required for on-the-fly visualization. These information are presented in Fig.3.3.

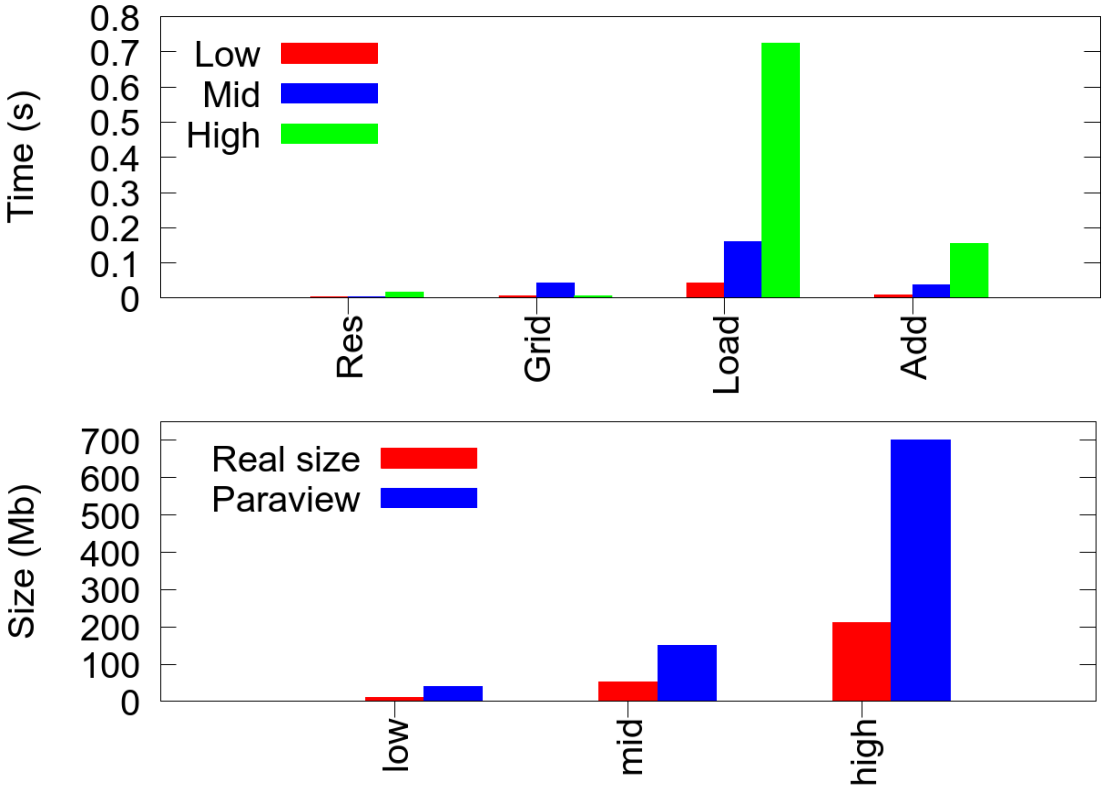


Figure 3.3: **Top:** execution time for the following methods: `SetResolution` (Res), `CreateRectilinearGrid` (Grid), `LoadNetCDF` (Load) and `createVTKscaf` (Add). The total time required is less than 1 second also for the high resolution case. **Bottom:** RAM consumption (blue) and actual array size (red) for all the resolutions.

As shown from Fig.3.3, the manipulation performed through the `OGSParaviewSuite` module is extremely efficient and allows to produce a visualizable output in less than $1s$ also for the less favourable case (high resolution). The drawback is paid in terms of memory usage, as this process introduces a noticeable overhead of roughly a factor of 3 with respect to the original variable size, giving some constraints on the maximum number of variables that can be loaded depending on the total free memory of the system.

3.2.2 LoadNetCDF

This is an auxiliary class used for automatically convert a biogeochemical variable stored inside a NetCDF file to a numpy 3D array. As can be observed from the example of the previous subsection, the `LoadNetCDF` class is embedded in the `OGSvtk` class, so that usually there is no need to initialize it separately.
For those cases in which a separate initialization of this class is required, a short description of how to use it is presented:

```
# Class initialization
LoadNetCDF = LoadNetCDF(z,y,x)    # Requires grid dimensions
                                  # of loaded variable
# Loads single variable
data = LoadNetCDF.GetVariable('path','name') # Result is
                                             # 3D numpy array
# Loads variables from list at given date
list = LoadNetCDF.GetListVariablesAtDate('path','date',var_list)
# Loads all variables at given date
list = LoadNetCDF.GetAllVariablesAtDate('path','date')
```

If initialized individually, the class requires as input the rectilinear dimensions of the imported variable. This step is skipped when the `OGSvtk` class is used, since it automatically initialize an `LoadNetCDF` class when the resolution is set. One way to proceed for a manual initialization is to load the file `MESHMASK/any_resolution/dims.npy` on a python shell and print it, since it contains exactly the information about grid size for every resolution.
The class then provides three methods for extracting variables: `GetVariable`, which allows to load a single variable, `GetListVariablesAtDate` and `GetAllVariablesAtDate`, which instead can be used for load more variables at once. In the latter cases the result comes in a list in which all elements are in the form `['var name',3D numpy array]`.

3.2.3 OGSStatistics

This class is the most complex and it is used to perform statistical analysis on a desired domain. The class' methods compute average, standard deviation, minima, maxima and percentiles for both areal (i.e. for any surface) and volume analysis. All the heaviest computations are performed through a custom C++ library, boosting up the computational speed and allowing this class to be efficiently used in a Paraview filter in order to keep the Pipeline responsive.
Two main procedures for evaluating statistics are offered by the class: the `FastStatistics` and the `FlexibleStatistics` methods. The former allows to work in all the regions defined by the masks, so that the cells of interest are accessed through an array slicing and all the preliminary informations, like cells surface and volume, are loaded from binary files; at this level most of the computational operation are made at python level by exploiting efficient array slicing, so that ctypes is needed only for the percentiles evaluation. The latter indeed offers the possibility to extract statistics on every kind of domain, allowing for example to cut or threshold the domain in any desired form, and

the information is extracted on the fly; since in this case a foor loop over all the cells is required (since they are not known a priori), all the routines are written in C++ and linked with ctypes in order to boost the speed of the process. Anyway, the execution is still slower with respect to the `FastStatistics` methods.
In the following the usage of the class is explained, starting again from a simple example:

```
# Initialize classes
OGS = OGSvtk('/MESHMASK')
OGSStatistics = OGSStatistics('/path_to_C_library')
# Define Grid and load data
OGS.SetResolution('mid')
rg = OGS.CreateRectilinearGrid()
OGS.AddMask(rg)
data = OGS.LoadNetCDF.GetVariable('data','N3n')
rg.GetCellData().AddArray(OGS.createVTKscaf('N3n',data))
# Compute Statistics
perc   = [0.05, 0.50, 0.95] # Percentiles
depths = [200,1000,2000]    # depths for volume stat
Areal  = OGSStatistics.FastArealStatistics(OGS,rg,sub_list,
                 'coast',var,perc)        # Areal Stat
Volume = OGSStatistics.FastVolumeStatistics(OGS,rg,sub_list,
                 'coast',depths,var,perc) # Volume Stat
```

As seen from the example, the `OGSStatistics` class requires a vtkRectilinearGrid with variables loaded in it, and thus the use of `OGSvtk` is necessary.
To initialize this class, the full path to the folder containing the C++ library must be given as argument; then a vtkRectilinearGrid is created with the `OGSvtk` 's methods and loaded with variables.
At this point the methods offered by `OGSStatistics` class can be used. In the example only the fast routines are showed, since the syntax of the flexible ones is very similar.
Two different methods are used for evaluating Areal and Volume statistics, and both take as arguments an `OGSvtk` object (OGS), a vtkRectilinearGrid (rg), a list with the sub-basins names (sub_list), a string for the coast levels ('coast'), the variable on which to perform the statistic (var) and an array with the percentiles to be evaluated (perc). The volume case requires an extra argument which corresponds to the depth levels to be integrated (depths). The output comes in a numpy 2D array with the depth levels on rows and the relative statistic values on columns, following this order: average, std, minima, percentiles and maxima. The output can be arranged into a vtkTable in order to be viewed inside Paraview's GUI, where can be plotted with a provided macro for obtaining plots as the ones showed in Fig.3.4.

