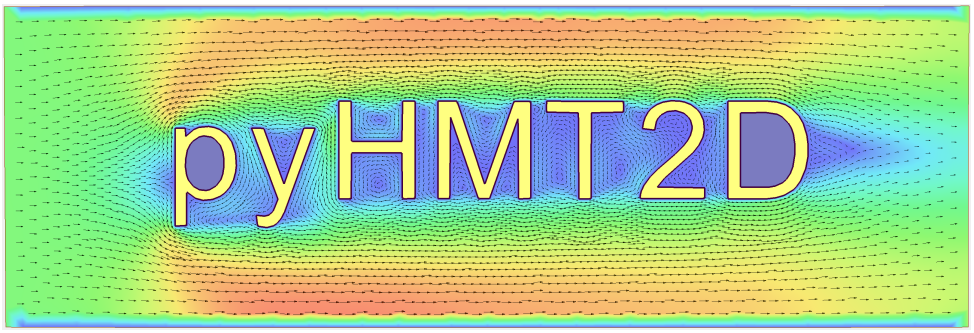
pyHMT2D v1.0: Two-Dimensional Hydraulic Modeling Tools in Python



https://github.com/psu-efd/pyHMT2D

Xiaofeng Liu, Ph.D., P.E., Associate Professor

Department of Civil and Environmental Engineering
Institute of Computational and Data Sciences
Pennsylvania State University

Email: xzl123@psu.edu
Web: http://water.engr.psu.edu/liu

March 11, 2024

Contents

Chapter 1

Introduction

1.1 What is *pyHMT2D*

pyHMT2D stands for Python Hydraulic Modeling Tools-2D. It is a Python package developed to control and (semi)automate 2D hydraulic modeling, and pre-/postprocessing simulation results. Currently, the following 2D hydraulic models are supported:

- SRH-2D
- HEC-RAS 2D

Despite the "2D" in the name *pyHMT2D*, this tool package can handle some "1D" functionality in hydraulic models, such as HEC-RAS. For example, *pyHMT2D* can control the run of a HEC-RAS case regardless it has 1D channels or not.

1.2 Motivations

Two-dimensional (2D) hydraulic modeling, replacing one-dimensional (1D) modeling, has become the work horse for most engineering purposes in practice. Many agencies, such as U.S. DOT, Bureau of Reclamation (USBR), FEMA, and U.S. Army Corp of Engineers (USACE), have developed and promoted 2D hydraulic models to fulfill their respective missions. Example 2D models are SRH-2D (USBR) and HEC-RAS 2D (USACE). The motivations of this package are as follows:

- One major motivation of this package is to efficiently and automatically run 2D hydraulic modeling simulations, for example, batch simulations to intelligently and efficiently calibrate models. Many of the 2D models have some automation to certain degree. However, these models and their GUIs are closed source. Therefore, a modeler is limited to what he/she can do.
- Most 2D models have good user interface and they have capability to produce good result visualizations and analysis. However, with this package and the power of the VTK library, 2D hydraulic modeling results can be visualized and analyzed with more flexibilty and efficiency.

- This package also serves as a bridge between 2D hydraulic models and the Python universe where many powerful libaries exist, for example statistics, machine learning, GIS, and parallel computing.
- The read/write and tranformation of 2D hydraulic model results can be used to feed other models which use the simulated flow field, for example external water quality models and fish models.
- Model inter-comparison and evaluation. Almost all 2D hydraulic models solve the shallow-water equations. However, every model does it differently. How these differences manifest in their results and how to quantify/interpret the differences are of great interest to practitioners.

1.3 Features and capabilities

The features and capabilities of *pyHMT2D* are briefly described in this section. More features will be added in the future. Application examples are provided with the package and some details will be provided in the following chapters.

For SRH-2D modeling:

- read SRH-2D results
- convert SRH-2D results to VTK format (if they are not already in VTK format)
- sample and probe simulation results (with the functionality of VTK library)
- control and automate SRH-2D simulations

For HEC-RAS 2D modeling:

- read RAS 2D results (HDF files and other case specification files)
- convert RAS 2D results to VTK, one of the most popular format for scientific data
  - point and cell center data (depht, water surface elevation, velocity, etc.)
  - interpolate between point and cell data
  - face data (e.g., subterrain data)
- convert RAS 2D mesh, boundary conditions, and Manning's n data into SRH-2D format such that SRH-2D and HEC-RAS 2D can run a case with exactly the same mesh for comparison purpose. Consequently, HEC-RAS 2D can be used as a mesh generator for SRH-2D.

- sample and probe simulation results (with the functionalities of VTK library)
- control and automate HEC-RAS 2D simulations

With the control and automation capability in *pyHMT2D*, it is much easier to do the following:

- automatic calibration of 2D models with any available optimization and calibration Python packages. Currently *pyHMT2D* supports *Scipy*'s optimization module, which includes many local and global optimization methods. In the future, other optimizers will be added.
- user can define any calibration/optimization objective function (e.g., water level, velocity, inundation area, flow split ratio, bed shear stress, recirculation zone size/shape, force on structures, flow hydrograph, etc, and/or their combinations).
- Monte-Carlo simulations with scripting and Python's statistic libraries
- ...

Other features of interests:

- with the unified result format of VTK, calculate the difference between simulation results, regardless they are on the same mesh or not.
- create rating curves for a boundary, which can be used in SRH-2D.
- extrude 2D model mesh to 3D (with either one layer or multiple layers) and map 2D result to the 3D mesh. This is useful if the 2D flow field is to be used in a 3D model application, or used as initial condition for 3D CFD simulations.

1.4 Limitations

As all other software packages, *pyHMT2D* also has its share of limitations. For example, the automation claimed in this document is only in the relative, not absolute, sense. Some manual modifications are still needed. Other significant limitations for each of the supported 2D models are listed below.

For SRH-2D:

- This package is developed and tested with SRH-2D v3.3; other versions may work but has not been tested.

For HEC-RAS 2D:

- Certain functionality in *pyHMT2D* can only deal with one 2D flow area.
- Only 2D flow area information is processed; others such as 1D channels and structures are not read from RAS 2D results. Addition of processing 1D data is doable in the framework of *pyHMT2D*.
- Only flow data is processed; others such as sediment and water quality are ignored. In the future, processing of other result data can be added.
- This package is developed and tested with HEC-RAS v5.0.7; other earlier versions may work but have not been tested. The latest HEC-RAS v6 beta versions are currently not supported because the result file does not contain some of the variables as described in its user manual.

These limitations exist only when HEC-RAS mesh and results are read and parsed. They do not exist if *pyHMT2D* is used to control the simulations.

1.5 License

pyHMT2D is released under the MIT license. Details about MIT license can be found at https://opensource.org/licenses/MIT. If you need other license, please contact Dr. Xiaofeng Liu.

1.6 Contributor Agreement

First of all, thanks for your interest in contributing to *pyHMT2D*. Collectively, we can make *pyHMT2D* more powerful, better, and easier to use.

Because of legal reasons and like many successful open source projects, contributors have to sign a "Contributor License Agreement" to grant their rights to "Us". See details of the agreement on GitHub. The signing of the agreement is automatic when a pull request is issued.

If you are just a user of *pyHMT2D*, the contributor agreement is irrelevant.

Chapter 2

Installation

2.1 Preparation of Python environment

pyHMT2D is a Python package which needs a proper Python environment and its dependency packages. *pyHMT2D* is developed with Python 3.7. The steps in this section is optional if you already have Python installed and you are familiar with Python package installation. If that is the case, you can skip to Section 2.2.

2.1.1 Installation of Python

There are several ways to install Python. For example, you can install the latest version of Anaconda Python from https://www.anaconda.com/products/. During the writing of this manual, the latest version is Anaconda3-2020.11. By default, it uses Python 3.8. But we can create a Python 3.7 virtual environment (see Section A.1 for more details).

The concept of virtual environment in Python is mainly to avoid package conflicts. There are large amount of Python packages (*pyHMT2D* is one of them) which have their own dependency and version requirements. It is very easy to induce conflicts. To solve this problem, we can create virtual environments to isolate from each other and be used for different purposes.

You can have multiple virtual environments on your machine. Here we create a virtual environment with Python 3.7. To create virtual environment, you can use different tools, such as `virtualenv`, `venv`, or `conda`. Here we use `conda` because we use Anaconda Python.

Basic steps and commands for virtual environment creation are as follows. We use Windows operating system as an example. *pyHMT2D* can be used in Linux and Mac OS.

- Open an "Anaconda Prompt (anaconda3)" terminal on Windows through "start->Anaconda3 (64-bit)->Anaconda Prompt (anaconda3)". In this command window, all proper Python and Anaconda environment variables are set properly and you can type your commands.

- To check the list of available Python versions:

```
$ conda search python
```

- Create a new virtual environment. E.g., the following creates a virtual environment called "py37".

  ```
  $ conda create -n py37 python=3.7 anaconda
  ```

- To check and view a list of created virtual environments:

  ```
  $ conda info -e
  ```

  You will see that in addition to the created virtual environment, there is also a default environment called "base".

- To activate the created virtual environment:

  ```
  $ conda activate py37
  ```

- To deactivate the current virtual environment and go back to the "base" environment:

  ```
  $ conda deactivate
  ```

- To delete a virtual environment (do not do this if you still need it in the future):

  ```
  $ conda remove -n py37 -all
  ```

2.1.2 Installation of dependent packages

Some of the dependent packages are only used when certain functionalities of *pyHMT2D* are called. If you do not use these functionalities, the dependent packages are optional.

pyHMT2D depends on the following packages (those with * are required):

- h5py*: for HDF file I/O.
- vtk*: for vtk file format.
- pywin32: for interacting with HEC-RAS through COM.
- gdal: for spatial data processing and artificial bathymetry creation.

- affine: for affine transformation of georeferenced system.

For *pyHMT2D*, make sure that the newly created virtual environment "py37" is activated and do everything within. If you are new to Python package installation, a quick tutorial can be found at https://packaging.python.org/tutorials/installing-packages/.

You can install the dependent packages individually. In the following, `pip` is used to install the dependent packages. Activate your new virtual environment "py37" and do everything within this virtual environment.

- `h5py`: https://pypi.org/project/h5py/

  ```
  $ pip install h5py
  ```

- `vtk`: https://pypi.org/project/vtk/

  ```
  $ pip install vtk
  ```

- `pywin32` (only if you want to run HEC-RAS through *pyHMT2D*):
  https://pypi.org/project/pywin32/

  ```
  $ pip install pywin32
  ```

- `affine`: https://pypi.org/project/affine/

  ```
  $ pip install affine
  ```

- `GDAL` (Only needed if you use "Terrain", "RAS_2D_Data", their related classes and tools, and other georeferencing functions):
  https://pypi.org/project/GDAL/.

  Read their instructions carefully. For example, on Windows, you need to install "GDAL Windows binaries" first. The Python "GDAL" package is only a binding/wrapper to the real "GDAL" binaries.

  ```
  $ pip install GDAL
  ```

  If there are errors reported in the installation of `GDAL`, you can use some pre-compiled wheels (a metaphorical name for Python package). For example on this website https://www.lfd.uci.edu/ gohlke/pythonlibs, you can download a proper version of `gdal`, then

  ```
  $ pip install GDAL-3.2.2-cp37-cp37m-win_amd64.whl
  ```

After `GDAL` is installed, it is important to add set its environment variable. Otherwise, `GDAL` may not function properly. The path to the `GDAL` package on your computer can be found using

```
$ pip show gdal
```

In fact, `GDAL` is installed under the name of `osgeo` and on the author's computer it is installed in:

```
c:\users\username\appdata\roaming\python\python37\site-packages
```

Thus, to set `GDAL`'s environment variable, in a Windows command terminal, I used the following setx commands:

```
$ setx GDAL_DATA 'c:\users\username\appdata\roaming\python
                  \python37\site-packages\osgeo\gdal-data'
$ setx GDAL_DRIVER_PATH 'c:\users\username\appdata\roaming\python
                  \python37\site-packages\osgeo\gdalplugins'
$ setx PROJ_LIB 'c:\users\username\appdata\roaming\python
                  \python37\site-packages\osgeo\projlib'
$ setx PYTHONPATH 'c:\users\username\appdata\roaming\python
                  \python37\site-packages\osgeo\'
```

2.2 Installation of *pyHMT2D*

There are several ways to install *pyHMT2D* on your computer. They are:

- Install from `pip`,
- Install from GitHub,
- Directly clone or download *pyHMT2D* code into your own computer from GitHub.

2.2.1 Installation from `pip`

pyHMT2D is available from the Python Package Index. To install, activate the desired virtual environment (if you have not done so) and simply do:

```
$ pip install pyHMT2D
```

After installation, you can check where *pyHMT2D* is installed and whether it is installed properly by

```
$ pip show pyHMT2D
```

The installation should be in the virtual environment, e.g.,
"`anaconda3\envs\py37\lib\site-packages`".

2.2.2 Installation from GitHub

You can directly install *pyHMT2D* from its repository on GitHub. One reason to do this could be that the latest version may have not been uploaded to pip yet or you want a specific old version (commit) on GitHub. Use the following command

```
$ pip install git+https://github.com/psu-efd/pyHMT2D.git
```

The installation location should be the same as in the previous section. For more information, you can search "git install from github".

2.2.3 Directly clone or download *pyHMT2D* code into your own computer from GitHub

To do this, you can use the "`git clone`" command or download a zip file from GitHub.

To clone the *pyHMT2D* repository from GitHub,

```
$ git clone https://github.com/psu-efd/pyHMT2D.git
```

or

```
$ git clone git@github.com:psu-efd/pyHMT2D.git
```

Note that the first one uses HTTPS and the second uses SSH. After cloning the repository, you should have "pyHMT2D" in your current directory.

If you want to directly download a zip file, you can go to *pyHMT2D*'s GitHub repository and click "Clone" and then choose "Download ZIP" (see Figure 2.1). Unzip the folder into your desired location.

To use the *pyHMT2D* package that you just cloned or downloaded, you need to let the Python Interpreter know where to find it. Python interpreter uses the "PYTHONPATH" environment variable to search for modules when it sees the "`import`" command in your Python code. Thus, you need to add the location of *pyHMT2D* to "PYTHONPATH".

Please note that if you also installed *pyHMT2D* before with `pip` or `git install`, you should make sure that you use the version that you want. The version used by Python Interpreter is the one it finds first for all the paths and in their order listed the "PYTHONPATH" environment variable.

How to modify the "PYTHONPATH" environment variable depends on your system and how you want to use *pyHMT2D*. Most commonly you will use it on

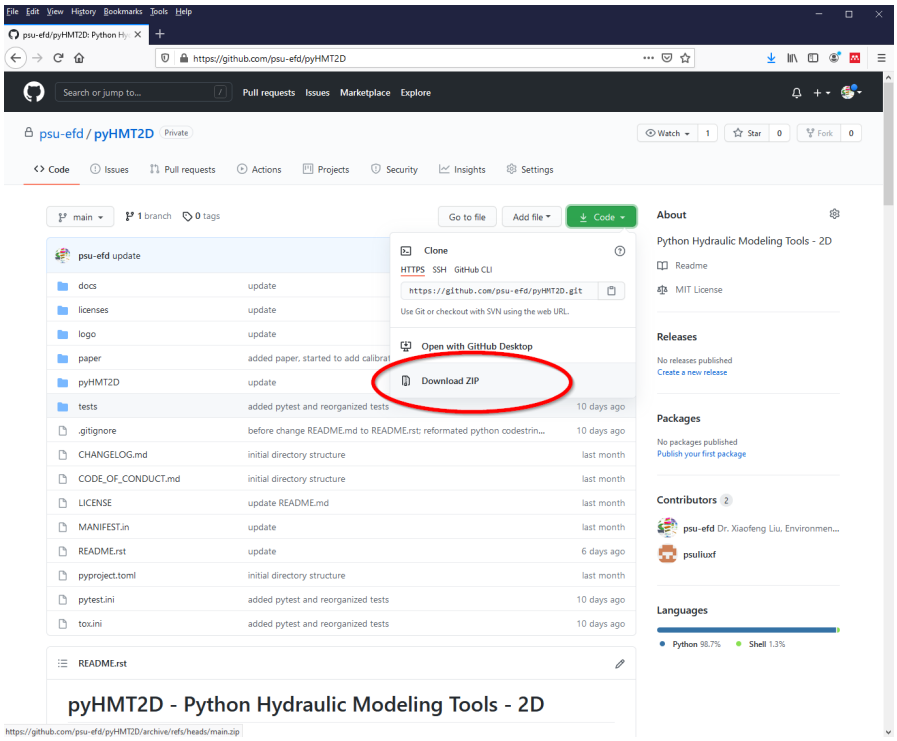


Figure 2.1 Download ZIP from GitHub.

Windows machines. You can change the "PYTHONPATH" environment variable through command line, PyCharm, in your Python code, or in a Jupyter Notebook.

Use *pyHMT2D* in a terminal

If you run your Python code in Window command line (terminal), you can do the following:

```
$ set PYTHONPATH=/path/to/pyHMT2D;%PYTHONPATH%
```

which will put your "path/to/pyHMT2D" on the top of "PYTHONPATH". To verify, you can use

```
$ echo %PYTHONPATH%
```

to check the content of "PYTHONPATH". Indeed, you can check all environment variables by typing

```
$ set
```

Use *pyHMT2D* in PyCharm

You can use different Python Integrated Development Editors (IDEs) to work on textitpyHMT2D's source code and run your own Python scripts. PyCharm is one of popular Python IDEs. You can obtain PyCharm from:

https://www.jetbrains.com/pycharm/download/

Launch PyCharm, click "File->Open", navigate to *pyHMT2D*'s directory and then click "OK". Note that you should select the *pyHMT2D*'s directory itself, not any files within. By default, PyCharm creates a project for *pyHMT2D*. You should see all the content of the project as in Figure 2.2.

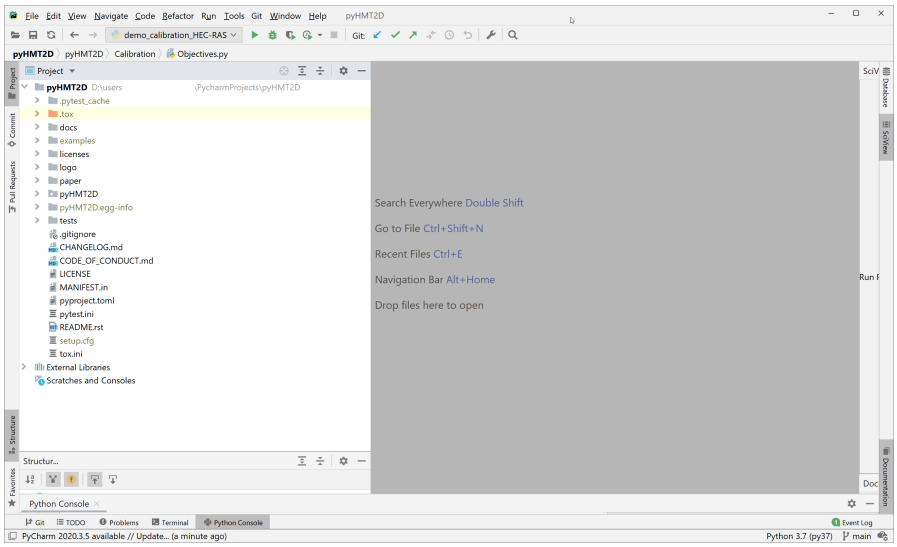


Figure 2.2 *pyHMT2D* project in PyCharm.

Next, we need to set PyCharm so it can use the virtual environment "`py37`" that we created. The steps are as follows:

- "File->Settings ...". In the pop-up window, select "Project: pyHMT2D->Python Interpreter".
- By default, PyCharm uses the default Python environment. Here we need to change it to "`py37`". On the right, click "Python Interpeter" and then "Show All ...". Select the "`py37`" environment.
- Once you selected the "`py37`" environment, PyCharm will list all the packages installed in this environment. You can have a look at whether *pyHMT2D*'s dependent packages are installed (see Figure 2.3).
- Click "OK".

