AvaFrame

AvaFrame developers

Nov 16, 2023

GENERAL

CHAPTER

ONE

INTRODUCTION

AvaFrame is a cooperation between the Austrian Avalanche and Torrent Service (Wildbach- und Lawinenverbauung; WLV) and the Austrian Research Centre for Forests (Bundesforschungszentrum für Wald; BFW) within the Federal Ministry Republic of Austria: Agriculture, Regions and Tourism (BMLRT).

Our mission is to develop an open source framework (licensed with the European Union Public license (EUPL)) for avalanche simulations which replicates and simplifies current simulation procedures and models at the WLV and make them accessible to the public domain.

Our aims are to make AvaFrame:

- applicable in an operational setting e.g. for hazard mapping or planning of mitigation measures within the WLV and civil engineering.
- extensible for scientific applications to be used to develop new methods, models and cover a wide range of scientific topics.
- easily usable and extendable by (inter-)national partners, academia and interested users.

AvaFrame is part of an BMLRT initiative to foster technological innovation within a strategic focus on the area of Austrian safety policies, digital innovations and climate mitigation strategies.

Information in German can be found on the BMLRT Homepage.

A summary of the latest advances of the project can be found on the Avaframe homepage web-page.

Feel free to contact us at our matrix room if you have any questions or suggestions. AvaFrame is in an early development phase and all ideas and contributions are welcome!

If you want to cite our project, please use:

CHAPTER

TWO

GENERAL

2.1 Introduction

AvaFrame is a cooperation between the Austrian Avalanche and Torrent Service (Wildbach- und Lawinenverbauung; WLV) and the Austrian Research Centre for Forests (Bundesforschungszentrum für Wald; BFW) within the Federal Ministry Republic of Austria: Agriculture, Regions and Tourism (BMLRT).

Our mission is to develop an open source framework (licensed with the European Union Public license (EUPL)) for avalanche simulations which replicates and simplifies current simulation procedures and models at the WLV and make them accessible to the public domain.

Our aims are to make AvaFrame:

- applicable in an operational setting e.g. for hazard mapping or planning of mitigation measures within the WLV and civil engineering.
- extensible for scientific applications to be used to develop new methods, models and cover a wide range of scientific topics.
- easily usable and extendable by (inter-)national partners, academia and interested users.

AvaFrame is part of an BMLRT initiative to foster technological innovation within a strategic focus on the area of Austrian safety policies, digital innovations and climate mitigation strategies.

Information in German can be found on the BMLRT Homepage.

A summary of the latest advances of the project can be found on the Avaframe homepage web-page.

Feel free to contact us at our matrix room if you have any questions or suggestions. AvaFrame is in an early development phase and all ideas and contributions are welcome!

If you want to cite our project, please use:

2.2 Installation

Note: There are two ways to install and use AvaFrame:

Operational (GUI)
If you want the operational workflow, running *com1DFA* (dense flow avalanche) and *com2AB* (alpha beta) with standard settings from within *QGis*, head over to the *Operational Installation* (*Deutsche Version*) section. Use this if you want to:

- use the standard, well tested and calibrated setup for hazard mapping or similar

- use QGis as frontend
- have results in a short amount of time
- use the latest release

Advanced (Script)
If you want to contribute and develop AvaFrame, head over to *Advanced (Script) Installation*. Use this if you want to:

- work on the code itself
- implement new features
- change/improve existing code
- have the latest development code. *Warning: might be unstable!*

2.2.1 Operational Installation

This is the quick start for the operational AvaFrame setup with QGis as frontend.

2.2.1.1 Requirements

The prerequisites are:

- QGis: install from here: QGis installation (we recommend using the latest version)

2.2.1.2 Setup AvaFrameConnector and run

1. Open QGis from your start menu and go to Plugins -> Manage and Install Plugins
2. Search for *AvaFrameConnector* and install it
3. Wait for it to finish and restart QGis
4. Access the QGis - Avaframe connector via Toolbox -> AvaFrame -> Operational -> FullOperationalRun
5. Add the described data and run. Results will be loaded after a while (depending on the size of your DEM)

This tries to install the AvaFrame core package as well (since AvaFrameConnector version 1.2). If it fails, head over to the operational:Manual Installation instructions.

For more information about the functions in the AvaFrameConnector, see connector:AvaFrameConnector (GUI)

2.2.1.3 Update Avaframe to a new release

1. Restart/Open QGis from your start menu and go to Plugins -> Manage and Install Plugins
2. Search for *AvaFrameConnector* and check whether it also needs updating
3. Access the QGis - Avaframe connector via Toolbox -> AvaFrame
4. Go to the *Admin* section, choose and run *Update*

2.2.2 Operationelle Installation (Deutsch)

Dies ist der Schnellstart für die Verwendung von AvaFrame mit QGis als Frontend.

2.2.2.1 Vorraussetzungen

Die Vorraussetzungen sind:

- QGis: QGis Installation (download) (Wir empfehlen die aktuellste Version zu verwenden)

2.2.2.2 Setup AvaFrameConnector

1. Öffnen Sie QGis über Ihr Startmenü und gehen Sie zu Erweiterungen -> Erweiterungen verwalten und installieren
2. Nach *AvaFrameConnector* suchen und installieren
3. Warten Sie, bis der Vorgang abgeschlossen ist und starten Sie QGis neu
4. Rufen Sie den QGis - AvaframeConnector über Verarbeitungswerkzeuge -> AvaFrame -> Operational -> FullOperationalRun auf
5. Fügen Sie die beschriebenen Daten hinzu und starten Sie den Connector. Die Ergebnisse werden nach einer Weile geladen (abhängig von der Größe Ihres DEM)

Hiermit wird versucht das AvaFrame-Kernpaket zu installieren (seit AvaFrameConnector Version 1.2). Wenn dies fehlschlägt, gehen Sie zu: operationalGerman:Manuelle Installation.

Für eine Kurzzusammenfassung der Funktionen des AvaFrameConnector, siehe connector:AvaFrameConnector (GUI).

2.2.2.3 Update Avaframe auf eine neue Version

1. Starten Sie QGis neu/öffnen Sie es über Ihr Startmenü und gehen Sie zu Plugins -> Plugins verwalten und installieren
2. Suchen Sie nach AvaFrameConnector und prüfen Sie, ob es aktualisiert werden muss
3. Rufen Sie den QGis - AvaframeConnector über Verarbeitungswerkzeuge -> AvaFrame auf
4. Dann -> Admin -> Update aufrufen

2.3 QGis AvaFrameConnector

The AvaFrameConnector allows QGis users to access certain base workflows directly from QGis. The connector only provides the interface to functions within the AvaFrame python package, and is developed separately, see the QGisAF github repository. It makes use of the QGis processing plugin, which is included in all current QGis releases.

Note: ALL (!!) data provided to the functions below HAVE to be in the same projection. It does not matter which projection it is as long as it it the same one.

2.3.1 Operational

Alpha Beta (com2)
Runs the alpha beta calculation via module com2AB. For more specific info about the inputs (shapefile attributes etc), see *Input*.

Dense Flow Standard (com1)
Runs dense flow avalanche module com1DFA. For more specific info about the inputs (shapefile attributes etc), see *Input*. You can select multiple shapefiles for release areas, each will be calculated as one scenario. Each shapefile can contain multiple polygons. If you provide entrainment/resistance areas, each scenario will be calculated once without them (i.e null simulation) and once with them.

Operational Run (com1 and com2)
Runs com1DFA and additionally com2AB if profile and splitpoint are set. Additional info see description of *Dense Flow Standard (com1)* and *Alpha Beta (com2)*.

2.3.2 Experimental

These either require deeper knowledge about the AvaFrame python package, i.e. configuration files etc., or are not yet fully tested and might produce unwanted results.

AvaFrameLayerRename
Renames com1DFA result layers by adding the values of the given variable (from the configuration file)

Probability run (ana4, com1)
Runs probability simulations via module com1DFA. The release shape HAS TO HAVE a ci95 field containing the 95 percentile confidence interval (same unit as the release thickness). Multiple scenarios can be provided, final map includes variations from all scenarios combined. Release thickness and SamosAT friction mu are being varied, 40 variations per scenario.

Release area stats(in1, com1)
Returns info for release area statistics in a csv file. See output of QGis processing for the location of the file.

Snow slide (com5)
Runs snow slide simulations via module com1DFA. For more info see com5SnowSlide section in the documentation. The resistance layer is meant for building outlines.

2.3.3 Admin

GetVersion
Displays the current version of the AvaFrame python package (NOT the AvaFrameConnector)

UpdateAvaFrame
Updates the AvaFrame python package

2.4 Advanced Usage (Script)

After the installation, the next sections describe on how to get started. The sections configuration and logging describe the general methods we use, this is helpful to understand how you can change model parameters and similar.

2.4.1 Advanced (Script) Installation

This is a quick guide on how to install AvaFrame and the required dependencies on your machine. AvaFrame is developed on **Linux machines** (Ubuntu/Manjaro/Arch) with recent Python versions > 3.8. These instructions assume you are familiar with working in a terminal. This guide is currently described for *Linux* only, but expert users should be able to adapt it to *Windows*.

2.4.1.1 Requirements

Install git and python, we suggest to work with miniconda/anaconda. For installation see miniconda or anaconda. Some operating systems might require the python headers (e.g python-dev on ubuntu) or other supporting libraries/packages (e.g. Visual Studio on Windows).

2.4.1.2 Setup AvaFrame

Create a new conda environment for AvaFrame, activate it and install pip, numpy and cython in this environment:

```
conda create --name avaframe_env
conda activate avaframe_env
conda install pip numpy cython
```

Clone the AvaFrame repository (in a directory of your choice: [YOURDIR]) and change into it:

```
cd [YOURDIR]
git clone https://github.com/avaframe/AvaFrame.git
cd AvaFrame
```

Compile the cython com1DFA part. You might also have to install a c-compiler (gcc or similar) through your systems package manager:

```
python setup.py build_ext --inplace
```

Warning: You will have to do this compilation every time something changes in the cython code. We also suggest to do this everytime updates from the repositories are pulled.

Install avaframe and its requirements:

```
pip install -e .
```

This installs avaframe in editable mode, so every time you import avaframe the current (local) version will be used.

Test it by starting `python` and do an `import avaframe`. If no error comes up, you are good to go.

Head over to gettingstarted:First run for the next steps.

2.4.1.3 Update AvaFrame

To update go to your avaframe repository [YOURDIR]/Avaframe, pull the latest changes via:

```
git pull
```

and repeat the compilation step from above:

```
python setup.py build_ext --inplace
```

If there are updates on the requirements inside `setup.py`, it might be necessary to run:

```
pip install -e .
```

again to get the additional requirements installed.

2.4.2 First run

Follow these steps to run your first simulation:

- change into your `AvaFrame` directory (replace [YOURDIR] with your path from the installation steps):

  ```
  cd [YOURDIR]/AvaFrame/avaframe
  ```

- run:

  ```
  python runCom1DFA.py
  ```

- a similar output should show up:

  ```
  logUtils - INFO -  Started logging at: 03.11.2020 22:42:04
  logUtils - INFO -  Also logging to: data/avaParabola/runCom1DFA.log
  runCom1DFA - INFO -  MAIN SCRIPT
  runCom1DFA - INFO -  Current avalanche: data/avaParabola
  ...
  ```

This will perform a dense flow avalanche simulation using the `com1DFA` module. The results are saved to `data/avaParabola/Outputs/com1DFA`. For a first look at the results, got to the folder `reports`, there you can find a markdown report of the simulations performed including some plots of the results.

To display markdown files in a nice way use a markdown viewer of your choice. Some other options are:

- Use the Atom editor with a markdown plugin
- If you have *pandoc* installed use this to convert it to pdf/html
- Some browsers have markdown extensions you can install easily

2.4.3 Workflow example

The following example should make it easier for you to find your way in AvaFrame and setup your own AvaFrame workflow after you did the full setup. There is also a directory with examples for different workflows, see more here: *Example runscripts*.

Make sure you change to your `AvaFrame` directory by:

```
cd [YOURDIR]/AvaFrame
```

Replace `[YOURDIR]` with the directory from your installation step.

2.4.3.1 Initialize project

To create the folder where the input data lies and where the output results will be saved, specify the full path to the folder in the `local_avaframeCfg.ini` (which is a copy of `avaframeCfg.ini` that you need to create). So:

```
cd avaframe
cp avaframeCfg.ini local_avaframeCfg.ini
```

and edit `local_avaframeCfg.ini` with your favorite text editor and adjust the variable `avalancheDir`.

Then run

```
python runScripts/runInitializeProject.py
```

This will create a new directory with the input required by AvaFrame structured as described in *Initialize Project*.

2.4.3.2 Input data

Check the input data required by the different modules you want to use and fill the `Inputs/` inside the `[avalancheDir]` folder from the initialize step accordingly.

For example the `com1DFA` module needs input as described in *Input*. You can also have a look at the default setting for the module you want to use (for example `com1DFACfg.ini` for module com1DFA). If you want to use different settings, create a `local_` copy of the `.ini` file and modify the desired parameters.

More information about the configuration can be found here: configuration:Configuration

2.4.3.3 Building your run script

Create your own workflow by taking the `runOperational.py` script as template.

We suggest you copy it and adjust it to your liking. There are annotations in the code that should help you to understand the structure.

A lot more examples can be found in the `runScripts` directory (see also *Example runscripts*).

2.5 Configuration

In order to set the configurations required by all the modules within Avaframe, the python module configparser is used.

This is done in two steps. The first step fetches the main settings:

```
from avaframe.in3Utils import cfgUtils
# Load avalanche directory from general configuration file
cfgMain = cfgUtils.getGeneralConfig()
avalancheDir = cfgMain['MAIN']['avalancheDir']
```

In the second step the specific settings to a given module are imported:

```
from avaframe.tmp1Ex import tmp1Ex
# Load all input Parameters from config file
# get the configuration of an already imported module
# Write config to log file
cfg = cfgUtils.getModuleConfig(tmp1Ex)
```

The *in3Utils.cfgUtils.getModuleConfig()* function reads the settings from a configuration file (`tmpEx.ini` in our example) and writes these settings to the log file. The default settings can be found in the configuration file provided within each module.

It is possible to modify these settings, there are two options:

- provide the path to your own configuration file when calling `cfgUtils.getModuleConfig(moduleName, path to config file)`
- create a copy of the module configuration file called `local_` followed by the name of the original configuration file and set the desired values of the individual parameters.

So the order is as follows:

1. if there is a path provided, configuration is read from this file.
2. if there is no path provided, the `local_...` configuration file is read if it exists.
3. if there is no `local_...`, the `getModuleConfig` function reads the settings from the default configuration file with the default settings.

In the configuration file itself, there are multiple options to vary a parameter:

- replace the default parameter value with desired value
- provide a number of parameter values separated by `|` (e.g. `relTh=1.|2.|3.`)
- provide a number of parameter values using `start:stop:numberOfSteps` (e.g. `relTh=1.:3.:3`)) - a single value can be added by appending `&4.0` for example

2.5.1 Override configuration

If tools of one module, let's call this module **A** for now, are called from another module **B**, there is the option to include an *A_override* section in the configuration file of module **B**. In this case, the default configuration of module **A** is read and the parameters in the *A_override* section in the module **B** configuration file are used to update the configuration settings of module **A**. This has the advantage of gathering all the configuration parameters used for one task in one configuration file. Find below an example of:

2.5.2 Logging

In order to generate simulation logs and to control what is prompted to the terminal, we use the python module logging.

Let's have a look at the simple example in `runScripts.runTmp1Ex` and *tmp1Ex.tmp1Ex()* on how this is used within AvaFrame.

In your main script call:

```
from avaframe.in3Utils import logUtils
# log file name; leave empty to use default runLog.log
logName = 'runTmp1Ex'
# specify the working directory
avalancheDir = './'
# -----------------------------------------------
# Start logging
log = logUtils.initiateLogger(avalancheDir, logName)
```

This will configure the logging (it sets the console output as well as the log file). In your modules/subscripts add:

```
import logging
log = logging.getLogger(__name__)
```

So you can use:

```
log.debug('Should be here')
log.info('DEM : %s',variable)
```

To get output that looks like this in your console:

```
tmp1Ex:DEBUG - Should be here
tmp1Ex:INFO - DEM : /path/to/DEM
```

and something similar in the `.log` file which is saved in `./runTmp1Ex.log` in this example. The logging configuration is set in `AvaFrame/avaframe/in3Utils/logging.conf`.

You can modify this `logging.conf` file to modify the levels or format of the messages to display (python doc will help you).

2.5.3 Example runscripts

In `runScripts` we provide ready-to-use scripts for different applications of the modules provided within AvaFrame.

2.5.3.1 Derive input data

- `runScripts.runComputeDist`

2.5.3.2 Create a new project

- runScripts.runInitializeProject

2.5.3.3 Generate idealized/generic topography data

- runScripts.runGenerateTopo
- runScripts.runGenProjTopoRelease

2.5.3.4 Postprocessing

- runScripts.runAna3AIMEC
- runScripts.runAna3AIMECCompMods
- runScripts.runAna3AIMECIndi
- runScripts.runStatsExample
- runScripts.runProbAna

2.5.3.5 Visualisation

- runScripts.runQuickPlotSimple
- runScripts.runQuickPlotOne
- runScripts.runPlotTopo
- runScripts.runExportToCsv

2.5.3.6 Testing

- runScripts.runDamBreak
- runScripts.runSimilaritySol
- runScripts.runTestFP
- runScripts.runStandardTestsCom1DFAOrig
- runScripts.runComparisonModules
- runScripts.runFetchBench
- runScripts.runWriteDesDict

2.6 Testing

In AvaFrame we provide a continuously growing test suite. This includes tests that target model verification and model validation. The former are used to test the numerical implementation of the mathematical model, whereas the latter are aiming at testing if the mathematical model employed is appropriate to describe the physical processes of the desired application, in our case the simulation of dense snow avalanches.

In the following, a brief description of the available tests is given.

Note: This section is currently under development and will be updated regularly.

Note: See this example pdf (25mb) how we apply these tests during development.

2.6.1 Tests for model verification

This section includes testing of bits and pieces of the model code up to tests that check whether the full workflow is numerically robust. However, at the 'smallest scale', where individual functions are tested, we also refer to the Section about *pytests*: *How to test code*.

2.6.1.1 Rotational Symmetry

In this test, a pyramid-shaped topography is used to test whether different grid alignments in respect to the topography and hence flow direction cause problems for the model implementation. For this test, we simply provide two pyramid shaped topographies, where one is rotated around the z-axis to change the orientation of the pyramid's facets with respect to the mesh.

2.6.1.2 Dambreak problem

This test is based on a Riemann problem, where the initial condition is described by two different states that are separated by a discontinuity. In [FM13] exact solutions are derived for this problem, based on the Savage-Hutter model to describe the granular flow. Here, we provide an implementation of one of the seven cases presented in [FM13]. In Test 2, the initial velocity is zero and a granular mass is suddenly released and flows downslope. For further details have a look at the module `ana1Tests.damBreak` and the Section *Dambreak test*.

2.6.1.3 Similarity Solution

This test makes use of semi-analytic solution of a sliding granular mass on an inclined plane. In [HSSN93] the semi-analytic solution is derived for this problem, based on the Savage-Hutter shallow flow model to describe the granular flow. Here, we provide an implementation of the similarity solution and compare it to the output of a numerical simulation. For further details have a look at the module `ana1Tests.simiSol` and the Section *Similarity solution*.

2.6.2 Tests for model validation

In this section, tests that check whether the chosen mathematical model is suitable to represent the problem at hand can be found. Currently, we provide idealized and real-world topographies that can be used for avalanche simulations. Additionally, there are several functions that can be used to analyse the simulation results and compare them to other data such as results from other models or observations. For this, we refer to Sections *ana3AIMEC: Aimec* and *outQuickPlot*.

2.6.2.1 Idealized test cases

We provide a range of idealized topographies to perform avalanche simulations including different release area scenarios. Some of the topographies also include entrainment and/or resistance areas. These can be found in `data/NameOfAvalanche`. The functions to produce these can be found in module `in3Utils/generateTopo.py`.

Bowl - BL

Bowl shaped topopraphy. Used to test e.g. rotational symmetry.

Flat Plane - FP

Inclined Plane - IP

IP (Inclined plane)

Parabolic slope - PF

Parabolic slope with flat foreland

Hockeystick - HS

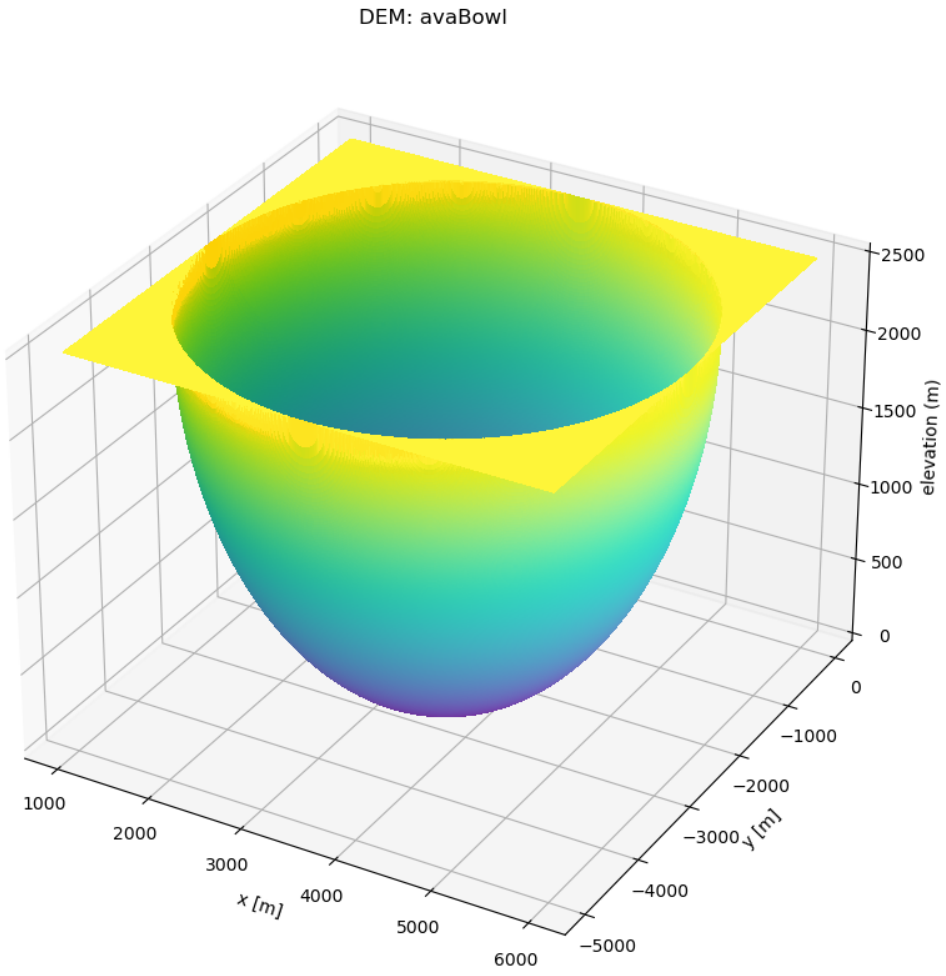
Hockeystick with linear slope and flat foreland and smooth transition

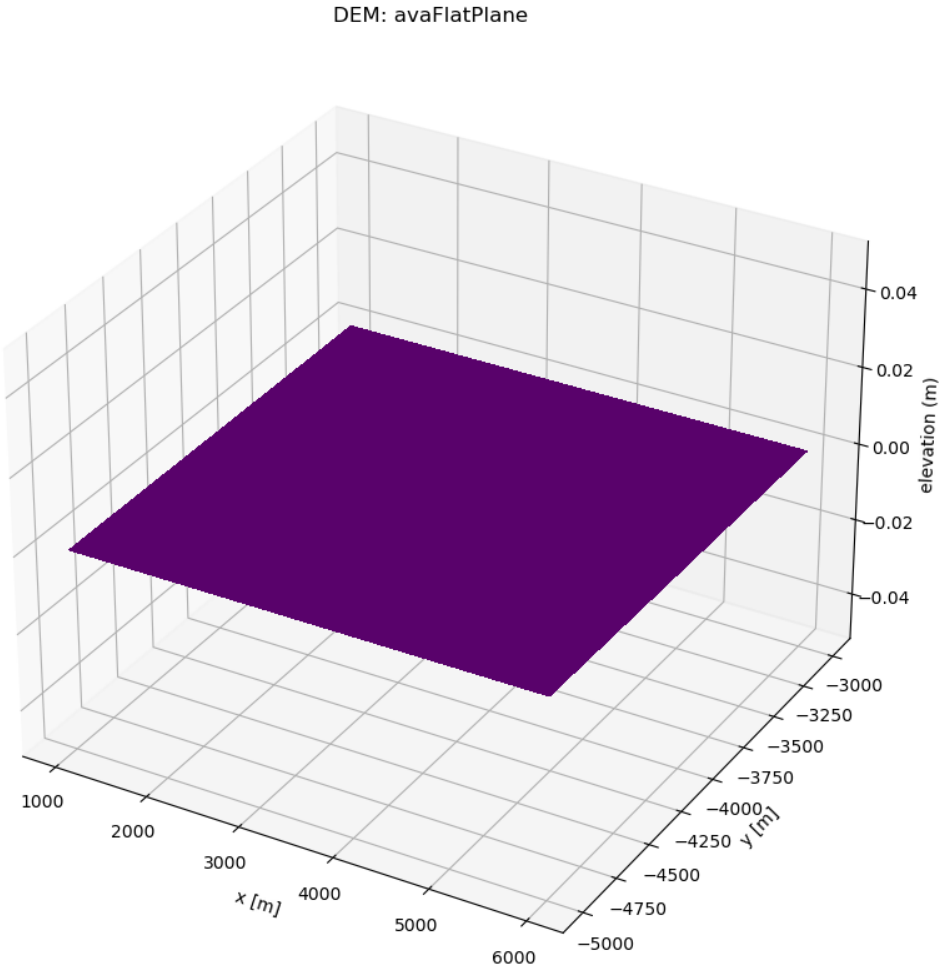
Helix - HX

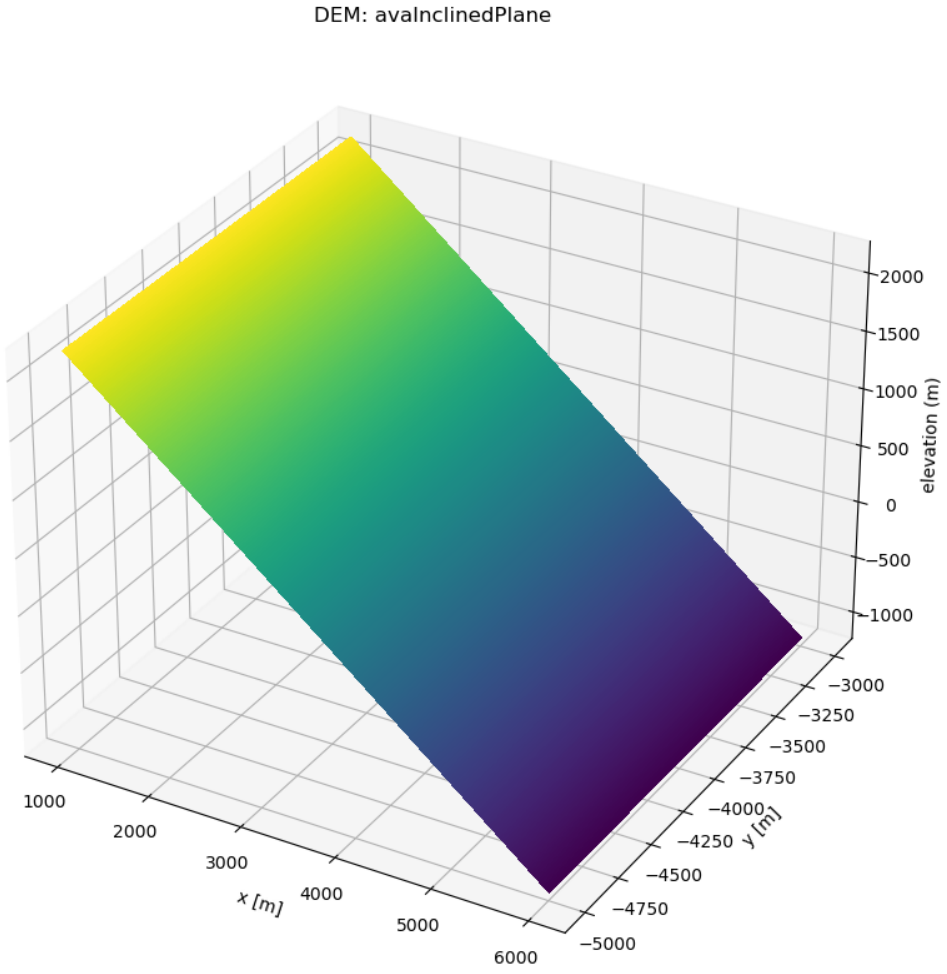
Helix-shaped topography

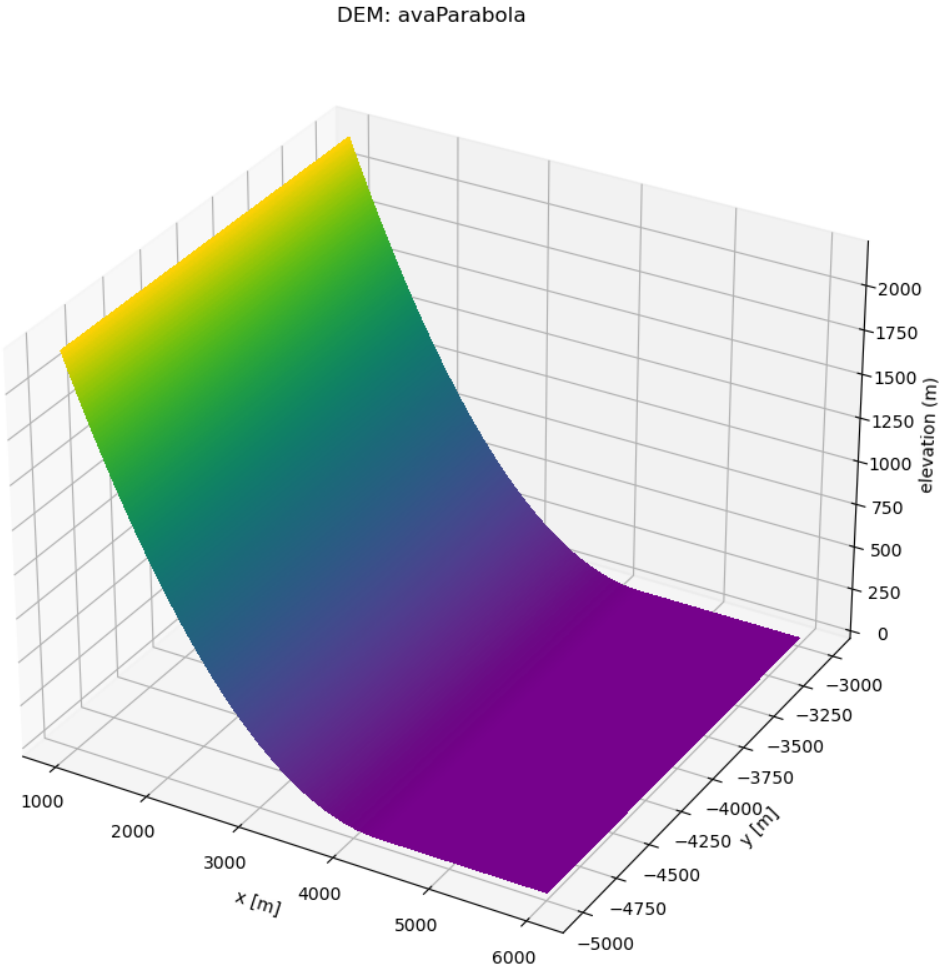
Pyramid - PY

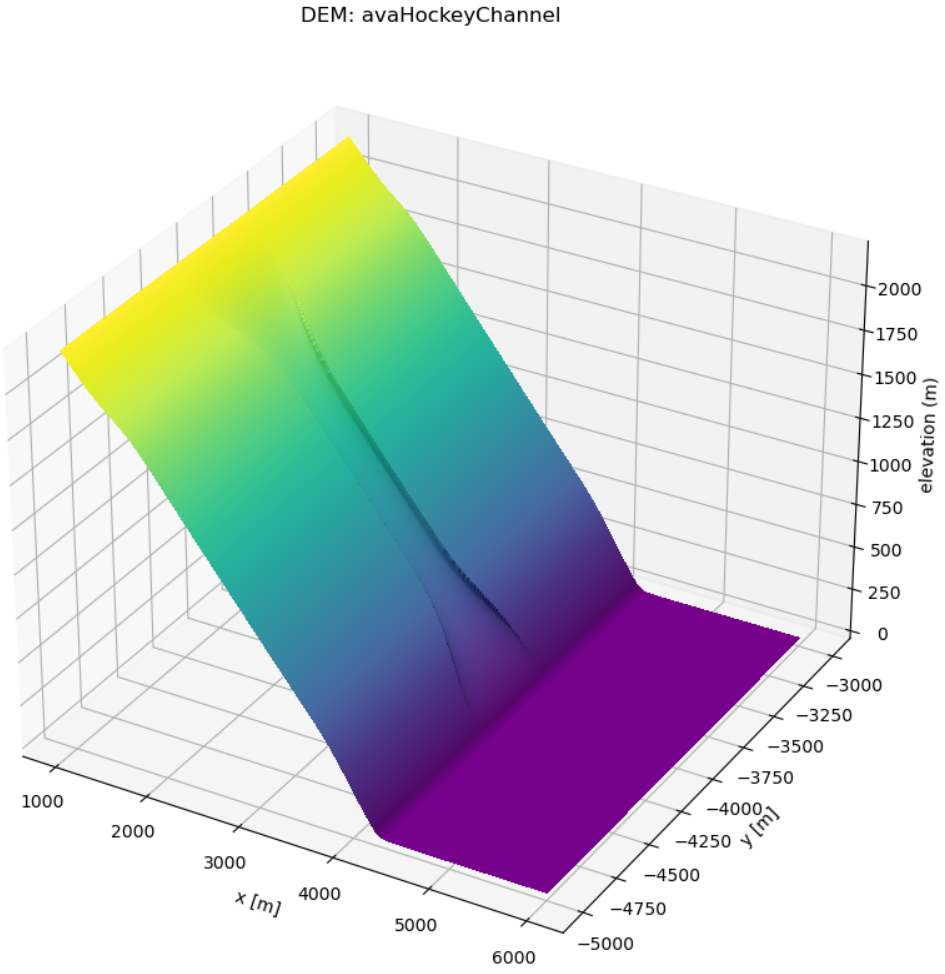
pyramid-shaped topography, optional with flat foreland

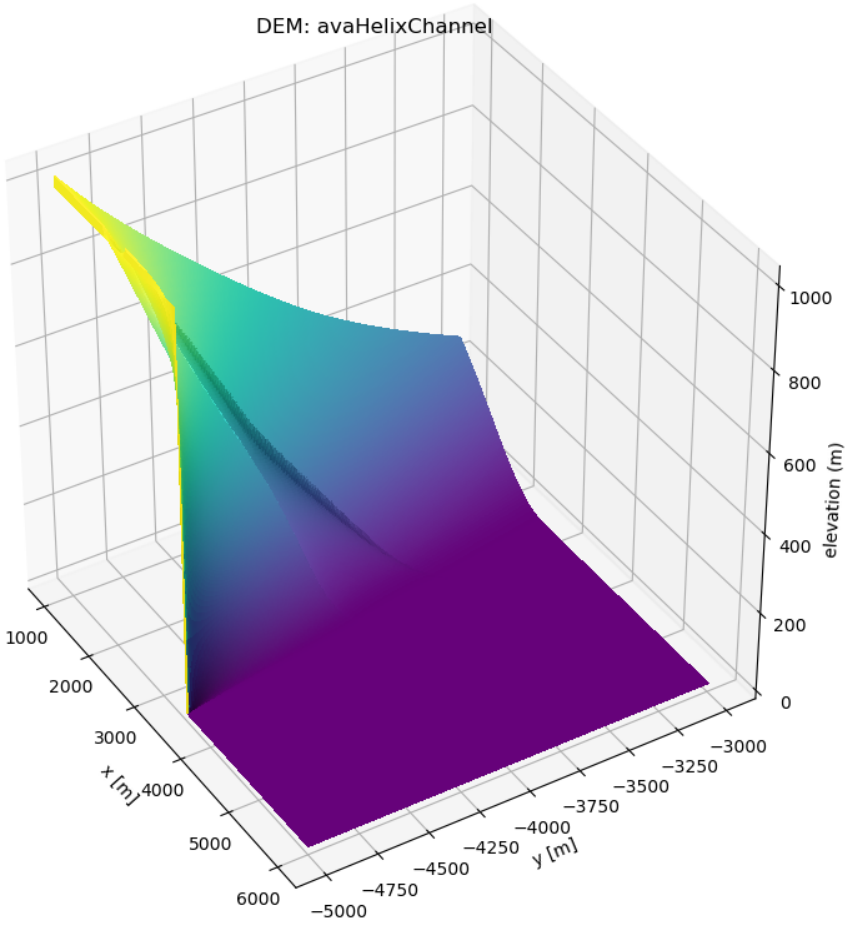

DEM: avaBowl
2500
2000
1500
1000
500
0
elevation (m)
0
−1000
−2000
−3000
−4000
−5000
y [m]
1000
2000
3000
4000
5000
6000
x [m]

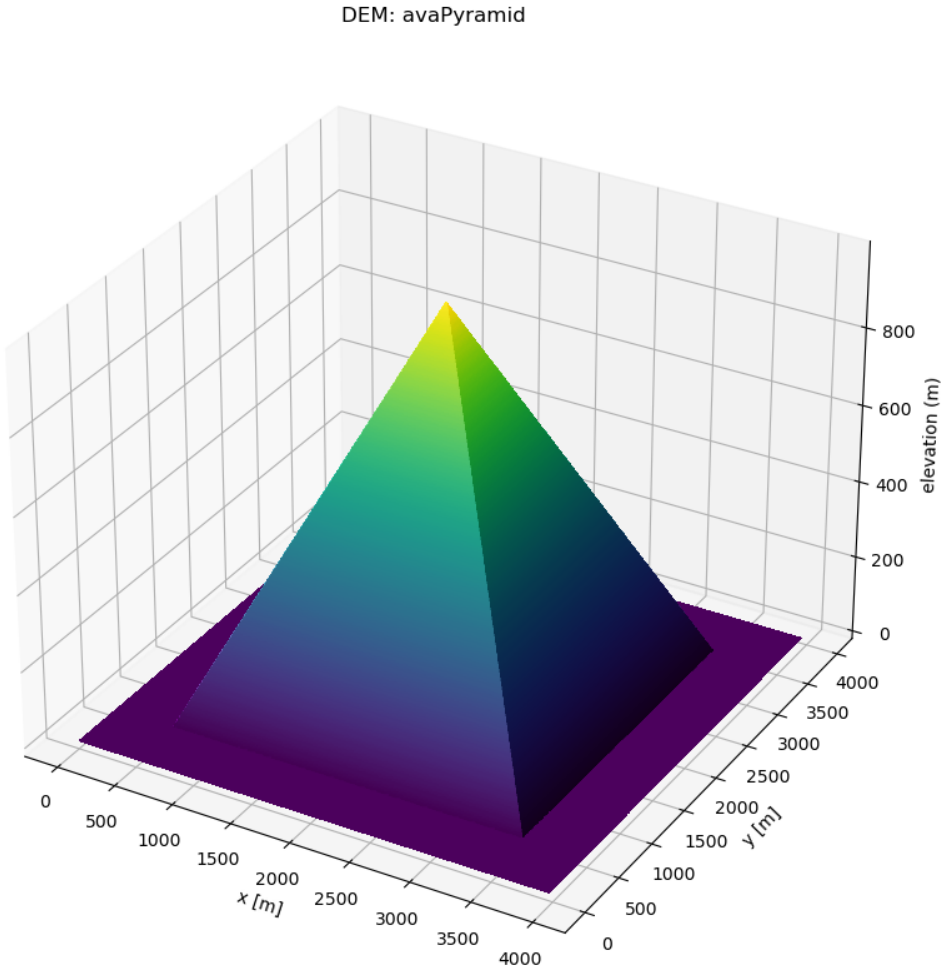

DEM: avaFlatPlane
0.04
0.02
0.00
−0.02
−0.04
elevation (m)
1000
2000
3000
4000
5000
6000
x [m]
−3000
−3250
−3500
−3750
−4000
−4250
−4500
−4750
−5000
y [m]


DEM: avaInclinedPlane
2000
1500
1000
500
0
−500
−1000
elevation (m)
−3000
−3250
−3500
−3750
−4000
−4250
−4500
−4750
−5000
y [m]
1000
2000
3000
4000
5000
6000
x [m]


DEM: avaParabola
2000
1750
1500
1250
1000
750
500
250
0
elevation (m)
−3000
−3250
−3500
−3750
−4000
−4250
−4500
−4750
−5000
y [m]
1000
2000
3000
4000
5000
6000
x [m]


DEM: avaHockeyChannel
2000
1750
1500
1250
1000
750
500
250
0
elevation (m)
−3000
−3250
−3500
−3750
−4000
−4250
−4500
−4750
−5000
y [m]
1000
2000
3000
4000
5000
6000
x [m]


DEM: avaHelixChannel
1000
800
600
400
200
0
elevation (m)
1000
2000
3000
4000
5000
6000
x [m]
−3000
−3250
−3500
−3750
−4000
−4250
−4500
−4750
−5000
y [m]

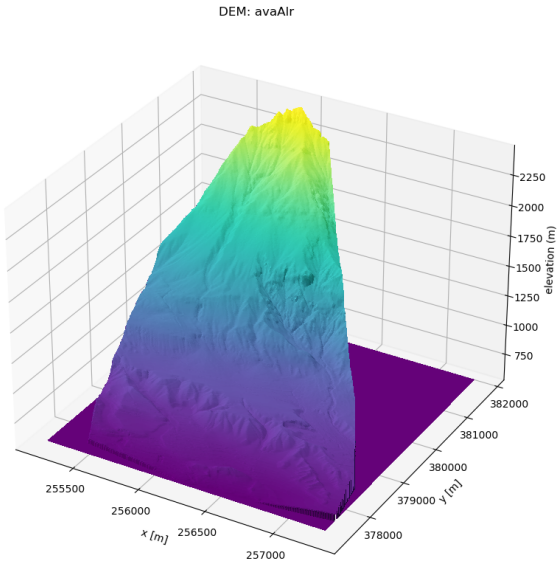

DEM: avaPyramid
elevation (m)
0
200
400
600
800
x [m]
0
500
1000
1500
2000
2500
3000
3500
4000
y [m]
0
500
1000
1500
2000
2500
3000
3500
4000

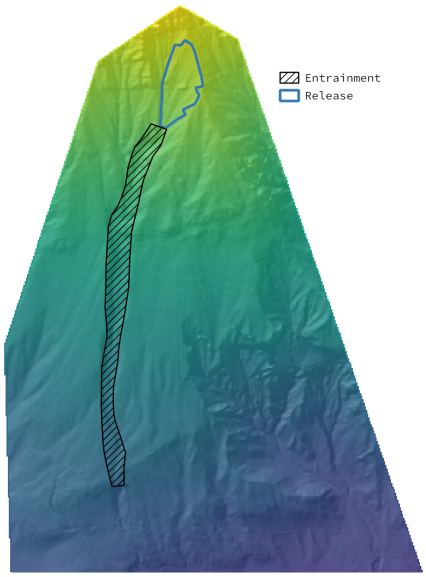
2.6.2.2 Real-world test cases

We provide a range of real-word topographies to perform avalanche simulations including different release area scenarios including entrainment areas. These can be found in `data/NameOfAvalanche`.

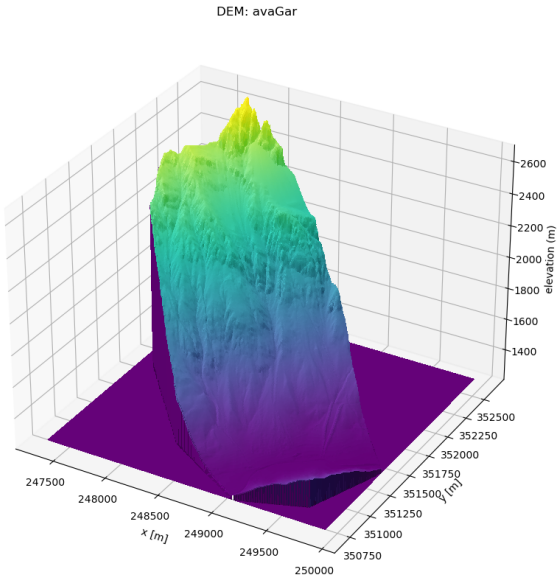
Note: All images give you a bigger version on click.

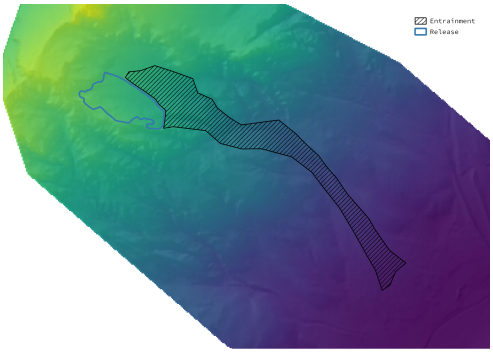
Alr



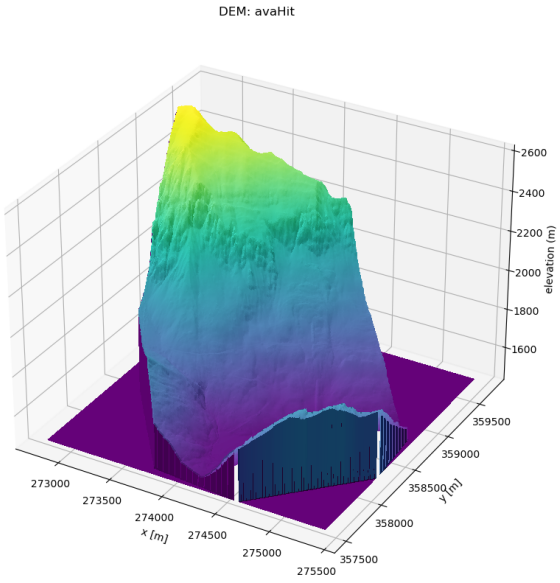


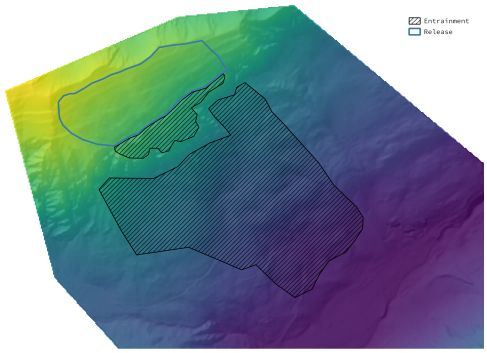
Gar


DEM: avaGar
2600
2400
2200
2000
1800
1600
1400
elevation (m)
352500
352250
352000
351750
351500
351250
351000
350750
y [m]
247500
248000
248500
249000
249500
250000
x [m]

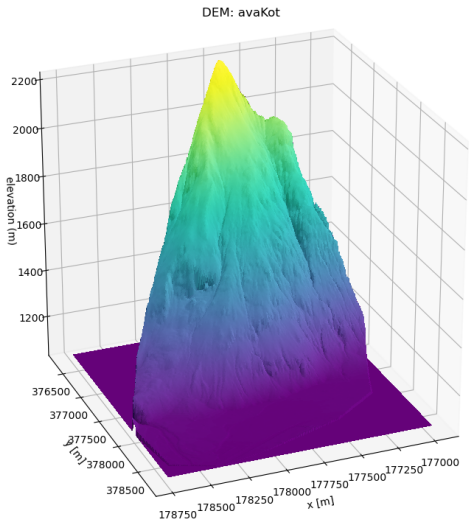

Entrainment
Release

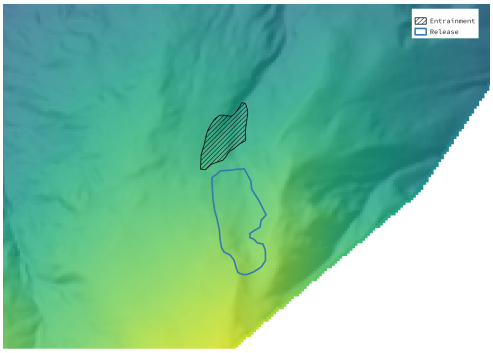
Hit


DEM: avaHit
2600
2400
2200
2000
1800
1600
elevation (m)
273000
273500
274000
274500
275000
275500
x [m]
357500
358000
358500
359000
359500
y [m]

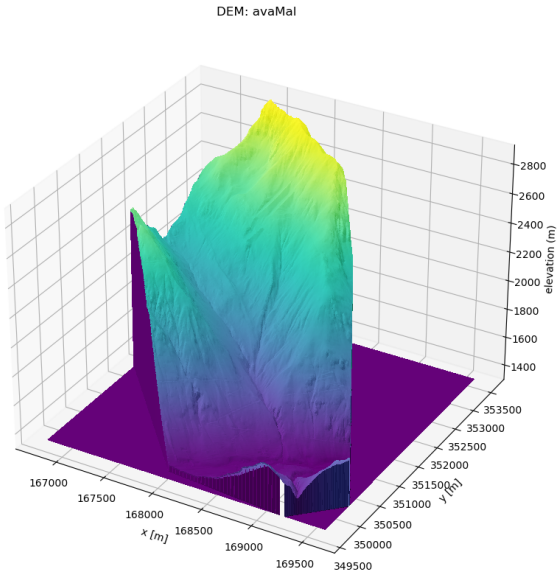

Entrainment
Release

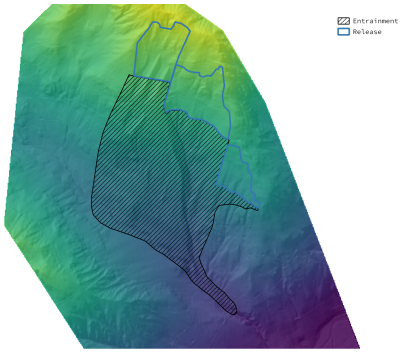
Kot


DEM: avaKot
elevation (m)
2200
2000
1800
1600
1400
1200
376500
377000
377500
378000
378500
y [m]
178750
178500
178250
178000
177750
177500
177250
177000
x [m]

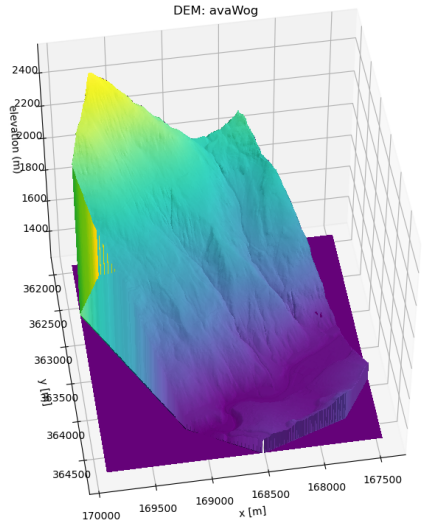

Entrainment
Release

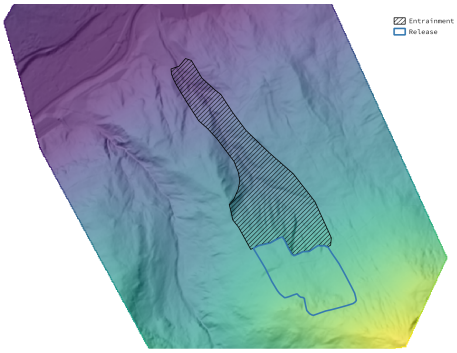
Mal


DEM: avaMal
elevation (m)
2800
2600
2400
2200
2000
1800
1600
1400
x [m]
167000
167500
168000
168500
169000
169500
y [m]
353500
353000
352500
352000
351500
351000
350500
350000
349500


Entrainment
Release

Wog





2.7 Data Visualisation

Main functions for creating visualisations of AvaFrame simulation results can be found in `out3Plot` and `out1Peak`. `com1DFA` also offers the possibility to export data on particle properties for visualisation using the open-source application ParaView. In order to start analysing particle data of `com1DFA`, follow these steps:

in your local copy of `com1DFA/com1DFACfg.ini`:

- *section [GENERAL]*: add *particles* to the *resType*
- *section [GENERAL]*: set which time steps you want to save at *tSteps*
- *section [VISUALISATION]*: set *writePartToCSV = True* and add the particles' properties you are interested in to the *particleProperties*

in ParaView

- open the particles file located in *data/avaDir/Outputs/com1DFA/particlesCSV* - there will be one available for each simulation performed containing the info on all saving time steps
- apply the filter **TableToPoints** and set X, Y, Z for the X, Y, Z columns
- switch to the **RenderView**: you can look at all the different particle properties you have saved for all exported time steps
- it is also possible to save an animation over all time steps

2.8 Frequently Asked Questions

Note: If you want to develop your own simulation workflow or develop new code we also suggest to have a look at the code examples provided in the *runScripts* directory.

2.8.1 Can the spatial resolution of simulations performed with com1DFA (dense flow) be changed?

The default setup of *com1DFA* is calibrated for medium to large dense flow snow avalanches in regards to hazard mapping. To provide calibrated results, new simulations with DEM cell sizes other than 5 meters are being remeshed by com1DFA to **5 meters** for computation.

It is possible to change the cell size, but keep in mind there are no tested/calibrated setups available. We want to list a few things that we strongly suggest to take into account when changing the mesh cell size:

- Start by changing the *meshCellSize* in the configuration file of *com1DFA* (`com1DFACfg.ini` or rather your local copy of it), and consider changing the *sphKernelRadius* to match the new mesh cell size.
- For the *SPH gradient*, required to compute the lateral forces, the number of particles per cell has to be adjusted. If the *sphKernelRadius* (see first point) is decreased, the number of particles should increase to ensure a reasonable estimate of the gradient. There are three options for setting the number of particles which is computing from the mass per particle (*massPerParticleDeterminationMethod*; see *Initialize particles*):
  - With the default setup *MPPDH* (mass per particle through release thickness), the number of particles per cell is independent of the mesh cell size. So no adjustment is necessary.
  - If *MPPKR* (mass per particles through number of particles per kernel radius) is chosen, the size of particles is adjusted to get a constant number of particles within an *sphKernelRadius*. This ensures a reasonable amount of particles for the gradient computation.
  - If *MPPDIR* (mass per particle set directly) is chosen, the number of particles per cell depends on the release snow mass within a cell. The number of particles is computed using the cell area and the release thickness. To ensure a reasonable amount of particles for decreased mesh size and *sphKernelRadius*, the *massPerParticle* value should be decreased accordingly. This ensures a reasonable amount of particles for the gradient computation.

Regarding DEM cell sizes: - if a cell size XX different than the default 5 meters is chosen - and the provided DEM in Inputs is of different mesh cell size than XX - the folder *DEMremeshed* will be checked for a matching DEM. - if no matching DEM is found -> remeshing takes place

Using the default settings, the subdirectory 'DEMremeshed' is cleaned when starting simulations, to ensure remeshing. This can be changed in the configuration file (further details are provided in *DEM input data*).

2.9 Release Notes

2.9.1 1.3 (12. Okt 2022)

Main change is the change of logic for secondary release areas. This was done to be able to expose this functions in the AvaFrameConnecter. So it is now possible to include secondary release areas in dense flow simulations done via QGis.

ENHANCEMENTS

- Change logic for secondary release areas: - Check for release and secondary release shapefiles - If only release available -> just run release - If both release and secondary release are available -> run with secondary release UNLESS secRelArea = False
  - ALL scenarios in release get the secondary release areas!
- Add rotational energy line test: helps to checks eg. for numerical grid independence
- Update ini file procedure for the energy line test and the rotation test
- Additional statistical plots
- New three panel plot of tt-diagram (plus animation)
- Add variation option for thickness settings and probrun based on normal distribution derived from ci and mean
- Add filtering option to aimec
- Add scenario name to configuration to be used for plotting example #757
- Add surface parallel coordinate computation to Aimec
- Improve operational installation instructions
- Add german version of operational installation

FIX - Contour legend bug with matplotlib 3.5.2 - Update installation instructions; fixes #764 - Bugs in deriving variation - Remeshing issue that lead to standard test differences (originated in commit 419c11f) - No calls to matplotlib for plotting purposes in com1DFA - Removes multiple printouts of config during run in, e.g. com1DFA - CompareConfig always honours the toPrint flag

Contributors:

Code: core team

2.9.2 1.2 (07. July 2022)

Main changes are the automatic split point generation and optional computation of fields inside the calculation loop. Furthermore renaming functions used for the QGis AvaFrameconnector are included.