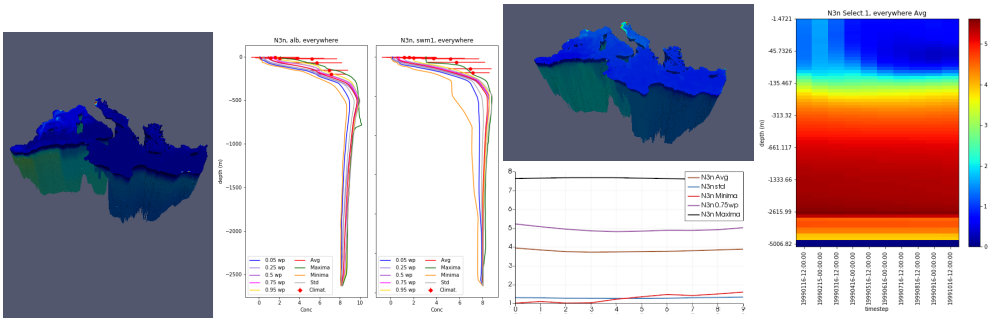


Figure 3.4: Some examples of what can be obtained from the `OGSStatistic` class through two screenshot of the Paraview's GUI. On the left, a depth analysis on two different sub-basins that shows relevant statistics (solid lines) and climatology (red points). On the right two examples of time-dependent statistics analysis are shown: an Hovmoller plot and the behaviour of some statistics quantities at fixed depth over time.

It is of interest to benchmark the performances also in this case. At first, the boost introduced by the use of ctypes for linking the python code to the C++ libraries is investigated and showed in Fig.3.5: here the three routines that have a ctypes implementation have been tested against their python corrispective and the histogram shows the ration between C++ and python times. The aforementioned routines are `FastStatistics`, `FlexibleStatistics` and `Statistic_to_vtkUnstructuredGrid` (Prop. in FIg.3.5, for "Propagation"). The first two of them were already mentioned, while the last one is an auxiliary function which allows to propagate the statistics results on the 3D view inside Paraview.

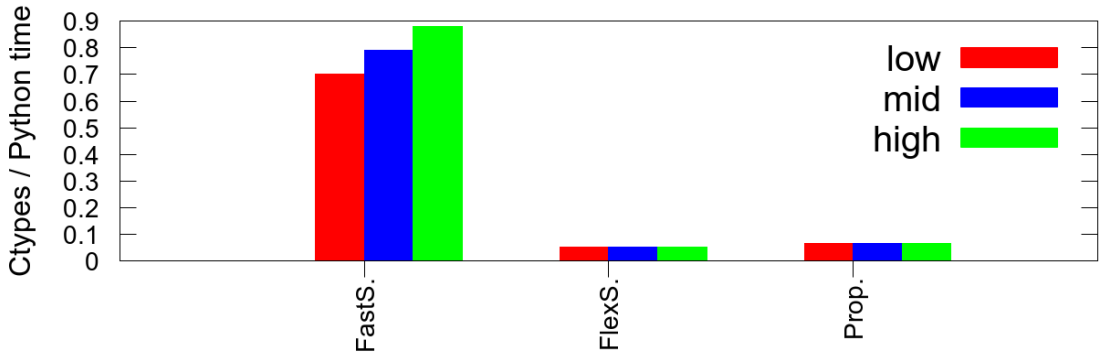


Figure 3.5: Histograms of the ratio between C++ time and python time for three different functions and resolutions.

As seen from Fig.3.5, the ctypes approach allows to obtain a relatively small speed-up ($10\% \div 30\%$) in the case of Fast methods, but the speed-up raises up to 90% for the other cases, so that an implementation of those without ctypes would be simply too slow for being used dynamically in a Paraview's pipeline.
Another important analysis is the total time required for the statistical routines to give back the results. In Fig.3.6 the execution time for the Fast and Flexible statistics is evaluated for all the resolutions by using the whole domain (which is the worst scenario) as test case.

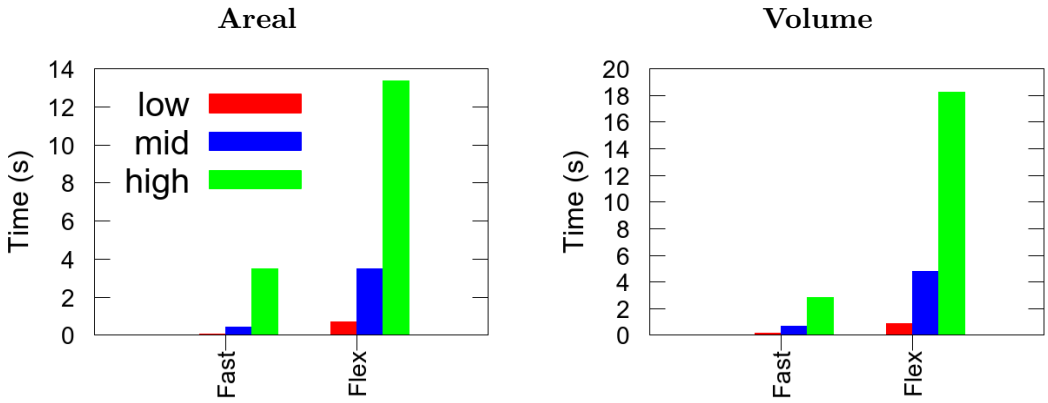


Figure 3.6: Histograms of the execution times for Fast and Flexible statistics at all the available resolutions, for both areal and volume analyses, using the whole domain as test case. As expected, Fast statistics is faster.

The plot shows, how expected, that Fast routines are much faster then Flexible ones, but it must be kept in mind that Flexible statistic has been developed for analyze small (and arbitrary) portions of the sea, and when the domain is on the order of a sub-basin size, Flexible statistics is also very fast.

3.3 From Python to Paraview Plugins

Now that the basic tool for filter developing has been explained in detail, the approach to plugins development is explained in this section. Since the objective is to obtain Paraview plugins, we must follow a specific way of writing the python scripts for the filter, i.e. the scripts must be splitted among three sections: the first one hosts some global variables that are used for defining the filter's name and output type, and a dictionary with all the variables that must appear in the filter's graphic interface, then the remaining sections are represented by two functions called `RequestInformation` and `RequestData`.
Essentially, `RequestInformation` is used for preparing the pipeline in the case that some information is required before the main code is executed (we have used it only for dealing with timesteps and sources), while `RequestData` hosts the main body of the code.
Let's try to go through this structure using a very simple example, i.e a filter that allows to load two biogeochemical variables (nitrate and phosphate) from the dataset:

Listing 1: custom filter script example

```
Name  = 'OGSImportNetCDF'
Label = 'OGS Import NetCDF'
Help  = 'Imports NetCDF data from OGS'
```

```
NumberOfInputs = 0 # programable source
InputDataType = ''
OutputDataType = 'vtkRectilinearGrid'
ExtraXml = ''

Properties = dict(
        File_date             = '20150605-12:00:00',
        Mesh_resolution       = 1, # To work with enum 1: high 2:
           mid 3: low
        Path_to_mesh          = '/MESHMASK/',
        Path_to_python        = 'path_to_bit.sea/',
        Path_to_ave_freq          = '/INPUT/',
        # Biogeochemical variables
        V_N1p                 = True,
        V_N3n                 = True,

def RequestInformation():
        executive = self.GetExecutive()
        outInfo = executive.GetOutputInformation(0)
        dataType = 10 # VTK_FLOAT
        numberOfComponents = 1
        vtk.vtkDataObject.SetPointDataActiveScalarInfo(outInfo,
           dataType, numberOfComponents)

def RequestData():
        import sys, os
        import numpy as np

        from OGSParaviewSuite import OGSvtk

        # Initialize class
        OGS = OGSvtk(Path_to_mesh);

        # Mesh resolution
        if Mesh_resolution == 1: OGS.SetResolution('high');
        if Mesh_resolution == 2: OGS.SetResolution('mid');
        if Mesh_resolution == 3: OGS.SetResolution('low');

        # Create a VTK rectilinear grid
        rg = OGS.CreateRectilinearGrid();

        # Initialize the sub masks
        OGS.AddMask(rg,AddSubMask=True,AddCoastMask=True,AddDepth=
           True);

        # Append the date to the dataset
        rg.GetFieldData().AddArray( OGS.createVTKstrf("Date","%s"
           % File_date) );
```

```
# Process biogeochemical variables
varn_list = [ "N1p", "N3n"]
varb_list = [ V_N1p, V_N3n]
# Postprocess variables in batch mode
for ii in range(0,len(varn_list)):
        if varb_list[ii]:
        fname = "%s/ave.%s.%s.nc" % (Path_to_ave_freq,
            File_date,varn_list[ii])
        # Load from NetCDF
        data = OGS.LoadNetCDF.GetVariable(fname,varn_list[
            ii])
        # Write data
        rg.GetCellData().AddArray(OGS.createVTKscaf(
            varn_list[ii],data))

# Output the file to ParaView
self.GetOutput().ShallowCopy(rg);
```

