ADCPtool

A postprocessing framework for ADCP measurements

Reference Manual

Jakob Steidl
Clemens Dorfmann

Institute of Hydraulic Engineering and Water Resources Management

Graz University of Technology

Abstract

Accoustic Doppler Current Profiler allows measuring discharges and velocities in fuids. However the software used for prostprocessing this data is specialized for the area of initial usage of the ADCP method.

This project aims to bypass this problem by creating a framework (you could also call it a Swiss army knife) for postprocessing ADCP measuring data.

Contents

1 Introduction to ADCPtool

1.1 What is it?

ADCPtool is as piece of software to post process discharge data produced with an ADCP, written in Python. It can read measurement files exported from *WinRiver* in ASCII format and represent the data as a Python object. This Python object can then be processed and exported to other formats. Currently the following features are available for postprocessing:

- geo mapping: add geographical coordinates to the profile
- projection: define how cells are being aligned on a profile
- velocity component rotation: align coordinate system for velocities to profile, or to global coordinates
- compute depth averaged velocities
- outliers: automatic detection and removal of outliers
- interpolation: smooth data
- roughness and shear stress estimation
- velocity extrapolation

Currently the following output formats are supported:

- freely configurable ASCII output
- BlueKenue
- Paraview
- DXF

1.2 Getting Started

1.2.1 Requirements

Before using ADCPtool, the following software packages are required:

- Python 2.7 or 3.x [1]
- matplotlib [2]

[1] It was tested with version 2.7.3 and 3.2.3
[2] Tested with version 1.1.1 and 1.2.0

- NumPy and SciPy [3]

Python and the extra modules should run on any Operating System. Users of other Operating Systems than Windows should consult the corresponding manual on how to install Python and Python packages[4].

Windows users new to Python could use a pre-packaged Python distribution, which includes all the required software[5].

1.2.2 Installing

Extract the provided ZIP file to a folder of your choice.

1.3 How to use it

ADCPtool doesnt have a GUI (yet), nor is it a command line program which one has to pass dozens of parameters to it. Instead it has to be imagined like a toolbox that can be used with Python.

To actually use ADCPtool one can either create a python script like in the tutorial in Chapter 4 or for experimenting, run it from the interactive Python console.

[3] Tested with 1.6.2 and 1.7.0-beta2
[4] Its much easier actually than on Windows
[5] A list of such distributions can be found on `http://www.python.org/download/`

2 Basics of Acoustic Doppler Current Profiler

2.1 ADCP Basics

The *Acoustic Doppler Current Profiler* is a device developed to measure currents in open waters. It relies on the Doppler-Effect, which describes the change of wavelengths when waves are reflected by a body with a relative velocity to the receiver:

$$\Delta f = \frac{\Delta v}{c} f_0$$

where

Δf	observed frequency shift
Δv	relative velocity between sender and reflector
f_0	emitted frequency
c	speed of wave propagation

If the emitter and the receiver are on the same location, the relative movement of objects reflecting the waves can be determined.

With the assumption that all particles (reflecting objects) are moving in the same direction, one can emit waves (beams) in three different directions to compute their three-dimensional movement in space. Most ADCP devices use four beams to detect deviations and to allow a report of quality of the measurement.

When further assuming a constant wave propagation velocity, the distance of the reflecting object to the emitter can also be recorded by measuring the signal reflection time. This allows the ADCP device to measure velocities in (almost) all depths. However there are limits for the time measuring method, which is why the water column under the ADCP device is divided into *cells* (also *bins*).

A much deeper introduction can be found in “ADCP: Principles of Operation - A Practical Primer” [5].

2.2 Conventions and Definitions

Profile

Profile is a virtual line over a river (or similar) spanning from reach to reach. It is where the ADCP measurement boat ideally *should* cruise.

Ensemble

The ADCP continously takes measurements on different positions. All measurements on the same position are summarized in one *ensemble*.

Cell (or Bin)

In practice the device can not measure infinitesimal areas, so cubic areas are summarized (averaged, etc) into one *cell*.

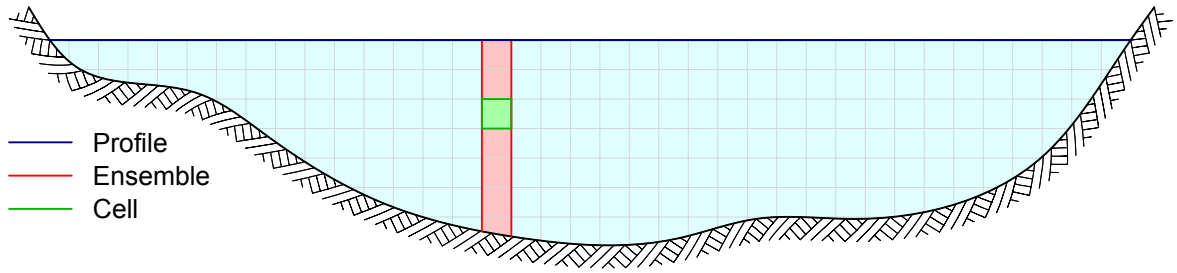


Figure 2.1: Definition of Profile, Ensemble, Cell

3 Program description

3.1 Overview

ADCPTOOL is to be understood as a set of functions and objects written in Python that can be either called from the interactive Python shell or from within a control file. The tool introduces two main objects:

1. `RawProfileObj`: represents data in the raw form and in the same units as in the WinRiver ASCII file.
2. `ProcessedRawObj`: represents the measurement data in a processed form.

These two contain other objects:

1. `ProcessedEnsembleObj`: holds data typically linked to an ensemble
2. `ProcessedCellObj`: holds data typically linked in a cell (eg. velocities)
3. `Vector`: whenever possible, this is used to store vector data. allows basic vector operations, but does'nt require NumPy

Functions can be classified into three groups:

1. import function: import the WinRiver ASCII file (there is only one of this)
2. processing functions: manipulate the profile's data
3. output functions: produce graphical output or convert data into a specific file format.

Consistency in processing function (and if applicable, others too):

- All processing functions return `ProcessedRawObj` which the specific processing applied.
- All processing functions follow the schema: `updated_profile = some_useful_function(profile, config_dictionary)`

A Note on Units

It should also be noted, that in the `ProcessedRawObj` all values are in the base SI-Units. This includes values that are usually given in other units, like the bed roughness which is stored in [m] instead of the more common [mm].

3.2 In-Depth Module Description

3.2.1 Initial Processing

After the WinRiver ASCII file has been successfully imported, the data needs to be geo-referenced. This includes the definition of a profile in global coordinates and how the measured data is positioned on it.

Technically this is done by the following line of code:

```
profile_stage0 = adcpprocess.ProcessedProfileObj(raw_profile, processing_settings,
    startingpoint)
```

While `raw_profile` is self explaining, `processing_settings` (it is a Python dictionary object) holds configuration on how data is being converted. With `startingpoint` (also a Python dictionary) the profile is being defined[1].

startingpoint

The profile can be imagined as virtual cross section of a river. To geo-reference it, the user can define one of the following options:

1. a start coordinate (and an optional offset)
2. a start coordinate and direction angle
3. a start coordinate and an end coordinate

In the first case the direction angle of the profile is taken from the first and last points of the measured data.