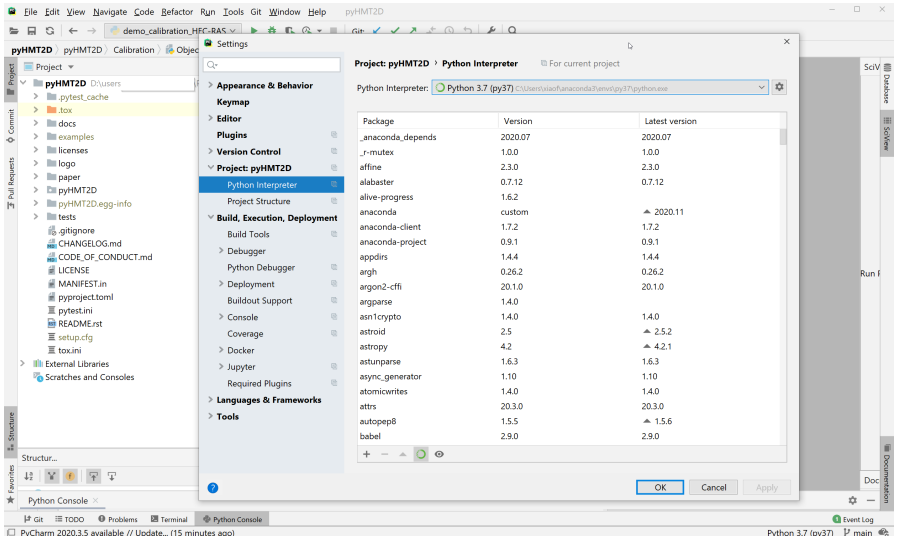


Figure 2.3 *pyHMT2D*'s Python Interpreter and environment setting in PyCharm.

After all these setups are done, you can try *pyHMT2D* using one of the examples or your own Python code.

- In PyCharm, double click the example Python code to open it.
- In the Python code editor window, right click and select "Run ..." (see Figure 2.4).

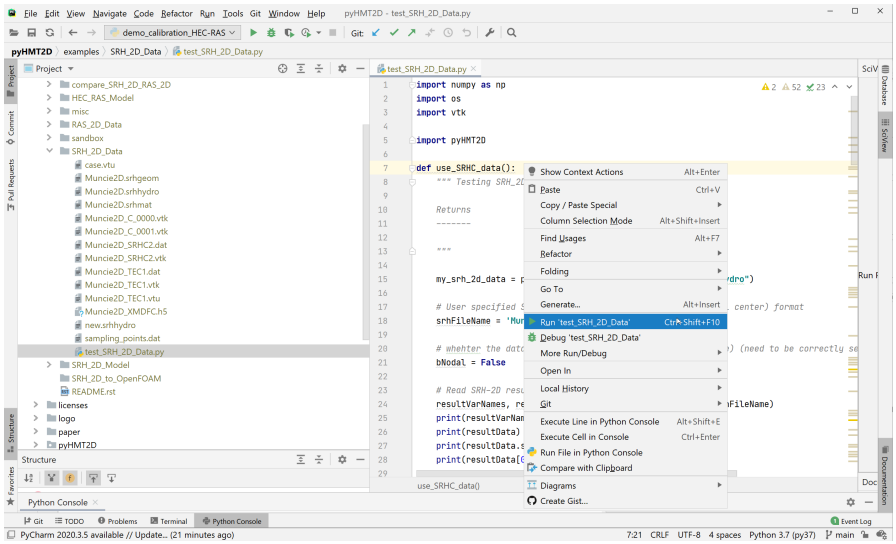


Figure 2.4 Run *pyHMT2D* code in PyCharm.

Chapter 3

Application Examples

This chapter shows some examples of using *pyHMT2D*. These examples range from simple backwater curve (in 2D), conversion of HEC-RAS mesh to SRH-2D, comparison of results between different models, control model runs, and automatic model calibration with Python optimization packages.

3.1 Simple Backwater Curve in 2D

This example is located in "`examples/compare_SRH_2D_RAS_2D/backwater_curves`". It has two folders, `SRH-2D` and `HEC-RAS-2D`, and some Python scripts. This example demonstrates the following *pyHMT2D*'s functionalities:

- Create an artificial georeferenced terrain for a rectangular channel with a given slope. Indeed, you can create any terrain by modifying the example Python script.
- Create HEC-RAS case for use in *pyHMT2D*.
- Run the HEC-RAS case to get results.
- Convert HEC-RAS mesh to SRH-2D mesh.
- Run the case in SRH-2D with exactly the same mesh as in HEC-RAS.
- Also run a one-dimensional backwater curve solver for the same case.
- Convert HEC-RAS and SRH-2D results into VTK.
- Sample on the VTK result files and make comparison.

In this example, a simple backwater curve is simulated with both HEC-RAS 2D and SRH-2D. The mesh is created in HEC-RAS 2D and then converted to SRH-2D. Specifically, the following steps are followed:

- Create an artificial bathymetry using `GDAL` with a given constant slope and domain size/extent. The script is named "`create_channel_terrain.py`".

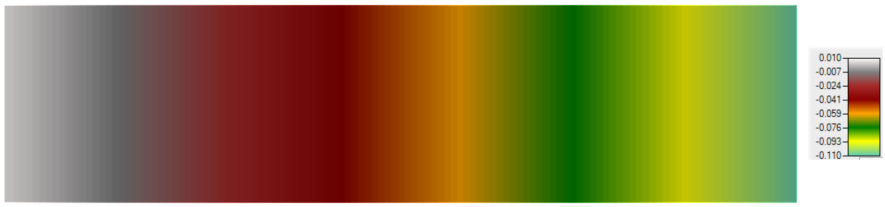


Figure 3.1 Generated terrain using *pyHMT2D*.

The resulted bathymetry data is in the format of GeoTIFF, which can be imported as terrain in HEC-RAS and SMS (see Figure 3.1).

Note: if you see the following error message when you run the script

```
ERROR 1: PROJ: proj_create_from_database: Cannot find proj.db
```

it is very likely that the `GDAL`'s environment variables are not set properly. Please refer to Section 2.1.2.

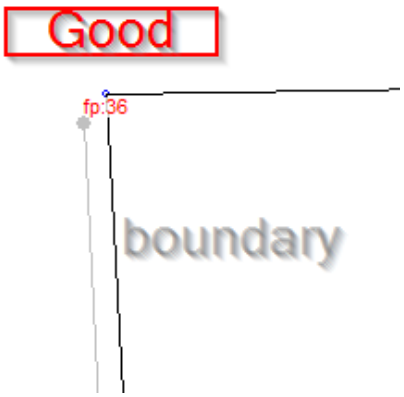
- Create and run HEC-RAS 2D simulation case (geometry, mesh, boundary conditions, unsteady flow data, simulation plan, etc.). See HEC-RAS's user manual for details if you are not familiar with HEC-RAS.

  Important notes:

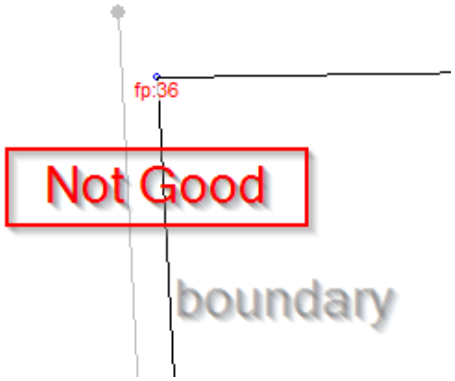
  – In HEC-RAS, when the "SA/2D Area BC Lines" are drawn, make sure the lines are slightly "shorter" than the intended boundary (and as close to the face points in the mesh as possible). This is because HEC-RAS use both stationing and face points to record BC lines. The reason is on page 3-43 in the section "Connecting 2D flow areas to 1D Hydraulic Elements". If the length of a BC line is longer than the total length of all face edges near the BC line, HEC-RAS will search up and down the boundary to add the adjacent boundary faces. This is reflected in the geometry HDF file ("[Geometry][Boundary Condition Lines][External Faces]"). *pyHMT2D* uses this entry in the HDF file to determine the face points on each boundary. If extra face points are included, it will confuse *pyHMT2D* and other models, such as 2D, which do not utilize this stationing concept.

    Figure 3.2a shows a proper placement of one end point, while Figure 3.2b shows a not proper example.

    To check whether the BC lines and their associated face points are created and exported properly, one can open the HEC-RAS .g##.hdf file in HDFView and examine the content in "[Geometry][Boundary Condition Lines][External Faces]". Also open RAS Mapper and turn



(a) BC line point slightly shorter than the intended boundary (good)



(b) BC line point longer than the intended boundary (not good)

Figure 3.2 Proper placing of BC line end points in HEC-RAS "Geometry Data" window to not include any unnecessary face points.

on cell number, face point number, face number and boundary. From the side comparison for example as shown in Figure 3.3, one can check whether the BC line's face points include unnecessary ones.

- Copy relevant HEC-RAS files (e.g., geometry .g01, .g01.hdf, plan .p01, .p01.hdf, project .prj, and bathymetry file .tif) to SRH-2D directory.
- Run the Python script named "`process_RAS_2D_Data.py`" to process HEC-RAS case data and save to VTK. The VTK file is called "`RAS2D_channel_0012.vtk`" in this case.

  The script also creates SRH-2D files (srhgeom, srhmat). The srhhydro file needs to be edited manually. A template can be used. All these SRH-2D files

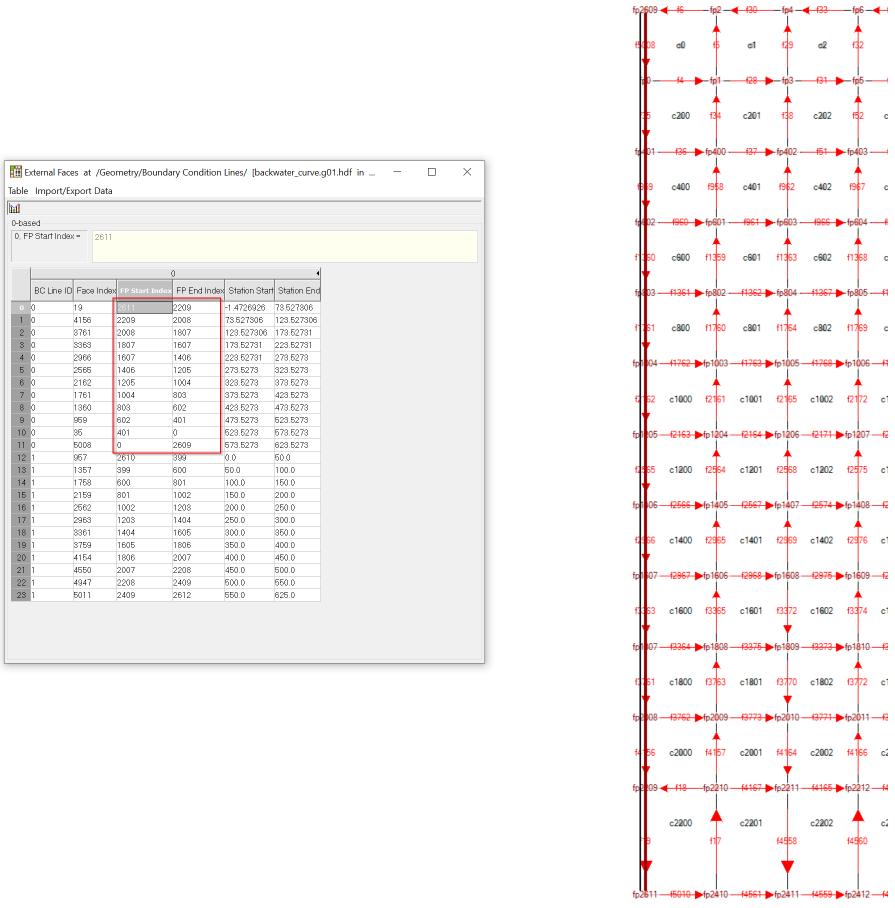


Figure 3.3 Inspection of BC line's face points with HDFView and RAS Mapper.

are editable text files which can be opened with e.g. Windows Notepad or Notepad++.

Inspect these text files (srhhydro, srhgeom, and srhmat) to make sure they are correct and consistent, e.g., boundary condition IDs, Manning's material IDs, etc. Also make sure that the number of `ManningsN` in srhhydro is consistent with the record in srhmat. For example, in the srhhydro file

```
ManningsN 0 0.03
ManningsN 1 0.03
ManningsN 2 0.04
```

the first line is the default Manning's n (with an ID of 0) and the rest is for different material zones. Correspondingly, in srhmat file,

```
SRHMAT 30
NMaterials 2
MatName 1 "zone_1"
Material 1 x x x
```

Here, `NMaterials 2` has to be the number of materials used in the case plus the default. In this case the value is 2, which means there is one material in the srhmat file (plus the default).

- Run the case with SRH-2D. We can use *pyHMT2D* to control the run of SRH-2D. But here, we will skip this function for the time being. We run the case through command window. You need to know where your SRH-2D is installed.

  For example, if you have SMS installed, it comes with SRH-2D. Thus, you first run the pre-processor:

  ```
  $ /path/to/SRH_Pre_Console.exe 3 backwater.srhhydro
  ```

  Then, run the SRH-2D solver:

  ```
  $ /path/to/SRH-2D_v330_Console.exe backwater_curve.DAT
  ```

- Run the Python script "`process_SRH_2D_Data.py`" to convert the SRH-2D results to VTK. The VTK file is named "`SRH2D_backwater_curve_C_0024.vtk`" in this case.

  Both VTK files from SRH-2D and HEC-RAS can be loaded into ParaView for inspection.

- Run the Python script "`compare_SRH_2D_HEC_RAS_2D.py`" to extract the water surface elevations from both SRH-2D and HEC-RAS 2D results in VTK format. It also run a simple 1D backwater curve solver. All three water surface profiles are plotted together for comparison (Figure 3.4). In addition, this script calculate the differences in simulation results between SRH-2D and HEC-RAS 2D and save them into separate VTK files. You can load these VTK files in ParaView to check the differences. For example, Figure 3.5 shows that the difference in water surface elevation is very small for this case.

  Note: Current version of SRH-2D v3.3's VTK format output (called Paraview in SMS and the srhhydro file) does not include any solution data. As such, make sure the output format is not "Paraview". Other formats, such as "SRHC" and "XMDFC" can be read by *pyHMT2D* and translate to VTK.

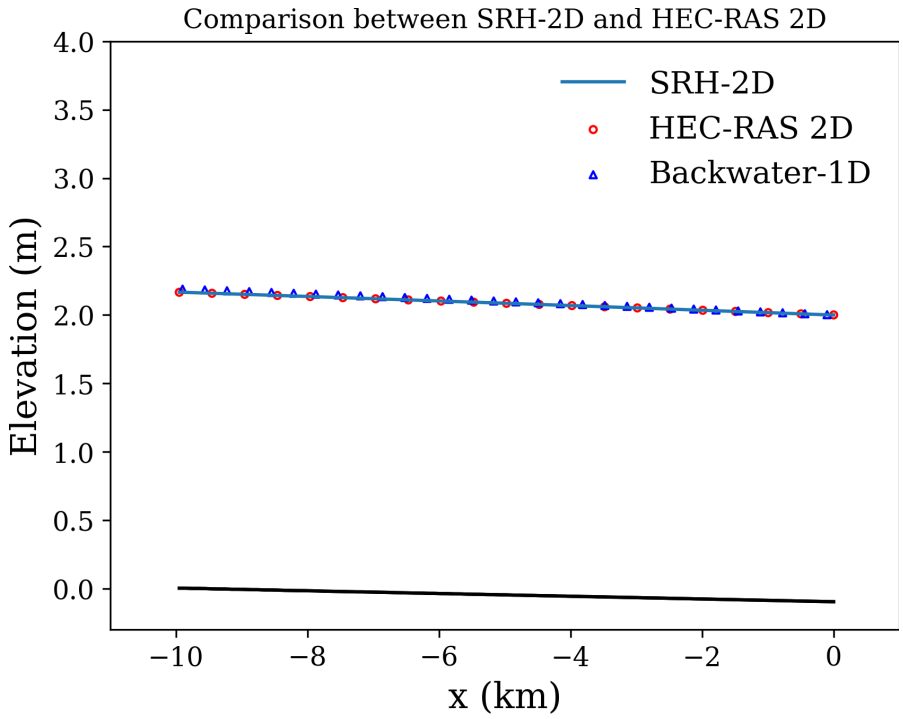


Figure 3.4 Comparison among SRH-2D, HEC-RAS 2D, and Backwater-1D solutions using *pyHMT2D*.

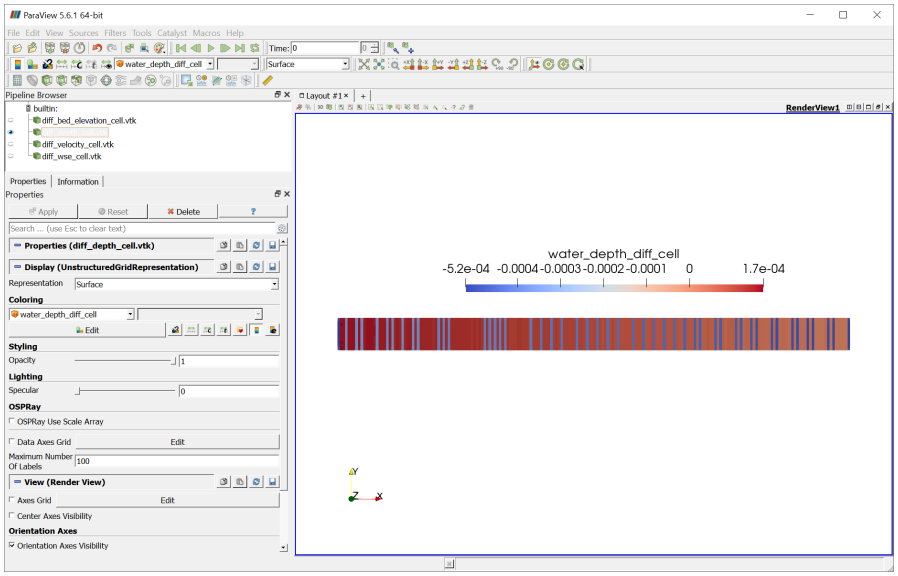
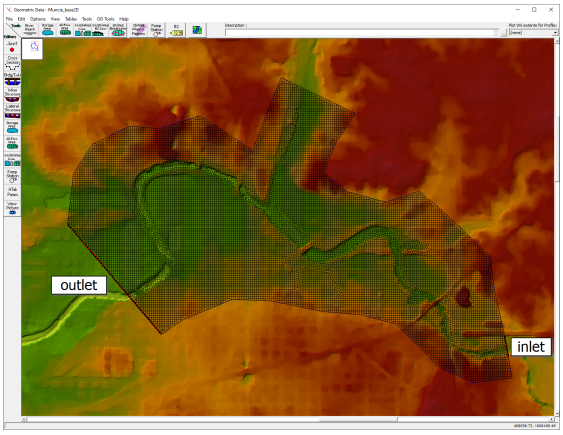


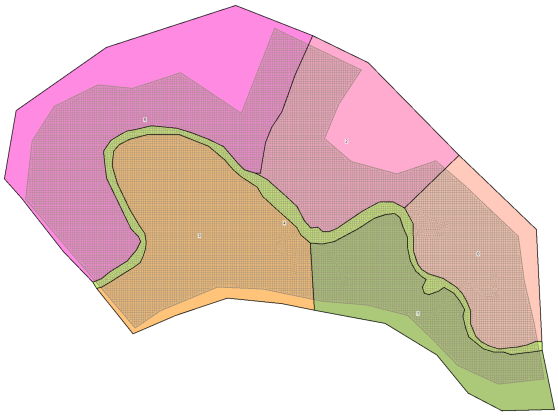
Figure 3.5 Difference in water surface elevations between SRH-2D and HEC-RAS 2D shown in ParaView.