ENHANCEMENTS

- Add function for renaming simulations, i.e. adding info to the simName. Used for AvaFrameConnector
- Split cfgUtils: Utils contains all reading and writing, cfgHandling contains functions that do something with the cfgInfos
- Make computation of ppr, pta, P, TA and pke optional within the calculation loop . Only compute them if they are required as output
- Add automatic split point generation: - First run a DFA simulation, either using the runDFAModule flag in runComputeDFAPath.py or yourself (do not forget that you need to saves particles or FD, FM, FV for multiple time steps)

  - What DFA parameters to use is not yet clear. I would use the BenMoussa time stepping and particles parameters, sphOption 2, explicit friction option 1, some artificial viscosity, and maybe activate curvature (with all this and a mu=0.42, I get ok results)
  - Then runComputeDFAPath computes the mass average path, extends it and defines a splitPoint

FIX

- Particle exiting domain was not working properly with the new cython flag (wraparound boundcheck…)
- correct hybrid Path ini file
- Update fields is too slow #455
- Automatic split point or beta point finding #706
- Add pfe to output files #694
- Make cflTime and sphKernelRadius a common option #668
- Change benchmarks not read particles from file (particles files still exist)
- Add consistency check for com1DFA configuration
- Fix problems with filtering of simulations
- Aimec comparison checks
- Add missing mass files to benchmarks

Contributors:

Code: core team

2.9.3 1.1 (19. May 2022)

The benchmark and thickness release. There are two main changes:

- The benchmarks (i.e. reference results) are updated and originate from com1DFA. Previously these were produced by com1DFAOrig. (Attached zip file contains the standardTest report for the switch)
- All references to *depth* are now switched to *thickness*. This is done to be more consistent and precise. It also means result types switch from *pfd / fd* (peak flow depth / flow depth) to *pft / ft* (peak flow thickness / flow thickness). Note that this is a *naming* change; nothing changed regarding the computation!

ENHANCEMENTS

- benchmarks originate from com1DFA
- change all depth variables to thickness
- change the order of simHash within the result names; fixes Move simHash in filename #690
- path finding added; see issue #610. This will be fully introduced in version 1.2, including automatic split point generation - refactor path computation functions - allow computing path from particles or fields (if *from fields*: needs the FM=FlowMass) - runscript to compute a path from com1DFA results (requires that one saves some time steps)
- automate the benchmark updating process
- improve energy line plot
- set deleteOutput to False in runOperational; addresses User Feedback (CT) #715. This means it is possible to reuse the same directory in the QGis Connector, adding results to existing ones
- add real area to aimec analysis #695

- update hybrid and energy test
- add com3Hybrid documentation #618

FIX

- hybrid model #611
- refactor com2AB for clarity and readability #446
- address savefigFormat TODO in outAB #560
- only one makeDomainTransfo #700

Contributors:

Code: core team

2.9.4 1.0.1 (20. April 2022)

FIX

- #712 , missing init files

2.9.5 1.0 (6. April 2022)

ENHANCEMENTS

- adds avaframe version to log
- appends date to logfile name
- update similarity solution plots
- re-add codecov
- add in addition to vary thickness values if read from shp - not just in percent but also in absolute value
- *ana1Test* energy line test
- *documentation* info on visualisation options (Paraview)
- update the pytest github action to version 3.9
- add ana5Hybrid, module that combines statistical module com2AB with the DFA module com1DFA
- new requirement shapely
- add release area info to benchmark ini files
- make AB optional in runOperational (related to QGis AvaFrameConnector)
- updates to ana1Tests
- hillshade and contours for peak plots
- documentation improvements
- reorder installation and get started documentation
- create distance-time diagrams of ava simulations from a reference point showing the avalanche front and the average values of a chosen result parameter (e.g. flow depth, flow velocity)
- *com1DFA* new flags/system for release thickness and entrainment thickness settings and options
- *com1DFA* add travel angle computation

- *com1DFA* release thickness percent variation option
- *com1DFA* unique simHash including info on release scenario with correct thickness
- *com1DFA* removed return parameters from com1DFAMain
- *com1DFA* update benchmark ini files
- *com1DFA* documentation for bottom friction and operator splitting
- *com1DFA* option to redistribute particles after initialisation in order to reduce SPH force
- *com1DFA* Implement Ata Viscosity and an SPH flow thickness computation
- *com1DFA* new splitting/merging of particles
- *com1DFA* enable to initialize particles with a non constant flow thickness
- *com1DFA* remove unmaintained leap frog time stepping scheme
- *com1DFA* new parameter: cleanDEMremeshed
- *com1DFA* add simulation DEM if remeshed to different cellSize #670
- *com1DFA* check for remeshed DEM, save remeshed DEM #675
- *com1DFA* enable to chose dem asc file for com1DFA #658
- *com1DFA* new parameter: cleanDEMremeshed
- *com1DFA* add simulation DEM if remeshed to different cellSize #670
- *com1DFA* check for remeshed DEM, save remeshed DEM #675
- *com1DFA* enable to chose dem asc file for com1DFA #658
- *ana4Prob* add example for performing a parameter variation run with prob analysis
- *ana4Prob* use default com module setup or specified in local - add variation for prob run
- *ana4Prob* perform analysis using probabilityConfiguration in runScript

FIX

- errors in com2AB documentation
- tcpu field in com1DFA
- ordering of dict for analysisAdd
- pytest errors related to matplotlib colors and legend
- particle splitting issue
- fix pypi related issues (pypi needs clean version tags)
- quickfix for shapely vs QGis problem with the AvaFrameConnector, see Linux QGis 3.24 crashes on Connector activation QGisAF#9
- move Release-version file for packaged releases
- change naming of log file: fix #689
- (hacky) solution to handle apostrophes in filenames #683
- allow choosing a tau0 in samosAT friction type (so far, tau0 was fixed and equal to 0)
- add tau0 to SamosAT friction #702
- address the wrong logName in runscript

- error running simulations one day after #701
- error on python 3.7 and QGis 3.12 #705
- python3-dev package required. #699

Contributors:

Code: core team, M. v. Busse (UIBK), M. Winkler (UIBK) Code review tt-diagram: A. Köhler (BFW)

2.9.6 v0.6 (24. September 2021)

ENHANCEMENTS

- installation via pypi (pip install)
- connection to QGis (via plugin manager)
- function to interpolate data on mesh of different cellSize using splinesp
- testing via pytest extended
- more pathlib usage
- ASCII header is read as dict
- documentation contains FAQ page
- reworked installation instructions
- cleaner test reports/inis
- github action to deploy to pypi
- switch to codeclimate
- use consistent thickness attributes (shapefiles etc)
- *com1DFA* any resolution is possible now
- *com1DFA* split the getWeight function in two: first get cell and then get weights.
- *com1DFA* avoid possibility of segfault because particles exit too quickly the domain.
- *com1DFA* additional particles info: unique identifier for each particle and parent particles
- *com1DFA* central time step calling
- *com1DFA* additional options to set mass per particle directly or via release thickness
- *com1DFA* interpolation option for initialization of Hpart
- *com1DFA* read entrainment thickness
- *ana3AIMEC* override option for raster cellsize
- *ana3AIMEC* mass analysis plot even if more than 2 simulations

FIX

- getTimeIndex problem if dtSave < actual dt
- better way to remove particles
- track particles exiting the computation domain
- fix issue save particles
- read aimec grid info from result files and not from dem

- add reasonString to removal of particles
- fix correct module name in AIMEC
- com2AB write out to shp

Contributors:

- **Code: core team**

2.9.7 v0.5 (13. July 2021)

ENHANCEMENTS

- filtering functions for com1DFA simulations
- flag to disable print at CFG reading
- new colormaps for ppr, pft, pfv
- *com1DFA* option to add friction explicitly using the method described in #273 .
- *com1DFA* Resistance force is added explicitly.
- *com1DFA* New method to get the release area
- *com2AB* function to write results to shapefile
- *ana3AIMEC* warning for empty runout zone
- *ana3AIMEC* enable simulation ordering/filtering

FIX

- beta angle issue i.e. distance below angle
- correct removal of particles
- AIMEC produces warning on empty runout area
- adapt quickplot to new naming scheme

Contributors:

- **Code: core team**
- **Colormaps: C.Tollinger**

DOI for this release:

2.9.8 v0.4.1 (9. June 2021)

Minor release to fix issue with zenodo

2.9.9 v0.4 (8. June 2021)

The switch release

This is a big release: we switched our dense flow module 'com1DFA' to the python version. This means that you know get to use the python version as default. However, the original version is still available in the module com1DFAOrig. The full documentation for the python com1DFA version as well as updated benchmarks will be released in the next version.

Module com2AB (AlphaBeta) recieved an update allowing for custom parameters.

Simulation naming and identification also recieved a major change, we introduced unique ID's for each individual configuration.

Contributors:

- **Code: core team**

2.9.10 v0.3 (26. April 2021)

The AIMEC and Windows release

This release brings an AIMEC refactor, plenty of improvements related to the test cases and Windows capabilities. 3 new idealised/generic test case are included: flat plane, inclined slope and pyramid.

Com1DFAPy recieved a lot of advancement as well, e.g. parts of it are converted to cython to speed up computation times.

Documentation regarding our testing is included, see more at the testing page.

Contributors:

- **Code: core team**

DOI for this release:

2.9.11 v0.2 (28. Dezember 2020)

The testing release

Version 0.2 includes the first real world avalanches. It provides data for 6 avalanches, including topographies, release areas and benchmark results. To know more about our data sources, head over to our data sources documentation. The existing test cases also recieved some updates by including multiple release areas and multiple scenarios per avalanche.

This release also is the first to include API documentation for our modules and functions. However not all functions are included yet.

Contributors:

- **Data: M.Granig, C. Tollinger**
- **Data: Land Tirol**
- **Code: core team**

2.9.12 v0.1 (06 November 2020)

Initial release.

This release is the result of several months of development.

Several people have contributed to this release, either directly or through code that was used as reference/basis:

- **Peter Sampl**, code base for com1DFA
- **Jan-Thomas Fischer**, code base AIMEC, code related to com1DFA
- **Michael Neuhauser**, code for helper and transformation utilities, com1DFA
- **Andreas Kofler**, code related to AIMEC and com1DFA

and the core team:

- **Anna Wirbel**
- **Matthias Tonnel**
- **Felix Oesterle**

2.10 API Reference

2.10.1 Computational Modules

com1DFA	Dense flow avalanche kernel (Python/cython)
com2AB	AlphaBeta kernel
com3Hybrid	Hybrid modelling: combining DFA simulation and statistical approaches

2.10.1.1 com1DFA

Dense flow avalanche kernel (Python/cython)

Modules

com1DFA.DFAtools	Basic tools for getting grid normals, area and working with vectors.
com1DFA.checkCfg	
com1DFA.com1DFA	
com1DFA.com1DFATools	
com1DFA.deriveParameterSet	Create dictionary for parameter variations
com1DFA.particleInitialisation	functions for initialising particle distribution
com1DFA.particleTools	
com1DFA.timeDiscretizations	Functions regarding time discretization and time stepping for com1DFA

com1DFA.DFAtools

Basic tools for getting grid normals, area and working with vectors.

Functions

crossProd	Compute cross product of vector u = (ux, uy, uz) and v = (vx, vy, vz).
getAreaMesh	Get area of grid cells.
getNormalArray	Interpolate vector field from grid to unstructures points
getNormalMesh	Compute normal to surface at grid points
norm	Compute the Euclidean norm of the vector (x, y, z).
norm2	Compute the square of the Euclidean norm of the vector (x, y, z).
normalize	Normalize vector (x, y, z) for the Euclidean norm.
scalProd	Compute scalar product of vector u = (ux, uy, uz) and v = (vx, vy, vz).

com1DFA.DFAtools.crossProd

crossProd(*ux*, *uy*, *uz*, *vx*, *vy*, *vz*)

Compute cross product of vector u = (ux, uy, uz) and v = (vx, vy, vz).

com1DFA.DFAtools.getAreaMesh

getAreaMesh(*dem*, *num*)

Get area of grid cells.

Parameters

- **dem dict updated with** –

  Nx: 2D numpy array
  x component of the normal vector field on grid points

  Ny: 2D numpy array
  y component of the normal vector field on grid points

  Nz: 2D numpy array
  z component of the normal vector field on grid points

  header
  [dict] dem header (cellsize)

- **num** (*int*) – chose between 4, 6 or 8 (using then 4, 6 or 8 triangles) or 1 to use the simple cross product method (with the diagonals)

Returns

areaRaster: 2D numpy array
real area of grid cells

Return type
dem dict updated with

com1DFA.DFAtools.getNormalArray

getNormalArray(*x*, *y*, *Nx*, *Ny*, *Nz*, *csz*)

Interpolate vector field from grid to unstructures points

Originaly created to get the normal vector at location (x,y) given the normal vector field on the grid. Grid has its origin in (0,0). Can be used to interpolate any vector field. Interpolation using a bilinear interpolation

Parameters

- **x** (*numpy array*) – location in the x location of desiered interpolation
- **y** (*numpy array*) – location in the y location of desiered interpolation
- **Nx** (*2D numpy array*) – x component of the vector field at the grid nodes
- **Ny** (*2D numpy array*) – y component of the vector field at the grid nodes
- **Nz** (*2D numpy array*) – z component of the vector field at the grid nodes
- **csz** (*float*) – cellsize of the grid

Returns

- **nx** (*numpy array*) – x component of the interpolated vector field at position (x, y)
- **ny** (*numpy array*) – y component of the interpolated vector field at position (x, y)
- **nz** (*numpy array*) – z component of the interpolated vector field at position (x, y)

com1DFA.DFAtools.getNormalMesh

getNormalMesh(*dem*, *num*)

Compute normal to surface at grid points

Get the normal vectors to the surface defined by a DEM. Either by adding the normal vectors of the adjacent triangles for each points (using 4, 6 or 8 adjacent triangles). Or use the next point in x direction and the next in y direction to define two vectors and then compute the cross product to get the normal vector

Parameters

- **dem** (*dict*) –

  header :
  dem header (cellsize, ncols, nrows)

  rasterData
  [2D numpy array] elevation at grid points

- **num** (*int*) – chose between 4, 6 or 8 (using then 4, 6 or 8 triangles) or 1 to use the simple cross product method (with the diagonals)

Returns

dem –

dem dict updated with:

Nx: 2D numpy array
x component of the normal vector field on grid points

Ny: 2D numpy array
y component of the normal vector field on grid points

Nz: 2D numpy array
z component of the normal vector field on grid points

outOfDEM: 2D boolean numpy array
True if the cell is out the dem, False otherwise

Return type
dict

com1DFA.DFAtools.norm

norm(*x*, *y*, *z*)

Compute the Euclidean norm of the vector (x, y, z).

(x, y, z) can be numpy arrays.

Parameters

- **x** (*numpy array*) – x component of the vector
- **y** (*numpy array*) – y component of the vector
- **z** (*numpy array*) – z component of the vector

Returns
norme – norm of the vector

Return type
numpy array

com1DFA.DFAtools.norm2

norm2(*x*, *y*, *z*)

Compute the square of the Euclidean norm of the vector (x, y, z).

(x, y, z) can be numpy arrays.

Parameters

- **x** (*numpy array*) – x component of the vector
- **y** (*numpy array*) – y component of the vector
- **z** (*numpy array*) – z component of the vector

Returns
norme2 – square of the norm of the vector

Return type
numpy array

com1DFA.DFAtools.normalize

normalize(*x*, *y*, *z*)

Normalize vector (x, y, z) for the Euclidean norm.

(x, y, z) can be np arrays.

Parameters

- **x** (*numpy array*) – x component of the vector
- **y** (*numpy array*) – y component of the vector
- **z** (*numpy array*) – z component of the vector

Returns

- **x** (*numpy array*) – x component of the normalized vector
- **y** (*numpy array*) – y component of the normalized vector
- **z** (*numpy array*) – z component of the normalized vector

com1DFA.DFAtools.scalProd

scalProd(*ux*, *uy*, *uz*, *vx*, *vy*, *vz*)

Compute scalar product of vector u = (ux, uy, uz) and v = (vx, vy, vz).

com1DFA.deriveParameterSet

Create dictionary for parameter variations

Functions

appendShpThickness	append thickness values to GENERAL section if read from shp and not varied
checkDEM	check if cell size of DEM in Inputs/ is same as desired meshCellSize if not - check for remeshed DEM or remesh the DEM
checkResType	Check if the resTypes asked for exist Warns the user if some do not, removes them from the resType list and updates the cfg
checkThicknessSettings	check if thickness setting format is correct
createSimDict	Create a simDict with all the simulations that shall be performed
getParameterVariationInfo	create a variation dictionary according to parameter variation given in cfg object
getThicknessValue	set thickness values according to settings chosen and add info to cfg if thFromShp = True - add in INPUT section thickness and id info and ci95 if thFromShp = False - check format of thickness value in GENERAL section
getVariationDict	Create a dictionary with all the parameters that shall be varied from the standard configuration; ONLY accounts for variations in section GENERAL and INPUT/releaseScenario
setPercentVariation	determine thickness value if set from percentVariation and set from shp file and update percentVariation value that is used for exactely this sim
setRangeFromCiVariation	determine thickness value if set from rangeFromCiVariation and set from shp file and update rangeFromCiVariation value that is used for exactely this sim
setRangeVariation	determine thickness value if set from rangeVariation and set from shp file and update rangeVariation value that is used for exactely this sim
setThicknessValueFromVariation	set thickness value for thickness parameter for all features if multiple according to desired variation
setVariationForAllFeatures	set thickness value for all features according to varType variation
splitVariationToArraySteps	split variation in percent to create a list of factors to set parameter value for variations or if a rangeVariation is given in absolute values or if distVariation create an info string on how the distribution can be build (e.g.
validateVarDict	Check if all parameters in variationDict exist in default configuration and are provided in the correct format
writeToCfgLine	write an array of values to a string of values separated by | for configuration

com1DFA.deriveParameterSet.appendShpThickness

appendShpThickness(*cfg*)

append thickness values to GENERAL section if read from shp and not varied

Parameters
cfg (*dict*) – configuration settings

Returns
cfg – updated configuartion settings

Return type
dict

com1DFA.deriveParameterSet.checkDEM

checkDEM(*cfgSim*, *demFile*, *onlySearch=False*)

check if cell size of DEM in Inputs/ is same as desired meshCellSize if not - check for remeshed DEM or remesh the DEM

Parameters

- **cfgSim** (*configparser object*) – configuration settings of com module
- **demFile** (*str or pathlib path*) – path to dem in Inputs/
- **onlySearch** (*bool*) – if True - only searching for remeshed DEM but not remeshing if not found

Returns
pathToDem – path to DEM with correct cellSize relative to Inputs/

Return type
str

com1DFA.deriveParameterSet.checkResType

checkResType(*fullCfg*, *section*, *key*, *value*)

Check if the resTypes asked for exist Warns the user if some do not, removes them from the resType list and updates the cfg

Parameters

- **fullCfg** (*configParser object*) – full configuration potentially including variations of parameter
- **section** (*str*) – section name
- **key** (*str*) – key name
- **value** (*str*) – corresponding value

Returns
fullCfg – full configuration updated with resType if this last one was modified

Return type
configParser object

com1DFA.deriveParameterSet.checkThicknessSettings

checkThicknessSettings(*cfg*, *thName*)

check if thickness setting format is correct

Parameters

- **cfg** (*configparser*) – configuration settings
- **thName** (*str*) – thickness parameter name (entTh, ...)

Returns

thicknessSettingsCorrect – True if settings are provided in the correct format

Return type

bool

com1DFA.deriveParameterSet.createSimDict

createSimDict(*avalancheDir*, *com1DFA*, *cfgInitial*, *inputSimFiles*, *simNameExisting*)

Create a simDict with all the simulations that shall be performed

Parameters

- **avalancheDir** (*pathlib path*) – path to avalanche directory
- **com1DFA** (*module*) – computational module
- **cfgStart** (*configparser object*) – configuration settings for com1DFA
- **inputSimFiles** (*dict*) – dictionary with info in input files (release area, dem, ...)
- **simNameExisting** (*list*) – list with names of sims that already exist in outputs

Returns

simDict – dicionary with info on simHash, releaseScenario, release area file path, simType and contains full configuration configparser object for simulation run

Return type

dict

com1DFA.deriveParameterSet.getParameterVariationInfo

getParameterVariationInfo(*avalancheDir*, *module*, *cfgStart*)

create a variation dictionary according to parameter variation given in cfg object

Parameters

- **avalancheDir** (*str or pathlib Path*) – path to avalanche directory
- **module** (*module*)
- **cfgStart** (*configparser object*) – full configuration object

Returns

- **modCfg** (*configparser object*) – configuration of simulations to be performed
- **variationDict** (*dict*) – dictionary with information on parameter variations

com1DFA.deriveParameterSet.getThicknessValue

`getThicknessValue`(*cfg*, *inputSimFiles*, *fName*, *thType*)

set thickness values according to settings chosen and add info to cfg if thFromShp = True - add in INPUT section thickness and id info and ci95 if thFromShp = False - check format of thickness value in GENERAL section

Parameters

- **cfg** (*configparser object*) – configuration settings
- **inputSimFiles** (*dict*) – dictionary with info on input files and attributes (id and thickness)
- **fName** (*str*) – name of scenario shp file (entrainment, release, ...)
- **thType** (*str*) – thickness parameter name (entTh, relTh, ...)

Returns

cfg – updated configuration settings with info on actual thickness values to be used for simulations

Return type

configparser object

com1DFA.deriveParameterSet.getVariationDict

`getVariationDict`(*avaDir*, *fullCfg*, *modDict*)

Create a dictionary with all the parameters that shall be varied from the standard configuration; ONLY accounts for variations in section GENERAL and INPUT/releaseScenario

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **fullCfg** (*configParser object*) – full configuration potentially including variations of parameter
- **modDict** (*dict*) – info on modifications to standard configuration

Returns

variationDict – dictionary with the parameters that shall be varied and the chosen flags for the run

Return type

dict

com1DFA.deriveParameterSet.setPercentVariation

`setPercentVariation`(*cfg*, *variationFactor*, *thNameId*)

determine thickness value if set from percentVariation and set from shp file and update percentVariation value that is used for exactely this sim

this is required for reproducing this sim when using its configuration file - so that in the percentVariation parameter it is only one value e.g. +50$1 so +50% in 1 step

Parameters

- **cfg** (*configparser object*) – comModule configuration file with info on thickness settings
- **variationFactor** (*float*) – value of percent variation in terms of required multiplication of reference value (e.g. variationFactor= 0.5 - a variation of 50% performed by multiplication of reference value times variationFactor)

- **thNameId** (*str*) – name of thickness feature (e.g. relTh0 if release thickness and feature with id 0)

Returns
variationIni – percentVariation parameter value for this sim to be added in cfg file

Return type
str

com1DFA.deriveParameterSet.setRangeFromCiVariation

`setRangeFromCiVariation`(*cfg*, *variationFactor*, *thValue*, *ciValue*)

determine thickness value if set from rangeFromCiVariation and set from shp file and update rangeFromCiVariation value that is used for exactely this sim

this is required for reproducing this sim when using its configuration file - so that in the rangeFromCiVariation parameter it is only one value e.g. ci95$4$1 so -ci95m

Parameters

- **cfg** (*configparser object*) – comModule configuration file with info on thickness settings
- **variationFactor** (*str*) – value of range variation in terms of required addition to the reference value (e.g. variationFactor= chi95$4$0 - a variation of -ci95 value performed by addition of reference value + the variation value)
- **thValue** (*str*) – thickness value of thickness feature in meter
- **ciValue** (*str*) – ci value of thickness features in meter

Returns

- **variationValue** (*float*) – actual thickness value modified according to variation for feature
- **variationIni** (*str*) – rangeFromCiVariation parameter value for this sim to be added in cfg file

com1DFA.deriveParameterSet.setRangeVariation

`setRangeVariation`(*cfg*, *variationFactor*, *thNameId*)

determine thickness value if set from rangeVariation and set from shp file and update rangeVariation value that is used for exactely this sim

this is required for reproducing this sim when using its configuration file - so that in the rangeVariation parameter it is only one value e.g. +0.5$1 so +0.5m in 1 step

Parameters

- **cfg** (*configparser object*) – comModule configuration file with info on thickness settings
- **variationFactor** (*float*) – value of range variation in terms of required addition to the reference value (e.g. variationFactor= +0.5 - a variation of +0.5m performed by addition of reference value + variationFactor)
- **thNameId** (*str*) – name of thickness feature (e.g. relTh0 if release thickness and feature with id 0)

Returns
variationIni – rangeVariation parameter value for this sim to be added in cfg file

Return type
str

com1DFA.deriveParameterSet.setThicknessValueFromVariation

`setThicknessValueFromVariation`(*key*, *cfg*, *simType*, *row*)

set thickness value for thickness parameter for all features if multiple according to desired variation

Parameters

- **key** (*str*) – thickness variation info
- **cfg** (*configparser object*) – configuration settings of comModule
- **simType** (*str*) – simulation type (null, ent, entres, ..)
- **row** (*pandas row*) – info on variation of parameters

Returns
cfg – updated dict with info on thickness

Return type
dict

com1DFA.deriveParameterSet.setVariationForAllFeatures

`setVariationForAllFeatures`(*cfg*, *key*, *thType*, *varType*, *variationFactor*)

set thickness value for all features according to varType variation

Parameters

- **cfg** (*configparser*) – configuration settings of comModule for thickness
- **key** (*str*) – name of parameter
- **thType** (*str*) – thickness type (e.g. relTh, entTh, ...)
- **varType** (*str*) – type of variation (range or percent)
- **variationFactor** (*float or str (if Dist or rangeFromCi)*) – value used for variation

Returns
cfg – updated configuration settings regarding thickness settings

Return type
configparser

com1DFA.deriveParameterSet.splitVariationToArraySteps

`splitVariationToArraySteps`(*value*, *key*, *fullCfg*)

split variation in percent to create a list of factors to set parameter value for variations or if a rangeVariation is given in absolute values or if distVariation create an info string on how the distribution can be build (e.g. of format typeOfDistribution$numberOfSteps$ci95value$ci95$support and append the step of the current variation in front)

Parameters

- **value** (*str*) – value read from configuration

- **key** (*str*) – name of parameter
- **fullCfg** (*configparser*) – full configuration settings

Returns
itemsArray – factor to change parameter values by multiplication (Percent) or addition (Range) or info string on how to build the distribution and which step to draw from it

Return type
numpy array

com1DFA.deriveParameterSet.validateVarDict

`validateVarDict`(*variationDict*, *standardCfg*)

Check if all parameters in variationDict exist in default configuration and are provided in the correct format

Parameters

- **variationDict** (*dict*) – dictionary with parameters that shall be varied and a list for the parameter values for each parameter
- **standardCfg** (*config Parser object*) – default model configuration

Returns
variationDict – cleaned variation dict that meets the required structure

Return type
dict

com1DFA.deriveParameterSet.writeToCfgLine

`writeToCfgLine`(*values*)

write an array of values to a string of values separated by | for configuration

Parameters
values (*numpy array*) – array of values

Returns
valString – string of array values separated by |

Return type
str

com1DFA.particleInitialisation

functions for initialising particle distribution

Functions

createReleaseBuffer	add a buffer around release polygons to get boundary particles
getIniPosition	Redistribute particles so that SPH force reduces with fixed particles as boundaries
resetMassPerParticle	recompute mass of particles according to their location with respect to relRaster

com1DFA.particleInitialisation.createReleaseBuffer

createReleaseBuffer(*cfg*, *inputSimLines*)

add a buffer around release polygons to get boundary particles

Parameters

- **cfg** (*configparser object*) – configuration settings
- **inputSimLines** (*dict*) – dictionary with input data info

Returns

inputSimLines – updated inputSimLines with releaseLineBuffer

Return type

dict

com1DFA.particleInitialisation.getIniPosition

getIniPosition(*cfg*, *particles*, *dem*, *fields*, *inputSimLines*, *relThField*)

Redistribute particles so that SPH force reduces with fixed particles as boundaries

Parameters

- **cfg** (*configparser object*) – configuration settings
- **particles** (*dict*) – dictionary with particle properties
- **dem** (*dict*) – dictionary with dem header and data
- **fields** (*dict*) – dictionary with fields of result types
- **inputSimLines** (*dict*) – info on input files
- **relThField** (*numpy array*) – release thickness field

Returns

- **particles** (*dict*) – updated particles dict with new positions
- **fields** (*fields*) – updated fields dictionary

com1DFA.particleInitialisation.resetMassPerParticle

resetMassPerParticle(*cfg*, *particles*, *dem*, *relRaster*, *relThField*)

recompute mass of particles according to their location with respect to relRaster

Parameters

- **cfg** (*configparser object*) – configuration settings
- **particles** (*dict*) – dictionary with particles properties
- **dem** (*dict*) – dictionary with info on dem
- **relRaster** (*np.array*) – raster of release thickness values

Returns

particles – updated particles dictionary with new mass

Return type

dict

com1DFA.timeDiscretizations

Functions regarding time discretization and time stepping for com1DFA

Functions

getSphKernelRadiusTimeStep	Compute the time step given the sph kernel radius and the cMax coefficient This is based on the article from Ben Moussa et Vila DOI:10.1137/S0036142996307119

com1DFA.timeDiscretizations.getSphKernelRadiusTimeStep

getSphKernelRadiusTimeStep(*dem*, *cfg*)

Compute the time step given the sph kernel radius and the cMax coefficient This is based on the article from Ben Moussa et Vila DOI:10.1137/S0036142996307119

Parameters

- **dem** (*dict*) – dem dictionary (with info about sph kernel radius and mesh size)
- **cfg** (*configparser*) – the cfg cith cMax

Returns

dtStable – corresponding time step

Return type

float

2.10.1.2 com2AB

AlphaBeta kernel

Modules

com2AB.com2AB	Main module for Alpha beta

com2AB.com2AB

Main module for Alpha beta

Functions

calcABAngles	Kernel function that computes the AlphaBeta model (angular results) for a given avaProfile and eqParameters
calcABDistances	Compute runout distances and points from angles computed in calcABAngles
com2ABKern	Compute AlpahBeta model for a given avapath
com2ABMain	Main AlphaBeta model function
readABinputs	Fetch inputs for AlpahBeta model
setEqParameters	Set alpha beta equation parameters

com2AB.com2AB.calcABAngles

calcABAngles(*avaProfile*, *eqParameters*, *dsMin*)

Kernel function that computes the AlphaBeta model (angular results) for a given avaProfile and eqParameters

Parameters

- **avaProfile** (*dict*) – dictionary with the name of the avapath, the x, y and z coordinates of the path
- **eqParameters** (*dict*) – AB parameter dictionary
- **dsMin** (*float*) – threshold distance [m] when looking for the Beta point

Returns

avaProfile – updated avaProfile with alpha, beta and other values resulting from the AlphaBeta model computation

Return type

dict

com2AB.com2AB.calcABDistances

calcABDistances(*avaProfile*, *name*)

Compute runout distances and points from angles computed in calcABAngles

Parameters

- **avaProfile** (*dict*) – dictionary with the name of the avapath, the x, y and z coordinates of the path
- **name** (*str*) – profile name

Returns

avaProfile – updated avaProfile with s index of alpha, and alphaSD points

Return type

dict

com2AB.com2AB.com2ABKern

com2ABKern(*avaPath*, *splitPoint*, *dem*, *eqParams*, *distance*, *dsMin*)

Compute AlpahBeta model for a given avapath

Call calcABAngles to compute the AlphaBeta model given an input raster (of the dem), an avalanche path and split points

Parameters

- **avaPath** (*dict*) – dictionary with the name of the avaPath, the x and y coordinates of the path
- **splitPoint** (*dict*) – dictionary split points
- **dem** (*dict*) – dem dictionary used to get the avaProfile from the avaPath
- **eqParams** (*dict*) – dict containing the AB model parameters (produced by setEqParameters and depends on the com2ABCfg.ini)
- **distance** (*float*) – line resampling distance
- **dsMin** (*float*) – threshold distance [m] when looking for the beta point

Returns

avaProfile – avaPath dictionary with AlphaBeta model results (path became a profile adding the z and s arrays. AB runout angles and distances)

Return type

dict

com2AB.com2AB.com2ABMain

com2ABMain(*cfg*, *avalancheDir*)

Main AlphaBeta model function

Loops on the given AvaPaths and runs com2AB to compute AlpahBeta model

Parameters

- **cfg** (*configparser*) – configparser with all requiered fields in com2ABCfg.ini
- **avalancheDir** (*str*) – path to directory of avalanche to analyze

Returns

- **pathDict** (*dict*) – dictionary with AlphaBeta inputs
- **dem** (*dict*) – dem dictionary used to get the avaProfile from the avaPath
- **splitPoint** (*dict*) – split point dict
- **eqParams** (*dict*) – dict containing the AB model parameters (produced by setEqParameters and depends on the com2ABCfg.ini)
- **resAB** (*dict*) – dictionary with AlphaBeta model results

com2AB.com2AB.readABinputs

`readABinputs`(*avalancheDir*, *path2Line=''*, *path2SplitPoint=''*)

Fetch inputs for AlpahBeta model

Get path to AlphaBeta model inputs (dem raster, avalanche path and split points)

Parameters

- **avalancheDir** (*str*) – path to directory of avalanche to analyze
- **path2Line** (*pathlib path*) – pathlib path to altrnative line (if empty, reading the line from the input directory Inputs/LINES/yourNameAB.shp)
- **path2SplitPoint** (*pathlib path*) – pathlib path to altrnative splitPoint (if empty, reading the point from the input directory Inputs/LINES/yourNameAB.shp)

Returns
pathDict – dictionary with path to AlphaBeta inputs (dem, avaPath, splitPoint)

Return type
dict

com2AB.com2AB.setEqParameters

`setEqParameters`(*cfg*, *smallAva*)

Set alpha beta equation parameters

Set alpha beta equation parameters to - standard (default) - small avalanche (if smallAva==True) - custom (if cfgsetup('customParam') is True use the custom values provided in the ini file)

Parameters

- **cfg** (*configParser*) – if cfgsetup('customParam') is True, the custom parameters provided in the cfg are used in theAlphaBeta equation. (provide all 5 k1, k2, k3, k4 and SD values)
- **smallAva** (*boolean*) – True if the small avallanche AlphaBeta equation parameters should be used

Returns
eqParameters – k1, k2, k3, k4 and SD values to be used in the AlphaBeta equation

Return type
dict

2.10.1.3 com3Hybrid

Hybrid modelling: combining DFA simulation and statistical approaches

Modules

com3Hybrid.com3Hybrid	

2.10.2 Input/Transformation Modules

in1Data	Tools for fetching or generating input data
in2Trans	Tools for input transformations
in3Utils	Tools for geo transformations

2.10.2.1 in1Data

Tools for fetching or generating input data

Modules

in1Data.computeFromDistribution	Generate sample of values following a beta-pert or a uniform distribution
in1Data.getInput	Fetch input data for avalanche simulations

in1Data.computeFromDistribution

Generate sample of values following a beta-pert or a uniform distribution

Functions

computeParameters	Compute alpha, beta and mu
computePert	Compute the CDF and PDF of the Pert distribution using scipy betainc function
extractFromCDF	Extract a sample from the CDF with prescribed steps
extractNormalDist	create a normal distribution from given parameters and draw a sample
extractUniform	Extract sample of a uniform distriution
getEmpiricalCDF	Derive empirical CDF using numpy histogram and cumsum
getEmpiricalCDFNEW	Derive empirical CDF using sorted sample

in1Data.computeFromDistribution.computeParameters

computeParameters(*a*, *b*, *c*)

Compute alpha, beta and mu

in1Data.computeFromDistribution.computePert

computePert(*a*, *b*, *c*, *x*, *alpha*, *beta*)

Compute the CDF and PDF of the Pert distribution using scipy betainc function

in1Data.computeFromDistribution.extractFromCDF

extractFromCDF(*CDF*, *CDFint*, *x*, *cfg*)

Extract a sample from the CDF with prescribed steps

in1Data.computeFromDistribution.extractNormalDist

extractNormalDist(*cfg*)

create a normal distribution from given parameters and draw a sample

Parameters

cfg (*configparser object or dict*) – configuration settings for computing distributions, std or ci95, minMaxInterval, flagMinMax

Returns

- **CDFint** (*function object*) – interpolated CDF function
- **sampleVect** (*numpy array*) – sample values
- **pdf** (*numpy array*) – pdf values computed for normal distribution and support x
- **x** (*numpy array*) – support x

in1Data.computeFromDistribution.extractUniform

extractUniform(*a*, *c*, *x*, *cfg*)

Extract sample of a uniform distriution

in1Data.computeFromDistribution.getEmpiricalCDF

getEmpiricalCDF(*sample*)

Derive empirical CDF using numpy histogram and cumsum

in1Data.computeFromDistribution.getEmpiricalCDFNEW

getEmpiricalCDFNEW(*sample*)

Derive empirical CDF using sorted sample

in1Data.getInput

Fetch input data for avalanche simulations

Functions

checkForMultiplePartsShpArea	check if in polygon read from shape file holes are present, if so error and save a plot to Outputs/com1DFA procedure: check if polygon has several parts
computeAreasFromLines	compute the area of a polygon in xy using shapely
computeAreasFromRasterAndLine	compute the area covered by a polygon by creating a raster from polygon projected area and actual area using a dem info
computeRelStats	compute stats of a polygon and a dem actual area (taking slope into account), projected area, max, mean, min elevation, mean slope
createReleaseStats	create a csv file with info on release shp file on: max, mean and min elevation, slope and projected and real area
fetchReleaseFile	select release scenario, update configuration to only include thickness info of current scenario and return file path
getAndCheckInputFiles	Fetch fileExt files and check if they exist and if it is not more than one
getDEMFromConfig	get dem file path in avaDir/Inputs
getDEMPath	get the DEM file path from a provided avalanche directory
getInputData	Fetch input datasets required for simulation, duplicated function because
getInputDataCom1DFA	Fetch input datasets required for simulation, duplicated function because now fetch all available files simulation type set differently in com1DFA compared to com1DFAOrig: TODO: remove duplicate once it is not required anymore
getInputPaths	Fetch paths to dem and first release area shp file found
getThicknessInputSimFiles	add thickness of shapefiles to dictionary
initializeDEM	check for dem and load to dict
readDEM	read the ascii DEM file from a provided avalanche directory
selectReleaseFile	select release scenario
updateThicknessCfg	add available release scenarios to ini file and set thickness values in ini files

in1Data.getInput.checkForMultiplePartsShpArea

checkForMultiplePartsShpArea(*avaDir*, *lineDict*, *modName*, *type=''*)

check if in polygon read from shape file holes are present, if so error and save a plot to Outputs/com1DFA procedure: check if polygon has several parts

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **lineDict** (*dict*) – dictionary with info read from shape file used: x, y, Start, Length, nParts, nFeatures
- **modName** (*str*) – name of computational module where to save error plots to Outputs sub-folder
- **type** (*str*) – type of shp file area (release, secondary release, entrainment, resistance)

Returns

- *error if number of parts is bigger 1*
- *save plot showing all parts of each polygon feature in Outputs/modName*

in1Data.getInput.computeAreasFromLines

computeAreasFromLines(*line*)

compute the area of a polygon in xy using shapely

Parameters
line (*dict*) – dictionary with info on line x, y coordinates start, end of each line feature

Returns
projectedAreas – list of projected area for each polygon in line dict

Return type
list

in1Data.getInput.computeAreasFromRasterAndLine

computeAreasFromRasterAndLine(*line*, *dem*)

compute the area covered by a polygon by creating a raster from polygon projected area and actual area using a dem info

Parameters

- **line** (*dict*) – dictionary with info on line x, y coordinates start, end of each line feature
- **dem** (*dict*) – dictionary with dem data, header and areaRaster

Returns

- **areaActual** (*float*) – actual area taking slope into account by using dem area
- **areaProjected** (*float*) – projected area in xy plane

in1Data.getInput.computeRelStats

`computeRelStats`(*line*, *dem*)

compute stats of a polygon and a dem actual area (taking slope into account), projected area, max, mean, min elevation, mean slope

Parameters

- **line** (*dict*) – dictionary with info on line (x, y, Start, Length, rasterData, ...)
- **dem** (*dict*) – dictionary with info on dem (header, rasterData, normals, areaRaster)

Returns

lineDict – dictionary with stats info: featureNames, zMax, zMin, Slope, Area, AreaP

Return type

dict

in1Data.getInput.createReleaseStats

`createReleaseStats`(*avaDir*, *cfg*)

create a csv file with info on release shp file on: max, mean and min elevation, slope and projected and real area

Returns

fPath – file path

Return type

pathlib Path

in1Data.getInput.fetchReleaseFile

`fetchReleaseFile`(*inputSimFiles*, *releaseScenario*, *cfgSim*, *releaseList*)

select release scenario, update configuration to only include thickness info of current scenario and return file path

Parameters

- **inputSimFiles** (*dict*) – dictionary with info on input data
- **releaseScenario** (*str*) – name of release scenario
- **cfgSim** (*conigparser object*) – configuration of simulation
- **releaseList** (*list*) – list of available release scenarios

Returns

- **releaseScenarioPath** (*pathlib path*) – file path to release scenario shp file
- **cfgSim** (*configparser object*) – updated cfg object, removed thickness info from not other release scenarios than used one and rename thickness values of chosen scenario to relThThickness, relThId, ...

in1Data.getInput.getAndCheckInputFiles

getAndCheckInputFiles(*inputDir*, *folder*, *inputType*, *fileExt='shp'*)

Fetch fileExt files and check if they exist and if it is not more than one

Raises error if there is more than one fileExt file.

Parameters

- **inputDir** (*pathlib object or str*) – path to avalanche input directory
- **folder** (*str*) – subfolder name where the shape file should be located (SECREL, ENT or RES)
- **inputType** (*str*) – type of input (used for the logging messages). Secondary release or Entrainment or Resistance
- **fileExt** (*str*) – file extension e.g. shp, asc - optional default is shp

Returns

- **OutputFile** (*str*) – path to file checked
- **available** (*str*) – Yes or No depending of if there is a shape file available (if No, OutputFile is None)

in1Data.getInput.getDEMFromConfig

getDEMFromConfig(*avaDir*, *fileName=''*)

get dem file path in avaDir/Inputs

Parameters

- **avaDir** (*str or pathlib path*) – to avalancheDir
- **fileName** (*pathlib path*) – to dem file with filename in avaDir/Inputs

Returns
demFile – to dem file

Return type
pathlib path

in1Data.getInput.getDEMPath

getDEMPath(*avaDir*)

get the DEM file path from a provided avalanche directory

Parameters
avaDir (*str*) – path to avalanche directory

Returns
demFile – full path to DEM .asc file

Return type
str (first element of list)

in1Data.getInput.getInputData

`getInputData`(*avaDir*, *cfg*)

Fetch input datasets required for simulation, duplicated function because
simulation type set differently in com1DFAOrig compared to com1DFA: TODO: remove duplicate once it is not required anymore

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **cfg** (*dict*) – configuration read from com1DFA simulation ini file

Returns

- **demFile[0]** (*str (first element of list)*) – list of full path to DEM .asc file
- **relFiles** (*list*) – list of full path to release area scenario .shp files
- **entFile** (*str*) – full path to entrainment area .shp file
- **resFile** (*str*) – full path to resistance area .shp file
- **wallFile** (*str*) – full path to wall line .shp file
- **entResInfo** (*flag dict*) – flag if Yes entrainment and/or resistance areas found and used for simulation

in1Data.getInput.getInputDataCom1DFA

`getInputDataCom1DFA`(*avaDir*)

Fetch input datasets required for simulation, duplicated function because now fetch all available files simulation type set differently in com1DFA compared to com1DFAOrig: TODO: remove duplicate once it is not required anymore

Parameters
avaDir (*str or pathlib object*) – path to avalanche directory

Returns

inputSimFiles – dictionary with all the input files

- demFile : str (first element of list), list of full path to DEM .asc file
- relFiles : list, list of full path to release area scenario .shp files
- secondaryReleaseFile : str, full path to secondary release area .shp file
- entFile : str, full path to entrainment area .shp file
- resFile : str, full path to resistance area .shp file
- entResInfo : flag dict

flag if Yes entrainment and/or resistance areas found and used for simulation flag True if a Secondary Release file found and activated

Return type
dict

in1Data.getInput.getInputPaths

`getInputPaths`(*avaDir*)

Fetch paths to dem and first release area shp file found

Parameters
avaDir (*str or pathlib object*) – path to avalanche directory

Returns

- **demFile** (*pathlib path*) – full path to DEM .asc file
- **relFiles** (*list*) – list of full paths to release area scenario .shp files found in avaDir/Inputs/REL
- **relFieldFiles** (*list*) – list of full paths to release area thickness .asc files found in avaDir/Inputs/RELTH

in1Data.getInput.getThicknessInputSimFiles

`getThicknessInputSimFiles`(*inputSimFiles*)

add thickness of shapefiles to dictionary

Parameters
inputSimFiles (*dict*) – dictionary with info on release and entrainment file paths

Returns
inputSimFiles – updated dictionary with thickness info read from shapefile attributes now includes one separate dictionary for each release, entrainment or secondary release scenario with a thickness and id value for each feature (given as list)

Return type
dict

in1Data.getInput.initializeDEM

`initializeDEM`(*avaDir*, *demPath=''*)

check for dem and load to dict

Parameters

- **avaDir** (*str or pathlib path*) – path to avalanche directory
- **demPath** (*str or pathlib Path*) – path to dem relative to Inputs - optional if not provided read DEM from Inputs

Returns
demOri – dem dictionary with header and data

Return type
dict

in1Data.getInput.readDEM

readDEM(*avaDir*)

read the ascii DEM file from a provided avalanche directory

Parameters
avaDir (*str*) – path to avalanche directory

Returns
dem – dict with header and raster data

Return type
dict

in1Data.getInput.selectReleaseFile

selectReleaseFile(*inputSimFiles*, *releaseScenario*)

select release scenario

Parameters

- **inputSimFiles** (*dict*) – dictionary with info on input data
- **releaseScenario** (*str*) – name of release scenario

Returns
inputSimFiles – dictionary with info on input data updated with releaseScenario

Return type
dict

in1Data.getInput.updateThicknessCfg

updateThicknessCfg(*inputSimFiles*, *cfgInitial*)

add available release scenarios to ini file and set thickness values in ini files

Parameters

- **inputSimFiles** (*dict*) – dictionary with info on release and entrainment file paths
- **cfgInitial** (*configParser object*) – configParser object with the current (and possibly overridden) configuration

Returns

- **inputSimFiles** (*dict*) – updated dictionary with thickness info read from shapefile attributes now includes one separate dictionary for each release, entrainment or secondary release scenario with a thickness and id value for each feature (given as list)
- **cfgInitial** (*configparser object*) – updated config object with release scenario, thickness info, etc.

2.10.2.2 in2Trans

Tools for input transformations

Modules

in2Trans.ascUtils	ASCII file reader and handler
in2Trans.shpConversion	Conversion functions to read/ write Shape files

in2Trans.ascUtils

ASCII file reader and handler

Functions

isEqualASCheader	Test if two headers (A,B) are the same (except for noData Values)
readASCdata2numpyArray	Read ascii matrix as numpy array
readASCheader	return a class with information from an ascii file header
readRaster	Read raster file (.asc)
writeResultToAsc	Write 2D array to an ascii file with header and save to location of outFileName

in2Trans.ascUtils.isEqualASCheader

isEqualASCheader(*headerA*, *headerB*)

Test if two headers (A,B) are the same (except for noData Values)

Parameters

- **headerA** (*class*)
- **headerB** (*class*)

Returns
boolean

Return type
True if header A and B are equal (disregrad the noData field)

in2Trans.ascUtils.readASCdata2numpyArray

readASCdata2numpyArray(*fName*)

Read ascii matrix as numpy array

Parameters
fname (*str or pathlib object*) – path to ascii file

Returns
-rasterdata – 2D numpy array of ascii matrix

Return type
2D numpy array

in2Trans.ascUtils.readASCheader

`readASCheader`(*fname*)

return a class with information from an ascii file header

Parameters
fname (*str or pathlib object*) – path to ascii file

Returns
headerInfo – information that is stored in header (ncols, nrows, xllcenter, yllcenter, nodata_value)

Return type
class

in2Trans.ascUtils.readRaster

`readRaster`(*fname*, *noDataToNan=True*)

Read raster file (.asc)

Parameters

- **fname** (*str or pathlib object*) – path to ascii file
- **noDataToNan** (*bool*) – if True convert nodata_values to nan and set nodata_value to nan

Returns

data –

-headerInfo: class
information that is stored in header (ncols, nrows, xllcenter, yllcenter, nodata_value)

-rasterdata
[2D numpy array] 2D numpy array of ascii matrix

Return type
dict

in2Trans.ascUtils.writeResultToAsc

`writeResultToAsc`(*header*, *resultArray*, *outFileName*, *flip=False*)

Write 2D array to an ascii file with header and save to location of outFileName

Parameters

- **header** (*class*) – class with methods that give cellsize, nrows, ncols, xllcenter yllcenter, nodata_value
- **resultArray** (*2D numpy array*) – 2D numpy array of values that shall be written to file
- **outFileName** (*str*) – path incl. name of file to be written
- **flip** (*boolean*) – if True, flip the rows of the resultArray when writing

in2Trans.shpConversion

Conversion functions to read/ write Shape files

Functions

SHP2Array	Read shapefile and convert it to a python dictionary
extractFeature	Extract feature nFeature2Extract from featureIn
getSHPProjection	Fetch projection from shp file
readLine	Read line from .shp Use SHP2Array to read the shape file.
readPoints	Read points from .shp Use SHP2Array to read the shape file.
readThickness	Read shapefile and fetch info on features' ids and thickness values
removeFeature	Remove feature number nFeature2Remove from featureIn
writeLine2SHPfile	write a line to shapefile
writePoint2SHPfile	write a point to shapefile

in2Trans.shpConversion.SHP2Array

SHP2Array(*infile*, *defname=None*)

Read shapefile and convert it to a python dictionary

The dictionary contains the name of the paths in the shape file, the np array with the coordinates of the feature points (all stacked in the same array) and information about the starting index and length of each feature

Parameters

- **infile** (*str*) – path to shape file
- **defname** (*str*) – name to give to the feature in the shape file

Returns

SHPdata –

sks :
projection information

Name
[str] list of feature names

x
[1D numpy array] np array of the x coord of the points in the features

y
[1D numpy array] np array of the y coord of the points in the features

z
[1D numpy array] np array of zeros and same size as the x an y coordinates array

Start
[1D numpy array] np array with the starting index of each feature in the coordinates arrays (as many indexes as features)

Length
[1D numpy array] np array with the length of each feature in the coordinates arrays (as many indexes as features)

thickness (optional)
[1D numpy array] np array with the (release or entrainment) thickness of each feature (as many values as features)

id
[list] list of oid as string for each feature

ci95: list
list of 95% confidence interval of thickness value

nParts: list
list of parts of polygon (added the total number of points as list item, so if multiple parts len>2)

nFeatures: int
number of features per line (parts)

Return type
dict

in2Trans.shpConversion.extractFeature

`extractFeature`(*featureIn*, *nFeature2Extract*)

Extract feature nFeature2Extract from featureIn

Parameters

- **featureIn** (*dict*) – shape file dicionary (structure produced by SHP2Array, readLine or readPoint)
- **nFeature2Extract** (*int*) – index of feature to extract from featureIn

Returns
featureOut – shape file dicionary with feature nFeature2Extract

Return type
dict

in2Trans.shpConversion.getSHPProjection

`getSHPProjection`(*infile*)

Fetch projection from shp file

Parameters
infile (*str*) – path to shape file

Returns
sks – projection string (if available, None if not)

Return type
str

in2Trans.shpConversion.readLine

`readLine`(*fname*, *defname*, *dem*)

Read line from .shp Use SHP2Array to read the shape file. Check if the lines are laying inside the dem extend

Parameters

- **fname** (*str*) – path to shape file
- **defname** (*str*) – name to give to the line in the shape file
- **dem** (*dict*) – dem dictionary

Returns

Line – Line['Name'] : list of lines names Line['Coord'] : np array of the coords of points in lines Line['Start'] : list of starting index of each line in Coord Line['Length'] : list of length of each line in Coord

Return type

dict

in2Trans.shpConversion.readPoints

`readPoints`(*fname*, *dem*)

Read points from .shp Use SHP2Array to read the shape file. Check if the points are laying inside the dem extend

Parameters

- **fname** (*str*) – path to shape file
- **defname** (*str*) – name to give to the points in the shape file
- **dem** (*dict*) – dem dictionary

Returns

Line – Line['Name'] : list of points names Line['Coord'] : np array of the coords of points in points Line['Start'] : list of starting index of each point in Coord Line['Length'] : list of length of each point in Coord

Return type

dict

in2Trans.shpConversion.readThickness

`readThickness`(*infile*, *defname=None*)

Read shapefile and fetch info on features' ids and thickness values

Parameters

- **infile** (*str*) – path to shape file
- **defname** (*str*) – name to give to the feature in the shape file

Returns

- **thickness** (*list*) – list of strings with the (release or entrainment) thickness of each feature (as many values as features)
- **id** (*list*) – list of strings for oid of each feature in shp file
- **ci95** (*list*) – list of all ci95 values if provided

in2Trans.shpConversion.removeFeature

`removeFeature`(*featureIn*, *nFeature2Remove*)

Remove feature number nFeature2Remove from featureIn

Parameters

- **featureIn** (*dict*) – shape file dicionary (structure produced by SHP2Array, readLine or readPoint)
- **nFeature2Remove** (*int*) – index of feature to remove from featureIn

Returns
featureOut – shape file dicionary without feature nFeature2Remove

Return type
dict

in2Trans.shpConversion.writeLine2SHPfile

`writeLine2SHPfile`(*lineDict*, *lineName*, *fileName*, *header=''*)

write a line to shapefile

Parameters

- **lineDict** (*dict*) – line dict
- **lineName** (*str*) – line name
- **fileName** (*str or pathlib path*) – path where the line will be saved line name
- **header** (*dict*) – optional argument (’’ by default). If provided, header dictionary with ‘xllcenter’ and ‘yllcenter’ to add to the line

Returns
fileName – path where the line has been saved

Return type
str

in2Trans.shpConversion.writePoint2SHPfile

`writePoint2SHPfile`(*pointDict*, *pointName*, *fileName*)

write a point to shapefile

Parameters

- **pointDict** (*dict*) – point dict
- **pointName** (*str*) – point name
- **fileName** (*str or pathlib path*) – path where the point will be saved

Returns
fileName – path where the point has been saved

Return type
str

2.10.2.3 in3Utils

Tools for geo transformations

Modules

in3Utils.cfgHandling	Utilities for working with cfg info
in3Utils.cfgUtils	Utilities for handling configuration files
in3Utils.fileHandlerUtils	Directory and file handling helper functions
in3Utils.generateTopo	Create generic/idealised topographies
in3Utils.geoTrans	Opperations and transformations of rasters and lines
in3Utils.getReleaseArea	Get release area corner coordinates for a rectangle of an area of approx.
in3Utils.initialiseDirs	Setup required directory structures by computational modules
in3Utils.initializeProject	Functions to initialize project, i.e create folder structure
in3Utils.logUtils	Defining a writable object to write the config to the log file

in3Utils.cfgHandling

Utilities for working with cfg info

Functions

addInfoToSimName	Add parameterName and value to simNames of simulation dataframe
applyCfgOverride	override configuration parameter values with the values provided in cfgWithOverrideParameters[modName_override] if addModValues True update the cfgWithOverrideParameters with the values for all parameters that are not provided in the override parameters
errorDuplicateListEntry	check if duplicate entries appear in a list and raise Assertion error using message
fetchAndOrderSimFiles	Filter simulations results using a list of parameters and a flag if in ascending or descending order
fetchValidationString	create a validation string to be checked if simDFrow matches filtering criteria (given by val)
filterCom1DFAThicknessValues	thickness settings different if read from shpfile - requires more complex filtering if read from shp - thickness values are provided per feature!! for example relTh = '' but relTh0 = 1 is appended for feature with id 0, relTh1 for feature with id 1, etc.
filterSims	Filter simulations using a list of parameters and a pandas dataFrame of simulation configurations if ~ is used as a prefix for a parameter - it is filtered according to values that do NOT match the value provided with the ~Parameter
insertIntoSimName	Add keys and values to name, in between parts of name split by index
orderSimFiles	Filter simulations results using a list of parameters and a flag if in ascending or descending order
orderSimulations	Order simulations datadframe using a list of parameters and a flag if in ascending or descending order
removeSimsNotMatching	remove simulations from simDF that do not match filtering critera

in3Utils.cfgHandling.addInfoToSimName

addInfoToSimName(*avalancheDir*, *csvString=''*)

Add parameterName and value to simNames of simulation dataframe

E.g used as helper routine for renaming layernames in qgis

Parameters

- **avalancheDir** (*str*) – path to avalanche directory
- **csvString** – comma separated list with parameter names, as found in com1DFA ini file eg. 'mu,tau0,tEnd'

Returns

simDF – containing index, the parameters and the old and new name

Return type

dataframe

in3Utils.cfgHandling.applyCfgOverride

applyCfgOverride(*cfgToOverride*, *cfgWithOverrideParameters*, *module*, *addModValues=False*)

override configuration parameter values with the values provided in cfgWithOverrideParameters[modName_override] if addModValues True update the cfgWithOverrideParameters with the values for all parameters that are not provided in the override parameters

Parameters

- **cfgToOverride** (*configparer object*) – configuration of module of interest
- **cfgWithOverrideParameters** (*configparser object*) – full configuration settings containing a section modName_override with parameter values that should be overriden in the cfgToOverride
- **module** – module of the cfgToOverride configuration
- **addModValues** (*bool*) – if True add all parameters from cfgToOverride module to cfgWithOverrideParameters override section

Returns

- **cfgToOverride** (*configparser object*) – updated configuration of module
- **cfgWithOverrideParameters** (*configparser object*) – updated configuration of module

in3Utils.cfgHandling.errorDuplicateListEntry

errorDuplicateListEntry(*listKeys*, *message*)

check if duplicate entries appear in a list and raise Assertion error using message

Parameters

- **listKeys** (*list*) – list with keys
- **message** (*str*) – message of error

in3Utils.cfgHandling.fetchAndOrderSimFiles

fetchAndOrderSimFiles(*avalancheDir*, *inputDir*, *varParList*, *ascendingOrder*, *specDir=''*, *resFiles=False*)

Filter simulations results using a list of parameters and a flag if in ascending or descending order

Parameters

- **avalancheDir** (*str*) – path to avalanche directory
- **inputDir** (*str*) – path to simulation results
- **varParList** (*str or list*) – simulation configuration parameters for ordering simulations
- **ascendingOrder** (*bool*) – True if simulations shall be ordered in ascending order regarding varPar
- **specDir** (*str*) – path to a directory where simulation configuration files can be found - optional

Returns

dataDF – dataFrame of simulation results (fileName, ... and values for parameters in varParList)

Return type

pandas dataFrame

in3Utils.cfgHandling.fetchValidationString

`fetchValidationString`(*val*, *thIdList*, *thNames*, *simDFrow*)

create a validation string to be checked if simDFrow matches filtering criteria (given by val)

Parameters

- **val** (*str, float*) – value to be checked
- **thIdList** (*list*) – list with thickness feature ids
- **thNames** (*list*) – list with thickness feature names
- **simDFrow** (*pandas dataframe row*) – parameters of simulation

Returns
validationString – bool if simulation given by simDFrow matches filtering criteria

Return type
bool

in3Utils.cfgHandling.filterCom1DFAThicknessValues

`filterCom1DFAThicknessValues`(*key*, *value*, *simDF*)

thickness settings different if read from shpfile - requires more complex filtering if read from shp - thickness values are provided per feature!! for example relTh = '' but relTh0 = 1 is appended for feature with id 0, relTh1 for feature with id 1, etc.