The profile direction vector is computed like this:

$$\boldsymbol{profile_{dir}} = \begin{cases} \frac{\boldsymbol{x_{end}} - \boldsymbol{x_{start}}}{|\boldsymbol{x_{end}} - \boldsymbol{x_{start}}|} & \text{for options 1 and 3} \\ \begin{bmatrix} \cos\alpha \\ \sin\alpha \end{bmatrix} & \text{for option 2} \end{cases}$$

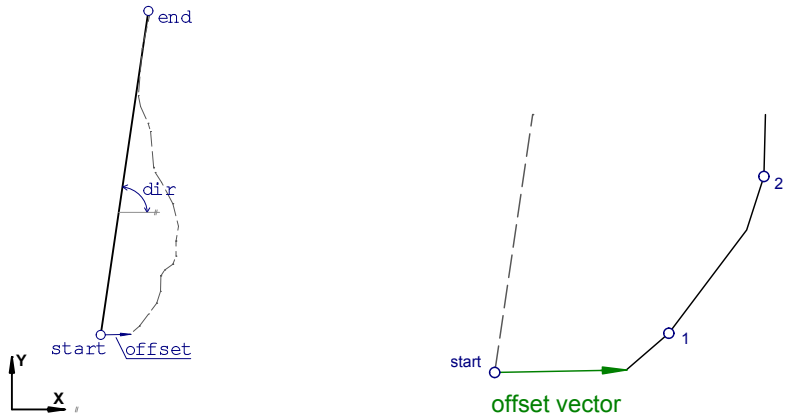
offset

In case where the first ensemble is too far away from from the ideal profile line, this approach would give a wrong direction angle. Therefore one can define an offset, which -for computing the direction angle- corrects the position of the first ensemble. See Fig. 3.1(b).

processing_settings

Besides other things (see Table 3.2), the configuration variables inside `processing_settings` control how the ensembles are ”glued” on the profile discussed above.

[1] One can also rename startingpoint to something that reflects its content more precise.



(a) Profile definition

(b) Offset vector definition

Figure 3.1: Definition of profile and offset vector

Variable	Type	•/○	Description
`start`	`Vector`	•	Start of Profile in X/Y/Z-Coordinates.
`end`	`Vector`	○	End of Profile in X/Y/Z-Coordinates. (Z-Value is being ignored)
`dir`	`float`	○	Direction of Profile in [rad]
`offset`	`Vector`	○	Offset.

Note: • . . . Argument mandatory, ○ . . . Argument optional

Table 3.1: Variables of startingpoint

Projection Method

There have been three ways been implemented for this:

1. put ensembles on the profile according to their distance made good.
2. ignore defined profile and put ensembles where they actually have been measured
3. make a planar projection of the ensembles on the defined profile

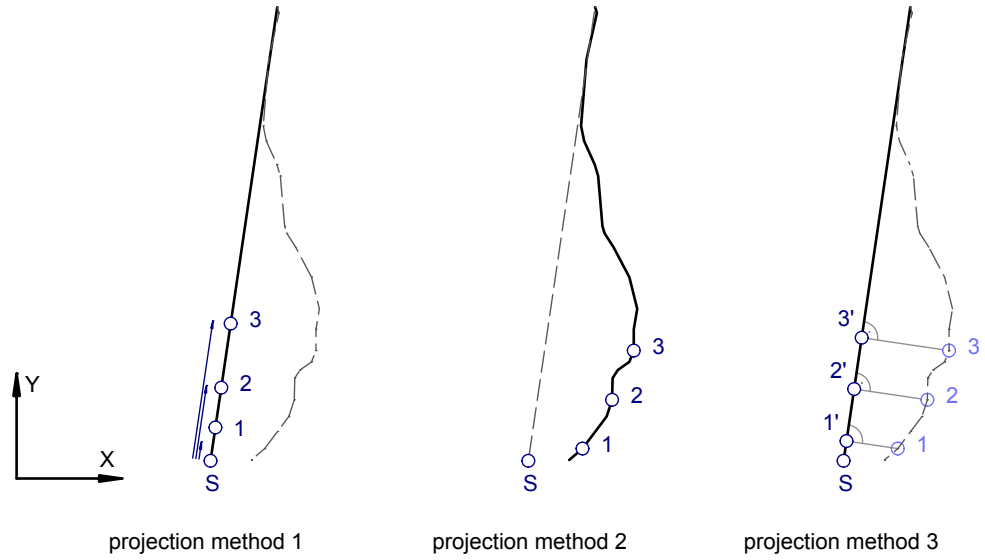


Figure 3.2: Available projection methods

Variable	Type	•/○	Description
`proj_method`	`int`	•	Defines how measured ensemble are projected on the profile. See 3.2.1
`uv_rot`	`bool`	○	`True`, if coordinate system for velocities shall be parallel to profile direction (first component will be parallel). Default: `False`
`dimensions`	`int`	○	If `2`, only depth averaged velocities will be computed. With `3` depth averaged velocities wont be computed. If undefined: averaged and cell velocities will be computed
`flip_z`	`bool`	○	`True` if z-axis shall point downwards. (affects `ProcessedCellObj.z_position` only). default: `False`
`avg_method`	`int`	○	Decides how the depth averaged velocity shall be computed. (currently only default method `0` is possible)

Note: • . . . Argument mandatory, ○ . . . Argument optional

Table 3.2: Variables of processing_settings

3.2.2 Outlier Detection and Removal

The automatic outlier removal can be called with:

```
profile_stage1 = outliers.interpolate_outliers(profile_stage0, cfg_outliers)
```

Algorithm Principle

The algorithm is based on analyzing the relative deviation of each velocity component in every cell. For each velocity $v_i(c_i)$ component in each cell c_i the following procedure is performing:

1. make a collection of neighbouring cells that are closer or equal the horizontal and vertical distance of r_h and r_v, which should describe a rectangle of the shape $(2r_h + 1) \times (2r_v + 1)$.

2. compute mean value μ_i and standard deviation σ_i
3. compute relative deviation d_i with:
$$d_i = \frac{|v_i - \mu|}{\sigma}$$
4. if d_i is greater than the defined limit α, the velocity component c_i is spotted as outlier
5. mark this velocity component as void in a separate matrix which will be used for replacing the outliers with values interpolated from its neighbors.

It should be mentioned, that in preparation for this algorithm, the velocity data are extracted to a NumPy array with the size $m \times n \times 3$ ($m \ldots$ number of ensembles, $n \ldots$ maximum number of cells in an ensemble)for optimized processing time.

Variable	Type	•/○	Description
`radius_h`	`int`	•	Horizontal radius of the sampling box
`radius_v`	`int`	○	Vertical radius of the sampling box (default: `0`)
`limit`	`bool`	•	limit of relative deviation, above which velocity components will be marked as outliers

Note: • . . . Argument mandatory, ○ . . . Argument optional

Table 3.3: Variables of `cfg_outliers`

Advanced Outlier Algorithm

In a more sophisticated setup one could save computing time by reusing matrices that are required for both outliers and data averaging. For that, the functions inside `outliers.interpolate_outliers()` are being called directly:

```
#
# outliers
import outliers, averaging, interpolation


# ProcessedCellObjs in a matrix
cm = outliers.get_cell_matrix(p)

# velocity components matrix, including a matrix that can be used to mask the first
    one
vm, vgm = outliers.get_valuematrix_from_cellmatrix(cm, '.velocity.v')

# matrix with relative deviations inside
rdm = outliers.get_relative_deviation_simple(vm, vgm, cfg)

# outliers: boolean matrix
olm = outliers.get_outliers(rdm, vgm, cfg_outliers)


# interpolation
from interpolation import interpolate
# interpolated velocity matrix (olm has to be "flipped" (0-->1, 1-->0))
ivm = interpolation.interpolate(vm, ~olm)
```

```
#
# averaging
vma = averaging.get_moving_average(ivm, vgm, dict(order=51))

# depth averaged velocities need to be updated as well
profile.update_velocities(vma, vgm)
```

3.2.3 Velocity Averaging

Data averaging can be useful to reduce the effect of random measurement errors, eg. when results shall be compared to numerical analysis.