3.2 A more realistic case - Munice

Use the simple example in the previous section as the foundation, this section shows the application of *pyHMT2D* to a more realistic 2D case. This case uses the terrain data in the "Munice" example that comes with HEC-RAS. The case is built from scratch with one single 2D flow area. Figure 3.6a shows the mesh and boundary conditions in HEC-RAS. The 2D flow area has one inlet with specified discharge and one outlet with specified water surface elevation. The Manning's n is not uniform in the domain. For demonstration purpose, the domain is divided into six zones, each of which has their own Manning's n value. The division is somewhat arbitrary. In reality, it should be based on the ground condition.



(a) The mesh and boundary conditions of the Muncie case in HEC-RAS.



(b) The Manning's n zones for the Muncie case

Figure 3.6 Mesh, boundary conditions, and Manning's n zones in HEC-RAS.

The boundary conditions are as follows. At the inlet boundary, the constant discharge is 20,000 cfs. At the outlet boundary, the constant water surface elevation

is 930 ft. The total simulation time is six hours, which is approximately long enough that at the end of simulation the flow field reached steady state. The simulated water depth at the end in RAS Mapper is shown in Figure 3.7.

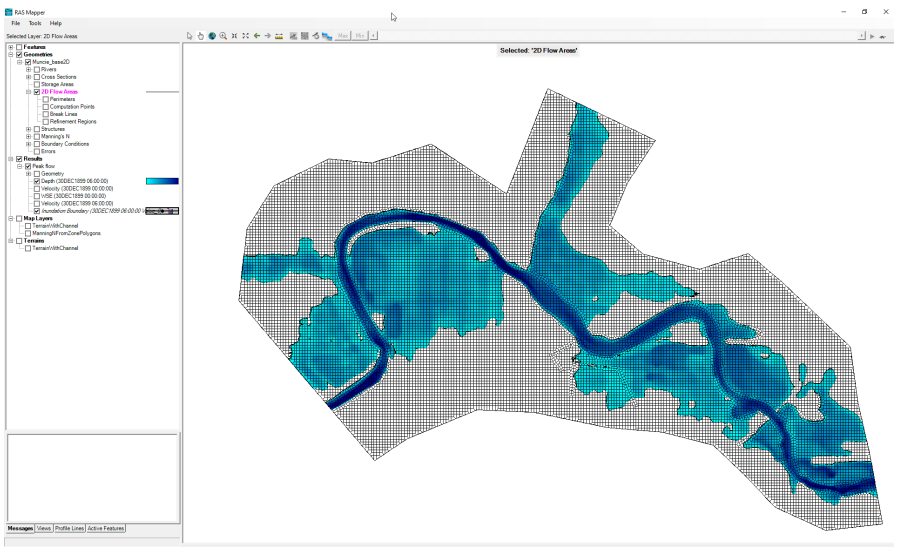


Figure 3.7 The simulated water depth for the Muncie case in HEC-RAS's RAS Mapper.

After the HEC-RAS simulation is done, you can use *pyHMT2D* to perform some processing:

- Run the Python script named "`process_RAS_2D.py`" to convert HEC-RAS 2D result to VTK and convert its mesh and Manning's n distribution to SRH-2D. The resulted VTK file is in the directory "`HEC-RAS`" and the SRH-2D files are in the directory "`SRH-2D`".
- In the directory "`SRH-2D`", modify and check the case configuration files, such as "srhhydro", "srhgeom", and "srhmat".
- Run the SRH-2D case to get the results.
- Run the Python script named "`process_SRH_2D.py`" which converts the SRH-2D simulation results into VTK.
- Run the Python script "`compare_SRH_2D_HEC_RAS_2D.py`", which reads the VTK files for HEC-RAS and SRH-2D results, calculate the differences in water depth, water surface, elevation, bed elevation, and velocity, save them into separate VTK files.

All the generated VTK files can be loaded into ParaView for inspection. For example, Figure 3.8 shows the comparison of simulated water depth from HEC-RAS 2D and SRH-2D in ParaView. As shown in the figure, the comparison is

qualitatively good for this case. To really quantify the difference, you can load the calculated difference files in ParaView. For example, Figure 3.9 shows the difference in bed elevation, water depth, velocity, and water surface elevation calculated using *pyHMT2D* in ParaView.

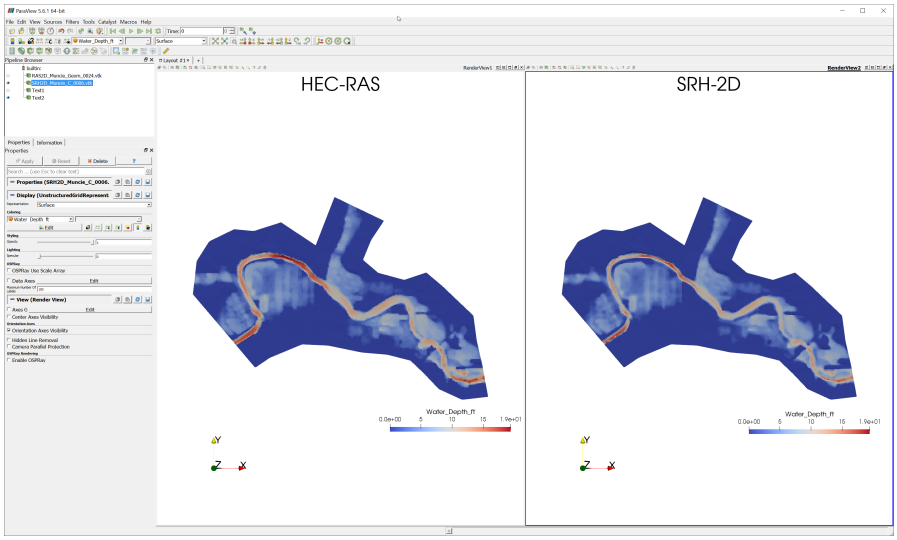


Figure 3.8 Side-by-side comparison of water depth simulated by HEC-RAS 2D and SRH-2D in ParaView.

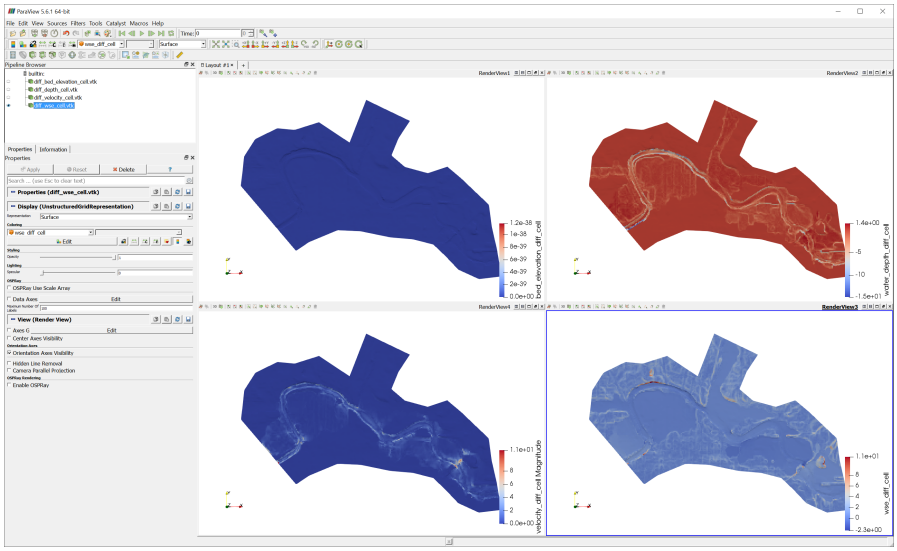


Figure 3.9 Differences in bed elevation, water depth, , velocity, and water surface elevation shown in ParaView.

3.3 Run SRH-2D simulation with *pyHMT2D*

This example is in the directory "`examples/SRH_2D_Model`". We will demonstrate how to use a Python script to run SRH-2D simulations (both pre-processing and solver) using *pyHMT2D*. The simulation case is the same Muncie case used in previous section. The SRH-2D configuration files, `srhhydro`, `srhgeom`, and `srhmat`, have been provided.

To run the case, run the Python script file named "`demo_SRH_2D_Model.py`", which will run the SRH-2D case and convert the result into VTK. In the script, the key part to control the run of SRH-2D is as follows:

```
1    #create a SRH-2D model instance
2    my_srh_2d_model = pyHMT2D.SRH_2D.SRH_2D_Model(version, srh_pre_path,
3                       srh_path, extra_dll_path, faceless=False)
4
5    #initialize the SRH-2D model
6    my_srh_2d_model.init_model()
7
8    print("Hydraulic model name: ", my_srh_2d_model.getName())
9    print("Hydraulic model version: ", my_srh_2d_model.getVersion())
10
11   #open a SRH-2D project
12   my_srh_2d_model.open_project("Muncie.srhhydro")
13
14   #run SRH-2D Pre to preprocess the case
15   my_srh_2d_model.run_pre_model()
16
17   #run the SRH-2D model's current project
18   my_srh_2d_model.run_model()
19
20   #close the SRH-2D project
21   my_srh_2d_model.close_project()
22
23   #quit SRH-2D
24   my_srh_2d_model.exit_model()
```

It shows the typical process of creation of a `SRH_2D_Model` object, initialize the model, open a simulation project (case), run the pre-processor, run the solver, close the project, and exit the model.

3.4 Run HEC-RAS simulation with *pyHMT2D*

This example is in the directory "`examples/HEC_RAS_Model`". We will demonstrate how to use a Python script to run HEC-RAS simulations using *pyHMT2D*. The simulation case is the same 2D Muncie case used in previous section. In fact, the control of HEC-RAS run is not limited to 2D cases. Any HEC-RAS case can be ran in the script shown here. The HEC-RAS case files have been provided.

To run the case, run the Python script file named "`demo_HEC_RAS_Model.py`", which will run the HEC-RAS case and convert the result into VTK. In the script, the key part to control the run of HEC-RAS is as follows:

```
1   #create a HEC-RAS model instance
2   my_hec_ras_model = pyHMT2D.RAS_2D.HEC_RAS_Model(version="5.0.7",
3                                                   faceless=False)
4
5   #initialize the HEC-RAS model
6   my_hec_ras_model.init_model()
7
8   print("Hydraulic model name: ", my_hec_ras_model.getName())
9   print("Hydraulic model version: ", my_hec_ras_model.getVersion())
10
11  #open a HEC-RAS project
12  my_hec_ras_model.open_project("Muncie2D.prj",
13                  "Terrain/TerrainMuncie_composite.tif")
14
15  #run the HEC-RAS model's current project
16  my_hec_ras_model.run_model()
17
18  #close the HEC-RAS project
19  my_hec_ras_model.close_project()
20
21  #quit HEC-RAS
22  my_hec_ras_model.exit_model()
```

It shows the typical process of the creation of "`HEC_RAS_Model`" object, model initialization, opening a project (case), running the model, closing project, and exiting the model.

3.5 Automatic Calibration

Calibration of hydraulic models is the process of adjusting model parameters such that the difference between model prediction and measurement data is minimized to be below a pre-determined threshold. With *pyHMT2D*'s control and automation of hydraulic models, and its ability to sample and probe simulation results, it is relatively easy to automate the calibration process.

This section demonstrate three examples, which include a simple 1D backwater curve solver, SRH-2D, and HEC-RAS 2D. These cases are located in "`examples/calibration`"

3.5.1 Simple 1D Backwater Curve

This is a simple toy problem to demonstrate the basic elements of the automatic calibration using *pyHMT2D*. It is a 1D backwater curve in a channel of 10 km. Figure 3.10 shows the setup.

There are two zones of Manning's n, with the value 0.055 and 0.04, for upstream and downstream zones, respectively. The bottom slope is 0.00001 and the specific discharge is 0.48 m^2/s. The backwater curve is firstly simulated with *pyHMT2D*'s `Backwater_1D_Model` class and four sampling points are used to sample the manufactured solution. The sampled water surface elevations and velocities are saved in two CSV files, i.e., `sampled_wse.csv` and `sampled_velocity.csv`, which will be used as measurement data for calibration.

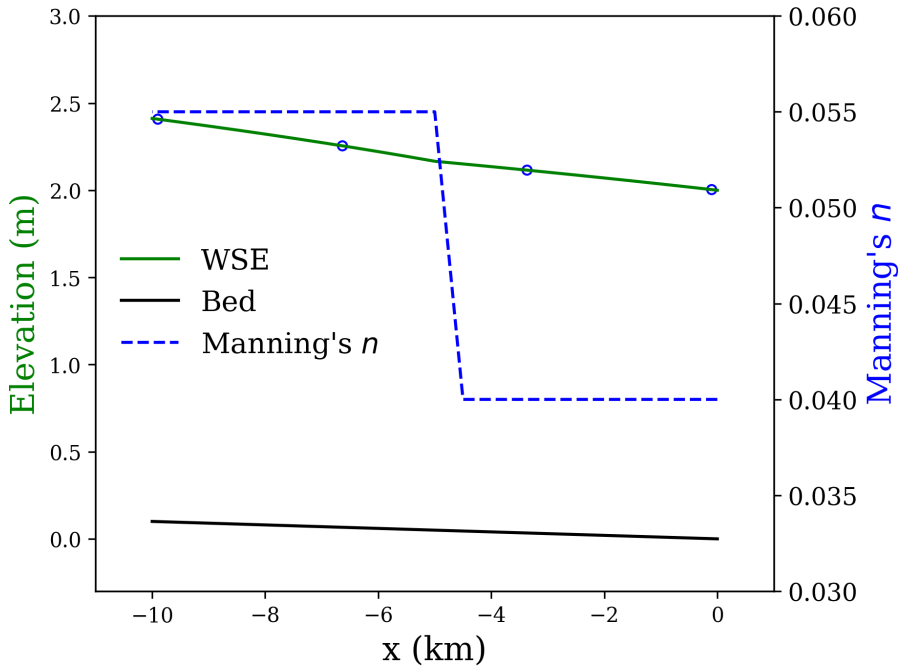


Figure 3.10 Setup of the 1D backwater curve calibration case.

In the Python script named "`demo_calibration_Backwater_1D.py`", two functions are provided, namely "`demo_backwater_1d_model_data()`" to produce the manufactured solution, and "`demo_calibrator()`" to perform the calibration. The calibration function is very simple and only has two lines:

```
1    my_calibrator = pyHMT2D.Calibration.Calibrator("calibration.json")
2
3    my_calibrator.calibrate()
```

It creates a "`Calibrator`" object, which reads the calibration configuration from the "`calibration.json`" JSON file. JSON stands for JavaScript Object Notation and is a data format easy for human to read and edit. Details about JSON can be found at https://www.json.org/json-en.html.

In *pyHMT2D*, this file has the following sections:

- `model`: It specifies which hydraulic model to run. Currently the three options are `Backwater-1D`, `SRH-2D`, and `HEC-RAS`.
- Section for the selected model: For this example, there is a section named `Backwater-1D` to specify the 1D backwater model, e.g., name, units, channel length, Maning's n zones, number of grid, specific discharge, etc. The content of this section depends on the specific model. For `SRH-2D` and `HEC-RAS`, see the two examples following this one.
- `calibration`: It specifies the following:
  - `calibration_parameters`: For example, the Manning's n for each zones. For each calibration parameter, an initial guess and calibration value range should be specified. Each calibration parameter can be optionally turn on or off. If it is off, the parameter will be fixed as its initial guess.
  - `objectives`: This is a list of calibration objectives (targets). For example, here we have two calibration targets, water surface elevation and velocity. For each target, we need to specify:
    - `name`: just a name to identify the target,
    - `solVarName`: solution variable name, used to sample the solution for comparison)
    - `type`: e.g., point measurement,
    - `weight`: used in the calculation of total error,
    - `file`: the measurement data file, and
    - `error_method`: how the error is calculated, either absolute or relative.

– `optimizer`: name of the optimizer. Currently the only options are "`scipy.optimize.local`", "`scipy.optimize.global`" and "`enumerator`". The former two use the "`scipy.optimize`" package. The last one is a customized "optimizer" which loops over user-defined enumeration of parameter combinations.

– Section for the specified optimizer. For example, if "`scipy.optimize.local`" is selected, this section should specify `method` and `options`. Information in this section is passed along to `scipy.optimize` module. More information can be found on its website.

  If the "`enumerator`" optimizer is specified, the user needs to list the calibration parameter combinations in the JSON file. This approach is useful if the user only wants to find the best performing parameter set out of a list. See the example "`calibration_enumerator.json`" file.

Run the python script "`demo_calibration_Backwater_1D.py`" to perform the calibration. If the calibration is successful, *pyHMT2D* will report the calibrated parameter. You can compare the results with the true values of 0.04 and 0.055 for downstream and upstream, respectively.

In this example, to show the power of Python code, another script named "`calibration_plot_and_animation.py`" is provided. It has two functions.

One function is to read the calibration results and plot the comparison of water surface elevation and velocity vectors between simulation and measurement. Figure 3.11 shows the plot, which can give the modeler direct visual comparison. For this simple case, the match between simulation and measurement is almost perfect.

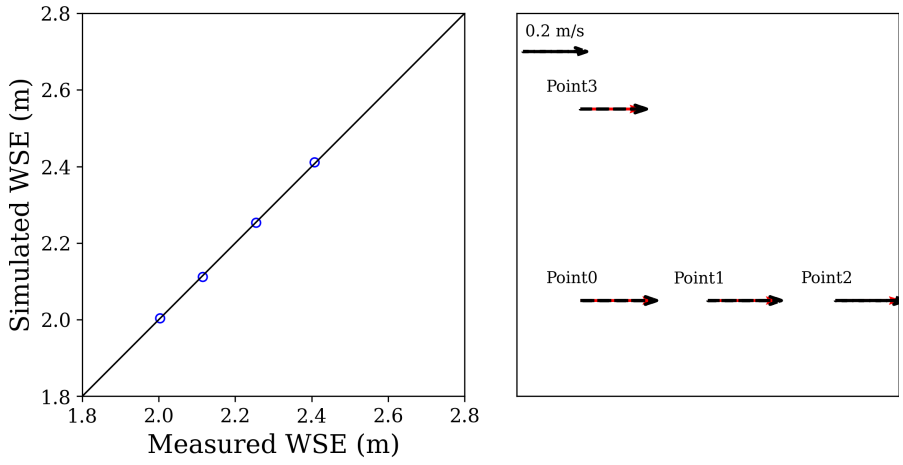


Figure 3.11 Comparison between simulation and measurement data for the 1D backwater curve calibration case. For the velocity plot, the red vectors are from measurement and the black ones are from simulation.

The second function is to animate the calibration process. For cases with less than three calibration parameters, we can plot the trajectory of the calibration in the parameter space. The results of this function are a MP4 video file named "`calibration_process.mp4`" and a figure for the final frame of the video as shown in Figure 3.12. The left figure shows the trajectory of (n_1, n_2) pair (downstream and upstream Manning's n) at each calibration iteration and the right figure shows the calibration error as a function of iteration number.

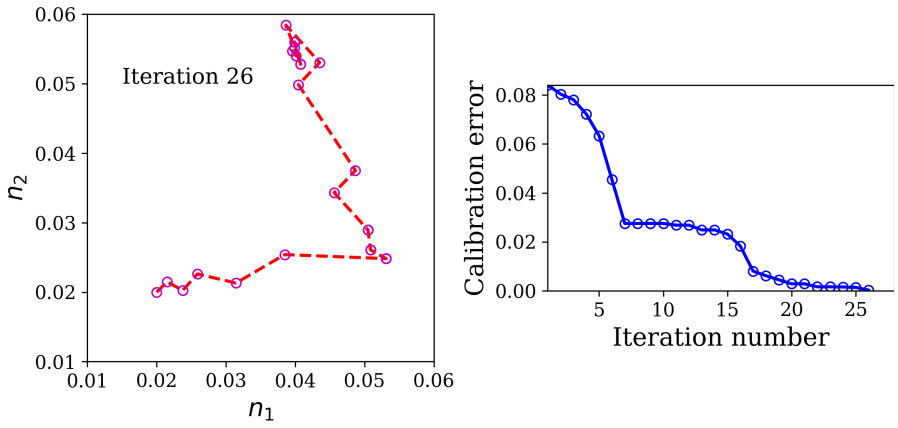


Figure 3.12 Comparison process for the 1D backwater curve calibration case.