The global variables at the beginning of the filters define some metadata informations, like the filter's name that will appear in Paraview's GUI. The variables related to the Input and Output are very important since define how to use the filter in a pipeline: it can be "attached" only to a filter that produces an output correspondent to this filter Input. Since in this case the filter is a source, Input information is blank.
The variables defined into the `Properties` dictionary will appear in the filter's GUI and allow the user to dynamically interact with the filter. In this case the `RequestInformation` is necessary as the filter is a source and must initialize a pipeline. It hosts some basic information related to the data handled by the filter.
The main script is defined inside the `RequestData` and it allows to select the variables that must be loaded in the grid (in this example case just two for simplicity).
Once the python script is complete, it must be converted to a proper `.xml` file. This can be done by a script provided by Paraview developers, that do almost all the job for us, but it is not perfect. The command line is:

```
python python_filter_generator.py "input_file".py "output_name".xml
```

At this point the `.xml` file can be loaded in Paraview and tested.
As aforementioned, the automatic conversion provided by the generator script is not perfect as the variables order defined in the `Properties` dictionary inside the python scripts will be not be respcted. Indeed in the `.xml` file these varaibles will appear in alphabetical order and in order to prevent this, some `PropertyGroup` sections can be manually added to the xml files. These sections allow to group some varialbes togheter, give a name to the group and control the order of appearance in the Paraview's GUI. The correct way to use `PropertyGroup` is shown in the code below:

```
    # How variables appear in the xml code
<IntVectorProperty
  name="V_N1p"
```

```
  label="N1p"
  initial_string="V_N1p"
  command="SetParameter"
  animateable="1"
  default_values="0"
  number_of_elements="1">
  <BooleanDomain name="bool" />
  <Documentation></Documentation>
</IntVectorProperty>
<IntVectorProperty
  name="V_N3n"
  label="N3n"
  initial_string="V_N3n"
  command="SetParameter"
  animateable="1"
  default_values="1"
  number_of_elements="1">
  <BooleanDomain name="bool" />
  <Documentation></Documentation>
</IntVectorProperty>

# To be added manually:
<PropertyGroup
  panel_visibility="default"
  label="Biogeochemical variables">
  <Property name="V_N1p" />
  <Property name="V_N3n" />
  <Documentation>
      Activate/Deactivate variables coming
      from AVE_FREQ files
  </Documentation>
</PropertyGroup>
```

The `PropertyGroup` section allows also to add labels to the included variables, and add a short documentation to them.
Note: if a filter needs to deal with time steps, also the following piece of code must be added manually to the `.xml` file:

```
<DoubleVectorProperty information_only="1"
                      name="TimestepValues"
                      repeatable="1">
<TimeStepsInformationHelper />
<Documentation>Available timestep values.</Documentation>
</DoubleVectorProperty>
```

It is too dispersive to explain in every detail how filter programming works in all the different cases and, moreover, that would go beyond the scope of this manuscript. An interested reader can investigate the source scripts contained in the `PW_filters` folder in the repository for further informations and examples.

3.4 Paraview Custom Filters in Action

Now that all the steps beyond the development of a custom filter have been highlighted, we are able to present the final product of the work, embodied in a complete suite for data visualization and analysis for the OGS MEDBFM model's datasets.
In this section some pipeline examples will be showed, starting from the simplest case aimed to visualize any desired region of the sea, and then adding some filters for statistical analysis.
The typical pipeline to visualize data is presented in Fig.3.7, as well as its appearance in the Paraview's GUI.

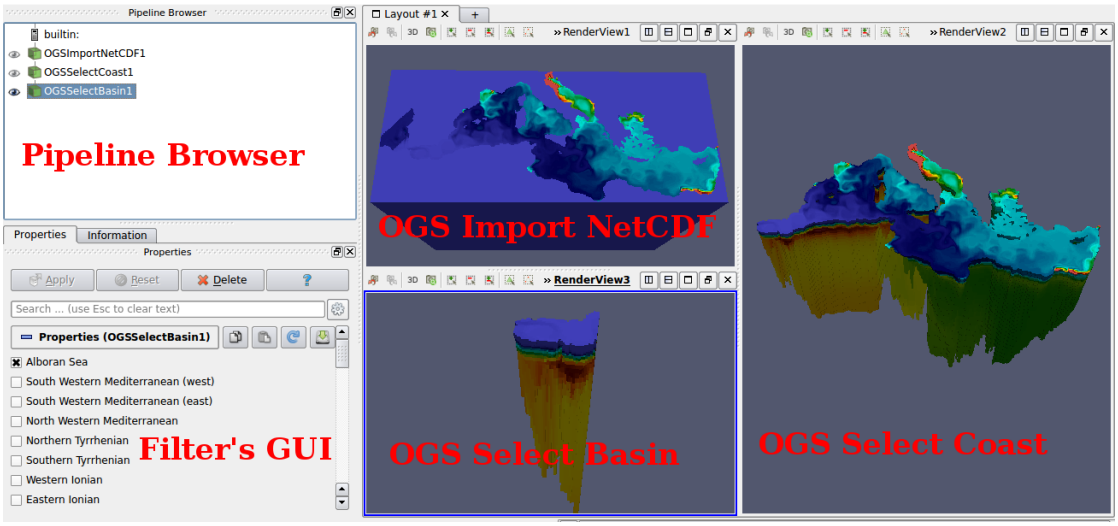


Figure 3.7: Screenshot from Paraview application. On upper-left corner it is shown the pipeline browser, where the three linked filters appear. On the bottom-left corner the OGS Select Basin custom GUI shows that only the Alboran sub-basin has been choosen for the visualization. The right side of the image hosts three render views, one for every step of the pipeline.

Every step of the pipeline allows to visualize a different level of the grid, so that the first component (OGS Import NetCDF) produces the raw grid populated with the desired variables, then the second component (OGS Select Coast) cuts away land sections and allows to select coast level. The last component (OGS Select Basin) is used to select the desired sub-basins for visualization.
One of the most interesting aspects of the Paraview's pipeline is its dynamic behaviour, so

for example, if some changes are performed at the top level filter (OGS Import NetCDf in this case), the effects of such modifications will be propagated downstream. A very useful application of this feature is the possibility to obtain movies representing the evolution of the system over the timesteps by just using the "OGS Import NetCDF (time series)" filter and press the "play" button: all the timesteps will be loaded sequentially and the pipeline will be applied to each of them.
Let's now apply some statistics by starting, as an example, from the depth analysis on the Alboran Sea through the "OGS Stat Table" filter, as shown in Fig.3.8. All the results obtained from a statistical filter come in a vtkTable (which is analogous to an Excel sheet) that can be plotted using provided macros.

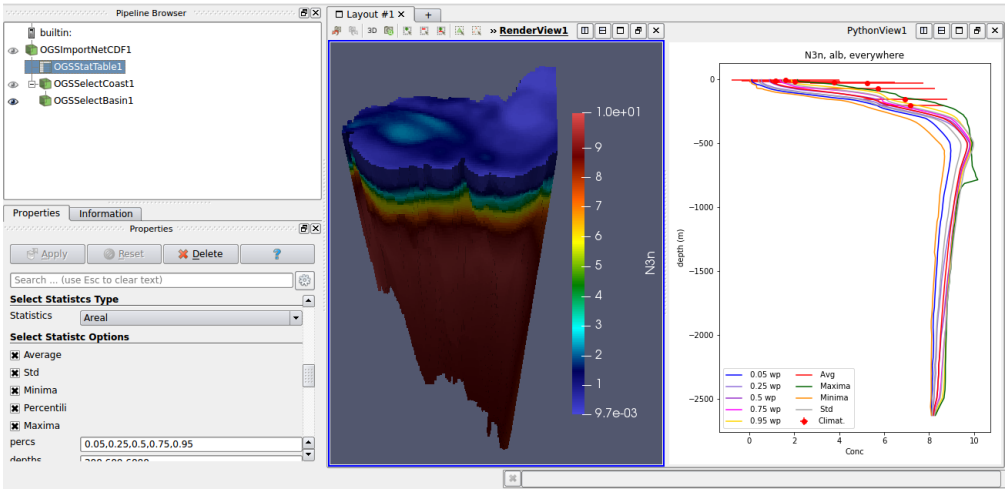


Figure 3.8: Snapshot of the Paraview's GUI. In this case the "OGS Stat Table filter" is used for obtaining the statistics of the Alboran sub-basin. The obtained values are then plotted using the provided macro PythonStatPlot. Note that the "OGS Stat Table" filter is applied directly to the "OGS Import NetCDF" filter.

