Visualizing SWASH wave model results with PARAVIEW — PySWASH

Marco Miani, Marcel Zijlema

Abstract
This user manual presents and describes PySWASH: a python based package for visualizing and processing model-generated simulation results obtained with SWASH: an open source wave-flow numerical model. The great complexity of these data structures, reflecting the complexity of the physical principles behind simulating its flow, requires a specialized platform for not only effective visualization, but also for the actual analysis, calculation and extraction of crucial information, otherwise hardly accessible. The platform used for data analysis and visualization is the open-source PARAVIEW suite, by Kitware. In this context, PySWASH is used to convert model-generated data into a convenient netCDF format, open cross-platform and binary, that is ingested into PARAVIEW and visualized easily and effectively. Within the PARAVIEW environment, data analysis is remarkably simplified thanks to a state-of-the-art graphical user interface, as well as the possibility of coding tailored numerical solutions. Model results presented and processed here are both one- and two-dimensional, deriving from both layerized as well as depth-averaged simulations. The processing/transformation process is described in detail and the application for a few meaningful practical cases is presented and explained.

Keywords
SWASH — Python — visualization — PARAVIEW — post-processing

Contents

Version : 0.01 (March 18, 2021)
Reviewed : MM, MZ
External reviewer(s) : None

Glossary

nZeroPads

The formatting used to annotate layers in variable names, depends on the number of layers: a leading padding zero is always added. Within Python environment, this information is needed in advance, for proper formatting of strings (variable names to be invoked), or when defining search patterns.

E.g.,: `zk01_000215_300` vs `zk1_000215_300`

The amount of leading zeros, depends on `kmax`:

$$nZeroPads = len(str(kmax))$$

RunFolder

User-defined directory containing SWASH input/output files.

ncdump

Command-line utility used to convert netCDF data to human-readable text form. Requires installation of external libraries (e.g., **NCO**, **NCL**, [1]).

zk

Vertical coordinate of the layer interface, as output by SWASH. Here, $k \in [0,$ **kmax**$]$. The number of layer *interfaces* always equals the number of *layers* +1. Not to be confused with Zk (capital Z): the container for zT.

varnames

List containing model-output variables, i.e. the list containing all active and physical meaningful variables to be plotted in PARAVIEW, and computed by SWASH.

PV

The PARAVIEW open-source visualization software developed by KitWare.

layerized

A SWASH simulation for which vertical discretization (over **kmax** layers) is activated. Layerized runs are the opposite of depth-averaged runs.

zT

Post-processed vertical coordinate of the layer interface, as prescribed to PV. Here, $k \in [0.5,$ **kmax**$-0.5]$ or $k \in [0,$ **kmax**$]$, depending on the settings adopted. See text for more details. The *time-varying matrix* (time, k, x[,y]) containing all zk is named **Zk** in PySWASH. See figure 1.

Zk = `Zk(time, z, x [,y])`

1. Introduction and Motivation

The wide applicability of SWASH numerical wave model requires an effective visualization platform, able to cope with the most diverse situations, deriving, in turn, from the complexity of the data to be visualized.

The authors identified PV as the most responding to the mentioned requirements. Not only PV represents an excellent visualization platform, but is also does allow to carry out calculations, data extraction, slicing, interpolation, coordinates manipulation and, in general, any complex operation that would be too tedious to achieve inside a pure python environment.

Essentially, PySWASH is an intermediate processing tool designed to convert model-generated results (mat file) into a memory- and access-efficient binary format (netCDF), easily assimilated by PV and allowing its visualization therein. Ideally, this set-up should be replaced in future version by implementing a direct access from PV environment to mat files, rendering the Python-based intermediate layer unnecessary.This will improving general performance and enhancing usability.

1.1 Versions

At the date of publishing this document, the version used were:

Python	:	3.7.4
xarray	:	v0.16.1
SWASH	:	7.01
Paraview	:	5.8

1.2 The netCDF format

The Network Common Data Form, or netCDF, is machine-independent, array-oriented binary format, widely used in the realm of scientific data storing and visualization. This file format was developed by the *Unidata* project at the University Corporation for Atmospheric Research (UCAR).

It presents a typical layerized array-oriented structure, where data subsets, even the largest, can be easily and very quickly accessed by specifying the indices, and is:

- self-descriptive: its metadata and named attributes may contain information about the process that created that data-set;
- particularly suited for storing large data, as its metadata allow a rapid an comprehensive inspection of its content.
- cross-platform, non proprietary format and compatible with widely accepted standard in scientific data storing.

Data can be accessed trough a simple interface. One of the goals of netCDF is to support efficient access to small subsets of large data-sets. To support this goal, netCDF uses direct access rather than sequential access.

More specific and detailed information can be found at: [2, 3]. More material about the CF conventions[1] is available at [4].

1.2.1 The `ncdump` function

The ncdump command-line utility converts netCDF data to human-readable text form. When entered in Linux terminal[2], it returns the header information of the netcdf file queried. The returned list, includes a succinct description of enclosed variable and their dimensions, along with their metadata.