3.5.2 SRH-2D

This the same 2D Munice case except that the Manning's n zones are modified. As shown in previous sections for this case, the majority of the inundation area is close to the main channel. Thus, the main calibration parameters should be the Manning's n values in the channel. To make the case more meaningful for practical purpose, the main channel is future divided into three segments, namely upstream, middle, and downstream. They have their own hypothetical Manning's n values. Figure 3.13 shows the zones and their names for this case.

A Python script named "`demo_calibration_SRH-2D.py`" is provided which has two functions. One is called "`demo_srh_2d_model_data()`" which run SRH-2D once to get the Manufactured solution and sample results as measurement data for calibration. The run of SRH-2D case depends on the usual `srhhydro`, `srhgeom`, and `srhmat` files in this directory. So make sure the Mannng's n values in the `srhhydro` files are the ones that you want to use to generate the manufactured solution.

The other function is called "`demo_calibrator`" which again only has two lines. It creates the "`Calibrator`" object and reads the calibration specification in the JSON file "`calibration.json`". The content in this file is similar to the previous section and self-explanatory. One note here is that there are eight Man-

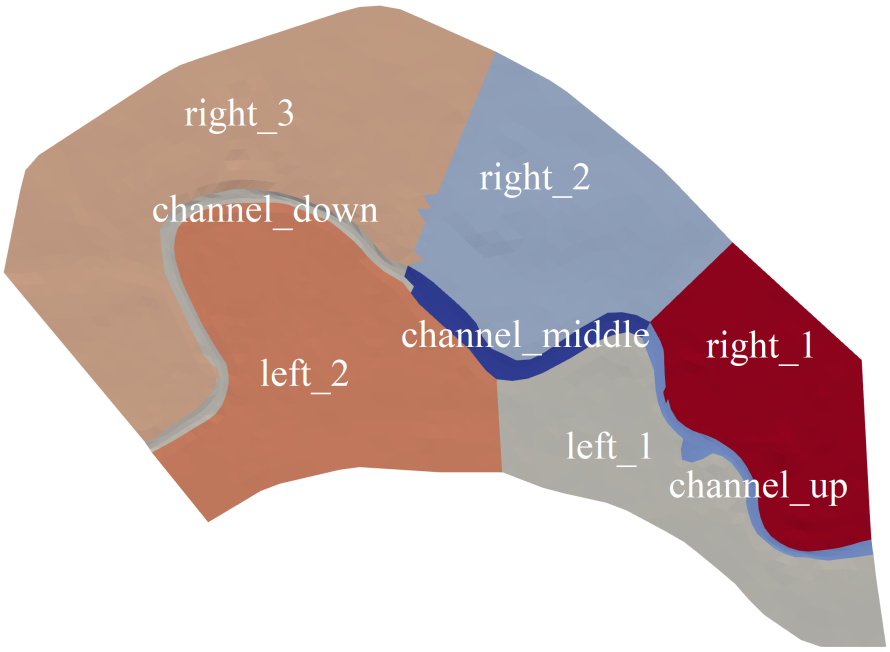


Figure 3.13 Manning's n zones for the Munice 2D calibration case.

ning's n calibration parameters. However, only three of them, namely `channel_up`, `channel_middle`, and `channel_down`, are activated because they are most important for the hydrodynamics here. During the calibration process, the optimization package will search for the best Manning's n values for the three zones. *pyHMT2D* will modify their values in the `srhhydro` file for SRH-2D to simulation at every iteration.

With the example configuration JSON where the "`Powell`" method is used as the optimizer and the tolerance for both parameter and error function to be 0.01, the calibrated Manning's n values for upstream, middle, and downstream zones of the channel are 0.02936002, 0.02139392, 0.04189711 on my computer (may slightly different on your computer). The true values are 0.03, 0.02, and 0.04, respectively. If you reduce the tolerance or change the optimization method, the calibrated values should be closer to the true values.

To visualize the calibration results, a Python script named "`calibration_plot.py`" is provided. It generate the plot shown in Figure 3.14.

3.5.3 HEC-RAS

The calibration case is the same as the SRH-2D case. The case has been created in HEC-RAS. We again want to calibrate only the three channel zones. A Python code named "`demo_calibration_HEC-RAS.py`" is provided which again contain two main functions. One is for the generation of manufactured solution and sample

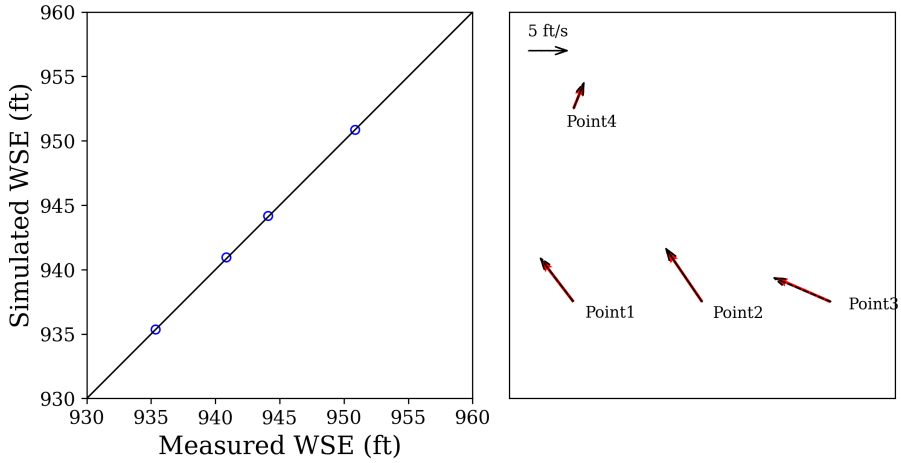


Figure 3.14 Comparison of water surface elevation and velocity vectors between SRH-2D simulation and measurement for the Munice 2D calibration case. For the velocity plot, the red vectors are from measurement and the black ones are from simulation.

result as measurement data for calibration. The other is to run the calibration.

In practice, there might be a need to calibrate the model with a limited number of user-supplied combinations of parameters, instead of using the optimizer's automatic calibration. This can be achieved with `scipy`'s brute force method. The JSON configuration file named "`calibration_brute_force.json`" shows how to do it. In this case, for each Manning's n parameter in its range, three values are taken (Ns =3). Therefore, 27 combinations (= 3^3) of Manning's n values will be used and the best of these 27 will be reported at the end. It is very obvious that this brute force method becomes very expensive as the number of calibration parameters and the sampling number N_s increase. This method is probably only suitable if you have a good sense of what the Manning's values should be and search in a small range.

```
1 ^^M    "optimizer": "scipy.optimize.global",^^M    "scipy.optimize.global": {^^M
```

Chapter 4

Code Structure and Implementation Details

4.1 Control SRH-2D with Python

(To be written)

4.2 Control HEC-RAS with Python

HEC-RAS is mainly a Window-based hydraulic model with a graphical user interface. It is designed to be operated with user's manual input (such as keyboard and mouse). While the manual operation provides direct access and feedback to the user, it is not efficient for other tasks where HEC-RAS is involved, for example uncertainty analysis and Monte-Carlo simulations where thousands of runs need to be performed. Another scenario where manual operation is cumbersome is model calibration. The major calibration parameter is the Manning's n which quantifies the channel roughness and flow resistance. The Manning's n can vary in space and time. Typically, different zones in the simulation domain have different Manning's n values which need to be calibrated.

4.2.1 Existing tools

To expedite the simulations, it is desired to interact and control HEC-RAS in an automatic fashion, typically in the form of program scripting. A scripting language, such as VBA, Matlab, and Python, can be written to control the pre-processing, simulation, and post-processing of HEC-RAS. Many previous research and development have been reported on the control of HEC-RAS with scripting languages. For example, Moya Quiroga et al. (2013) showed the evaluation of HEC-RAS modeling uncertainties with the combination of Delphi, VB.NET, and Windows command line batch scripts. In Goodell (2014) and its accompany code, VBA was used to interact with HEC-RAS and a variety of demonstration examples were provided. Leon and Goodell (2016) developed several MATLAB scripts for controlling HEC-RAS, which include input file reading and writing, output file

reading, plotting, and parallel computation. Similar implementation using Python has been reported in Dysarz (2018).

In this document, it is unnecessary to describe in details all existing tools and codes that can control HEC-RAS. The following is an incomplete list:

- razviz: a tool package written in R for visual inspection of HEC-RAS results and calibration. It only has the reading function for HEC-RAS result files, not controlling of HEC-RAS program itself.
- mcat-ras: HEC-RAS model content and analysis tool (MCAT). It is written in the Go programming language
- pyras: A python package to control HEC-RAS (last update is six year ago).
- rascontrol: A Python package to control HEC-RAS. It is an equivalent of the VBA code in (Goodell, 2014). It can be used to open, run and extract results from HEC-RAS models. It seems to be developed mainly for 1D models.
- parserasgeo: a companion Python package to deal with HEC-RAS geometry files (importing, editing, and exporting). It is also mainly for 1D models.
- hecrasio: a collection of Python scripts to read HEC-RAS results.
- FluEgg: a fluvial fish egg drifting simulator which utilizes the flow result from HEC-RAS (Garcia et al., 2013). The code is written in MATLAB and has a script to extract HEC-RAS results.
- RAS-Controller-with-Matlab: a Matlab script to control HEC-RAS for flood inundation forecasting and evaluation (Gomez et al., 2019).
- MDAL: Mesh Data Abstraction Library, which is a C++ plugin for QGIS. This library can read unstructured mesh and results from many 2D hydraulic models and visualize in QGIS. It is not designed to control hydraulic modeling runs.

4.2.2 HEC-RAS Interface

Although without official documentation, HEC-RAS does expose part of its internal functions through the Component Object Model (COM), which is the foundation for Microsoft's OLE and ActiveX technologies. With COM, a Windows application can be controlled from outside. All existing HEC-RAS control scripts and packages listed in the previous section utilize this functionality to communicate with it. *pyHMT2D* does the same using the Python language.

Python itself as a programming language can not directly interface with COM. However, there are Python libraries that have been developed for that purpose.

One such example is the "Python for Windows Extensions" (pywin32) package (Hammond and Robinson, 2000). In *pyHMT2D*, *pywin32* is used to access Win32 API, create and use COM objects that interact with HEC-RAS. In the Python environment on a computer, *pywin32* can be easily installed through Anaconda's Environments GUI or the `conda` command, or through `pip`:

```
$ pip install pywin32
```

With *pywin32*, there are some different options to interact with HEC-RAS from a Python script. Essentially, all these options uses the same single functionality that comes with *pywin32*.

Option 1: Directly use a HEC-RAS COM object from Python

A typical, and most straightforward, way of creating and using a COM object in Windows with Python is as follows:

```
1 import win32com.client as win32
2 comObj = win32.Dispatch("Object.Name")
3 comObj.Method()
4 comObj.property = "New Value"
5 print(comObj.property)
```

Here, a COM object with the name of "Object.Name" is created and launched "(dispatched)". Then, the methods of this COM object can be called and its properties can be modified. For the case of HEC-RAS, the "Object.Name" is "RAS507.HECRASController" for HEC-RAS version 5.0.7 (as of the writing of this document, it seems "HEC-RAS 6.0 Beta Update 1" uses the name "'RAS5x.HECRASController', which should be corrected in future release). A full example is as follows:

To make sure the "Object.Name" is correctly specified to *pywin32*'s dispatch function, one can further inspect the Windows's Registry Editor window under `Computer\HKEY_CLASSES_ROOT` as shown in Figure 4.1. In this example, there are two versions of HEC-RAS registered on this computer. Thus, there are two HEC-RAS Controllers: `RAS507.HECRASController` for version 5.0.7 and `RAS5x.HECRASController` for version 6 (which again needs to be corrected by HEC in future release).

If the Python environment is set correctly (most importantly the *pywin32* is installed) and HEC-RAS version 5 or 6 is installed, this script will launch HEC-RAS, open the specified project, run its current plan, print the message from the run, and then exit. The power of scripting and automation with HEC-RAS rely on this COM mechanism.

```python
import win32com.client as win32
import os

ras = win32.Dispatch("RAS5x.HECRASController")

project = 'Muncie2D/Muncie2D.prj'

#get the absolute path of the project file (it seems
#HEC-RAS does not like the relative path)
project = os.path.abspath(project)

#open the project
ras.Project_Open(project)

#show HEC-RAS main window
ras.ShowRas()

#get the current project
res = ras.Project_Current()
print('Project_Current:', res, '\n ')

#compute the current plan
nmsg = None
msg = None
res = ras.Compute_CurrentPlan(nmsg, msg)
print(res, nmsg, msg)

ras.QuitRas()
```

Script 4.1 Control HEC-RAS with *pywin32*

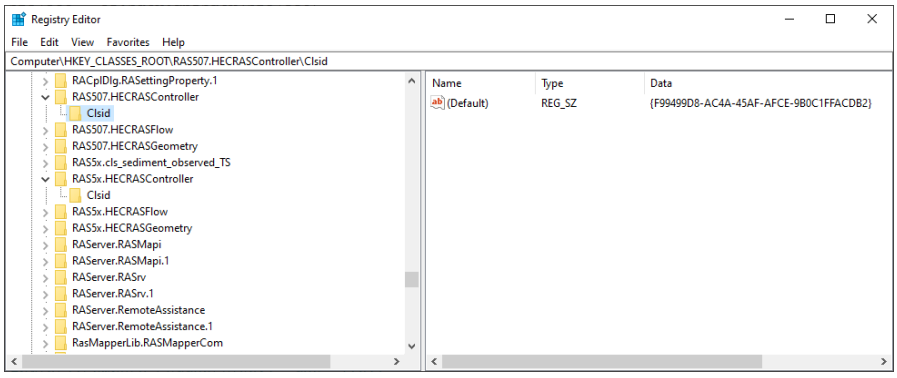


Figure 4.1 HEC-RAS's COM object name in Windows Registry Editor

All the magics of *pywin32* manipulating a COM object hinge on how an application, such as HEC-RAS, register themselves in Window's registry. One can open Windows "Registry Editor" and search "HEC River Analysis System". If HEC-RAS is installed, you will see something similar to Figure 4.2. The exact ID in the registry depends on the version of HEC-RAS.

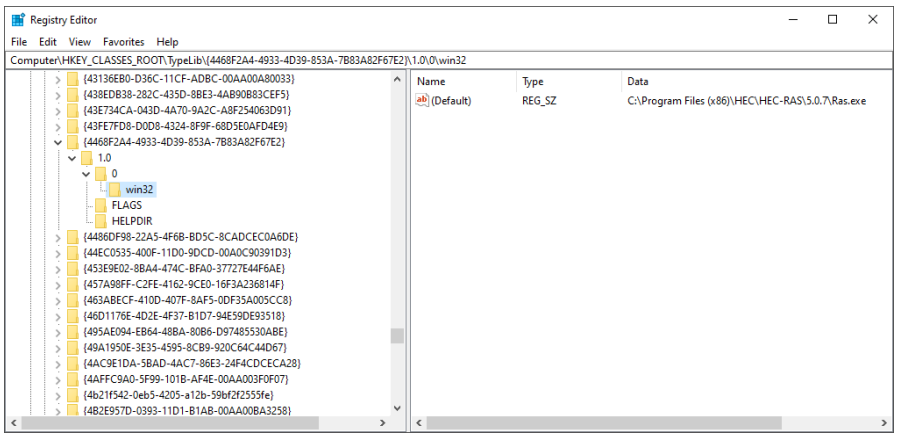


Figure 4.2 HEC-RAS in Windows Registry Editor

If you have multiple versions of HEC-RAS installed, they will correspond to different entries with different IDs (named "CLSID" - Class ID, to be specific). And that is how Windows differentiate them. These ID numbers were generated by HEC-RAS developers and they are (theoretically) globally unique. So these ID numbers are like the social security number of each HEC-RAS release (if HEC developers change them in the source code of different version). In fact, you can also generate such unique ID by yourself in Windows command window. Microsoft provides a utility program called "GUIDGEN.exe" which generates an unique ID based on current time, your computer's network adapter address, and other information.

Microsoft calls these CLSID numbers "globally unique identifier (GUID)", which consists of 128 bits in the form of one group of 8 hexadecimal digits, followed by three groups of 4 hexadecimal digits each, and followed by one group of 12 hexadecimal digits. For the example in Figure 4.2 for HEC-RAS 5.0.7, its unique CLSID is **4468F2A4-4933-4D39-853A-7B83A82F67E2**

Although useful, the Windows registry is not so helpful to us because it does not provide more information about what an application's COM object can do. For that, a user needs the documentation of the application, in particular what COM interfaces it provides and how to use them. Unfortunately, HEC-RAS currently does not have official documentation on its COM capability. There are some explanations and examples in the literature, for example Goodell (2014). It is hoped that in the future HEC will release such information.

As an alternative, there are several ways to explore what a HEC-RAS COM object can do, i.e., its functions and variables. One way is to use a COM browser. *pywin32* ships with a simple COM browser which can be used as follows:

- First go to the "client" directory of *pywin32*, which depends on where and how Python is installed. For example, on the author's computer, "Anaconda" is installed in the author's user account (not for the system). For the ease of use, it is suggested that "Anaconda" is installed individually for each user. In the author's case, the directory is

  ```
  C:\Users\xiaof\anaconda3\Lib\site-packages\win32com\client
  ```

  and its content looks like the following:

  ```
  C:\Users\xiaof\anaconda3\Lib\site-packages\win32com\client>dir
   Volume in drive C is OS
   Volume Serial Number is 8233-87F2

   Directory of C:\Users\xiaof\anaconda3\Lib\site-packages\win32com\client

  03/07/2021  12:44 AM    <DIR>          .
  03/07/2021  12:44 AM    <DIR>          ..
  01/15/2020  10:54 AM            23,485 build.py
  11/13/2019  06:29 PM             1,705 CLSIDToClass.py
  01/15/2020  10:54 AM            20,236 combrowse.py
  11/13/2019  06:29 PM             1,310 connect.py
  01/15/2020  10:54 AM            22,086 dynamic.py
  01/15/2020  10:54 AM            23,678 gencache.py
  01/15/2020  10:54 AM            46,987 genpy.py
  01/15/2020  10:54 AM            12,411 makepy.py
  ```

```
01/15/2020  10:54 AM             4,959 selecttlb.py
11/13/2019  06:29 PM             7,827 tlbrowse.py
01/15/2020  10:54 AM             2,968 util.py
01/15/2020  10:54 AM            22,594 __init__.py
03/07/2021  12:44 AM    <DIR>          __pycache__
```

- Then run the "combrowse.py" script:

```
$ python combrowse.py
```

A "Python Object Browser" window will show. Browse in the "Registered Type Libraries" and find the "HEC River Analysis System" entry (see Figure 4.3). This will show you everything in HEC-RAS this is exposed through COM. If you have multiple versions of HEC-RAS installed, you will see multiple entries of "HEC River Analysis System" (it is suggested that HEC adds the version information in the entry name to differentiate different versions).

Among many other things, the most important classes (COM uses object-oriented programming concept) are "`_HECRASController`", "`_HECRASGeometry`", and "`_HECRASFlow`". All callable methods (functions) in these classes are listed. For example, under "`_HECRASController`" one can find "`Project_Open`", "`ShowRas`", "`Project_Current`", "`Compute_CurrentPlan`" and "`QuitRas`" methods used in the simple example above.

In the "Python Object Browser" window, one can also find other classes with similar names of "`HECRASController`", "`HECRASGeometry`", and "`HECRASFlow`". Respectively, they correspond to "`_HECRASController`", "`_HECRASGeometry`", and "`_HECRASFlow`". They are a wrapper to the classes with the names staring with an underscore. In fact, in a Python code, if one wants to directly call the functions (not through *pywin32*), one can create an object from the wrapper classes (demonstration examples will be shown later).