This approach is based on the Fast statistic routines of the OGSParaviewSuite module, so that is the fastest way to perform statistical analyses, but it only allows to examine fixed regions of the domain. Indeed, as can be seen from Fig.3.8, this filter directly applies on the "OGS Import NeCDF" filter, since it only needs to take the desired variables from it, then the domain selection for the analysis is made through the filter's GUI.
In order to perform analysis on any domain's region, the right instrument to use is the "OGS Custom Stat Table" filter, as shown in the example reported in Fig.3.9.

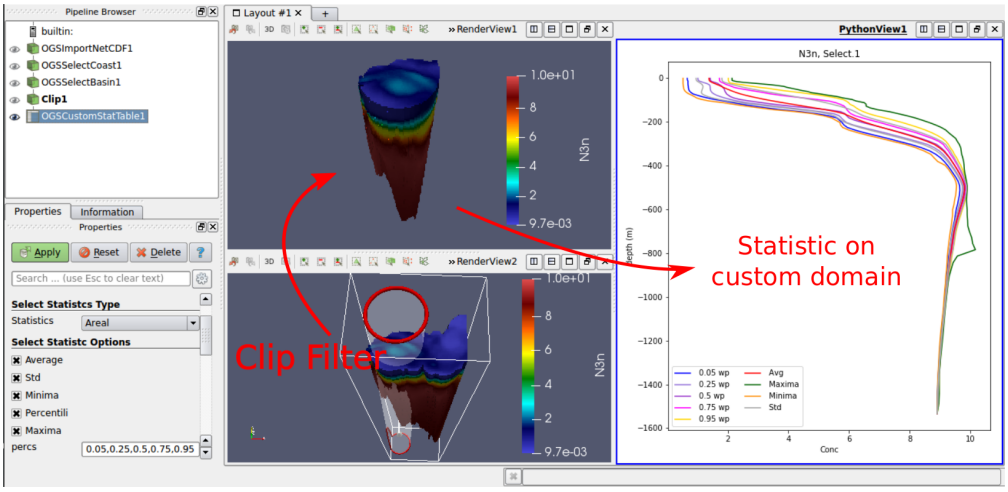


Figure 3.9: Snapshot of the Paraview's GUI. Now "OGS Custom Stat Table filter" is directly applied to the result of a clip filter acting on the Alboran sub-basin. The statistic is computed on the arbitrary region extracted by the clip filter and plotted with the "PythonStatPlot" macro.

In this case, a section of the Alboran sub-basin is selected through a clip filter and then the statistic is evaluated on this region using the "OGS Custom Stat Table" filter. The interaction between Paraview's existing filters with the new custom filters allows a completely new class of investigations, since for the first time a statistical analysis can be focused on very small regions of the sea, or on specific sections, in a very easy and intuitive way.
Not only vertical profiles analysis, but also time investigation is possible. The *sources* which deal with time analysis are "OGS Hovmoller Plot" and "OGS Time Statistic". The former provides the information required for producing an Hovmoller plot, which contains a depth analysis for every time steps of a specified statistical quantity (average for example). The latter instead is used to perform a 2D plot, with time on **x** axis, of the desired statistics at a fixed depth. Different values of time aggregation can be selected among, ranging from monthly to yearly, and allowing to obtain useful averages of the desired quantities also over long periods.
Two examples of a possible application of these sources are shown in Fig.3.10.

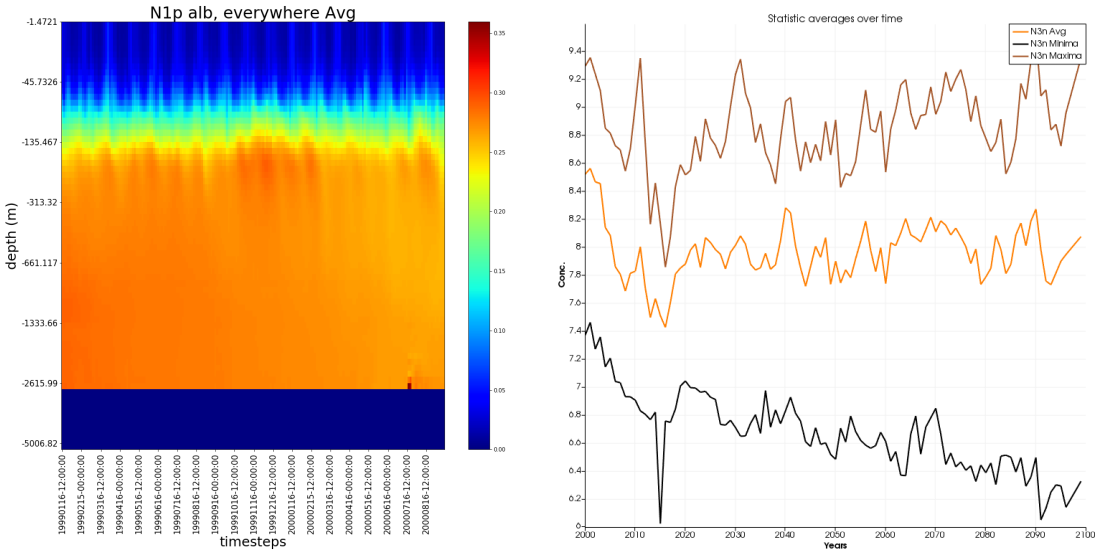


Figure 3.10: **Left:** "OGS Hovmoller Plot" filter's output, showing the depth distribution of the N3n variable average from a two years simulation. In order to obtain the plot, the macros `HovmollerPLot` must be used. **Right:** plot of the "OGS Time Statistic" filter's output.The plot is obtained using the Paraview's Plot Data filter on the output of "OGS Time Statistic" and it shows Minima, Maxima and Averages of variable N3n on the Alboran sub-basin over a 100 years simulation.

These kinds of applications require to manipulate a large number of files, since long climatologic simulations can easily compute more than 1000 timesteps, and thus the same amount of files is saved for every variable. The time required to process such big number of files can become large, and this ends up to be a problem for pratical usage of these filters, as shown in Fig.3.11.

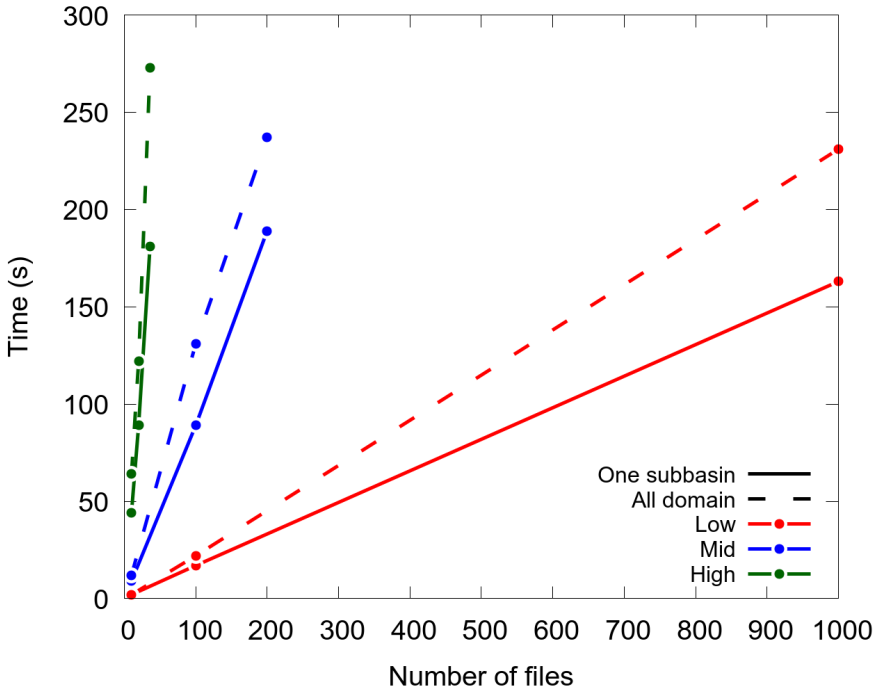


Figure 3.11: Execution time of "OGS Time Statistic" filter against number of files on all domain (dashed) and smallest sub-basin (solid), for each resolution. While for the low resolution case the execution time is limited until the database becomes very large (on the order of 1000 files) for the medium and high resolution it quickly raises, reducing the actual praticity of this kind of applications.