Algorithm principle

Similar to the outlier detection the algorithm walks through every velocity component:

1. if current velocity component is marked in the `velocity_matrix_mask`: create matrix of neighbors with width defined in config parameter `order`
2. compute average of values in created matrix
3. assign this average as value of current velocity

Simple Usage

The simple way is as easy as the following:

```
import averaging
profile_averaged = averaging.get_averaged_profile(profile, cfg_averaging)
```

Advanced Usage

To use this, it is first required to create a velocity matrix, like mentioned in Chap. 3.2.2. It is longer to type, but saves a few seconds of life time.

```
import outliers
velocity_matrix, velocity_matrix_mask = outliers.get_valuematrix_from_cellmatrix(cm,
    '.velocity.v')
vma = averaging.get_moving_average(velocity_matrix, velocity_matrix_mask,
    cfg_averaging)
profile.update_velocities(vma, velocity_matrix_mask)
```

Variable	Type	•/○	Description
`order`	`int`	•	Horizontal number of cells in the sampling box.

Note: • . . . Argument mandatory, ○ . . . Argument optional

Table 3.4: Variables of `cfg_average`

3.2.4 Roughness and Shear Stress Estimation

The estimation of bed roughness k_s, bed shear stress τ_0 and shear velocities v_0 is based on following formula, which describes a logarithmic velocity distribution in open channels (see [4]):

$$\frac{v(z)}{v^*} = \frac{1}{\kappa} \ln \frac{z}{z_0}$$

where

$v(z)$	velocity
z	depth of channel starting from river bed
z_0	depth of channel where $v = 0$; for hydraulically rough flows: $z_0 = \frac{1}{30} k_s$
v^*	shear velocity
κ	VON KÁRMÁN constant: 0.40
k_s	bed roughness

With this formula and the measured velocities (magnitude of x and y component) the unknowns k_s, v^* can be determined via the method of least squares.

Least Square Fit Algorithm

To achieve that, the logarithmic formula above needs to be linearized:

$$\frac{v(z)}{v^*} = \frac{1}{\kappa}(\ln z - \ln z_0)$$
$$\ln z = \frac{\kappa}{v^*} v + \ln z_0$$

To finally get a least square solution, the NUMPY function `linalg.lstsq()` is used. It returns a, b of the equation $y = ax + b$. With these, the unknowns k_s, v^* can be determined:

$$v^* = \frac{\kappa}{b}$$
$$k_s = 30e^a$$

In addition, the shear stress can be back calculated with:

$$\tau_0 = {v^*}^2 \rho_{water}$$

Usage

Roughness estimation can be invoked with:

```
profile2 = logfit.logfit_profile(profile1, cfg_logfit)
```

Afterwards, the ensembles in `profile2` will have some extra properties:

`ks`	equivalent bed roughness
`tau_shear`	τ_0
`v_shear`	v^*
`logfit_debug`	a list containing a and b, r and p from pearson correlation test, no. of cells used

Variable	Type	•/○	Description
`logheight`	`int`	•	relative height of the layer where the power law can be applied (recommended: `0.2`)
`component`	`int`	•	velocity component that will be used for least squares regression `0`: x, `1`: y, `2`: z, `3`: magnitude of x and y (recommended)

Note: • . . . Argument mandatory, ○ . . . Argument optional

Table 3.5: Variables of `cfg_logfit`

Variable	Type	•/○	Description
`topcells`	`int`	•	number of cells used for linear extrapolation new water surface
`forcepowerlaw`	`bool`	•	use power law for extrapolation on both zones

Note: • . . . Argument mandatory, ○ . . . Argument optional

Table 3.6: Variables of `cfg_extrapolation`

3.2.5 Extrapolation of cells

In order to fill the gaps in be unmeasured areas below water surface and above river bed, extrapolation methods are being used. While this can not produce any new or additional information about the unmeasured zone, extrapolation can be useful when estimating total discharge.

Usage

Extrapolation can be invoked with:

```
profile_stage3 = extrapolation.extrapolate_profile(profile_stage1, cfg_extrapolation)
```

Where `cfg_extrapolation` is explained in Table 3.6.

Extrapolation Algorithms

The algorithm used for the area near water surface (referred as "zone T") can be described as follows:

1. take the `topcells` topmost cells and fit a linear function in it.
2. use this function to create additional velocity components

Is done for each velocity component seperatly.

For the area near river bed, a number of different methods, depending on available data is being used (See Figure 3.4).

The simplest one makes use of k_s and v^*. If a log law velocity distribution is assumed.

1. compute velocity magnitude with $v = v^* \cdot 2.5 \cdot \ln \frac{30}{k_s(d+z)}$

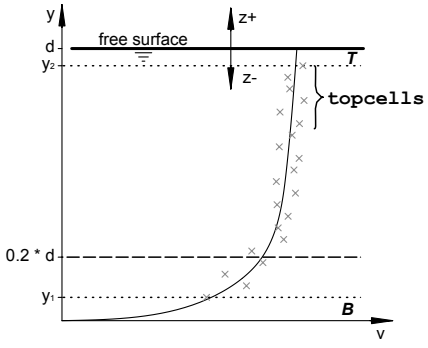


Figure 3.3: Variables within velocity extrapolation

2. compute average direction angles
3. split velocity magnitude into vector components with computed angles

If no roughness data is available, velocities in “zone b”, an estimation via the power law is being done. The following velocity distribution can be used to approximate the full depth of a channel, as shown by Cheng [3]:

$$v(y) = a \cdot y^b \qquad (b = 1/6)$$

While a could also be derived from bed roughness and shear stress, it isn't necessary for extrapolation.

Similar to WinRiver [6], ADCPtool first tries to find cells at $0.2 \cdot d$ to determine a with:

$$a = \frac{v(0.2 \cdot d)}{y^b}$$

Otherwise a will be determined by equalizing the measured discharge with the discharge in the measured area using the power law velocity distribution:

$$Q_{ADCP} = \sum_i v_i \cdot d_i$$

$$Q_{PL} = \int_{y_1}^{y_2} a y^b \, \mathrm{d}y = a \frac{y_2^{b+1} - y_1^{b+1}}{b+1}$$

With the initial condition

$$Q_{PL} = Q_{ADCP}$$

a can be explicitly written as

$$a = \frac{\sum_i v_i \cdot d_i}{\frac{y_2^{b+1} - y_1^{b+1}}{b+1}}$$

It is worth mentioning, that this algorithm also works on the magnitude velocities, which require to be split up into its components.