Following command, issued in the command line, saves the information returned from ncdump to a text file, in human readable format:

```
ncdump -h FileName.nc > out.txt
```

This will pipe all the information contained in the header, normally echoed on screen, to a plain text file. The header returns a synopsis of the file content.
However, printing netcdf headers is only meaningful for a first, superficial inspection. A more detailed and complete scan can be performed in Python when pre-processing the original data-set (with x-array: see [5]).

2. SWASH output structure

For some simple cases, a mere 1D depth-averaged flow simulation might already represent an fair approximation of the physical phenomenon to be investigated. However more complex cases might require the inclusion of additional spatial dimensions: lateral and vertical coordinate, that is y and z respectively. It is this important to understand the data structure, before addressing the pure conversion aspect. PySWASH cannot (yet) handle unstructured grids.

2.1 Model output and its topology

Generally speaking, each dimension (horizontal or vertical) included in the simulation requires a corresponding dimension when pre-allocating data within Python environment, and finally saving the netCDF file to disk. Typical dimensions are: horizontal coordinate(s), vertical coordinate and time[3]. Thus, for instance, the magnitude of flow velocity, might have as much as 4 dimensions, for the most complex configurations:

```
Vmag_k[time, z, y, x]
```

2.2 Preserving original topology

When carrying out layerized simulations, there are more layer interfaces than there are layers. Thus, from a formal point of view, pairing each layer with its corresponding layer interface, is an ambiguous formulation (in Python), as the value of the layer is not “cell centered” (i.e., not assigned at the center

[1] **C**limate and **F**orecast metadata
[2] Assuming the library is correctly installed
[3] However, more dimensions can be defined and used

of the layer, bounded between layer k and k+1) but rather prescribed at its edge. SWASH interprets this as the bounds where that layer is valid within. However, in PV this is not (yet) implemented and to each layer must correspond *one vertical coordinate*, instead of *two vertical bounds*. This poses a practical problem, addressed and described in section 3.2, and concerning the consistency of the vertical topology (grid unstaggering) and its physical meaning. See figure 1.

3. Structure and content of the package

This section describes the most significant functions involved in the generation process: the netCDF file. An example of its practical implementation is available at the end of this document (section 5) and the reader should confront to section to follow along each step. Following command install PySWASH in (mini)conda:

```
#not yet implem., copy & paste functions
#on the file header, on top of
#the main file, and run the py file instead

#not yet implemented:
?? conda install -c forge PySWASH #soon...
```

Virtual environment surely represent an ideal solution for case like this. The creation of a dedicated virtual environment is encouraged. In Anaconda Prompt Shell, type:

```
#Tested on 26.11.2020, on Windows, conda 4.9.2

#Create virtual Env., with bare minimum on it:
conda create -n PySWASH scipy xarray pandas
#or NameOfYourEnv instead of PySWASH

#Inside that Env., start building:
conda install -c conda-forge/label/gcc7 regex
conda install -c conda-forge r-sys
conda install -c jmcmurray json
conda install matplotlib
conda install -c jmcmurray os
```

The user may add all the convenient packages at their discretion. More details available at [6, 7, 8]. Once created, the virtual environment can be activated by typing:

```
conda activate PySWASH # or NameOfYourEnv
```

3.1 File structure and global variables

In principle, the only user input should the path pointing to the run folder, here named **`RunFolder`**. Starting from that element, a series of steps are initiated and all details are deducted and extracted from the files located in the RunFolder, resulting from SWASH simulation, and without any further user input.
As the name suggests, global variables are persistent. Any change of value to that variable, in any function, is visible to all the other functions throughout the package structure. For instance echo and plot flags are defined as global variables:

```
echo     = False  # print only minimal echo
plotFlag = False  # no plots, just mere calculations
```

3.2 Representation of variables

In depth-averaged mode, each flow variable is represented by *one* value, valid throughout the whole water column. This no longer holds for layerized runs where, instead, the flow variable is discretized and changes over the vertical coordinate: z.

The stacked layerized variable, needs to be accompanied by its corresponding stacked coordinate matrix; a matrix, consistent in size with the stacked variable, and representing the vertical levels (z) at which that variable is valid (see section 2.2).

To unstagger the vertical grid, that is to transform and make it consistent in size, following is implemented: for each pair of layer interfaces[4], middle heights are calculated. Formally:

$$zT_{k+0.5} = (z_k - z_{k+1})/2 + z_k, \quad k \in [0,\ kmax\text{-}1] \qquad (1)$$

A graphical representation of the above, is shown in figure 1.

As a result, the (newly calculated) zT matrix and the variable matrix assigned at those zT levels are now consistent is size (along z). There are, however, some considerations deriving from the method described in Eq. 1:

1. The upper and the lower "half layers", adjacent to surface and bottom respectively, are left out of the representation, as they are not filled with data.