Parameters

- **key** (*str*) – name of parameter
- **value** (*list*) – list of values used for filtering
- **simDF** (*pandas dataframe*) – configuration info for each simulation

Returns
simDF – updated dataframe

Return type
pandas data frame

in3Utils.cfgHandling.filterSims

`filterSims`(*avalancheDir*, *parametersDict*, *specDir=''*, *simDF=''*)

Filter simulations using a list of parameters and a pandas dataFrame of simulation configurations if ~ is used as a prefix for a parameter - it is filtered according to values that do NOT match the value provided with the ~Parameter

Parameters

- **avalancheDir** (*str*) – path to avalanche directory
- **parametersDict** (*dict*) – dictionary with parameter and parameter values for filtering
- **specDir** (*str*) – path to a directory where simulation configuration files can be found - optional
- **simDF** (*pandas DataFrame*) – optional - if simDF already available

Returns
simNameList – list of simNames that match filtering criteria

Return type
list

in3Utils.cfgHandling.insertIntoSimName

`insertIntoSimName`(*name*, *keys*, *values*, *index*)

Add keys and values to name, in between parts of name split by index

Parameters

- **name** (*str*) – name to extend
- **keys** (*list*) – list with keys
- **values** (*list*) – list with values
- **index** (*str*) – used to split name

Returns
newName – containing newName, with keys and values inserted after index

Return type
string

in3Utils.cfgHandling.orderSimFiles

`orderSimFiles`(*avalancheDir*, *inputDir*, *varParList*, *ascendingOrder*, *specDir=''*, *resFiles=False*)

Filter simulations results using a list of parameters and a flag if in ascending or descending order

Parameters

- **avalancheDir** (*str*) – path to avalanche directory
- **inputDir** (*str*) – path to simulation results
- **varParList** (*str or list*) – simulation configuration parameters for ordering simulations
- **ascendingOrder** (*bool*) – True if simulations shall be ordered in ascending order regarding varPar
- **specDir** (*str*) – path to a directory where simulation configuration files can be found - optional

Returns
dataDF – dataFrame of simulation results (fileName, . . . and values for parameters in varParList)

Return type
pandas dataFrame

in3Utils.cfgHandling.orderSimulations

`orderSimulations`(*varParList*, *ascendingOrder*, *simDF*)

Order simulations datadframe using a list of parameters and a flag if in ascending or descending order

Parameters

- **varParList** (*str or list*) – simulation configuration parameters for ordering simulations
- **ascendingOrder** (*bool*) – True if simulations shall be ordered in ascending order regarding varPar
- **simDF** (*pandas dataFrame*) – dataFrame of simulations (one line per simultaion with fileName, ... and values for parameters in varParList)

Returns

simDF – sorted dataFrame of simulation results (fileName, ... and values for parameters in varParList)

Return type

pandas dataFrame

in3Utils.cfgHandling.removeSimsNotMatching

`removeSimsNotMatching`(*simDF*, *key*, *value*)

remove simulations from simDF that do not match filtering critera

Parameters

- **simDF** (*pandas dataframe*) – dataframe with one row per simulation and info on its characteristics, parameters used,..
- **key** (*str*) – name of parameter that shall be used for filtering
- **value** (*list*) – list of parameter values used for filtering

Returns

simDF – updated dataframe with only those simulations that match filtering criteria

Return type

pandas dataframe

in3Utils.cfgUtils

Utilities for handling configuration files

Functions

appendCgf2DF	append simulation configuration to the simulation dataframe only account for sections GENERAL and INPUT
appendTcpu2DF	append Tcpu dictionary to the dataframe
cfgHash	UID hash of a config.
compareTwoConfigs	compare locCfg to defCfg and return a cfg object and modification dict Values are merged from locCfg to defCfg: - parameters already in defCfg get the value from locCfg - additional values in locCfg get added in the resulting Cfg
convertConfigParserToDict	create dictionary from configparser object
convertDF2numerics	convert a string DF to a numerical one
convertDictToConfigParser	create configParser object from dict
convertToCfgList	convert a list into a string where inidividual list items are separated by |
createConfigurationInfo	Read configurations from all simulations configuration ini files from directory
getDefaultModuleConfig	Returns the default configuration for a given module returns a configParser object
getGeneralConfig	Returns the general configuration for avaframe returns a configParser object
getModuleConfig	Returns the configuration for a given module returns a configParser object
getNumberOfProcesses	Determine how many CPU cores to take for parallel tasks
readAllConfigurationInfo	Read allConfigurations.csv file as dataFrame from directory
readCfgFile	Read configuration from ini file, if module is provided, module configuration is read from Ouputs, if fileName is provided configuration is read from fileName
readCompareConfig	Read and optionally compare configuration files (if a local and default are both provided) and inform user of the eventual differences.
setStrnanToNan	set pandas element to np.nan if it is a string nan
writeAllConfigurationInfo	Write cfg configuration to allConfigurations.csv
writeCfgFile	Save configuration used to text file in Outputs/moduleName/configurationFiles/modName.ini or optional to filePath and with fileName
writeDictToJson	write a dictionary to a json file

in3Utils.cfgUtils.appendCgf2DF

appendCgf2DF(*simHash*, *simName*, *cfgObject*, *simDF*)

append simulation configuration to the simulation dataframe only account for sections GENERAL and INPUT

Parameters

- **simHash** (*str*) – hash of the simulation to append
- **simName** (*str*) – name of the simulation
- **cfgObject** (*configParser*) – configuration coresponding to the simulation

- **simDF** (*pandas dataFrame*) – configuration dataframe

Returns
simDF – DFappended with the new simulation configuration

Return type
pandas DataFrame

in3Utils.cfgUtils.appendTcpu2DF

`appendTcpu2DF`(*simHash*, *tCPU*, *tCPUDF*)

append Tcpu dictionary to the dataframe

Parameters

- **simHash** (*str*) – hash of the simulation corresponding to the tCPU dict to append
- **tCPU** (*dict*) – cpu time dict of the simulation
- **tCPUDF** (*pandas dataFrame*) – tCPU dataframe

Returns
simDF – DFappended with the new simulation configuration

Return type
pandas DataFrame

in3Utils.cfgUtils.cfgHash

`cfgHash`(*cfg*, *typeDict=False*)

UID hash of a config. Given a configParser object cfg, or a dictionary - then typeDict=True, returns a uid hash

Parameters

- **cfg** (*configParser object*)
- **typeDict** (*dict*) – dictionary
- **Returns**
- **———**
- **uid** (*str*) – uid hash

in3Utils.cfgUtils.compareTwoConfigs

`compareTwoConfigs`(*defCfg*, *locCfg*, *toPrint=False*)

compare locCfg to defCfg and return a cfg object and modification dict Values are merged from locCfg to defCfg: - parameters already in defCfg get the value from locCfg - additional values in locCfg get added in the resulting Cfg

Parameters

- **defCfg** (*configparser object*) – default configuration
- **locCfg** (*configuration object*) – configuration that is compared to defCfg
- **toPrint** (*bool*) – flag if config shall be printed to log

Returns

- **modInfo** (*dict*) – dictionary containing only differences from default
- **cfg** (*configParser object*) – contains combined config

in3Utils.cfgUtils.convertConfigParserToDict

`convertConfigParserToDict`(*cfg*)

create dictionary from configparser object

in3Utils.cfgUtils.convertDF2numerics

`convertDF2numerics`(*simDF*)

convert a string DF to a numerical one

Parameters
simDF (*pandas dataFrame*) – dataframe

Returns
simDF

Return type
pandas DataFrame

in3Utils.cfgUtils.convertDictToConfigParser

`convertDictToConfigParser`(*cfgDict*)

create configParser object from dict

in3Utils.cfgUtils.convertToCfgList

`convertToCfgList`(*parameterList*)

convert a list into a string where inidividual list items are separated by |

Parameters
parameterList (*list*) – list of parameter values

Returns
parameterString – str with parameter values separated by |

Return type
str

in3Utils.cfgUtils.createConfigurationInfo

`createConfigurationInfo`(*avaDir*, *comModule='com1DFA'*, *standardCfg=''*, *writeCSV=False*, *specDir=''*)

Read configurations from all simulations configuration ini files from directory

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **standardCfg** (*dict*) – standard configuration for module - option

- **writeCSV** (*bool*) – True if configuration dataFrame shall be written to csv file
- **specDir** (*str*) – path to a directory where simulation configuration files can be found - optional

Returns
simDF – DF with all the simulation configurations

Return type
pandas DataFrame

in3Utils.cfgUtils.getDefaultModuleConfig

`getDefaultModuleConfig`(*module*, *toPrint=True*)

Returns the default configuration for a given module returns a configParser object

module object: module
[the calling function provides the already imported] module eg.: from avaframe.com2AB import com2AB leads to getModuleConfig(com2AB) whereas from avaframe.com2AB import com2AB as c2 leads to getModuleConfig(c2)

in3Utils.cfgUtils.getGeneralConfig

`getGeneralConfig`(*nameFile=''*)

Returns the general configuration for avaframe returns a configParser object

Parameters
nameFile (*pathlib path*) – optional full path to file, if empty use avaframeCfg from folder one level up

in3Utils.cfgUtils.getModuleConfig

`getModuleConfig`(*module*, *fileOverride=''*, *modInfo=False*, *toPrint=True*, *onlyDefault=False*)

Returns the configuration for a given module returns a configParser object

module object: module
[the calling function provides the already imported] module eg.: from avaframe.com2AB import com2AB leads to getModuleConfig(com2AB) whereas from avaframe.com2AB import com2AB as c2 leads to getModuleConfig(c2)

Str: fileOverride
[allows for a completely different file location. However note:] missing values from the default cfg will always be added!

modInfo: bool
true if dictionary with info on differences to standard config

onlyDefault: bool
if True, only use the default configuration

Order is as follows: fileOverride -> local_MODULECfg.ini -> MODULECfg.ini

in3Utils.cfgUtils.getNumberOfProcesses

`getNumberOfProcesses`(*cfgMain*, *nSims*)

Determine how many CPU cores to take for parallel tasks

Parameters

- **cfgMain** (*configuration object*) – the main avaframe configuration
- **nSims** (*integer*) – number of simulations that need to be calculated

Returns

nCPU – number of cores to take

Return type

int

in3Utils.cfgUtils.readAllConfigurationInfo

`readAllConfigurationInfo`(*avaDir*, *specDir=''*, *configCsvName='allConfigurations'*)

Read allConfigurations.csv file as dataFrame from directory

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **specDir** (*str*) – path to a directory where simulation configuration files can be found - optional
- **configCsvName** (*str*) – name of configuration csv file

Returns

- **simDF** (*pandas DataFrame*) – DF with all the simulation configurations
- **simDFName** (*array*) – simName column of the dataframe

in3Utils.cfgUtils.readCfgFile

`readCfgFile`(*avaDir*, *module=''*, *fileName=''*)

Read configuration from ini file, if module is provided, module configuration is read from Ouputs, if fileName is provided configuration is read from fileName

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **module** – module
- **fileName** (*str*) – path to file that should be read - optional

Returns

cfg – configuration that is from file

Return type

configParser object

in3Utils.cfgUtils.readCompareConfig

`readCompareConfig`(*iniFile*, *modName*, *compare*, *toPrint=True*)

Read and optionally compare configuration files (if a local and default are both provided) and inform user of the eventual differences. Take the default as reference.

Parameters

- **iniFile** (*path to config file*) – Only one path if compare=False
- **compare** (*boolean*) – True if two paths are provided and a comparison is needed
- **toPrint** (*boolean*) – True (default) to print configuration to terminal. Differences to default will ALWAYS be printed

Returns

- **Output** (*ConfigParser object*) – contains combined config
- **modDict** (*dict*) – dictionary containing only differences from default

in3Utils.cfgUtils.setStrnanToNan

`setStrnanToNan`(*simDF*, *simDFTest*, *name*)

set pandas element to np.nan if it is a string nan

Parameters

- **simDF** (*pandas dataFrame*) – dataframe
- **simDFTest** (*pandas series*) – series of sim DF column named name replaced "." with " "
- **name** (*str*) – name of pandas dataframe column

Returns
simDF – updated pandas dataframe with np.nan values where string nan was

Return type
pandas dataframe

in3Utils.cfgUtils.writeAllConfigurationInfo

`writeAllConfigurationInfo`(*avaDir*, *simDF*, *specDir=''*, *csvName='allConfigurations.csv'*)

Write cfg configuration to allConfigurations.csv

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **simDF** (*pandas dataFrame*) – daaframe of the configuration
- **specDir** (*str*) – path to a directory where simulation configuration shal be saved - optional
- **csvName** (*str*) – name of csv file in which to save to - optional

Returns
configFiles – path where the configuration dataframe was saved

Return type
pathlib Path

in3Utils.cfgUtils.writeCfgFile

`writeCfgFile`(*avaDir*, *module*, *cfg*, *fileName=''*, *filePath=''*)

Save configuration used to text file in Outputs/moduleName/configurationFiles/modName.ini or optional to filePath and with fileName

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **module** – module
- **cfg** (*configparser object*) – configuration settings
- **fileName** (*str*) – name of saved configuration file - optional
- **filePath** (*str or pathlib path*) – path where file should be saved to except file name - optional

in3Utils.cfgUtils.writeDictToJson

`writeDictToJson`(*inDict*, *outFilePath*)

write a dictionary to a json file

in3Utils.fileHandlerUtils

Directory and file handling helper functions

Functions

checkCommonSims	Check which files are common between local and full ExpLog
checkIfFileExists	test if file exists if not throw error
checkPathlib	check if pathlib.PurePath if not convert to
exportcom1DFAOrigOutput	Export the simulation results from com1DFA output to desired location
extractLogInfo	read log file and extract info on time, mass stop criterion
extractParameterInfo	Extract info about simulation parameters from the log file
fetchFlowFields	fetch paths to all desired flow fields within folder
fileNotFoundMessage	throw error if file not found with message and path
getFilterDict	Create parametersDict from ini file, for filtering simulations
makeADir	Check if a directory exists, if not create directory
makeSimDF	Create a dataFrame that contains all info on simulations
makeSimFromResDF	Create a dataFrame that contains all info on simulations in output/comModule/peakFiles
readLogFile	Read experiment log file and make dictionary that contains general info on all simulations
splitIniValueToArraySteps	read values in ini file and return numpy array or list if the items are strings; values can either be separated by | or provided in start:end:numberOfSteps format if separated by : or $ also optional add one additional value using & if format of refVal$percent$steps is used - an array is created with +- percent of refVal in nsteps
splitTimeValueToArrayInterval	read save time step info from ini file and return numpy array of values

in3Utils.fileHandlerUtils.checkCommonSims

checkCommonSims(*logName*, *localLogName*)

Check which files are common between local and full ExpLog

in3Utils.fileHandlerUtils.checkIfFileExists

checkIfFileExists(*filePath*, *fileType=''*)

test if file exists if not throw error

Parameters

- **filePath** (*pathlib path*) – path to file
- **fileType** (*str*) – string for error message which kind of file is not found

in3Utils.fileHandlerUtils.checkPathlib

checkPathlib(*checkPath*)

check if pathlib.PurePath if not convert to

Parameters

- **checkPath** (*str or pathlib path*)
- **path to be checked**

Returns
checkPath – pathlib path version of checkPath

Return type
pathlib path

in3Utils.fileHandlerUtils.exportcom1DFAOrigOutput

exportcom1DFAOrigOutput(*avaDir*, *cfg=''*, *addTSteps=False*)

Export the simulation results from com1DFA output to desired location

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **cfg** (*dict*) – configuration read from ini file that has been used for the com1DFAOrig simulation
- **addTSteps** (*bool*) – if True: first and last time step of flow thickness are exported

in3Utils.fileHandlerUtils.extractLogInfo

extractLogInfo(*fileName*)

read log file and extract info on time, mass stop criterion

Parameters
fileName (*str or pathlib path*) – path to log file

Returns
logDict – dictionary with arrays for mass entrained mass time step and info on simulation run and stop criterion

Return type
dict

in3Utils.fileHandlerUtils.extractParameterInfo

extractParameterInfo(*avaDir*, *simName*, *reportD*)

Extract info about simulation parameters from the log file

Parameters

- **avaDir** (*str*) – path to avalanche
- **simName** (*str*) – name of the simulation
- **reportD** (*dict*) – report dictionary

Returns

- **parameterDict** (*dict*) – dictionary listing name of parameter and value; release mass, final time step and current mast
- **reportD** (*dict*) – upated report dictionary with info on simulation

in3Utils.fileHandlerUtils.fetchFlowFields

`fetchFlowFields`(*flowFieldsDir*, *suffix=''*)

fetch paths to all desired flow fields within folder

Parameters

- **flowFieldsDir** (*str or pathlib path*) – path to flow field ascii files
- **suffix** (*str*) – suffix in flow field name to be searched for

Returns
flowFields – list of pathlib paths to flow fields

Return type
list

in3Utils.fileHandlerUtils.fileNotFoundMessage

`fileNotFoundMessage`(*messageName*)

throw error if file not found with message and path

Parameters
messageName (*str*) – error message

in3Utils.fileHandlerUtils.getFilterDict

`getFilterDict`(*cfg*, *section*)

Create parametersDict from ini file, for filtering simulations

Parameters

- **cfg** (*configParser object*) – configuration with information on filtering criteria
- **section** (*str*) – section of cfg where filtering criteria can be found

Returns
parametersDict – dictionary with parameter and parameter values for filtering simulation results

Return type
dict

in3Utils.fileHandlerUtils.makeADir

makeADir(*dirName*)

Check if a directory exists, if not create directory

Parameters
dirName (*str*) – path of directory that should be made

in3Utils.fileHandlerUtils.makeSimDF

makeSimDF(*inputDir*, *avaDir=''*, *simID='simID'*)

Create a dataFrame that contains all info on simulations

this can then be used to filter simulations for example

Parameters

- **inputDir** (*str*) – path to directory of simulation results
- **avaDir** (*str*) – optional - path to avalanche directory
- **simID** (*str*) – optional - simulation identification, depending on the computational module: com1DFA: simHash com1DFAOrig: Mu or parameter that has been used in parameter variation

Returns
dataDF – dataframe with full file path, file name, release area scenario, simulation type (null, entres, etc.), model type (dfa, ref, etc.), simID, result type (ppr, pft, etc.), simulation name, cell size and optional name of avalanche, optional time step

Return type
dataFrame

in3Utils.fileHandlerUtils.makeSimFromResDF

makeSimFromResDF(*avaDir*, *comModule*, *inputDir=''*, *simName=''*)

Create a dataFrame that contains all info on simulations in output/comModule/peakFiles

One line for each simulation - so all peakfiles that belong to one simulation are listed in one line that corresponds to that simulation

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **comModule** (*str*) – module used to create the results
- **inputDir** (*str*) – optional - path to directory of simulation results
- **simName** (*str*) – optional - key phrase to be found in the simulation result name

Returns

- **dataDF** (*dataFrame*) – dataframe with for each simulation, the full file path, file name, release area scenario, simulation type (null, entres, etc.), model type (dfa, ref, etc.), simID, path to result files (ppr, pft, etc.), simulation name, cell size and optional name of avalanche, optional time step
- **resTypeListAll** (*list*) – list of res types available for all simulations

in3Utils.fileHandlerUtils.readLogFile

readLogFile(*logName*, *cfg=''*)

Read experiment log file and make dictionary that contains general info on all simulations

Parameters

- **logName** (*str*) – path to log file
- **cfg** (*dict*) – optional - configuration read from com1DFA simulation

Returns

logDict – dictionary with number of simulation (noSim), name of simulation (simName), parameter variation, full name

Return type

dict

in3Utils.fileHandlerUtils.splitIniValueToArraySteps

splitIniValueToArraySteps(*cfgValues*, *returnList=False*)

read values in ini file and return numpy array or list if the items are strings; values can either be separated by | or provided in start:end:numberOfSteps format if separated by : or $ also optional add one additional value using & if format of refVal$percent$steps is used - an array is created with +- percent of refVal in nsteps

Parameters

- **cfgValues** (*str*) – values of parameter to be read from ini file
- **returnList** (*bool*) – if True force to return values as list

Returns

items – values as 1D numpy array or list (in the case of strings)

Return type

1D numpy array or list

in3Utils.fileHandlerUtils.splitTimeValueToArrayInterval

splitTimeValueToArrayInterval(*cfgValues*, *endTime*)

read save time step info from ini file and return numpy array of values

values can either be separated by | or provided in start:interval format

Parameters

- **cfgValues** (*str*) – time steps info
- **endTime** (*float*) – end time

Returns

items – time step values as 1D numpy array

Return type

1D numpy array

in3Utils.generateTopo

Create generic/idealised topographies

Functions

addDrop	Add a drop to a given topography
bowl	Compute coordinates of sphere with given radius (rBowl)
computeCoordGrid	
createParabolaAxis	create the s coordinate for a lined sloped from theta - phi from x axis
flatplane	Compute coordinates of flat plane topography
generateTopo	Compute coordinates of desired topography with given inputs
getGridDefs	
getParabolaParams	Compute parameters for parabola
helix	Compute coordinates of helix-shaped topography with given radius (rHelix)
hockey	Compute coordinates of an inclined plane with a flat foreland defined by total fall height z0, angle to flat foreland (meanAlpha) and a radius (rCirc) to smooth the transition from inclined plane to flat foreland
inclinedplane	Compute coordinates of inclined plane with given slope (meanAlpha)
parabola	Compute coordinates of a parabolically-shaped slope with a flat foreland defined by total fall height C, angle (meanAlpha) or distance (fLen) to flat foreland
parabolaRotation	Compute coordinates of a parabolically-shaped slope with a flat foreland defined by total fall height C, angle (meanAlpha) or distance (fLen) to flat foreland One parabolic slope in x direction, one sloped with 45° and one with 60°
pyramid	Generate a pyramid topography - in this case rectangular domain
writeDEM	Write topography information to file

in3Utils.generateTopo.addDrop

addDrop(*cfg*, *x*, *y*, *zv*)

Add a drop to a given topography

The drop is added in the x drection

Parameters

- **cfg** (*configparser*) – configuration for generateTopo
- **x** (*2D numpy array*) – x coordinate of the raster
- **y** (*2D numpy array*) – y coordinate of the raster

- **zv** (*2D numpy array*) – z coordinate of the raster

Returns
zv – z coordinate of the raster taking the drop into account

Return type
2D numpy array

in3Utils.generateTopo.bowl

bowl(*cfg*)
Compute coordinates of sphere with given radius (rBwol)

in3Utils.generateTopo.computeCoordGrid

computeCoordGrid(*dx*, *xEnd*, *yEnd*)

in3Utils.generateTopo.createParabolaAxis

createParabolaAxis(*phi*, *theta*, *r*, *zv*, *fLen*, *fFlat*)
create the s coordinate for a lined sloped from theta - phi from x axis

in3Utils.generateTopo.flatplane

flatplane(*cfg*)
Compute coordinates of flat plane topography

in3Utils.generateTopo.generateTopo

generateTopo(*cfg*, *avalancheDir*)
Compute coordinates of desired topography with given inputs

in3Utils.generateTopo.getGridDefs

getGridDefs(*cfg*)

in3Utils.generateTopo.getParabolaParams

getParabolaParams(*cfg*)
Compute parameters for parabola

in3Utils.generateTopo.helix

helix(*cfg*)

Compute coordinates of helix-shaped topography with given radius (rHelix)

in3Utils.generateTopo.hockey

hockey(*cfg*)

Compute coordinates of an inclined plane with a flat foreland defined by total fall height z0, angle to flat foreland (meanAlpha) and a radius (rCirc) to smooth the transition from inclined plane to flat foreland

in3Utils.generateTopo.inclinedplane

inclinedplane(*cfg*)

Compute coordinates of inclined plane with given slope (meanAlpha)

in3Utils.generateTopo.parabola

parabola(*cfg*)

Compute coordinates of a parabolically-shaped slope with a flat foreland defined by total fall height C, angle (meanAlpha) or distance (fLen) to flat foreland

in3Utils.generateTopo.parabolaRotation

parabolaRotation(*cfg*)

Compute coordinates of a parabolically-shaped slope with a flat foreland defined by total fall height C, angle (meanAlpha) or distance (fLen) to flat foreland One parabolic slope in x direction, one sloped with 45° and one with 60°

in3Utils.generateTopo.pyramid

pyramid(*cfg*)

Generate a pyramid topography - in this case rectangular domain

in3Utils.generateTopo.writeDEM

writeDEM(*cfg*, *z*, *outDir*)

Write topography information to file

in3Utils.geoTrans

Opperations and transformations of rasters and lines

Functions

areaPoly	Gauss's area formula to calculate polygon area
cartToSpherical	convert from cartesian to spherical coordinates
checkOverlap	Check if two rasters overlap
checkParticlesInRelease	remove particles laying outside the polygon
checkProfile	check that the avalanche profiles goes from top to bottom flip it if not and adjust the splitpoint in consequence
computeAlongLineDistance	compute distance along a dict of coordinates or a shapely lineString incrementally
computeLengthOfLine2D	compute distance along a line in 2D
computeS	compute s coordinate given a path (x, y)
findAngleProfile	Find the beta point: first point under the beta value given in prepareAngleProfile.
findClosestPoint	find the closest point of pointDict along line defined by xcoor and ycoor - only xy plane!
findPointOnDEM	find point on dem given a direction and a z value to reach
findSplitPoint	Finds the closest point in Points to the avaProfile and returns its projection on avaProfile.
getCellsAlongLine	Find all raster cells crossed by the line line has to be entierly contained on the raster extend.
getNeighborCells	Find the neighbour cells to a given cell
isCounterClockWise	Determines if a polygon path is mostly clockwise or counter clockwise
makeCoordGridFromHeader	Get x and y (2D) grid description vectors for a mesh with a given number of rows and columns, lower left center and cellSize.
makeCoordinateGrid	Create grid
path2domain	Creates a domain (irregular raster) along a path, given the path xyPath, a domain width and a raster cellsize
pointInPolygon	find particles within a polygon
polygon2Raster	convert line to raster
prepareAngleProfile	Prepare inputs for findAngleProfile function Read profile (s, z), compute the slope Angle look for points for which the slope is under the given Beta value and that are located downstream of the splitPoint
prepareArea	convert shape file polygon to raster
prepareLine	Resample and project line on dem 1- Resample the avapath line with an interval of approximately distance in meters between points (projected distance on the horizontal plane).
projectOnGrid	Projects Z onto points (x,y) using a bilinear or nearest interpolation and returns the z coord
projectOnRaster	Projects Points on raster using a bilinear or nearest interpolation and returns the z coord (no for loop)

continues on next page

Table 2.1 – continued from previous page

remeshDEM	change DEM cell size by reprojecting on a new grid - first check if remeshed DEM available
remeshData	compute raster data on a new mesh with cellSize using the specified remeshOption.
resizeData	Reproject raster on a grid of shape rasterRef
rotate	rotate a vector provided as start and end point with theta angle rotation counter-clockwise
rotateRaster	rotate clockwise a raster arround (0, 0) with theta angle
searchRemeshedDEM	search if remeshed DEM with correct name and cell size already available
snapPtsToLine	snap points to line in dataframe only considering x, y plane!

in3Utils.geoTrans.areaPoly

areaPoly(*X*, *Y*)

Gauss's area formula to calculate polygon area

Parameters

- **X** (*1D numpy array*) – x coord of the vertices
- **Y** (*1D numpy array*) – y coord of the vertices
- **(Without repeating the first vertex!!!)**

Returns

area – Area of the polygon

Return type

float

in3Utils.geoTrans.cartToSpherical

cartToSpherical(*X*, *Y*, *Z*)

convert from cartesian to spherical coordinates

Parameters

- **X** (*float*) – x coordinate
- **Y** (*float*) – y coordinate
- **Z** (*float*) – z coordinate

Returns

- **r** (*float*) – radius
- **phi** (*float*) – azimuth angle [degrees]
- **theta** (*float*) – for elevation angle defined from Z-axis down [degrees]

in3Utils.geoTrans.checkOverlap

checkOverlap(*toCheckRaster*, *refRaster*, *nameToCheck*, *nameRef*, *crop=False*)

Check if two rasters overlap

Parameters

- **toCheckRaster** (*2D numpy array*) – Raster to check
- **refRaster** (*2D numpy array*) – reference Raster
- **nameToCheck** (*str*) – name of raster that might overlap
- **nameRef** (*str*) – name of reference raster
- **crop** (*boolean*) – if True, remove overlaping part and send a warning

Returns

toCheckRaster – if crop is True, return toCheckRaster without the overlaping part and send a warning if needed if crop is False, return error if Rasters overlap otherwise return toCheckRaster

Return type

2D numpy array

in3Utils.geoTrans.checkParticlesInRelease

checkParticlesInRelease(*particles*, *line*, *radius*)

remove particles laying outside the polygon

Parameters

- **particles** (*dict*) – particles dictionary
- **line** (*dict*) – line dictionary
- **radius** (*float*) – threshold val that decides if a point is in the polygon, on the line or very close but outside

Returns

particles – particles dictionary where particles outside of the polygon have been removed

Return type

dict

in3Utils.geoTrans.checkProfile

checkProfile(*avaProfile*, *projSplitPoint=None*)

check that the avalanche profiles goes from top to bottom flip it if not and adjust the splitpoint in consequence

Parameters

- **avaProfile** (*dict*) – line dictionary with x and y coordinates
- **projSplitPoint** (*dict*) – a point dictionary already projected on the avaProfile

Returns

- **avaProfile** (*dict*) – avaProfile, fliped if needed
- **projSplitPoint** (*dict*) – point dictionary

in3Utils.geoTrans.computeAlongLineDistance

computeAlongLineDistance(*line*, *dim='2D'*)

compute distance along a dict of coordinates or a shapely lineString incrementally

Parameters

- **line** (*lineString shapely or dict*) – lineString object or dict with x, y, z keys and np.arrays as items
- **dim** (*str*) – 2D only in xy, 3D in xyz

Returns
distancePoints – list of starting (at zero) distance along this line

Return type
list

in3Utils.geoTrans.computeLengthOfLine2D

computeLengthOfLine2D(*x*, *y*)

compute distance along a line in 2D

Parameters
x, y (*np array*) – x, y coordinates of line

Returns
s – accumulated distance measured along line from point to point

Return type
np array

in3Utils.geoTrans.computeS

computeS(*avaPath*)

compute s coordinate given a path (x, y)

Parameters
avaPath (*dict*) – path dictionary with x and y coordinates as 1D numpy arrays

Returns
avaPath – path dictionary updated with s coordinate

Return type
dict

in3Utils.geoTrans.findAngleProfile

findAngleProfile(*tmp*, *ds*, *dsMin*)

Find the beta point: first point under the beta value given in prepareAngleProfile. Make sure that at least dsMin meters behind the point are also under the beta value otherwise keep searching

Parameters

- **tmp** (*1D numpy array*) – index array of point in profile with slope bellow the given beta angle and bellow the splitPoint

- **ds** (*1D numpy array*) – distance between points discribed in tmp
- **dsMin** (*float*) – threshold distance [m] for looking for the beta point (at least dsMin meters below beta degres)

Returns
idsAnglePoint – index of beta point

Return type
int

in3Utils.geoTrans.findClosestPoint

`findClosestPoint`(*xcoor*, *ycoor*, *pointsDict*)

find the closest point of pointDict along line defined by xcoor and ycoor - only xy plane!

Parameters

- **xcoor, ycoor** (*np array*) – x and y coordinates of line
- **pointsDict** (*dict*) – a dictionary with coordinates of points with keys x and y

Returns
indSplit – index of closest point found on the line

Return type
int

in3Utils.geoTrans.findPointOnDEM

`findPointOnDEM`(*dem*, *vDirX*, *vDirY*, *vDirZ*, *zHighest*, *xFirst*, *yFirst*, *zFirst*)

find point on dem given a direction and a z value to reach

Parameters

- **dem** (*dict*) – dem dict
- **vDirX, vDirY, vDirZ** (*floats*) – x, y and z components of the direction in which to extend
- **zHighest** (*float*) – z value to reach
- **xFirst, yFirst, zFirst** (*floats*) – x, y and z coordinates of the starting point

Returns
xExtTop, yExtTop, zExtTop – x, y and z coordinates of the point found

Return type
floats

in3Utils.geoTrans.findSplitPoint

`findSplitPoint`(*avaProfile*, *Points*)

Finds the closest point in Points to the avaProfile and returns its projection on avaProfile.

Parameters

- **avaProfile** (*dict*) – line dictionary with x and y coordinates
- **Points** (*dict*) – a point dictionary

Returns

projPoint – point dictionary projected on the profile (if several points were give in input, only the closest point to the profile is projected)

Return type

dict

in3Utils.geoTrans.getCellsAlongLine

`getCellsAlongLine`(*header*, *lineDict*, *addBuffer=True*)

Find all raster cells crossed by the line line has to be entierly contained on the raster extend. If addBuffer is True, add neighbour cells to the result based on https://stackoverflow.com/a/35808540/15887086

Parameters

- **header** (*dict*) – raster header
- **lineDict** (*dict*) – line dictionary
- **addBuffer** (*boolean*) – True to add a 1 cell buffer around the line

Returns

lineDict – line dictionary updated with the "cellsCrossed" 1D array (boolean array of 0 and 1 if the cell is crossed by the line or in its neigborhood)

Return type

dict

in3Utils.geoTrans.getNeighborCells

`getNeighborCells`(*indX*, *indY*, *ncols*, *nrows*, *cellsArray*)

Find the neighbour cells to a given cell

Parameters

- **indX** (*int*) – x index of the cell for which you want to find the direct neighbors
- **indY** (*int*) – y index of the cell for which you want to find the direct neighbors
- **ncols** (*int*) – number of cols in the raster
- **nrows** (*int*) – number of rows in the raster
- **cellsArray** (*1D int array*) – boolean array of 0 and 1 if the cell is crossed by the line or in its neigborhood

Returns

cellsArray – updated boolean array of 0 and 1 if the cell is crossed by the line or in its neigborhood

Return type
1D int array

in3Utils.geoTrans.isCounterClockWise

`isCounterClockWise`(*path*)

Determines if a polygon path is mostly clockwise or counter clockwise

https://stackoverflow.com/a/45986805/15887086

Parameters
path (*matplotlib.path*) – polygon path

Returns
isCounterCloc1 – 1 if the path is counter clockwise, 0 otherwise

Return type
int

in3Utils.geoTrans.makeCoordGridFromHeader

`makeCoordGridFromHeader`(*rasterHeader*, *cellSizeNew=None*, *larger=False*)

Get x and y (2D) grid description vectors for a mesh with a given number of rows and columns, lower left center and cellSize. If 'cellSizeNew' is not None use cellSizeNew instead of rasterHeader['cellsize'] Make sure the new grid is at least as big as the old one if larger=True (can happen if 'cellSizeNew' is not None)

Parameters

- **rasterHeader** (*dict*) – ratser header with info on ncols, nrows, csz, xllcenter, yllcenter, no-data_value
- **cellSizeNew** (*float*) – If not None, use cellSizeNew as cell size
- **larger** (*boolean*) – If True, make sure the extend of the (xGrid, yGrid) is larger or equal than the header one

Returns

- **xGrid, yGrid** (*2D numpy arrays*) – 2D vector of x and y values for mesh center coordinates (produced using meshgrid)
- **ncols, nrows** (*int*) – number of columns and rows

in3Utils.geoTrans.makeCoordinateGrid

`makeCoordinateGrid`(*xllc*, *yllc*, *csz*, *ncols*, *nrows*)

Create grid

Parameters

- **xllc, yllc** (*float*) – x and y coordinate of the lower left center
- **csz** (*float*) – cell size
- **ncols, nrows** (*int*) – number of columns and rows

Returns
xGrid, yGrid – 2D vector of x and y values for mesh center coordinates (produced using meshgrid)

Return type
2D numpy arrays

in3Utils.geoTrans.path2domain

`path2domain`(*xyPath*, *rasterTransfo*)

Creates a domain (irregular raster) along a path, given the path xyPath, a domain width and a raster cellsize

Parameters:

xyPath: dict
line dictionary with coordinates x and y

rasterTransfo: dict

rasterTransfo['w']: float
Domain width

rasterTransfo['cellSizeSL']: float
cellsize expected for the new raster

Returns:

rasterTransfo: dict
rasterTransfo updated with xp, yp Arrays determining a path of width w along a line

rasterTransfo['DBXl']:
x coord of the left boundary

rasterTransfo['DBXr']:
x coord of the right boundary

rasterTransfo['DBYl']:
y coord of the left boundary

rasterTransfo['DBYr']:
y coord of the right boundary

[Fischer2013] Fischer, Jan-Thomas. (2013). A novel approach to evaluate and compare computational snow avalanche simulation. Natural Hazards and Earth System Sciences. 13. 1655-. 10.5194/nhess-13-1655-2013.
Uwe Schlifkowitz/ BFW, June 2011

in3Utils.geoTrans.pointInPolygon

pointInPolygon(*demHeader*, *points*, *Line*, *radius*)

find particles within a polygon

Parameters

- **demHeader** (*dict*) – dem header dictionary
- **points** (*dict*) – points to check
- **Line** (*dict*) – line dictionary
- **radius** (*float*) – threshold val that decides if a point is in the polygon, on the line or very close but outside

Returns
Mask – Mask of particles to keep

Return type
1D numpy array

in3Utils.geoTrans.polygon2Raster

polygon2Raster(*demHeader*, *Line*, *radius*, *th=''*)

convert line to raster

Parameters

- **demHeader** (*dict*) – dem header dictionary
- **Line** (*dict*) – line dictionary
- **radius** (*float*) – include all cells which center is in the polygon or close enough
- **th** (*float*) – thickness value ot the line feature

Returns
Mask – updated raster

Return type
2D numpy array

in3Utils.geoTrans.prepareAngleProfile

prepareAngleProfile(*beta*, *avaProfile*, *raiseWarning=True*)

Prepare inputs for findAngleProfile function Read profile (s, z), compute the slope Angle look for points for which the slope is under the given Beta value and that are located downstream of the splitPoint

Parameters

- **beta** (*float*) – beta angle in degrees
- **avaProfile** (*dict*) – profile dictionary, s, z and a split point(optional)
- **raiseWarning** (*bool*) – True to raise eventual warnings

Returns

- **angle** (*1D numpy array*) – profile angle

- **tmp** (*1D numpy array*) – index array of point in profile with slope bellow the given beta angle and bellow the splitPoint
- **ds** (*1D numpy array*) – distance between points discribed in tmp

in3Utils.geoTrans.prepareArea

`prepareArea`(*line*, *dem*, *radius*, *thList=''*, *combine=True*, *checkOverlap=True*)

convert shape file polygon to raster

Parameters

- **line** (*dict*) – line dictionary
- **dem** (*dict*) – dictionary with dem information
- **radius** (*float*) – include all cells which center is in the polygon or close enough
- **thList** (*list*) – thickness values for all features in the line dictionary
- **combine** (*Boolean*) – if True sum up the rasters in the area list to return only 1 raster if False return the list of distinct area rasters this option works only if thList is not empty
- **checkOverlap** (*Boolean*) – if True check if features are overlapping and return an error if it is the case if False check if features are overlapping and average the value for overlapping areas (Attention: if combine is set to False, you do not see the result of the averaging since the list of raters was not affected by the averaging step)

Returns

contains either

- Raster: 2D numpy array, raster of the area (returned if relRHlist is empty OR if combine is set

to True) - RasterList: list, list of 2D numpy array rasters (returned if relRHlist is not empty AND if combine is set to False)

Return type
updates the line dictionary with the rasterData

in3Utils.geoTrans.prepareLine

`prepareLine`(*dem*, *avapath*, *distance=10*, *Point=None*)

Resample and project line on dem 1- Resample the avapath line with an interval of approximately distance in meters between points (projected distance on the horizontal plane). 2- Make avalanche profile out of the path (affect a z value using the dem) 3- Get projection of points on the profil (closest point)

Parameters

- **dem** (*dict*) – dem dictionary
- **avapath** (*dict*) – line dictionary
- **distance** (*float*) – resampling distance
- **Point** (*dict*) – a point dictionary (optional, can contain several point)

Returns

- **avaProfile** (*dict*) – the resampled avapath with the z coordinate

- **projPoint** (*dict*) – point dictionary projected on the profile (if several points were give in input, only the closest point to the profile is projected)

in3Utils.geoTrans.projectOnGrid

`projectOnGrid`(*x*, *y*, *Z*, *csz=1*, *xllc=0*, *yllc=0*, *interp='bilinear'*)

Projects Z onto points (x,y) using a bilinear or nearest interpolation and returns the z coord

Parameters

- **x** (*array*) – x coord of the points to project
- **y** (*array*) – y coord of the points to project
- **Z** (*2D numpy array*) – raster data
- **csz** (*float*) – cellsize corresponding to the raster data
- **xllc** (*float*) – x coord of the lower left center of the raster
- **yllc** (*float*) – y coord of the lower left center of the raster
- **interp** (*str*) – interpolation option, between nearest or bilinear

Returns

- **z** (*2D numpy array*) – projected data on the raster data
- **ioob** (*int*) – number of out of bounds indexes

in3Utils.geoTrans.projectOnRaster

`projectOnRaster`(*dem*, *Points*, *interp='bilinear'*, *inData='rasterData'*, *outData='z'*)

Projects Points on raster using a bilinear or nearest interpolation and returns the z coord (no for loop)

Parameters

- **dem** (*dict*) – dem dictionary
- **Points** (*dict*) – Points dictionary (x,y)
- **interp** (*str*) – interpolation option, between nearest or bilinear
- **inData** (*str*) – key in the dem dict of the 2D field to use for the interpolation.
- **outData** (*str*) – key in the Points dict toe updat with the interpolated data.

Returns

- **Points** (*dict*) – Points dictionary with z coordinate added or updated
- **ioob** (*int*) – number of out of bounds indexes

in3Utils.geoTrans.remeshDEM

remeshDEM(*demFile*, *cfgSim*, *onlySearch=False*)

change DEM cell size by reprojecting on a new grid - first check if remeshed DEM available

the new DEM is as big or smaller as the original DEM and saved to Inputs/DEMremshed as remeshedDEMcellSize

Interpolation is based on griddata with a cubic method. Here would be the place to change the order of the interpolation or to switch to another interpolation method.

Parameters

- **demFile** (*str or pathlib path*) – file path to DEM in Inputs/
- **cfgSim** (*configParser*) – meshCellSizeThreshold : threshold under which no remeshing is done meshCellSize : desired cell size
- **onlySearch** (*bool*) – if True - only searching for remeshed DEM but not remeshing if not found

Returns
pathDem – path of DEM with desired cell size relative to Inputs/

Return type
str

in3Utils.geoTrans.remeshData

remeshData(*rasterDict*, *cellSizeNew*, *remeshOption='griddata'*, *interpMethod='cubic'*, *larger=True*)

compute raster data on a new mesh with cellSize using the specified remeshOption.

remeshOption are to choose between 'griddata' or 'RectBivariateSpline' Only the 'griddata' works properly if the input data contains noData points, 'RectBivariateSpline' is faster but fails if input data contains noData points. The new mesh is as big or smaller as the original mesh if larger is False and bigger if larger is True

Parameters

- **rasterDict** (*dict*) – raster dictionary (with header and rasterData)
- **cellSizeNew** (*float*) – mesh size of new mesh
- **remeshOption** (*str*) – method used to remesh ('griddata' or 'RectBivariateSpline') Check the scipy documentation for more details default is 'griddata'
- **interpMethod** (*str*) – interpolation order to use for the interpolation ('linear', 'cubic' or 'quintic')
- **larger** (*Boolean*) – if true (default) output grid is at least as big as the input

Returns
data – remeshed data dict with data as numpy array and header info

Return type
dict

in3Utils.geoTrans.resizeData

`resizeData`(*raster*, *rasterRef*)

Reproject raster on a grid of shape rasterRef

Parameters

- **raster** (*dict*) – raster dictionary
- **rasterRef** (*dict*) – reference raster dictionary

Returns

- **data** (*2D numpy array*) – reprojected data
- **dataRef** (*2D numpy array*) – reference data

in3Utils.geoTrans.rotate

`rotate`(*locationPoints*, *theta*, *deg=True*)

rotate a vector provided as start and end point with theta angle rotation counter-clockwise

Parameters

- **locationPoints** (*list*) – list of lists with x,y coordinate of start and end point of a line
- **theta** (*float*) – rotation angle of the vector from start point to end point - degree default
- **deg** (*bool*) – if true theta is converted to rad from degree

Returns

rotatedLine – list of lists of x,y coordinates of start and end point of rotated vector

Return type

list

in3Utils.geoTrans.rotateRaster

`rotateRaster`(*rasterDict*, *theta*, *deg=True*)

rotate clockwise a raster arround (0, 0) with theta angle

Parameters

- **rasterDict** (*dict*) – raster dictionary
- **theta** (*float*) – rotation angle of the vector from start point to end point - degree default
- **deg** (*bool*) – if true theta is converted to rad from degree

Returns

rotatedRaster – rotated raster dictionary

Return type

dict

in3Utils.geoTrans.searchRemeshedDEM

`searchRemeshedDEM`(*demName*, *cfgSim*)

search if remeshed DEM with correct name and cell size already available

Parameters

- **demName** (*str*) – name of DEM file in Inputs/
- **cfgSim** (*configparser object*) – configuration settings: avaDir, meshCellSize, meshCellSizeThreshold

Returns

- **remshedDEM** (*dict*) – dictionary of remeshed DEM if not found empty dict
- **DEMFound** (*bool*) – flag if dem is found
- **allDEMNames** (*list*) – of all names of dems found in Inputs/DEMremeshed

in3Utils.geoTrans.snapPtsToLine

`snapPtsToLine`(*dbData*, *projstr*, *lineName*, *pointsList*)

snap points to line in dataframe only considering x, y plane!

Parameters

- **dbData** (*pandas dataframe*) – dataframe with geometry info of events
- **lineName** (*str*) – name of line column except projstr
- **pointsList** (*list*) – list with point column names except projstr
- **projstr** (*str*) – projection string to append to all names

Returns
dbData – updated dataframe with . . . _snapped point column

Return type
pandas dataframe

in3Utils.getReleaseArea

Get release area corner coordinates for a rectangle of an area of approx. 100 000 m2

This file is part of Avaframe.

Functions

correctOrigin	Move the points so that the correct origin is set
getCornersFP	
getCornersHS	
getCornersIP	
getReleaseArea	Main function to compute release areas
makexyPoints	Make xy Points of release area from given extent and start and end points
writeReleaseArea	Write topography information to file

in3Utils.getReleaseArea.correctOrigin

correctOrigin(*xyPoints*, *cfgT*)

Move the points so that the correct origin is set

in3Utils.getReleaseArea.getCornersFP

getCornersFP(*cfgR*)

in3Utils.getReleaseArea.getCornersHS

getCornersHS(*cfgR*, *cfgT*)

in3Utils.getReleaseArea.getCornersIP

getCornersIP(*cfgR*, *cfgT*)

in3Utils.getReleaseArea.getReleaseArea

getReleaseArea(*cfgT*, *cfgR*, *avalancheDir*)

Main function to compute release areas

in3Utils.getReleaseArea.makexyPoints

makexyPoints(*x1*, *x2*, *y1*, *cfgR*)

Make xy Points of release area from given extent and start and end points

in3Utils.getReleaseArea.writeReleaseArea

writeReleaseArea(*xyPoints*, *demType*, *cfgR*, *outDir*)

Write topography information to file

in3Utils.initialiseDirs

Setup required directory structures by computational modules

Functions

initialiseRunDirs	Initialise Simulation run with input data

in3Utils.initialiseDirs.initialiseRunDirs

initialiseRunDirs(*avaDir*, *modName*, *cleanDEMremeshed*)

Initialise Simulation run with input data

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **modName** (*str*) – name of module
- **cleanDEMremeshed** (*bool*) – if True directory Inputs/DEMremeshed shall be cleaned

Returns

- **workDir** (*str*) – path to Work directory
- **outputDir** (*str*) – path to Outputs directory

in3Utils.initializeProject

Functions to initialize project, i.e create folder structure

This file is part of Avaframe.

Functions

checkMakeDir	combines base and dir, checks whether it exists, if not creates it
cleanDEMremeshedDir	clean DEMremeshed folder in avaDir Inputs
cleanModuleFiles	Cleans all generated files from the provided module in the outputs folder and work folder
cleanSingleAvaDir	Clean a single avalanche directory of the work and output directories
createFolderStruct	creates the folder structure with avalanche base path
initializeFolderStruct	Initialize the standard folder structure.

in3Utils.initializeProject.checkMakeDir

checkMakeDir(*base*, *dirName*)

combines base and dir, checks whether it exists, if not creates it

in3Utils.initializeProject.cleanDEMremeshedDir

cleanDEMremeshedDir(*avaDir*)

clean DEMremeshed folder in avaDir Inputs

Parameters

avaDir (*str or pathlib patch*) – path to avalanche directory

in3Utils.initializeProject.cleanModuleFiles

cleanModuleFiles(*avaDir*, *module*, *alternativeName=''*, *deleteOutput=True*)

Cleans all generated files from the provided module in the outputs folder and work folder

Parameters

- **avaDir** (*path/string*) – Avalanche directory path
- **module** (*module object*) – The module do delete. e.g. from avaframe.com2AB import com2AB leads to cleanModuleFiles(com2AB) whereas from avaframe.com2AB import com2AB as c2 leads to cleanModuleFiles(c2) Boolean to be able to avoid deletion of Outputs (true by default)

in3Utils.initializeProject.cleanSingleAvaDir

cleanSingleAvaDir(*avaDir*, *deleteOutput=True*)

Clean a single avalanche directory of the work and output directories

Parameters

- **avaDir** (*path/string*) – Avalanche directory path
- **deleteOutput** (*boolean*) – If True (default), directory output and the log files are deleted

in3Utils.initializeProject.createFolderStruct

createFolderStruct(*pathAvaName*)

creates the folder structure with avalanche base path

in3Utils.initializeProject.initializeFolderStruct

`initializeFolderStruct`(*pathAvaName*, *removeExisting=False*)

Initialize the standard folder structure. If removeExisting is true, deletes any existing folders! BEWARE!

Parameters

- **pathAvaName** (*str*) – string with base avalanche path
- **removeExisting** (*boolean, optional, default: False*) – remove existing directory, use this to completely clean out the directory

in3Utils.logUtils

Defining a writable object to write the config to the log file

This file is part of Avaframe.

Functions

initiateLogger	Initiates logger object based on setup in logging.conf
writeCfg2Log	write a configparser object to log file

in3Utils.logUtils.initiateLogger

`initiateLogger`(*targetDir*, *logName='runLog'*, *modelInfo=''*)

Initiates logger object based on setup in logging.conf

Parameters

- **targetDir** (*str*) – folder to save log file to
- **logName** (*str*) – filename of log file; optional; defaults to runLog.log
- **modelInfo** (*str*) – if not emtpy add info on modelInfo to log

Returns
log

Return type
logging object

in3Utils.logUtils.writeCfg2Log

`writeCfg2Log`(*cfg*, *cfgName='Unspecified'*)

write a configparser object to log file

2.10.3 Analysis/Output/Helper Modules

ana1Tests	Code for ana1Tests module
ana3AIMEC	Code for ana3AIMEC analysis module
ana4Stats	Tools for statistical analysis of simulation results
ana5Utils	Generate thalweg-time-diagrams and simulograms
log2Report	Tools for generating reports
out1Peak	Simple tools for visualising datasets.
out3Plot	Plotting
tmp1Ex.tmp1Ex	This is the template for new modules, with the bare minimal required files

2.10.3.1 ana1Tests

Code for ana1Tests module

Modules

ana1Tests.FPtest	Flat plane test
ana1Tests.analysisTools	
ana1Tests.damBreak	Simple python script to reproduce analytic solution for a Riemann problem, following the derivations in Faccanoni and Mangeney (2012), Test 2, Case 1.2.
ana1Tests.energyLineTest	Energy line test This module runs a DFA simulation extracts the center of mass path and compares it to the analytic geometric/alpha line solution
ana1Tests.rotationTest	Rotation test
ana1Tests.simiSolTest	Similarity solution module
ana1Tests.testUtilities	Functions for handling benchmark tests data, filtering, sorting

ana1Tests.FPtest

Flat plane test

Functions

getReleaseThickness	define release thickness for Flat Plane solution test
plotProfilesFPtest	Plot flow thickness and gradient for FlatPlane simulation results
postProcessFPcom1DFA	get fields and particles dictionaries for given time step

ana1Tests.FPtest.getReleaseThickness

getReleaseThickness(*avaDir*, *cfg*, *demFile*)

define release thickness for Flat Plane solution test

ana1Tests.FPtest.plotProfilesFPtest

plotProfilesFPtest(*cfg*, *ind_time*, *relDict*, *comSol*)

Plot flow thickness and gradient for FlatPlane simulation results

Parameters

- **cfg** (*configparser*)
- **ind_time** (*int*) – time index for simiSol
- **relDict** (*dict*) – dictionary of release area info
- **comSol** (*dict*) – dictionary of simulation results and info (particles, fields, indices, time step)

ana1Tests.FPtest.postProcessFPcom1DFA

postProcessFPcom1DFA(*cfgGen*, *particles*, *fields*, *ind_t*, *relDict*)

get fields and particles dictionaries for given time step

ana1Tests.analysisTools

Functions

L2Norm	Compute L2 norm of an array
computeErrorAndNorm	Compute error between two functions given their norm 2
normL2Scal	Compute L2 and Lmax norm of the error between the analytic and numerical solution
normL2Vect	Compute L2 and Lmax norm of the error between the analytic and numerical solution

ana1Tests.analysisTools.L2Norm

L2Norm(*norm2Array*, *cellSize*, *cosAngle*)

Compute L2 norm of an array

Parameters

- **norm2Array** (*numpy array*) – norm2 of the function
- **cellSize** (*float*) – grid cell size
- **cosAngle** (*float*) – cosine of the slope angle

Returns

normL2 – normL2 of the function

Return type
float

ana1Tests.analysisTools.computeErrorAndNorm

`computeErrorAndNorm`(*localError*, *analyticalSol2*, *cellSize*, *cosAngle*)

Compute error between two functions given their norm 2

Parameters

- **localError** (*numpy array*) – norm2 of the error function
- **analyticalSol2** (*numpy array*) – norm2 of the reference function
- **cellSize** (*float*) – grid cell size
- **cosAngle** (*float*) – cosine of the slope angle

Returns

- **errorL2** (*float*) – L2 error
- **errorL2Rel** (*float*) – Relativ L2 error
- **errorMax** (*float*) – LMax error
- **errorMaxRel** (*float*) – Relativ LMax error

ana1Tests.analysisTools.normL2Scal

`normL2Scal`(*analyticalSol*, *numericalSol*, *cellSize*, *cosAngle*)

Compute L2 and Lmax norm of the error between the analytic and numerical solution

Parameters

- **analyticalSol** (*numpy array*) – analytic solution array
- **numericalSol** (*numpy array*) – numericalSol solution array
- **cellSize** (*float*) – grid cell size
- **cosAngle** (*float*) – cosine of the slope angle

Returns

- **errorL2** (*float*) – L2 error
- **errorL2Rel** (*float*) – Relativ L2 error
- **errorMax** (*float*) – LMax error
- **errorMaxRel** (*float*) – Relativ LMax error

ana1Tests.analysisTools.normL2Vect

normL2Vect(*analyticalSol*, *numericalSol*, *cellSize*, *cosAngle*)

Compute L2 and Lmax norm of the error between the analytic and numerical solution

Parameters

- **analyticalSol** (*dictionary*) – analytic solution dictionary
  - fx: x component of the vector
  - fy: y component of the vector
  - fz: z component of the vector
- **numericalSol** (*dictionary*) – numericalSol solution dictionary
  - fx: x component of the vector
  - fy: y component of the vector
  - fz: z component of the vector
- **cellSize** (*float*) – grid cell size
- **cosAngle** (*float*) – cosine of the slope angle

Returns

- **errorL2** (*float*) – L2 error
- **errorL2Rel** (*float*) – Relativ L2 error
- **errorMax** (*float*) – LMax error
- **errorMaxRel** (*float*) – Relativ LMax error

ana1Tests.damBreak

Simple python script to reproduce analytic solution for a Riemann problem, following the derivations in Faccanoni and Mangeney (2012), Test 2, Case 1.2. but scaled up in size.