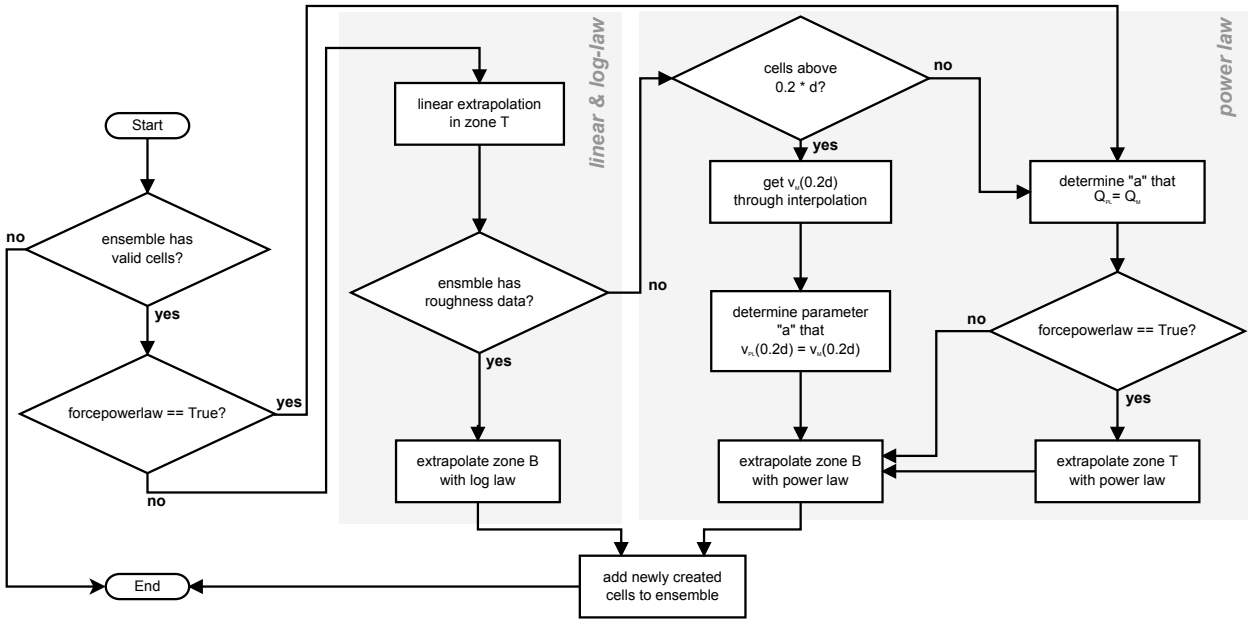


Figure 3.4: Flow chart for extrapolation

3.2.6 Profile Visualisation

For a quick visual feedback profile data can be plotted. For this two functions in the module `quickviz` exist:

`plot_profile_2d(p, cfg)`	plot the plan view of the profile including depth averaged velocities
`plot_profile_3d(p, cfg)`	plot a cross section view trough the profile showing data specified inside `cfg` (See Table 3.8)

The second function provides a few features that require explanation:

What Data to Show

Virtually any data inside an `ProcessedCellObj` can be displayed. (See the `datatype` variable in Table 3.8). If "`custom`" is provided, then `cellattr` has to be filled with the cell attributes (including the dot at the beginning) to be plotted. See Section 4.2 for examples.

How to Show the Data

If more than one data is being selected, it is possible to display it as a vector field[2]. If two or more attributes are selected and a display `style` other than "`vector`" is used, the "length" of the attributes will be displayed:

$$a = \sqrt{\sum a_i^2}$$

[2]Please note: If it is desired to plot secondary currents, it is strongly recommended to use the `uv_rot` option when creating the ProcessedProfileObj

Variable	Type	•/○	Description
`title`	`string`	○	title of the plot
`saveas`	`string`	○	path and filename where to save output graphic, if not specified, plot will be displayed on screen

Note: • . . . Argument mandatory, ○ . . . Argument optional

Table 3.7: Variables inside `cfg` for `plot_profile_2d()`

3.2.7 Export Formats

For passing on the data to other software, the following export functions are available:

Pre-Defined Output Methods

The names should be mostly self explaining. Only some parts of the names need explanation:

`2D`	will export data (usually depth averaged velocities) stored directly in the ensemble
`3D`	will export data stored directly in the cells
`BlueKenue`	Output for BlueKenue™[2]
`Paraview`	Output for ParaView [1]
`DXF`	famous CAD drawing exchange file format

where

`profile`	the `ProcessedProfileObj` to be exported
`f`	the output file
`format`	a format string. See Sec. 3.2.7
`vel_scale`	a scaling value for displaying velocities. If `1`, a velocity of $1[\mathrm{m/s}]$ will be drawn with the length of 1
`f_depth4Beams`	filename for storing all beams
`f_depth4Beam`*1-4*	filename for storing beam 1-4

Customizable Output Methods

The functions `writeAscii2D()` and `writeAscii3D()` allow the user to define which information is being written and how it is formatted.

They will be called with:

```
# for ensemble data
writeAscii2D(profile, formatstring, f, header)

# and respectively for cell data
writeAscii3D(profile, formatstring, f, header)
```

where

`profile`	is the ProcessedProfileObj to be exported
`formatstrting`	Python format string for the `print() function`
`f`	output file name
`h`	string as file header, optional

As one might expect, the definition of the format can be defined in `formatstrting`, which itself will be parsed by Python's `.format()` method [3].

Essentially, a format string contains the variables to be printe d in curly brackets surrounded by white space:

```
formatstrting = "{vx}, {vy}, {vz}"
```

The a list of available variables can be found in Table 3.10. Usage examples can be found in Section 4.3.2

[3] See `http://docs.python.org/2/tutorial/inputoutput.html#fancier-output-formatting` and `http://docs.python.org/2/library/string.html#formatspec`

Variable	Type	•/○	Description
datatype	string	○	determines what to plot, possible values: • velocity: requires variable components (default) • custom: requires variable cellattr
components	list	○	a list of of velocity components, that will be used. valid components: "x", "y", "z" (default: ['x','y'])
cellattr	list	○	the attribute of cell the cells that will be plotted
style	string	○	visualization method. implemented methods: "contour", "gradient" (default), "vector"
title	string	○	title of the plot (default: "plan view")
saveas	string	○	path and filename where to save output graphic, if not specified, plot will be displayed on screen

Note: • . . . Argument mandatory, ○ . . . Argument optional

Table 3.8: Variables inside cfg for plot_profile_3d()

Function Name	Arguments
writeAscii3D	profile, format, f, voidtext="---", header=None
writeAscii2D	profile, format, f, voidtext="---", header=None
write2DVelocity	profile, f
write2DVelocity_BlueKenue	profile, f
write2DVelocity_Paraview	profile, f
write3DVelocity_Paraview	profile, f
write3DVelocity	profile, f
write3DVelocity_BlueKenue	profile, f
write3DVelocity_W0_BlueKenue	profile, f
writeAverageDepth	profile, f
write4BeamDepths	profile, f_depth4Beams, f_depthBeam1, f_depthBeam2, f_depthBeam3, f_depthBeam4
writeDXF2D	profile, f, vel_scale=50
writeDXF3D	profile, f, vel_scale=50

Table 3.9: Available Export Functions

Variable	2D	3D	Description
x	●	●	X coordinate of cell or ensemble
y	●	●	Y coordinate of cell or ensemble
z	●	●	Z coordinate of cell or ensemble
vx	●	●	x component of velocity
vy	●	●	y component of velocity
vz	●	●	z component of velocity
vmag	●	●	magnitude of x and y components of velocity
vmag3d	●	●	magnitude of all components of velocity
artificial	○	●	True, if this is an extrapolated cell
ks	○	●	bed roughness
tau_shear	○	●	bed shear stress
v_shear	○	●	shear velocity
depth	●	○	averaged depth reading of ensemble
four_depths_1	●	○	depth reading of beam 1 for ensemble
four_depths_2	●	○	depth reading of beam 2 for ensemble
four_depths_3	●	○	depth reading of beam 3 for ensemble
four_depths_4	●	○	depth reading of beam 4 for ensemble

Table 3.10: Available Variables for formatstrting

4 Tutorial

In this tutorial, we will demonstrate all features of ADCPTOOL. We will be using the PYTHON interactive shell, because it is the easiest way to play with the ADCP data[1]. Of course it is also possible to start the commands from a Python script file. See also `missioncontrol.py` or `mc_advanced.py` for examples.