2. The topology is not preserved, as the final vertical distance between bottom and top layer interfaces, was modified (after being recalculated).

3. The model-generated bottom layer (zk_{kmax}) corresponds to the model-prescribed bathymetry. Now, with the transformation described above, this stops from being true as the bottom corresponds with the newly defined interface: zT_{kmax}.

For all the reasons listed above, it was decided to include following corrective measure: the extreme layer interfaces, i.e. closest to free surface ($\mathrm{zT}_{k=0.5}$) and bottom ($\mathrm{zT}_{k=kmax-0.5}$), are imposed to coincide with corresponding model-output interfaces. This, in turn, implies:

- no more unpsychical fluctuation of model bathymetry (unlike it was before applying the final correction described here);

- topology is preserved, as vertical metrics is not distorted or altered (unlike it was before applying the final correction described here);

- the final number of layers is preserved and consistent with original simulation.

[4] here, k is pointing downwards, so the algorithm is moving from top to bottom

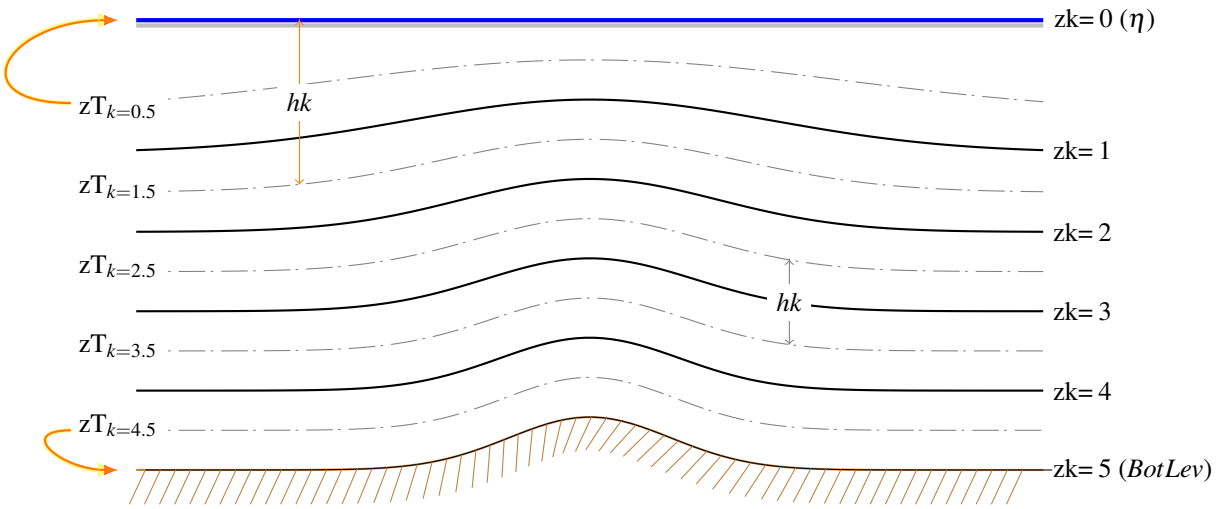

Figure 1. Treatment of layer interfaces, also known as terrain-following sigma levels, for *true* layer interfaces, zT, as prescribed in netCDF. These are derived from original interface levels, zk, generated by SWASH. This grid **un**staggering makes zT and levels consistent in size, allowing array pairing. In addition, top and bottom layers are explicitly assigned to free surface (η) and bottom level, respectively, as indicated by curly arrows. Space- and time-varying layer thickness, hk, is shown too.

3.3 Extraction of variable names

Variable names are *deduced* from the variable list contained in the matfile generated by SWASH, and are assumed to have the form:

```
# E.g., for time =0 minutes, 1.2 seconds:
# ---------------------------------------
qName_000001_200        # depth-averaged
qName_k03_000001_200    # layerized
```

After extracting all the variable names contained in the mat-file, the extraction (deduction) of the actual model variables consists of 3 sequential steps, that progressively filter out redundant information:

1. Remove, where present, the trailing timestamp (**`HHMMss_mms`**);
2. from layerized quantities, remove the layer number, for instance: **`Vmag_k05`** becomes **`Vmag_k`**;
3. From the result of steps 1 and 2, extract the unique elements composing that subset. That outcome is a list[5] containing all activate variables (**`varnames`**).

Steps 1 and 2, rely on **`regexp`** and the pattern depends on nZeroPads[6]. Following variables are systematically removed from variable list, as they should not be plotted: **`Time`**, **`Xp`**, **`Yp`**.

PySWASH is tested for mat files **`LAY 3`** (as per SWASH user manual). Other configurations are not tested.