Here the instantanous release of fluid from rest is described using incompressible, thickness-avaeraged mass and momentum conservation equations and a Coulomb-tpye friction law.

Functions

analyzeResults	Compare analytical and com1DFA results, compute error
damBreakSol	Compute analytical Flow thickness and velocity for dam break test case
getDamExtend	Get the extend where the analytic and simulation should be compared
postProcessDamBreak	loop on all DFA simulations and compare then to the anlytic solution

ana1Tests.damBreak.analyzeResults

analyzeResults(*avalancheDir*, *fieldsList*, *timeList*, *solDam*, *fieldHeader*, *cfg*, *outDirTest*, *simHash*, *simDFrow*)

Compare analytical and com1DFA results, compute error

Parameters

- **avalancheDir** (*pathlib path*) – path to avalanche directory
- **fieldsList** (*list*) – list of fields dictionaries
- **timeList** (*list*) – list of time steps
- **solDam** (*dict*) – analytic solution dictionary
- **fieldHeader** (*dict*) – header dictionary with info about the extend and cell size
- **cfg** (*configParser object*) – configuration setting for avalanche simulation including DAMBREAK section
- **outDirTest** (*pathlib path*) – path output directory (where to save the figures)
- **simHash** (*str*) – com1DFA simulation id
- **simDFrow** (*pandas object*) – com1DFA simulation rox coresponding to simHash

Returns

- **hErrorL2Array** (*numpy array*) – L2 error on Flow thickness for saved time steps
- **hErrorLMaxArray** (*numpy array*) – LMax error on Flow thickness for saved time steps
- **vErrorL2Array** (*numpy array*) – L2 error on flow velocity for saved time steps
- **vErrorLMaxArray** (*numpy array*) – LMax error on flow velocity for saved time steps
- **tSave** (*float*) – time corresponding the errors

ana1Tests.damBreak.damBreakSol

damBreakSol(*avaDir*, *cfg*, *cfgC*, *outDirTest*)

Compute analytical Flow thickness and velocity for dam break test case

for a granular flow over a dry rough sloping bed with the Savage Hutter model

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **cfg** (*configParser object*) – main avaframe configuration - here showPlot flag used
- **cfgC** (*configParser object*) – configuration setting for avalanche simulation including DAMBREAK section
- **outDirTest** (*pathlib path*) – path to output directory

Returns

solDam –

analytic solution dictionary:

tAna: 1D numpy array
time array

h0: float
release thickness

hAna: 2D numpy array
Flow thickness (rows for x and columns for time)

uAna: 2D numpy array
flow velocity (rows for x and columns for time)

xAna: 2D numpy array
extent of domain in the horizontal plane coordinate system (rows for x and columns for time)

xMidAna: 1D numpy array
middle of the material in x dir in the horizontal plane coordinate system (used to compute the error)

Return type
dict

ana1Tests.damBreak.getDamExtend

`getDamExtend`(*fieldHeader*, *xM*, *cfgDam*)

Get the extend where the analytic and simulation should be compared

Parameters

- **fieldHeader** (*dict*) – header dictionary with info about the extend and cell size
- **xM** (*float*) – x coordinate ot the start of the comparizon domain
- **cfgDam** (*configParser object*) – configuration setting for the DAMBREAK section

Returns

- **xDamPlus** (*1D numpy array*) – x array corresponding to the comparizon domain
- **nColMid** (*int*) – index of the column corresponding to xM
- **nColMax** (*int*) – index of the column corresponding to xEnd
- **nRowMin** (*int*) – index of the row corresponding to yStart
- **nRowMax** (*int*) – index of the row corresponding to yEnd

ana1Tests.damBreak.postProcessDamBreak

`postProcessDamBreak`(*avalancheDir*, *cfgMain*, *cfgDam*, *simDF*, *solDam*, *outDirTest*)

loop on all DFA simulations and compare then to the anlytic solution

Parameters

- **avalancheDir** (*str or pathlib path*) – avalanche directory
- **cfgMain** (*confiparser*) – avaframeCfg configuration
- **cfgDam** (*configParser object*) – configuration setting for avalanche simulation including DAMBREAK section
- **simDF** (*pandas dataFrame*) – configuration DF
- **solDam** (*dict*) – analytic solution dictionary

- **outDirTest** (*pathlib path*) – path to output directory

Returns
simDF – configuration DF appended with the analysis results

Return type
pandas dataFrame

ana1Tests.energyLineTest

Energy line test This module runs a DFA simulation extracts the center of mass path and compares it to the analytic geometric/alpha line solution

Functions

generateCom1DFAEnergyPlot	Make energy test analysis and plot results
getAlphaProfileIntersection	Extend the profile path and compute the intersection between the theoretical energy line and the path profile The profile is extended by a line.
getEnergyInfo	Compute energy dots and errors
getRunOutAngle	Compute the center of mass runout angle
mainEnergyLineTest	This is the core function of the energyLineTest module This module extracts the center of mass path from a DFA simulation and compares it to he analytic geometric/alpha line solution

ana1Tests.energyLineTest.generateCom1DFAEnergyPlot

generateCom1DFAEnergyPlot(*avalancheDir*, *energyLineTestCfg*, *com1DFACfg*, *avaProfileMass*, *dem*, *fieldsList*, *simName*)

Make energy test analysis and plot results

Parameters

- **avalancheDir** (*pathlib*) – avalanche directory pathlib path
- **energyLineTestCfg** (*configParser*) – energy line test config
- **com1DFACfg** (*configParser*) – com1DFA config
- **avaProfileMass** (*dict*) – particle mass averaged properties
- **dem** (*dict*) – com1DFA simulation dictionary
- **fieldsList** (*list*) – field dictionary list
- **simName** (*str*) – simulation name

ana1Tests.energyLineTest.getAlphaProfileIntersection

`getAlphaProfileIntersection`(*energyLineTestCfg*, *avaProfileMass*, *mu*, *csz*)

Extend the profile path and compute the intersection between the theoretical energy line and the path profile The profile is extended by a line. The line slope is computed from the slope of the regression on the last points of the profile

Parameters

- **energyLineTestCfg** (*configParser*) – energy test config
- **avaProfileMass** (*dict*) – particle mass average properties
- **mu** (*float*) – friction coefficient
- **csz** (*float*) – dem cell size

Returns

- **slopeExt** (*float*) – slope of the extrapolation line
- **sIntersection** (*float*) – s coord of the intersection between the line of slope mu and the mass average path profile
- **zIntersection** (*float*) – z coord of the intersection between the line of slope mu and the mass average path profile
- **coefExt** (*float*) – coefficient saying how long the path was extended to find the intersection

ana1Tests.energyLineTest.getEnergyInfo

`getEnergyInfo`(*avaProfileMass*, *g*, *mu*, *sIntersection*, *zIntersection*, *runOutAngleDeg*, *alphaDeg*)

Compute energy dots and errors

Parameters

- **avaProfileMass** (*dict*) – particle mass average properties
- **g** (*float*) – gravity
- **mu** (*float*) – friction coefficient
- **sIntersection** (*float*) – s coord of the intersection between the line of slope mu and the mass average path profile
- **zIntersection** (*float*) – z coord of the intersection between the line of slope mu and the mass average path profile
- **runOutAngleRad** (*float*) – center of mass runout angle in radians
- **runOutAngleDeg** (*float*) – center of mass runout angle in degrees

Returns

- **zEne** (*numpy 1D array*) – energy height of the particle averaged time steps
- **u2Path** (*numpy 1D array*) – kinetic energy of the particle averaged time steps
- **sGeomL** (*2 element list*) – s coord (start and end) of the run out angle line
- **zGeomL** (*2 element list*) – z coord (start and end) of the run out angle line

- **resultEnergyTest** (*dict*) – zEnd, sEnd, runoutAngle as well as rmseVelocityElevation, runOutSError, runOutZError, runOutAngleError between theoretical solution and simulation result

ana1Tests.energyLineTest.getRunOutAngle

getRunOutAngle(*avaProfileMass*, *indStart=0*, *indEnd=-1*)

Compute the center of mass runout angle

Parameters

- **avaProfileMass** (*dict*) – particle mass average properties
- **indStart** (*int*) – index of the start of the mass averaged path (to enable e.g. to discard the top extension). 0 by default - so full mass averaged path from top
- **indEnd** (*int*) – index of the start of the mass averaged pass (to discard the bottom extension). -1 by default

Returns

- **runOutAngleRad** (*float*) – Center of Mass runout angle in radians
- **runOutAngleDeg** (*float*) – Center of Mass runout angle in degrees

ana1Tests.energyLineTest.mainEnergyLineTest

mainEnergyLineTest(*avalancheDir*, *energyLineTestCfg*, *com1DFACfg*, *simName*, *dem*)

This is the core function of the energyLineTest module This module extracts the center of mass path from a DFA simulation and compares it to he analytic geometric/alpha line solution

ana1Tests.rotationTest

Rotation test

This module runs a DFA simulation for different grid alignments and compares the results

Functions

buildRotationTestReport	Write the rotation test report
initializeRotationTestReport	Initialize dictionary that is used for markdown report generation for rotation test
mainRotationTest	This is the core function of the Rotation Test module
rotateDFAResults	Rotate the DFA results

ana1Tests.rotationTest.buildRotationTestReport

buildRotationTestReport(*avalancheDir*, *reportRotationTest*, *simDF*, *resAnalysisDF*, *aimecPlotDict*, *flagMass*)

Write the rotation test report

Parameters

- **avalancheDir** (*pathlib path*) – path to avalanche directory
- **reportRotationTest** (*dict*) – report dictionary
- **simDF** (*dataFrame*) – DFA simulation dataFrame
- **resAnalysisDF** (*dataFrame*) – aimec results dataFrame
- **aimecPlotDict** (*dict*) – aimec plots dictionary
- **flagMass** (*boolean*) – Was a mass analysis conducted?

Return type
generates the markDown report

ana1Tests.rotationTest.initializeRotationTestReport

initializeRotationTestReport(*avalancheDir*, *resTypeList*, *comModule*, *refSimName*, *flagMass*)

Initialize dictionary that is used for markdown report generation for rotation test

Parameters

- **avalancheDir** (*pathlib path*) – path to avalanche directory
- **resTypeList** (*list*) – list of result types available
- **comModule** (*str*) – computation module name
- **refSimName** (*str*) – name of the reference simulation in the simDF
- **flagMass** (*boolean*) – Was a mass analysis conducted?

Returns
reportRotationTest – report dictionary

Return type
dict

ana1Tests.rotationTest.mainRotationTest

mainRotationTest(*avalancheDir*, *energyLineTestCfg*, *com1DFACfg*, *dem*, *simDF*, *resTypeList*, *flagMass*, *refSimRowHash*, *comModule*)

This is the core function of the Rotation Test module

This module runs the energy line test and rotates the simulation results for the aimec analysis

Parameters

- **avalancheDir** (*pathlib path*) – path to avalanche directory
- **energyLineTestCfg** (*configParser*) – energy line test configuration object
- **com1DFACfg** (*configParser*) – com1DFA configuration object
- **dem** (*dict*) – dem dictionary

- **simDF** (*dataFrame*) – DFA simulation dataFrame
- **resTypeList** (*list*) – list of result types available
- **flagMass** (*boolean*) – should the aimec mass analysis be done
- **refSimRowHash** (*str*) – row index of the reference simulation in the simDF
- **comModule** (*str*) – computation module used for the DFA simulation

Returns

- **simDF** (*dataFrame*) – DFA simulation dataFrame updated with the energy line test results
- **flagMass** (*boolean*) – should the aimec mass analysis be done (switched to true if it is an entrainment simulation)

ana1Tests.rotationTest.rotateDFAResults

`rotateDFAResults`(*avalancheDir*, *simDF*, *rowSimHash*, *resTypeList*, *thetaRef*, *comModule*)

Rotate the DFA results

Parameters

- **avalancheDir** (*pathlib path*) – path to avalanche directory
- **simDF** (*dataFrame*) – DFA simulation dataFrame
- **rowSimHash** (*str*) – row index (in the simDF dataframe) of the simulation to analyze
- **resTypeList** (*list*) – list of result types available
- **thetaRef** (*float*) – angle of the path of the reference simulation
- **comModule** (*str*) – computation module used for the DFA simulation

Return type

saves the rotated rasters to avalancheDir / 'Outputs' / comModule + 'Rotated' / 'peakFiles'

ana1Tests.simiSolTest

Similarity solution module

This module contains functions that compute the similarity solution for a gliding avalanche on a inclined plane according to similarity solution from : Hutter, K., Siegel, M., Savage, S.B. et al. Two-dimensional spreading of a granular avalanche down an inclined plane Part I. theory. Acta Mechanica 100, 37–68 (1993). https://doi.org/10.1007/BF01176861

Functions

Ffunction	Calculate right hand side of the differential equation : dx/dt = F(x,t) F is discribed in Hutter 1993.
analyzeResults	Compare analytical and com1DFA results
calcEarlySol	Compute the early solution for 0<t<t_1 to avoid singularity in the Runge-Kutta integration process
computeEarthPressCoeff	Computes the earth pressure coefficients function of sng of f and g i.e depending on if we are in the active or passive case
computeFCoeff	Compute coefficients eq 3.2 for the function F
computeH	get flow thickness from f and g solutions
computeU	get flow velocity in x direction from f and g solutions
computeV	get flow velocity in y direction from f and g solutions
computeXC	get center of mass location
defineEarthPressCoeff	Define earth pressure coefficients
getReleaseThickness	Define release thickness for the similarity solution test
getSimiSolParameters	get flow thickness, flow velocity and center location of flow mass of similarity solution for required time step
mainSimilaritySol	Compute similarity solution :Parameters: **simiSolCfg** (*pathlib path*) -- path to simiSol configuration file
odeSolver	Solve the ODE using a Runge-Kutta method
postProcessSimiSol	loop on all DFA simulations and compare then to the anlytic solution
prepareParticlesFieldscom1DFA	get fields and particles dictionaries for given time step, for com1DFA domain origin is set to 0,0 for particles - so info on domain is required

ana1Tests.simiSolTest.Ffunction

Ffunction(*t*, *x*, *earthPressureCoefficients*, *zeta*, *delta*, *eps_x*, *eps_y*)

Calculate right hand side of the differential equation : dx/dt = F(x,t) F is discribed in Hutter 1993.

Parameters

- **t** (*float*) – curent time
- **x** (*numpy array*) – initial condition, column vector of size 4
- **earthPressureCoefficients** (*numpy array*) – earth Pressure Coefficients
- **zeta** (*float*) – slope angle
- **delta** (*float*) – friction angle
- **eps_x** (*float*) – scale in x dir
- **eps_y** (*float*) – scale in y dir
- **Returns**
- **F** (*numpy array*) – column vector of size 4

ana1Tests.simiSolTest.analyzeResults

`analyzeResults`(*avalancheDir*, *fieldsList*, *timeList*, *solSimi*, *fieldHeader*, *cfgMain*, *cfgSimi*, *outDirTest*, *simHash*, *simDFrow*)

Compare analytical and com1DFA results

Parameters

- **avalancheDir** (*pathlib path*) – path to avalanche directory
- **fieldsList** (*list*) – list of fields dictionaries
- **timeList** (*list*) – list of time steps
- **solSimi** (*dictionary*) – similarity solution
  - time: time array (without dimension)
  - time: time array (with dimension)
  - g_sol: g array
  - g_p_sol: first derivativ of g array
  - f_sol: f array
  - f_p_sol: first derivativ of f array
- **fieldHeader** (*dict*) – header dictionary with info about the extend and cell size
- **cfgMain** (*dict*) – main confguration
- **cfgSimi** (*dict*) – simisol confguration including SIMISOL section
- **outDirTest** (*pathlib path*) – path output directory (where to save the figures)
- **simHash** (*str*) – com1DFA simulation id

Returns

- **hErrorL2Array** (*numpy array*) – L2 error on flow thickness for saved time steps
- **hErrorLMaxArray** (*numpy array*) – LMax error on flow thickness for saved time steps
- **vErrorL2Array** (*numpy array*) – L2 error on flow velocity for saved time steps
- **vErrorLMaxArray** (*numpy array*) – LMax error on flow velocity for saved time steps
- **tSave** (*float*) – time corresponding the errors

ana1Tests.simiSolTest.calcEarlySol

`calcEarlySol`(*t*, *earthPressureCoefficients*, *x_0*, *zeta*, *delta*, *eps_x*, *eps_y*)

Compute the early solution for 0<t<t_1 to avoid singularity in the Runge-Kutta integration process

Parameters

- **t** (*numpy array*) – time array
- **earthPressureCoefficients** (*numpy array*) – earth Pressure Coefficients
- **x_0** (*numpy array*) – initial condition
- **zeta** (*float*) – slope angle
- **delta** (*float*) – friction angle

- **eps_x** (*float*) – scale in x dir
- **eps_y** (*float*) – scale in y dir

Returns

solSimi – similarity solution (for early times)

- time: time array (without dimension)
- g_sol: g array
- g_p_sol: first derivativ of g array
- f_sol: f array
- f_p_sol: first derivativ of f array

Return type
dictionary

ana1Tests.simiSolTest.computeEarthPressCoeff

`computeEarthPressCoeff`(*x*, *earthPressureCoefficients*)

Computes the earth pressure coefficients function of sng of f and g i.e depending on if we are in the active or passive case

Parameters

- **x** (*float*) – internal friction angle
- **earthPressureCoefficients** (*numpy array*) – [Kxact Kxpass Kyact(Kxact) Kypass(Kxact) Kyact(Kxpass) Kypass(Kxpass)]

Returns

- **K_x** (*float*) – earth pressure coefficient in x direction
- **K_y** (*float*) – earth pressure coefficient in y direction

ana1Tests.simiSolTest.computeFCoeff

`computeFCoeff`(*K_x*, *K_y*, *zeta*, *delta*, *eps_x*, *eps_y*)

Compute coefficients eq 3.2 for the function F

Parameters

- **K_x** (*float*) – Kx earth pressure coef
- **K_y** (*float*) – Ky earth pressure coef
- **zeta** (*float*) – slope angle
- **delta** (*float*) – friction angle
- **eps_x** (*float*) – scale in x dir
- **eps_y** (*float*) – scale in y dir

Returns
A, B, C, D, E – coefficients of eq 3.2

Return type
floats

ana1Tests.simiSolTest.computeH

computeH(*solSimi*, *x1*, *y1*, *i*, *L_x*, *L_y*, *H*, *AminusC*)

get flow thickness from f and g solutions

Parameters

- **solSimi** (*dictionary*) – similarity solution
- **x1** (*numpy array*) – x coordinate location desiered for the solution
- **y1** (*numpy array*) – y coordinate location desiered for the solution
- **i** (*int*) – time index
- **L_x** (*float*) – scale in x dir
- **L_y** (*float*) – scale in y dir
- **H** (*float*) – scale in z dir
- **AminusC** – A-C coefficient

Returns

h – h similarity solution at (x1, y1)

Return type

numpy array

ana1Tests.simiSolTest.computeU

computeU(*solSimi*, *x1*, *y1*, *i*, *L_x*, *U*, *AminusC*)

get flow velocity in x direction from f and g solutions

Parameters

- **solSimi** (*dictionary*) – similarity solution
- **x1** (*numpy array*) – x coordinate location desiered for the solution
- **y1** (*numpy array*) – y coordinate location desiered for the solution
- **i** (*int*) – time index
- **L_x** (*float*) – scale in x dir
- **U** (*float*) – x velocity component scale
- **AminusC** – A-C coefficient

Returns

u – u similarity solution at (x1, y1)

Return type

numpy array

ana1Tests.simiSolTest.computeV

`computeV`(*solSimi*, *x1*, *y1*, *i*, *L_y*, *V*)

get flow velocity in y direction from f and g solutions

Parameters

- **solSimi** (*dictionary*) – similarity solution
- **x1** (*numpy array*) – x coordinate location desiered for the solution
- **y1** (*numpy array*) – y coordinate location desiered for the solution
- **i** (*int*) – time index
- **L_y** (*float*) – scale in y dir
- **V** (*float*) – y velocity component scale

Returns

v – v similarity solution at (x1, y1)

Return type

numpy array

ana1Tests.simiSolTest.computeXC

`computeXC`(*solSimi*, *x1*, *y1*, *i*, *L_x*, *AminusC*)

get center of mass location

Parameters

- **solSimi** (*dictionary*) – similarity solution
- **x1** (*numpy array*) – x coordinate location desiered for the solution
- **y1** (*numpy array*) – y coordinate location desiered for the solution
- **i** (*int*) – time index
- **L_x** (*float*) – scale in x dir
- **AminusC** – A-C coefficient

Returns

xc – x position of the center of the similarity solution pile

Return type

numpy array

ana1Tests.simiSolTest.defineEarthPressCoeff

`defineEarthPressCoeff`(*phi*, *delta*)

Define earth pressure coefficients

Parameters

- **phi** (*float*) – internal friction angle
- **delta** (*float*) – bed friction angle

Returns
earthPressureCoefficients – [Kxact Kxpass Kyact(Kxact) Kypass(Kxact) Kyact(Kxpass) Kypass(Kxpass)]

Return type
numpy array

ana1Tests.simiSolTest.getReleaseThickness

`getReleaseThickness`(*avaDir*, *cfg*, *demFile*)

Define release thickness for the similarity solution test

Release area is defined as an elipse or main radius Lx and Ly. Release thickness has a parabolic shape from relTh in the center to 0 on the edges

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **cfg** (*dict*) – confguration settings
- **demFile** (*str*) – path to DEM file

Returns
relDict – dictionary with info on release thickness distribution

Return type
dict

ana1Tests.simiSolTest.getSimiSolParameters

`getSimiSolParameters`(*solSimi*, *header*, *indTime*, *cfgSimi*, *Hini*, *gravAcc*)

get flow thickness, flow velocity and center location of flow mass of similarity solution for required time step

Parameters

- **solSimi** (*dict*) – similarity solution
- **header** (*dict*) – header dictionary with info about the extent and cell size
- **indTime** (*int*) – index for required time step in similarity solution
- **cfg** (*dict*) – configuration
- **Hini** (*float*) – initial release thickness
- **gravAcc** (*float*) – gravity acceleration

Returns
simiDict – dictionary of similiarty solution with flow thickness, flow velocity, and center location in x for required time step

Return type
dict

ana1Tests.simiSolTest.mainSimilaritySol

`mainSimilaritySol`(*simiSolCfg*)

Compute similarity solution :Parameters: **simiSolCfg** (*pathlib path*) – path to simiSol configuration file

Returns

solSimi –

similarity solution:
timeAdim: time array (without dimension) time: time array (with dimension) g_sol: g array g_p_sol: first derivativ of g array f_sol: f array f_p_sol: first derivativ of f array

Return type
dictionary

ana1Tests.simiSolTest.odeSolver

`odeSolver`(*solver*, *dt*, *tEnd*, *solSimi*)

Solve the ODE using a Runge-Kutta method

Parameters

- **solver** (*ode* instance) – *ode* instance corresponding to out ODE
- **dt** (*float*) – time step
- **tEnd** (*float*) – end time
- **solSimi** (*dictionary*) – similarity solution (for early times)
  - time: time array (without dimension)
  - g_sol: g array
  - g_p_sol: first derivativ of g array
  - f_sol: f array
  - f_p_sol: first derivativ of f array

Returns

solSimi – similarity solution (completed with all time steps)

- time: time array (without dimension)
- g_sol: g array
- g_p_sol: first derivativ of g array
- f_sol: f array
- f_p_sol: first derivativ of f array

Return type
dictionary

ana1Tests.simiSolTest.postProcessSimiSol

postProcessSimiSol(*avalancheDir*, *cfgMain*, *cfgSimi*, *simDF*, *solSimi*, *outDirTest*)

loop on all DFA simulations and compare then to the anlytic solution

Parameters

- **avalancheDir** (*str or pathlib path*) – avalanche directory
- **cfgMain** (*confiparser*) – avaframeCfg configuration
- **cfgSimi** (*dict*) – configuration for similarity sol
- **simDF** (*pandas dataFrame*) – configuration DF
- **solSimi** (*dict*) – similarity solution
- **outDirTest** (*pathlib path*) – path to output directory

Returns

simDF – configuration DF appended with the analysis results

Return type

pandas dataFrame

ana1Tests.simiSolTest.prepareParticlesFieldscom1DFA

prepareParticlesFieldscom1DFA(*fields*, *particles*, *header*, *simiDict*, *axis*)

get fields and particles dictionaries for given time step, for com1DFA domain origin is set to 0,0 for particles - so info on domain is required

Parameters

- **Fields** (*dictionary*) – fields dictionary
- **Particles** (*dictionary*) – particles dictionary
- **header** (*dict*) – header dictionary with info about the extend and cell size
- **simiDict** (*dict*) – dictionary with center location in x for similarity solution
- **axis** (*str*) – axis (x or y) for profile

Returns

com1DFASol – dictionary with location of particles, flow thickness, flow velocity, fields, and index for x or y cut of domain at the required time step

Return type

dict

ana1Tests.testUtilities

Functions for handling benchmark tests data, filtering, sorting

Functions

createDesDictTemplate	create an empty dictionary with all required test info keys
fetchBenchmarkResults	Fetch a list of all paths to result files in a benchmark test, default all result file paths, if resType list provided only for resTypes in list
filterBenchmarks	filter benchmarks according to characteristic
getTestAvaDirs	get a list of avalanche directories of tests
readAllBenchmarkDesDicts	get descritption dicts for all benchmark tests and add test name as key
readDesDictFromJson	read dict from json file
writeDesDicttoJson	create a json file with all required test info from desdict

ana1Tests.testUtilities.createDesDictTemplate

createDesDictTemplate()

create an empty dictionary with all required test info keys

ana1Tests.testUtilities.fetchBenchmarkResults

fetchBenchmarkResults(*testName*, *resTypes=[]*, *refDir=''*)

Fetch a list of all paths to result files in a benchmark test, default all result file paths, if resType list provided only for resTypes in list

Parameters

- **refDir** (*str*) – path to benchmark test directory
- **resTypes** (*list*) – list of all resTypes that shall be returned

Returns

refFiles – list of paths to all result files found for benchmark test and resType

Return type

list

ana1Tests.testUtilities.filterBenchmarks

filterBenchmarks(*testDictList*, *filterType*, *valuesList*, *condition='or'*)

filter benchmarks according to characteristic

Parameters

- **testDictList** (*list*) – list of benchmark dictionaries
- **filterType** (*str*) – key which is used for filtering (options: TAGS, TYPE)
- **valuesList** (*list*) – list of values used for filtering
- **condition** (*str*) – if multiple values given in valuesList - if 'or' then all test dictionaries are returned that have either of the values, if 'and' then only those are returned that feature both values

Returns

testList – list of all the benchmark dicionaries that meet filter criterion

Return type
list

ana1Tests.testUtilities.getTestAvaDirs

`getTestAvaDirs`(*testList*)

get a list of avalanche directories of tests

Parameters
testList (*list*) – list of test dictionaries

Returns
avaDirs – list of paths to avalanche directories

Return type
list

ana1Tests.testUtilities.readAllBenchmarkDesDicts

`readAllBenchmarkDesDicts`(*info=False*, *inDir=''*)

get descritption dicts for all benchmark tests and add test name as key

Parameters

- **info** (*bool*) – True if info shall be printed to log
- **inDir** (*pathLib object*) – path to benchmarks directory

Returns
testDictList – list of test info dictionaries

Return type
list

ana1Tests.testUtilities.readDesDictFromJson

`readDesDictFromJson`(*fileName*)

read dict from json file

Parameters
fileName (*str or pathlib path*) – path to json file

Returns
desDict – dictionary read from json file

Return type
dict

ana1Tests.testUtilities.writeDesDicttoJson

`writeDesDicttoJson`(*desDict*, *testName*, *outDir*)

create a json file with all required test info from desdict

Parameters

- **desDict** (*dict*) – description dictionary of test
- **testName** (*str*) – name of benchmark test
- **outDir** (*str or pathlib Path*) – path where json file is saved

Returns
fileName – path to json file

Return type
pathlib path

2.10.3.2 ana3AIMEC

Code for ana3AIMEC analysis module

Modules

ana3AIMEC.aimecTools	
ana3AIMEC.ana3AIMEC	AIMEC post processing workflow
ana3AIMEC.dfa2Aimec	

ana3AIMEC.ana3AIMEC

AIMEC post processing workflow

Functions

addSLToParticles	add aimec (thalweg) s,l coordinates to particle dicts use dem from sim corresponding to particle dicts by providing demFileName if demFileName='' standard dem in avaDir/Inputs is used
aimecRes2ReportDict	Gather aimec results and append them to report the dictionary
aimecTransform	Adding s projected and l projected in thalweg/aimec coordinate system to the particle dictionary
computeSLParticles	find the closest s, l coordinates in the aimec/thalweg coordinate system to a particles x,y location
fullAimecAnalysis	fetch all data required to run aimec analysis, setup pathDict and reference sim, read the inputs, perform checks if all required data is available, run main aimec
mainAIMEC	Main logic for AIMEC postprocessing
postProcessAIMEC	Apply domain transformation and analyse result data (for example pressure, thickness, velocity...)

ana3AIMEC.ana3AIMEC.addSLToParticles

addSLToParticles(*avaDir*, *cfgAimec*, *demFileName*, *particlesList*, *saveToPickle=False*)

add aimec (thalweg) s,l coordinates to particle dicts use dem from sim corresponding to particle dicts by providing demFileName if demFileName=’’ standard dem in avaDir/Inputs is used

Parameters

- **avaDir** (*pathlib path*) – path to avalanche directory
- **cfgAimec** (*configparser object*) – configuration settings of aimec
- **demFileName** (*str*) – path including fileName and extension to demFile located in avaDir/Inputs
- **particlesList** (*list*) – list of particles dicts of sim
- **saveToPickle** (*bool*) – if updated particle dicts shall be saved to pickle

Returns

- **particlesList** (*list*) – list of particles dicts updated with s,l
- **rasterTransfo** (*dict*) – dictionary with info on rasterTransformation

ana3AIMEC.ana3AIMEC.aimecRes2ReportDict

aimecRes2ReportDict(*resAnalysisDF*, *reportD*, *benchD*, *pathDict*)

Gather aimec results and append them to report the dictionary

Parameters

- **resAnalysisDF** (*dataFrame*) – dataframe with aimec analysis results
- **reportD** (*dict*) – report dictionary to be updated
- **benchD** (*dict*) – benchmark dictionary to be updated

- **pathDict** (*dict*) – dictionary with refSimRowHash

Returns

- **reportD** (*dict*) – report dictionary updated with the 'Aimec analysis' section
- **benchD** (*dict*) – benchmark dictionary updated with the 'Aimec analysis' section

ana3AIMEC.ana3AIMEC.aimecTransform

`aimecTransform`(*rasterTransfo*, *particle*, *demHeader*, *timeSeries=False*)

Adding s projected and l projected in thalweg/aimec coordinate system to the particle dictionary

Parameters

- **rasterTransfo** (*dict*) – domain transformation information
- **particle** (*dict*) – dictionary with particle properties, particles x,y coordinate origin is 0,0
- **demHeader** (*dict*) – dict with info on dem cellSize, xllcenter, ..
- **timeSeries** (*bool*) – if timeSeries consider that particles dict is provided as time series if False particles dict is provided for one time step only

Returns
particle – particle dictionary with s grid and l grid added

Return type
dict

ana3AIMEC.ana3AIMEC.computeSLParticles

`computeSLParticles`(*rasterTransfo*, *demHeader*, *particlesX*, *particlesY*)

find the closest s, l coordinates in the aimec/thalweg coordinate system to a particles x,y location

Parameters

- **rasterTransfo** (*dict*) – info on rasterTransformation, here gridx, gridy coordinates
- **demHeader** (*dict*) – dict with info on dem xllcenter, yllcenter
- **particlesX** (*np array*) – x coordinates of particles location for one time step but all particles (origin 0,0)
- **particlesY** (*np array*) – Y coordinates of particles location for one time step but all particles (origin 0,0)

Returns
sList, lList – list of s, l coordinates for respective particle

Return type
list

ana3AIMEC.ana3AIMEC.fullAimecAnalysis

`fullAimecAnalysis`(*avalancheDir*, *cfg*, *inputDir=''*, *demFileName=''*)

fetch all data required to run aimec analysis, setup pathDict and reference sim, read the inputs, perform checks if all required data is available, run main aimec

Parameters

- **avalancheDir** (*str or pathlib path*) – path to avalanche directory
- **cfg** (*configparser object*) – main aimec configuration settings
- **inputDir** (*str or pathlib path*) – optional- directory where peak files are located, if '', avaDir/Outputs/comMod/peakFiles is set
- **demFileName** (*pathlib path*) – path to dem file

Returns

- **rasterTransfo** (*dict*) – domain transformation information
- **resAnalysisDF** (*dataFrame*) – results of ana3AIMEC analysis
- **plotDict** (*dict*) – dictionary for report
- **newRasters** (*dict*) – dictionary containing pressure, velocity and flow thickness rasters after transformation for the reference and the current simulation

ana3AIMEC.ana3AIMEC.mainAIMEC

`mainAIMEC`(*pathDict*, *inputsDF*, *cfg*)

Main logic for AIMEC postprocessing

Reads the required files location for ana3AIMEC postprocessing given an avalanche directory Make the domain transformation corresponding to the input avalanche path Transform 2D fields (pressure, flow thickness ...) Analyze transformed rasters and mass Produce plots and report

Parameters

- **pathDict** (*dict*) – dictionary with paths to dem and lines for Aimec analysis
- **inputsDF** (*dataFrame*) – dataframe with simulations to analyze and associated path to raster data
- **cfg** (*configparser*) – configparser with ana3AIMEC settings defined in ana3AIMECCfg.ini

Returns

- **rasterTransfo** (*dict*) – domain transformation information
- **resAnalysisDF** (*dataFrame*) – results of ana3AIMEC analysis
- **plotDict** (*dict*) – dictionary for report

ana3AIMEC.ana3AIMEC.postProcessAIMEC

postProcessAIMEC(*cfg*, *rasterTransfo*, *pathDict*, *resAnalysisDF*, *newRasters*, *timeMass*, *simRowHash*, *contourDict*)

Apply domain transformation and analyse result data (for example pressure, thickness, velocity…)

Apply the domain tranformation to peak results Analyse them. Calculate runout, Max Peak Values, Average Peak Values… Get mass and entrainment

The output resAnalysisDF contains:

-maxACrossMax: float
max max A (A depends on what is in resTypes)

-ACrossMax: 1D numpy array
max A in each cross section (A depends on what is in resTypes)

-ACrossMean: 1D numpy array
mean A in each cross section (A depends on what is in resTypes)

-xRunout: float
x coord of the runout point calculated from the MAX peak result in each cross section (runoutResType provided in the ini file)

-yRunout: float
y coord of the runout point calculated from the MAX peak result in each cross section (runoutResType provided in the ini file)

-sRunout: float
projected runout distance calculated from the MAX peak result in each cross section (runoutResType provided in the ini file)

-xMeanRunout: float
x coord of the runout point calculated from the MEAN peak result in each cross section (runoutResType provided in the ini file)

-yMeanRunout: float
y coord of the runout point calculated from the MEAN peak result in each cross section (runoutResType provided in the ini file)

-sMeanRunout: float
projected runout distance calculated from the MEAN peak result in each cross section (runoutResType provided in the ini file)

-elevRel: float
elevation of the release area (based on first point with peak field > thresholdValue)

-deltaH: float
elevation fall difference between elevRel and altitude of run-out point

-TP: float
ref = True sim2 = True

-FN: float
ref = False sim2 = True

-FP: float
ref = True sim2 = False

-TN: float
ref = False sim2 = False

-resTypeFieldMax: float
max value of peak field resType raster (original raster read from result file (no coordinate transformation)

-resTypeFieldMin: float
min value of peak field resType raster (original raster read from result file (no coordinate transformation) 0 values are masked

-resTypeFieldMean: float
mean value of peak field resType raster (original raster read from result file (no coordinate transformation) 0 values are masked

-resTypeFieldStd: float
std value of peak field resType raster (original raster read from result file (no coordinate transformation) 0 values are masked

if mass analysis is performed

-relMass: float
release mass

-entMass: float
entrained mass

-finalMass: float
final mass

-relativMassDiff: float
the final mass diff with ref (in %)

-growthIndex: float
growth index

-growthGrad: float
growth gradient

Parameters

- **cfg** (*configParser objec*) – parameters for AIMEC analysis
- **rasterTransfo** (*dict*) – transformation information
- **pathDict** (*dict*) – dictionary with paths to dem and lines for Aimec analysis
- **resAnalysisDF** (*dataFrame*) – dataframe with simulations to analyze and associated path to raster data
- **newRasters** (*dict*) – dictionary containing pressure, velocity and flow thickness rasters after transformation for the reference and the current simulation
- **timeMass** (*1D numpy array*) – time array for mass analysis (if flagMass=True, otherwise None)
- **simRowHash** (*str*) – dataframe hash of the current simulation to analyze
- **contourDict** (*dict*) – dictionary with one key per sim and its x, y coordinates for contour line of runoutresType for thresholdValue

Returns

- **resAnalysisDF** (*dataFrame*) – input DF with results from Aimec Analysis updated with results from current simulation
- **contourDict** (*dict*) – dictionary with one key per sim and its x, y coordinates for contour line of runoutresType for thresholdValue - updated with info for current simulation

2.10.3.3 ana4Stats

Tools for statistical analysis of simulation results

Modules

ana4Stats.getStats	Function to determine statistics of datasets
ana4Stats.probAna	This is a simple function for computing a probability map of all peak files of one parameter that exceed a particular threshold

ana4Stats.getStats

Function to determine statistics of datasets

Functions

extractMaxValues	Extract max values of result parameters and save to dictionary

ana4Stats.getStats.extractMaxValues

extractMaxValues(*inputDir*, *avaDir*, *varPar*, *restrictType=''*, *nameScenario=''*, *parametersDict=''*)

Extract max values of result parameters and save to dictionary

- optionally restrict data of peak fields by defining which result parameter with restrictType, provide nameScenario and a parametersDict to filter simulations

Parameters

- **inputDir** (*str*) – path to directoy where peak files can be found
- **avaDir** (*str*) – path to avalanche directoy
- **varPar** (*str*) – parameter that has been varied when performing simulations (for example relTh)
- **restrictType** (*str*) – optional -result type of result parameters that should be used to mask result fields (eg. ppr, pft, ..)
- **nameScenario** (*str*) – optional -parameter that shall be used for color coding of simulation results in plots (for example releaseScenario)
- **parametersDict** (*dict*) – optional -dictionary with parameter and parameter values to filter simulations

Returns

peakValues – dictionary that contains max values for all result parameters for each simulation

Return type

dict

ana4Stats.probAna

This is a simple function for computing a probability map of all peak files of one parameter that exceed a particular threshold

Functions

cfgFilesGlobalApproach	create configuration files with all parameter variations - drawn from full sample for performing sims with modName comModule
cfgFilesLocalApproach	create configuration file for performing sims with modName com module
checkForNumberOfReferenceValues	check if in reference configuration no variation option of varPar is set if set - throw error
checkParameterSettings	check if parameter settings in comMod configuration do not inlcude variation for parameters to be varied
createCfgFiles	create all config files required to run com Module from parameter variations using paramValues
createComModConfig	create configuration file for performing sims with modName com module
createSample	create a sample of parameters
createSampleFromConfig	Create a sample of parameters for a desired parameter variation, and draw nSample sets of parameter values if thickness values read from shp for comMod, convert sample values for these
createSampleWithVariationForThParameters	Create a sample of parameters for a desired parameter variation, and fetch thickness values from shp file and perform variation for each feature within shapefile but treating the features of one shapefile as not-independent
createSampleWithVariationStandardParameters	create a sample for a parameter variation using latin hypercube sampling
fetchProbConfigs	fetch configurations of prob run in order to filter simulations e.g.
fetchStartCfg	fetch start configuration of comMod if onlyDefault use default comModCfg.ini and if false check if there is a local_comModCfg.ini
fetchThThicknessLists	fetch the desired thickness shp file info on thickness, id and ci values of all available features in shp file
fetchThicknessInfo	Fetch input data for avaDir and thickness info
makeDictFromVars	create a dictionary with info on parameter variation for all parameter in varParList
probAnalysis	Compute propability map of a given set of simulation result exceeding a particular threshold and save to outDir
updateCfgRange	update cfg with a range for parameters in cfgProb

ana4Stats.probAna.cfgFilesGlobalApproach

`cfgFilesGlobalApproach`(*avaDir*, *cfgProb*, *modName*, *outDir*)

create configuration files with all parameter variations - drawn from full sample for performing sims with modName comModule

Parameters

- **cfgProb** (*configParser object*) – configuration settings
- **avaDir** (*pathlib path*) – path to avalanche directory
- **modName** (*module*) – computational module

Returns

cfgFiles – list of paths to newly generated configuration files for com module inlcuding parameter variations

Return type

list

ana4Stats.probAna.cfgFilesLocalApproach

`cfgFilesLocalApproach`(*variationsDict*, *cfgProb*, *modName*, *outDir*)

create configuration file for performing sims with modName com module

Parameters

- **variationsDict** (*dict*) – dictionary with for each varName, varVariation, varSteps, and type of variation
- **cfgProb** (*configParser object*) – configuration settings
- **avaDir** (*pathlib path*) – path to avalanche directory
- **modName** (*module*) – computational module

Returns

cfgFiles – dictionary of paths to newly generated configuration files for com module for all parameters

Return type

dict

ana4Stats.probAna.checkForNumberOfReferenceValues

`checkForNumberOfReferenceValues`(*cfgGen*, *varPar*)

check if in reference configuration no variation option of varPar is set if set - throw error

Parameters

- **cfgGen** (*configparser object*) – reference configuration settings
- **varPar** (*str*) – name of parameter to be checked

ana4Stats.probAna.checkParameterSettings

checkParameterSettings(*cfg*, *varParList*)

check if parameter settings in comMod configuration do not inlcude variation for parameters to be varied

Parameters

- **cfg** (*configparser object*) – configuration settings
- **varParList** (*list*) – list of parameters (names) that shall be varied

ana4Stats.probAna.createCfgFiles

createCfgFiles(*paramValuesDList*, *comMod*, *cfg*, *cfgPath=''*)

create all config files required to run com Module from parameter variations using paramValues

Parameters

- **paramValuesDList** (*list*) – list of dictionaries with parameter names and values (array of all sets of parameter values, one row per value set) multiple dictionaries if multiple release area scenarios and thFromShp
- **comMod** (*com module*) – computational module
- **cfg** (*configparser object*) – configuration settings
- **cfgPath** (*str*) – path where cfg files should be saved to

Returns

cfgFiles – list of cfg file paths for comMod including the updated values of the parameters to vary

Return type

list

ana4Stats.probAna.createComModConfig

createComModConfig(*cfgProb*, *avaDir*, *modName*)

create configuration file for performing sims with modName com module

Parameters

- **cfgProb** (*configParser object*) – configuration settings
- **avaDir** (*pathlib path*) – path to avalanche directory
- **modName** (*module*) – computational module

Returns

cfgFiles – list of paths to newly generated configuration files for com module inlcuding parameter variations

Return type

list

ana4Stats.probAna.createSample

`createSample`(*cfgProb*, *varParList*)

create a sample of parameters

Parameters

- **cfgProb** (*configparser object*) – configuration settings
- **varParList** (*list*) – list of parameters used for creating a sample

Returns

sample – sample object of given dimension that can be adjusted to desired bounds

Return type

scipy object

ana4Stats.probAna.createSampleFromConfig

`createSampleFromConfig`(*avaDir*, *cfgProb*, *comMod*)

Create a sample of parameters for a desired parameter variation, and draw nSample sets of parameter values if thickness values read from shp for comMod, convert sample values for these

Parameters

- **avaDir** (*pathlib path*) – path to avalanche directory
- **cfgProb** (*configparser object*) – configuration settings for parameter variation
- **comMod** (*computational module*) – module to perform then sims for parameter variation

Returns

paramValuesDList – list of paramValuesD (multiple if multiple release area scenarios)

- names: list, list of parameter names (that are varied)
- values: numpy nd array, as many rows as sets of parameter values and as many rows as parameters
- typeList: list, list of types of parameters (float, ...)
- thFromIni: str, str of parameter names where the base value is read from shape

Return type

list

ana4Stats.probAna.createSampleWithVariationForThParameters

`createSampleWithVariationForThParameters`(*avaDir*, *cfgProb*, *cfgStart*, *varParList*, *valVariationValue*, *varType*, *thReadFromShp*)

Create a sample of parameters for a desired parameter variation, and fetch thickness values from shp file and perform variation for each feature within shapefile but treating the features of one shapefile as not-independent

paramsValuesD dict in output list contains

Parameters

- **cfgProb** (*configparser object*) – configuration settings for parameter variation
- **cfgStart** (*configparser object*) – configuration settings for comMod without variation values

- **varParList** (*list*) – list of parameters that shall be varied
- **valVariationValue** (*list*) – list if value used for variation
- **varType** (*list*) – list of type of variation for each parameter (percent, range, rangefromci)

Returns

paramValuesDList – list of paramValuesD (multiple if multiple release area scenarios)

- names: list, list of parameter names (that are varied)
- values: numpy nd array, as many rows as sets of parameter values and as many rows as parameters
- typeList: list, list of types of parameters (float, ...)
- thFromIni: str, str of parameter names where the base value is read from shape

Return type
list

ana4Stats.probAna.createSampleWithVariationStandardParameters

`createSampleWithVariationStandardParameters`(*cfgProb*, *cfgStart*, *varParList*, *valVariationValue*, *varType*)

create a sample for a parameter variation using latin hypercube sampling

Parameters

- **cfgProb** (*configparser object*) – configuration settings for parameter variation
- **cfgStart** (*configparser object*) – configuration settings for comMod without variation values
- **varParList** (*list*) – list of parameters that shall be varied
- **valVariationValue** (*list*) – list if value used for variation
- **varType** (*list*) – list of type of variation for each parameter (percent, range, rangefromci)

Returns

paramValuesD – dictionary used to pass parameter variation values

- names: list, list of parameter names (that are varied)
- values: numpy nd array, as many rows as sets of parameter values and as many rows as parameters
- typeList: list, list of types of parameters (float, ...)
- thFromIni: str, str of parameter names where the base value is read from shape

Return type
dict

ana4Stats.probAna.fetchProbConfigs

`fetchProbConfigs`(*cfg*)

fetch configurations of prob run in order to filter simulations e.g. to create probability maps for different scenarios

Parameters
cfg (*configparser object*) – configuration setting, here used: samplingStrategy, varParList

Returns
probConfigs – dictionary with one key per config and a dict per key with parameter and value

Return type
dict

ana4Stats.probAna.fetchStartCfg

`fetchStartCfg`(*comMod*, *cfgProb*)

fetch start configuration of comMod if onlyDefault use default comModCfg.ini and if false check if there is a local_comModCfg.ini

Parameters

- **comMod** (*computational module*) – module where configuration is read from
- **cfgProb** (*configparser object*) – configuration settings of probAna with comMod_override section

Returns
cfgStart – configuration object of comMod

Return type
configparser object

ana4Stats.probAna.fetchThThicknessLists

`fetchThThicknessLists`(*varPar*, *inputSimFiles*, *releaseFile*, *ciRequired=False*)

fetch the desired thickness shp file info on thickness, id and ci values of all available features in shp file

Parameters

- **varPar** (*str*) – name of thickness parameter
- **inputSimFiles** (*dict*) – dictionary with info in input data
- **ciRequired** (*bool*) – if True throw error if ci Values not provided

Returns

- **thicknessFeatureNames** (*list*) – list of names of thickness features
- **thValues** (*list*) – list of thickness values for all features
- **ciValues** (*list*) – list of ci values for all feature

ana4Stats.probAna.fetchThicknessInfo

`fetchThicknessInfo`(*avaDir*)

Fetch input data for avaDir and thickness info

Parameters

- **cfg** (*configparser object*) – configuration settings
- **avaDir** (*pathlib path or str*) – path to avalanche directory

Returns

inputSimFilesAll – dictionary with info on available input data (release areas, entrainment, and thickness info)

Return type

dict

ana4Stats.probAna.makeDictFromVars

`makeDictFromVars`(*cfg*)

create a dictionary with info on parameter variation for all parameter in varParList

Parameters

cfg (*configparser object*) – configuration settings, here varParList, variationValue, numberOfSteps

Returns

variationsDict – dictionary with for each varName, varVariation, varSteps, and type of variation

Return type

dict

ana4Stats.probAna.probAnalysis

`probAnalysis`(*avaDir*, *cfg*, *module*, *parametersDict=''*, *inputDir=''*, *probConf=''*, *simDFActual=''*)

Compute propability map of a given set of simulation result exceeding a particular threshold and save to outDir

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **cfg** (*dict*) – configuration read from ini file of probAna function
- **module** – computational module that was used to run the simulations
- **parametersDict** (*dict*) – dictionary with simulation parameters to filter simulations
- **inputDir** (*str*) – optional - path to directory where data that should be analysed can be found in a subfolder called peakFiles and configurationFiles, required if not in module results
- **probConf** (*str*) – name of probability configuration
- **simDFActual** (*pandas dataFrame*) – dataframe of simulation configurations that shall be used for prob analysis

ana4Stats.probAna.updateCfgRange

updateCfgRange(*cfg*, *cfgProb*, *varName*, *varDict*)

update cfg with a range for parameters in cfgProb

Parameters

- **cfg** (*configparser object*) – configuration object to update
- **cfgProb** (*configParser object*) – configparser object with info on update
- **varName** (*str*) – name of parameter used for variation
- **varDict** (*dict*) – dictionary with variationValue and numberOfSteps for varName

Returns

cfg – updated configuration object

Return type

configParser

2.10.3.4 ana5Utils

Generate thalweg-time-diagrams and simulograms

Modules

ana5Utils.DFAPathGeneration	Tools for generating an avalanche path from a DFA simulation
ana5Utils.distanceTimeAnalysis	functions to convert to a different coordinate system and produce a range-time diagram from simulation results options: 1) range-time diagram from radar's field of view capped to this 2) thalweg-time diagram from beta point of view along avalanche path

ana5Utils.DFAPathGeneration

Tools for generating an avalanche path from a DFA simulation

Functions

appendAverageStd	append averaged to path
extendDFAPath	extend the DFA path at the top and bottom avaProfile with x, y, z, s information
extendProfileBottom	extend the DFA path at the bottom (runout area)
extendProfileTop	extend the DFA path at the top (release)
generateAveragePath	extract path from fileds or particles
getDFAPathFromField	compute mass averaged path from fields
getDFAPathFromPart	compute mass averaged path from particles
getParabolicFit	fit a parabola on a set of (s, z) points
getSplitPoint	find the split point corresponding to an avalanche profile, with parabolic fit and the slopeSplitPoint
resamplePath	resample the path profile and keep track of indStartMassAverage and indEndMassAverage
saveSplitAndPath	Save avaPath and split point to directory
weightedAvgAndStd	Return the weighted average and standard deviation.

ana5Utils.DFAPathGeneration.appendAverageStd

appendAverageStd(*propList*, *avaProfile*, *particles*, *weights*, *naming=''*)

append averaged to path

Parameters

- **propList** (*list*) – list of properties to average and append
- **avaProfile** (*dict*) – path
- **particles** (*dict*) – particles dict
- **weights** (*numpy array*) – array of weights (same size as particles)
- **naming** (*list*) – optional - list of properties at source (if different than final naming in avaProfile)

Returns

avaProfile – averaged profile

Return type

dict

ana5Utils.DFAPathGeneration.extendDFAPath

extendDFAPath(*cfg*, *avaProfile*, *dem*, *particlesIni*)

extend the DFA path at the top and bottom avaProfile with x, y, z, s information

Parameters

- **cfg** (*configParser*) – configuration object with:
  - extTopOption: int, how to extend towards the top? 0 for heighst point method, a for largest runout method
  - nCellsResample: int, resampling length is given by nCellsResample*demCellSize
  - nCellsMinExtend: int, when extending towards the bottom, take points at more

than nCellsMinExtend*demCellSize from last point to get the direction - nCellsMaxExtend: int, when extending towards the bottom, take points at less than nCellsMaxExtend*demCellSize from last point to get the direction - factBottomExt: float, extend the profile from factBottomExt*sMax

- **avaProfile** (*dict*) – profile to be extended
- **dem** (*dict*) – dem dict
- **particlesIni** (*dict*) – initial particles dict

Returns
avaProfile – extended profile at top and bottom (x, y, z).

Return type
dict

ana5Utils.DFAPathGeneration.extendProfileBottom

`extendProfileBottom`(*cfg*, *dem*, *profile*)

extend the DFA path at the bottom (runout area)

Find the direction in which to extend considering the last point of the profile and a few previous ones but discarding the ones that are too close (nCellsMinExtend* csz < distFromLast <= nCellsMaxExtend * csz). Extend in this diretion for a distance factBottomExt * length of the path.