In order to speed up these applications, parallelization is required. In Fig.3.12 the speed up obtained with a parallel version of the "OGS Time Statistic" filter is shown. The parallel implementation has been done through the exploitation of `mpi4py` python module, using a classic data distribution approach.

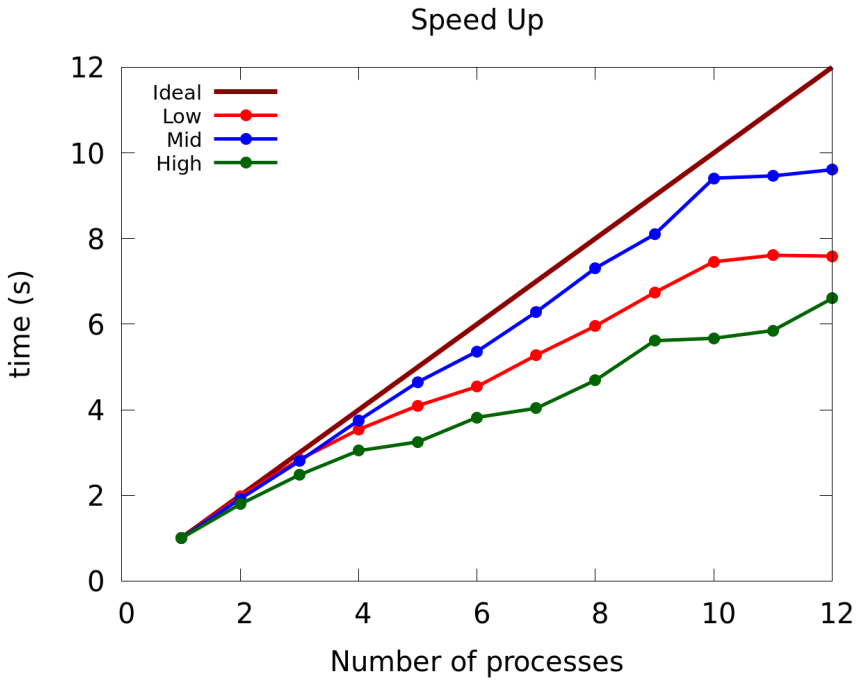


Figure 3.12: Speed up obtained with a parallel version of the "OGS Time Statistic" filter.

As shown by the plot above, the speed up in this configuration is evident and now the filter allows to obtain a result over an analysis on more than 1000 files in roughly $15s$, running on 12 processors.
Unfortunately, in order to exploit a filter that offers a parallelization, Paraview must run in client-server mode, i.e. a Paraview server must be launched in parallel on a server machine and then the user has to connect to it through a local client (see Sec.2.2). This configuration requires the user to have the same version of Paraview as the one present on the server installed on its local machine.

4 Documentation

This section will present a detailed documentation of both the `OGSParaviewSuite` and custom filters. The `OGSParaviewSuite` classes and methods documentation is hosted in subsection 4.1, where some examples are also provided in order to show their workin principles. A detailed list of all custom filters features and properties is presented in subsection 4.2.

4.1 OGSParaviewSuite

The `OGSParaviewSuite` python module groups the classes used for low level vtk, NetCDF and Statistics manipulation of OGS native datasets. The main purpose is to connect the existing OGS structure with complex vtk manipulation through classes methods, so that the complex vtk manipulations are hidden to the user.
Three classes are defined inside the module:

- OGSvtk
- LoadNetCDF
- OGSStatistic

4.1.1 OGSvtk

```
from OGSParaviewSuite import OGSvtk
```

This class provides the basic tools for the conversion to raw NetCDF files to a vtkRectilinearGrid that can be visualized into Paraview.
In the following boxes the class methods are explained, and a general example is provided at the end.

Public Member Functions

`__init__(self,maskpath)`

Parameters:

maskpath: string.
Full path to meshmask folder.

Description: initialize OGSvtk class. Maskpath folder must contain three subfolders called low, mid and high in which the files required by the class are stored (that can be generated using the scripts `mask_extractor.py` and `LonLattoMeters.py`).

```
SetResolution(self,resolution)
```

Parameters:

resolution: string.
Value of the resolution to be set. Accepted values are `'low'`,`'mid'` and `'high'`

Description: Used for setting the resolution of the vtk structure accordingly with the OGS input data that will be used. Once the `SetResolution` method is used, the following binaries files are loaded from the respective resolution folder, and stored in an internal class variable: `Lon2Meters.npy, Lat2Meters.npy, nav_lev.npy, dims.npy`. All this files are generated with the scripts `mask_extractor.py` and `LonLattoMeters.py`. Also an internal `LoadNetCDF` class is initialized.
This method should be used just after the class initialization since it is required by all the following methods.

```
CreateRectilinearGrid(self)
```

Returns:

out: vtkRectilinearGrid.

Description: returns a vtkRectilinearGrid object with the proper dimensions, accordingly with the resolution set by the `SetResolution` method. The output contains also two string arrays containing the full path to `meshmask` folder and the resolution.

```
AddMask(self,input,AddSubMask=True,AddCoastMask=True,AddDepth=True)
```

Parameters:

input: vtkRectilinearGrid.

AddSubMask: boolean.
If `True`, adds the sub-basins mask to the input vtkRectilinearGrid

AddCoastMask: boolean.
If `True`, adds the coasts mask to the input vtkRectilinearGrid

AddDepth: boolean.
If `True`, adds the depth mask the to input vtkRectilienarGrid

Description: adds different masks to the rectilinear grid as CellData variables. The masks variables will appear in the Paraview GUI with all the other biogeochemical variables. Masks are used by normal and custom threshold filters in order to dynamically select different regions of the sea for both visualization and statistics evaluation.

```
createVTKscaf(self,varname,data,type = vtk.VTKFLOAT)
```

Parameters:

varname: string.
Name of the resulting vtkFloatArray.

data: np.array.
Three dimensional numpy array containing the data related to a native varaible. The correct format is just the one obtained after the extraction using one of the methods of the LoadNetCDF class.

type: vtkDataFormat.
Specifies the data format, default is VTKFLOAT.

Returns:

out: vtkFloatArray.
A vtkFloatArray conversion of the input **data** in np.array format.

Description: creates a vtk scalar field array from variable **data** with a specific variable name.

```
createVTKvecf3(self,varname,xdata,ydata,zdata,type=vtk.VTK_FLOAT)
```

Parameters:

varname: string.
VTK variable name

xdata, ydata, zdata: np.array.
Numpy array containing variable data of 1st, 2nd and 3rd dimensions.

type: vtkDataFormat.
Specifies the data format, default is VTKFLOAT.

Returns:

out: vtkFloatArray.
A 3D vtkFloatArray conversion of the input data in np.array format.

Description: Creates a VTK 3D vector field array from variable data with a specific variable name.

`createVTKstrf(self,varname,data)`

Parameters:

varname: string.
Name of the output vtkStringArray.

data: string.
String variable that will be added into the first index of the output vtkStringArray.

Returns:

out: vtkStringArray.
A vtkStringArray containing the string variable **data**. The output array name will be set with string variable **varname**

Description: Creates a VTK string field array of dimension 1 from a string variable **data**, with a specific variable name. This is usually used in order to add some metadata to the vtkRectilinearGrid.

`LoadVolume(self)`

Returns:

out: np.ndarray.
A 3D numpy array containing the information of each cells' volume.

Description: loads from the respective resolution folder the binary `volume.npy` into a np.ndarray.

`LoadArea(self)`

Returns:

out: np.ndarray.
A 3D numpy array containing the information of each cells' area.

Description: loads from the respective resolution folder the binary `area.npy` into a np.ndarray.

`WriteGridToVtk(self,inp,savename)`

Parameters:

inp: vtkRectilinearGrid or vtkUnstructuredGrid.

savename: string.
Full path of vtk file to save on disk.

Description: saves on disk at a given path `savename` the input vtk Grid. Useful for batch testing or for export an analysis to a computer which does not have access to OGS databases.

Example

```
# Load module
from OGSParaviewSuite import *

# Initialize class
OGS = OGSvtk('/home/user/MESHMASK')
OGS.SetResolution('mid')

# Create vtkRectilinearGrid
rg = OGS.CreateRectilinearGrid()
OGS.AddMask(rg)

# Load data from NetCDF
data = OGS.LoadNetCDF.GetVariable('/home/user/database/ave
    .19990116-12:00:00.N3n.nc','N3n')

# Add data to vtkRectilinearGrid
rg.GetCellData().AddArray(OGS.createVTKscaf("N3n",data))

# Write vtkRectilineargrid on disk
OGS.WriteGridToVtk(rg,"test.tk")
```

4.1.2 LoadNetCDF

```
from OGSParaviewSuite import LoadNetCDF
```

This class provides the basic functions required to open and store the OGS biogeochemical variables produced by the model as NetCDF files into numpy arrays.