[5] Precisely, a type **`set`**

[6] See glossary for more details

3.3.1 Minimal content

The "minimal content" refers to the least number of variables (not only flow-related) that should be included in the mat file, when requesting an output. These variables are:

- **`Xp`** - x coordinate (always needed)
- **`Yp`** - y coordinate (for 2D runs only)
- **`Botlev`** - the model bathymetry (always needed)
- **`zk`** - vertical coordinates (for layerized runs only)
- **`Time`** (always needed)
- **`Watlev`** (for depth-averaged runs only)

Their lack, will result in a computational halt. The function performing this check is:

```
Check_Min_Requirements(varnames, mat,...
                       Hdims, Nvert, Time)
```

3.4 Extraction of time-steps

The function **`get_Tsteps()`** governs the extraction of time information from the mat file. The variable name **`Time`**, is used to recursively find the pattern containing all timestamps. Should that variable miss, one other attempt is made using the first variable in **`varnames`**. However this variable should always be included in the output, and belongs to the minimal requirements (see Section 3.3.1). A list of strings is returned in the format **`HHmmSS_mms`**, used as input argument for generating the time stamps into a compatible format for PV.

3.5 Dictionary

PySWASH uses a tailored dictionary[7] to store and, when prompted, share information about each model output variable. This information is used to control pre-allocation, storing and manipulation. The dictionary path is saved as a global variable (see section 3.1) **and should be set before running the package**. It is important to note that if the dictionary is not properly set up and accessible, no netCDF file can be produced.
The dictionary *path* can be set in following function:

```
Initiate_from_Run_Folder
```

The python command:

```
inspect_dict()
```

provides basic instruction about dictionary structure, usage and access (printing path on screen) as well as showing its last modification date.

A misspelled term in the dictionary may cause malfunctions or run-time errors; it is therefore advised to edit the dictionary with care, respecting syntax (parenthesis, commas, indentation) and structure (number of fields). Dictionary content is case sensitive.

3.6 Array pre-allocation

Instead of importing variables manually, PySWASH scans **`varnames`** and, recursively, pre-allocates[8] each variable. When doing so, each variable name is tested against the dictionary to extract and append its metadata. Pre-allocation clearly depends on number of time-steps, grid-points and vertical levels. The final netCDF file, saved to disk, will finally contain the same (number of) variables contained in **`varnames`**.

3.7 Variable assignments

For each time-step and for each variable, pre-allocated matrices are populated with numbers, based on following conditions:

- **only if** the variable name is found in the dictionary it is included, and discarded otherwise.
- if the variable name equals **`zk`**, each time-step (or each "data layer") is saved on a dedicated variable used later by the programmable filter in PV (see sections 2.2 and 4.3) to reconstruct time-varying vertical fluctuation of the mesh.
- if (for layerized results) a depth-averaged variable is included, a filling method is applied to control its filling and thus the way it will be later visualized in PV. The type of method employed depends on the settings specified in the dictionary. The filling method can be either:
  - **`bottom`**: populate only bottom line of the array, and leave NaN elsewhere (like for **`BotLev`**);
  - **`surf`**: populate only top line of the array, and leave NaN elsewhere (like for **`WatLev`**).;
  - **`fill`**: populate all lines, by tiling the one line over the whole array (like for **`Depth`**).

The filling method is used as a mere artefact to visualize depth averaged variables in a context that was in fact specifically designed for quantities presenting vertical discretization: more details can be found in section 4.2.

Note how **`BotLev`**, a stationary field, is included in a time varying loop. This is, again, a pure artefact as the same, invariant, structure is prescribed for this variable consistenyl over time. This was done to avoid conflicts with PV's programmable filter, which is not (yet) accepting stationary fields.

3.8 Appended metadata

Metadata is assigned to both individual variables (variable metadata) and main root structure (global attributes). Individual variables include: unit, long-name, axis description, et al. Global attributes include information about platform on which the package was executed on, the **`*.sws`** file*name*, the run name and number contained in it, and the RunFolder name, appending some basic information about the instructions that used to generate the data being saved on netCDF.

The command:

```
ncdump -h TheFileName.nc
```

can be typed in command window to inspect them.

3.9 The programmable filter (PV)

Amongst the metadata fields appended to the final netcdf file (global attributes), the user will find:

```
coordinates:   yc xc
```

This reserved field contains the (SWASH generated) problem coordinates, and is used by PV to manipulate the topology, and remap the (curvilinear) grid. Regrettably, however, *this variable is not (yet?) allowing time-varying allocation* and would thus only serve for the first time step or, in other words, would not allow temporal evolution of the domain topology.

To overcome this problem, the time-varying grid coordinates were stored as a separate variable (zk). This information if then fed to the programmable filter that perform the remapping.

Possibly, future version might allow time-dependent allocation, making the filter obsolete, and remarkably simplifying the assimilation process within PV. At the time of writing version 0.01 of this manual, this issue was also brought up to the developing community. This, along with the improper axis assimilation/recognition (x, y instead of x, z plane) culminated

[7] format of the dictionary: **`json`**. Editable with any text editor.
[8] NaN arrays

in tracked issue **18035** on gitlab.kitware tracking system.

The actual content of the filter is available in section 4.3.