Since ADCPtool isn't installed inside the Python directory, we can't start it directly. So we either have to

- add the directory where ADCPTOOL is installed to the Python path, or
- start the Python shell in the corresponding directory.

For Windows users the easiest way is probably to open the folder in Explorer, and then with *[shift]+[right mouse button]* an extended context menu appears, where we can select *"Open command window here"*, which should open up `cmd.exe`, the windows command line. In that window, the Python shell can finally be started with typing: `python`.

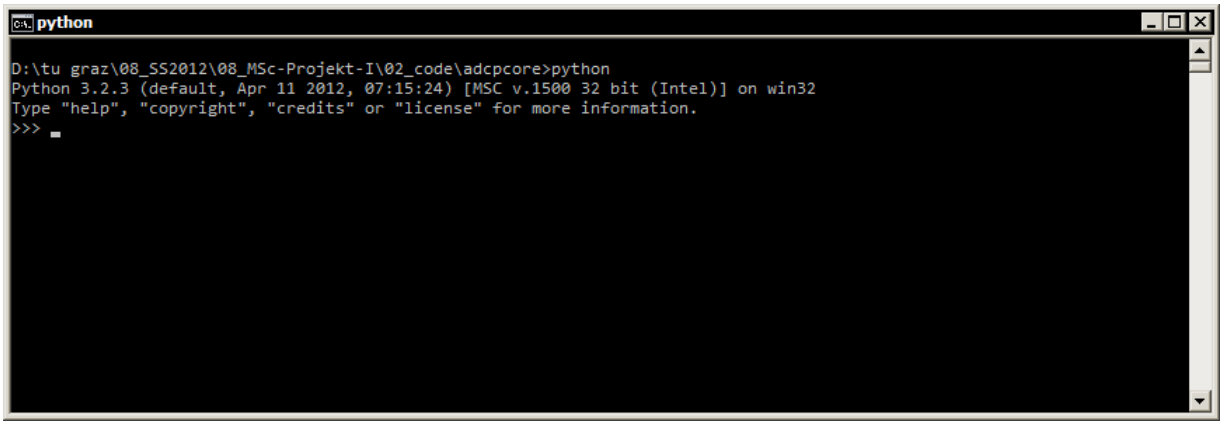


Figure 4.1: A sucessfully opened Python interactive shell on Windows

In order to have access to all the modules provided by ADCPTOOL it is recommended to import (load) them with the `adcploader` script:

```
>>> from adcploader import *
>>>
```

Something that applies for everything from now on: If there is no text output, everything worked fine.

[1] We will use Windows as operating system. If you are running a different OS, then you are probably capable of opening a python shell on your own anyway.

4.1 Required

4.1.1 Import

The basis for every processing is to convert the original WinRiver ASCII file into a Python object with:

```
>>> p_raw = RawProfileObj('../testfiles/demodata.txt')
```

4.1.2 Geo-Mapping

Geo-Mapping is required per definition, because the ensembles need to have some coordinates later on. But since we don't want have any reference coordinates, and don't care about profile projection we use the following settings:

```
>>> startingpoint = dict(start=Vector(0,0,0))
>>> processing_settings = dict(proj_method=3)
```

Then we can use these variables for the processing:

```
>>> p0 = ProcessedProfileObj(p_raw, processing_settings, startingpoint)
```

And the result can be previewed as well:

```
>>> plot_profile_2d(p0)
```

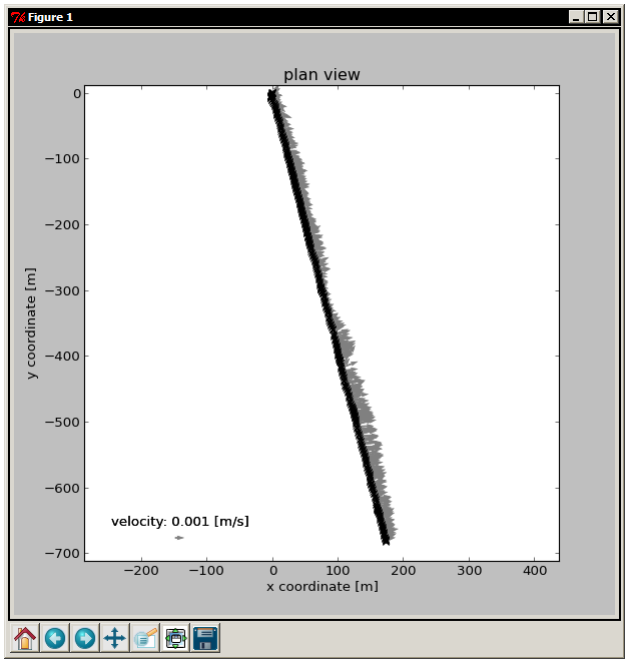


Figure 4.2: Output of `plot_profile_2d(p0)`

See Fig. 4.2 for how the output can look like. If you want to save the graphic, either use the floppy symbol found in the window. Or if you want to save the graphic without displaying it, specify that in the `cfg`:

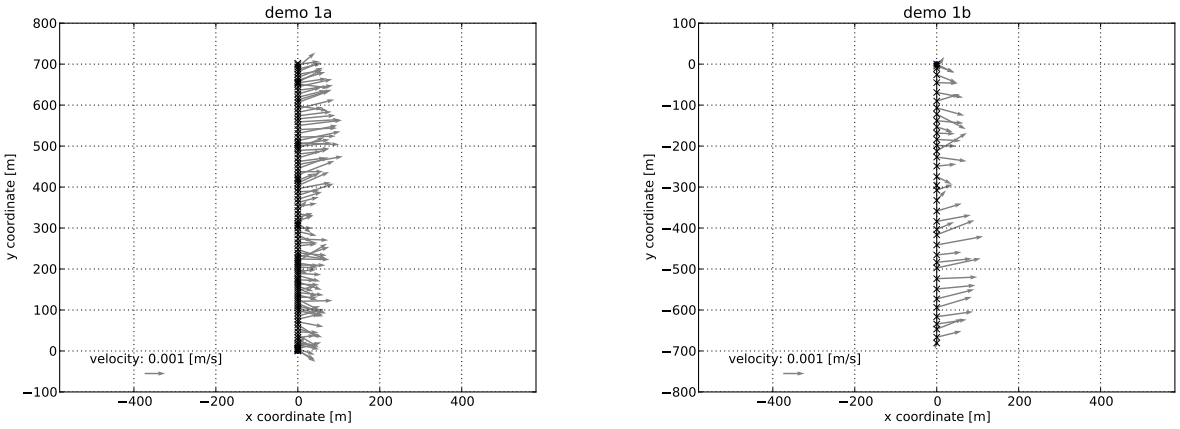
```
>>> plot_2d_cfg = dict(saveas='../testfiles/demo1.pdf')
>>> plot_profile_2d(p0)
```

The file type is automatically detected by the file extension.