Parameters

- **cfg** (*configParser*) – nCellsMinExtend: int, when extending towards the bottom, take points at more than nCellsMinExtend*demCellSize from last point to get the direction

  nCellsMaxExtend: int, when extending towards the bottom, take points at less than nCellsMaxExtend*demCellSize from last point to get the direction factBottomExt: float, extend the profile from factBottomExt*sMax
- **dem** (*dict*) – dem dict
- **profile** (*dict*) – profile to extend

Returns
profile – extended profile

Return type
dict

ana5Utils.DFAPathGeneration.extendProfileTop

`extendProfileTop`(*extTopOption*, *particlesIni*, *profile*)

extend the DFA path at the top (release)

Either towards the highest point in particlesIni (extTopOption = 0) or the point leading to the longest runout (extTopOption = 1)

Parameters

- **extTopOption** (*int*) – decide how to extend towards the top if 0, extrapolate towards the highest point in the release if 1, extrapolate towards the point leading to the lonest runout
- **particlesIni** (*dict*) – initial particles dict

- **profile** (*dict*) – profile to extend

Returns
profile – extended profile

Return type
dict

ana5Utils.DFAPathGeneration.generateAveragePath

generateAveragePath(*avalancheDir*, *pathFromPart*, *simName*, *dem*, *addVelocityInfo=False*, *flagAvaDir=True*, *comModule='com1DFA'*)

extract path from fileds or particles

Parameters

- **avalancheDir** (*pathlib*) – avalanche directory pathlib path
- **pathFromPart** (*boolean*) – compute path from particles if True, from fields (FT, FM, FV) if False
- **simName** (*str*) – simulation name
- **dem** (*dict*) – com1DFA simulation dictionary
- **addVelocityInfo** (*boolean*) – True to add (u2, ekin, totEKin) to result Will only work if the particles (ux, uy, uz) exist or if 'FV' exists
- **flagAvaDir** (*bool*) – if True avalancheDir corresponds to an avalanche directory and data is read from avaDir/Outputs/comModule/particles or avaDir/Outputs/comModule/peakFiles depending on if pathFromPart is True or False
- **comModule** (*str*) – module that computed the particles or fields

Returns

- **avaProfileMass** (*dict*) – mass averaged profile (x, y, z, 's') if addVelocityInfo is True, kinetic energy and velocity information are added to the avaProfileMass dict (u2, ekin, totEKin)
- **particlesIni** (*dict*) – x, y coord of the initial particles or flow thickness field

ana5Utils.DFAPathGeneration.getDFAPathFromField

getDFAPathFromField(*fieldsList*, *fieldHeader*, *dem*)

compute mass averaged path from fields

Also returns the averaged velocity and kinetic energy associated The dem and fieldsList (FT, FV and FM) need to have identical dimentions and cell size. If FV is not provided, information about velocity and kinetic energy is not computed

Parameters

- **fieldsList** (*list*) – time sorted list of fields dict
- **fieldHeader** (*dict*) – field header dict
- **dem** (*dict*) – dem dict

Returns
avaProfileMass – mass averaged profile (x, y, z, 's') if 'FV' in fieldsList, kinetic energy and velocity information are added to the avaProfileMass dict (u2, ekin, totEKin)

Return type
dict

ana5Utils.DFAPathGeneration.getDFAPathFromPart

`getDFAPathFromPart`(*particlesList*, *addVelocityInfo=False*)

compute mass averaged path from particles

Also returns the averaged velocity and kinetic energy associated If addVelocityInfo is True, information about velocity and kinetic energy is computed

Parameters

- **particlesList** (*list*) – list of particles dict
- **addVelocityInfo** (*boolean*) – True to add (u2, ekin, totEKin) to result

Returns
avaProfileMass – mass averaged profile (x, y, z, 's', 'sCor') if addVelocityInfo is True, kinetic energy and velocity information are added to the avaProfileMass dict (u2, ekin, totEKin)

Return type
dict

ana5Utils.DFAPathGeneration.getParabolicFit

`getParabolicFit`(*cfg*, *avaProfile*, *dem*)

fit a parabola on a set of (s, z) points

first and last point match. Last constraint is given by fitOption (see below)

Parameters

- **cfg** (*configParser*) – configuration object with: fitOption: int how to optimize the fit 0 minimize distance between points 1 match the end slope
- **avaProfile** (*dict*) – profile to be extended
- **dem** (*dict*) – dem dict

Returns
parabolicFit – a, b, c coefficients of the parabolic fit (y = a*a*x + b*x + c)

Return type
dict

ana5Utils.DFAPathGeneration.getSplitPoint

`getSplitPoint`(*cfg*, *avaProfile*, *parabolicFit*)

find the split point corresponding to an avalanche profile, with parabolic fit and the slopeSplitPoint

Parameters

- **cfg** (*configParser*) – configuration object with
  - slopeSplitPoint: float, desired slope at split point (in degrees)
  - dsMin: float, threshold distance [m] for looking for the

- split point (at least dsMin meters below split point angle)

- **avaProfile** (*dict*) – profile to be extended
- **parabolicFit** (*dict*) – a, b, c coeficients of the parabolic fit (y = a*a*x + b*x + c)

Returns
splitPoint – (x, y, z, zPra, s) at split point location.

Return type
dict

ana5Utils.DFAPathGeneration.resamplePath

`resamplePath`(*cfg*, *dem*, *avaProfile*)

resample the path profile and keep track of indStartMassAverage and indEndMassAverage

Parameters

- **cfg** (*configParser*) – configuration object with
  - nCellsResample: int, resampling length is given by nCellsResample*demCellSize
- **dem** (*dict*) – dem dict
- **avaProfile** (*dict*) – profile to be resampled

Returns
avaProfile – resampled path profile

Return type
dict

ana5Utils.DFAPathGeneration.saveSplitAndPath

`saveSplitAndPath`(*avalancheDir*, *simDFrow*, *splitPoint*, *avaProfileMass*, *dem*)

Save avaPath and split point to directory

Parameters

- **avalancheDir** (*pathlib*) – avalanche directory pathlib path
- **simDFrow** (*pandas series*) – row of the siulation dataframe coresponding to the current simulation analyzed
- **splitPoint** (*dict*) – point dictionary
- **avaProfileMass** (*dict*) – line dictionary
- **dem** (*dict*) – com1DFA simulation dictionary

Returns
avaProfile – resampled path profile

Return type
dict

ana5Utils.DFAPathGeneration.weightedAvgAndStd

weightedAvgAndStd(*values*, *weights*)

Return the weighted average and standard deviation.

values, weights – Numpy ndarrays with the same shape.

ana5Utils.distanceTimeAnalysis

functions to convert to a different coordinate system and produce a range-time diagram from simulation results options: 1) range-time diagram from radar's field of view capped to this 2) thalweg-time diagram from beta point of view along avalanche path

Functions

approachVelocity	compute maximal approach velocity based on front location (range) and time step
createThalwegTimeInfoFromSimResults	top level function to create mtiInfo dict from com Module simulation results required are flow variable fields for different time steps
exportData	save all info about range, time steps, average values to pickle for producing plots
extractFrontAndMeanValuesRadar	extract average values of flow parameter result at certain distance intervals (range gates) add info to be used for colorcoding range-time diagram
extractFrontAndMeanValuesTT	extract avalanche front and mean values of flow parameter result field used for colorcoding range-time diagram
fetchRangeTimeInfo	determine avalanche front and average values of flow parameter along path update mtiInfo dictionary with this information
fetchTimeStepFromName	split path name to fetch info on time step
getRadarLocation	fetch radar location and direction of of field of view coordinates
importMTIData	import mtiInfo data for creating range time plots, if no inputDir is provided, pickles are fetched from avaDir/Outputs/modName/distanceTimeAnalysis multiple pickles allowed- these carry also configuration info for distanceTimeAnalysis
initializeRangeTime	initialize generation of range-time diagram for visualizing simulation data
maskRangeFull	mask range (already masked with radar field of view) also with avalanche result field where NOT above threshold
radarMask	function to create masked array of radar range using dem and radar's field of view
setDemOrigin	reset original origin to simulation DEM - required if for ava sim is set to something different
setupRangeTimeDiagram	create setup for creating range-time diagrams from avalanche simulation results with respect to a radar's field of view
setupThalwegTimeDiagram	initialize parameters and prerequisites for generating thalweg-time diagrams

ana5Utils.distanceTimeAnalysis.approachVelocity

approachVelocity(*mtiInfo*)

compute maximal approach velocity based on front location (range) and time step

- cleans nan in range
- neglects anything behind maximal runout
- neglects non-unique range values, e.g. the front did not move
- **cleans approach velocity: velocity in cell under test can not be higher**
  than the double of the mean from the surrounding cells

Parameters
mtiInfo (*dict*) – info on distance to front and time steps

Returns

- **maxVel** (*float*) – max value of approach velocity
- **rangeVel** (*float*) – range of max value of approach velocity
- **timeVel** (*float*) – time of max value of approach velocity

ana5Utils.distanceTimeAnalysis.createThalwegTimeInfoFromSimResults

`createThalwegTimeInfoFromSimResults`(*avalancheDir*, *cfgRangeTime*, *modName*, *index*, *simDF*, *dem*)

top level function to create mtiInfo dict from com Module simulation results required are flow variable fields for different time steps

Parameters

- **avalancheDir** (*str or pathlib path*) – path to avalanche directory
- **cfgRangeTime** (*configparser object*) – configuration settings for range time diagram
- **modName** (*str*) – name of computational module that has been used to create ava sim
- **index** (*str*) – index of current sim in simDF
- **simDF** (*dataFrame*) – dataframe with info on sim configuration
- **dem** (*dict*) – dictionary with simulation dem

Returns

- **cfgRangeTime** (*configparser object*) – configuration settings for range time
- **mtiInfo** (*dict*) – dictionary with info for creating range time plots

ana5Utils.distanceTimeAnalysis.exportData

`exportData`(*mtiInfo*, *cfgRangeTime*, *modName*)

save all info about range, time steps, average values to pickle for producing plots

Parameters

- **mtiInfo** (*dict*) – dictionary with rangeList, timeList, mti (average values along range gates or cross profiles)
- **cfgRangeTime** (*configparser*) – configuration settings of distance time
- **modName** (*str*) – name of computational module

ana5Utils.distanceTimeAnalysis.extractFrontAndMeanValuesRadar

`extractFrontAndMeanValuesRadar`(*cfgRangeTime*, *flowF*, *mtiInfo*)

extract average values of flow parameter result at certain distance intervals (range gates) add info to be used for colorcoding range-time diagram

Parameters

- **cfgRangeTime** (*configparser object*) – configuration settings for creating range-time diagram
- **flowF** (*numpy array*) – flow parameter result field
- **mtiInfo** (*dict*) – info here used: rArray, bmaskRadar rangeMasked, rangeGates, mti, timeList

Returns

mtiInfo – updated mtiInfo dict with info on mti values

Return type

dict

ana5Utils.distanceTimeAnalysis.extractFrontAndMeanValuesTT

`extractFrontAndMeanValuesTT`(*cfgRangeTime*, *flowF*, *demHeader*, *mtiInfo*)

extract avalanche front and mean values of flow parameter result field used for colorcoding range-time diagram

Parameters

- **cfgRangeTime** (*configparser object*) – configuration settings
- **flowF** (*numpy nd array*) – flow parameter result field
- **demHeader** (*dict*) – dictionary with info on DEM header used for locating fields
- **mtiInfo** (*dict*) – dictionary with arrays, lists for range-time diagram: mti for colorcoding (average values of range gates/ crossprofiles) timeList time step info rangeList range info rangeGates info on cross profile locations rangeRaster - info on distance from beta point in path following coordinate system rasterTransfo - information on domain transformation to path following coordinate system

Returns

mtiInfo – updated dictionary with info on average value along crossprofiles (mti) and distance to front from reference point along path (rangeList)

Return type

dict

ana5Utils.distanceTimeAnalysis.fetchRangeTimeInfo

`fetchRangeTimeInfo`(*cfgRangeTime*, *cfg*, *dtRangeTime*, *t*, *demHeader*, *fields*, *mtiInfo*)

determine avalanche front and average values of flow parameter along path update mtiInfo dictionary with this information

Parameters

- **cfgRangeTime** (*configparser*) – configuration settings for range-time diagram
- **cfg** (*configParser object*) – configuration settings of computational module

- **dtRangeTime** (*list*) – list of time steps where avalanche front shall be exported
- **t** (*float*) – actual simulation time step
- **demHeader** (*dict*) – dictionary with DEM header
- **fields** (*dict*) – dictionary with flow parameter result fields
- **mtiInfo** (*dict*) – info on time steps (timeList) and distance (rangeList) of avalanche front averaged values of result parameter (mti) for each range gate (rangeGates) for colorcoding optional info on masks for radar field of view (rangeMasked), etc.

Returns

- **mtiInfo** (*dict*) – updated dictionary
- **dtRangeTime** (*list*) – updated list of time steps

ana5Utils.distanceTimeAnalysis.fetchTimeStepFromName

`fetchTimeStepFromName`(*pathNames*)

split path name to fetch info on time step

Parameters

pathName (*pathlib path or list of paths*) – path to file including time step info as _t%.2f_

Returns

- **timeStep** (*list*) – actual time steps
- **indexTime** (*numpy array*) – index of timeStep list to order them in ascending order

ana5Utils.distanceTimeAnalysis.getRadarLocation

`getRadarLocation`(*cfg*)

fetch radar location and direction of of field of view coordinates

Parameters

cfg (*configparser object*) – configuration settings

Returns

radarFov – array with x and y coordinates of radar location and point in direction of field of view

Return type

numpy array

ana5Utils.distanceTimeAnalysis.importMTIData

`importMTIData`(*avaDir*, *modName*, *inputDir=''*, *simHash=''*)

import mtiInfo data for creating range time plots, if no inputDir is provided, pickles are fetched from avaDir/Outputs/modName/distanceTimeAnalysis multiple pickles allowed- these carry also configuration info for distanceTimeAnalysis

Parameters

- **avaDir** (*pathlib path or str*) – path to avalanche directory

- **modName** (*str*) – name of computational module that has been used to produce flow variable fields
- **inputDir** (*pathlib path or str*) – optional: path to pickle location
- **simHash** (*str*) – optional simulation ID

Returns
mtiInfoDicts – list of mtiInfo dictionaries where key name has been added with file Path

Return type
list

ana5Utils.distanceTimeAnalysis.initializeRangeTime

`initializeRangeTime`(*modName*, *cfg*, *dem*, *simHash*)

initialize generation of range-time diagram for visualizing simulation data

Parameters

- **modName** – module name
- **cfg** (*configparser object*) – configuration settings from computational module
- **dem** (*dict*) – dictionary with DEM header and data
- **simHash** (*str*) – unique simulation ID

Returns
mtiInfo – info on time steps (timeList) and distance (rangeList) of avalanche front averaged values of result parameter (mti) for each range gate (rangeGates) for colorcoding x, y origin of DEM, cellSize and plotTitle optional info on masks for radar field of view (rangeMasked), etc.

Return type
dict

ana5Utils.distanceTimeAnalysis.maskRangeFull

`maskRangeFull`(*flowF*, *threshold*, *rangeMasked*)

mask range (already masked with radar field of view) also with avalanche result field where NOT above threshold

Parameters

- **flowF** (*numpy array*) – flow variable result field
- **threshold** (*float*) – threshold of result field - masked where values NOT above this threshold
- **rangeMasked** (*numpy masked array*) – range to radar location masked with radar field of view

Returns

- **maskAva** (*numpy mask*) – mask where result field NOT above threshold
- **bmaskAvaRadar** (*numpy mask*) – mask where result field NOT above threshold and NOT in radar field of view
- **rMaskedAvaRadar** (*numpy masked array*) – masked range array with NOT in radar field of view and result field NOT above threshold

ana5Utils.distanceTimeAnalysis.radarMask

radarMask(*demOriginal*, *radarFov*, *aperture*, *cfgRangeTime*)

function to create masked array of radar range using dem and radar's field of view

Parameters

- **dem** (*dict*) – DEM dictionary with header info and rasterData
- **radarFov** (*numpy array*) – radars field of view min and max points
- **aperture** (*float*) – aperature angle
- **cfgRangeTime** (*configparser object*) – configuration settings here used: avalancheDir info

Returns

- **radarRange** (*numpy masked array*) – masked array of radar range covering full DEM extent
- **rangeGates** (*numpy array*) – range gates

ana5Utils.distanceTimeAnalysis.setDemOrigin

setDemOrigin(*demSims*)

reset original origin to simulation DEM - required if for ava sim is set to something different

Parameters

demSims (*dict*) – dictionary with header and data of DEM used to run ava simulations must contain a key called originalHeader - where the original origin is given

Returns

demOriginal – updated DEM with xllcenter and yllcenter set to original origin cellsize and nrows and columns kept from simulation DEM

Return type

dict

ana5Utils.distanceTimeAnalysis.setupRangeTimeDiagram

setupRangeTimeDiagram(*dem*, *cfgRangeTime*)

create setup for creating range-time diagrams from avalanche simulation results with respect to a radar's field of view

Parameters

- **dem** (*dict*) – DEM dictionary with header and raster data
- **cfgRangeTime** (*configparser object*) – configuration settings for range-time diagrams

Returns

rangeMasked – radar range retrieved from DEM and masked with radar's field of view

Return type

numpy masked array

ana5Utils.distanceTimeAnalysis.setupThalwegTimeDiagram

setupThalwegTimeDiagram(*dem*, *cfgRangeTime*)

initialize parameters and prerequisites for generating thalweg-time diagrams

Parameters

- **dem** (*dict*) – dictionary with info on dem header and data
- **cfgRangeTime** (*configparser object*) – configuration settings of range-time diagram

Returns

mtiInfo – dictionary with arrays, lists for range-time diagram: mti for colorcoding (average values of range gates/ crossprofiles) timeList time step info rangeList range info rangeGates info on cross profile locations rangeRaster - info on distance from beta point in path following coordinate system betaPoint - coordinates of start of runout zone point betaPointAngle - angle of beta point used

Return type

dict

2.10.3.5 log2Report

Tools for generating reports

Modules

log2Report.generateCompareReport	Generate a markdown report for data provided in dictionary with format to compare two simulations
log2Report.generateReport	Generate a markdown report for data provided in dictionary

log2Report.generateCompareReport

Generate a markdown report for data provided in dictionary with format to compare two simulations

Functions

copyAimecPlots	copy the aimec plots to report directory
copyQuickPlots	copy the quick plots to report directory
makeLists	Create a list for table creation
writeCompareReport	Write a report in markdown format of the comparison of simulations to reference simulation results; report is saved to location of reportFile

log2Report.generateCompareReport.copyAimecPlots

copyAimecPlots(*plotFiles*, *testName*, *outDir*, *plotPaths*, *rel=''*)

copy the aimec plots to report directory

log2Report.generateCompareReport.copyQuickPlots

copyQuickPlots(*avaName*, *testName*, *outDir*, *plotListRep*, *rel=''*)

copy the quick plots to report directory

log2Report.generateCompareReport.makeLists

makeLists(*simDict*, *benchDict*)

Create a list for table creation

log2Report.generateCompareReport.writeCompareReport

writeCompareReport(*reportFile*, *reportD*, *benchD*, *avaName*, *cfgRep*)

Write a report in markdown format of the comparison of simulations to reference simulation results; report is saved to location of reportFile

Parameters

- **reportFile** (*str*) – path to report file
- **reportD** (*dict*) – report dictionary with simulation info
- **benchD** (*str*) – dictionary with simulation info of refernce simulation
- **avaName** (*str*) – name of avalanche
- **cfgRep** (*dict*) – dictionary with configuration info

log2Report.generateReport

Generate a markdown report for data provided in dictionary

Functions

addLineBlock	add lineblock to report
copyPlots2ReportDir	copy the plots to report directory The plots are in a dictionary: plotDict = {'plot1': PurePath to plot1, 'plot2': PurePath to plot2...}
writeReport	Write a report in markdown format for simulations, saved to outDir
writeReportFile	Create markdown report with blocks, tables, list according to type key

log2Report.generateReport.addLineBlock

addLineBlock(*titleString*, *reportDKey*, *pfile*, *italicFont=False*, *onlyFirstLine=False*)

add lineblock to report

Parameters

- **titleString** (*str*) – string that shall be added
- **reportDKey** (*dict*) – dictionary with info for string
- **pfile** (*file*)
- **italicFont** (*bool*) – if True write value in italic
- **onlyFirstLine** (*bool*) – if first item in reportDKey is added to pfile - break

log2Report.generateReport.copyPlots2ReportDir

copyPlots2ReportDir(*reportDir*, *plotDict*)

copy the plots to report directory The plots are in a dictionary: plotDict = {'plot1': PurePath to plot1, 'plot2': PurePath to plot2... }

Parameters

- **reportDir** (*pathlib path*) – path to directory where to copy the plots to
- **plotDict** (*dict*) – dict of the location of the plots to copy

log2Report.generateReport.writeReport

writeReport(*outDir*, *reportDictList*, *reportOneFile*, *plotDict=''*, *standaloneReport=False*, *reportName='fullSimulationReport'*)

Write a report in markdown format for simulations, saved to outDir

Parameters

- **outDir** (*str*) – path to output directory
- **reportDictList** (*list*) – list of report dictionaries from simulations
- **reportOneFile** (*boolean*) – True to write all info in the same file
- **plotDict** (*dict*) – optional dictionary with info on plots that shall be included in report
- **standaloneReport** (*bool*) – if True copy plots to reportDir
- **reportName** (*str*) – report file name, fullSimulationReport is the default value

log2Report.generateReport.writeReportFile

writeReportFile(*reportD*, *pfile*)

Create markdown report with blocks, tables, list according to type key

Parameters

- **reportD** (*dict*) – report dictionary
- **pfile** (*file object*) – file where markdown format report is written to

2.10.3.6 out1Peak

Simple tools for visualising datasets.

Modules

out1Peak.outPlotAllPeak	Functions to plot 2D simulation results: plot of all peak files at once

out1Peak.outPlotAllPeak

Functions to plot 2D simulation results: plot of all peak files at once

Functions

addConstrainedDataField	find fileName data, constrain data and demField to where there is data, create colormap, define extent, add hillshade contours, add to axes and add colorbar
plotAllFields	Plot all fields within given directory and save to outDir
plotAllPeakFields	Plot all peak fields and return dictionary with paths to plots with DEM in background

out1Peak.outPlotAllPeak.addConstrainedDataField

addConstrainedDataField(*fileName*, *resType*, *demField*, *ax*, *cellSize*, *alpha=1.0*, *setLimits=False*, *oneColor=''*)

find fileName data, constrain data and demField to where there is data, create colormap, define extent, add hillshade contours, add to axes and add colorbar

Parameters

- **fileName** (*pathlib path*) – path to data
- **resType** (*str*) – name of result variable type
- **demField** (*numpy ndarray*) – array of dem data
- **ax** (*matplotlib axes object*) – axes where to add data plot to
- **cellSize** (*float*) – cellSize of data
- **alpha** (*float*) – from 0 transparent to 1 opaque for plot of constrained data

- **setLimits** (*bool*) – if True set limits of constrained data to plot
- **oneColor** (*str*) – optional to add a color for a single color for field

Returns
ax – axes updated

Return type
matplotlib axes object

out1Peak.outPlotAllPeak.plotAllFields

`plotAllFields`(*avaDir*, *inputDir*, *outDir*, *unit=''*, *constrainData=True*)

Plot all fields within given directory and save to outDir

Parameters

- **avaDir** (*str*) – path to avalanche directoy
- **inputDir** (*str*) – path to input directoy
- **outDir** (*str*) – path to directoy where plots shall be saved to
- **unit** (*str*) – unit of result type

out1Peak.outPlotAllPeak.plotAllPeakFields

`plotAllPeakFields`(*avaDir*, *cfgFLAGS*, *modName*, *demData=''*)

Plot all peak fields and return dictionary with paths to plots with DEM in background

Parameters

- **avaDir** (*str*) – path to avalanche directoy
- **cfgFLAGS** (*str*) – general configuration, required to define if plots saved to reports directoy
- **modName** (*str*) – name of module that has been used to produce data to be plotted
- **demData** (*dictionary*) – optional - if not the dem in the avaDir/Inputs folder has been used but a different one

Returns
plotDict – dictionary with info on plots, like path to plot

Return type
dict

2.10.3.7 out3Plot

Plotting

Modules

out3Plot.amaPlots	functions for plotting ama event data and geometry info
out3Plot.in1DataPlots	make plots for in1Data module
out3Plot.outAIMEC	Plotting and saving AIMEC results
out3Plot.outAna1Plots	
out3Plot.outCom1DFA	
out3Plot.outCom3Plots	
out3Plot.outContours	Plotting and saving contour line plots
out3Plot.outDebugPlots	
out3Plot.outDistanceTimeAnalysis	functions to plot range-time diagrams
out3Plot.outParticlesAnalysis	Tools to extract information on the avalanche simulations run in the Output files
out3Plot.outQuickPlot	Functions to plot 2D avalanche simulation results as well as comparison plots between two datasets of identical shape.
out3Plot.outTopo	Simple plotting for DEMs
out3Plot.statsPlots	plot statistics of simulations

out3Plot.amaPlots

functions for plotting ama event data and geometry info

Functions

plotBoxPlot	create a boxplot of all columns of dbData
plotHist	create a histogram of name column of dbData
plotPathAngle	make a two panel plot of avalanche thalweg, release point, runout point, origin, transit and deposition point and xy distances and computed angles

out3Plot.amaPlots.plotBoxPlot

plotBoxPlot(*dbData*, *colList*, *outDir*, *namePlot*, *renameCols=[]*)

create a boxplot of all columns of dbData

Parameters

- **dbData** (*pandas dataframe*) – dataframe with geometry info
- **colList** (*list*) – names of columns in dbData to plot
- **outDir** (*pathlib path or str*) – path to folder where plot shall be saved to
- **namePlot** (*str*) – name of plot to be added to boxplot

out3Plot.amaPlots.plotHist

plotHist(*dbData*, *name*, *cfgMain*, *colorcode=''*)

create a histogram of name column of dbData

Parameters

- **dbData** (*pandas dataframe*) – dataframe with geometry info of events and name_Line and name_Angle
- **name** (*str*) – name of Line and Angle to plot
- **cfgMain** (*configparser*) – configuration settings
- **colorcode** (*str*) – name of column to colorcode data

out3Plot.amaPlots.plotPathAngle

plotPathAngle(*dbData*, *cfgMain*, *nameEvent*, *tickOrigin='orig-transit'*)

make a two panel plot of avalanche thalweg, release point, runout point, origin, transit and deposition point and xy distances and computed angles

Parameters

- **dbData** (*pandas dataframe*) – dataframe with geometry info of events
- **cfgMain** (*configparser object*) – config settings here used: avalancheDir, projstr,
- **nameEvent** (*str*) – name of xy distance lines and angles
- **tickOrigin** (*str*) – origin for panel 2 Sxy distance origin

out3Plot.in1DataPlots

make plots for in1Data module

Functions

plotAreaShpError	plot polygon parts of polygon read from shp file to check if holes
plotDist	plot the CDF
plotECDF	make a comparison plot of desired CDF and empirical CDF of sample
plotEmpCDF	make a comparison plot of desired CDF and empirical CDF of sample
plotEmpPDF	make a comparison plot of desired CDF and empirical CDF of sample
plotSample	Generate bar plot of sample values
plotSamplePDF	make comparison plot of desired PDF and approximated sample PDF

out3Plot.in1DataPlots.plotAreaShpError

plotAreaShpError(*xFeat*, *yFeat*, *nParts*, *pathDict*)

plot polygon parts of polygon read from shp file to check if holes

Parameters

- **xFeat, yFeat** (*numpy array*) – x, y coordinates of polygon
- **nParts** (*list*) – indices of parts of polygon (2 make only an outer polygon as entire number of points is added)
- **pathDict** (*dict*) – dictionary with info on outDir, outFileName and title of plot

out3Plot.in1DataPlots.plotDist

plotDist(*workingDir*, *CDF*, *a*, *b*, *c*, *cfg*, *flagShow*)

plot the CDF

out3Plot.in1DataPlots.plotECDF

plotECDF(*workingDir*, *CDF*, *sample*, *cfg*, *methodAbbr*, *flagShow*)

make a comparison plot of desired CDF and empirical CDF of sample

out3Plot.in1DataPlots.plotEmpCDF

plotEmpCDF(*workingDir*, *CDF*, *CDFEmp*, *xSample*, *cfg*, *methodAbbr*, *flagShow*, *x=''*)

make a comparison plot of desired CDF and empirical CDF of sample

out3Plot.in1DataPlots.plotEmpPDF

plotEmpPDF(*workingDir*, *PDF*, *sampleVect*, *cfg*, *flagShow*, *x=''*)

make a comparison plot of desired CDF and empirical CDF of sample

out3Plot.in1DataPlots.plotSample

plotSample(*workingDir*, *sample*, *cfg*, *flagShow*)

Generate bar plot of sample values

out3Plot.in1DataPlots.plotSamplePDF

plotSamplePDF(*workingDir*, *sampleVect*, *kdeDict*, *PDF*, *cfg*, *flagShow*)

make comparison plot of desired PDF and approximated sample PDF

out3Plot.outAIMEC

Plotting and saving AIMEC results

This file is part of Avaframe.

Functions

addLinePlot	add a contour line with label only for line that has _0 in name
addThalwegAltitude	add thalweg-altitude plot to axes the thalweg z profile and the mpfv^2/2g (velocity-altitude) colorcoded using the mpfv values the mpfv values come from the cross max values along the thalweg coordinate system (aimec)
fetchContourLines	fetch contour line of transformed field data
getIndicesVel	create indices of peak flow velocity cross max vector first above tresholdValue and first below threshold again
plotContoursTransformed	plot contour lines of all transformed fields colorcode contour lines with first parameter in varParList if not type string of value
plotMaxValuesComp	plot result type name1 vs name 2 and colorcode scenarios using hue add reference sim value with label
plotThalwegAltitude	create thalweg-altitude plot the thalweg z profile and the mpfv^2/2g (velocity-altitude) colorcoded using the mpfv values the mpfv values come from the cross max values along the thalweg coordinate system (aimec) plot is saved to avaDir/Outputs/out3Plot/thalwegAltSimname
plotVelThAlongThalweg	plot the velocity and thickness cross max values along the thalweg, with pft x10 only plot every barInt value
resultVisu	plot the normalized area difference between reference and other simulations
resultWrite	This function writes the main Aimec results to a file (outputFile) in pathDict
visuComparison	Plot and save the comparison between current simulation and Reference in the runout area
visuMass	Plot and save the results from mass analysis
visuRunoutComp	Plot and save the Peak Fields distribution (max mean per cross section) after coordinate transformation
visuRunoutStat	Panel1 reference peak field with runout points of all sims and distribution of runout SXY Panel 2 crossMax values of peak field along thalweg for all sims Panel 3 mean, median and envelope of cross max values for all sims
visuSimple	Plot and save the Peak Pressure Peak Flow thickness and Peak speed fields after coord transfo
visuTransfo	Plot and save the domain transformation figure The left subplot shows the reference result raster with the outline of the new domain.

out3Plot.outAIMEC.addLinePlot

addLinePlot(*contourDict*, *colorStr*, *labelStr*, *ax*, *key*, *zorder=''*, *linestyle='solid'*, *alpha=1.0*)

add a contour line with label only for line that has _0 in name

Parameters

- **contourDict** (*dict*) – dict with x, y info of contourline coordinates
- **colorStr** (*str*) – color of line
- **labelStr** (*str*) – name of legend entry for line
- **ax** (*matplotlib axes object*) – axes where to add the lines too
- **key** (*str*) – name of the contour line
- **zorder** (*int*) – order of line on plot
- **linestyle** (*str*) – style of line plot: dashed, solid, dotted,..
- **alpha** (*float*) – between 0 and 1 to define how opaque the line

out3Plot.outAIMEC.addThalwegAltitude

addThalwegAltitude(*ax1*, *rasterTransfo*, *pfvCrossMax*, *velocityThreshold*, *zMaxM=numpy.nan*)

add thalweg-altitude plot to axes the thalweg z profile and the mpfv^2/2g (velocity-altitude) colorcoded using the mpfv values the mpfv values come from the cross max values along the thalweg coordinate system (aimec)

Parameters

- **rasterTransfo** (*dict*) – dict with info on transformation from cartesian to thalweg coordinate system
- **pfvCrossMax** (*pandas dataFrame series*) – cross profile max values of peak flow velocity transformed into thalweg coordinate system
- **zMaxM** (*float*) – optional - value to define max limit of y -axis

out3Plot.outAIMEC.fetchContourLines

fetchContourLines(*rasterTransfo*, *inputs*, *level*, *contourDict*)

fetch contour line of transformed field data

Parameters

- **rasterTransfo** (*dict*) – raster transformation dictionary
- **inputs** (*dict*) – input data here needed the transformed field data and simName
- **level** (*float*) – the contour level
- **contourDict** (*dict*) – dictionary with x, y coordinates for each sim contour line

Returns

contourDict – updat. dictionary with name of contour line: x, y coordinates for each sim contour line -> added info of current simulation

Return type

dict

out3Plot.outAIMEC.getIndicesVel

`getIndicesVel`(*pfvCM*, *velocityThreshold*)

create indices of peak flow velocity cross max vector first above tresholdValue and first below threshold again

Parameters

- **pfvCM** (*numpy array*) – peak flow velocity cross max values along thalweg
- **velocityThreshold** (*float*) – threhshold value

Returns

- **indVelStart** (*int*) – index where pfvCM first exceeds threshold
- **indVelZero** (*int*) – index where pfvCM first smaller than threshold but only further downstream than indVelStart

out3Plot.outAIMEC.plotContoursTransformed

`plotContoursTransformed`(*contourDict*, *pathDict*, *rasterTransfo*, *cfgSetup*)

plot contour lines of all transformed fields colorcode contour lines with first parameter in varParList if not type string of value

Parameters

- **contourDict** (*dict*) – dictionary with contourline coordinates
- **pathDict** (*dict*) – dictionary with info on project name
- **rasterTransfo** (*dict*) – raster transformation dictionary
- **cfgSetup** (*configparser*) – configuration settings for AIMECSETUP

out3Plot.outAIMEC.plotMaxValuesComp

`plotMaxValuesComp`(*pathDict*, *resultsDF*, *name1*, *name2*, *hue=None*)

plot result type name1 vs name 2 and colorcode scenarios using hue add reference sim value with label

resultsDF: pandas dataFrame
dataframe with one row per simulation with info on parameters and aimec analysis results

name1: str
name of result type one

name2: str
name of result type two

hue: str
name of parameter used for colorcoding

out3Plot.outAIMEC.plotThalwegAltitude

plotThalwegAltitude(*pathDict*, *rasterTransfo*, *pfvCrossMax*, *simName*, *velocityThreshold*)

create thalweg-altitude plot the thalweg z profile and the mpfv^2/2g (velocity-altitude) colorcoded using the mpfv values the mpfv values come from the cross max values along the thalweg coordinate system (aimec) plot is saved to avaDir/Outputs/out3Plot/thalwegAltSimname

Parameters

- **avaDir** (*pathlib path*) – path to avalanche directory
- **rasterTransfo** (*dict*) – dict with info on transformation from cartesian to thalweg coordinate system
- **pfvCrossMax** (*pandas dataFrame series*) – cross profile max values of peak flow velocity transformed into thalweg coordinate system
- **simName** (*str*) – simulation name

out3Plot.outAIMEC.plotVelThAlongThalweg

plotVelThAlongThalweg(*pathDict*, *rasterTransfo*, *pftCrossMax*, *pfvCrossMax*, *cfgPlots*, *simName*)

plot the velocity and thickness cross max values along the thalweg, with pft x10 only plot every barInt value

Parameters

- **pathDict** (*dict*) – info on avalancheDir
- **rasterTransfo** (*dict*) – info on domain transformation
- **pftCrossMax** (*numpy nd array*) – peak flow thickness max values along cross profiles of thalweg coordinate system
- **pfvCrossMax** (*numpy nd array*) – peak flow velocity max values along cross profiles of thalweg coordinate system
- **cfgPlots** (*configparser object*) – used: barInterval: width to compute the values that should be used for plotting velocityTreshold: threshold for computation of alpha angle where to identify stop of avalanche
- **simName** (*str*) – simulation name

out3Plot.outAIMEC.resultVisu

resultVisu(*cfgSetup*, *inputsDF*, *pathDict*, *cfgFlags*, *rasterTransfo*, *resAnalysisDF*)

plot the normalized area difference between reference and other simulations

out3Plot.outAIMEC.resultWrite

`resultWrite`(*pathDict*, *cfg*, *rasterTransfo*, *resAnalysisDF*)

This function writes the main Aimec results to a file (outputFile) in pathDict

out3Plot.outAIMEC.visuComparison

`visuComparison`(*rasterTransfo*, *inputs*, *pathDict*)

Plot and save the comparison between current simulation and Reference in the runout area

Parameters

- **rasterTransfo** (*dict*) – domain transformation information
- **inputs** (*dict*) –

  input data for plot:
  'refData' and 'compData' arrays 'refRasterMask' and 'compRasterMask' arrays 'thresholdArray' 'i' id of the simulation 'runoutResType' result type analyzed 'diffLim'
- **pathDict** (*dict*) – dictionary with path to data to analyze

out3Plot.outAIMEC.visuMass

`visuMass`(*resAnalysisDF*, *pathDict*, *simRowHash*, *refSimRowHash*, *timeMass*)

Plot and save the results from mass analysis

Parameters

- **resAnalysis** (*dict*) –

  mass results from Aimec analysis:
  entMassFlowArray: entrained mass array corresponding to the time array for each simulation to analyse totalMassArray: entrained mass array corresponding to the time array for each simulation to analyse entMass: final entrained for each simulation finalMass: final mass for each simulation time: time array
- **pathDict** (*dict*) – dictionary with path to data to analyze

out3Plot.outAIMEC.visuRunoutComp

`visuRunoutComp`(*rasterTransfo*, *resAnalysisDF*, *cfgSetup*, *pathDict*)

Plot and save the Peak Fields distribution (max mean per cross section) after coordinate transformation

Parameters

- **rasterTransfo** (*dict*) – domain transformation information
- **resAnalysis** (*dict*) –

  results from Aimec analysis (for ppr, pft and pfv):
  PPRCrossMax: numpy array with max peak field along path for each file to analyse PPRCrossMean: numpy array with mean peak field along path for each file to analyse
- **cfgSetup** (*configparser*) – configparser with ana3AIMEC settings defined in ana3AIMECCfg.ini 'runoutResType' and 'thresholdValue'

- **pathDict** (*dict*) – dictionary with path to data to analyze

out3Plot.outAIMEC.visuRunoutStat

`visuRunoutStat`(*rasterTransfo*, *inputsDF*, *resAnalysisDF*, *newRasters*, *cfgSetup*, *pathDict*)

Panel1 reference peak field with runout points of all sims and distribution of runout SXY Panel 2 crossMax values of peak field along thalweg for all sims Panel 3 mean, median and envelope of cross max values for all sims

Parameters

- **rasterTransfo** (*dict*) – domain transformation information
- **inputsDF** (*dataFrame*) – aimec inputs DF
- **resAnalysis** (*dataFrame*) –

  results from Aimec analysis:
  numpy array with max peak field (of the 'runoutResType') along path for each file to analyse runout for 'runoutResType'
- **newRasters** (*dict*) – dictionary with new (s, l) raster for the 'runoutResType'
- **cfgSetup** (*configparser*) – configparser with ana3AIMEC settings defined in ana3AIMECCfg.ini 'percentile', 'runoutResType' and 'thresholdValue'
- **pathDict** (*dict*) – dictionary with path to data to analyze

out3Plot.outAIMEC.visuSimple

`visuSimple`(*cfgSetup*, *rasterTransfo*, *resAnalysisDF*, *newRasters*, *pathDict*)

Plot and save the Peak Pressure Peak Flow thickness and Peak speed fields after coord transfo

Parameters

- **rasterTransfo** (*dict*) – domain transformation information
- **resAnalysis** (*dict*) –

  results from Aimec analysis:
  numpy array with the 'runout' for each simulation and the 'thresholdValue'
- **newRasters** (*dict*) – dictionary with new (s, l) raster for ppr, pft and pft
- **pathDict** (*dict*) – dictionary with path to data to analyze

out3Plot.outAIMEC.visuTransfo

`visuTransfo`(*rasterTransfo*, *inputData*, *cfgSetup*, *pathDict*)

Plot and save the domain transformation figure The left subplot shows the reference result raster with the outline of the new domain. The second one shows this same data in the (s,l) coordinate system define by the outline in the first plot.

Parameters

- **rasterTransfo** (*dict*) – domain transformation information
- **inputData** (*dict*) –

inputData dictionary:
slRaster: numpy array with (s,l) raster xyRaster: numpy array with (x,y) raster headerXY: header corresponding to xyRaster

- **cfgSetup** (*configparser*) – configparser with ana3AIMEC settings defined in ana3AIMECCfg.ini 'runoutResType'
- **pathDict** (*dict*) – dictionary with path to data to analyze

out3Plot.outAna1Plots

Functions

addContour2Plot	Make a contour plot of flow thickness for analytical solution and simulation result
addErrorTime	plot error between a given com1DFA sol and the analytic sol function of time on ax1 and ax2
first_nonzero	Get index of first non zero value
getLabel	Get error plot title (relativ error or not?)
getPlotLimits	Get x and y axis limits for the profile and contour plots
getTitleError	Get error plot title (relativ error or not?)
last_nonzero	Get index of last non zero value
makeContourSimiPlot	
plotComparisonDam	Generate plots that compare the simulation results to the analytical solution
plotDamAnaResults	Create plots of the analytical solution for the given settings, including an animation
plotDamBreakSummary	Plot sumary figure of the damnreak test
plotErrorConvergence	plot error between all com1DFA sol and analytic sol function of whatever you want
plotErrorTime	plot and save error between a given com1DFA sol and the analytic sol function of time
plotPresentation	plot error between all com1DFA sol in simDF and analytic sol function of whatever you want
plotSimiSolSummary	Plot sumary figure of the similarity solution test
plotTimeCPULog	plot computation time function of nParts function of whatever (ini parameter given in the simDF) you want
saveSimiSolProfile	Generate plots of the comparison of DFA solution and simiSol

out3Plot.outAna1Plots.addContour2Plot

addContour2Plot(*ax1*, *fieldFT*, *simiDict*, *fieldHeader*, *limits*, *nLevels=9*)

Make a contour plot of flow thickness for analytical solution and simulation result

out3Plot.outAna1Plots.addErrorTime

addErrorTime(*ax1*, *time*, *hErrorL2Array*, *hErrorLMaxArray*, *vhErrorL2Array*, *vhErrorLMaxArray*, *relativ*, *t*)

plot error between a given com1DFA sol and the analytic sol function of time on ax1 and ax2

Parameters

- **ax1** (*matplotlib axis*) – axis where the erro should be ploted
- **time** (*1D numpy array*) – time array
- **hErrorL2Array** (*1D numpy array*) – flow thickness L2 error array
- **hErrorLMaxArray** (*1D numpy array*) – flow thickness LMax error array
- **vhErrorL2Array** (*1D numpy array*) – flow momentum L2 error array
- **vhErrorLMaxArray** (*1D numpy array*) – flow momentum LMax error array
- **relativ** (*str*)
- **t** (*float*) – time for vertical line

out3Plot.outAna1Plots.first_nonzero

first_nonzero(*arr*, *axis*, *invalid_val=-1*)

Get index of first non zero value

Parameters

- **arr** (*numpy array*) – data array
- **axis** (*int*) – axis along which you want to get the index

Return type

index of first non zero in axis direction

out3Plot.outAna1Plots.getLabel

getLabel(*start*, *end*, *dir=''*, *vert=True*)

Get error plot title (relativ error or not?)

out3Plot.outAna1Plots.getPlotLimits

getPlotLimits(*cfgSimi*, *fieldsList*, *fieldHeader*)

Get x and y axis limits for the profile and contour plots

Parameters

- **cfgSimi** (*configparser*) – simiSol cfg
- **fieldsList** (*list*) – list of fields Dictionaries
- **fieldHeader** (*dict*) – field header dictionary

out3Plot.outAna1Plots.getTitleError

getTitleError(*relativ*, *ending=''*)

Get error plot title (relativ error or not?)

out3Plot.outAna1Plots.last_nonzero

last_nonzero(*arr*, *axis*, *invalid_val=-1*)

Get index of last non zero value

Parameters

- **arr** (*numpy array*) – data array
- **axis** (*int*) – axis along which you want to get the index

Return type
index of last non zero in axis direction

out3Plot.outAna1Plots.makeContourSimiPlot

makeContourSimiPlot(*avalancheDir*, *simHash*, *fieldFT*, *limits*, *simiDict*, *fieldHeader*, *tSave*, *outDirTest*)

out3Plot.outAna1Plots.plotComparisonDam

plotComparisonDam(*cfg*, *simHash*, *fields0*, *fieldsT*, *header*, *solDam*, *tSave*, *limits*, *outDirTest*)

Generate plots that compare the simulation results to the analytical solution

Parameters

- **cfgC** (*configParser object*) – configuration setting for avalanche simulation including DAMBREAK section
- **simHash** (*str*) – simulation hash
- **fields0** (*dict*) – initial time step field dictionary
- **fieldsT** (*dict*) – tSave field dictionary
- **header** (*dict*) – fields header dictionary
- **solDam** (*dict*) –

  analytic solution dictionary:

  tAna: 1D numpy array
  time array

  h0: float
  release thickness

  hAna: 2D numpy array
  Flow thickness (rows for x and columns for time)

  uAna: 2D numpy array
  flow velocity (rows for x and columns for time)

xAna: 2D numpy array
extent of domain in the horizontal plane coordinate system (rows for x and columns for time)

xMidAna: 1D numpy array
middle of the material in x dir in the horizontal plane coordinate system (used to compute the error)

- **tSave** (*float*) – time step of analaysis
- **limits** (*dict*) – y extend for profile plots
- **outDirTest** (*pathli path*) – path to output directory

out3Plot.outAna1Plots.plotDamAnaResults

`plotDamAnaResults`(*t*, *x*, *xMiddle*, *h*, *u*, *tSave*, *cfg*, *outDirTest*)

Create plots of the analytical solution for the given settings, including an animation

out3Plot.outAna1Plots.plotDamBreakSummary

`plotDamBreakSummary`(*avalancheDir*, *timeList*, *fieldsList*, *fieldHeader*, *solDam*, *hErrorL2Array*, *hErrorLMaxArray*, *vhErrorL2Array*, *vhErrorLMaxArray*, *outDirTest*, *simDFrow*, *simHash*, *cfg*)

Plot sumary figure of the damnreak test

Parameters

- **avalancheDir** (*pathlib path*) – path to avalanche directory
- **timeList** (*list*) – list of time steps
- **fieldsList** (*list*) – list of fields dictionaries
- **timeList** (*list*) – list of time steps
- **fieldHeader** (*dict*) – header dictionary with info about the extend and cell size
- **solDam** (*dict*) – analytic solution dictionary
- **fieldHeader** (*dict*) – header dictionary with info about the extend and cell size
- **hErrorL2Array** (*numpy array*) – L2 error on flow thickness for saved time steps
- **hErrorLMaxArray** (*numpy array*) – LMax error on flow thickness for saved time steps
- **vErrorL2Array** (*numpy array*) – L2 error on flow velocity for saved time steps
- **vErrorLMaxArray** (*numpy array*) – LMax error on flow velocity for saved time steps
- **outDirTest** (*pathlib path*) – path output directory (where to save the figures)
- **simDFrow** (*dataFrame*) – data frame row corresponding to simHash
- **simHash** (*str*) – com1DFA simulation id
- **cfg** (*configParser object*) – configuration setting for avalanche simulation including DAMBREAK section

out3Plot.outAna1Plots.plotErrorConvergence

plotErrorConvergence(*simDF*, *outDirTest*, *cfgSimi*, *xField*, *yField*, *coloredBy*, *sizedBy*, *logScale=False*, *fit=False*)

plot error between all com1DFA sol and analytic sol function of whatever you want

The coloredBy, sizedBy can not be corresponding to non numeric parameters.

Parameters

- **simDF** (*dataFrame*) – the simulation data with the postprocessing results
- **outDirTest** (*str or pathlib*) – output directory
- **cfgSimi** (*configparser*) – the cfg
- **xField** (*str*) – column of the simDF to use for the x axis
- **yField** (*str*) – column of the simDF to use for the y axis
- **coloredBy** (*str*) – column of the simDF to use for the colors
- **sizedBy** (*str*) – column of the simDF to use for the marker size
- **logScale** (*boolean*) – If you want a loglog scale
- **fit** (*boolean*) – if True add power law regression

out3Plot.outAna1Plots.plotErrorTime

plotErrorTime(*time*, *hErrorL2Array*, *hErrorLMaxArray*, *vhErrorL2Array*, *vhErrorLMaxArray*, *relativ*, *t*, *outputName*, *outDirTest*)

plot and save error between a given com1DFA sol and the analytic sol function of time

Parameters

- **time** (*1D numpy array*) – time array
- **hErrorL2Array** (*1D numpy array*) – flow thickness L2 error array
- **hErrorLMaxArray** (*1D numpy array*) – flow thickness LMax error array
- **vhErrorL2Array** (*1D numpy array*) – flow momentum L2 error array
- **vhErrorLMaxArray** (*1D numpy array*) – flow momentum LMax error array
- **relativ** (*str*)
- **t** (*float*) – time for vertical line
- **outputName** (*str*) – figure name
- **outDirTest** (*str or pathlib*) – output directory

out3Plot.outAna1Plots.plotPresentation

plotPresentation(*simDF*, *outDirTest*, *cfgSimi*, *xField*, *yField*, *coloredBy*, *sizedBy*, *logScale=False*, *fit=False*)

plot error between all com1DFA sol in simDF and analytic sol function of whatever you want

The coloredBy, sizedBy can not be corresponding to non numeric parameters.

Parameters

- **simDF** (*dataFrame*) – the simulation data with the postprocessing results
- **outDirTest** (*str or pathlib*) – output directory
- **cfgSimi** (*configparser*) – the cfg
- **xField** (*str*) – column of the simDF to use for the x axis
- **yField** (*str*) – column of the simDF to use for the y axis
- **coloredBy** (*str*) – column of the simDF to use for the colors
- **sizedBy** (*str*) – column of the simDF to use for the marker size
- **logScale** (*boolean*) – If you want a loglog scale
- **fit** (*boolean*) – if True add power law regression

out3Plot.outAna1Plots.plotSimiSolSummary

plotSimiSolSummary(*avalancheDir*, *timeList*, *fieldsList*, *fieldHeader*, *simiDict*, *hErrorL2Array*, *hErrorLMaxArray*, *vhErrorL2Array*, *vhErrorLMaxArray*, *outDirTest*, *simDFrow*, *simHash*, *cfgSimi*)

Plot sumary figure of the similarity solution test

Parameters

- **avalancheDir** (*pathlib path*) – path to avalanche directory
- **timeList** (*list*) – list of time steps
- **fieldsList** (*list*) – list of fields dictionaries
- **timeList** (*list*) – list of time steps
- **fieldHeader** (*dict*) – header dictionary with info about the extend and cell size
- **simiDict** (*dict*) – analytic solution dictionary
- **fieldHeader** (*dict*) – header dictionary with info about the extend and cell size
- **hErrorL2Array** (*numpy array*) – L2 error on flow thickness for saved time steps
- **hErrorLMaxArray** (*numpy array*) – LMax error on flow thickness for saved time steps
- **vErrorL2Array** (*numpy array*) – L2 error on flow velocity for saved time steps
- **vErrorLMaxArray** (*numpy array*) – LMax error on flow velocity for saved time steps
- **outDirTest** (*pathlib path*) – path output directory (where to save the figures)
- **simDFrow** (*dataFrame*) – data frame row corresponding to simHash
- **simHash** (*str*) – com1DFA simulation id
- **cfgSimi** (*configParser object*) – SIMISOL configuration

out3Plot.outAna1Plots.plotTimeCPULog

`plotTimeCPULog`(*simDF*, *outDirTest*, *cfgSimi*, *xField*, *coloredBy*, *sizedBy*, *logScale=False*)

plot computation time function of nParts function of whatever (ini parameter given in the simDF) you want

Parameters

- **simDF** (*dataFrame*) – the simulation data with the postprocessing results
- **outDirTest** (*str or pathlib*) – output directory
- **cfgSimi** (*configparser*) – the cfg
- **xField** (*str*) – column of the simDF to use for the x axis
- **coloredBy** (*str*) – column of the simDF to use for the colors
- **sizedBy** (*str*) – column of the simDF to use for the marker size
- **logScale** (*boolean*) – If you want a loglog scale

out3Plot.outAna1Plots.saveSimiSolProfile

`saveSimiSolProfile`(*cfgMain*, *cfgSimi*, *fields*, *limits*, *simiDict*, *tSave*, *header*, *outDirTest*, *simHash*)

Generate plots of the comparison of DFA solution and simiSol

Parameters

- **cfgMain** (*configparser*) – main cfg
- **cfgSimi** (*configparser*) – simiSol cfg
- **fieldsList** (*list*) – list of fields Dictionaries
- **solSimi** (*dict*) – dictionary with similiarty solution
- **tSave** (*float*) – time coresponding to fields
- **header** (*dict*) – field header dictionary
- **outDirTest** (*str or pathlib*) – output directory
- **simHash** (*str*) – hash of the curent simulation

out3Plot.outCom1DFA

Functions

addDem2Plot	Add dem to the background of a plot
addParticles2Plot	Update axes with particles
addResult2Plot	Add raster data to a plot
createContourPlot	create a contour line plot of all simulations of current run
fetchContCoors	fetch coordinates of contour line
plotAllPartAcc	Plot time series of tracked particles :Parameters: * **outDirData** (*str*) -- path to output directory * **particlesList** (*list*) -- list with dictionary of all particles time steps particles * **cfg** (*dict*) -- configuration read from ini file * **Tsave** (*list*) -- list of saving time step info * **cuSimName** (*str*) -- name of current sim
plotParticles	Plot particles on dem
plotReleaseScenarioView	plot release polygon, area with thickness on dem hillshade saved to avaDir/Outputs/com1DFA/reports
plotTrackParticle	Plot time series of tracked particles
plotTrackParticleAcceleration	Plot time series of tracked particles :Parameters: * **outDirData** (*str*) -- path to output directory * **trackedPartProp** (*dict*) -- dictionary with time series of the wanted properties for tracked particles * **cfg** (*dict*) -- configuration read from ini file * **cuSimName** (*str*) -- name of simulation
updateTrackPart	Update axes with particles (tracked particles are highlighted in red)

out3Plot.outCom1DFA.addDem2Plot

addDem2Plot(*ax*, *dem*, *what='slope'*, *extent=''*, *origHeader=False*)

Add dem to the background of a plot

Parameters

- **ax** (*matplotlib ax object*)
- **dem** (*dict*) – dem dictionary with normal information
- **what** (*str*) – what information about the dem will be plotted? slope: use the dem slope (computed from the normals) to color the plot z : use the elevation to color the plot
- **origHeader** (*bool*) – if True use originalHeader and not header

out3Plot.outCom1DFA.addParticles2Plot

addParticles2Plot(*particles*, *ax*, *dem*, *whatS='m'*, *whatC='h'*, *colBarResType=''*)

Update axes with particles

Parameters

- **particles** (*dict*) – particles dictionary
- **ax** (*matplotlib ax object*)
- **dem** (*dict*) – dem dictionary with normal information

- **whatS** (*str*) – which particle property should be used for the markersize
- **whatC** (*str*) – which particle property should be used for the marker color

out3Plot.outCom1DFA.addResult2Plot

`addResult2Plot`(*ax*, *header*, *rasterData*, *resType*, *colorbar=True*, *contour=False*)

Add raster data to a plot

Parameters

- **ax** (*matplotlib ax object*)
- **header** (*dict*) – raster header dictionary
- **rasterData** (*2D numpy array*) – data to plot
- **resType** (*str*) – what kinf of result is it? ppr, pft...
- **colorbar** (*bool*) – If true add the colorbar

out3Plot.outCom1DFA.createContourPlot

`createContourPlot`(*reportDictList*, *avalancheDir*, *simDF*)

create a contour line plot of all simulations of current run

Parameters

- **reportDictList** (*list*) – list of com1DFA dictionary with info on contour dicts for each sim
- **avalancheDir** (*str or pathlib path*) – path to avalanche directory
- **simDF** (*pandas DataFrame*) – dataframe with one row per simulation performed and its parameter settings

Returns

reportDictList – updated reportDictList - deleted contours dict

Return type

list

out3Plot.outCom1DFA.fetchContCoors

`fetchContCoors`(*demHeader*, *flowF*, *cfgVisu*, *simName*)

fetch coordinates of contour line

Parameters

- **demHeader** (*dict*) – dictionary of dem nrows, ncols, cellsize
- **flowF** (*np.ndarray*) – field data used to compute contour line
- **cfgVisu** (*configparser object*) – configuration settings for visualisation, here used: contourResType, thresholdValue
- **simName** (*str*) – simName
- **Rertuns**
- **———**

- **contDictXY** (*dict*) – dictionary with simName and subDict with x, y coordinates of contour line contourResType and thresholdValue

out3Plot.outCom1DFA.plotAllPartAcc

`plotAllPartAcc`(*outDirData*, *particlesList*, *cfg*, *Tsave*, *cuSimName*)

Plot time series of tracked particles :Parameters: * **outDirData** (*str*) – path to output directory

- **particlesList** (*list*) – list with dictionary of all particles time steps particles
- **cfg** (*dict*) – configuration read from ini file
- **Tsave** (*list*) – list of saving time step info
- **cuSimName** (*str*) – name of current sim

out3Plot.outCom1DFA.plotParticles

`plotParticles`(*particlesList*, *cfg*, *dem*)

Plot particles on dem

Parameters

- **particlesList** (*list*) – list or particles dictionaries
- **cfg** (*dict*) – configuration read from ini file
- **dem** (*dict*) – dem dictionary with normal information

out3Plot.outCom1DFA.plotReleaseScenarioView

`plotReleaseScenarioView`(*avaDir*, *releaseLine*, *dem*, *titleFig*, *cuSimName*)

plot release polygon, area with thickness on dem hillshade saved to avaDir/Outputs/com1DFA/reports

Parameters

- **avaDir** (*str*) – path to ava directory
- **dem** (*dict*) – dict with dem header and data
- **releaseLine** (*dict*) – dict with raster of release line and x,y coors
- **titleFig** (*str*) – title of figure
- **cuSimName** (*str*) – name of simulation

out3Plot.outCom1DFA.plotTrackParticle

`plotTrackParticle`(*outDirData*, *particlesList*, *trackedPartProp*, *cfg*, *dem*, *cuSimName*)

Plot time series of tracked particles

Parameters

- **outDirData** (*str*) – path to output directory
- **particlesList** (*list*) – list or particles dictionaries

- **trackedPartProp** (*dict*) – dictionary with time series of the wanted properties for tracked particles
- **cfg** (*dict*) – configuration read from ini file
- **dem** (*dict*) – dem dictionary with normal information
- **cuSimName** (*str*) – name of current simulation

out3Plot.outCom1DFA.plotTrackParticleAcceleration

plotTrackParticleAcceleration(*outDirData*, *trackedPartProp*, *cfg*, *cuSimName*)