3.10 Python utilities

3.10.1 Creation of initial fields (only layerized mode)

Together with bathymetry creation (section 3.10.2) this script, only available for 1DV runs, serves as facility for setting up basic one-dimensional SWASH runs. For each vertical layer, this python script assigns a value for either sediment, salinity or temperature, depending on (user-defined) horizontal position. Thus, for instance, the initial spatial structure of salinity can be defined based on top and bottom bounding layer, **sal1** and **sal2**, between **Xp1** and **Xp2** coordinates[9].

Nested in the **RunFolder**, subfolders are created[10] (with according name), each containing **kmax** files (i.e. 1 per layer). In addition to that, the filelist required for the ***.sws** file, and containing all initial file**names** over all **k** levels, is created too.

3.10.2 Creation of 1D bathymetry

When testing, a quick tool for creating **1D** bathymetry, might be convenient. In that spirit, a python-coded function is available for achieving that:

1. The user can set region **Lx**, and **nx**;
2. the array, of arbitrary length, containing x, z pairs, can be explicitly prescribed;
3. consecutive x, z are joined, so to draw a continuous uninterrupted line: $z(x)$, $x \in [0, \mathbf{Lx}]$;
4. the resulting plot, is saved as a figure in **RunFolder**, including partial slopes along the section. This can be used for future references or quick inspection.

An example of that, is available in the attached annex. The python script in question is, generating that image is:

```
Create_1D_bathy.py
```

3.10.3 Inspection of initial conditions

In the main routine, the user can request the graphical output for the initial condition of any of the available quantities contained in **varnames**:

```
showPlot    = False
show_zLines = True
```

The second flag, includes layer interfaces in the plot. This is used to quickly inspect model output, at $t = t_0$.

3.10.4 Extraction of vertical profiles from 2DV runs

This script serves to extract vertical profiles (for layerized runs) at any given grid point. This script still requires to be completed, and is not fully tested yet - use with care!

[9] Note: $\min(\mathbf{Xp}) < \mathbf{Xp1}, \mathbf{Xp2} < \max(\mathbf{Xp})$

[10] if not existing already

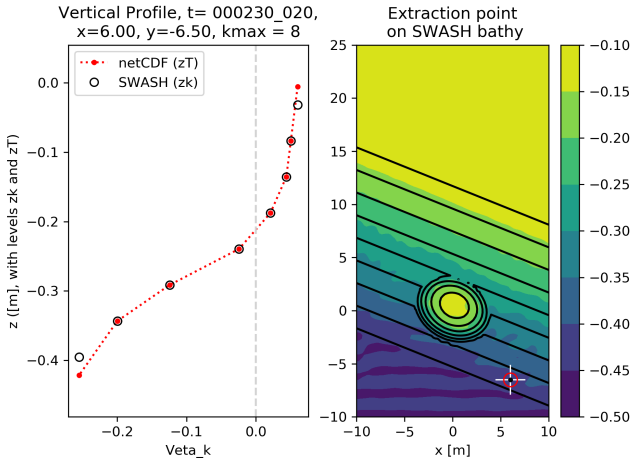

Figure 2. Extraction of vertical Profile for `l41berkh` testcase, here run with 8 vertical layers. The location of the extraction point is shown, too. An inverted vertical velocity profile, also known as *undertow*, is clearly visible. The height of first and last vertical level are not matching: see Figure 1.

4. Visualizing SWASH results with PARAVIEW

4.1 Strengths and advantages

The extraction of crucial information form complex model-generated results typically requires tedious coding. Also, special effort might have to be invested when trying to keep the code versatile for different runs (varying model configurations). Instead, visualizing model results with PV not only permits to overcome this bottleneck but, most importantly, helps getting a better, faster and clearer insight of the content of the data-set being inspected.

PV permits one-click data extraction, immediate visualization, and makes visual comparison effortless. The vast plethora of in-built functions and functionalities allows the user, even the less experienced, to intuitively achieve very good results, by investing modest effort.

In addition to this, large model results can be still read in at an acceptable speed, as PV accesses the netCDF file directly, instead of sequentially.

When animating time-varying results, the timing of labels, titles, or features intrinsically hidden in the results, can be easily controlled and managed. PV hosts a Python shell facility that permits running (self-developed) Python code to be embedded in the animation, to improve it significantly.

Another remarkable feature is certainly the programmable filter, enabling the implementation of tailored functionalities. This platform was actually used when developing this toolbox and is briefly described in section 4.3.

Of marginal importance, some remarks concerning general appearance such as: logos and background images, or the possibility of embedding animations into geo-referenced images or geographical databases to increase realism.

4.2 Importing data in PARAVIEW

At this point, the netCDF file can be imported in PV for being visualized and post-processed:

1. FILE menu → Open file. PV has 3 types of netCDF readers. Choose "*NetCDF Reader*" in the pop-up menu (2^{nd} option).

2. **Deselect** spherical coordinates, and apply. You should now see the data appearing on RenderView. In addition, the data should be defined over a x,y plane. This is clearly wrong and the tailored filter, described in section 4.3, will restore the right topology.