In the full Python example above, the "Dispatch" method in *pywin32*'s "win32com.client" module. There is in fact another method that can be used. Simply replace

```
1 ras = win32.Dispatch("RAS5x.HECRASController")
```

with

```
1 ras = win32.gencache.EnsureDispatch('RAS5x.HECRASController')
```

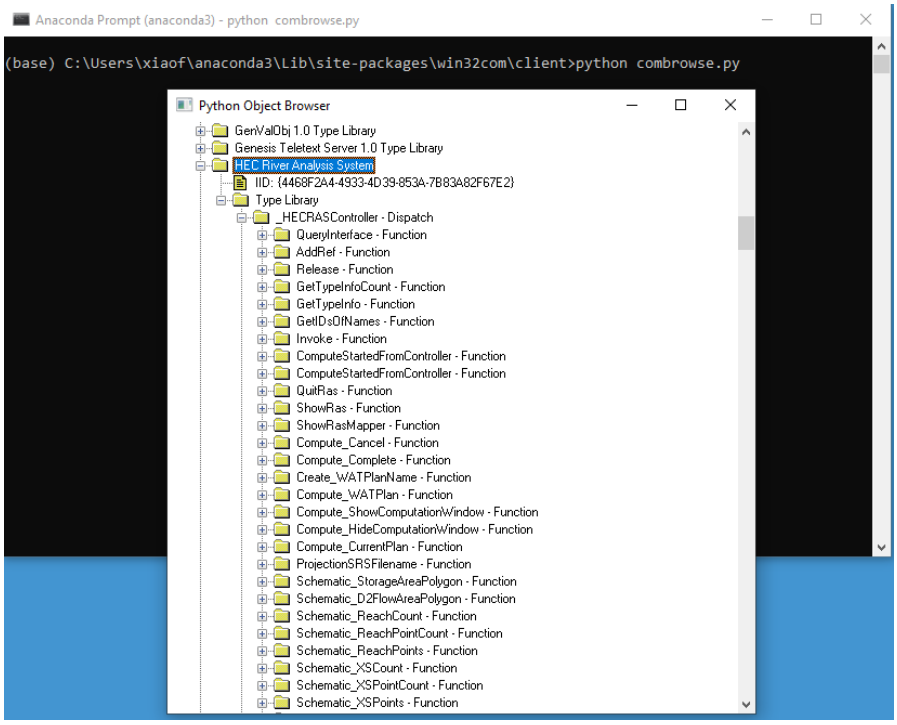


Figure 4.3 HEC-RAS in "combrowser"

The difference here is probably not that critical for our use of *pyHMT2D*. But it is good to know. The difference is whether a cache (pre-generated) HEC-RAS python module will be produced or not. This cache module is typically located in

`USER_PROFILE\AppData\Local\Temp\gen_py\PYTHON_VERSION\`

For example, on the author the computer, the location is

`C:\Users\xiaof\AppData\Local\Temp\gen_py\3.8`

Note: Content in this directory can be safely cleared if you need a fresh start. *pywin32* will recreate the content when necessary.

If a cache HEC-RAS python module already exists, there is practically no difference from the point of view of a user. To show some idea about when is in the cache module, the following is an excerpt from the file "_HECRASController.py" within the directory

`C:\Users\xiaof\AppData\Local\Temp\gen_py`
`\3.8\6C8C3DF5-4478-4673-9215-17ACF18140DDx0x1x0`

on the author's computer (corresponding to HEC-RAS 6.0 Beta Update 1):

```
from win32com.client import DispatchBaseClass
class _HECRASController(DispatchBaseClass):
        ...
        def Compute_Cancel(self):
                return self._oleobj_.InvokeTypes(1610809350, LCID, 1, (11, 0), (),)

        def Compute_Complete(self):
                return self._oleobj_.InvokeTypes(1610809351, LCID, 1, (11, 0), (),)

        ...

        def Project_Current(self):
                # Result is a Unicode object
                return self._oleobj_.InvokeTypes(1610809428, LCID, 1, (8, 0), (),)

        ...

        def QuitRas(self):
                return self._oleobj_.InvokeTypes(1610809346, LCID, 1, (24, 0), (),)

        ...

        def ShowRas(self):
                return self._oleobj_.InvokeTypes(1610809347, LCID, 1, (24, 0), (),)
```

It is clear that all the functions what we call in the example are declared here.

When we call `win32.Dispatch("RAS5x.HECRASController")` and **if there is no cache HEC-RAS version 5.x python model exists**, *pywin32* simply builds a generic COM object without knowing anything special about it, for example what methods and constants it has. This is called "dynamic dispatch" or "late binding" because the knowledge about the object will be built dynamically (later). Your can check this in Python with

```
>>> repr(ras)
```

here the `repr()` returns a printable representation of the object `ras`. Also, you can check whether auto-completion gives your the member functions and constants by hitting the "Tab" key after typing the following line:

```
>>> ras.
```

If the "Tab" does not give you the HEC-RAS functions such as "Project_Open()" and "ShowRas()", that means a cache module does not exist and Python has no clue what it has.

On the other hand, when we call `win32.gencache.EnsureDispatch('RAS5x.HECRASController` it enforce the generation of the cache HEC-RAS Python module (if it does not exist already). Then, *pywin32* builds a COM object knowing all its functions and constants. This is called "static dispatch" or "early binding".

To explicitly generate the cache python module (without having to call the `gencache.EnsureDispatch()` function), one can use the `makepy.py` script in *pywin32*. Navigate to *pywin32*'s installation location and run the Python script by typing

```
$ python makepy.py
```

In the pop up window, select the library that you want to use and click OK (Figure 4.4). In this case, select the version of "HEC River Analysis System" you want (again, unfortunately different versions of HEC-RAS use the same name and therefor it is hard to distinguish; the user has to try and examine the generated Python files and the CLSID number).

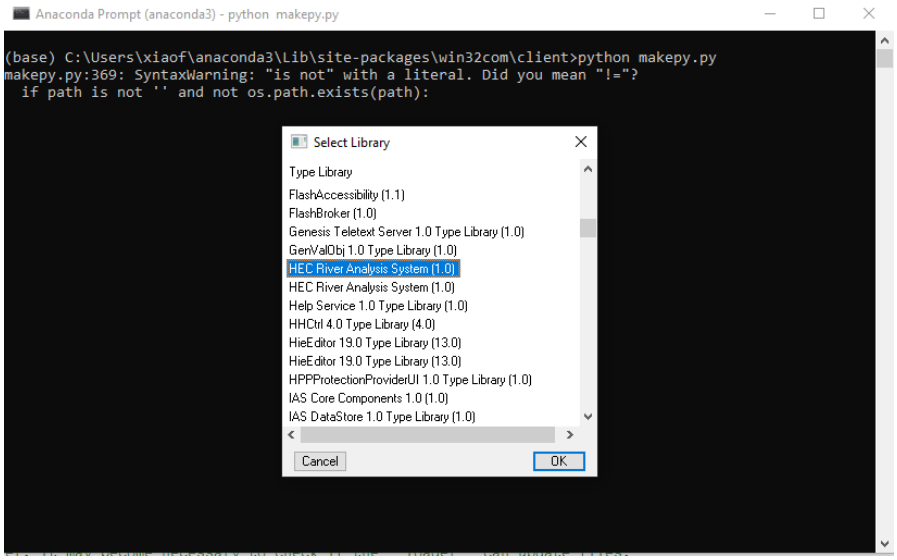


Figure 4.4 Run the "makepy.py" script to generate HEC-RAS Python module.

This step is similar to adding reference to "HEC River System Analysis" in MS Excel if VBA is used as the scripting language to interact with HEC-RAS (Figure 4.5).

In summary, to use Python to control HEC-RAS, one can simply use *pywin32* by calling some simple functions as shown in the example (Script 4.1). However, it

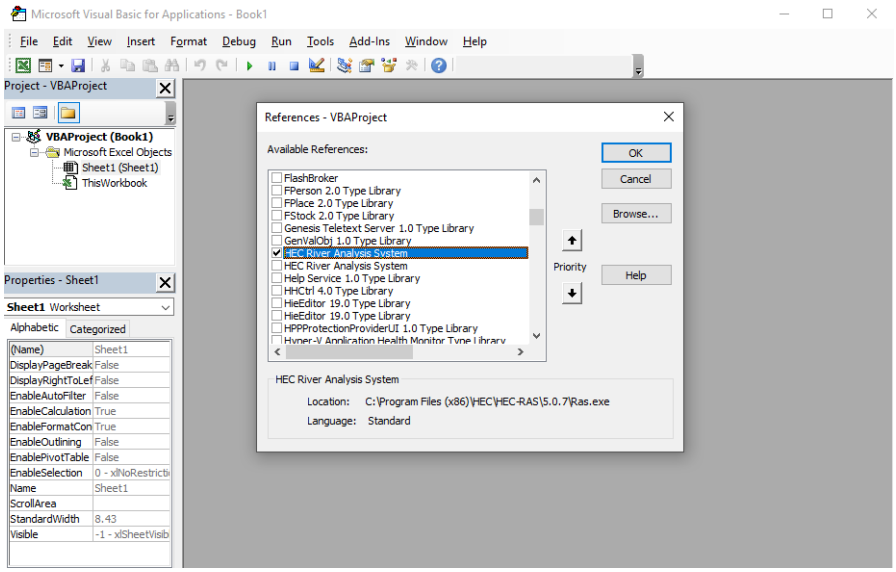


Figure 4.5 Add the HEC-RAS reference in MS Excel's VBA.

might be beneficial to know the inner workings and how all the parts play together. It helps identify problems and troubleshooting.

Option 2:

Option 3:

Chapter 5

SRH-2D Basics, Result Format, and I/O

(To be written)

This chapter describes how to use *pyHMT2D* to process SRH-2D results and demonstrate some typically applications, including format conversion, advanced calibration using machine learning and data-driven approaches.

5.1 SRHHYDRO file format

Mesh unit: specified in srhgeom. keyword is `GRIDUNIT` (case insensitive):

- US: `FOOT` or `FEET` (default)
- SI: `METERS`

In the entry `OutputFormat`, there are six options:

- `SRHN`: result at nodal (vertex) values
- `SRHC`: result at cell center values
- `TEC`: result in Tecplot format for nodal values
- `XMDF`: result in XMDF file for nodal values
- `XMDFC`: result in XMDF file for cell center values
- `Paraview`: result in VTK format for nodal values

5.2 Convert SRH-2D result to 3D

Chapter 6

HEC-RAS Basics, Result Format, and I/O

6.1 Background

HEC-RAS provides 2D modeling capability and its use has increased in the recent years.

A functionality missing is that the export of the HEC-RAS 2D results in a common format such that these results can be used for further analysis. One such example is the use of the 2D flow field to assess the fish passage design.

6.2 Basic steps to setup and run a HEC-RAS 2D case

The setup and run of a HEC-RAS 2D case are very straightforward. The GUI provided is relatively easy to use.

The basic steps are as follows:

- Create and save a new project
- Open RAS-Mapper and add terrain.
  - In this step, the terrain file is loaded and processes (Figure 6.1). It creates Terrain.hdf, Terrain.vrt, Stitch-TIN for merging rasters, and a projection file Projection.prj.
- Exit RAS-Mapper and go to the main window: select Edit » Geometric Data. You will see the terrain created in previous step
- Start to create 2D Flow Area.
- Create boundaries with the SA/2D Area BC lines tool. Note: for external boundaries, they have to be drawn close to the boundary faces and totally outside of the 2D area. No intersection allowed.
- Create breaklines to specify significant terrain features such as river banks, levees, etc.

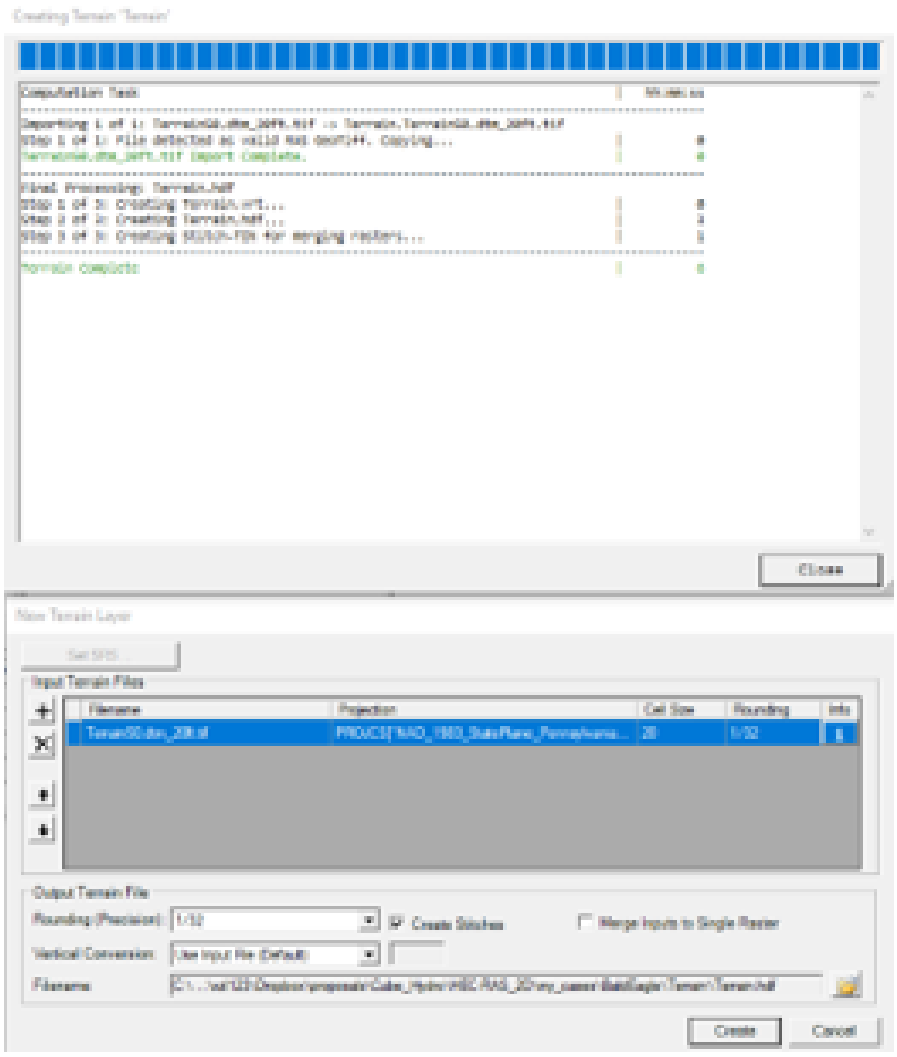


Figure 6.1 Adding terrain in RAS-Mapper

- Select and edit the 2D Flow area by right click and choose “Edit 2D Flow Area ...”
  - Specify the background mesh resolution deltaX and deltaY
  - Specify the breaklines to be enforced
  - Click ok to generate the mesh.

6.3 HEC-RAS files

The major references for this section is the HEC-RAS User Manual and this blog post.

When using HEC-RAS, a user can create files and the software itself also creates some file. Here is a list:

- .prj: project file, which contains current plan files, units, and project description
- .g01, .g02, ..., .g##: geometry file, which contains cross-sectional data, hydraulic structures and modeling approach data.
- .f01, .f02, ..., .f##: steady flow file, which contains profile information, flow data, and boundary conditions
  - For unsteady flow, .u## is flow file, which contains hydrographs and initial conditions, and other user defined flow options.
  - For quasi-unsteady flow (for sediment transport), .q## is the flow file
- .p01, .p02, ..., .p##: plan file, which contains a list of the associated input files, and all simulation options.
- .p01.rst, ..., p##.rst: a restart file which can be switched on/off in the Output Control Options window.

For unsteady flow analysis, there are also some "intermediate" files which are created by HEC-RAS at run-time. They are not essential for postprocessing.

- .c01: geometric pre-processor binary output file, which contains the hydraulic properties table, rating curves, and family of rating curves fro each cross-section, bridge, culvert, storage area, inline and lateral structure. These information is recalculated and the file is re-written when the geometry is changed.
- .b01: boundary condition file
- .bco01: unsteady flow log output file, for things like mass balance error
- .p01.blf: binary log file
- .IC.O01 initial condition binary file
- .O01, O02, ..., .O##: is the output file, which contains all the computed results from the associated plan. It is written in binary format and can only be read by the software.
  - For unsteady flow, a .dss file is generated as an output which contains time series data.

- .r01, .r02, ..., .r##: run file for steady flow analysis, which contains all the necessary input data for the RAS computational engine. It is created during the model execution and is not required for postprocessing.
  - For unsteady flow, the file extension is .x01, ..., .x##.
- .comp_msgs.txt: the computational message text file, which contains the message shown in the computation window. It is a good place to examine and trouble shoot simulation problems.
- .hyd01: the detailed computational level output file. It can switched on/off in the Unsteady Flow Analysis window.

When sharing a HEC-RAS model with others, one can submit only the necessary input files. The results and intermediate files are optional, without which they will have to rerun the model to get the results.

There are also special files associated with sediment transport and water quality analysis.

When cleaning a project folder, the following files should be kept and the others will be regenerated:

- .prj, .p##, .g##, .f##, .u##, .s##, .h##, and .w## files

For 2D models, the following is a list of extra files. Files with the hdf extension are in HDF5 format:

- .g##.hdf: geometry hdf file which contains all the geometric data (similar to the .g## file.
- .p##.hdf: plan HDF5 file which contains all the output and all the geometry input.
- .rasmapper file: which contains some basic information about layers, projections, and settings for RAS mapper.

To view the 2D model results in HDF5, one can use either the RAS-Mapper (Figure 6.2) or the HDFView software (Figure 6.3). RAS-Mapper comes with HEC-RAS. The HDFView software is an open source code which can be downloaded from the website of the HDF Group. HDFView is extremely useful to inspect and understand how HEC-RAS organizes its data.

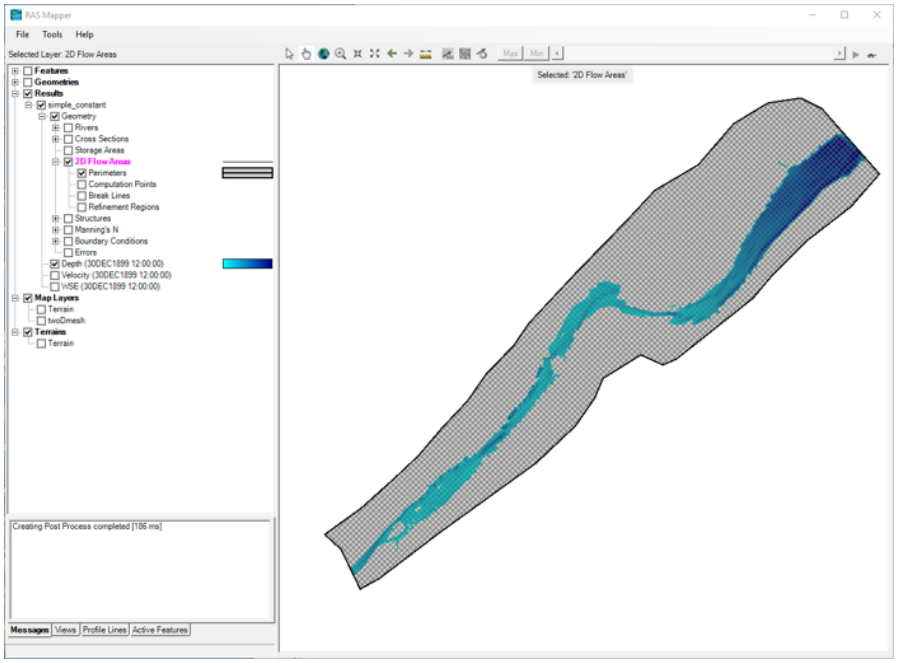


Figure 6.2 2D simulation results visualized in RAS-Mapper

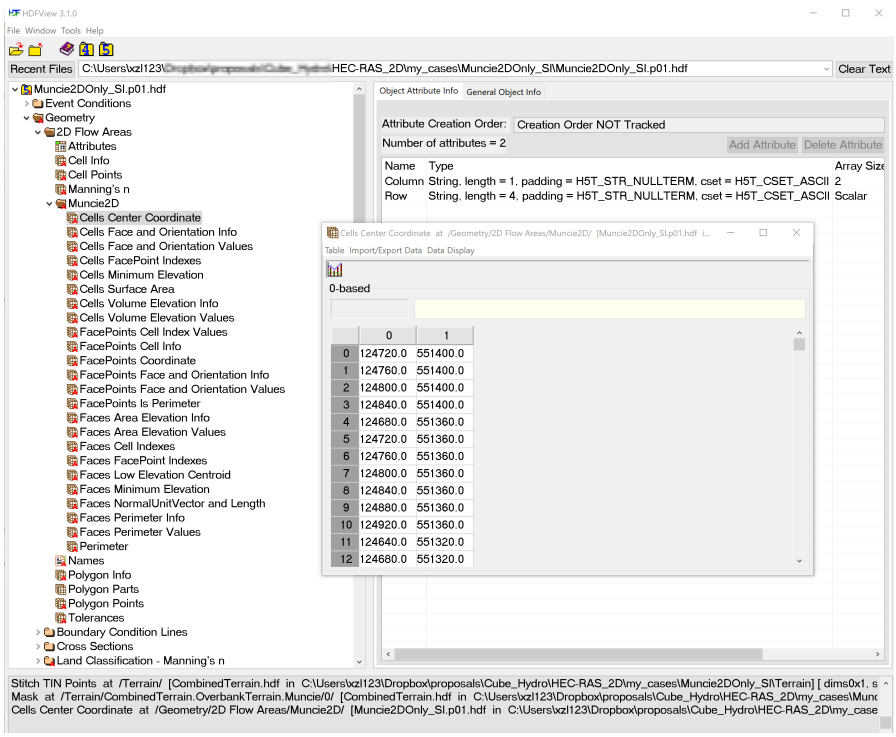


Figure 6.3 2D simulation results shown in HDFView

6.4 HEC-RAS results in HDF5 format

This section documents the findings on how HEC-RAS organizes its 2D results in HDF5 format. The findings were obtained through a simple 2D rectangular channel case where all the mesh information (cells, faces, face points, etc) can be easily tracked manual and compared with the data in HDFView. The sketch of the simple 2D case is shown in Figure 6.4. It consists of one 2D flow area with one upstream BC and one downstream BC. In total, there are 12 cells, 32 faces,and 21 face points.