Public Member Functions

`__init__(self, jpk, jpj, jpi)`

Parameters:

jpk, jpj, jpi: integer.
Respectively z, y and x dimensions of the NetCDF variables.

Description: initializes LoadNetCDF class. The proper value of `jpk, jpj, jpi` can be obtained from `OGSvtk.SetResolution` method. Note that the usage of this class as stand-alone can be usually avoided since it is included into the `OGSvtk` class.

`GetVariable(self,filename,varname,maxval=1e20)`

Parameters:

filename: string.
Full path to the NetCDF file

varname: string.
Name of the variable to extract

maxval: float.
value from which start to convert to 0, default = 1e20

Returns:

out: np.ndarray.
A 3D numpy array containing the extracted variable.

Description: extracts a selected variable from a NetCDF4 file and stores it into a 3D numpy array.

`GetAllVariablesAtDate(self,filename,date)`

Parameters:

filename: string.
Full path to the NetCDF file.

date: string.
Date corresponding to extracted variables, must be in the format 'yyyymmdd-hh:mm:ss'.

Returns:

out: list.
List of touples, each list element is in the form [varname, np.ndarray].

Description: extracts all variables at a selected date and stores the result into a list of touples. Each element is composed by the variable name and the corresponding 3D np.ndarray.

`GetListVariablesAtDate(self,pathtodata,date,varlist)`

Parameters:

filename: string.
Full path to the NetCDF file.

date: string.
Date corresponding to extracte variables, must be in the format 'yyyymmdd-hh:mm:ss'.

varlist: list of strings.
List containing the names of the variables to be extracted.

Returns:

out: list.
List of touples, each list element is in the form [varname, np.ndarray].

Description: Extracts variables in `varlist` at a selected date and stores the result into a list of touples. Each element is composed by the variable name and the corresponding 3D np.ndarray.

faces2cellcenter(self,U,V,W)

Parameters:

U: np.ndarray.
1st component of the Velocity.

V: np.ndarray.
2nd component of the Velocity.

U: np.ndarray.
3rd component of the Velocity.

Returns:

out1, out2, out3 : np.ndarrays.
Three components of the converted face-centered velocity.

Description: Converts a face centered velocity into a cell centered field. Useful for further use on VTK grids.

Example

```
# Load module
from OGSParaviewSuite import *

# Initialize class
OGS = OGSvtk('/home/user/MESHMASK')
OGS.SetResolution('mid')

LoadNetCDF = LoadNetCDF(OGS.jpk,OGS,jpj,OGS.jpi)

# Load variables

SingleVarData = LoadNetCDF.GetVariable('/home/user/data/ave
    .19990116-12:00:00.N3n.nc','N3n')

MultiVarData = LoadNetCDF.GetVariablesAtDate('/home/user/data/','
    19990116-12:00:00')
```

4.1.3 OGSStatistics

```
from OGSParaviewSuite import OGSStatistics
```

This class relies on the use of precompiled C++ shared libraries for the most intensive routines. These libraries are linked to the python module with ctypes.
The class provides efficient algorithms for evaluating statistical quantities of interest, for example averages, percentiles and standard deviations, over areal and volume domain analyses. As well as other classes implemented in OGSParaviewSuite module, OGSStatistics is built with

the aim to produce fast and efficient Paraview pipelines through the implementation of custom filters that relies on the functionalities provided by the module.
The results of the statistics can be visualized using custom OGS Paraview filters, in the 3D space or in a spread sheet table that can be plotted using provided macros.

Public Member Functions

`__init__(self,librarypath)`

Parameters:

librarypath: string.
Full path to the shared c++ shared library.

Description: Initialize class by giving full path to the c++ shared libraries. Once initialized, all the c++ functions required by che class are instiantiated with the ctypes syntax.

`CoreStatistics(self,Conc, Weight,perc)`

Parameters:

Conc: np.array.
Numpy array containing the variable values on which perform the statistics analysis.

Weight: np.array.
Weight of the cell, i.e. the area or the volume of the relative cell.

perc: np.array.
Values of the percentiles on which perform the statistic. Its length must not be greater than 5.

Returns:

out: np.array.
Numpy array with lenght = 9 which stores for every element a different statistics, in the following order: average, std, minima, perc0, perc1, perc2, perc3, perc4, maxima. So that for example the average is accessible with `out[0]`.

Description: Core algorithm for statistics. It will be used by the two main statistics methods, `FastArealStatistics` and `FastVolumeStatistics`. It uses a C++ functions for evaluating percentiles, with a gain in performances up to 30% (see Subection 3.2) with respect to the equivalent function written in python. Note that this method should not be used directly by the user, since it is embodied in the aforementioned functions.

```
FastArealStatistic(self,OGSvtk,rg,sublist,coast,var,
            perc = [0.05,0.25,0.5,0.75,0.95],ReturnCutMask = False)
```

Parameters:

OGSvtk: OGSvtk.
A precedently initialized OGSvtk class variable. Note that the resolution must have been set before passing it as argument.

rg: vtkRectilinearGrid.
The vtkRectilinearGrid from which variables are extracted.

sublist: list of strings.
List containing the names of the sub-basins where the statistic analysis is executed. Note that the statistics will be evaluated on the aggregation of the sub-basins listed in `sublist`, so that if for example `sublist = ['alb','swm']` the statistic will be evaluated on the region of the sea 'alb'+'swm'. Admitted values for sub-basins are: alb, swm1, swm2, nwm, tyr1, tyr2, adr1, adr2, aeg, ion1, ion2, ion3, lev1, lev2, lev3, lev4.

coastlist: string.
Name of the coast level where statistic analysis is performed. Admitted values are `'coast'`, `'open_sea'` and `'everywhere'`.

var: string.
Name of the variable where statistic analysis is executed.

perc: np.array or list.
Values of the percentiles to evaluate. Default are `[0.05,0.25,0.5,0.75,0.95]`.

ReturnCutMask: boolean.
If `True` returns the cut mask (i.e. the cells on which the analysis have been performed) as a np.ndarray. Default is False.

Returns:

out: np.ndarrays.
Numpy ndarray with the shape `(len(nav_lev),9)` which stores for every column a different statistics evaluated at every depth level, in the following order: average, std, minima, perc0, perc1, perc2, perc3, perc4, maxima. So that for example the average is accessible with `out[:,0]`.

Description: main method for evaluating areal statistics of an input variable at selected a sub-basins and bathymetric region (coast level).

```
FastVolumeStatistic(self,OGSvtk,rg,sublist,coast,depth_list,
            var,perc = [0.05,0.25,0.5,0.75,0.95],ReturnCutMask = False)
```

Parameters:

OGSvtk: OGSvtk.
A precedently initialized OGSvtk class variable. Note that the resolution must have been set before passing it as argument.

rg: vtkRectilinearGrid.
The vtkRectilinearGrid from which variables are extracted.

sublist: list of strings.
List containing the names of the sub-basins where the statistic analysis is performed. Note that the statistics will be evaluated on the aggregation of the sub-basins listed in `sublist`, so that if for example `sublist = ['alb','swm']` the statistic will be evaluated on the region of the sea 'alb'+'swm'. Admitted values for sub-basins are: alb, swm1, swm2, nwm, tyr1, tyr2, adr1, adr2, aeg, ion1, ion2, ion3, lev1, lev2, lev3, lev4.

coastlist: string.
Name of the coast level where statistic analysis is performed. Admitted values are `'coast'`, `'open_sea'` and `'everywhere'`.

depth_list: np.array or list.
Depth levels on which the volume analysis is executed.

var: string.
Name of the variable where the statistic analysis is performed.

perc: np.array or list.
Values of the percentiles to evaluate. Default are `[0.05,0.25,0.5,0.75,0.95]`.

ReturnCutMask: boolean.
If `True` returns the cut mask (i.e. the cells on which the analysis have been performed) as a np.ndarray. Default is False.

Returns:

out: np.ndarrays.
Numpy ndarray with the shape `(len(depth_list+1),9)` which stores for every column a different statistics evaluated at every depth level, in the following order: average, std, minima, perc0, perc1, perc2, perc3, perc4, maxima. So that for example the average is accessible with `out[:,0]`.