3. Make sure your data-set is selected in PV's pipeline browser (it is highlighted in blue): that same set will be the one undergoing the actual filtering.
   Open a *Programmable Filter*.

4. Copy and paste the content of the programmable filter enclosed in this document (the actual code), into the *Script* window of PV's filter. Leave all settings and check boxes untouched.

5. If you are dealing with two-dimensional data, comment out the line where, in the for loop, `y = 0`:

   ```
   # y=0 #comment this line for 2D runs!
   ```

   Remember to apply changes.

6. After applying, the data will disappear (if 1D data-sets, or *reshape* if 2D data-sets). This is because axis were swapped: one-dimensional data have no extension along y axis, and are thus invisible. Re-orient your axes (for 1D data-sets: choose x,z instead of x,y). At this time, your data should be visible again. If so, orientation and axis labels are now consistent. Indexing and scaling can now follow the conventional x,y,z metric.

7. If depth-averaged variables were included in the netCDF file, and if being actively visualized, the user should **set NaN opacity to zero**, using the slider in the colorbar section. This will filter out empty value, leaving the user with the only visible (valid) portion of the data.

Import process is now completed. Future versions aim at skipping this step and, instead, allocating this conversion inside the netCDF itself (see Fig. 3).

4.3 A simple programmable filter

For the reasons described in section 3.2 and 3.9, a programmable filter was developed: its practical usage is described in section 4.2 (steps 3 to 7).

First, input and output objects are created. The latter is a copy of the former meaning that, at this stage, data are still unprocessed.

Then, the variable named **`zk`** is extracted from the input (original) data-set and, for each grid point in the domain, all 3 spatial coordinates are extracted.

At this point, a for loop cycles trough all grid points: here, y and z coordinates are swapped (plane tilting) and the data contained in **`zk`** is imposed on the original (swapped) z and finally shipped to the output object.

The so-obtained newly calculated points, have now the correct axis orientation and layer interfaces are now at the correct height. **This procedure is repeated for each time-step available in the netCDF file**. If the *Output Message* facility is activated, each time-step is printed out.

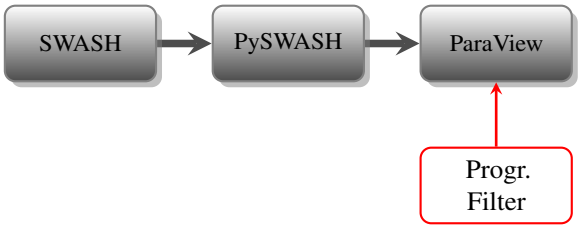

Figure 3. Conceptual architecture of PySWASH and role of the programmable filter used in PV. This figure clearly illustrates how the filter is outside the native flow. Future versions will work their way away from this scheme, making the filter unnecessary.

5. Limitations

Presently, PySWASH cannot process unstructured meshes (triangles). Dictionary should be updated by the user, depending on their specific needs, and preserving the structure.

Acknowledgments

So long, and thanks for the fish...

This document was produced using LaTeX, on Overleaf.

References

[1] NCAR. Ncar command language (ncl). https://www.ncl.ucar.edu/index.shtml. (Accessed on 11/26/2020).

[2] Netcdf: The netcdf-c tutorial. https://www.unidata.ucar.edu/software/netcdf/docs/tutorial_8dox.html#sec_tut. (Accessed on 11/15/2020).

[3] Netcdf: Introduction and overview. https://www.unidata.ucar.edu/software/netcdf/docs/index.html. (Accessed on 11/15/2020).

[4] Netcdf climate and forecast (CF) metadata conventions. http://cfconventions.org/cf-conventions/cf-conventions.html. (Accessed on 11/15/2020).

[5] Python xarray. xarray.Dataset.info — xarray 0.16.2.dev3+g18a59a6d.d20200920 documentation. http://xarray.pydata.org/en/stable/generated/xarray.Dataset.info.html. (Accessed on 11/29/2020).

[6] Conda official web page. Managing environments — conda. https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html. (Accessed on 11/22/2020).

[7] Conda official web page. Conda environments — condan. https://docs.conda.io/projects/conda/en/latest/user-guide/concepts/environments.html. (Accessed on 11/22/2020).

[8] Conda. Cheat sheet — conda 4.9.2.post2+b8c0efc1 documentation. https://docs.conda.io/projects/conda/en/latest/user-guide/cheatsheet.html. (Accessed on 11/26/2020).

.