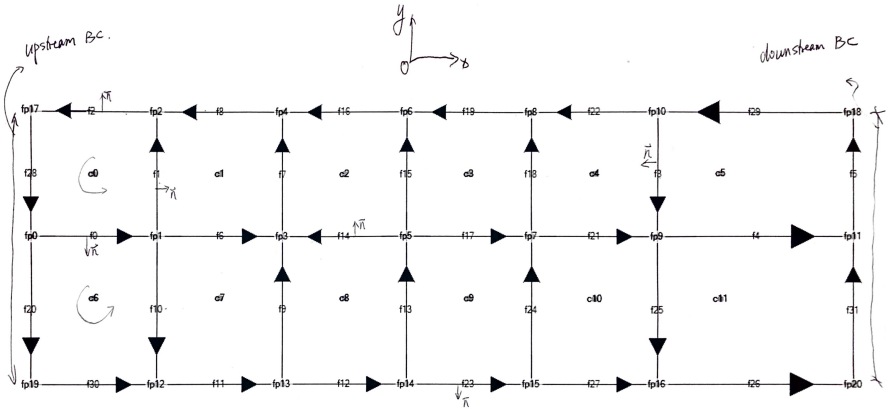


Figure 6.4 Sketch of the simple 2D case

HEC-RAS uses the finite-volume method (FVM) to solve the 2D depth-averaged shallow water equation. The mesh it uses is unstructured and each computational cell is a polygon. However, HEC-RAS allows a polygon to have at most eight faces (i.e., eight face points).

- `FacePoints Cell Index Values` and `FacePoints Cell Index Info`: for each face point, these two data sets record the cells who share the point. A face point can be shared by many cells. The two data sets also include the boundary "extra" cells.
- `FacePoints Face and Orientation Values`: stores all faces who use this face point. The sign "+" or "-" signifies the direction of the face to ("+") or away from ("-") the face point. For example, Figure 6.5 shows the values for `fp3`.
- `FacePoints Is Perimeter`: whether a face point is on the perimeter (Yes: -1, No:0)
- `Faces Cell Indexes`: stores the two neighbor cells who share the face. If the face is on boundary (perimeter), one cell is the ghost cell in the "expanded" domain.

For fp3

face	direction
6	1
2	-1
10	-1
8	1

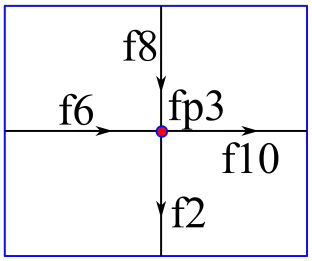
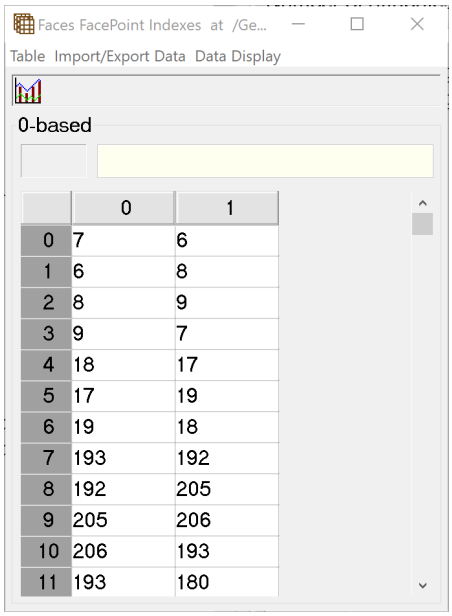


Figure 6.5 An example illustration of face point's face and orientation values (with `fp3` as an example)

- `Faces FacePoint Indexes`: shores the two ends (face points). Note that the order of the two points signifies the direction of the face. Each face corresponds to one row in the table as shown in Figure 6.6.



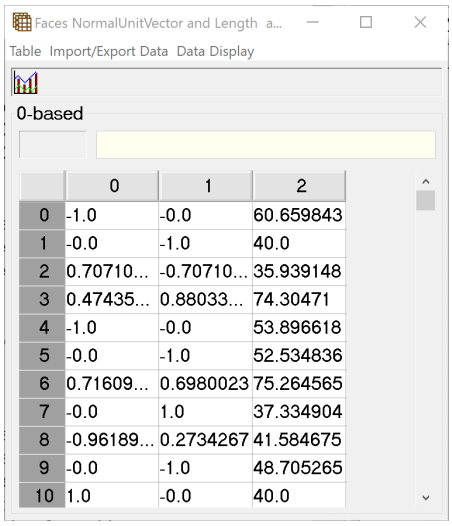
	0	1
0	7	6
1	6	8
2	8	9
3	9	7
4	18	17
5	17	19
6	19	18
7	193	192
8	192	205
9	205	206
10	206	193
11	193	180

Figure 6.6 The content of the `Faces FacePoint Indexes` table

- `Faces NormalUnitVector and Length`: stores the two components (n_x and n_y) of a face's unit normal vector ($\mathbf{n}$) and the face length (see Figure 6.7).

  Note the definition of the normal vector follows the right hand rule ($\mathbf{n} \times |p1 \rightarrow p2|$) as shown in Figure 6.9.

- `Faces Perimeter Info`: whether the face is on the perimeter (Yes: 1, No:0)



	0	1	2
0	-1.0	-0.0	60.659843
1	-0.0	-1.0	40.0
2	0.70710...	-0.70710...	35.939148
3	0.47435...	0.88033...	74.30471
4	-1.0	-0.0	53.896618
5	-0.0	-1.0	52.534836
6	0.71609...	0.6980023	75.264565
7	-0.0	1.0	37.334904
8	-0.96189...	0.2734267	41.584675
9	-0.0	-1.0	48.705265
10	1.0	-0.0	40.0

Figure 6.7 The content of the `Faces NormalUnitVector and Length` table

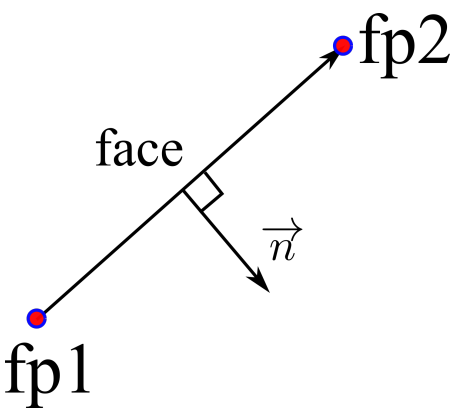
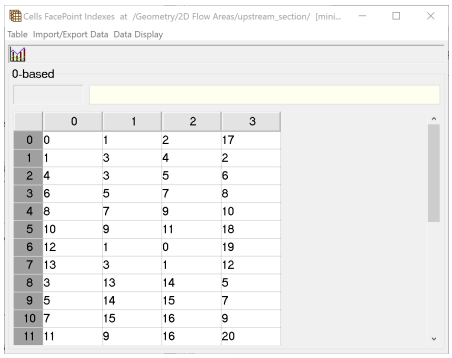


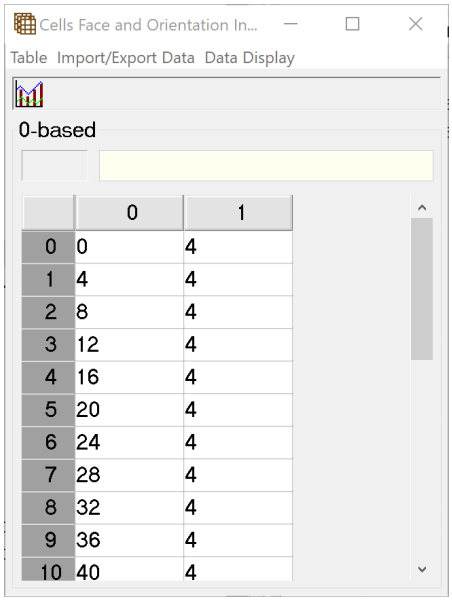
Figure 6.8 Definition of the face normal vector

- `Cells FacePoint Indexes`: for each cell, the corresponding row stores its face point indexes (Figure 6.6).
- `Cell Points`: the coordinates of the cell centers in the whole simulation domain (may consists of more than one 2D areas).
- `Cells Center Coordinate`: for the current 2D area, the coordinates of the cell centers (`Cell Points`), plus the extra points at the center of boundary faces.
- `Cells Face and Orientation Info`: store the number faces and the starting index of the faces in the face list (Figures 6.10 and 6.11).
- `Cells Face and Orientation Values`: this table works in conjunction with `Cells Face and Orientation Info` (see Figure 6.12).



	0	1	2	3
0	0	1	2	17
1	1	3	4	2
2	4	3	5	6
3	6	5	7	8
4	8	7	9	10
5	10	9	11	18
6	12	1	0	19
7	13	3	1	12
8	3	13	14	5
9	5	14	15	7
10	7	15	16	9
11	11	9	16	20

Figure 6.9 The content of the `Cells FacePoint Indexes` table



	0	1
0	0	4
1	4	4
2	8	4
3	12	4
4	16	4
5	20	4
6	24	4
7	28	4
8	32	4
9	36	4
10	40	4

Figure 6.10 The content of the `Cells Face and Orientation Info` table

- `Cells FacePoint Indexes`: stores the face point indexes of each cell (see Figures 6.13 and 6.14). Since cells may have different number of face points, the table's number of columns is the maximum number of face points for all cells. Those with less face points will be padded with -1.

The definition of boundaries:

- `External Faces`: it is not totally clear how this table is organized, but some of the columns are obvious. It is not clear how the column of `BC LineID` is defined and how the station is defined.
- `Polyline Info` and `Polyline Points`: only store the original points given by the user for the definition of the boundary lines.

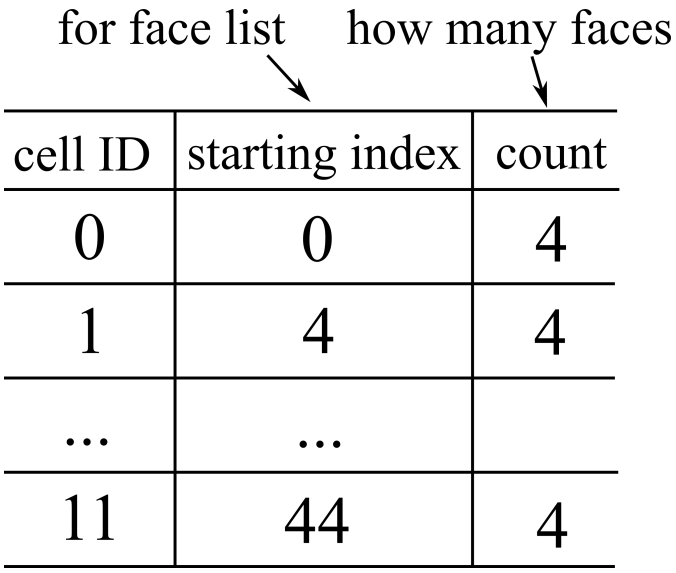


Figure 6.11 The `Cells Face and Orientation Info` table

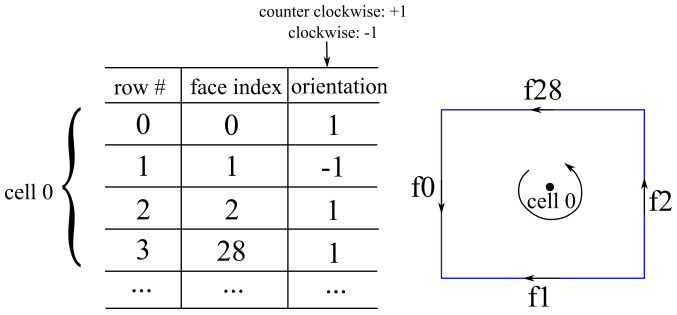


Figure 6.12 The `Cells Face and Orientation Values` table

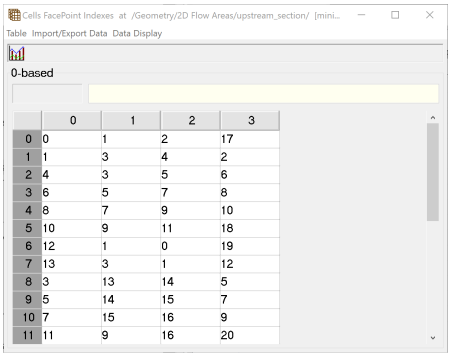


Figure 6.13 The `Cells FacePoint Indexes` table

		point number				
	cell ID	0	1	2	3	4
For all cells	0	0	1	2	17	
	1	1	3	4	2	
	...	...	...	...	...	
	11	11	9	16	20	
For all boundary faces	12	2	11	-1	-1	
	13	11	18	-1	-1	
	...	...	...	...	...	
	27	20	11	-1	-1	

face point IDs

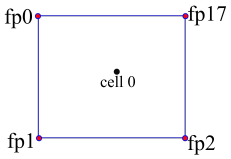
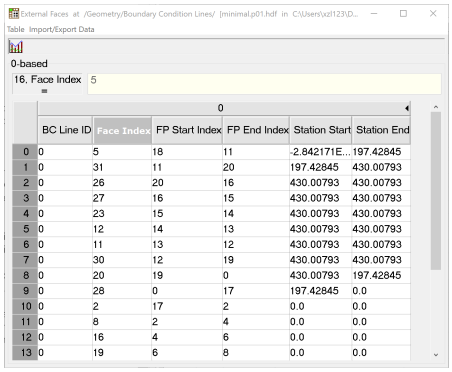


Figure 6.14 The `Cells FacePoint Indexes` table



	0					
	BC Line ID	Face Index	FP Start Index	FP End Index	Station Start	Station End
0	0	5	18	11	-2.842171E...	197.42845
1	0	31	11	20	197.42845	430.00793
2	0	26	20	16	430.00793	430.00793
3	0	27	16	15	430.00793	430.00793
4	0	23	15	14	430.00793	430.00793
5	0	12	14	13	430.00793	430.00793
6	0	11	13	12	430.00793	430.00793
7	0	30	12	19	430.00793	430.00793
8	0	20	19	0	430.00793	197.42845
9	0	28	0	17	197.42845	0.0
10	0	2	17	2	0.0	0.0
11	0	8	2	4	0.0	0.0
12	0	16	4	6	0.0	0.0
13	0	19	6	8	0.0	0.0

Figure 6.15 The `External Faces` table

There are other nonessential data such as `Cells Minimum Elevation`, `Cells Surface Area`, `Cells Volume Elevation Info` and `Cells Volume Elevation Values`.

6.5 A python script to extract HEC-RAS 2D data

A python script has been developed to extract HEC-RAS 2D result data mainly from the HDF5 file. This script does the following:

- read the HDF5 file with the `h5py` library. It is found that the main data structure in HEC-RAS 2D HDF5 file is similar. But discrepancies do exist. It depends what data the user chooses to store as some of them are optional. Sometime the names in the tree structure are slightly different. As such, the user of this script needs to make some adjustment if necessary.
- write out the results in VTK format with the `meshio` library. Some notes:
  - `meshio` only supports `UNSTRUCTURED\_GRID` VTK format and its support for polygonal cells is very recent. Thus, the version of `meshio` should be $\geq$ 2.3.7. In Windows with Anaconda, the installation is as follows:

```
$ pip install -U meshio
```

Even with the latest version, it is found that there might be some bug in the code in the part the writing of the mesh is called. The problem lies in the following part (Lines 8 and 17) of the code in the `vtk_io.py` file of the library:

```
1 def write(filename, mesh, write_binary=True):
2     if mesh.points.shape[1] == 2:
3
4     ...
5
6         if mesh.point_data:
7             for name, values in mesh.point_data.items():
8                 if values.shape[1] == 2:
9                     logging.warning(
10                        "VTK requires 3D vectors, but 2D vectors given. "
11                        "Appending 0 third component to {}.".format(name)
12                    )
13                    mesh.point_data[name] = pad(values)
14        if mesh.cell_data:
15            for t, data in mesh.cell_data.items():
16                for name, values in data.items():
17                if values.shape[1] == 2:
18                    logging.warning(
19                         "VTK requires 3D vectors, but 2D vectors given. "
20                         "Appending 0 third component to {}.".format(name)
21                    )
22    ...
```

For vectors (such as coordinates and velocity), `VTK` requires 3D data, i.e., with three components. If only two components are given in the data `values`, `meshio` will try to pad the third component with zero. This is what the above code does. However, if `values` is not vector, for example it is a scalar, then the access of `values.shape[1]` will throw an "out of bound" exception.

The solution to get around this is add a conditional before the access of `values.shape[1]` as shown below. In this way, the padding is only applied if the data have two columns (meaning two components for a vector).

```
def write(filename, mesh, write_binary=True):
    if mesh.points.shape[1] == 2:

        def pad(array):
            return numpy.pad(array, ((0, 0), (0, 1)), "constant")

        logging.warning(
            "VTK requires 3D points, but 2D points given. "
            "Appending 0 third component."
        )
        mesh.points = pad(mesh.points)

        if mesh.point_data:
            for name, values in mesh.point_data.items():
                if values.ndim == 2:
                    if values.shape[1] == 2:
                        logging.warning(
                            "VTK requires 3D vectors, but 2D vectors given.
                            "Appending 0 third component to {}.".format(name
                        )
                        mesh.point_data[name] = pad(values)
        if mesh.cell_data:
            for t, data in mesh.cell_data.items():
                for name, values in data.items():
                    if values.ndim == 2:
                        if values.shape[1] == 2:
                            logging.warning(
                                "VTK requires 3D vectors, but 2D vectors giv
                                "Appending 0 third component to {}.".format
                            )
```

Typically the python package `meshio` is installed here. Before this fix is adopted in the official release, one needs to go here and manually make the change.