Now a few config settings and the `thin_out()` function shall be demonstrated:

```
>>> import math
>>> startingpoint_1a = dict(start=Vector(0,0,0), dir=0.5*math.pi)
>>> startingpoint_1b = dict(start=Vector(0,0,0), end=Vector(0,1,0))
>>> processing_settings_1a = dict(proj_method=1)
>>> processing_settings_1b = dict(proj_method=3)
>>> p1a = ProcessedProfileObj(p_raw, processing_settings_1a, startingpoint_1a)
>>> p1b = ProcessedProfileObj(p_raw, processing_settings_1b, startingpoint_1b)
>>> plot_profile_2d(thin_out(p1a, {'keep_ensemble':5}), {'saveas':'../testfiles/
    demo_p1a.pdf', 'title':'demo 1a'})
>>> plot_profile_2d(thin_out(p1b, {'keep_ensemble':15}), {'saveas':'../testfiles/
    demo_p1b.pdf', 'title':'demo 1b'})
```

The following should give two profiles, both pointing upwards and the second with less ensembles remaining. However, comparing with the output in Fig. 4.3 the second, (b) the profile is pointing downwards! This can be explained by how the projection method 3 works.[2]



(a) projection method 1

(b) projection method 2

Figure 4.3: Demonstration of `thin_out()` and different projection methods and profile definitions

For comparison with the following steps, we also want to take a look at the velocities:

[2] As seen in Fig. 4.2 the measurement boat actually moved downwards, so the ensembles will be projected on the "negative" end of the profile, which is defined by a starting point, but however internally it is extending infinite in both directions

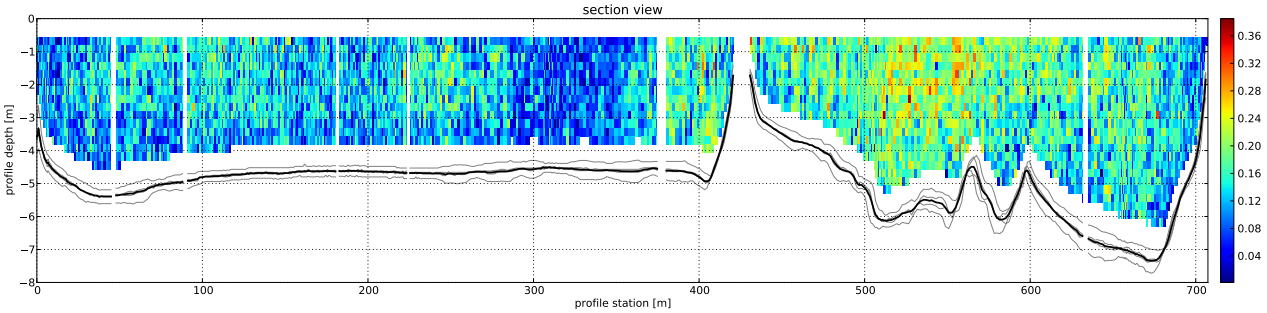


Figure 4.4: Velocities at the unmodified profile

4.2 Optional

4.2.1 Outlier Removal

The outlier removal is not that exciting:

```
>>> p2 = interpolate_outliers(p0, cfg={'limit':2.5, 'radius_h':10)
>>> plot_profile_3d(p2)
```

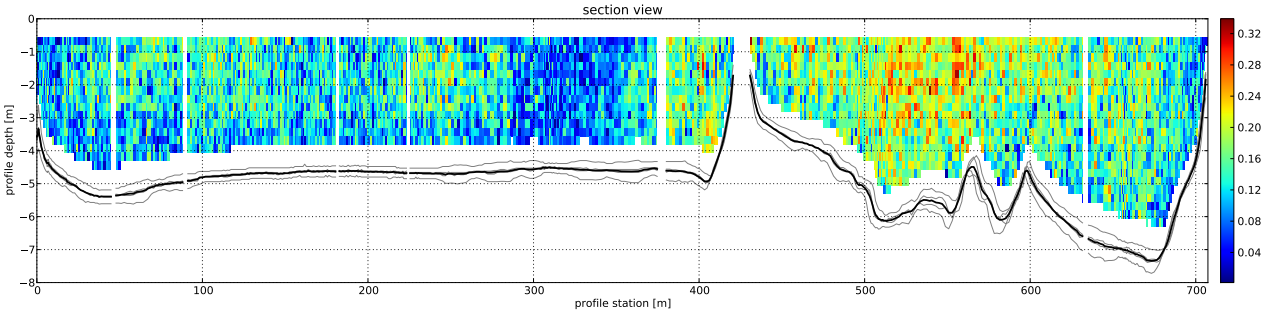


Figure 4.5: Velocities without outliers

4.2.2 Averaging

Lets do some averaging. Output see Fig. 4.6

```
p3 = get_averaged_profile(p2, cfg={'order':21})
plot_profile_3d(p3, cfg={'saveas':'../testfiles/demo_p3_3d.pdf'})
```

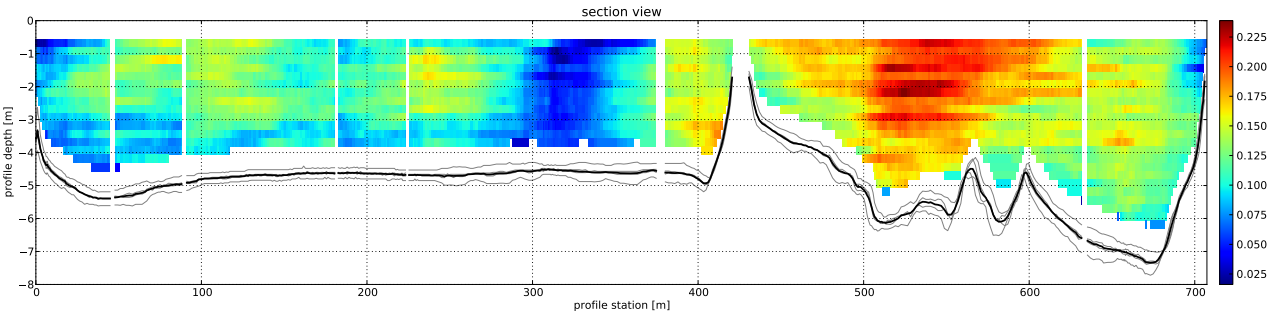


Figure 4.6: Averaged Velocities

4.2.3 Roughness and Shear Stress Estimation

This topic is a bit tricky, because the results heavily depend on the input data, which require measurements below $0.2 \cdot depth$, which is not always the case. In order to give judge the quality of the output data, we need to introduce a previously undocumented function: `plot_logfit_profile()` which can be found inside `quickviz.py`.