License

Various supporting material

Input file

```
#########################

global echo, timing
echo    = False
timing  = True

showPlot    = False
show_zLines = True
LenghtBar   = 30


RunFolder   = r'C:\Users\swash\Run\l12bbbar'
Initiate_from_Run_Folder(RunFolder)

matFileName      = get_matFiles(RunFolder)
swsFile      = get_swsFiles(RunFolder)
print("")

if not swsFile:
    Nvert    = Prompt_VERT()
else:
    Nvert    = Get_VERT(swsFile)

nZeroPads               = ZeroPads(Nvert)
Hdims       = Get_dims_from_sws(swsFile)

mat         = sp.io.loadmat(matFileName)
Xp, Yp, botlev, Time    = get_static_variables(mat, matFileName)
varnames, rawList       = getQnames(mat,nZeroPads)
t                       = get_Tsteps(rawList, varnames)

Print_Summary()

xk, Zk = getVertStructure(mat, t, Nvert, Xp, nZeroPads, Hdims)
if Nvert<1:
    varnames.add('zk')

arrPre, Realnames       = MergeAllVars_1DV(varnames, t, Zk, Xp, swsFile)
arr = assiging_eachTstep_eachRealVar(arrPre,Realnames,t, Zk)

save_netCDF(arr, [], matFileName)
```

Figure 4. Example of input file

Programmable filter

```
############################################################
# Part of this code can be used to simply generate a copy
# of the original data set, so to visualize additional variables
############################################################
#                                              \__/
#                                                |
#                                                |
 input = self.GetInputDataObject(0, 0)#         |
output = self.GetOutputDataObject(0)  #_________|
output.ShallowCopy( input)            #


newPoints = vtk.vtkPoints()
numPoints =  input.GetNumberOfPoints()

t =self.GetInputDataObject(0,0).GetInformation().Get(vtk.vtkDataObject.DATA_TIME_STEP())
print("time:",t)


q = "zk"
input0 = inputs[0]
zlev = input0.PointData[q]
print(zlev)



#foreach gridpoint in my domain:
for i in  range(0, numPoints):
    #extract coordinates
    coord =  input.GetPoint(i)
    x, y, z = coord[:3]
    y = 0   # <--------------------  comment this line for 2D runs!
    z = zlev[i]

    #insert
    newPoints.InsertPoint(i, x, y, z)


#impose new points (modified!)
output.SetPoints(newPoints)
```

Figure 5. Content of the programmable filter to be used in PV. The fraction of highlighted can be used, as a standalone, to generate a convenient copy of input data-set, so to visualize several variables at the same time during the animation. To do that, copy and paste that sole fraction of code into a *new* programmable filter. The marked line, should be deactivted (commented) when applying the filters to 2D runs.