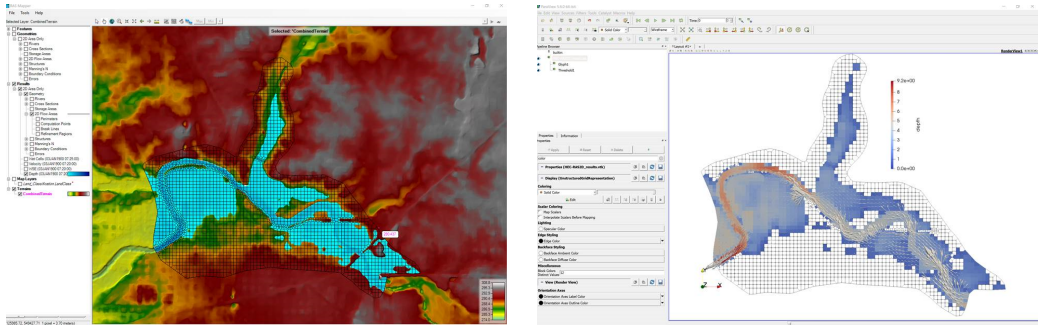
`C:\Users\xiaof\AppData\Local\Continuum\anaconda3\Lib\site-packages\meshio`

– As of now, only one 2D area is tested and results at only the last time step are exported. The script needs to be adjusted if multiply 2D areas present and more time steps need to be export, both of which are not difficult.

– Useful data in HEC-RAS 2D results are cell data (`Depth` and `Water Surface`) and point data (`Node X Vel` and `Node Y Vel`). With VTK (and in par-

aview, cell data and point data can be interpolated to points and cells, respectively.

After the data are exported in VTK format, they can be loaded into ParaView or used for other purposes. Figure B.1 is a comparison between RAS-Mapper and ParaView showing the same result.



(a) RAS-Mapper **(b)** Paraview

Figure 6.16 Comparison between RAS-Mapper and Paraview showing the same HEC-RAS 2D result

6.5.1 Changes to meshio package in version 4.2

It seems the Python package meshio has made some changes in recent versions. I have to make the following changes to the meshio package.

- "_common.py" file:

```
def raw_from_cell_data(cell_data):
    return {name: numpy.concatenate([value]) for name, value in cell_data.items(
    #return {name: numpy.concatenate(value) for name, value in cell_data.items()
```

- "_vtk.py": need to add the output of field_data (for time information)

```
    with open_file(filename, "wb") as f:
        f.write(b"# vtk DataFile Version 4.2\n ")
        f.write(f"written by meshio v{__version__}\n ".encode("utf-8"))
        f.write(("BINARY\n " if binary else "ASCII\n ").encode("utf-8"))
        f.write(b"DATASET UNSTRUCTURED_GRID\n ")
```

```
7         # write field data (added by XL)
8         _write_field_data(f, mesh.field_data, binary)
9
10        # write points and cells
11        _write_points(f, points, binary)
12        _write_cells(f, mesh.cells, binary)
13
```

6.5.2 Notes about RAS 2D

- It seems HEC-RAS 2D (up till v6 beta) does not export "Face Profile" in the geometry and result HDF files, although the face profile can be plotted in RAS Mapper. Need to externally extract the face profile (approximately as what HEC-RAS does, something like tolerance etc.)
- HEC-RAS 2D (up till v6 beta) can have only one Manning's n value per face although Face's Manning's n is exported as a function of elevation. HEC-RAS 2D User Manual states the following (Page 2-22):

  For this version of HEC-RAS, the program will select only one Manning's n value for the entire cell face. Future versions of HEC-RAS will allow for multiple Manning's n values across each cell face. So, this is a limitation right now when using large cell sizes.
- In HEC-RAS, we need to export velocity at nodes in order for pyHMT2D to extract. To do that, in HEC-RAS's "Unsteady Flow Analysis", select "Options" and then "HDF5 Write Parameters". Make sure to check the box "Write velocity data at the face node locations in 2D meshes".

6.5.3 Some useful packages

- convert GIS shape file to VTK: https://github.com/paulo-herrera/PyGTV

6.6 Colormaps in RAS-Mapper

RAS-Mapper comes with some predefined colormaps for the contours of depth, WSE, velocity, etc. If desired, one can define the same colormap in ParaView so it can produce similar contour figures.

Colormap in ParaView can be defined and imported through a XML (Extensible Markup Language) file. The definition and examples of ParaView colormaps can be found at

https://www.paraview.org/Wiki/Colormaps

The colormap definition in RAS-Mapper can be found by first selecting a data layer, right clicking and select “Layer Properties”. In the popup window, one can access the colormap by following the steps highlighted in Figure 6.17. Note the definition and the RGB values for each colormap.

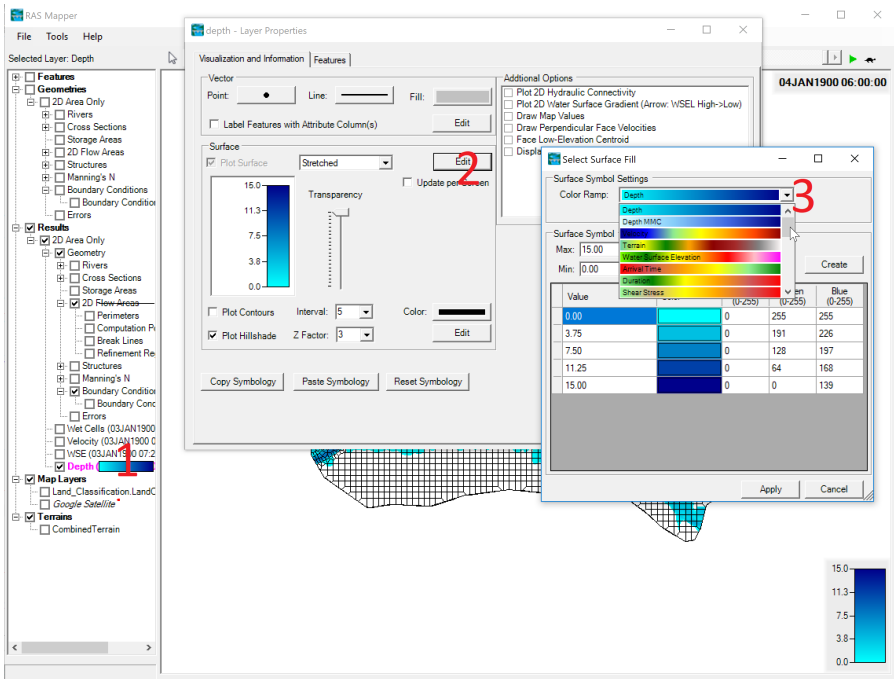


Figure 6.17 Colormap in RAS-Mapper

To define equivalent colormap for ParaView. One can manually edit the XML file. An automated process is to do it through a script. An example python script is written as follows (indeed a ipython notebook `generate_HEC_RAS_colormap.ipynb` was created):

```
import xml.etree.cElementTree as ET
import numpy as np

#add new line after each element
def indent(elem, level=0):
    i = "\n " + level*"   "
    if len(elem):
        if not elem.text or not elem.text.strip():
            elem.text = i + "   "
        if not elem.tail or not elem.tail.strip():
            elem.tail = i
```

```
12          for elem in elem:
13              indent(elem, level+1)
14          if not elem.tail or not elem.tail.strip():
15              elem.tail = i
16      else:
17          if level and (not elem.tail or not elem.tail.strip()):
18              elem.tail = i
19
20 #HEC-RAS 2D colormap for depth
21 values = np.array([0, 3.75, 7.5, 11.25, 15])
22 r=np.array([0,0,0,0,0])/255.0
23 g=np.array([255,191,128,64,0])/255.0
24 b=np.array([255,226,197,168,139])/255.0
25
26
27 root = ET.Element("ColorMaps")
28
29 ColorMap = ET.SubElement(root, "ColorMap", name="HECRAS_Water_Depth", space="RGB")
30 for i in range(5):
31     ET.SubElement(ColorMap, "Point x=\" "  + str(values[i]) + "\"  o=\"1\" " \^^M
32
33 tree = ET.ElementTree(root)
34
35 indent(root)
36
37 # writing xml
38 tree.write("HEC_RAS_colormaps.xml", encoding="utf-8", xml_declaration=True)
39
40 print("Done!")
41
```

In this script, the `ElementTree` package is used for `XML`.

Appendix A

Python Related

A.1 Virtual Environment

Many Python applications require certain versions of dependent packages. There can easily be conflicts among different applications. To solve this problem, virtual environment can be used. Multiple virtual environments can be created and one of them can be activated. There are several ways to create and management virtual environment, for example `virtualenv`, `venv`, and `conda`. If the Anaconda Python distribution is used, `conda` is the preferred way.

The following is a simple demonstration on how to use `conda`. But before `conda` is used, it may be beneficial to update the `conda` environment by

```
$ conda update conda
```

- check the list of available Python versions:

  ```
  $ conda search python
  ```

  Make sure the version of Python that you want to create the virtual environment with is available.

- create the new virtual environment:

  ```
  $ conda create -n vename python=x.x anaconda
  ```

  Here, `vename` is the name of the created virtual environment, `x.x` is the Python version. For example,

  ```
  $ conda create -n py37 python=3.7 anaconda
  ```

  will create a new virtual environment named `py37` with Python 3.7.

- view the list of available virtual environments:

  ```
  $ conda info -e
  ```

- activate the virtual environment:

  ```
  $ conda activate py37
  ```

  which will activate the virtual environment named `py37` and modify `PATH` and other environment variables to point to the current active virtual environment.

- deactivate the virtual environment:

  ```
  $ conda deactivate
  ```

  which will deactivate the current virtual environment and fall back to the `base` virtual environment.

- delete a virtual environment:

  ```
  $ conda remove -n py37 --all
  ```

  which will delete the virtual environment named `py37`.

A.2 Python Package Installation

Python package installation can be done through different ways.

pip

`pip` is the package installer for Python. It can install packages from the Python Package Index (PyPI) and other indexes.

- `pip`'s website: https://pypi.org/project/pip/
- To check information about `\pip`:

  ```
  $ pip -V  (note: this will show the version of pip)
  $ pip -h  (note: this will show the help information about pip)
  ```

- To search for a package:

  ```
  $ pip search package_name
  ```

- To install a package:

  ```
  $ pip install package_name
  ```


- To upgrade a package:

  ```
  $ pip install package_name --upgrade
  ```

- To install a package of a specific version:

  ```
  $ pip install -Iv package_name==version_number
  ```

  Here, `-I` means ignore package already installed.

- To list all installed packages in the current virtual environment:

  ```
  $ pip list
  ```

- To show information about a particular package

  ```
  $ pip show package_name
  ```

- To uninstall a package:

  ```
  $ pip uninstall package_name
  ```

conda

We already used `conda` in previous section to create virtual environment. Indeed, `conda` is a package management system and environment management system. It not only manages Python packages, but also virtual environments.

- `conda`'s website: https://docs.conda.io/en/latest/
- To check `conda`'s information:

  ```
  $ conda -V (note: this will show its version information)
  $ conda -h (note: this will show its help information)
  ```

- To install a package:

  ```
  $ conda install package_name
  ```

  Whether to use `pip` or `conda` mainly depends on:

- availability of your desired packages (and their different versions) through `pip` and `conda`,
- personal preference.

A.3 Common problems and errors

- How to deal with the error message “win32com.gen_py has no attribute 'CLSIDToPackageMap' ”?

  Delete the content in the `C:\Users\<username>\AppData\Local\Temp\gen_py`.

Appendix B

File Format and Processing

B.1 Terrain and bathymetric data

There are two formats for terrain and bathymetric data: raster and vector. Examples of raster format are TIFF, GeoTIFF.

B.1.1 TIFF and GeoTIFF

Tagged Image File Format (TIFF or TIF) is a file format for storing raster images. A TIFF file has information about an image and its data, which include size, definition, image compression, etc. The image contained in a TIFF file can use lossy compression or lossless compression, or no compression at all to ensure the image quality.

GeoTIFF is for built on top of TIFF with additional georeferencing information, such as map projection, coordinate systems, ellipsoids, and datum.

BigTIFF: HEC-RAS's re-samples and rounds the terrain data and create a new TIFF file inside the "Terrain" directory. It seems the created new TIFF is in the BigTIFF format. Earlier versions of GDAL ($\leq$ 2.3.3) can not read BigTIFF format. The user can either install a later version of GDAL (>3.0) or use some tools to convert the BigTIFF file to regular GeoTIFF file, for example the "tifffile" library at https://pypi.org/project/tifffile/.

In HEC-RAS, it is also important to pay attention to the rounding precision when importing terrain data. HEC-RAS re-samples the terrain data and rounds the elevation. It is recommended that the rounding precision to be compatible with the terrain elevation precision. Otherwise, the resulted terrain will look like Figure B.1a, instead of Figure B.1b.

B.1.2 Artificial Terrain Creation

- https://github.com/caseman/noise/blob/master/examples/2dtexture.py

(a) Rounding (Precision) = 1/128

(b) Rounding (Precision) = 1/1000

Figure B.1 Effect of rounding precision on the imported (re-sampled) terrain data

B.2 Coordinate systems, projections, and transformations

Spatial data are defined in specific coordinate systems, both horizontally (to place a point on the surface of the earth) and vertically (to place a point in height).

There are three different types of horizontal coordinate systems (named "Coordinate Reference System (CRS)" in `GDAL`'s `OGRSpatialReference` class):

- geographic coordinate system (GCS): uses longitudinal (x) and latitude (y) coordinates in degrees. For example, the WGS84 - World Geodetic System 1984 is often used in GPS.
- projected coordinate system (PCS): uses linear (length) measurements for the x and y coordinates. PCS needs an underlying GCS and a definition for the projection. For example, the Universal Transverse Mercator (UTM) coordinate system is one kind of PCS. For instance, the PCS "WGS 84 / UTM zone 18N" defines UTM zone 18 in northern hemisphere with an underlying GCS of WGS 84.
- local coordinate system: coordinates in a locally defined system with a false origin.

There are two different types of vertical coordinate systems:

- gravity based: relative to a mean sea level
- Ellipsoidal coordinate system: relative to a spheroidal or ellipsoidal surface approximating the earth.

Projection is the process of mathematically transforming the coordinate system for the earth onto a flat surface. Different projection methods have been proposed with different goals, such as preserving shape, distance, area, or direction.

B.2.1 UTM

The UTM system divides the Earth into 60 zones, each 6 degrees of longitude in width. Figure B.2 shows the UTM zones for the U.S. The location of a point in UTM coordinate system is in the form of a UTM zone number and the easting and northing coordinates in that zone. The coordinate origin for each UTM zone is at the intersection of the zone's central meridian and the equator. In fact, to avoid negative easting coordinate, a constant of 500,000 meters is added to the easting, which in effect puts the origin of each UTM zone at easting 500,000 meters.

Fro a point in the northern hemisphere, it has a northing coordinate of 0 at the equator and increases to about 9,300,000 meters at the northern end.

As an example, the center of the Penn State Beaver Stadium is at "18N 259101 4521837", which means it is in UTM zone 18 in the northern hemisphere with an easting coordinate of 259101 and a northing coordinate of 4521837.

In degrees, minutes, and seconds: 40°48'43.95"N, 77°51'22.04"W

In decimal degrees: latitude = 40.81220833333333, and longitude = -77.85612222222221.

European Petroleum Survey Group, known as EPSG, is a public registry of geodetic systems. Each entry in the dataset has an EPSG code between 1024 and 32767 and a standard text file called "well-known text (WKT)". Many GIS libraries support EPSG to identify spatial reference systems and their coordinate reference systems and projections.

Some example EPSG coordinate systems of interest:

- EPSG:4326, WGS (World Geodetic System) 84, latitude/longitude coordinate system based on the Earth's center of mass, used by the Global Positioning System. This is used for the whole world.
- EPSG:3857, Web Mercator projection. It is used for display by many web-based mapping tools, including Google Maps.
- EPSG:32128, NAD83 Pennsylvania North (unit = meter)
- EPSG:2271, NAD83 Pennsylvania North (unit = foot)
- EPSG:26918, NAD83 / UTM zone 18N
- EPSG:32618, WGS 84 / UTM zone 18N. Same UTM zone 18N, but with WGS 84 datum.

More information can be found at http://epsg.io.

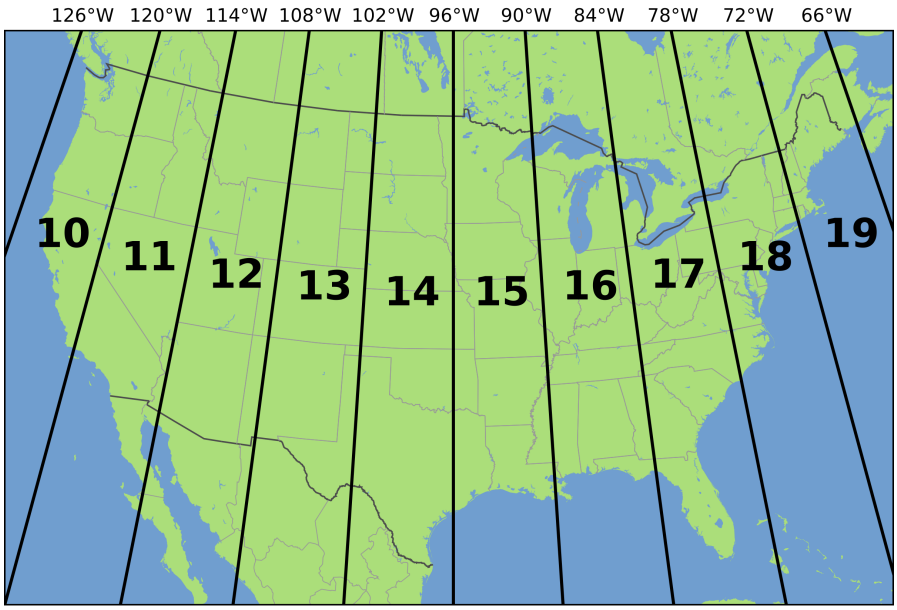


Figure B.2 UTM zones for the U.S. (source: wikipeida)

B.3 Python libraries for georeferenced data

B.3.1 GDAL

The Geospatial Data Abstraction Library (`GDAL`) is a computer library for reading and writing geospatial data in both raster and vector formats. Because HEC-RAS uses the BigTiff format for terrain data and the support of BigTiff is only in recent versions of `GDAL`, it is important to install the correct and recent version of `GDAL`. For example, in a Python 3.7 virtual environment, if `GDAL` v2.3.3 is installed, an error will occur complaining about the BigTiff support.

One can use `pip` or `conda` to install recent version of `GDAL`, e.g, 3.2. However, if there are errors reported in the installation, one can use some pre-compiled wheels, for example on this website: https://www.lfd.uci.edu/ gohlke/pythonlibs/. Download a proper version, then

```
$ pip install GDAL-3.2.2-cp37-cp37m-win_amd64.whl --user
```

An example Python code using `GDAL` is as follows:

```
1 def create_bathymetry_GeoTiff(nx, ny, geoTiffFileName,GCSName,):
2     """
3     create a bathymetry GeoTiff
```

```
4
5       :return:
6       """
7
8       driver = gdal.GetDriverByName( 'GTiff' )
9       dst_filename = 'x_tmp.tif'
10      dst_ds=driver.Create(dst_filename,nx,ny,1,gdal.GDT_Float32)
11
12      dst_ds.SetGeoTransform([444720, 30, 0, 3751320, 0, -30])
13
14      srs = osr.SpatialReference()
15      srs.ImportFromEPSG(32128)  # NAD83 Pennsylvania North (unit = meter)
16
17      dst_ds.SetProjection(srs.ExportToWkt())
18      raster = np.zeros((ny, nx))
19
20      for ix in range(nx):
21          for iy in range(ny):
22              raster[iy,ix] = np.sin(ix/nx*2*np.pi)
23              #print(raster[ix,iy])
24
25      dst_ds.GetRasterBand(1).WriteArray(raster)
26
27      # Once we're done, close properly the dataset
28      dst_ds = None
29
30
```

`SetGeoTransform` sets the parameters for the affine geotransformation to describe the relationship between raster position (pixel/line coordinates) and georeferenced coordinates. The affine transformation needs six coefficients, say in a list `GT[0:5]`. The transformation is then

$$Xgeo = GT[0] + Xpixel * GT[1] + Yline * GT[2] \quad \text{(B.1)}$$

$$Ygeo = GT[3] + Xpixel * GT[4] + Yline * GT[5] \quad \text{(B.2)}$$

where ($Xgeo$, $Ygeo$) are the georeferenced coordinates, ($Xpixel$,$Yline$) are the raster position in the image. For a north up image, the $GT[2]$ and $GT[4]$ rotation coefficients are zero, and the $GT[1]$ is pixel width, and $GT[5]$ is pixel height. The ($GT[0]$,$GT[3]$) position is the georeferenced location of the top left pixel of the raster image.