Starting the roughness estimation is easy however:

```
>>> cfg_logfit = {'logheight':0.30, 'component':3}
>>> p4 = logfit_profile(p3, cfg_logfit)
>>> plot_logfit_profile(p4, cfg=cfg_logfit)
```

The result of `plot_logfit_profile` can be seen in Fig. 4.7. It is left to the user, if they trust these results.

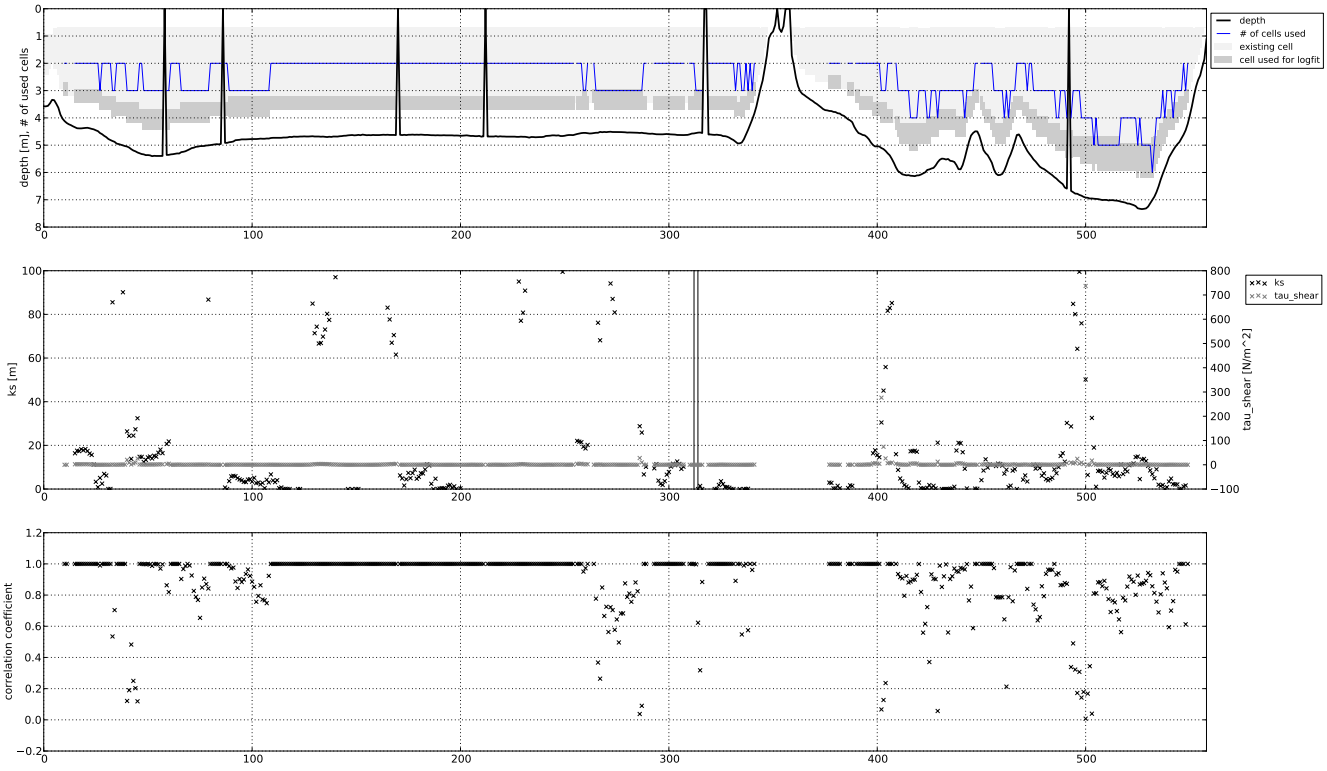


Figure 4.7: Roughness and Shear Stress

4.2.4 Velocity Extrapolation

To demonstrate the velocity extrapolation, we use the following lines of code:

```
>>> p5 = extrapolate_profile(p4, cfg={'topcells':5, 'forcepowerlaw':True})
>>> plot_profile_3d(p5, cfg={'saveas':'../testfiles/tut_demo5.pdf'})
```

Note that in this example, we used `forepowerlaw=True`, because the roughness values were not realistic and matplotlib would run into numerical problems.

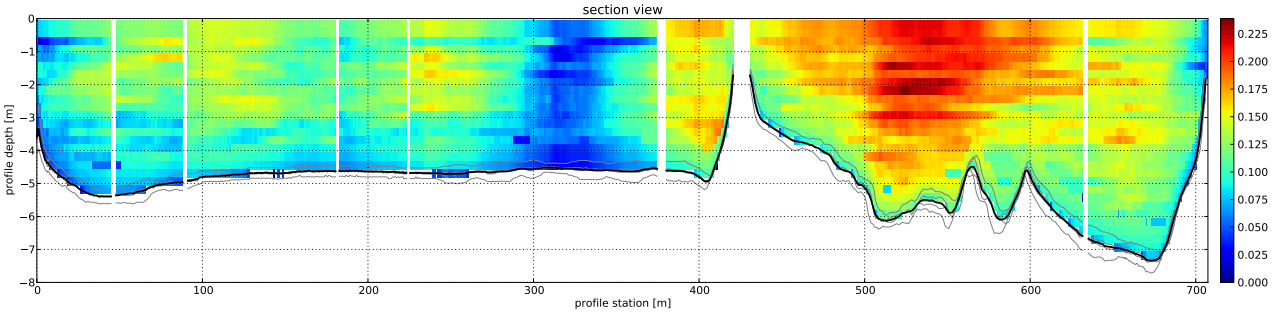


Figure 4.8: Extrapolated velocities

4.3 Recommended

4.3.1 Profile Visualisation

plot_profile_3d

So far we have only seen one way to visualize profile data. Now lets see the others:

```
>>> plot_profile_3d(p3, cfg=dict(style='contour', saveas='../testfiles/tut_demo6a.pdf
    ', title='contour demo'))
```

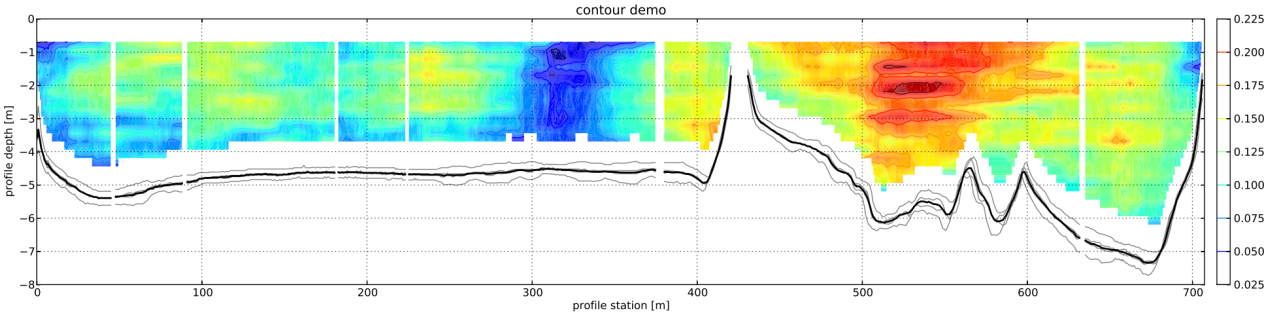


Figure 4.9: Contour Plot

When trying to plot secondary flows it is recommended to make sure the y-axis for velocities is aligned parallel to the profile. For optical reasons, one would probably want to thin out the data a bit:

```
>>> p0_sf = ProcessedProfileObj(p_raw, dict(proj_method=3, uv_rot=1), startingpoint)
>>> p1_sf = get_averaged_profile(p2, cfg={'order':5})
>>> plot_profile_3d(thin_out(p1_sf, dict(keep_ensemble=5)), cfg=dict(style='vector',
    components=['y','z'], saveas='../testfiles/tut_demo6b.pdf', title='vector demo'))
```

It should also be noted, that the demo measuremend data are not really suited, to demonstrate this effect, but it was demonstrated how it *could* work.