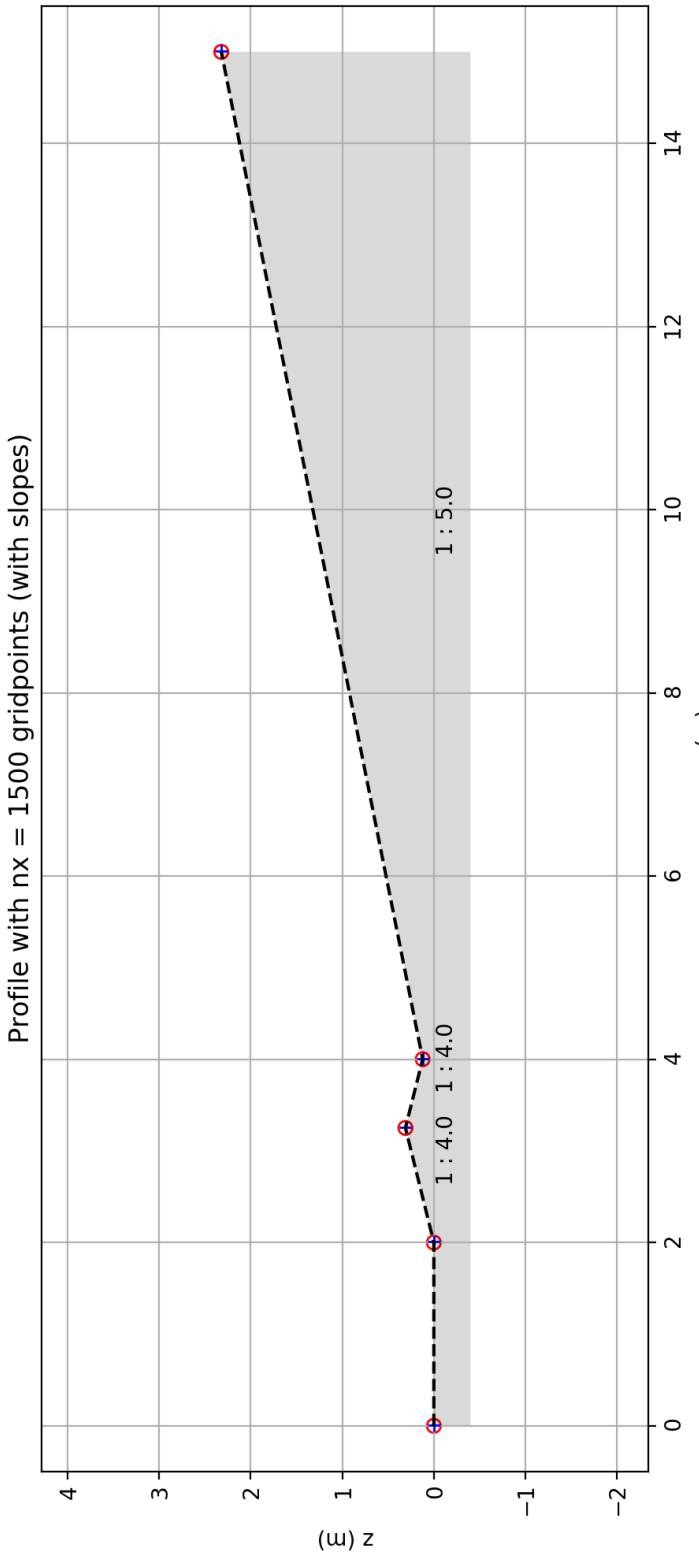

Figure 6. Example of bathymetry generated in Python, for a 1D test run

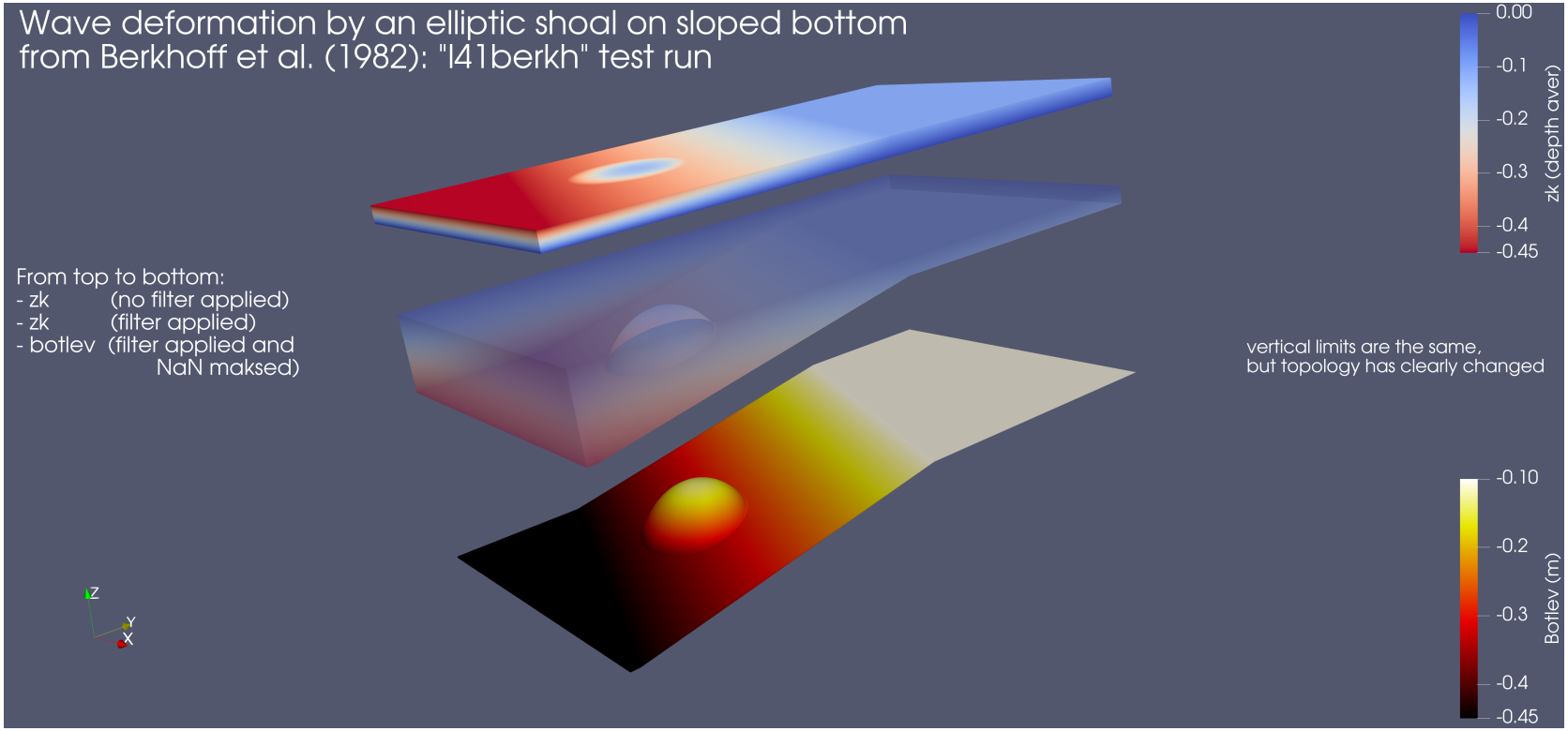

Figure 7. The effect of the programmable filter and of how it grabs the variable ”zk” and applies it to all other quantities, by altering the topography, without changing their numerical content. Displayed here, the bathymetry used for testrun l41berkh.