Plot time series of tracked particles :Parameters: * **outDirData** (*str*) – path to output directory

- **trackedPartProp** (*dict*) – dictionary with time series of the wanted properties for tracked particles
- **cfg** (*dict*) – configuration read from ini file
- **cuSimName** (*str*) – name of simulation

out3Plot.outCom1DFA.updateTrackPart

updateTrackPart(*particles*, *ax*, *dem*)

Update axes with particles (tracked particles are highlighted in red)

out3Plot.outCom3Plots

Functions

generateCom1DFAPathPlot	Make energy test analysis and plot results
hybridPathPlot	Update path plot with result of curent iteration
hybridProfilePlot	Update profile plot with result of curent iteration

out3Plot.outCom3Plots.generateCom1DFAPathPlot

generateCom1DFAPathPlot(*avalancheDir*, *cfgPath*, *avaProfileMass*, *dem*, *parabolicFit*, *splitPoint*, *simName*)

Make energy test analysis and plot results

Parameters

- **avalancheDir** (*pathlib*) – avalanche directory pathlib path
- **cfgPath** (*configParser*) – path finding config
- **avaProfileMass** (*dict*) – particle mass averaged properties
- **dem** (*dict*) – com1DFA simulation dictionary
- **fieldsList** (*list*) – field dictionary list
- **simName** (*str*) – simulation name

out3Plot.outCom3Plots.hybridPathPlot

hybridPathPlot(*avalancheDir*, *dem*, *resultsHybrid*, *fields*, *particles*, *muArray*)

Update path plot with result of curent iteration

out3Plot.outCom3Plots.hybridProfilePlot

hybridProfilePlot(*avalancheDir*, *resultsHybrid*)

Update profile plot with result of curent iteration

out3Plot.outContours

Plotting and saving contour line plots

Functions

createRasterContourDict	create a dict with contour line coordinates for an ascii file
plotContoursFromDict	plot contour lines of two contourLine dicts only plot lines that are available within ref and save to file
readShpLines	derive a dict with x, y coordinates for each contourline according to given contour lines dependent on contLine name format here used resType_unit_level (for example: pft_m_1.0)

out3Plot.outContours.createRasterContourDict

createRasterContourDict(*inFile*, *levels*)

create a dict with contour line coordinates for an ascii file

Parameters

- **inFile** (*pathlib path*) – path to ascii file
- **levels** (*list*) – list of levels for contour lines

Returns

contourDict –

dictionary with keys: name of contourlines (format: resType_unit_level)
x: x coordinates y: y coordinates contourlevel: float of contour line value

Return type
dict

out3Plot.outContours.plotContoursFromDict

plotContoursFromDict(*contourDictRef*, *contourDictSim*, *pathDict*, *levels*, *multiplePlots=True*)

plot contour lines of two contourLine dicts only plot lines that are available within ref and save to file

Parameters

- **contourDict** (*dict*) – dictionary with contourline coordinates for reference
- **contourDict** (*dict*) – dictionary with contourline coordinates for simulation
- **pathDict** (*dict*) – dictionary with info on avaDir, outDir, title, parameter, unit
- **levels** (*list*) – list of contour levels
- **multiplePlots** (*bool*) – if True one plot for each contour level

out3Plot.outContours.readShpLines

readShpLines(*shpData*, *layerName='name'*)

derive a dict with x, y coordinates for each contourline according to given contour lines dependent on contLine name format here used resType_unit_level (for example: pft_m_1.0)

Parameters

- **shpData** (*dict*) – shp file data extracted with SHP2Array
- **layerName** (*str*) – name that is used to name contourline and to identify lines in shpData

Returns

contourDict – contour line dictionary with name and in dict x, y coordinates

Return type

dict

out3Plot.outDebugPlots

Functions

analysisPlots	create analysis plots during simulation run
plotAreaDebug	
plotBondsSnowSlideFinal	With snowSlide option on, plot the bonds between particles as well as the particles properties
plotBufferRelease	plot release lines with added bufferLine
plotContours	
plotFindAngle	helper plot for the getSplitPoint, findAngleProfile and prepareAngleProfile functions Plots the slope angle and elevation function of s
plotPartAfterRemove	
plotPartIni	
plotPathExtBot	Plot the extended path towards the bottom of the avalanche
plotPathExtTop	Plot the extended path towards the top of the release
plotPosition	
plotProfile	plot profile, add beta info
plotRemovePart	
plotSlopeAngle	plot slope angle along a profile, add beta info
plotVolumeRelease	create a plot of the release line raster, the relThField for release thickness, releaseLineField - combination of relThField and release line raster mask

out3Plot.outDebugPlots.analysisPlots

analysisPlots(*particlesList*, *fieldsList*, *cfg*, *demOri*, *dem*, *outDir*)

create analysis plots during simulation run

out3Plot.outDebugPlots.plotAreaDebug

plotAreaDebug(*header*, *avapath*, *Raster*)

out3Plot.outDebugPlots.plotBondsSnowSlideFinal

plotBondsSnowSlideFinal(*cfg*, *particles*, *dem*, *inputSimLines=''*)

With snowSlide option on, plot the bonds between particles as well as the particles properties

out3Plot.outDebugPlots.plotBufferRelease

plotBufferRelease(*inputSimLines*, *xBuffered*, *yBuffered*)

plot release lines with added bufferLine

out3Plot.outDebugPlots.plotContours

plotContours(*fig*, *ax*, *t*, *header*, *data*, *Cmap*, *unit*, *last=False*)

out3Plot.outDebugPlots.plotFindAngle

plotFindAngle(*avaProfile*, *angleProf*, *parabolicProfile*, *anglePara*, *s0*, *sEnd*, *splitPoint*, *indSplitPoint*)

helper plot for the getSplitPoint, findAngleProfile and prepareAngleProfile functions Plots the slope angle and elevation function of s

out3Plot.outDebugPlots.plotPartAfterRemove

plotPartAfterRemove(*points*, *xCoord0*, *yCoord0*, *mask*)

out3Plot.outDebugPlots.plotPartIni

plotPartIni(*particles*, *dem*)

out3Plot.outDebugPlots.plotPathExtBot

plotPathExtBot(*profile*, *xInterest*, *yInterest*, *zInterest*, *xLast*, *yLast*)

Plot the extended path towards the bottom of the avalanche

out3Plot.outDebugPlots.plotPathExtTop

plotPathExtTop(*profile*, *particlesIni*, *xFirst*, *yFirst*, *zFirst*, *dz1*)

Plot the extended path towards the top of the release

out3Plot.outDebugPlots.plotPosition

plotPosition(*fig*, *ax*, *particles*, *dem*, *data*, *Cmap*, *unit*, *plotPart=False*, *last=False*)

out3Plot.outDebugPlots.plotProfile

plotProfile(*s*, *z*, *idsBetaPoint*)

plot profile, add beta info

out3Plot.outDebugPlots.plotRemovePart

plotRemovePart(*xCoord0*, *yCoord0*, *header*, *X*, *Y*, *Mask*, *mask*)

out3Plot.outDebugPlots.plotSlopeAngle

plotSlopeAngle(*s*, *angle*, *idsBetaPoint*)

plot slope angle along a profile, add beta info

out3Plot.outDebugPlots.plotVolumeRelease

plotVolumeRelease(*releaseLine*, *relThField*, *releaseLineField*)

create a plot of the release line raster, the relThField for release thickness, releaseLineField - combination of relThField and release line raster mask

out3Plot.outDistanceTimeAnalysis

functions to plot range-time diagrams

Functions

addRangeTimePlotToAxes	add plot range-time diagram with avalanche front and colorcoded average values of result parameter
addTitleBox	add velocity legend title and boundary box and diagonal lines if maxVel=True add max velocity line in red
addVelocityValues	add velocity values as labels
animationPlot	3 panel plot: result in x,y, result in s, l, tt diagram
getMaxVelocityPoint	get point of max approach velocity in axes coordinates
getVelocityPoints	get points for legend creation (box and sloping lines)
plotMaskForMTI	plot masks used to compute averages for range-time diagram with radar field of view
plotRangeRaster	create plot of distance from start of runout area point extracted avalanche front and result field
plotRangeTime	plot range-time diagram with avalanche front and colorcoded average values of result parameter
radarFieldOfViewPlot	Create radar field of view plot
rangeTimeVelocityLegend	set legend in range time diagram for velocity in terms of steepness of front position

out3Plot.outDistanceTimeAnalysis.addRangeTimePlotToAxes

addRangeTimePlotToAxes(*mtiInfo*, *cfgRangeTime*, *ax*)

add plot range-time diagram with avalanche front and colorcoded average values of result parameter

Parameters

- **mtiInfo** (*dict*) – dictionary with average values for range to reference point (mti), timeStep list, list with distance to reference point of avalanche front (rangeList)
- **cfgRangeTime** (*configparser object*) – configuration settings for range time diagram - here used avalancheDir, rangeTimeResType, simHash, and from plots width, height, lw, ..
- **ax** (*matplotlib axes object*) – axes where plot shall be added to

out3Plot.outDistanceTimeAnalysis.addTitleBox

addTitleBox(*ax*, *width*, *height*, *lw*, *point*, *textsize*, *maxVel=True*)

add velocity legend title and boundary box and diagonal lines if maxVel=True add max velocity line in red

Parameters

- **ax** (*matplotlib axes object*) – axes object for legend
- **width** (*float*) – fractional percentage of legend width
- **height** (*float*) – fractional percentage of legend height
- **lw** (*float*) – linewidth
- **point** (*list*) – list of points coordinates for velocity legend
- **textsize** (*float*) – size of legend entries
- **maxVel** (*bool*) – if True add max velocity line in red

out3Plot.outDistanceTimeAnalysis.addVelocityValues

addVelocityValues(*ax*, *dataPoint*, *point*, *first=True*, *inititalNrOfTextElements=numpy.nan*)

add velocity values as labels

Parameters

- **ax** (*matplotlib axis object*)
- **dataPoint** (*list*) – list of points coordinates for velocity label locations
- **point** (*list*) – list of points coordinates for velocity lines
- **first** (*bool*) – if True text elements are created - initially required, if False then only changed
- **inititalNrOfTextElements**

out3Plot.outDistanceTimeAnalysis.animationPlot

animationPlot(*demData*, *data*, *cellSize*, *resType*, *cfgRangeTime*, *mtiInfo*, *timeStep*)

3 panel plot: result in x,y, result in s, l, tt diagram

Parameters

- **demData** (*dict*) – info on dem with header and rasterData
- **data** (*numpy array*) – result type data
- **cellSize** (*float*) – cell size of result type and dem
- **resType** (*str*) – name of result type, e.g. FV, FT
- **cfgRangeTime** (*configparser object*) – configuration of range time diagram settings
- **mtiInfo** (*dict*) – info dictionary for color plot of tt diagram, also info on domain transformation into s,l
- **timeStep** (*float*) – actual time step of sim result

out3Plot.outDistanceTimeAnalysis.getMaxVelocityPoint

getMaxVelocityPoint(*width*, *height*, *maxVel*, *diagVel*)

get point of max approach velocity in axes coordinates

Parameters

- **width** (*float*) – fractional percentage of legend width
- **height** (*float*) – fractional percentage of legend height
- **maxVel** (*float*) – maximum approach velocity value
- **diagVel** (*float*) – x/y ratio of axes

Returns
point – x, y axes coordinates of maximum velocity

Return type
tuple

out3Plot.outDistanceTimeAnalysis.getVelocityPoints

`getVelocityPoints`(*xMax*, *yMax*, *width*, *height*, *xinterval*, *yinterval*)

get points for legend creation (box and sloping lines)

if xMax, yMax, xinterval, yinterval are all 1 -> points refer to axes coordinates, that range from 0 to 1

if given true values, e.g. in seconds and meters, points are in data coordinates and can be used for velocity calculus etc.

Note: used to be an internal function to rangeTimeVelocityLegend,
so use with care.

Parameters

- **xMax** (*float*) – max x extent of data (timeSteps)
- **yMax** (*float*) – max y extent of data (rangeGates)
- **width** (*float*) – fractional percentage of legend width
- **height** (*float*) – fractional percentage of legend height
- **xinterval** (*float*) – interval on x
- **yinterval** (*float*) – interval on y

Returns
points – list of point coordinates

Return type
list

out3Plot.outDistanceTimeAnalysis.plotMaskForMTI

`plotMaskForMTI`(*cfgRangeTime*, *bmaskRange*, *bmaskAvaRadar*, *bmaskAvaRadarRangeslice*, *mtiInfo*)

plot masks used to compute averages for range-time diagram with radar field of view

Parameters

- **cfgRangeTime** (*configparser settings*) – configuration settings for range time diagram, avalancheDir
- **bmaskRange** (*numpy mask*) – mask for radar range slice for range gate and rgWidth
- **bmaskAvaRadar** (*numpy mask*) – mask for result parameter below threshold and out of radar field of view
- **bmaskAvaRadarRangeslice** (*numpy mask*) – full mask combination

out3Plot.outDistanceTimeAnalysis.plotRangeRaster

`plotRangeRaster`(*slRaster*, *rasterTransfo*, *cfgRangeTime*, *rangeRaster*, *cLower*)

create plot of distance from start of runout area point extracted avalanche front and result field

Parameters

- **slRaster** (*numpy array*) – transformed result field
- **rasterTransfo** (*dict*) – info on coordinate transformation
- **cfgRangeTime** (*configparser object*) – configuration settings of range time diagram
- **rangeRaster** (*np array*) – distance to runout area point in s coordinate
- **cLower** (*int*) – index of avalanche front

out3Plot.outDistanceTimeAnalysis.plotRangeTime

`plotRangeTime`(*mtiInfo*, *cfgRangeTime*)

plot range-time diagram with avalanche front and colorcoded average values of result parameter

Parameters

- **mtiInfo** (*dict*) – dictionary with average values for range to reference point (mti), timeStep list, list with distance to reference point of avalanche front (rangeList)
- **cfgRangeTime** (*configparser object*) – configuration settings for range time diagram - here used avalancheDir, rangeTimeResType, simHash, and from plots width, height, lw, ..

out3Plot.outDistanceTimeAnalysis.radarFieldOfViewPlot

`radarFieldOfViewPlot`(*radarFov*, *radarRange*, *cfgRangeTime*, *rangeGates*, *dem*)

Create radar field of view plot

Parameters

- **radarFov** (*numpy array*) – list with radar location and end point of field of view, x and y coors
- **radarRange** (*masked array*) – masked array of DEM with radar field of view - showing distance to radar
- **cfgRangeTime** (*configparser object*) –

  configuration settings section - here used avalancheDir, simHash,
  aperture angle [degree]
- **rangeGates** (*numpy array*) – range gates of radar field of view
- **dem** (*dict*) – dictionary with dem header and data

out3Plot.outDistanceTimeAnalysis.rangeTimeVelocityLegend

rangeTimeVelocityLegend(*ax*, *maxVel*, *width*, *height*, *lw*, *textsize*)

set legend in range time diagram for velocity in terms of steepness of front position
connects to figure callback to react to zoom/pan events

Parameters

- **ax** (*matplotlib axes object*) – figure axes object
- **maxVel** (*float*) – maximum approach velocity value
- **width** (*float , *0.25*) – fractional percentage of legend width
- **height** (*float*) – fractional percentage of legend height
- **lw** (*float*) – linewidth for legend line
- **textsize** (*float*) – textsize for legend entries

out3Plot.outParticlesAnalysis

Tools to extract information on the avalanche simulations run in the Output files

Functions

addPeakFieldConstrained	use out1Peak functions to add plot of a peak field
addTrOrMe	add a line plot of x: prop1, y: prop2 and label if label=True
plotParticleMotionTracking	Create plot showing particle properties over time and along avalanche thalweg in light blue envelope for all particles (filled between min and max values) in dark blue the values for tracked particles
plotParticleThalwegAltitudeVelocity	plot peak flow fields and velocity thalweg envelope
plotThalwegTimeAltitudes	Create plot showing the resType peak field with thalweg, thalweg vs altitude with max peak field values along thalweg derived from peak fields and the tt-diagram
readMeasuredParticleData	fetch data on measured particles from pickle in Inputs/measuredParticles only one pickle file allowed if no pData (name of particle file) is provided
velocityEnvelope	compute the velocity envelope from particle values
velocityEnvelopeThalweg	function to generate the velocity envelope along the thalweg from simulated flows

out3Plot.outParticlesAnalysis.addPeakFieldConstrained

addPeakFieldConstrained(*avaDir*, *modName*, *simName*, *resType*, *demData*, *ax*, *alpha*, *oneColor=''*)

use out1Peak functions to add plot of a peak field

Parameters

- **avaDir** (*pathlib path*) – path to avalanche dir
- **modName** (*str*) – name of computational module used to produce result peak fields
- **simName** (*str*) – name of simulation
- **resType** (*str*) – result variable name to look for peak field
- **demData** (*dict*) – dict with info on dem used to run sim of peak field data
- **ax** (*matplotlib axes object*) – axes where plot shall be added to
- **alpha** (*float*) – value for transparency from 0-1
- **oneColor** (*empty str*) – optional to add a color for a single color for field

out3Plot.outParticlesAnalysis.addTrOrMe

addTrOrMe(*ax*, *pDict*, *prop1*, *prop2*, *cmap*, *label=False*, *zorder=1*, *lineStyle='-'*)

add a line plot of x: prop1, y: prop2 and label if label=True

Parameters

- **ax** (*matplotlib axes*) – axes where plot should be added to
- **prop1** (*str*) – name of property in pDict used for x axis
- **prop2** (*str*) – name of property in pDict used for y axis
- **cmap** (*matplotlib colormap*) – cmap to be used for multiple particles in pDict
- **label** (*bool*) – if True add label to lines for legend
- **zorder** (*int*) – order of the plot object on the axes

out3Plot.outParticlesAnalysis.plotParticleMotionTracking

plotParticleMotionTracking(*avalancheDir*, *simName*, *dictVelAltThalweg*, *trackedPartProp*, *dictVelEnvelope*, *demSim*, *modName*, *rasterTransfo*, *measuredData=''*)

Create plot showing particle properties over time and along avalanche thalweg in light blue envelope for all particles (filled between min and max values) in dark blue the values for tracked particles

panel 1: map view of flow variable peak field panel 2: particle trajectoryLengthXYZ vs time panel 3: particle velocityMagnitude vs time panel 4: particle acceleration vs time panel 5: particle trajectoryLengthXYZ vs thalweg Sxy panel 6: particle velocity vs thalweg Sxy panel 7: particle acceleration vs thalweg Sxy

Parameters

- **avalancheDir** (*pathlib path or str*) – path to avalanche directory
- **simName** (*str*) – name of simulation
- **dictVelAltThalweg** (*dict*) – dict with velocity and altitude envelope info for all particles
- **trackedPartProp** (*dict*) – dict with time series of tracked particle properties

- **dictVelEnvelope** (*dict*) – dict with velocity envelope info
- **demSim** (*dict*) – dict with sim dem info
- **modName** (*str*) – name of computational module that has been used to produce the sims
- **measuredData** (*dict*) –

  dict data: with time series of measured data (same format as trackedPartProp)
  labelName: str name of label for plots t: 1D array of time steps

out3Plot.outParticlesAnalysis.plotParticleThalwegAltitudeVelocity

`plotParticleThalwegAltitudeVelocity`(*avalancheDir*, *simIndex*, *simDF*, *rasterTransfo*, *dictVelAltThalweg*, *resTypePlots*, *modName*, *demData*)

plot peak flow fields and velocity thalweg envelope

Parameters

- **avalancheDir** (*pathlib path or str*) – path to avalanche directory
- **simIndex** (*str*) – index of current simulation
- **rasterTransfo** (*dict*) – info on domain transformation from xy to sl thalweg
- **dictVelAltThalweg** (*dict*) – info on velocity and altitude of particles along thalweg s
- **resTypePlots** (*list*) – list of result types that shall be plotted, one plot for each resType
- **modName** (*str*) – name of com module used to perform sims
- **demData** (*dict*) – dict of dem from sim

out3Plot.outParticlesAnalysis.plotThalwegTimeAltitudes

`plotThalwegTimeAltitudes`(*avalancheDir*, *simIndex*, *simDF*, *rasterTransfo*, *pfvCrossMax*, *modName*, *demSim*, *mtiInfo*, *cfgRangeTime*, *velocityThreshold*, *measuredData=''*)

Create plot showing the resType peak field with thalweg, thalweg vs altitude with max peak field values along thalweg derived from peak fields and the tt-diagram

Parameters

- **avalancheDir** (*str or pathlib path*) – path to avalanche directory
- **simIndex** (*str*) – index of current sim in simDF
- **simDF** (*dataFrame*) – dataframe with one row per simulation, and all model config parameters
- **rasterTransfo** (*dict*) – dict with info on transformation from cartesian to thalweg coordinate system
- **dictRaster** (*dict*) – dict with info on peak fields
- **modName** (*str*) – name of com module used to perform sims
- **demSim** (*dict*) – dict with info on dem used for sims
- **measuredData** (*dict*) –

  dict data: with time series of measured data (same format as trackedPartProp)
  labelName: str name of label for plots t: 1D array of time steps

out3Plot.outParticlesAnalysis.readMeasuredParticleData

`readMeasuredParticleData`(*avalancheDir*, *demHeader*, *pData=''*)

fetch data on measured particles from pickle in Inputs/measuredParticles only one pickle file allowed if no pData (name of particle file) is provided

Parameters

- **avalancheDir** (*pathlib path*) – path to avalanche directory
- **demHeader** (*dict*) – currently xllcenter, yllcenter required to set x, y coordinates to origin 0,0
- **pData** (*str*) – name of mesured particle file

Returns

mParticles – dict with info on measured particles properties (veloctiyMag, x, y, z, uAcc, t) all properties except t are of shape: mxn matrix, where m refers to the time steps and n to the individual particles t is a vector of the time step values corresponding to m label is a list of names of the labels for the measured particles corresponding to n

Return type

dict

out3Plot.outParticlesAnalysis.velocityEnvelope

`velocityEnvelope`(*particlesTimeArrays*)

compute the velocity envelope from particle values

Parameters

particlesTimeArrays (*dict*) – dict with time series for particle properties

Returns

dictVelEnvelope – max, mean, min values of velocity over all particles for each time step velocity, acceleration, min and max of trajectoryLengthXYZ of particles for each time step

Return type

dict

out3Plot.outParticlesAnalysis.velocityEnvelopeThalweg

`velocityEnvelopeThalweg`(*particlesTimeArrays*)

function to generate the velocity envelope along the thalweg from simulated flows

Parameters

particlesTimeArrays (*dict*) – dict with time series of particle properties

Returns

dictVelAltThalweg – dict with min, max, mean, … values of particles elevation, velocity and acceleration for each thalweg coordinate

Return type

dict

out3Plot.outQuickPlot

Functions to plot 2D avalanche simulation results as well as comparison plots between two datasets of identical shape.

Functions

generateOnePlot	Create plot of ascii dataset
generatePlot	Create comparison plots of two ascii datasets
plotContours	plot contour lines of all transformed fields
quickPlotBench	Plot simulation result and compare to reference solution (two raster datasets of identical dimension) and save to Outputs/out3Plot within avalanche directoy.
quickPlotOne	Plots one raster dataset and a cross profile
quickPlotSimple	Plot two raster datasets of identical dimension and difference between two datasets

out3Plot.outQuickPlot.generateOnePlot

generateOnePlot(*dataDict*, *outDir*, *cfg*, *plotDict*)

Create plot of ascii dataset

Parameters

- **dataDict** (*dict*) – dictionary with info of the dataset to be plotted
- **outDir** (*pathlib path*) – path to dictionary where plots shall be saved to
- **cfg** (*dict*) – main configuration settings
- **plotDict** (*dict*) – dictionary with information about plots, for example release area…

Returns

plotDict – updated plot dictionary with path to plot

Return type

dict

out3Plot.outQuickPlot.generatePlot

generatePlot(*dataDict*, *avaName*, *outDir*, *cfg*, *plotDict*, *crossProfile=True*)

Create comparison plots of two ascii datasets

This function creates two plots, one plot with four panels with, first dataset, second dataset, the absolute difference of the two datasets and the absolute difference capped to a smaller range of differences (ppr: +- 100kPa, pft: +- 1m, pfv:+- 10ms-1). The difference plots also include an insert showing the histogram and the cumulative density function of the differences. The second plot shows a cross- and along profile cut of the two datasets. The folder and simulation name of the datasets has to be passed to the function.

Parameters

- **dataDict** (*dict*) –

  dictionary with info on both datasets to be plotted:

name1: str
string with the name of the first data set

name2: str
string with the name of the second data set

data1: 2D numpy arrays
raster of the first data sets

data2: 2D numpy arrays
raster of the first second sets

cellSize: float
cell size

suffix: str
optional information about the data type compared ('ppr', 'pft', 'pfv', 'P', 'FV', 'FT', 'Vx'…)

- **avaName** (*str*) – name of avalanche
- **outDir** (*pathlib path*) – path to dictionary where plots shall be saved to
- **cfg** (*configParser*) – cfg['FLAGS'].getboolean('showPlot')
- **plotDict** (*dict*) – dictionary with information about plots, for example release area

Returns
plotDict – updated plot dictionary with info about e.g. min, mean, max of difference between datasets

Return type
dict

out3Plot.outQuickPlot.plotContours

`plotContours`(*contourDict*, *resType*, *thresholdValue*, *pathDict*, *addLegend=True*)

plot contour lines of all transformed fields

Parameters

- **contourDict** (*dict*) – dictionary with contourline coordinates
- **resType** (*str*) – result type
- **thresholdValue** (*float*) – value for contour level
- **pathDict** (*dict*) – dictionary with info on project name, …
- **addLegend** (*bool*) – if True add legend to plot

out3Plot.outQuickPlot.quickPlotBench

quickPlotBench(*avaDir*, *simNameRef*, *simNameComp*, *refDir*, *compDir*, *cfg*, *suffix*)

Plot simulation result and compare to reference solution (two raster datasets of identical dimension) and save to Outputs/out3Plot within avalanche directoy.

figure 1, plot raster data for dataset1, dataset2 and their difference, their difference limited to specified range, including a histogram and the cumulative density function of the differences

figure 2, plot cross and longprofiles for both datasets (ny_loc and nx_loc define location of profiles)

plots are saved to Outputs/out3Plot

Parameters

- **avaDir** (*str or pathlib path*) – path to avalanche directory
- **simNameRef** (*str*) – name of reference simulation
- **simNameComp** (*str*) – name of comparison simulation
- **refDir** (*str or pathlib path*) – path to reference file
- **compDir** (*str or pathlib path*) – path to comparison file
- **cfg** (*dict*) – global configuration settings
- **suffix** (*str*) – result type

Returns

plotList – plot dictionaries (path to plots, min, mean and max difference between plotted datasets, max and mean value of reference dataset)

Return type

dict

out3Plot.outQuickPlot.quickPlotOne

quickPlotOne(*avaDir*, *datafile*, *cfg*, *locVal*, *axis*, *resType=''*)

Plots one raster dataset and a cross profile

figure 1: plot raster data for dataset and profile -plot is saved to Outputs/out3Plot

Parameters

- **avaDir** (*str or pathlib Path*) – path to avalanche directory
- **datafile** (*str or pathlib path*) – path to data file
- **cfg** (*dict*) – configuration including flags for plotting
- **locVal** (*float*) – location of cross profile
- **resType** (*str*) – result parameter type e.g. 'pft' - optional

out3Plot.outQuickPlot.quickPlotSimple

quickPlotSimple(*avaDir*, *inputDir*, *cfg*)

Plot two raster datasets of identical dimension and difference between two datasets

figure 1: plot raster data for dataset1, dataset2 and their difference figure 2: plot cross and longprofiles for both datasets (ny_loc and nx_loc define location of profiles) -plots are saved to Outputs/out3Plot

Be aware: files are being sorted after getting them from the directory! (Important for the differences)

Parameters

- **avaDir** (*str or pathlib path*) – path to avalanche directory
- **inputDir** (*str or pathlib path*) – path to directory of input data (only 2 raster files allowed)
- **cfg** (*configParser object*) – global configuration settings

out3Plot.outTopo

Simple plotting for DEMs

Functions

plotDEM3D	Plots the DEM from the avalancheDir in cfg alongside it
plotGeneratedDEM	Plot DEM with given information on the origin of the DEM
plotReleasePoints	

out3Plot.outTopo.plotDEM3D

plotDEM3D(*cfg*, *showPlot=False*)

Plots the DEM from the avalancheDir in cfg alongside it

Parameters

- **cfg** (*configparser object*) – the main configuration
- **showPlot** (*boolean*) – If true shows the matplotlib plot

out3Plot.outTopo.plotGeneratedDEM

plotGeneratedDEM(*z*, *nameExt*, *cfg*, *outDir*, *cfgMain*)

Plot DEM with given information on the origin of the DEM

out3Plot.outTopo.plotReleasePoints

plotReleasePoints(*xv*, *yv*, *xyPoints*, *demType*)

out3Plot.statsPlots

plot statistics of simulations

Functions

plotDistFromDF	create a dist plot from dataframe for name1 on x axis and name2 on y axis, optionally colorcoded with scenario name and filtered with parametersDict
plotHistCDFDiff	Produce histogram and CDF plot of the raster difference of two simulations
plotProbMap	plot probability maps including contour lines
plotSample	plot the parameter sample only if two parameters are varied
plotThSample	plot the parameter sample if generated for th thickness values read from shp file one plot for each release Scenario using simDF
plotThSampleFromVals	plot the parameter sample if generated for thickness values read from shp file one plot for each release Scenario - hence paramValuesD should be for one release scenario only
plotValuesScatter	Produce scatter plot of statistical measures (eg.
plotValuesScatterHist	Produce scatter and marginal kde plot of max values, for one set of simulations or multiple
resultHistPlot	create a histogram of values and optional colorcode using scenario name and option to filter simulations using parametersDict

out3Plot.statsPlots.plotDistFromDF

plotDistFromDF(*cfg*, *dataDF*, *name1*, *name2*, *scenario=''*, *parametersDict=''*, *type=''*)

create a dist plot from dataframe for name1 on x axis and name2 on y axis, optionally colorcoded with scenario name and filtered with parametersDict

Parameters

- **cfg** (*configparser object*) – configuration settings here outDir
- **dataDF** (*dataframe*) – dataframe with one line per simulation and info on model parameters and results
- **name1** (*str*) – column name of dataDF to use for plot x axis
- **name2** (*str*) – column name of dataDF to use for plot y axis
- **scenario** (*str*) – optional name of column used to colorcode points in plots for type=scatter or kde for type=dist
- **parametersDict** (*dict*) – optional - dictionary filter criteria

- **type** (*str*) – optional - type of plot dist or scatter

Returns
plotPath – path to figure

Return type
pathlib path

out3Plot.statsPlots.plotHistCDFDiff

`plotHistCDFDiff`(*dataDiffPlot*, *ax1*, *ax2*, *insert='True'*, *title=['', '']*)

Produce histogram and CDF plot of the raster difference of two simulations

Parameters

- **dataDiffPlot** (*2D numpy array*) – raster of the difference of the two simulations
- **ax1** (*axes*) – axes for the histogram plot
- **ax2** (*axes*) – axes for the CDF plot
- **insert** (*boolean*) – true if the plots are in inserted axes (size of the lables is then smaller)
- **title** (*list*) – if not inserts, title for the plots

out3Plot.statsPlots.plotProbMap

`plotProbMap`(*avaDir*, *inDir*, *cfgFull*, *demPlot=False*)

plot probability maps including contour lines

Parameters

- **avaDir** (*str*) – path to avalanche directory
- **inDir** (*str*) – path to datasets that shall be plotted
- **cfgFull** (*configParser object*) – configuration settings for probAna keys used: name, cmapType, levels, unit
- **demPlot** (*bool*) – if True plot dem in background with contourlines for elevation that is found in avaDir/Inputs

out3Plot.statsPlots.plotSample

`plotSample`(*paramValuesD*, *outDir*, *releaseScenario=''*)

plot the parameter sample only if two parameters are varied

Parameters

- **paramValuesD** (*dict*) – dictionary with parameter names and sets of values
- **outDir** (*pathlib path*) – path where to save the plot

out3Plot.statsPlots.plotThSample

plotThSample(*simDF*, *name1*, *thName*, *outDir*)

plot the parameter sample if generated for th thickness values read from shp file one plot for each release Scenario using simDF

Parameters

- **simDF** (*pandas DataFrame*) – dataframe with one row per sim and info on sim configuration
- **name1** (*str*) – name of parameter that is varied for x-axis
- **thName** (*str*) – name of the thickness parameter (relTh, entTh, secondaryRelTh)
- **outDir** (*pathlib path*) – path to folder where to save plot

out3Plot.statsPlots.plotThSampleFromVals

plotThSampleFromVals(*paramValuesD*, *outDir*)

plot the parameter sample if generated for thickness values read from shp file one plot for each release Scenario - hence paramValuesD should be for one release scenario only

Parameters

- **paramValuesD** (*dict*) – dict with info on parameter variation and initial values of parameters
- **outDir** (*pathlib path*) – path to folder for saving plot

out3Plot.statsPlots.plotValuesScatter

plotValuesScatter(*peakValues*, *resType1*, *resType2*, *cfg*, *avalancheDir*, *statsMeasure='max'*, *flagShow=False*)

Produce scatter plot of statistical measures (eg. max values) (resType1 und resType2), for one set of simulations or multiple

Parameters

- **peakDictList** (*dict*) – peakValues dictionary that contain max values of peak parameters and parameter variation info
- **resType1** (*str*) – result parameter 1, 'ppr', 'pft', 'pfv'
- **resType2** (*str*) – result parameter 1, 'ppr', 'pft', 'pfv'
- **cfg** (*dict*) – configuration, for now contains output location and varPar: parameter that is varied to perfom a set of simulations
- **statsMeasure** (*str*) – statistical measure for plotting, options: max, mean, min, std
- **flagShow** (*bool*) – if True show plot

out3Plot.statsPlots.plotValuesScatterHist

`plotValuesScatterHist`(*peakValues*, *resType1*, *resType2*, *cfg*, *avalancheDir*, *statsMeasure='max'*, *flagShow=False*, *flagHue=False*)

Produce scatter and marginal kde plot of max values, for one set of simulations or multiple

Parameters

- **peakValues** (*dict*) – peakValues dictionary that contain max values of peak parameters and parameter variation info
- **resType1** (*str*) – result parameter 1, 'ppr', 'pft', 'pfv'
- **resType2** (*str*) – result parameter 1, 'ppr', 'pft', 'pfv'
- **cfg** (*dict*) – configuration, for now contains output location and varPar: parameter that is varied to perfom a set of simulations
- **statsMeasure** (*str*) – statistical measure for plotting, options: max, mean, min, std
- **flagShow** (*bool*) – if True show plot

out3Plot.statsPlots.resultHistPlot

`resultHistPlot`(*cfg*, *dataDF*, *xName=''*, *scenario=''*, *stat='count'*, *parametersDict=''*)

create a histogram of values and optional colorcode using scenario name and option to filter simulations using parametersDict

Parameters

- **cfg** (*configparser object*) – configuration info here used outDir
- **dataDF** (*dataFrame*) – dataFrame with info on simulation results and configuration one line per simulation (e.g. aimec resAnalysisDF)
- **xName** (*str*) – column name for x axis
- **scenario** (*str*) – column name used to colorcode values
- **stat** (*str*) – statistical measure to show (percent, probability, density, count, frequency), default count
- **parametersDict** (*dict*) – optional - dictionary filter criteria, parameter name and list of values

Returns
plotPath – path to figure

Return type
pathlib path

2.10.3.8 tmp1Ex.tmp1Ex

This is the template for new modules, with the bare minimal required files

Functions

tmp1ExMain	Main function for module tmp1Example

tmp1Ex.tmp1Ex.tmp1ExMain

tmp1ExMain(*cfg*)

Main function for module tmp1Example

Parameters

- **foo** (*int, float, str, or tf.Tensor*) – The foo to bar, which has a really really, reeeeeeeeeeeeeeeeeally unnecessarily long multiline description.
- **bar** (*str*) – Bar to use on foo
- **baz** (*float*) – Baz to frobnicate

Returns
The frobnicated baz

Return type
float

2.11 Development

2.11.1 Notes to developers

Here you can find notes on design principles and how to contribute to AvaFrame.

First a few general remarks. We are aware there's no right or wrong here, we decided to stick to a few principles. If you contribute code, please try to observe these:

AF-DESIGN-1: Use SI units
Units in APIs should be SI units: seconds, meters, etc.

AF-STYLE-1: Line length
If you want a guide for line length — we think line breaking at 120 characters makes more sense than line breaking at 80.

AF-STYLE-2: Naming
Use short, but descriptive enough names for variables. Use longer descriptive names for things with bigger scopes (functions, modules). Main theme: as short as possible, but as long as necessary.

Please use lowerCamelCase (e.g.: myFancyVariable) for naming, avoid underscores.

AF-STYLE-3: Commenting
Exposed API functions need documentation comments, most importantly a docstring, other code only needs to be commented as necessary. Write your code clearly and use sensible names to reduce the need for comments.

AF-STYLE-4: No dead code
Unused code, commented out code, functions that are never called, etc, should be removed from the project to reduce the cognitive load of reading the source code. Old code is available in the source history if it is needed.

AF-STYLE-5: Use spaces for indentation
Use 4 spaces as indentation throughout the python code.

2.11.2 Our suggested git workflow

Clone repository:

```
git clone https://github.com/avaframe/AvaFrame.git
```

Clones the repository to your local machine into the directory AvaFrame. Sets the repository to track to *origin*

Branch:

```
git checkout -b myAwesomeFeature
```

This changes your working directory to the myAwesomeFeature branch. Try to keep any changes in this branch specific to one bug or feature. You can have many branches and switch in between them using the git checkout command.

Work on it and from time to time commit your changes using following commands as necessary:

```
git add
git commit
```

To update this branch, you need to retrieve the changes from the master branch:

```
git rebase origin master
```

or:

```
git checkout master
git pull
git checkout myAwesomeFeature
git rebase master
```

This replays all your changes on the current status of the master (i.e main) branch. If conflicts arise, now is the time to solve them.

Push your changes to the main repository:

```
git push origin
```

Once you feel you are done, start a pull request on github.com.

Pull requests are reviewed and handled. Once the pull request is included into the master, the local myAwesomeFeature branch can be deleted (the one in the main repository/origin will be handled by the pull request):

```
git checkout master
git branch -d myAwesomeFeature
```

2.11.3 Build the documentation

If you want to work on the documentation you need to install *Sphinx*. If you have followed the conda installation using `avaframe_env_spec.txt`, you can omit the following steps. If not, you can install Sphinx, the *ReadTheDocs* theme, and the *sphinxcontrib-bibtex*, which we use to include references, by running:

```
pip install sphinx
pip install sphinx-rtd-theme
pip install sphinxcontrib-bibtex
```

In order to build the documentation you need to install make

```
sudo apt install make
```

Then go to the `docs\` directory and run:

```
make html
```

Html files of the documentation can be found in the _build directory.

2.11.4 How to test code

AvaFrame uses pytest to test code. If you add new code, consider including a pytest for it in `Avaframe/avaframe/tests/`. In order to perform the pytests, just run:

```
pytest
```

and you should see something like:

```
============================test session starts ======================
platform linux -- Python 3.8.3, pytest-5.4.3, py-1.9.0, pluggy-0.13.1
collected 1 item
tests/test_tmp1Ex.py .                                          [100%]

============================1 passed in 0.02s ========================
```

2.11.5 How to add a benchmark test

AvaFrame offers an expanding benchmark test suite. At the moment this test suite includes avalanche simulations for various idealised topographies. The `runStandardTestsCom1DFA.py` facilitates running all the available benchmark tests for com1DFA at once. With this script, the avalanche simulations are performed, plotted and a report of the comparison between simulation results and the benchmark data is generated. If you plan to add a new benchmark test case, follow these steps

- first chose a name, we suggest to start it with `ava` (for now let's refer to it as `NameOfAvalanche`)
- add all the required input data in `data/NameOfAvalanche`. Follow the required directory structure which can be generated using: *Initialize Project*

as a next step, you need to add the benchmark results:

- go to `AvaFrame/benchmarks` and add the subdirectory named after your test name

- add benchmark data i.e. peak values of result parameters as ascii files. This data will be used as reference for the new test!
- add the configuration file as `NameOfAvalanche_com1DFACfg.ini`
- add a json file with required info on benchmark test - you can use the example provided in `runScripts/runWriteDesDict.py`
- go to `AvaFrame/benchmarks/simParametersDict.py` and add a simulation dictionary that contains all the info on the new benchmark

Now, you are ready to go! Move to `AvaFrame/avaframe` and run:

```
python runStandardTestsCom1DFA.py
```

You can check out the markdown-style report of the comparison at: `tests/reports/standardTestsReportPy.md`.

CHAPTER

THREE

COMPUTATIONAL MODULES

3.1 com1DFA: DFA-Kernel

com1DFA is a module for dense flow (snow) avalanche computations (DFA) . It is a python and cython implementation of the DFA C++ implementation samosAT (Snow Avalanche Modeling and Simulation- Advanced Technologies) developed by the Austrian government in cooperation with the company AVL List GmbH in Graz (see *com1DFAOrig: Original DFA-Kernel*). Calculations are based on the thickness integrated governing equations and solved numerically using the smoothed particle hydrodynamics (sph) method. Please note the use of *thickness averaged/integrated* instead of *depth averaged/integrated* for clarity and consistency.

Dense flow avalanche simulations can be performed for different release area scenarios, with or without entrainment and/or resistance areas, and is controlled via a configuration file. The configration can be modified in order to change any of the default settings and also allows to perform simulations for varying parameters all at once.

Note: The configuration provided with com1DFA is well-tested and applied for hazard mapping (in Austria). If you change configuration parameters, be aware that unwanted/unexpected/spurious side-effects might appear. This is especially true if you switch to something far outside the intended range (i.e. changing density from snow to something like rock). Furthermore, be aware that the parameters are calibrated in connection, so changing one might necessitate also changing other connected parameters!

3.1.1 Input

DFA simulations are performed within an avalanche directory, organized with the folder structure described below.

Note: An avalanche directory can be created by running: `runInitializeProject.py`, which creates the required folder structure:

```
NameOfAvalanche/
  Inputs/
    REL/      - release area scenario
    RES/      - resistance areas
    ENT/      - entrainment areas
    POINTS/   - split points
    LINES/    - avalanche paths
    SECREL/   - secondary release areas
  Outputs/
  Work/
```

In the directory `Inputs`, the following files are required. Be aware that ALL inputs have to be provided in the same projection:

- digital elevation model as .asc file with ESRI grid format
- release area scenario as (multi-) polygon shapefile (in Inputs/REL; multiple features are possible)
  - the release area name should not contain an underscore, if so '_AF' is added.
  - recommended attributes are *name*, *thickness* (see *Release-, entrainment thickness settings*) and *ci95* (see *probAna - Probability maps*)
  - ALL features within one shapefile are released at the same time (and interact), this is what we refer to as *scenario*
  - if you want to simulate different scenarios with the same features, you have to copy them to separate shapefiles

and the following files are optional:

- one entrainment area (multi-) polygon shapefile (in Inputs/ENT)
  - marks the (multiple) areas where entrainment can occur.
  - attribute *thickness* (see *Release-, entrainment thickness settings*)
- one resistance area (multi-) polygon shapefile (in Inputs/RES)
  - marks the (multiple) areas where resistance is considered
- one secondary release area (multi-) polygon shapefile (in Inputs/SECREL)
  - can have multiple release areas, each as one feature
  - same setup as the release area scenario (see above)
  - features will release as soon as at least one particle enters its area

3.1.1.1 Release-, entrainment thickness settings

Release, entrainment and secondary release thickness can be specified in two different ways:

1. Via **shape file**:

- add an attribute called *thickness* for each feature
- important: ALL features have to have a single thickness value, which can differ between features
- for entrainment area only: if the thickness value is missing, the thickness value is taken from *entThIfMissingInShp* (default 0.3 m) in the configuration file
- for backwards compatibility, the attribute 'd0' also works, but we suggest to use *thickness* in new projects
- set the flag *THICKNESSFromShp* (i.e. relThFromShp, entThFromShp, secondaryRelthFromShp) to True in the configuration file (default is True)
- a parameter variation can be added with the *THICKNESSPercentVariation* parameter in the configuration file in the form of `+-percentage$numberOfSteps`. Provided a + a positive variation will be performed, if - is given, only a negative variation is performed. If no sign is given: both directions will be used. Additionally, a variation can be added with the *THICKNESSRangeVariation* parameter in the configuration file in the form of `+-range$numberOfSteps`. Provided a + a positive variation will be performed, if - is given, only a negative variation is performed. If no sign is given: both directions will be used. Furthermore, there is the option to vary the thickness in a range of +- the 95% confidence interval value, which is also read from the shape file

(requires an attribute called ci95). In order to use this variation, set the 'THICKESSRangeFromCiVariation' to `ci95$numberOfSteps`.

2. Via **configuration file (ini)**:

- set the flag 'THICKNESSFromShp' to False
- provide your desired thickness value in the respective THICKNESS parameter (i.e. relTh, entTh or secondaryRelth)
- in addition to the *THICKNESSPercentVariation* and *THICKNESSRangeVariation* options (see option 1) and the standard variation options in configuration:Configuration, you can also directly set e.g. *relTh = 1.$50$2*, `referenceValue$+-percentage$numberOfSteps`, resulting in a variation of relTh from 0.5 to 1.5m in two steps.

Only available for release thickness:

3. Via **release thickness file**:

- set the flag 'relThFromShp' to False
- set the flag 'relThFromFile' to True
- save a raster file with info on release thickness as .asc file in `Inputs/RELTH` the number of rows and columns must match the DEM raster with desired meshCellSize

3.1.1.2 DEM input data

Regarding the DEM data: if the DEM in `Inputs` is not of cell size 5 meters, it is remeshed to a cell size of 5 meters. However, it is also possible to specify a desired cell size in the configuration file (parameter *meshCellSize*). In this case, also consider reading *Can the spatial resolution of simulations performed with com1DFA (dense flow) be changed?*. If the cell size of the DEM in `Inputs` is equal to the desired mesh cell size, the DEM is used without modification. If the cell sizes do not match, several options are available:

- cleanDEMremeshed = True, directory `Inputs/DEMremeshed` is cleaned, and the DEM in Inputs/ is remeshed to the desired cell size - this is the default setting
- cleanDEMremeshed = False and a DEM including the name of the DEM in Inputs/ and the desired cell size is found in Inputs/DEMremeshed - this DEM is used without modification
- cleanDEMremeshed = False and no matching DEM is found in Inputs/DEMremeshed - the DEM in Inputs/ is remeshed to the desired cell size

If the DEM in Inputs/ is remeshed, it is then saved to `Inputs/DEMremeshed` and available for subsequent simulations.

3.1.1.3 Dam input

The com1DFA module provides the option to take the effect of dams into account. This is done using a ad-hoc method based on particles being reflected/deflected by a dam wall.

The dam is described by the crown line, the slope and the restitution coefficient:

- crown line as shape file (use the line type and enable the "additional dimensions" option in order to specify the z coordinate). The z coordinate corresponds to the absolute height (terrain elevation plus dam height). The dam is then located on the left side of the dam (when one travels from the first point to the last point of the shapefile line). The dam shape files live in the `avaDir/Inputs/DAM/` directory (only one file is allowed).
- the `slope` of the dam (in degrees °) between the horizontal plane and the wall to be provided in the shape file as an attribute (default value is 60° in the provided examples: avaSlide, avaKot and avaBowl)

- the restitution coefficient (α_{rest}), a float between 0 (no reflection in the normal direction) and 1 (full reflection) to be specified in the ini file (default value is 0)

3.1.2 Model configuration

The model configuration is read from a configuration file: `com1DFA/com1DFACfg.ini`. In this file, all model parameters are listed and can be modified. We recommend to create a local copy and keep the default configuration in `com1DFA/com1DFACfg.ini` untouched. For this purpose, in `AvaFrame/avaframe/` run:

```
cp com1DFA/com1DFACfg.ini com1DFA/local_com1DFACfg.ini
```

and modify the parameter values in there. For more information see configuration:Configuration.

It is also possible to perform multiple simulations at once, with varying input parameters.

3.1.3 Output

Using the default configuration, the simulation results are saved to: *Outputs/com1DFA* and include:

- raster files of the peak values for pressure, flow thickness and flow velocity (*Outputs/com1DFA/peakFiles*)
- raster files of the peak values for pressure, flow thickness and flow velocity for the initial time step (*Outputs/com1DFA/peakFiles/timeSteps*)
- markdown report including figures for all simulations (*Outputs/com1DFA/reports*)
- mass log files of all simulations (*Outputs/com1DFA*)
- configuration files for all simulations (*Outputs/com1DFA/configurationFiles*)

optional outputs

- pickles of particles properties (*Particle properties*.) for saving time steps if particles are added to the list of resTypes in your local copy of `com1DFACfg.ini`
- a csv file of specified particle properties for the saving time steps if particles are added to the list of resTypes in your local copy of `com1DFACfg.ini` and if in the VISUALISATION section writePartToCsv is set to True

However, in the configuration file, it is possible to change the result parameters and time Steps that shall be exported. The result types that can be chosen to be exported are (all correspond to fields except the particles):

- ppr - peak pressure
- pfv - peak flow velocity
- pft - peak flow thickness
- pta - peak travel angle
- FV - flow velocity
- FT - flow thickness
- P - pressure
- FM - flow mass
- Vx, Vy, Vz - velocity x-, y- and z-component
- TA - travel angle
- particles (*Particle properties*)

Have a look at the designated subsection Output in `com1DFA/com1DFACfg.ini`.

3.1.4 Parallel computation

If multiple runs of com1DFA are to be executed, these will be calulated in parallel via multiprocessing. So each task itself is calculated on only one core, but different tasks are run at the same time.

This happens if you have one of the following (or a combination of them):

- multiple scenarios (multiple input release shapefiles)
- multiple runtypes, i.e null variant and entrainment/resistance variant (e.g.: simTypeList = null|ent)
- some kind of parameter variation (e.g.: relTh = 1.0|1.5|1.7)

The number of CPU cores is controlled in the main `avaframeCfg.ini` file. By default a maximimum of 50 percent of your available cores is being utilized. However you can set a different number if needed. For sequential execution set nCPU to 1.

3.1.5 To run

- first go to `AvaFrame/avaframe`
- copy `avaframeCfg.ini` to `local_avaframeCfg.ini` and set your desired avalanche directory name
- create an avalanche directory with required input files - for this task you can use *Initialize Project*
- copy `com1DFA/com1DFACfg.ini` to `com1DFA/local_com1DFACfg.ini` and if desired change configuration settings
- if you are on a develop installation, make sure you have an updated compilation, see developinstall:Setup AvaFrame
- run:

```
python3 runCom1DFA.py
```

3.1.6 Theory

The governing equations of the dense flow avalanche are derived from the incompressible mass and momentum balance on a Lagrange control volume ([Zw2000] [ZwKlSa2003]). Assuming the avalanche is much longer and larger than thick, it is possible to integrate the governing equations over the thickness of the avalanche and operate some simplifications due to the shape of the avalanche. This leads, after some calculation steps described in details in Theory *Governing Equations for the Dense Flow Avalanche* to:

$$\frac{\mathrm{d}V(t)}{\mathrm{d}t} = \frac{\mathrm{d}(A_b\overline{h})}{\mathrm{d}t} = \frac{\rho_{\text{ent}}}{\rho_0}\, w_f\, h_{\text{ent}}\, \|\overline{\mathbf{u}}\|$$

$$\frac{\mathrm{d}\overline{u}_i}{\mathrm{d}t} = g_i + \frac{K_{(i)}}{\overline{\rho}\, A\, \overline{h}} \oint_{\partial A} \left(\frac{\overline{h}\, \sigma^{(b)}}{2} \right) n_i\, \mathrm{d}l - \delta_{i1} \frac{\tau^{(b)}}{\overline{\rho}\, \overline{h}} - C_{\text{res}}\, \overline{\mathbf{u}}^2\, \frac{\overline{u_i}}{\|\overline{\mathbf{u}}\|} - \frac{\overline{u_i}}{A\, \overline{h}} \frac{\mathrm{d}(A\, \overline{h})}{\mathrm{d}t} + \frac{F_i^{\text{ent}}}{\overline{\rho}\, A\, \overline{h}}$$

$$\overline{\sigma}_{33}^{(b)} = \rho \left(g_3 - \overline{u_1}^2\, \frac{\partial^2 b}{\partial x_1^2} \right) \overline{h}$$

3.1.7 Numerics

Those equations are solved numerically using a **SPH** method ([LL10, Sam07]). **SPH** is a mesh free method where the basic idea is to divide the avalanche into small mass particles. The particles interact with each other according to the equation of motion described in *Theory* and the chosen kernel function. This kernel function describes the domain of influence of a particle (through the smoothing length parameter). See theory *com1DFA DFA-Kernel theory* for further details.

3.2 com1DFAOrig: Original DFA-Kernel

`com1DFAOrig` is a simulation tool for dense flow (snow) avalanches (DFA). It is a python wrapper function calling the samos-AT C++ (Snow Avalanche Modeling and Simulation- Advanced Technologies) DFA simulation code developed by the Austrian government in cooperation with the company AVL List GmbH in Graz. This code served as base for the implementation of the python DFA simulation module *com1DFA* presented in *com1DFA: DFA-Kernel*. *com1DFA* module is now the default DFA simulation module in AvaFrame.

The calculation of the DFA is based on the depth integrated governing equations and solved numerically using the smoothed particle hydrodynamic (sph) method.

Dense flow avalanche simulations can be performed for different release area scenarios, with or without entrainment and/or resistance areas. There is the option to vary the internal friction parameter or the release snow thickness.

3.2.1 C++ Executable

The computation of the `com1DFAOrig` dense flow avalanche module relies on a C++ executable. The executable (for now 64bit linux and windows) and needed files are available in this git repository. To install, change into your directory [YOURDIR] from the AvaFrame installation above and clone the repository:

```
cd [YOURDIR]
git clone https://github.com/avaframe/com1DFA_Exe
```

Rename the executables according to your operating system, i.e. for Linux do:

```
mv com1DFA_Exe/com1DFA_x86_64.exe com1DFA_Exe/com1DFA.exe
mv com1DFA_Exe/SHPConv_linux.exe com1DFA_Exe/SHPConv.exe
```

for Windows do:

```
mv com1DFA_Exe/com1DFA_win64.exe com1DFA_Exe/com1DFA.exe
mv com1DFA_Exe/SHPConv_win.exe com1DFA_Exe/SHPConv.exe
```

Go to the `com1DFAOrig` directory of the AvaFrame repository from above and copy the configuration file:

```
cd AvaFrame/avaframe/com1DFAOrig
cp com1DFACfg.ini local_com1DFACfg.ini
```

Open the `local_com1DFACfg.ini` file in your preferred text editor and change the `com1Exe` variable to reflect your paths, i.e.:

```
com1Exe = [YOURDIR]/com1DFA_Exe/com1DFA.exe -files [YOURDIR]/com1DFA_Exe/files/AK_
↪Attributes
```

Attention: We suggest to use the full path.

To test go to [YOURDIR], change into the com1DFA_Exe repository and run the executable:

```
cd [YOURDIR]
cd com1DFA_Exe
./com1DFA.exe -files files/AK_Attributes/
```

The output should start like this:

```
Setting config files directory: files/AK_Attributes/  (src/SW_Workspace.cpp:3435)
./com1DFA.exe -files files/AK_Attributes/     (src/SW_Workspace.cpp:3453)
================================================================
./com1DFA.exe
Compiled Oct 19 2020 21:34:15
...
```

Exit by pressing `q`

3.2.2 Input

The module requires an avalanche directory, that follows a specified folder structure. This avalanche directory can be created by running `runInitializeProject.py`. In the directory *Inputs*, the following files are required:

- digital elevation model as .asc file -> use ESRI grid format
- release area scenario as shapefile (in Inputs/REL); multiple are possible -> the shapefile name should not contain an underscore, if so '_AF' is added

and the following files are optional:

- entrainment area as shapefile (in *Inputs/ENT*)
- resistance area as shapefile (in *Inputs/RES*)
- secondary release area as shapefile (in Inputs/SECREL)

The simulation settings area defined in the configuration file `com1DFAOrig/com1DFAOrigCfg.ini`:

- com1Exe - path to com1DFA executable
- flagOut - print full model output
- simTypeList - simulation types that shall be performed (options: null, ent, res, entres, available; if multiple, separate by '|'))
- releaseScenario - name of release area scenario shapefile (with or without extension -shp, if multiple, separate by '|')
- flagVarPar - perform parameter variation
- varPar - parameter to be varied
- varParValues - values for parameter variation

3.2.3 Output

The simulation results are saved to: *Outputs/com1DFAOrig* and include:

- raster files of the peak values for pressure, flow thickness and flow velocity (*Outputs/com1DFAOrig/peakFiles*)
- reports of all simulations (*Outputs/com1DFAOrig/reports*)
- log files of all simulations
- experiment log that lists all simulations

3.2.4 To run

Attention: Please refer to the instructions in C++ Executable on how to get the necessary C++ executable and setup the correct paths.