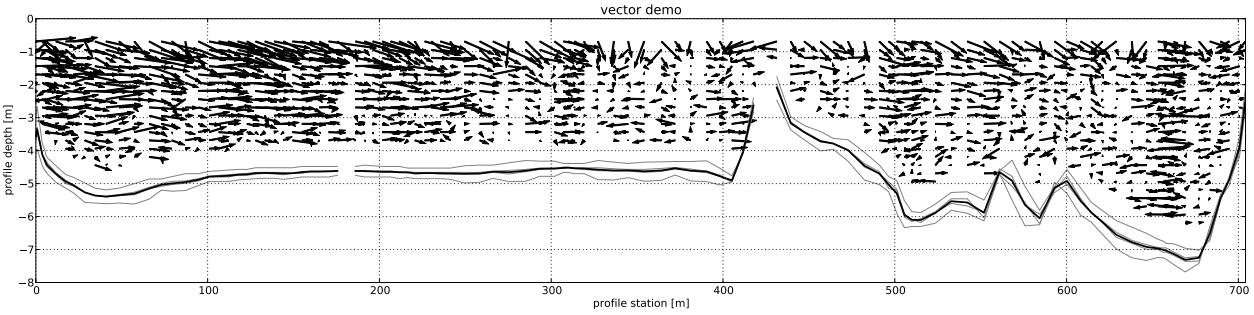


Figure 4.10: Attempted visualisation of secondary flows

4.3.2 Export in other Formats

The different output formats have already been discussed in Sec. 3.2.7 and they are pretty similar, so we just want to demonstrate it for DXF and custom ASCII output.

DXF Output

DXF output is as easy as:

```
>>> writeDXF3D(p5, '../testfiles/demo7_p5_3d.dxf', vel_scale=40)
>>> writeDXF2D(p5, '../testfiles/demo7_p5_2d.dxf', vel_scale=40)
```

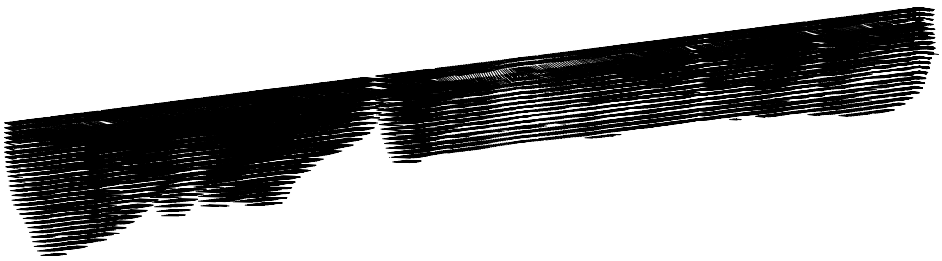


Figure 4.11: Output of `writeDXF3D()`

Custom ASCII Output - Example 1

In a first example, we want to export the cell coordinates and their velocity components.

```
>>> writeAscii3D(p5, '{x} {y} {z} {vx} {vy} {vz}', '../testfiles/tut_demo8_velocities
    .txt')
```

This produces an output which is correct, but not very pleasing:

```
...
0.0 0.0 -3.44  0.0487707013245  0.00411604185165 -0.00629261246467
-0.0352388493007 0.138668689947 -0.19  0.0809096270561  -0.00156761831666 -0.011711294767
-0.0352388493007 0.138668689947 -0.44  0.0798823693507  -0.00154771527109 -0.0115626039594
-0.0352388493007 0.138668689947 -0.69  0.0787844753962  -0.00152644365317 -0.0114036888811
...
```

Therefore we use the specify the space each argument is allowed to fill and the number of decimals:

```
>>> import datetime
>>> header='# generated on: {} \n# x y z v_x v_y v_z \n'.format(datetime.datetime.now
    ().strftime('%x %X'))
>>> writeAscii3D(p5, '{x:6.2f} {y:6.2f} {z:6.2f} {vx:9.2e} {vy:9.2e} {vz:9.2e}', '../
    testfiles/tut_demo8_velocities_awesome.txt', header=header)
```

Which produces an output that is much more readable:

```
# generated on: 12/20/12 13:21:44
# x      y     z      v_x       v_y       v_z
  0.00   0.00  -0.19  8.32e-02  7.02e-03 -1.07e-02
  0.00   0.00  -0.44  8.21e-02  6.93e-03 -1.06e-02
  0.00   0.00  -0.69  2.96e-02 -5.45e-04 -1.51e-02
  0.00   0.00  -0.94  7.77e-02  9.00e-03 -1.14e-02
...
```

Custom ASCII Output - Example 2

As a final exercise, we will print the estimated roughness values:

```
>>> header2d='# generated on: {} \n# x y ks tau_shear v_shear\n'.format(datetime.
    datetime.now().strftime('%x %X'))
>>> writeAscii2D(p4, '{x:6.2f} {y:6.2f} {ks:9.2e} {tau_shear:9.2e} {v_shear:9.2e}', '
    ../testfiles/tut_demo8_rouhgness.txt', header=header2d, voidtext=-999)
```

And the output should look like this:

```
# generated on: 12/20/12 13:21:44
# x      y     ks        tau_shear  v_shear
  0.00   0.00 -9.99e+02 -9.99e+02 -9.99e+02
  0.42  -1.64 -9.99e+02 -9.99e+02 -9.99e+02
...
  0.56  -2.19  2.14e+15  1.12e-03 -1.06e-03
  0.64  -2.51  2.13e+03  6.76e-02 -8.22e-03
...
```

The roughness values are now ready for further processing in other programs. Note that values of -999 are invalid values. This was set with `voidtext=-999`.

4.3.3 Conclusion

This chapter has provided everything needed to use ADCPtool. For developing and testing own scripts, it is recommended to comment out the `plot` functions, as they consume a significant amount of time. Also Pythons `pickel` module is a good alternative way to re-compute data every run by storing temporary results (all Python objects, variables) on hard disk.

Bibliography

[1] Paraview: open-source, multi-platform data analysis and visualization application. http://www.paraview.org.

[2] N. R. C. Canada. Blue kenue^{TM}: Software tool for hydraulic modellers. http://www.nrc-cnrc.gc.ca/eng/solutions/advisory/blue_kenue_index.html.

[3] N.-S. Cheng. Power-law index for velocity profiles in open channel flows. *Advances in Water Resources*, 30:1775 – 1784, 2007.

[4] T. Karvonen. Hydraulics script. http://civil.tkk.fi/fi/tutkimus/vesitalous/www_oppikirjat/yhd_122010/, 2009.

[5] Teledyne RD Instruments. *Acoustic Doppler Current Profiler - Principles of Operation*, 1996.

[6] Teledyne RD Instruments. *WinRiver II User's Guide*, 2007.

List of Figures

List of Tables