It is necessary to know that the pixel/line coordinates in a raster in GDAL are from (0.0,0.0) at the top left corner of the top left pixel to (width_in_pixels,

height_in_pixels) at the bottom right corner of the bottom right pixel. The pixel/line location of the center of the top left pixel is then (0.5,0.5).

How to convert VRT file format to GeoTiff?

VRT is the short name for GDAL's Virtual Format. It allows the composition of a "virtual" GDAL dataset from other sources with optional transformations. As such, VRT is often used as a container with links to other datasets without actually including them. The VRT file is in fact an XML file with the extension of ".vrt".

Some hydraulic models, such as HEC-RAS, can use VRT as its terrain data. For example, in its example "Muncie", there exist the following files:

- `TerrainWithChannel.vrt`
- `TerrainWithChannel.muncie_clip.tif`
- `TerrainWithChannel.ChannelOnly.tif`

The content of the `TerrainWithChannel.vrt` file is as follows:

```
<VRTDataset rasterXSize="7892" rasterYSize="4538">
  <SRS>PROJCS["NAD_1983_StatePlane_Indiana_East_FIPS_1301_Feet",GEOGCS["NAD83",DATUM[
  <GeoTransform> 3.8497784867792000e+005, 5.0000000000000000e+000, 0.000000000000000
  <VRTRasterBand dataType="Float32" band="1">
    <Metadata>
      <MDI key="STATISTICS_MAXIMUM">1013.15625</MDI>
      <MDI key="STATISTICS_MEAN">951.21268449051</MDI>
      <MDI key="STATISTICS_MINIMUM">898.90625</MDI>
      <MDI key="STATISTICS_STDDEV">15.679383378553</MDI>
    </Metadata>
    <NoDataValue>-9.99900000000000E+003</NoDataValue>
    <ColorInterp>Gray</ColorInterp>
    <Histograms>
      <HistItem>
        <HistMin>898.90625</HistMin>
        <HistMax>1013.15625</HistMax>
        <BucketCount>256</BucketCount>
        <IncludeOutOfRange>1</IncludeOutOfRange>
        <Approximate>0</Approximate>
        <HistCounts>142|557|635|524|411|719|542|507|1307|641|1857|1234|1599|4555|3133
      </HistItem>
    </Histograms>
    <ComplexSource>
      <SourceFilename relativeToVRT="1">TerrainWithChannel.ChannelOnly.tif</SourceFil
      <SourceBand>1</SourceBand>
```

```
26        <SourceProperties RasterXSize="1891" RasterYSize="1155" DataType="Float32" Bloc
27        <SrcRect xOff="0" yOff="0" xSize="1891" ySize="1155" />
28        <DstRect xOff="3846" yOff="1168" xSize="1891" ySize="1155" />
29        <NODATA>-9999</NODATA>
30      </ComplexSource>
31      <ComplexSource>
32        <SourceFilename relativeToVRT="1">TerrainWithChannel.muncie_clip.tif</SourceFil
33        <SourceBand>1</SourceBand>
34        <SourceProperties RasterXSize="7892" RasterYSize="4538" DataType="Float32" Bloc
35        <SrcRect xOff="0" yOff="0" xSize="7892" ySize="4538" />
36        <DstRect xOff="0" yOff="0" xSize="7892" ySize="4538" />
37        <NODATA>-9999</NODATA>
38      </ComplexSource>
39    </VRTRasterBand>
40 </VRTDataset>
```

At the end of the VRT file, it can be observed that there are two "ComplexSource" entries, which are the two external data sources in the "GeoTiff" format. The order of these source data files matters. However, it matters in an opposite way in HEC-RAS and GDAL. In HEC-RAS, the top files get priority when the terrain is interpolated onto the computational mesh. On the other hand, in GDAL, the bottom files get priority.

Datasets in a VRT file can be converted into one composite "GeoTiff" using GDAL's "gdal_translate" command line tool. If your GDAL is installed with Anaconda, open "Anaconda Prompt" with the proper Python virtual environment. Then, use the command, for example

```
$ gdal_translate -co compress=LZW TerrainWithChannel.vrt composite_terrain.tif
```

This will create a new GeoTiff file named `composite_terrain.tif` using the compression algorithm "LZW". Compression is used to reduce the size of the generated GeoTiff file. However, one should be mindful of characteristics of different compression algorithms. GDAL supports many compression algorithm options, for example, JPEG, LZW, PACKBITS, DEFLATE, LZMA, ZSTD, NONE. The details can be found here:

https://gdal.org/drivers/raster/gtiff.html

There are two types of compression: lossless (original data values preserved) and lossy (accuracy degradation). Examples of lossless comperssion are LZW, DEFLATE, and PACKBITS. It is recommended to use lossless compression for terrain data. Example of lossy compression is JPEG.

In GDAL, there are also some other options for each compression algorithm which can affect the compression efficiency and the file size. For example, with LZW and DEFLATE, one can do

```
$ gdal_translate -co compress=LZW -co predictor=2 TerrainWithChannel.vrt composite_
```

which uses a predictor level of 2. This is especially useful if the terrain is changing smoothly and the algorithm only records the inter-pixel differences, instead of their absolute values. This can potentially reduce the GeoTiff file size drastically. For more information about available options for each GDAL driver, use the following command for example to get information about GeoTiff:

```
$ gdalinfo --format GTIFF
```

This command can also be used to get detailed information, such as projection, compression, units, origin, and pixel size, about a GeoTiff file, for example:

```
$ gdalinfo TerrainWithChannel.ChannelOnly.tif
```

To compose a VRT file from multiple GeoTiff source files, use the `gdalbuildvrt` command:

```
$ gdalbuildvrt result.vrt -input_file_list myRasterList.txt
```

where the `myRasterList.txt` contains a list of GeoTiff files, such as

```
file1.tif
file2.tif
file3.tif
```

Due to the difference in the priority assigned to file order in VRT file, it is important to make sure the priority terrain files are located at the bottom (reverse the order in HEC-RAS) and then use the `gdal_translate` command. In the example above, the order needs to be changed as follows such that the channel bathymetry has priority over the terrain.

```
1  <VRTDataset rasterXSize="7892" rasterYSize="4538">
2      <ComplexSource>
3
4        ...
5
6      <ComplexSource>
7        <SourceFilename relativeToVRT="1">TerrainWithChannel.muncie_clip.tif</SourceFil
8        <SourceBand>1</SourceBand>
9        <SourceProperties RasterXSize="7892" RasterYSize="4538" DataType="Float32" Bloc
10       <SrcRect xOff="0" yOff="0" xSize="7892" ySize="4538" />
11       <DstRect xOff="0" yOff="0" xSize="7892" ySize="4538" />
12       <NODATA>-9999</NODATA>
```



```
13      </ComplexSource>
14        <SourceFilename relativeToVRT="1">TerrainWithChannel.ChannelOnly.tif</SourceFil
15        <SourceBand>1</SourceBand>
16        <SourceProperties RasterXSize="1891" RasterYSize="1155" DataType="Float32" Bloc
17        <SrcRect xOff="0" yOff="0" xSize="1891" ySize="1155" />
18        <DstRect xOff="3846" yOff="1168" xSize="1891" ySize="1155" />
19        <NODATA>-9999</NODATA>
20      </ComplexSource>
21    </VRTRasterBand>
22 </VRTDataset>
```

In fact, HEC-RAS (RAS Mapper) uses another file in the HDF format to identify all GeoTiff files in the terrain layer and their priority order (not the order in the VRT file). Like the VRT file, the HDF file itself does not contain the terrain data.

Appendix C

VTK

C.1 Install VTK

C.1.1 Windows

To use `pyHMT2D`, you only need the Python-combined version of VTK. See Section 2.1.2 for details.

C.1.2 Linux

The installation of Python-combined version of VTK is the same as in Windows. However, one may also need to install the VTK library for other purposes. For example, if 2D hydraulic model results are extruded to 3D and imported into 3D CFD code such as OpenFOAM, the VTK library needs to be installed so its functions can be called within C++ or other programming languages.

Depending on what Linux system is used, the detailed steps may vary. On a public shared Linux system, such as a HPC cluster, VTK might have already been installed. You just need to figure out where it is installed.

To check a package

```
$ apt-cache search vtk
```

will search all packages that matches the work "vtk". There will be many matches. The ones we need here are the library and the development header files.

```
$ sudo apt-get install libvtk7.1
```

which will install the vtk library version 7.1. To check where the library is installed on your computer, do

```
$ dpkg -L libvtk7.1
```

By default, the library will be installed in `/usr/lib`. For example, on my computer it is installed in

```
/usr/lib/x86_64-linux-gnu
```

Note that the above step will only install the shared library files, not the headers. Headers files, e.g., "vtkPoints.h", are needed when compile a source code which includes them, for example the `mapSRH2DToFoam` too. To install the header files,

```
$ sudo apt-get install libvtk7-dev
```

By default, all the header files will be installed in

```
/usr/include/vtk-7.1
```

You can add this location to the `PATH` environment variable by

```
export PATH=/usr/include/vtk-7.1:$PATH
```

Adding this line to the `~/.bashrc` file will automatically set this every time you open a new terminal.

There is an issue for the use of earlier versions of VTK with OpenFOAM. The issues is the ambiguity of the `sqrt(...)` function call reported here: https://gitlab.kitware.com/vtk/vtk/-/issues/17144. This is in the `Common/Core/vtkMath.h` file. In recent versions (>8), the `sqrt(...)` call is replaced with `std::sqrt(...)` and the issue goes away. So either install a latest version of VTK (version > 8), or compile from source code as instructed here: https://vtk.org/Wiki/VTK/Building/Linux. To compile, I did the following:

- install a relatively recent version of `CMake`, e.g., v3.20.

  https://cmake.org/download/

  And put the `CMake`'s `bin` path to the `PATH`, e.g.,

  ```
  export PATH = path/to/CMake/bin:$PATH
  ```

- install VTK

  https://vtk.org/Wiki/VTK/Building/Linux

  ```
  $ make install
  ```

  By default, it will install VTK to `/usr/local`. But if you don't have admin previlige, you can configure CMake to install in user's home directory.

- add the VTk include and lib paths to OpenFOAM code's `Make/options` file, e.g.

```
EXE_INC = \
...
      -I/usr/local/include/vtk-9.0

EXE_LIB = \
...
      -L/usr/local/lib -lvtkIOImport-9.0 -lvtkIOGeometry-9.0
```

When run the compiled OpenFOAM code, it is important also to set the `LD_LIBRARY_PATH` so the code can find the VTK shared libraries:

```
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
```

In OpenFOAM v5.x, one needs to increase `writePrecision` to say 9 (instead of the default 6) when the coordinates are georeferenced which typically entails large value. If the write precision is low, the mesh points location will not be accurate and the mesh is corrupted. In other versions of OpenFOA, such as v2006, there seems no such problem even when `writePrecision` is set to 6. Maybe the default is changed when the value exceeds certain range. But to be safe, just set it a large value like 9.

In OpenFOAM, to change boundary (and other fields, which are all dictionaries), we can use `changeDictionary`. For example, `gmshToFoam` does not create `empty` patches, we can change that with `changeDictionary`.

C.2 Glyph in Paraview

VTk files can be loaded into Paraview for visualization. One potential problem in the context of 2D hydraulic modeling is that the coordinates in a real-world coordinate system. The horizontal coordinates (x and y) can be large numbers, which may cause problem for the visualization of glyph (vectors). Figure C.1 shows such problem. The arrows are calculated wrong. It is suspected that this is due to the loss numerical accuracy when operating on a large value and a small value. This problem is present in the latest version of ParaView version 5.9.1 as of the writing of this document.

To further prove this point, a simple VTK unstructured grid with only one 2D cell is created. At the cell center, a vector is defined. Two VTK files are used for comparison: one is with large coordinates and the other with small coordinates (by shifting the cell toward origin).

The first example VTK file with small horizontal coordinates is as follows:

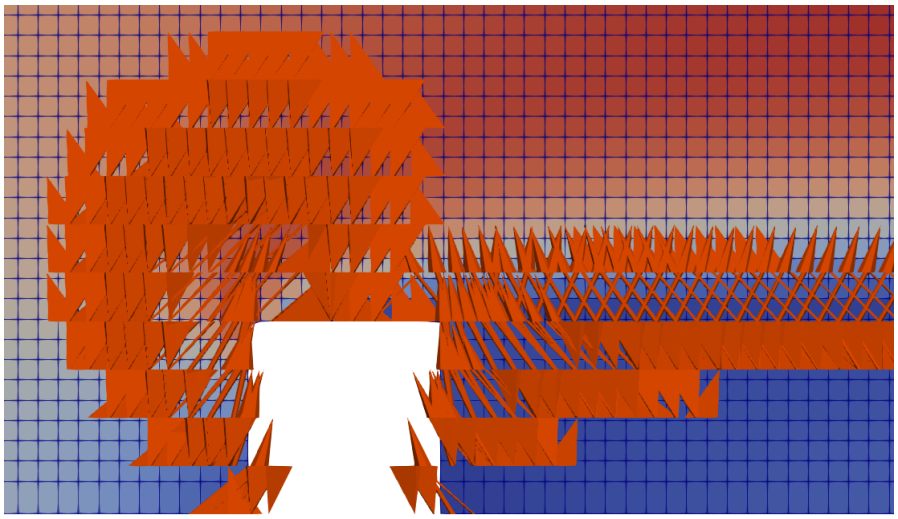


Figure C.1 Problem of glyph in ParaView for showing vectors with large coordinates.

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET UNSTRUCTURED_GRID
POINTS 4 double
103 35 0
104 35 0
103 36 0
104.1 36.1 0
CELLS 1 5
4 0 1 3 2
CELL_TYPES 1
9
CELL_DATA 1
FIELD FieldData 1
Velocity_ft_p_s 3 1 double
0.3 0.2 0
```

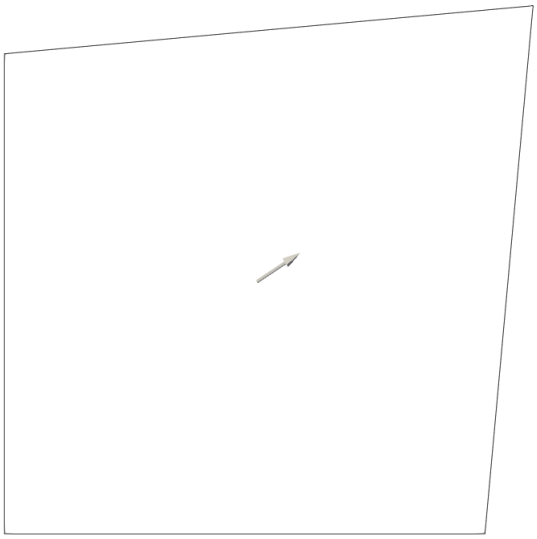
The second example VTK file with large horizontal coordinates is as follows. It is generated by shifting the coordinates of the first example.

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET UNSTRUCTURED_GRID
POINTS 4 double
```

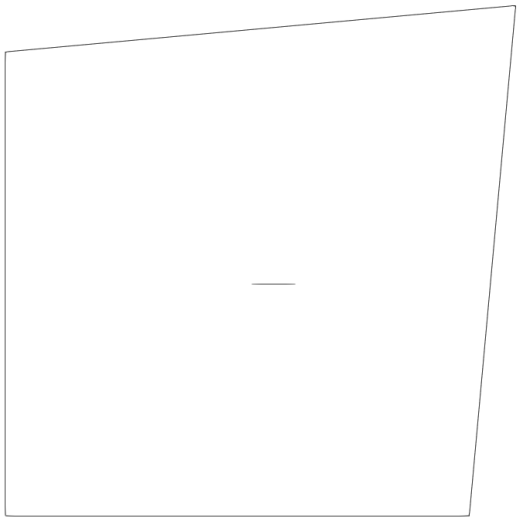


```
259103 4521835 0
259104 4521835 0
259103 4521836 0
259104.1 4521836.1 0
CELLS 1 5
4 0 1 3 2
CELL_TYPES 1
9
CELL_DATA 1
FIELD FieldData 1
Velocity_ft_p_s 3 1 double
0.3 0.2 0
```

So how to solve this problem when using ParaView to visualize VTK generated from *pyHMT2D*? We can wait for ParaView to resolve this issue (it should not be very difficult to fix this now we know what is causing the problem). Before this is fixed in ParaView, a more practical solution is that we can shift the data set toward the origin by using the "Transform" filter in ParaView. For the same data set shown in Figure C.1, if it is translated toward the origin such that all the x and y coordinates are not large values, the glyph is plotted correctly (Figure C.3.



(a) The first example VTK file where the horizontal coordinates are small and the glyph is plotted correctly.



(b) The second example VTK file where the horizontal coordinates are large and the glyph is plotted wrong.

Figure C.2 Glyph plots from the two example VTK files whose only difference is the shifting of horizontal coordinates.

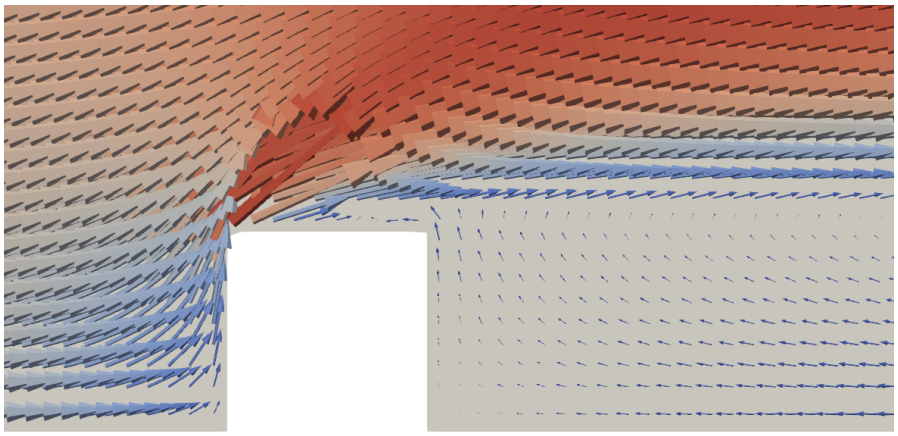


Figure C.3 Solution of the glyph problem in ParaView by using the "Transform" filter to move the data set toward the origin.

Bibliography

T. Dysarz. Application of python scripting techniques for control and automation of hec-ras simulations. *Water*, 10(10):1382, 2018. doi: 10.3390/w10101382.

T. Garcia, P. R. Jackson, E. A. Murphy, A. J. Valocchi, and M. H. Garcia. Development of a fluvial egg drift simulator to evaluate the transport and dispersion of asian carp eggs in rivers. *Ecological Modelling*, 263:211–222, 2013.

M. Gomez, S. Sharma, S. Reed, and A. Mejia. Skill of ensemble flood inundation forecasts at short- to medium-range timescales. *Journal of Hydrology*, 568: 207–220, 2019.

C. R. Goodell. *Breaking the HEC-RAS Code: A User's Guide to Automating HEC-RAS*. h2ls, Portland, OR, 1 edition, 2014.

M. Hammond and A. Robinson. *Python Programming On Win32: Help for Windows Programmers*. O'Reilly Media, Sebastopol, CA, 1 edition, 2000.

A. S. Leon and C. Goodell. Controlling hec-ras using matlab. *Environmental Modelling & Software*, 84:339–348, 2016. ISSN 1364-8152. doi: https://doi.org/10.1016/j.envsoft.2016.06.026.

V. Moya Quiroga, I. Popescu, D. P. Solomatine, and L. Bociort. Cloud and cluster computing in uncertainty analysis of integrated flood models. *Journal of Hydroinformatics*, 15(1):55–70, 2013.