- first go to `AvaFrame/avaframe`
- create an avalanche directory with required input files - for this task you can use *Initialize Project*
- copy `avaframeCfg.ini` to `local_avaframeCfg.ini` and set your desired avalanche directory name
- run:

```
python3 com1DFAOrig/runCom1DFA.py
```

3.2.5 Theory

The governing equations of the dense flow avalanche are derived from the incompressible mass and momentum balance on a Lagrange control volume ([Zwi00, ZKS03]). Assuming the avalanche is much longer and larger than thick, it is possible to integrate the governing equations over the thickness of the avalanche and operate some simplifications due to the shape of the avalanche. This leads, after some calculation steps described in details in Theory *Governing Equations for the Dense Flow Avalanche* to:

$$\frac{\mathrm{d}V(t)}{\mathrm{d}t} = \frac{\mathrm{d}(A_b\overline{h})}{\mathrm{d}t} = \frac{\rho_{\text{ent}}}{\rho_0}\, w_f\, h_{\text{ent}}\, \|\overline{\mathbf{u}}\|$$

$$\frac{\mathrm{d}\overline{u}_i}{\mathrm{d}t} = g_i + \frac{K_{(i)}}{\overline{\rho}\, A\, \overline{h}} \oint_{\partial A} \left(\frac{\overline{h}\, \sigma^{(b)}}{2} \right) n_i\, \mathrm{d}l - \delta_{i1} \frac{\tau^{(b)}}{\overline{\rho}\, \overline{h}} - C_{\text{res}}\, \overline{\mathbf{u}}^2\, \frac{\overline{u_i}}{\|\overline{\mathbf{u}}\|} - \frac{\overline{u_i}}{A\, \overline{h}} \frac{\mathrm{d}(A\, \overline{h})}{\mathrm{d}t} + \frac{F_i^{\text{ent}}}{\overline{\rho}\, A\, \overline{h}}$$

$$\overline{\sigma}_{33}^{(b)} = \rho \left(g_3 - \overline{u_1}^2 \frac{\partial^2 b}{\partial x_1^2} \right) \overline{h}$$

3.2.6 Numerics

Those equations are solved numerically using a **SPH** method ([LL10, Sam07]). **SPH** is a mesh free method where the basic idea is to divide the avalanche into small mass particles. The particles interact with each other according to the equation of motion described in *Theory* and the chosen kernel function. This kernel function describes the domain of influence of a particle (through the smoothing length parameter). See theory theoryCom1DFA:Numerics for further details.

3.3 com2AB: Alpha Beta Model

com2AB calculates the runout of an avalanche according to the statistical $\alpha - \beta$ model. An avalanche is defined by its DEM (digital elevation model), a path and a split point. The runout is calculated according to the $\alpha - \beta$ model calibrated for avalanches in the Austrian Alps. It is also possible to adapt the model parameters for other regions.

3.3.1 Input

Please be aware that all input data have to be provided in the same projection.

- digital elevation model as .asc file with ESRI grid format
- set of avalanche paths as (multi-) line shapefile. I.e there can be multiple paths in the shapefile
  - recommended attribute *name*
- Split points as (multi-) point shapefile. I.e. there can be multiple split points in the shapefile. For each profile/path the closest split point is considered.

3.3.2 Outputs

- profile plot with alpha, beta and runout points
- txt file with angle and coordinates of the different points

3.3.3 To run

- go to `AvaFrame/avaframe`
- copy `com2AB/com2ABCfg.ini` to `com2AB/local_com2ABCfg.ini` and edit (if not, default values are used)
- make sure all the required inputs are available in the avalanche directory
- enter the path to the desired dataset in `local_avaframeCfg.ini`
- run:

```
python3 runCom2AB.py
```

3.3.4 Theory

The snow avalanche runout distance is calculated using a statistical model based on data collected for real avalanches ([BDL83, LB80, Wag16]). An equation of the following type is fitted to the data:

$$\alpha_j = k_1\beta + k_2 z'' + k_3 H_0 + k_4 + jSD$$

where H_0 is the elevation loss of the quadratic fit of the avalanche profile. z'' is the curvature of this same quadratic fit. β is the angle of the line between the $10°$ point (first point where the avalanche profiles slope is under $10°$) and the starting point. The coefficients (k_1, k_2, k_3, k_4) and the standard deviation SD are calculated during the fitting process. Index $j = \{-1, -2, 0, 1\}$ and $\alpha_j = \alpha + jSD$. These coefficients obviously depend on the initial set of chosen data. $\alpha_0 = \alpha$ is the angle between the stopping and the starting point of the avalanche. $\alpha_j = \alpha + jSD$ takes into account the variability of the process. The values of the SD used are based on normal distribution. It is important to note that a bigger runout angle leads to a shorter runout distance. This means that $\alpha_{-1} = \alpha - SD$ leads to a longer runout. In other words, the probability of the runout being shorter than $s_{\alpha_{-1}}$ corresponding to α_{-1} is approximately 83%.

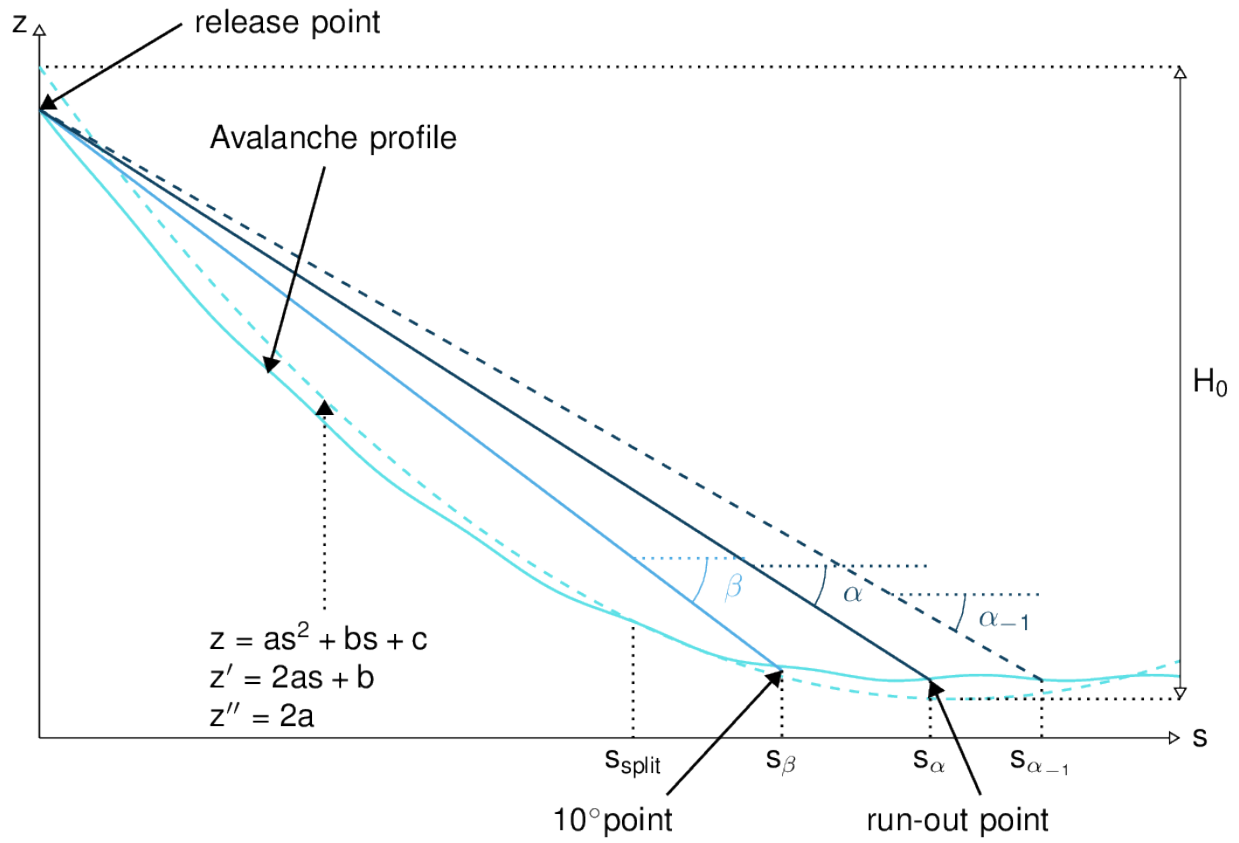


Fig. 3.1: Topographical variables for the calculation of α

In this module, the coefficients (k_1, k_2, k_3, k_4) and the standard deviation SD are already known, they are simply used in the α equation to calculate the runout on a new profile.

3.3.5 Procedure

Preprocessing :

- The avalanche path (x,y) is first resampled. Default value for resampling is distance=10m (maximal horizontal distance between two points). Note that it does not make much sense to decrease this value to be smaller than the raster grid resolution. We then introduce the curvilinear coordinate s which represents the projected horizontal distance along the path.
- The avalanche path is projected on the DEM to generate the profile using a bi-linear interpolation on the DEM to the point of interest.
- The split point (which is not necessarily given on the avalanche path) is projected on the avalanche path.

From this we obtain the (x,y,z) and (s,z) coordinates of the avalanche profile.

AlphaBeta Model:

- Find the $10°$ point from (s,z).
- Calculate β.
- Calculate the α_j angles using the adequate standard, small avalanche or custom parameter set.

Postprocessing:

- Plot and save results.

3.3.6 Configuration parameters

distance
: resampling distance. The given avalanche path is resampled with a step <= 10m (default).

dsMin
: float. Threshold distance [m]. When looking for the beta point make sure at least dsMin meters after the beta point also have an angle bellow $10°$ (dsMin=30m as default).

smallAva
: boolean (False as default) if True apply (k_1, k_2, k_3, k_4, SD) set of small avalanches or False, standard avalanches

customParam
: boolean (False as default). Enables to choose custom (k_1, k_2, k_3, k_4, SD). If True, the values from the configuration file are used

k1
: float. Use this value if `customParam=True`

k2
: float. Use this value if `customParam=True`

k3
: float. Use this value if `customParam=True`

k4
: float. Use this value if `customParam=True`

SD
float. Use this value if `customParam=True`

PlotPath
Plot Avalanche path on raster; default False

PlotProfile
Plot profile; default False

SaveProfile
Save profile to file; default True

WriteRes
Write result to file: default True

3.4 com3Hybrid: Hybrid modeling

com3Hybrid is a computational module that combines the dense flow avalanche (DFA) simulation model (com1DFA) and the Alpha-Beta statistical one (com2AB), taking advantage of their strength and trying to reduce their weaknesses. The weakness of the DFA simulation lies in its required inputs, among which is the friction parameter μ. The disadvantage of the statistical model lies in the necessary path, which is used to extract an avalanche profile and compute a runout angle. The idea here is to determine the avalanche path in an automated way by:

- first, running a DFA simulation
- computing a mass averaged path from the results
- use the path to compute the runout angle corresponding to this specific avalanche
- using a coulomb friction method, the friction parameter μ is extracted from the runout
- the resulting μ is used in the input for a new DFA simulation.

This iteration process can be repeated multiple times until "convergence", which means until the μ or α value stops varying.

3.4.1 Input

- raster of the DEM (.asc file)
- a release feature (shapefile) in `Inputs/REL`
- Split point (shapefile). This requirement is planned to be automated as well.

3.4.2 Outputs

- The avalanche path specific for the input topography
- the μ value (α travel angle) specific for the input topography
- results from the DFA simulation for the input topography
- a combined plot showing the results of the DFA simulation and Alpha-Beta model

3.4.3 To run

- go to `AvaFrame/avaframe`
- copy `com3Hybrid/com3HybridCfg.ini` to `com3Hybrid/local_com3HybridCfg.ini` and edit (if not, default values are used)
- put your com1DFA and com2AB settings in `com3Hybrid/hybridModel_com1DFACfg.ini` and `com3Hybrid/hybridModel_com2ABCfg.ini` (these files replace the local `.ini` files in `com1DFA` and `com2AB`)
- make sure all the required inputs are available in the avalanche directory
- enter the path to the desired dataset in `local_avaframeCfg.ini`
- run:

```
python3 runScripts/runCom3Hybrid.py
```

3.4.4 Procedure

1. A first com1DFA simulation is run using the `com3Hybrid/hybridModel_com1DFACfg.ini` configuration and the μ provided in `com3Hybrid/local_com3HybridCfg.ini`.
2. The mass averaged path is computed from the DFA simulation using the *ana5Utils.DFAPathGeneration*. From this we obtain the (x,y,z) and (s,z) coordinates of the avalanche profile (the path is extended towards the relase and in the runout). This path is saved to `avalancheDir/Inputs/LINES` and replaces the old path. A copy is also saved to `avalancheDir/Outputs/com3Hybrid`.
3. com2AB module is run using the `com3Hybrid/hybridModel_com2ABCfg.ini` to compute the $\mu = \tan\alpha$ friction parameter corresponding to this specific avalanche and area.
4. The `com3Hybrid/hybridModel_com1DFACfg.ini` is updated with the new $\mu = \tan\alpha$ value and com1DFA simulation is run again.
5. Steps 2. and 3. are repeated to get the new avalanche path and $\mu = \tan\alpha$ value.

This process is repeated as long as the $\alpha = \arctan\mu$ value varies from more than `alphaThreshold` which you can specify in `com3Hybrid/local_com3HybridCfg.ini` (this is a value in degrees) and this for a maximum of `nIterMax` iterations.

3.4.5 Configuration parameters

The parameters for path generation (section `PATH` of `com3Hybrid/local_com3HybridCfg.ini`) are described in *Automated path generation*.

The parameters for the DFA simulation are to be specified in `com3Hybrid/hybridModel_com1DFACfg.ini`. This file replaces the `local_com1DFACfg.ini` that is usually used to run a com1DFA simulation.

The parameters for the $\alpha - \beta$ model are to be specified in `com3Hybrid/hybridModel_com2ABCfg.ini`. This file replaces the `local_com2ABCfg.ini` that is usually used to run a com2AB simulation.

3.5 com5SnowSlide: Snow slide

The com5SnowSlide computational module provides the option to add an elastic cohesion force between particles based on com1DFA. This can be used for example to simulate small snow slides.

The cohesion between particles is modeled using elastic bonds with maximum strain rate. This means that the cohesion force $\mathbf{F}_{lk}$ exerted by l on k is expressed by:

$$\mathbf{F}_{lk} = -\mathbf{F}_{kl} = \frac{L - L_0}{L_0} A_{kl} E \mathbf{e_{kl}}$$

Where L and L_0 are the distances between the particles k and l at a time t and at rest (initial time step), E is the elastic modulus of the material, A_{kl} is the contact area between particles k and l and $\mathbf{e_{kl}}$ is the unit vector

$$\epsilon = \frac{L - L_0}{L_0}$$

represents the strain (ϵ is positive in the case of elongation and negative in compression). If the strain exceeds a critical value ϵ_c the bond between the particles k and l breaks.

The contact area A_{kl} between particles k and l reads:

$$A_{kl} = h_{kl} d_{kl} = \frac{h_k + h_l}{2} \frac{L}{\sqrt{3}}$$

h_{kl} represents the height of the contact surface and d_{kl} represents the length of the contact surface in the horizontal plane assuming that each particle has 6 neighbours.

3.5.1 Input

The standard inputs required to perform a simulation run using *com1DFA* can be found here: *Input*. There is a run script to perform a snow slide com1DFA run: `runCom5SnowSlide.py`, and the configuration settings can be found in `com5SnowSlide/com5SnowSlideCfg.ini`. The snow slide-specific parameters are:

- snowSlide is activated by setting `snowSlide` to 1
- the maximum strain before breaking of the bond `cohesionMaxStrain`
- the Young modulus in N/m^2 `cohesiveSurfaceTension`

However, also several other parameters, for example the particle initialization method, friction model and parameters, are changed compared to the default configuration of *com1DFA* and listed in the `com5SnowSlide/com5SnowSlideCfg.ini` in the com1DFA_override section.

Please note that the provided default setup for snow slide calculation is defined with the following premises:

- slope gradient in the release area from approx. 28° to approx. 50°.
- a return level for hazard mapping of 150-yr of snow depth as the initial value.
- movement as an intact unit without turbulence
- height difference from approx. 30 m to max. 60-80 m
- valid for altitude levels > 500 m and < 1,500 m sea level

3.5.2 Initialization of bonds

If the elastic cohesion (`snowSlide` set to 1) is activated, the bonds between particles are initialized. The construction of the bonds is done by building a triangular mesh on the particles (used as a point cloud) and the Delaunay triangulation function from matplotlib. The length L_0 of the bonds at rest (initial distance between two bonded particles) is also initialized here.

Note: It is recommended to use the triangular (`triangular`) initialization option (*Initialize particles*) rather than the random or the uniform one in the case where cohesion is activated.

3.5.3 To run

- first go to `AvaFrame/avaframe`
- copy `avaframeCfg.ini` to `local_avaframeCfg.ini` and set your desired avalanche directory name
- create an avalanche directory with required input files - for this task you can use *Initialize Project*
- copy `com5SnowSlide/com5SnowSlideCfg.ini` to `com5SnowSlide/local_com5SnowSlideCfg.ini` and if desired change configuration settings
- if you are on a develop installation, make sure you have an updated compilation, see installation:Setup AvaFrame
- run:

```
python3 runCom5SnowSlide.py
```

3.6 com4FlowPy: Flow-Py

Note: THIS MODULE IS CURRENTLY UNDER HEAVY DEVELOPMENT!

The code is not automatically tested in any way and not included in the code coverage!

It also does not adhere to the AvaFrame coding and naming conventions (yet)...

Use at your own risk and if you want to contribute to the modules improvement, you are very welcome to do so!

Flow-Py is an open source tool to compute gravitational mass flows (GMF) run out and intensity. The main objective is to compute the spatial extent of GMF, which consists of the starting, transit and runout zones of GMF on the surface of a three dimensional terrain. The resulting run out is mainly dependent on the terrain and the location of the starting/release point. No time-dependent equations are solved in the model. com4FlowPy uses existing statistical-data-based approaches for solving the routing and stopping of GMF.

The tool has been designed to be computationally light, allowing the application on a regional scale including a large number of GMF paths. The implementation allows users to address specific GMF research questions by keeping the parameterization flexible and the ability to include custom model extensions and add-ons.

3.6.1 Running the Code

You are required to install rasterio and gdal separately, since they are not included in the general AvaFrame requirements.

If you have trouble installing GDAL or rasterio on Windows use these links to get the required version directly from their website, first install *GDAL* and then *rasterio*.

GDAL: https://www.lfd.uci.edu/~gohlke/pythonlibs/#gdal

rasterio: https://www.lfd.uci.edu/~gohlke/pythonlibs/#rasterio

Once the required libraries are installed the model runs via the `runCom4FlowPy.py` script.

3.6.2 Configuration

The configuration can be found in `com4FlowPyCfg.ini`

- alpha_angle (controls the run out angle induced stopping and routing)
- exponent (controls concentration of routing flux and therefore the lateral spread)
- working directory path
- path to DEM raster (.tiff or .asc)
- path to release raster (.tiff or .asc)
- (Optional) flux threshold (positive number) flux_threshold=xx (limits spreading with the exponent)
- (Optional) Max Z^δ (positive number) max_z_delta=xx (max kinetic energy height, turbulent friction)

3.6.3 Input Files

All raster files (DEM, release, . . .) must be in the .asc or .tif format.

All rasters need the same resolution (normal sizes are e.g. 5x5 or 10x10 meters).

All Layers need the same spatial extend, with no data values < 0 (standard no data values = -9999).

The locations identified as release areas need values > 0. (see release.tif in examples)

3.6.4 Output

All outputs are in the .tiff raster format in the same resolution and extent as the input raster layers.

- z_delta: the maximum z_delta of all paths for every raster cell (geometric measure of process magnitude, can be associated to kinetic energy/velocity)
- Flux: The maximum routing flux of all paths for every raster cell
- sum_z_delta: z_delta summed up over all paths on every raster cell
- Cell_Counts: number of paths that route flux through a raster cell
- Flow Path Travel Angle, FP_TA: the gamma angle along the flow path
- Straight Line Travel Angle, SL_TA: Saves the gamma angle, while the distances are calculated via a straight line from the release cell to the current cell

CHAPTER

FOUR

INPUT/TRANSFORMATION MODULES

4.1 in1Data: Input data utilities

4.1.1 Get input data

in1Data.getInput module provides functions to fetch or generate input data for avalanche simulations within AvaFrame. The main functions are described in more detail in the following sections. A detailed description of all the small helper functions is provided in *in1Data.getInput*

4.1.1.1 getInputCom1DFA

`in1Data.getInput.getInputCom1DFA()` fetches the input data from an avalanche director required to start avalanche simulations with *com1DFA*. This data consists of a digital elevation model (DEM), a release scenario shapefiles, optional entrainment and resistance area shapefiles and is returned as a dictionary.

4.1.2 computeFromDistribution

in1Data.computeFromDistribution is collection of functions that facilitates retrieving a sample of values that are distributed following a uniform or Pert distribution. This is useful if you want to perform avalanche simulations for a range of release thickness values (or any other parameter), that should be distributed following a specific distribution. Besides returning a sample of values following the desired distribution, various plots can be generated detailing the characteristics of the sample. The required input parameters can be found in the respective configuration file `in1Data.computeFromDistributionCfg.ini`. Detailed information on the functions can be found in *in1Data.computeFromDistribution*.

4.1.2.1 To run

An example of how a parameter set distributed following a uniform or Pert distribution can be generated, is provided by `runScripts/runComputeDist.py`.

- first go to `AvaFrame/avaframe`
- copy `in1Data/computeFromDistributionCfg.ini` to `in1Data/local_computeFromDistributionCfg.ini` (if not, the standard settings are used)
- adjust path to the desired `NameOfAvalanche/` folder in your local copy of `avaframeCfg.ini`
- run:

```
python3 runScripts/runComputeDist.py
```

4.2 in2Trans: Transformation Utilities

4.2.1 Working with ASCII files

in2Trans.ascUtils is a module created to handle raster ASCII files. It contains different functions to read ASCII files and write the data to a numpy array, to compare raster file headers or to write a raster to an ASCII file. A description of the functions is available in *in2Trans.ascUtils*.

4.2.2 Working with shapefiles

in2Trans.shpConversion is a module for handling shapefiles. It contains different functions to read shapefiles and convert them to a python dictionary. It also provides functions to extract or remove a feature from the shapefile dictionary.

4.2.2.1 Reading shapefiles

Shapefiles are converted to a python dictionary. The dictionary has information about the number of features as well as the coordinates of the points. The output dictionary SHPdata looks like this:

```
SHPdata['Name'] = ['nameFeature1', 'nameFeature2', 'nameFeature3']
SHPdata['x'] = [xCoordsssFeature1, xCoordsssFeature2, xCoordsssFeature3]
SHPdata['y'] = [yCoordsssFeature1, yCoordsssFeature2, yCoordsssFeature3]
SHPdata['z'] = [zerosss, zerosss, zerosss]
SHPdata['Start'] = [indexStartFeature1, indexStartFeature2, indexStartFeature3]
SHPdata['Length'] = [lenghtFeature1, lenghtFeature2, lenghtFeature3]
SHPdata['sks'] = 'ProjectionInformation'
```

A description of the functions is available in *in2Trans.shpConversion*

4.3 in3Utils: Various Utilities Modules

4.3.1 geoTrans

The *in3Utils.geoTrans* module provides useful functions to operate transformations, comparison or interpolation on rasters, lines, points… Further information about the available functions can be found in *in3Utils.geoTrans*

4.3.2 Generate Topography

The *in3Utils.generateTopo* module provides functions to generate DEM files for idealized/generic topographies that can be used for snow avalanche simulations.

This module can generate the following topographies:

- flat plane (FP)
- inclined plane of constant slope (IP)
- parabola - parabolic slope transitioning into a flat foreland (PF)
- hockey stick - inclined plane of constant slope with a smooth transition to a flat foreland (HS)
- bowl-shaped topography (BL)

- helix-shaped topography (HX)
- pyramid-shaped topography (PY)

Extra features can be added to the above topographies:

- **a channel can be introduced (then set `channel=True`).**
  This channel can also be set to widen towards the top and bottom of the channel (then set narrowing=True). The channel can be added by either adding a 'channel layer' (max thickness=channel depth) on top of the topography (`topoAdd=True`) or by cutting them into the original topography (`topoAdd=False`).
- **in case of the parabola topography, a dam can be added by setting ``dam=True``.**

This modules returns a 3D plot of the generated topography as well as an .asc file of the DEM data. The input parameters are defined in the respective configuration file `in3Utils.generateTopoCfg.ini`. Detailed information on the individual functions used to create the topographies can be found in *in3Utils.generateTopo*

4.3.2.1 To run generateTopo

- first go to `AvaFrame/avaframe`
- copy `in3Utils/generateTopoCfg.ini` to `in3Utils/local_generateTopoCfg.ini` and set the desired parameter values (if not, the default values are used)
- run:

```
python3 runScripts/runGenerateTopo.py
```

4.3.2.2 Theory

Topographies are generated using inclined and flat planes, parabolas, spheres and circles. Channels are introduced as half-sphere shaped features with smooth transition from the no-channel area to the channel using cumulative distribution functions.

4.3.2.3 Configuration parameters

In the case of the pyramid-shaped topography, the domain extent is defined by the max elevation (z0 - elevation of the appex point) and the slope of the pyramid facets (meanAlpha)

Domain parameters:

dx
: DEM spatial resolution [m]

xEnd
: total horizontal extent of the domain [m]

yEnd
: total vertical extent of the domain [m]

Topography parameters:

flens
: distance to the point where the slope transitions into a flat plane [m]

meanAlpha
: slope angle from the max. elevation to the start of the flat plane [°] - or slope of the inclined plane [°]

C
total fall height [m]

rBowl
bowl radius [m]

rHelix
radius for helix [m]

z0
max elevation [m]

zElev
elevation of the flat plane [m]

rCirc
radius of the smoothing circle [m]

demType
topography types (FP, IP, PF, HS, BL, HX, PY - explanation given in the introductory description)

flatx
extent of the flat foreland for the pyramid in x

flaty
extent of the flat foreland for the pyramid in y

phi
rotation angle for the pyramid

Flags for channels and plotting:

channel
True - introduce channel; False - no channel

narrowing
True - channel is wide at start and end and narrow in the middle part; False - channel is uniform

topoAdd
True - add channel layer; False: cut channel into original topography;

flagRot
True - rotate pyramid along z-axis

Channel parameters:

cRadius
standard channel radius

cInit
start and end half width of channel that is narrower in the middle part

cff
standard deviation sigma

cMustart
mean mu - represents upper part of the channel

cMuend
mean mu - represents lower part of the channel

4.3.3 Get Release Area

in3Utils.getReleaseArea generates a release area for a topography created with *in3Utils.generateTopo*, this function is available for the following topographies:

- flat plane (FP)
- inclined plane (IP)
- parabola (PF)
- hockey stick (HS)

The release areas are defined as rectangular features build by four corner points, which are based on the following conditions:

- prescribed vertical stretch in meters (difference in altitude)
- prescribed volume of the release area
- lower margin is located where the slope angle falls below 30°
- if slope does not fall below 30 °, the upper margin is located xStart away from the upper margin of the DEM

The release areas can be saved as shapefile, .nxyz and .txt file. The required input parameters can be set in the respective configuration files `in3Utils/getReleaseAreaCfg.ini` and `in3Utils.generateTopoCfg.ini`. Detailed information on the individual functions used to create the release areas can be found in *in3Utils.getReleaseArea*

4.3.3.1 To run getReleaseArea

Following these steps, you can generate an avalanche test case including a DEM and a simple release area.

- first go to `AvaFrame/avaframe`
- copy `in3Utils/generateTopoCfg.ini` and `in3Utils/getReleaseAreaCfg.ini` to `in3Utils/local_generateTopoCfg.ini` and `in3Utils/local_getReleaseAreaCfg.ini` and set desired parameter values (if not, the default values are used)
- run:

```
python3 runGenProjTopoRelease.py
```

Parameters:

hr
: release area vertical stretch [m]

vol
: volume of snow in release area [m3]

dh
: release snow thickness [m]

xStart
: upper margin of the release area distance in x from origin [m]

lenP
: number of release area polygon points

outputtxt
: True - copy the output to txt file

xExtent
horizontal extent of release area for flat plane

alphaStop
slope angle that defines lower margin of release area

relNo
number of release area for name

relName
name of release area feature in shapefile

4.3.4 Initialize Project

in3Utils.initializeProject provides functions to initialize a project, create the required directory structure and delete specified files or directories.

The main function *in3Utils.initializeProject.initializeFolderStruct()*, creates the folder structure required to perform avalanche simulations:

```
NameOfAvalanche/
        Inputs/
                ENT/            - entrainment areas
                LINES/          - avalanche paths
                POINTS/         - split points
                REL/            - release area scenario
                RES/            - resistance areas
                SECREL/ - secondary release areas
                .asc            - DEM
        Outputs/
        Work/
```

The path to this folder is specified in the configuration file `avaframeCfg.ini`, with the parameter *avalancheDir*.

4.3.4.1 To run

- first go to `AvaFrame/avaframe`
- copy `avaframeCfg.ini` to `local_avaframeCfg.ini` and set your desired avalanche directory name
- run:

```
python3 runInitializeProject.py
```

4.3.5 fileHandlerUtils

in3Utils.fileHandlerUtils gathers useful functions to create directories, read log files, extract information from logs, fetch and export data and fetch simulation info into a dataFrame. Details on these functions can be found in *in3Utils.fileHandlerUtils*.

CHAPTER

FIVE

ANALYSIS/HELPER MODULES

5.1 ana1Tests: Testing

In this module we gather tests with analytical or semi-analytical solution. These tests are used to evaluate the precision and accuracy of the com1DFA numerical solution.

5.1.1 Deviation/difference/error computation

In order to either assess the accuracy of a numerical method or to compare results it will be necessary to compute errors. The methods used to compute deviations are described in this section.

Two deviation/difference/error measures are used. The first one is based on the $\mathcal{L}_{max}$ norm (uniform norm), the second on the Euclidean norm ($\mathcal{L}_2$ norm). In both cases, the aim is to measure the deviation between a numerical solution and reference solution on an interval (one or two dimensional). Let f_{num} be the numerical solution and f_{ref} the reference solution defined on an interval Ω. The deviation is defined by $\epsilon(x) = f_{num}(x) - f_{ref}(x)$

5.1.1.1 Uniform norm

The $\mathcal{L}_{max}$ norm measures the largest absolute value:

$$\mathcal{L}_{max}(\epsilon(x)) = \sup_{x \in \Omega}(|\epsilon(x)|)$$

This will give an idea of the largest difference between the two solutions. It can be applied to one or two dimensional results.

It can also be normalized by dividing the uniform norm of the deviation by the uniform norm of the reference:

$$\frac{\mathcal{L}_{max}(\epsilon(x))}{\mathcal{L}_{max}(f_{ref}(x))}$$

5.1.1.2 Euclidean norm

The $\mathcal{L}_2$ norm is defined by:

$$\mathcal{L}_2(\epsilon) = \int_{x \in \Omega} ||\epsilon(x)||^2 \, dx$$

This norm will give an overall measure of the deviations. It is useful to normalize the norm of the deviation either by dividing with the norm of the reference solution:

$$\frac{\mathcal{L}_2(\epsilon(x))}{\mathcal{L}_2(f_{ref}(x))}$$

or by the measure of the interval ($\mathcal{L}_2(1) = \int_{x\in\Omega} dx$):

$$\sqrt{\frac{\mathcal{L}_2(\epsilon(x))}{\mathcal{L}_2(1)}}$$

The first normalization approach will give a relative deviation in % whereas the second will give an average deviation of f on Ω.

5.1.2 Dambreak test

The Dambreak test compares the results of DFA simulations to the analytical solution of the "dam break" problem. In this test, a granular mass (Coulomb material) is suddenly released from rest on an inclined plane. In the case of a thickness integrated model as derived by Savage and Hutter (e.g. in [HSSN93]), an analytical solution exists. This solution is described in [FM13] and corresponds to a Riemann problem.

The problem considered has the following initial conditions:

$$(h, \mathbf{u})(x, t = 0) = \begin{cases} (h_0, \mathbf{0}), & \text{if } x \leq 0 \\ (0\ \ , \mathbf{0}), & \text{if } x > 0 \end{cases}$$

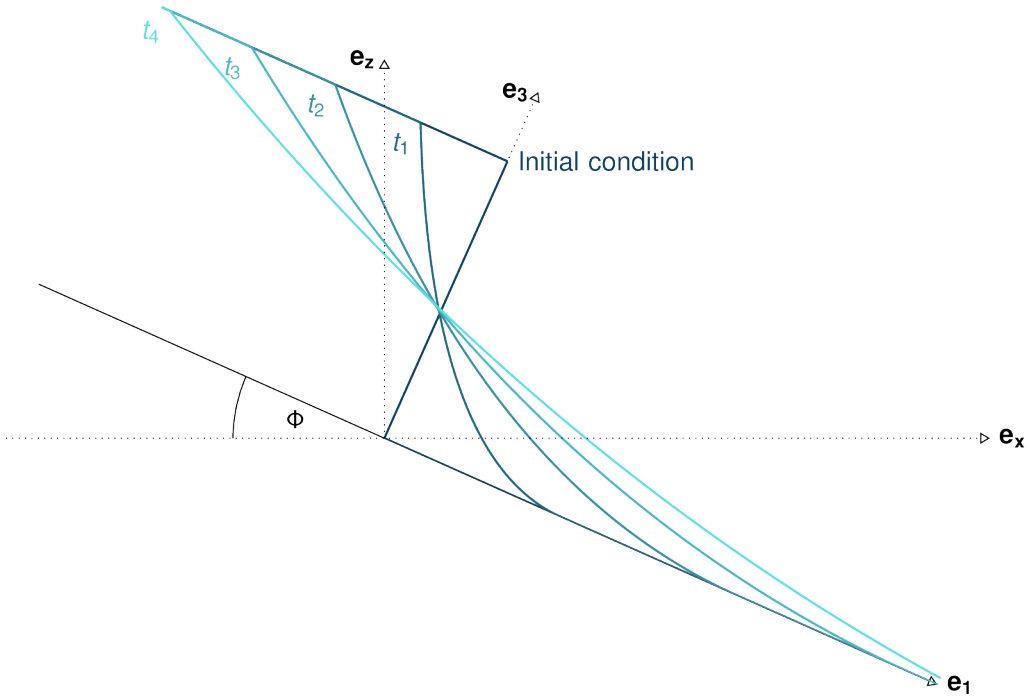


Fig. 5.1: Dam break theoretical evolution

The functions computing the analytical solution and comparing it to the simulation results can be found in *ana1Tests.damBreak* and *ana1Tests.analysisTools*. Plotting routines are located in *out3Plot.outAna1Plots*. The input data for this test case can be found in `data/avaDamBreak`.

This test produces a summary figure combining a comparison between the analytical solution and simulation result (cross cut along the flow direction) as well as a map view and an error measure plot as shown in Fig. 5.2.

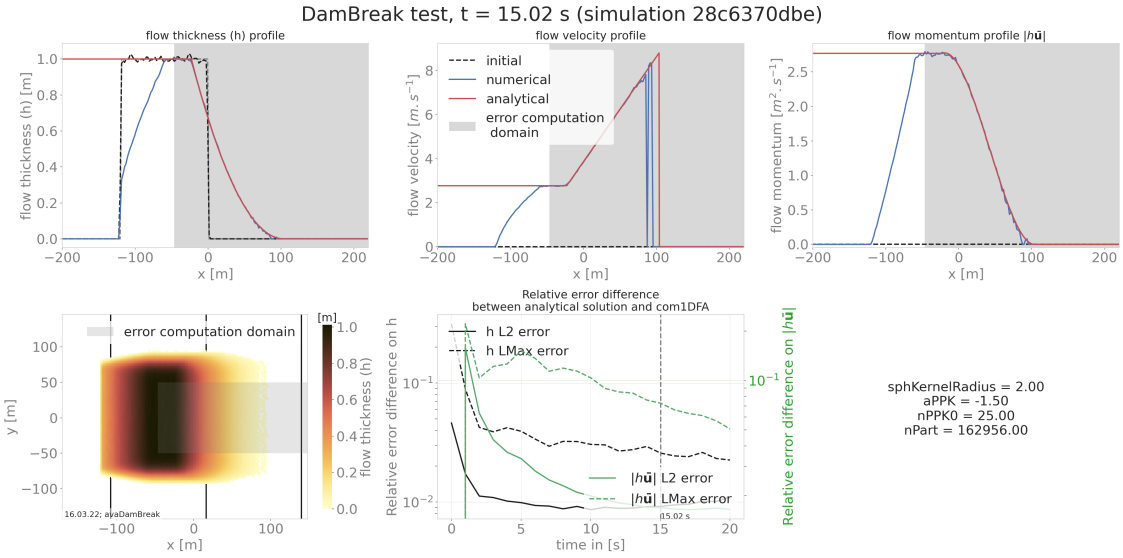


Fig. 5.2: Summary figure produced by the damBreak test (here for a cell size of 2m)

Another optional result is the comparison cross cut figure for all saved time steps as shown in the following animated figure.

Fig. 5.3: Time evolution of the flow thickness, velocity and momentum

5.1.2.1 To run

An example on how to apply this test is provided in `runScripts/runDamBreak` and `runScripts/runAnalyzeDamBreak`. The required input files are located in `data/avaDamBreak` (including the configuration file `data/avaDamBreak/Inputs/damBreak_com1DFACfg.ini`). In this configuration file, there is a specific section `'DAMBREAK'` providing the required input parameters to compute the analytical solution. In order to run the test example:

- in `AvaFrame/avaframe` run:

```
python3 runScripts/runDamBreak.py
python3 runScripts/runAnalyzeDamBreak.py
```

5.1.3 Similarity solution

The similarity solution is one of the few cases where a semi-analytic solution can be derived for solving the thickness integrated equations. It is a useful test case for validating simulation results coming from the dense flow avalanche computational module. This semi-analytic solution can be derived under very strict conditions and making one major assumption on the shape of the solution (symmetry/anti-symmetry of the solution around the x and y axis). The full development of the conditions and assumptions as well as the derivation of the solution is presented in details in [HSSN93]. The term semi-analytic is here used because the method enables to transform the PDE (partial differential equation) of the problem into an ODE using a similarity analysis method. Solving the ODE still requires a numerical integration but this last one is more accurate (when conducted properly) and requires less computation power than solving the PDE.

In this problem, we consider an avalanche governed by a dry friction law (Coulomb friction) flowing down an inclined plane. The released mass is initially distributed in an ellipse with a parabolic depth shape. This mass is suddenly released at $t = 0$ and flows down the inclined plane.

The `anaTests.simiSol` module provides functions to compute the semi-analytic solution and to compare it to the output from the DFA computational module as well as some plotting routines to visualize this solution in *`out3Plot.outAna1Plots`*.

Comparing the results from the DFA module to the similarity solution leads to the following plots:

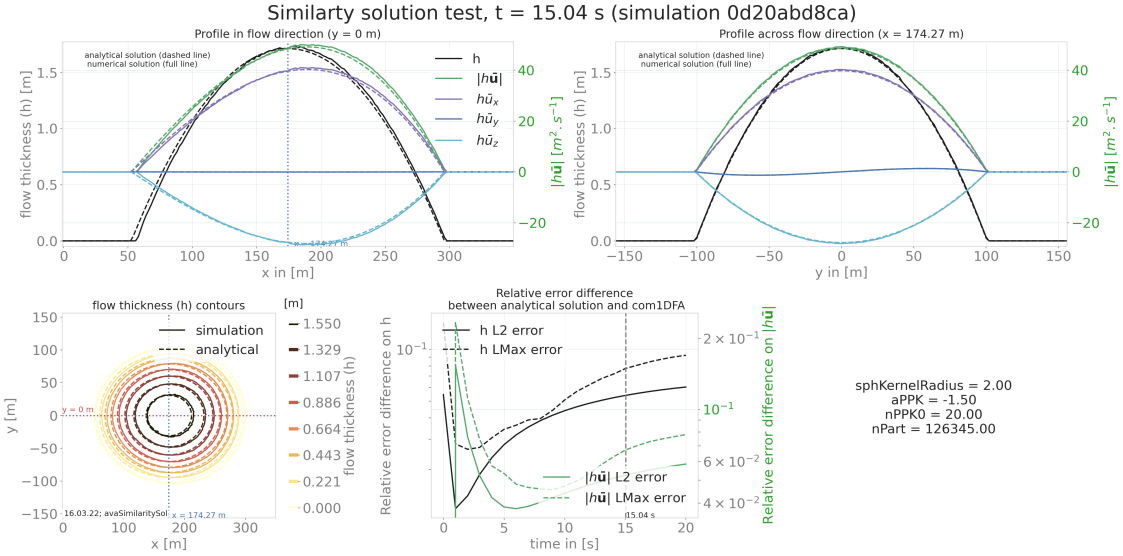


Fig. 5.4: Summary figure produced by the simiSol test (here for a cell size of 2m)

Fig. 5.5: Time evolution of the flow thickness contours in the x, y domain

Fig. 5.6: Time evolution of the profile in and across flow direction

5.1.3.1 To run similarity solution

A workflow example is given in `runScripts/runSimilaritySol`, where the semi-analytical solution is computed and avalanche simulations are performed and both results are then compared. The input data for this example can be found in `data/avaSimilaritySol` with the configuration settings of com1DFA including a section 'SIMISOL' (see `data/avaSimilaritySol/Inputs/simiSol_com1DFACfg.ini`).

The *`out3Plot.outAna1Plots`* function generate profile plots for the flow thickness and momentum in both flow and cross flow directions. The simulation results are plotted alongside the analytical solution for the given time step.

5.1.4 Energy line test

The Energy line test compares the results of the DFA simulation to a geometrical solution that is related to the total energy of the system. Solely considering Coulomb friction this solution is motivated by the first principle of energy conservation along a simplified topography. Here friction force only depends on the slope angle. The analytical runout is the intersection of the path profile with the geometrical (α) line defined by the friction angle (δ) . From the geometrical line it is also possible to extract information about the flow mass averaged velocity at any time or position along the path profile.

5.1.4.1 Theory

Applying the energy conservation law to a material block flowing down a slope with Coulomb friction and this between two infinitesimally close time steps reads:

$$\begin{aligned} E_{t+dt}^{tot} - E_t^{tot} &= E_{t+dt}^{kin} + E_{t+dt}^{pot} - (E_t^{kin} + E_t^{pot}) = \delta E_{fric} \\ &= \frac{1}{2}mv_{t+dt}^2 + mgz_{t+dt} - \frac{1}{2}mv_t^2 - mgz_t \\ &= \mathbf{F}_{fric}.\mathbf{dl} = -\mu||\mathbf{N}||\frac{\mathbf{dl}}{dl}.\mathbf{dl} = -\mu mg(\mathbf{e_z}.\mathbf{n})dl \end{aligned}$$

where δE_{fric} is the energy dissipation due to friction, $\mathbf{N}$ represents the normal (to the slope surface) component of the gravity force, $\mathbf{n}$ the normal vector to the slope surface and $\mathbf{dl}$ is the vector representing the distanced traveled by the material between t and $t + dt$. The normal vector reads $\mathbf{e_z}.\mathbf{n} = cos(\theta)$, where θ is the slope angle. m represents the mass of the material, g the gravity, $\mu = \tan\delta$ the friction coefficient and friction angle, z, respectively v the elevation respectively velocity of the material block. Finally, in the 2D case, $dl = \frac{ds}{cos(\theta)}$, which means that the material is flowing in the steepest slope direction ($\mathbf{ds}$ is the horizontal component of $\mathbf{dl}$).

Integrating the energy conservation between the start and a time t reads:

$$\begin{aligned} E_t^{tot} - E_{t=0}^{tot} &= \frac{1}{2}mv_t^2 + mgz_t - \frac{1}{2}m0v_{t=0}^2 - mgz_0 z_{t=0} \\ &= \int_{t'=0}^{t'=t} \delta E_{fric} dt' = -\int_{s'=s_0}^{s'=s_t} \mu mg ds' = -\mu mg(s_t - 0s_0) \end{aligned}$$

Speaking in terms of altitude, the energy conservation equation can be rewritten:

$$z_0 = z_t + \frac{v_t^2}{2g} + s_t \tan\alpha \tag{5.1}$$

This result is illustrated in the following figure.

Considering a system of material blocks flowing down a slope with to Coulomb friction: we can sum the previous equation Eq.5.1 of each block after weighting it by the block mass. This leads to the mass average energy conservation equation:

$$\bar{z}_0 = \bar{z}_t + \frac{\overline{v^2}_t}{2g} + \bar{s}_t \tan\alpha \tag{5.2}$$

where the mass average $\bar{a}$ value of a quantity a is:

$$\bar{a} = \frac{\sum_k m_k a_k}{\sum_k m_k}$$

This means that the mass averaged quantities also follow the same energy conservation law when expressed in terms of altitude. The same figure as in Fig. 5.7 can be drawn for the center of mass profile path.

The aim is to take advantage of this energy conservation line to evaluate the DFA simulation. Computing the mass averaged path profile for the particles in the simulation and comparing it to the α line allows to compute the error

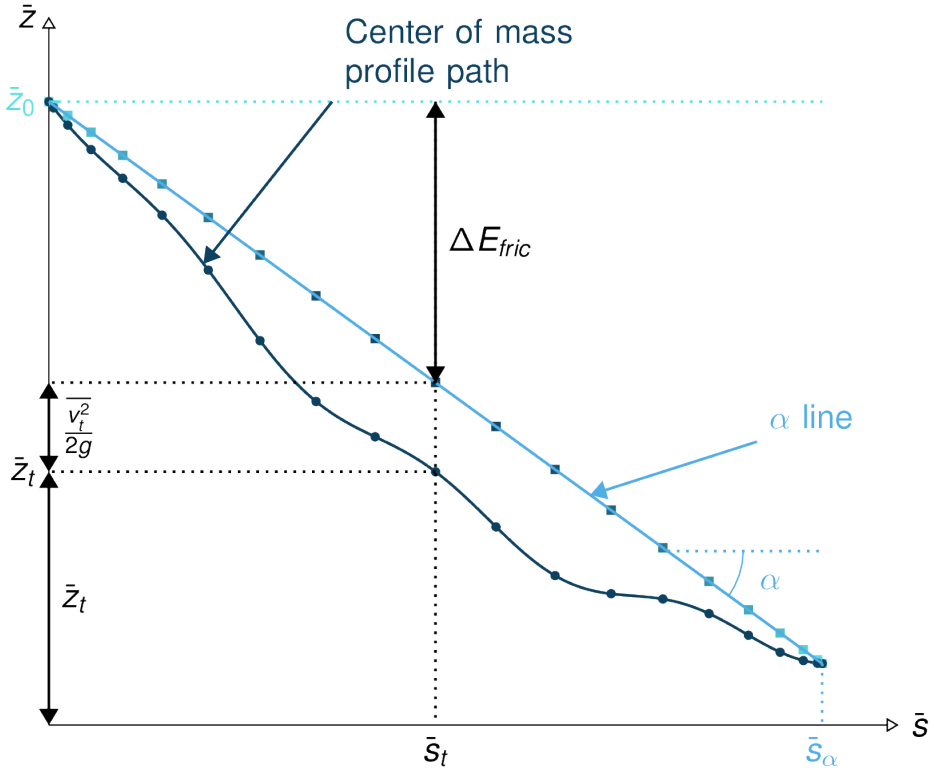


Fig. 5.7: Center of mass profile (dark blue line with the dots) with on top, the energy line (light blue) and the velocity altitude points (colored squared)

compared to the energy line runout. This also applies to the error on the velocity altitude. The following figures illustrate the concept.

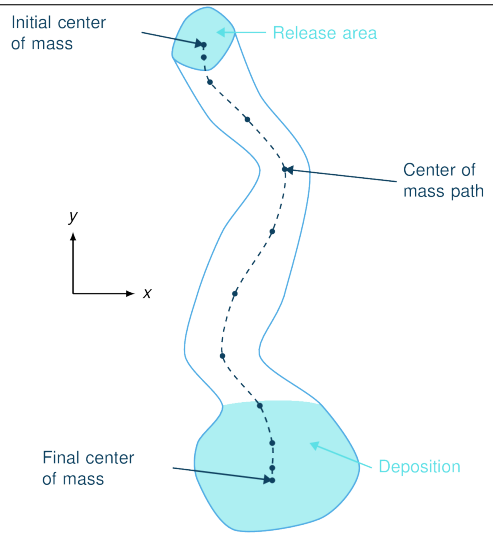


Fig. 5.8: View of the avalanche simulation and extracted path

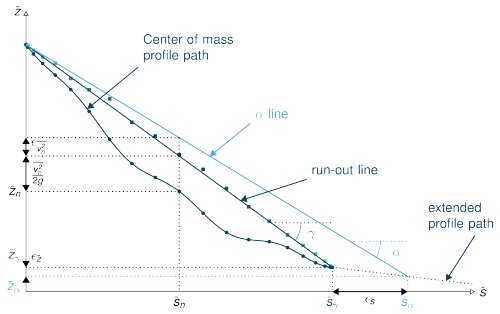


Fig. 5.9: Simulation path profile (dark blue curve and dots) with the runout line (dark blue line and velocity altitude squares), α line and energy points

From the different mass averaged simulation quantities and the theoretical α line it is possible to extract four error indicators. The first three related to the runout point defined by the intersection between the α line and the mass averaged path profile (or its extrapolation if the profile is too short) and the last one is related to the velocity :

- The horizontal distance between the runout point and the end of the path profile defines the $\epsilon_s = \bar{s}_\gamma - \bar{s}_\alpha$ error in meters.
- The vertical distance between the runout point and the end of the path profile defines the $\epsilon_z = \bar{z}_\gamma - \bar{z}_\alpha$ error in meters.
- The angle difference between the α line angle and the DFA simulation runout line defines the $\epsilon_\alpha = \gamma - \alpha$ angle error.
- The Root Mean Square Error (RMSE) between the α line and the DFA simulation energy points defines an error on the velocity altitude $\frac{\overline{v^2}}{2g}$.

Limitations and remarks

It is essential to stay where the assumptions of this test hold. Indeed, one of the important hypotheses when developing the energy solution, is that the material is flowing in the steepest slope direction (i.e. where $dl = \frac{ds}{cos(\theta)}$ theta holds). If this hypothesis fails (as illustrated in Fig. 5.10), then it is not possible to develop the analytic energy solution anymore. In the 3D case, the distance vector $\mathbf{dl}$ traveled by the particles reads $dl = \frac{ds}{cos(\gamma)}$, where γ is the angle between the $\mathbf{dl}$ vector and the horizontal plane which can differ from the slope angle θ. In this case, the energy solution is not the solution of the problem anymore and can not be used as reference.

It is also possible with this test to observe the effect of terms such as curvature acceleration, artificial viscosity or pressure gradients. The curvature acceleration modifies the friction term (depending on topography curvature and particle velocity). This leads to a mismatch between the energy solution and the DFA simulation. Artificial viscosity can lead to viscous dissipation leading to shorter runouts then what the energy solution predicts. Finally, the effect of the pressure force can be studied, especially the effect of the computation options. The results of this test can also be reused for other purposes or in other test such as in the rotation test described below (*Rotation test*)

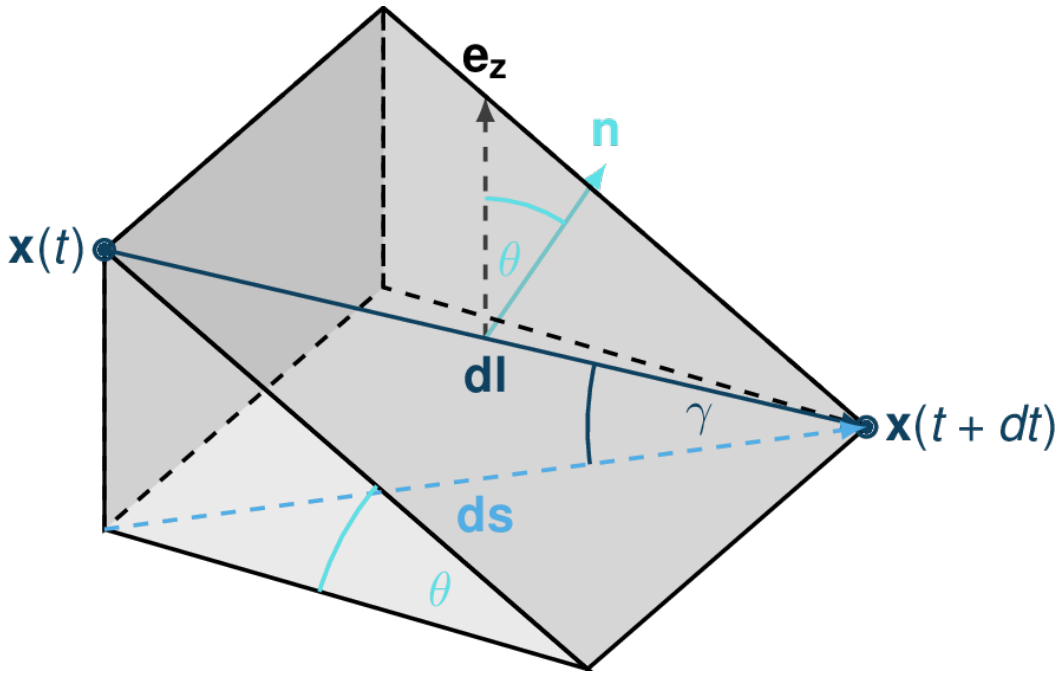


Fig. 5.10: Example of trajectory where the steepest descent path hypothesis fails. The mass point is traveling from $\mathbf{x}(t)$ to $\mathbf{x}(t+dt)$. The slope angle θ and travel angle γ are also illustrated. Here $(\mathbf{e_z}.\mathbf{n})dl = cos\theta\frac{ds}{cos\gamma} \neq ds$.

5.1.4.2 Procedure

First, the DFA simulation is ran (in our case using the com1DFA module) on the desired avalanche, saving the particles (at least the initial and final particles information). Then, the particles mass averaged quantities are computed ($\bar{x}, \bar{y}, \bar{z}, \bar{s}, \bar{v^2}$) to extract a path and path profile. Finally, the mass averaged path profile, the corresponding runout line and the expected α are displayed and the runout angle and distance errors as well as the velocity altitude error are computed.

5.1.4.3 To run energy line test

A workflow example is given in `runScripts.runEnergyLineTest.py`.

5.1.5 Rotation test

The rotation test aims at verifying that a DFA computation module produces similar results, if not identical, independently of the underlying mesh or grid orientation used for the computation. Indeed, numerical solvers using any sort of mesh based discretization or interpolation method tend to give different results for the same physical problem, boundary conditions and initial conditions if the mesh or grid orientation is different. In this test, the same physical problem is fed to the DFA module changing only the grid orientation. The energy line test is applied to each of the simulations and the runouts are compared. An AIMEC analysis of the peak results (previously rotated so that the results are aligned) is also carried out to give a more detailed idea of the spatial differences between the simulations.

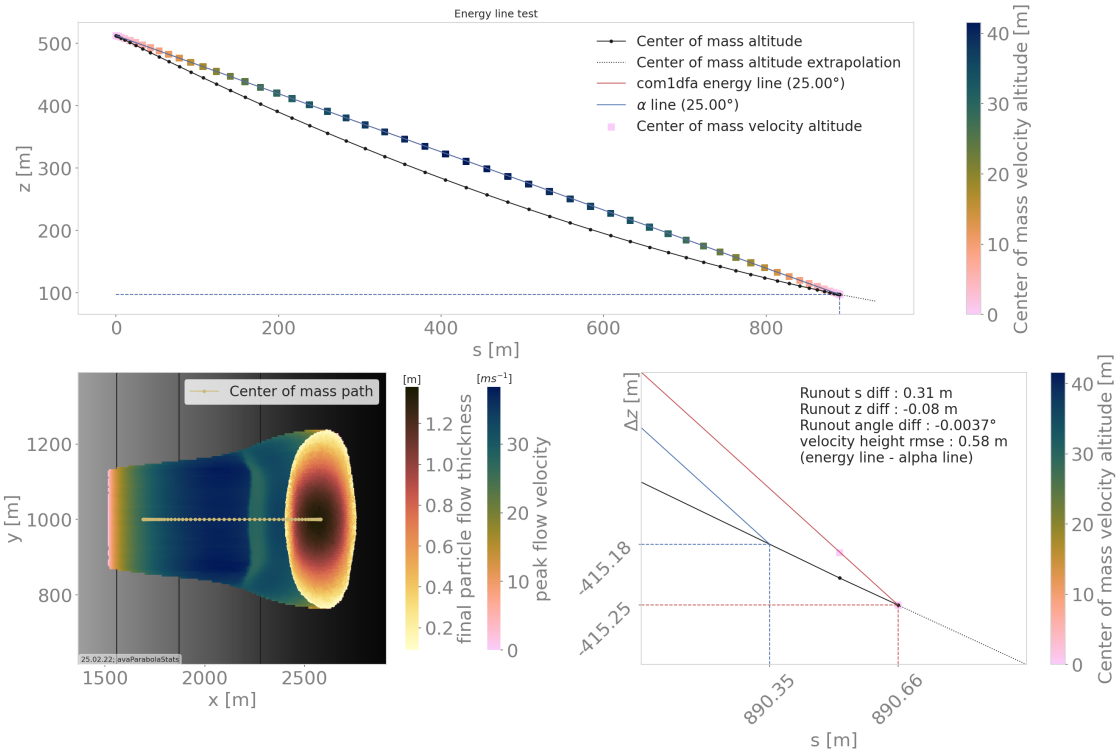


Fig. 5.11: Results from the ana1EnergyLineTest for the avaParabola

5.1.5.1 Inputs

This module requires an avalanche directory with inputs for the com1DFA and AIMEC module.

com1DFA inputs

- The DEM needs to have a center of symmetry located at the origin (0, 0).
- There needs to be multiple release features also symmetric in regard of the origin. These features need to be named relXXX.shp, where XXX is the rotation angle in the clockwise direction compared to the $[-\infty, 0]$ x axis.
- There can also be some entrainment or resistance features (also satisfying the symmetry criterion).

ana3AIMEC inputs

- The line describing the avalanche path for the reference simulation
- A split point.