Description: main method for evaluating volume statistics of an input variable at a selected sub-basins, bathymetric region and depths.

```
FlexibleArealStatistic(self,rg,nav_lev,var,
                           percs = [0.05,0.25,0.5,0.75,0.95])
```

Parameters:

rg: vtkUnstructuredGrid.
The vtkUnstructuredGrid from which variables are extracted.

var: string.
Name of the variable on which the statistic analysis is performed.

nav_lev: np.array.
Numpy array containing all depth values on which the variables are defined. Can be obtained from the variable `OGSvtk.nav_lev` after having set a resolution of a `OGSvtk` class.

perc: np.array or list.
Values of the percentiles to evaluate. Default are `[0.05,0.25,0.5,0.75,0.95]`.

Returns:

out: np.ndarrays.
Numpy ndarray with the shape `(len(nav_lev),9)` which stores for every column a different statistics eveluated at every depth level, in the following order: average, std, minima, perc0, perc1, perc2, perc3, perc4, maxima. So that for example the average is accessible with `out[:,0]`.

Description: method for evaluating areal statistic of any vtkUnstructuredGrid. It is slower with respect to `FastArealStatistics` since the computation of the areal surface of every cell is performed on the fly, but allows to perform statistic analyses on every kind of domain, so that it can be applied on the result of a clipping filter or a threshold filter, allowing to investigate domains of custom shape, that are not limited to basins or coasts fixed selections. Note that this routine heavily relies on C++ shared libraries.

```
FlexibleVolumeStatistic(self,rg,nav_lev,var,depth_list,
                            percs = [0.05,0.25,0.5,0.75,0.95])
```

Parameters:

rg: vtkUnstructuredGrid.
The vtkUnstructuredGrid from which variables are extracted.

var: string.
Name of the variable on which the statistic analysis is performed.

nav_lev: np.array.
Numpy array containing all depth values on which the variables are defined. Can be obtained from the variable `OGSvtk.nav_lev` after having set a resolution of a `OGSvtk` class.

depth_list: np.array or list.
Depth levels on which the volume analysis is performed.

perc: np.array or list.
Values of the percentiles to evaluate. Default are `[0.05,0.25,0.5,0.75,0.95]`.

Returns:

out: np.ndarrays.
Numpy ndarray with the shape `(len(nav_lev),9)` which stores for every column a different statistics evaluated at every depth level, in the following order: average, std, minima, perc0, perc1, perc2, perc3, perc4, maxima. So that for example the average is accessible with `out[:,0]`.

Description: method for evaluating volume statistic of any vtkUnstructuredGrid. It is slower with respect to `FastVolumeStatistics` since the computation of cells volume is performed on the fly, but allows to perform statistic analyses on every kind of domain, so that it can be applied on the result of a clipping filter or a threshold filter, allowing to investigate domains of custom shape, that are not limited to basins or coasts fixed selections. Note that this routine heavily relies on c++ shared libraries.

`Statistic_to_vtkUnstructuredGrid(self,grid,varname,input_array)`

Parameters:

grd: vtkUnstructuredGrid.
The vtkUnstructuredGrid on which statistics variable will be added.

varname: string.
Name of the new variable added on `grid`

input_array: np.array.
Array containing the statistics on every depth level. Its values will be propagated on relative surfaces of input grid in order to being able to visualize it in the 3D render view.

Description: method used to propagate an array with `len(nav_lev)` elements on a vtkUnstructuredGrid, in order to be able to visualize it in the 3D render view, as well as all the other variables. This method can be used for an intuitive visualizations of the results obtained from statistic analyses routines.

Example

```
# Load module
from OGSParaviewSuite import *

# Initialize class
OGS = OGSvtk('/home/user/MESHMASK')
OGS.SetResolution('mid')
# Create vtkRectilinearGrid
rg = OGS.CreateRectilinearGrid()
OGS.AddMask(rg)
# Load data from NetCDF
data = OGS.LoadNetCDF.GetVariable('/home/user/database/ave
    .19990116-12:00:00.N3n.nc','N3n')
# Add data to vtkRectilinearGrid
rg.GetCellData().AddArray(OGS.createVTKscaf("N3n",data))

# Start statistic evaluation
OGSStatistics = OGSStatistics('path/to/c++ lib')
Areal = OGSStatistics.FastArealStatistic(OGS,rf,['alb'],'
    everywhere',"N3n")
# Areal is now a 2D array with depths on rows and
# a different statistic value on every column
```

4.2 OGS Custom Filters

In this section the novel custom filters developed for OGS data analysis and visualizations are explained in details. All the filters have been developed using the `OGSParaviewSuite` python module and are implemented into Paraview as plugins, giving the possibility to add an intuitive custom interface to any of them.
There are two main kinds of custom filters: sources and filters. The main difference between a Paraview Source and a Paraview Filter is that a source does not need any input. Indeed, sources usually represent the starting point of a pipeline and does not depend on any other filter or source.
On the other hand, filters act on an input and produce an output, so that they are input-dependent and can be applied only if the previous pipeline component's output is of the correct type for the filter's input.
Table 1 shows all the developed custom plugins, grouped following these two main categories.

Sources	**Filters**
OGS Import NetCDF	OGS Select Coast
OGS Import NetCDF (time series)	OGS Select Basin
OGS Hovmoller	OGS Stat Table
OGS Time Statistics	OGS Custom Stat Table
OGS Parallel Time Statistics	OGS Okubo-Weiss Criterion
	OGS Select Okubo-Weiss
	OGS Domain Analysis

Table 1: List of all custom sources and filters

In the following a specific paragraph hosts a general description the relative source or filter, as well as a detailed table explaining the filter's inputs, outputs and properties.

4.2.1 Sources

OGS Import NetCDF

The source loads and visualize inside the Paraview 3D render view a desired list of variables from a source path defined by the **AVE FREQ path** property. The path must contains the original NetCDF files. In order to properly work, the MESHMASK folder must contains all the required files specified in Sec.2. The source also adds a mask for sub-basins and different coast levels required by other custom filters.

Property	Description	Default Value(s)	Restrictions
Output	This property defines the output of the source.	vtkRectilinearGrid	The output can only be a vtkRectilinearGrid
Date of the file	Insert the date of the NetCDF4 file to be loaded. Format is yyyymmdd-hh:mm:ss.	19990116-12:00:00	Inserted date must corresponds to the correct format and without blank spaces at the end.
Mesh resolution	Select the correct mesh resolution among Low, Mid or High.	High	Correct resolution with respect to selected file must be chosen, otherwise prints error.
AVE FREQ path	Path to data folder		Must be the exact path to data.
AVE PHYS path	Path to physical data folder		Full path to the location of AVE_PHYS data
FORCINGS path	Path to forcings data folder		Full path to the location of FORCINGS data
Time for the forcings	Forcings files are expected to have different dates.		
MESHMASK path	Path to MESHMASK folder		MESHMASK folder must contains three sub-directories: low, mid and high with inside all the required files. See Setup section.
Path to python classes	Path to bit.sea folder		
Biogeochemical variables	Select desired variables from list.	N3n	

OGS Import NetCDF (time series)

The source is analogous to **OGS Import NetCDF**, with the exception that the whole time series is loaded, therefore the property **Date of the file** is absent in this case.

OGS Hovmoller

The source produces an areal statistics of all timesteps and all depths between **Start date** and **End Date** properties. The output comes in a vtkTable and can be plotted with the provided macro HovmollerPlot in order to obtain the Hovmoller plot inside a Paraview's python view. The statistics type can be chosen among average, standard deviation, minima, percentiles and maxima, for every combination of coast levels and sub-basins.

Property	Description	Default Value(s)	Restrictions
Output	This property defines the output of the source.	vtkTable	
Mesh resolution	Select the correct mesh resolution among Low, Mid or High.	High	Correct resolution with respect to selected data must be chosen, otherwise prints error.
Path to ave freq	Path to data folder		Must be the exact path to data.
Path to mesh	Path to MESHMASK folder		MESHMASK folder must contains three sub-directories: low, mid and high with inside all the required files. See Setup section.
Path to python	Path to bit.sea folder		
Path to library	Path to precompiled c++ shared library.		The shared library must have been compiled before paraview execution.
Start date	Starting date for analysis	19000101	If start date des not belong to dataset, the nearest date will be taken.
End date	Ending date for analysis	22000101	If end date does not belong to dataset, the nearest date will be taken.
Coasts selection	Select coast level	coast and open sea	Any selection can be made. If both coast and open sea are selected, all the domain will be evaluated.
Basins selection	Select sub-basins	none	Any selection can be made. If more than one sub-basins is selected the aggregated domain will be evaluated.
Variable	Select variable for analysis	N3n	
Statistic type	Select statistic for analysis	Average	
Perc to visualize	Select desired statistics percentiles for analysis	0.05	Relevant only if Percentiles has been chosen in Statistic type. Admitted values are: 0.05, 0.25, 0.5, 0.75 and 0.95

OGS Time Statistic

The source produce an areal statistics of all timesteps, at a selected depth, between **Start date** and **End Date** properties. The output comes in a vtkTable and can be plotted with paraview's plot data filter. The source allows to obtain all the statistics for the desired timesteps and sea region, with the possibility to aggregate the results on different time periods, which are: All (no filter), Weekly, Monthly, Seasonly and Yearly.

Property	Description	Default Value(s)	Restrictions
Output	This property defines the output of the source.	vtkTable	
AVE FREQ path	Path to data folder		Must be the exact path to data.
MESHMASK path	Path to MESHMASK folder		MESHMASK folder must contains three sub-directories: low, mid and high with inside all the required files. See Setup section.
Mesh resolution	Select the correct mesh resolution among Low, Mid or High.	High	Correct resolution with respect to selected data must be choosen, otherwise prints error.
Path to python	Path to bit.sea folder		
Path to library	Path to precompiled c++ shared library.		The shared library must have been compiled before paraview execution.
Start date	Starting date for analysis	19000101	If start date does not belong to dataset, the nearest date will be taken.
End date	Ending date for analysis	22000101	If end date does not belong to dataset, the nearest date will be taken.
Coasts selection	Select coast level	coast and open sea	Any selection can be made. If both coast and open sea are selected, all the domain will be evaluated.
Basins selection	Select sub-basins	none	Any selection can be made. If more than one sub-basins is selected the aggregated domain will be evaluated.
Variables	Select variables for analysis separated by a comma.	N3n	
Periodicity	Select time aggregation	All	If original data has a lower periodicity than the selected one returns error.
Depth	Select depth level for analysis	200	
Statistic type	Select desired statistics	Average, Std, Minima, Percentiles, Maxima	
Dump to file	Saves statistic result to desired path		Resulting file has the following column order: Average, Std, Minima, Percentiles, Maxima.

OGS Parallel Time Statistic

The source is exactly the same as **OGS Time Statistic**, but it contains a parallel implementation for distributing data among more processes. It can be only exploited in the client-server configuration.

4.2.2 Filters

OGS Select Coast
The filter acts as a custom threshold and is able to visualize from the vtkRectilinearGrid produced by the **OGS Import NetCDF** filter only the selected coast region. The possibilities are: **coast**, which will show only near-coast regions, **open sea**, which instead will show only areas far from coast regions, or both, so that all the sea will be selected.

Property	Description	Default Value(s)	Restrictions
Input	Accepted filter input	vtkRectilinearGrid	
Output	Filter output	vtkUnstructuedGrid	
Coast selection	Select desired coast region.	**coast** and **open sea**	

OGS Select Basin
The filter is a custom threshold and is intended to act after the **OGS Select Coast** filter, in order to apply a further selection of the domain. The filter allows to visualize a domain section obtained from an aggregation of the selected sub-basins from the input.

Property	Description	Default Value(s)	Restrictions
Input	Accepted filter input	vtkUnstructuredGrid	
Output	Filter output	vtkUnstructuedGrid	
Basins selection	Select desired sub-basins.	All sub-basins	

OGS Stat Table

This filter receives the variables loaded from **OGS Import NetCDF** and evaluates their statistics on a selected domain. The analysis can be made both from areal or volume averages. In the former case, the output will have the same dimension as the original file depth levels, while in the latter case an additional information with the depth levels to be averaged must be provided. The output comes in a vtkTable variable and can be plotted directly in Paraview by using the provided macro PythonPlot.

Property	Description	Default Value(s)	Restrictions
Input	Accepted filter input	vtkRectilinearGrid	
Output	Filter output	vtkTable	
Path to library	Path to precompiled c++ shared library.		The shared library must have been compiled before Paraview execution.
Coast selection	Select desired coast region.	**coast** and **open sea**	
Basins selection	Select desired sub-basins.	None	
Statistic Type	Select between areal or volume statistic	Areal	
Statistic Options	Select statistic type	All	
percs	Select percentiles to be evaluated	0.05, 0.25, 0.5, 0.75, 0.95	This applies only if Percentily is selected in Statistic options.
depths	Select depth levels for volume analysis	200, 600, 6000	The first level goes fom 0 to the first element, the second from the first element to the second and so on.
Climatology	Adds climatology observation to the output	On	Only a few variables have climatological observables: N3n, N1p, O2o, N5s, O3h, Ac, O3c, DIC.
Dump to file	Saves statistic result to desired path		Resulting file has the following column order: Average, Std, Minima, Percentiles, Maxima.

OGS Custom Stat Table

This filter is analogous to **OGS Stat table**, but the input can also be a vtkUnstructuredGrid, as the one produced by any of the Paraview internal clip or threshold filters. This allows to performs statistics on any piece of domain obtained after a clip or a threshold, at the price of a slower speed with respect to the **OGS Stat table** filter.

Property	Description	Default Value(s)	Restrictions
Input	Accepted filter input	vtkUnstructuredGrid	
Output	Filter output	vtkTable	
Path to library	Path to precompiled c++ shared library.		The shared library must have been compiled before Paraview execution.
Statistic Type	Select between areal or volume statistic	Areal	
Statistic Options	Select statistic type	All	
percs	Select percentiles to be evaluated	0.05, 0.25, 0.5, 0.75, 0.95	This applies only if Percentiles is selected in Statistic options.
depths	Select depth levels for volume analysis	200, 600, 6000	The first level goes fom 0 to the first element, the second from the first element to the second and so on.
Dump to file	Saves statistic result to desired path		Resulting file has the following column order: Average, Std, Minima, Percentiles, Maxima.

OGS Okubo-Weiss
This filter allow to compute the Okubo-Weiss parameter in 2D geophysical turbulence according to Okubo 1970, Weiss 1991 and Isern-Fontanet 2004.
The Okubo-Weiss parameter is given by $W = S_n^2 + S_s^2 - w^2$, where S_n^2 is the normal strain squared, S_s^2 is the shear strain squared and w^2 is the vorticity squared.
The filter acts on the result of the "OGS Import NetCDF" filter and returns a vtkRectilinearGrid with two variables:

- OW: array containing the Okubo-Weiss parameter expanded in the third dimension.
- OWmask: mask to differentiate from the 3 different flows (W_0 is an input coefficient):
  - -1 for vorticity-dominated ($W < -W_0$)
  - 1 for strain-dominated ($W > W_0$)
  - 0 for background field ($|W| \leq W_0$)

Property	Description	Default Value(s)	Restrictions
Input	Accepted filter input	vtkRectilinearGrid	The input must be the output of OGS Import NetCDf and must contains velocity vectors
Output	Filter output	vtkRectilinearGrid	
var	Name of velocity vector as Paravire variable	"Velocity"	The valuemust match with the variable name from the input.
coef	Okubo-Weiss coefficient W_0	"0.2"	

OGS Select Okubo-Weiss
Filter intended to show only a region of the sea depending on the selected ranges of the Okubo-Weiss criterion. The filters takes as input a vtkRectilinearGrid produced by "OGS Okubo Weiss" filter and returns a vtkUnstructuredGrid.

Property	Description	Default Value(s)	Restrictions
Input	Accepted filter input	vtkRectilinearGrid	The input must be the output of OGS Okubo-Weiss.
Output	Filter output	vtkRectilinearGrid	
Vorticity Dominated	Selects only regions where $W < W_0$	"True"	
Strain Dominated	Selects only regions where $W > W_0$	"True"	
Background Field	Selects only regions where $\|W\| \leq W_0$	"True"	

5 Aknowledgements

The research reported in this work was supported by OGS and CINECA under HPC-TRES program award number 2016-02.
The developing work has been carried out by me, Cosimo Livi, and Arnau Mirò, whom I profundly thanks for the support and help he gave to me. Arnau works at Universitat Politècnica de Catalunya, and collaborated with me during his period spent at Cineca for the "Summer of HPC" project, thanks to a grant provided by PRACE. I have been mainly involved into the developing and benchmarking of the `OGSParaviewSuite` module and of the custom filters related to statistic analysis. I also worked on the set-up of the remote server for working in the client-server configuration. Arnau worked on the implementation of the Paraview Web service and on the production of the visualization and Okubo-Weiss filters. He also provided me some really time saving installation scripts.