The most simple example is a bowl centered on (0, 0) with some circular release features all located at the same distance from the origin and a ring shaped entrainment feature (as shown on Fig. 5.12).

One then needs to specify the the com1DFA configuration (through the `com1DFACfg.ini` and its local version) as well as the AIMEC configuration (through the `ana3AIMECCfg.ini` and its local version) and the path generation configuration (through the `pathGenerationCfg.ini` and its local version).

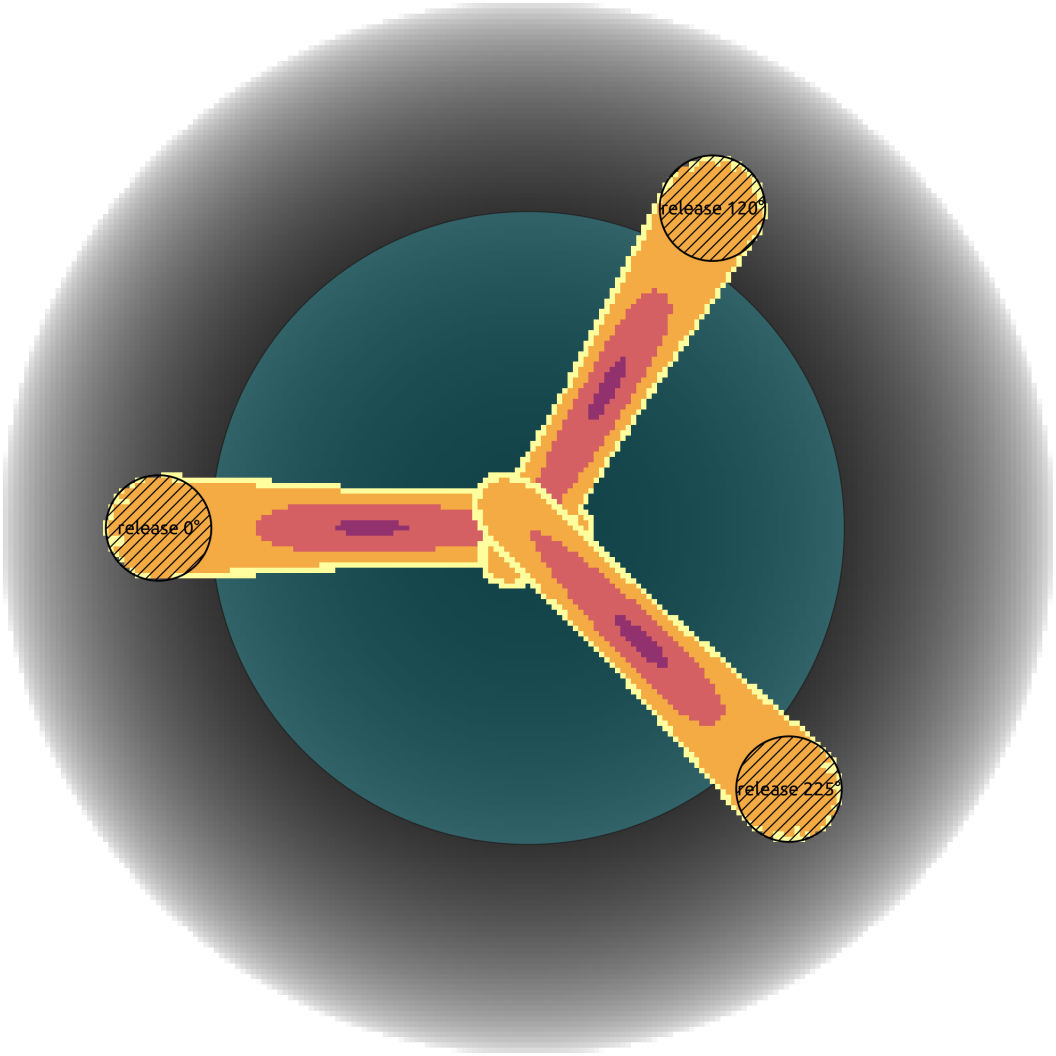


Fig. 5.12: Example of input data used for the rotation test and peak flow thickness result produced by the com1DFA module. In the background the bowl DEM is shown, in dashed areas (small circles) the three release features and in blue (big circle) the entrainment feature. One release feature is aligned with the x axis (rel0), the other is rotated 120° clockwise (rel120) and the last on 225° clockwise (rel225). All three input scenarios are identical (identical in terms of extend and release mass).

5.1.5.2 To run rotation test

A workflow example is given in `runScripts.runRotationTest.py`. Some example input data is given in avaTriple-Bowl (bowl topography).

5.2 ana3AIMEC: Aimec

ana3AIMEC (Automated Indicator based Model Evaluation and Comparison, [Fis13]) is a post-processing module to analyze and compare results from avalanche simulations. For this purpose, avalanche simulation results are transformed into an avalanche *thalweg* following coordinate system (see Fig. 5.13, Fig. 5.14, Fig. 5.15), which enables the comparison of different simulations (for example varying parameter sets, or performed with different models) of the same avalanche catchment, in a standardized way.

In `AvaFrame/avaframe/runScripts`, two different run scripts are provided and show examples on how the post-processing module *ana3AIMEC* can be used:

- full Aimec analysis for simulation results of one computational module (from 1 simulation to x simulations). `runScripts.runAna3AIMEC.runAna3AIMEC()`
- using Aimec to compare the results of two different computational modules (one reference for the reference computational module and multiple simulations in the other computational module). `runScripts.runAna3AIMECCompMods.runAna3AIMECCompMods()`

In all cases, one needs to provide a minimum amount of input data. Below is an example workflow for the full Aimec analysis, as provided in `runScripts/runAna3AIMEC.py`:

5.2.1 Inputs

- DEM (digital elevation model) as .asc file with ESRI grid format
- avalanche thalweg in LINES (as a shapefile named `NameOfAvalanche/Inputs/LINES/path_aimec.shp`), the line needs to cover the entire affected area but is not allowed to exceed the DEM extent
- results from avalanche simulation (when using results from com1DFA, the helper function `ana3AIMEC.dfa2Aimec.mainDfa2Aimec()` in `ana3AIMEC.dfa2Aimec` fetches and prepares the input for Aimec)
- a method to define the reference simulation. By default, an arbitrary simulation is defined as reference. This can be changed in the `ana3AIMEC/local_ana3AIMECCfg.ini` as explained in *Defining the reference simulation*.
- consider adjusting the default settings to your application in the Aimec configuration (in your local copy of `ana3AIMEC/local_ana3AIMECCfg.ini`) regarding the domain transformation, result types, runout computation and figures

---

Note: The spatial resolution of the DEM and its extent can differ from the result raster data. Spatial resolution can also differ between simulations. If this is the case, the spatial resolution of the reference simulation results raster is used (default) or, if provided, the resolution specified in the configuration file (`cellSizeSL`) is used. This is done to ensure that all simulations will be transformed and analyzed using the same spatial resolution.

5.2.1.1 Optional inputs

There is the option to define a runout area for the analysis, which is based on the slope angle of the avalanche thalweg. This can be defined in the Aimec configuration by setting the *defineRunoutArea* flag to True. The start of runout area is then determined by finding the first point on the thalweg profile where the slope angle falls below the *startOfRunoutAreaAngle* value and all points upslope of the a additionally provided split point are ignored (see Fig. 5.15). This functionality requires an additional input:

- a splitPoint in POINTS (as a shapefile named `NameOfAvalanche/Inputs/POINTS/splitPoint.shp`), this point is then snapped onto the thalweg based on the shortest horizontal distance

5.2.2 Outputs

- output figures in `NameOfAvalanche/Outputs/ana3AIMEC/anaMod/`
- csv file with the results in `NameOfAvalanche/Outputs/ana3AIMEC/anaMod/` (a detailed list of the results is described in *Analyze results*)

Note: *anaMod* refers to the computational module that has been used to create the avalanche simulation results and is specified in the **ana3AIMEC** configuration.

5.2.3 To run

- first go to `AvaFrame/avaframe`
- in your local copy of `ana3AIMEC/ana3AIMECCfg.ini` you can adjust the default settings (if not, the standard settings are used)
- enter path to the desired `NameOfAvalanche/` folder in your local copy of `avaframeCfg.ini`
- run:

```
python3 runScripts/runAna3AIMEC.py
```

Note: In the default configuration, the analysis is performed on the simulation result files located in `NameOfAvalanche/Outputs/anaMod/peakFiles`, where anaMod is specified in the aimecCfg.ini. There is also the option to directly provide a path to an input directory to the *ana3AIMEC.ana3AIMEC.fullAimecAnalysis()*. However, the peak field file names need to have a specific format: *A_B_C_D_E.asc*, where:

- A - *releaseAreaScenario*: refers to the name of the release shape file
- B - *simulationID*: needs to be unique for the respective simulation
- C - *simType*: refers to null (no entrainment, no resistance), ent (with entrainment), res (with resistance), entres (with entrainment and resistance)
- D - *modelType*: can be any descriptive string of the employed model (here dfa for dense flow avalanche)
- E - *result type*: is pft (peak flow thickness) and pfv (peak flow velocity)

5.2.4 Theory

AIMEC (Automated Indicator based Model Evaluation and Comparison, [Fis13]) was developed to analyze and compare avalanche simulations. The computational module presented here is inspired from the original AIMEC code. The simulations are analyzed and compared by projecting the results along a chosen poly-line (same line for all the simulations that are compared) called avalanche *thalweg*. The raster data, initially located on a regular and uniform grid (with coordinates x and y) is transformed based on a regular non uniform grid (grid points are not uniformly spaced) that follows the avalanche thalweg (with curvilinear coordinates (s,l)). This grid can then be "straightened" or "deskewed" in order to plot it in the (s,l) coordinates system.

The simulation results (two dimensional fields of e.g. peak flow velocities, pressure or flow thickness) are processed in a way that it is possible to compare characteristic values that are directly linked to the flow variables such as maximum peak flow thickness, maximum peak flow velocity or deduced quantities, for example maximum peak pressure, pressure based runout (including direct comparison to possible references, see *Area indicators*) for different simulations. The following figure illustrates the raster transformation process.

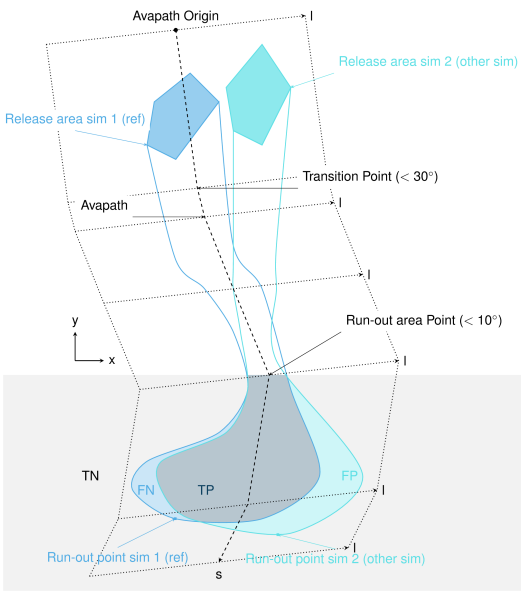


Fig. 5.13: In the real coordinate system (x,y)

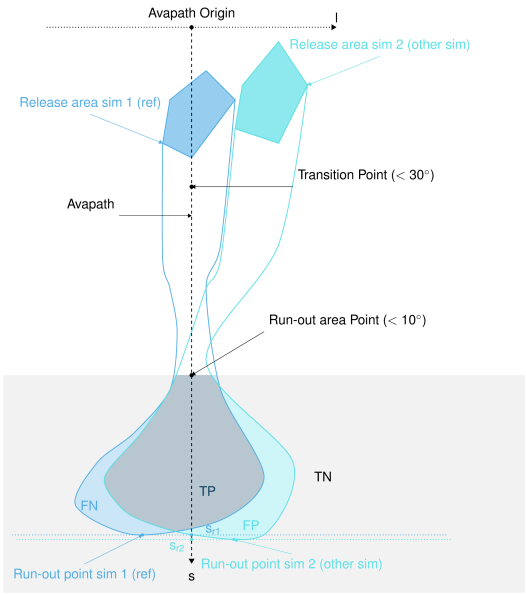


Fig. 5.14: In the new coordinate system (s,l)

Here is the definition of the different indicators and outputs from the AIMEC post-processing process:

5.2.4.1 Mean and max values along thalweg

All two dimensional field results (for example peak flow velocities / pressure or flow thickness) can be transformed into the curvilinear system using the previously described method. The maximum and average values of those fields are computed in each cross-section (l direction) along the thalweg. For example the maximum $A^{max}_{cross}(s)$ and average $\bar{A}_{cross}(s)$ of the two dimensional distribution $A(s,l)$ is:

$$A^{max}_{cross}(s) = \max_{\forall l \in [-\frac{w}{2}, \frac{w}{2}]} A(s,l) \quad \text{and} \quad \bar{A}_{cross}(s) = \frac{1}{w} \int_{-\frac{w}{2}}^{\frac{w}{2}} A(s,l) dl$$

5.2.4.2 Runout point

The runout point is always given with respect to a peak result field ($A(s,l)$ which could be peak pressure or flow thickness, etc.) and a threshold value ($A_{lim} > 0$). The runout point ($s = s_{runout}$) and the respective (x_{runout}, y_{runout}) in the original coordinate system, correspond to the last point in flow direction where the chosen peak result $A^{max}_{cross}(s)$ is above the threshold value A_{lim}.

Note: It is very important to note that the position of the runout point depends on the chosen threshold value and peak result field. It is also possible to use $\bar{A}_{cross}(s) > A_{lim}$ instead of $A^{max}_{cross}(s) > A_{lim}$ to define the runout point.

5.2.4.3 Runout length

This length depends on what is considered to be the beginning of the avalanche $s = s_{start}$. It can be related to the release area, to the transition point (first point where the slope angle is below 30°), to the runout area point (first point where the slope angle is below 10°) or in a similar way as $s = s_{runout}$ is defined saying that $s = s_{start}$ is the first point where $A^{max}_{cross}(s) > A_{lim}$ (this is the option implemented in `ana3AIMEC.ana3AIMEC.py`). The runout length is then defined as $L = s_{runout} - s_{start}$. In the analysis results, this is called $deltaSXY$, whereas s_{runout} gives the length measured from the start of the thalweg to the runout point.

5.2.4.4 Mean and max indicators

From the maximum values along path of the distribution $A(s,l)$ calculated in moduleAna3AIMEC:Mean and max values along path, it is possible to calculate the global maximum (MMA) and average maximum (AMA) values of the two dimensional distribution $A(s,l)$:

$$MMA = \max_{\forall s \in [s_{start}, s_{runout}]} A^{max}_{cross}(s) \quad \text{and} \quad AMA = \frac{1}{s_{runout} - s_{start}} \int_{s_{start}}^{s_{runout}} A^{max}_{cross}(s) ds$$

5.2.4.5 Area indicators

When comparing the runout area (corresponding to a given threshold $A^{max}_{cross}(s) > A_{Lim}$) of two simulations, it is possible to distinguish four different zones. For example, if the first simulation (sim1) is taken as reference and if True corresponds to the assertion that the avalanche reached this zone (reached means $A^{max}_{cross}(s) > A_{Lim}$) and False the avalanche did not reached this zone, those four zones are:

- TP (true positive) zone: green zone on Fig. 5.14 , sim1 = True sim2 = True
- FP (false positive) zone: blue zone on Fig. 5.14 , sim1 = False sim2 = True
- FN (false negative) zone: red zone on Fig. 5.14 , sim1 = True sim2 = False
- TN (true negative) zone: gray zone on Fig. 5.14 , sim1 = False sim2 = False

The two simulations are identical (in the runout zone) when the area of both FP and FN are zero. In order to provide a normalized number describing the difference between two simulations, the area of the different zones is normalized by the area of the reference simulation $A_{ref} = A_{TP} + A_{FP}$. This leads to the 4 area indicators:

- $\alpha_{TP} = A_{TP}/A_{ref}$, which is 1 if sim2 covers at least the reference
- $\alpha_{FP} = A_{FP}/A_{ref}$, which is a positive value if sim2 covers an area outside of the reference
- $\alpha_{FN} = A_{FN}/A_{ref}$, which is a positive value if the reference covers an area outside of sim2
- $\alpha_{TN} = A_{TN}/A_{ref}$ (this value may not be of great interest because it depends on the width and length of the entire domain of the result rasters (s,l))

Identical simulations (in the runout zone) lead to $\alpha_{TP} = 1$, $\alpha_{FP} = 0$ and $\alpha_{FN} = 0$

5.2.4.6 Mass indicators

From the analysis of the release mass (m_r at the beginning, i.e $t = t_{ini}$), total mass (m_t at the end, i.e $t = t_{end}$) and entrained mass (m_e at the end, i.e $t = t_{end}$) it is possible to calculate the growth index GI and growth gradient GG of the avalanche:

$$GI = \frac{m_t}{m_r} = \frac{m_r + m_e}{m_r} \quad \text{and} \quad GG = \frac{m_r + m_e}{t_{end} - t_{ini}}$$

Time evolution of the total mass and entrained one are also analyzed.

5.2.5 Procedure

This section describes how the theory is implemented in the `ana3AIMEC` module.

5.2.5.1 Defining the reference simulation

To apply a complete Aimec analysis, a reference simulation needs to be defined. The analysis of the other simulations will be compared to the one of the reference simulation. The reference simulation can be determined by its name (or part of the name) or based on some configuration parameter and value (to adjust in the local copy of `ana3AIMEC/ana3AIMECCfg.ini`) if it comes from the *com1DFA* module (or any computational module that provides a configuration):

- **based on simulation name**
  one needs to provide a non-empty string in the AIMEC configuration file for the `referenceSimName` parameter. This string can be a part or the full name of the reference simulation. A warning is raised if several simulation match the criterion (can happen if part of the name is given) and the first simulation found is arbitrarily taken as reference.
- **based on some configuration parameter**
  one needs to provide a `varParList` (parameter or list of parameters separated by `|`) in the AIMEC configuration file as well as the desired sorting order (`ascendingOrder`, True by default) for these parameters and optionally a `referenceSimValue`. The simulations are first going to be sorted according to `varParList` and `ascendingOrder` (this is done by the pandas function sort_values). The reference simulation is either the first simulation found after sorting if no `referenceSimValue` or the simulation matching the `referenceSimValue` provided (closest value if the parameter is a float or integer, case insensitive for strings). If multiple simulations match the criterion, the first simulation is taken as reference and a warning is raised.

5.2.5.2 Perform thalweg-domain transformation

First, the transformation from (x,y) coordinate system (where the original rasters lie in) to (s,l) coordinate system is applied given a new domain width (`domainWidth`). This is done by `ana3AIMEC.aimecTools.makeDomainTransfo()`. A new grid corresponding to the new domain (following the avalanche thalweg) is built with a cell size defined by the reference simulation (default) or `cellSizeSL` if provided. The transformation information are stored in a `rasterTransfo` dictionary (see `ana3AIMEC.aimecTools.makeDomainTransfo()` for more details).

5.2.5.3 Assign data

The simulation results (for example peak velocities / pressure or flow thickness) are projected on the new grid using the transformation information by `ana3AIMEC.aimecTools.assignData()`. The projected results are stored in the `newRasters` dictionary.

This results in the following plot:

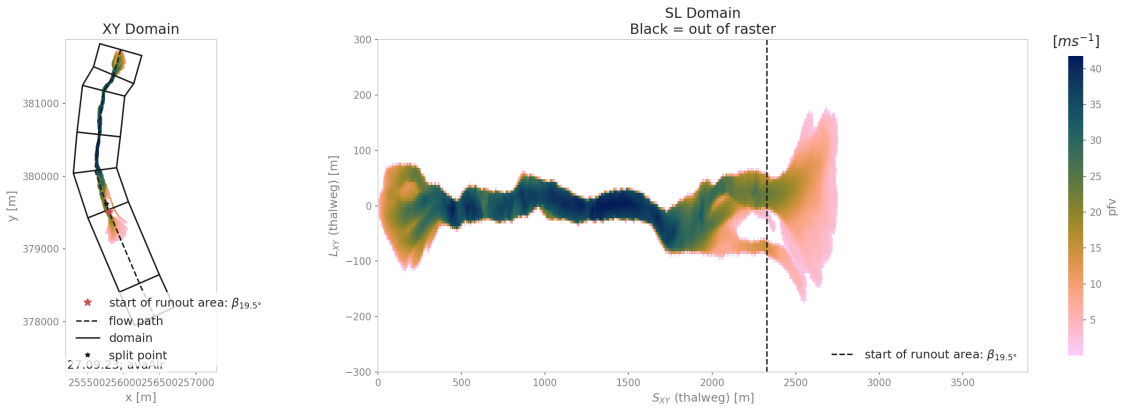


Fig. 5.15: Alr avalanche coordinate transformation and peak pressure field reprojetion.

5.2.5.4 Analyze results

Calculates the different indicators described in the *Theory* section for a given threshold. The threshold can be based on pressure, flow thickness, … (this needs to be specified in the configuration file). Returns a `resAnalysisDF` dataFrame with the analysis results (see *`ana3AIMEC.ana3AIMEC.postProcessAIMEC()`* for more details). In this dataFrame there are multiple columns, one for each result from the analysis (one column for runout length, one for MMA, MAM…) and one row for each simulation analyzed.

5.2.5.5 Plot and save results

Plots and saves the desired figures and writes results in `resAnalysisDF` to a csv file. By default, Aimec saves five summary plots plus three plots per simulation comparing the numerical simulations to the reference. The five summary plots are:

5.2.5.6 Domain overview plots

- "DomainTransformation" shows the computational domain (xy) on the left and new thalweg-following domain (sl) on the right (Fig. 5.15)
- "referenceFields" shows the peak pressure, flow thickness and flow velocity in the sl domain

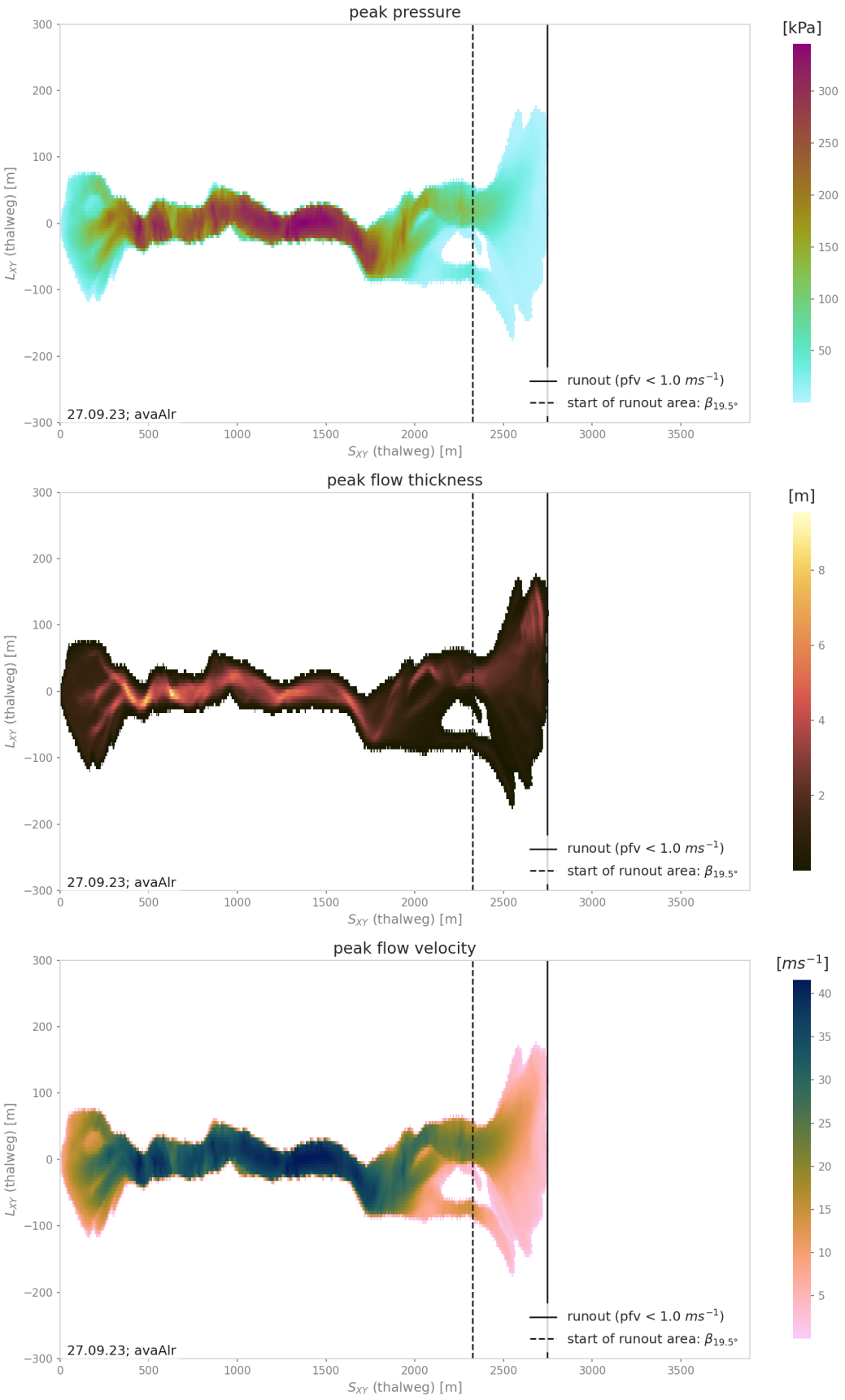


Fig. 5.16: Reference peak fields

5.2.5.7 Analysis summary plots

- "slComparisonStats" shows the peak field specified in the configuration and computed runout points in the top panel, wherease in the bottom panel the statistics in terms of cross maximum peak value along profile are plotted (mean, max and quantiles)

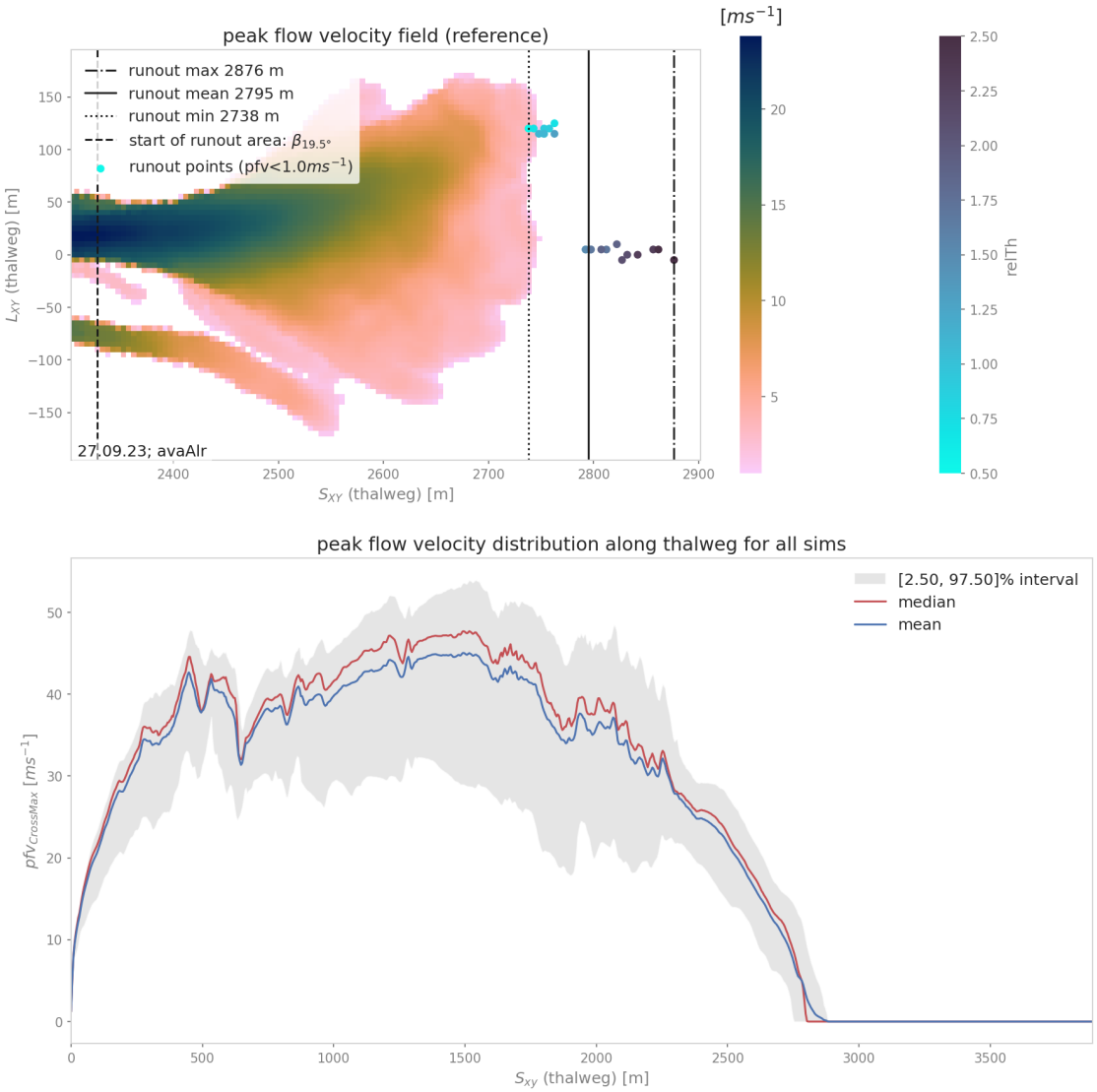


Fig. 5.17: Reference peak field runout area with runout points and distribution of cross max values of peak field along thalweg

- if only two simulations are compared, additionally the difference between the two simulations in terms of peak values along profile are shown for peak pressure, peak flow velocity and peak flow thickness in "slComparison".
- "ROC" shows the normalized area difference between reference and other simulations.
- "deltaSXvsrunoutAngle" and "pfvFieldMaxvspftFieldMax" show the distribution of these values for all the simulations. There is the option to indicate scenarios in these plots using the available simu-

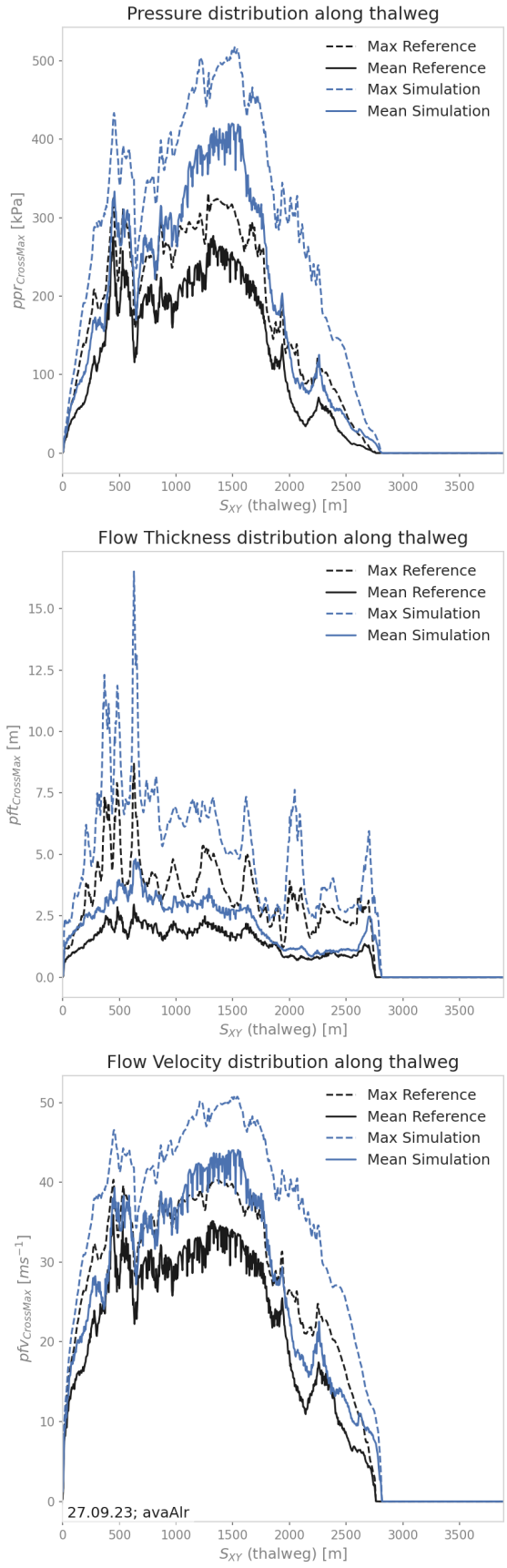


Fig. 5.18: Maximum peak fields comparison between two simulations

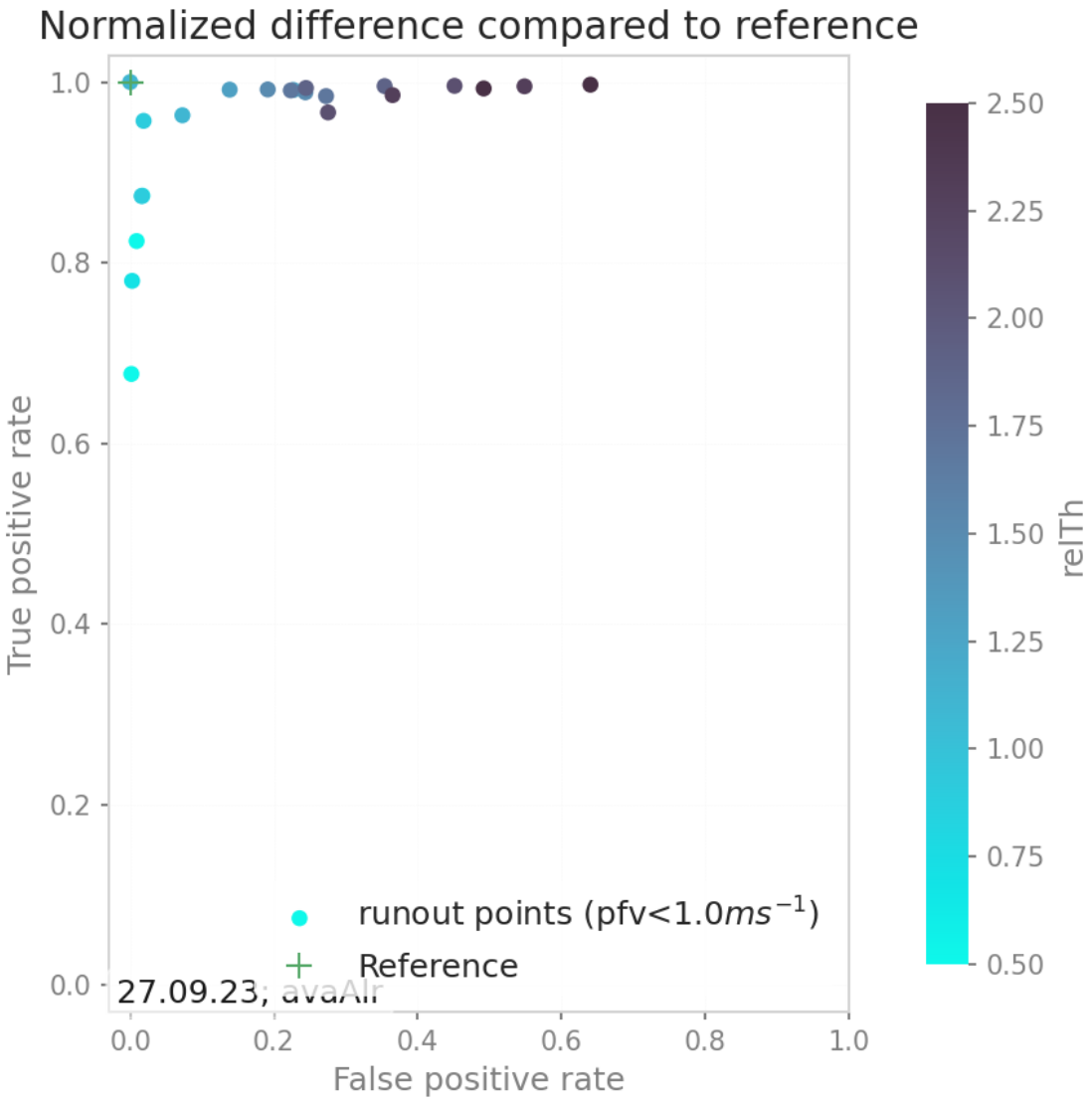


Fig. 5.19: Area analysis plot

lation parameters Acutally other parameter vs parameter plots can be added by setting the parameter names in the **ana3AIMEC** configuration in the section called PLOTS.

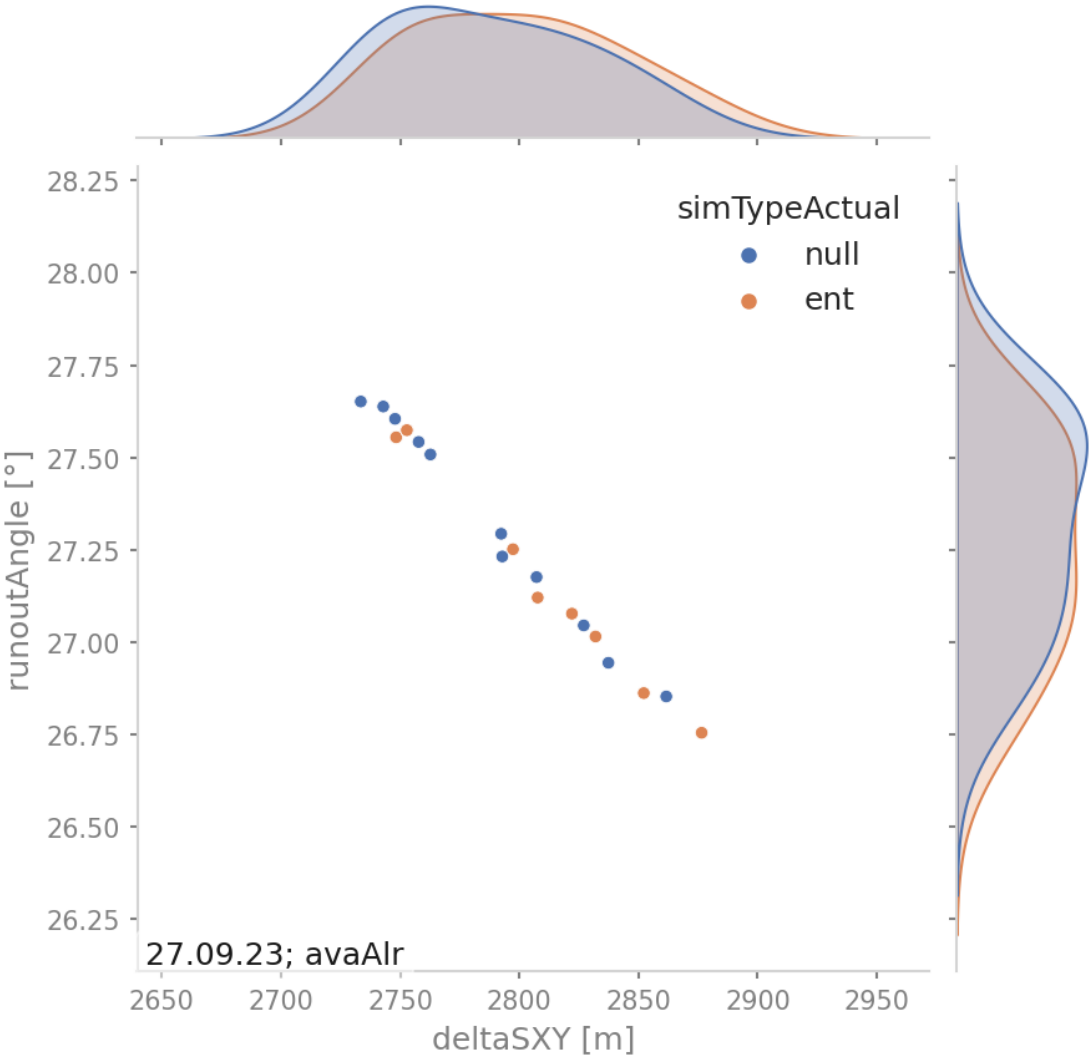


Fig. 5.20: DeltaSXY vs runout angle of all sims colorcoded using the simType

5.2.5.8 Analysis on simulation level plots

- "thalwegAltitude" shows the cross section max values of peak flow thickness as bars on top of the thalweg elevation profile. The colors of the bars correspond to the cross section max values of the respective peak flow velocity. Using the specified *velocityThresholdValue* (aimecCfg.ini section PLOTS), runout length $/DeltaS_{xy}$, elevation drop $/Deltaz$ and the corresponding runout angle α are computed. The red dots indicate the location of overall max peak flow velocity and thickness and the grey shaded area the runout area.

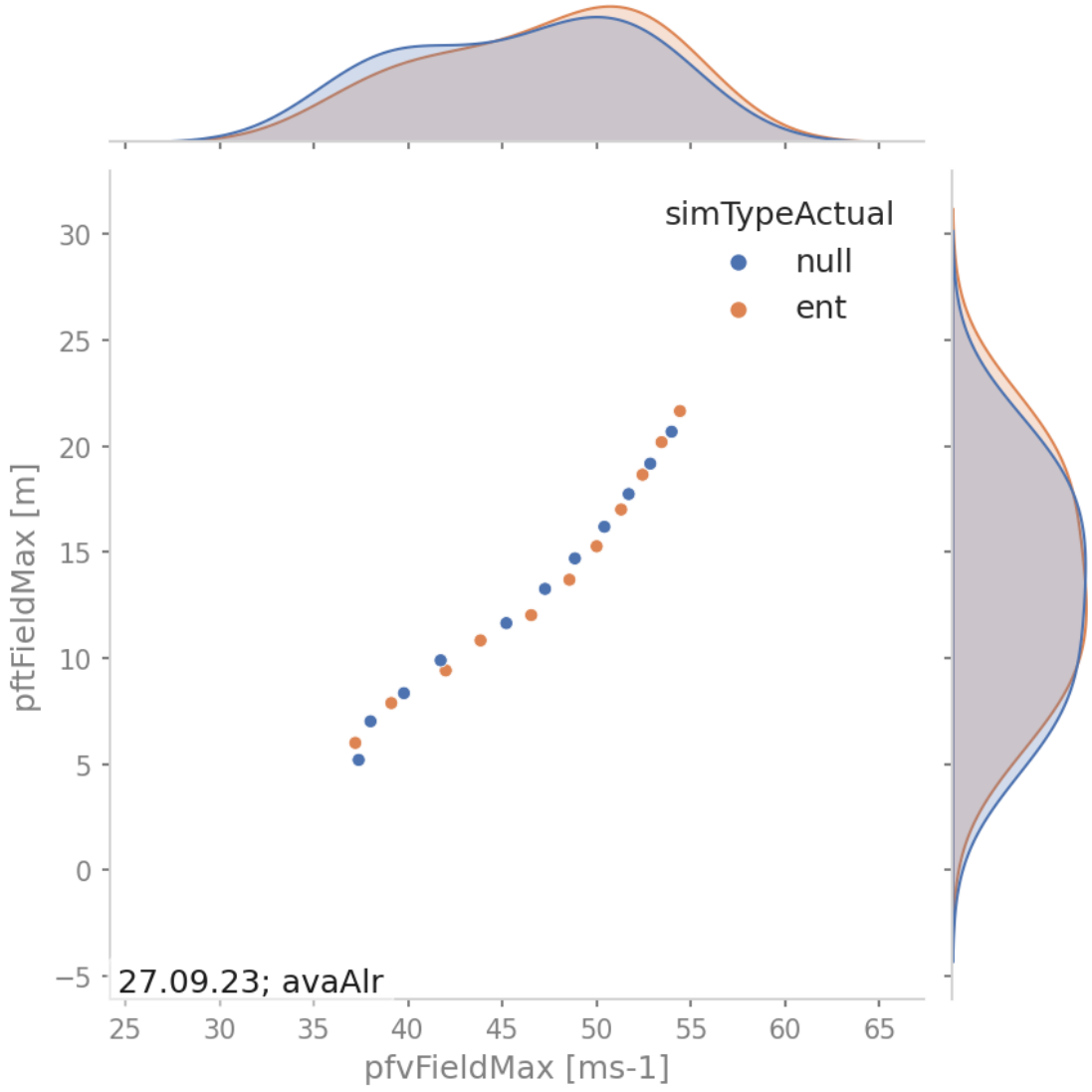


Fig. 5.21: Max field values of peak flow velocity vs flow thickness of all sims colorcoded using the simType

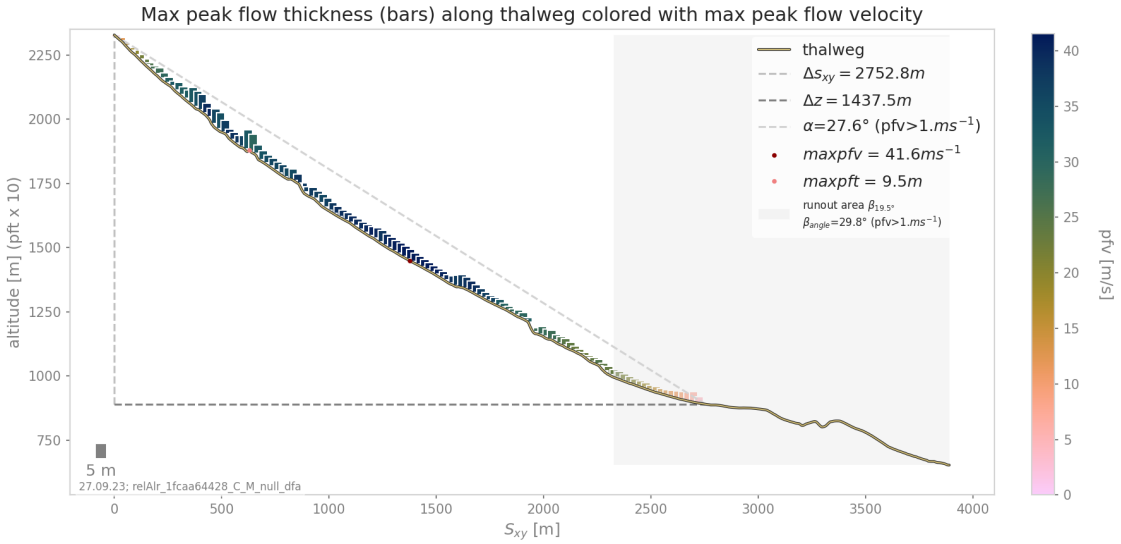


Fig. 5.22: Thalweg altitude plot

5.2.5.9 Comparison to reference simulation plots

The plots "_hashID_ContourComparisonToReference" and "_hashID_massAnalysis" where "hashID" is the name of the simulation show the 2D difference to the reference, the statistics associated and the mass analysis figure (this means these figures are created for each simulation).

5.2.5.10 List of Aimec result variables

The result variables listed below are partly included in the above described plots and are also saved to the Aimec results directory as *..resAnalysisDF.csv*, with one row per simulation. In the following, the resType represents the result type(s) set in the aimecCfg.ini for the parameter *resTypes*.

- resTypeFieldMax: maximum value of resType simulation result (simulation coordinate system)
- resTypeFieldMin: minimum value of resType simulation result (simulation coordinate system), values smaller than a threshold (minValueField in aimecCfg.ini) are masked
- resTypeFieldMean: average value of resType simulation result (simulation coordinate system), values smaller than a threshold (minValueField in aimecCfg.ini) are masked
- resTypeFieldStd: standard deviation of resType simulation result (simulation coordinate system), values smaller than a threshold (minValueField in aimecCfg.ini) are masked
- maxresTypeCrossMax: maximum value of cross profile maximum values of resType field (transformed in thalweg-following coordinate system)
- sRunout: the last point along the thalweg (S_{XY}) where the transformed resType field (cross profile maximum values) still exceeds a threshold value (thresholdValue in aimecCfg.ini)
- lRunout: the cross profile coordinate (L_{XY}) of the last point along the thalweg (S_{XY}) where the transformed resType field (cross profile maximum values) still exceeds a threshold value (thresholdValue in aimecCfg.ini)
- xRunout, yRunout: x and y coordinates of the *sRunout* point (simulation coordinate system)
- sMeanRunout, xMeanRunout, yMeanRunout: runout point coordinates derived with cross profile mean values of transformed resType field instead of cross profile maximum values

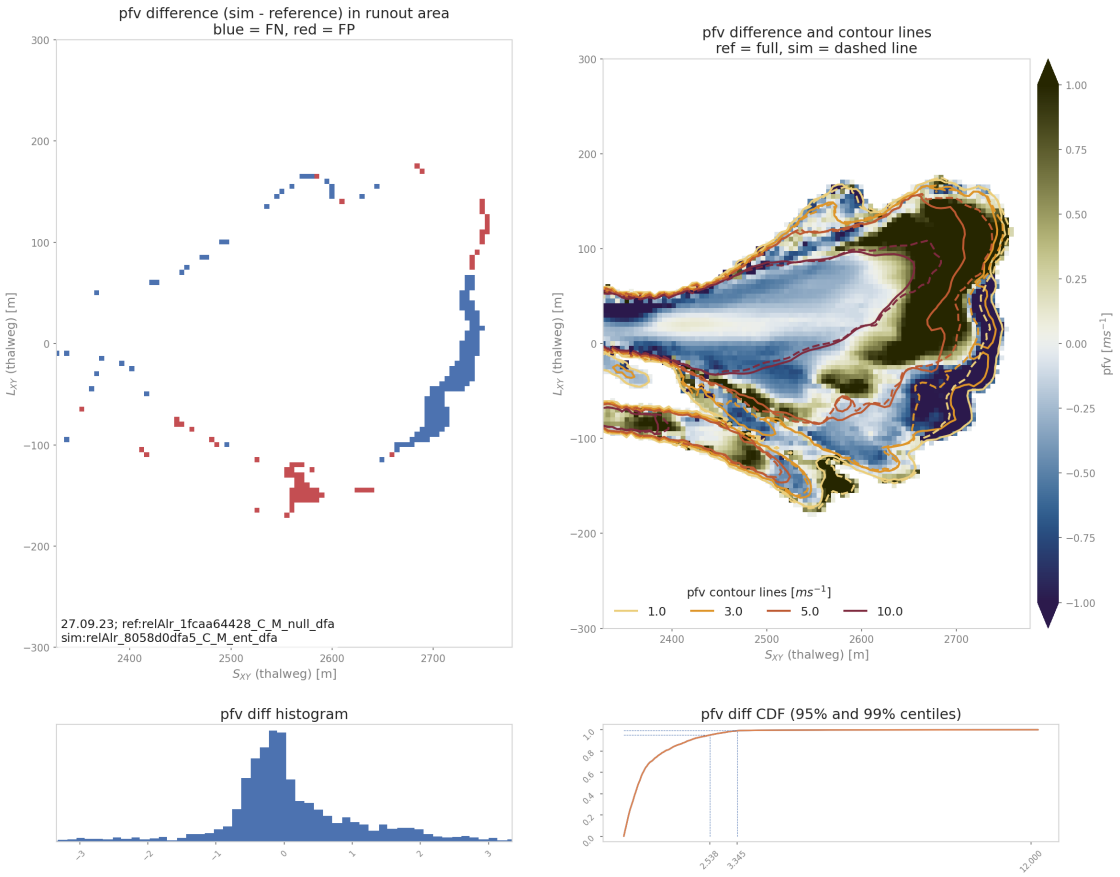


Fig. 5.23: The area comparison panel (top left) shows the false negative (FN in blue, which is where the reference field exceeds the threshold but not the simulation) and true positive (TP in red, which is where the simulation field exceeds the threshold but not the reference) areas. The contour comparison panel (top right) shows the contour lines of the reference (full lines) and the simulation (dashed lines) of the desired result fields in the runout area. It also shows the difference between the reference and simulation and computes the repatriation of this difference (Probability Density Function and Cumulative Density Function of the difference)

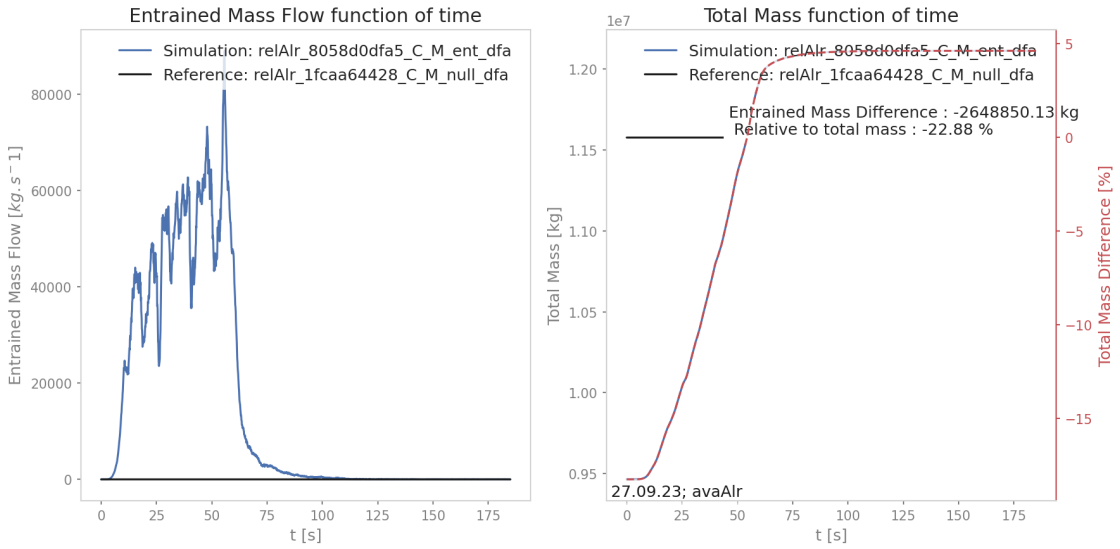


Fig. 5.24: The mass analysis plot shows the evolution of the total and entrained mass during the simulation and compares it to the reference

- deltaSXY: distance along thalweg (S_{XY}) where the transformed resType field exceeds a threshold value (thresholdValue in aimecCfg.ini)
- zRelease: altitude along thalweg where the transformed resType field first exceeds a threshold value (thresholdValue in aimecCfg.ini)
- zRunout: altitude of the *sRunout* point
- deltaZ: altitude difference between *zRelease* and *zRunout*
- runoutAngle: corresponding runout angle based on *deltaSXY* and *deltaZ*

5.2.6 Configuration parameters

All configuration parameters are explained in `ana3AIMEC/ana3AIMECCfg.ini` (and can be modified in a local copy `ana3AIMEC/local_ana3AIMECCfg.ini`):

```
### Config File - This file contains the main settings for the ana3AIMEC run
## Set your parameters
# This file is part of Avaframe.
# This file will be overridden by local_ana3AIMEC.ini if it exists
# So copy this file to local_ana3AIMEC.ini, adjust your variables there

# Optional settings------------------------------
[AIMECSETUP]
# width of the domain around the avalanche path in [m]
domainWidth = 600

# The cell size for the new (s,l) raster is automatically computed from the reference␣
↪result file (leave the following field empty)
# It is possible to force the cell size to take another value, then specify this new␣
↪value below, otherwise leave empty (default).
cellSizeSL =
```

(continues on next page)

(continued from previous page)

```
# define a runout area based on an angle of the thalweg profile (requires splitPoint in␣
↪avaName/Inputs/POINTS as shpFile)
defineRunoutArea = True
# only used if defineRunoutArea=True - angle for the start of the run-out zone
startOfRunoutAreaAngle = 10

# data result type for general analysis (ppr|pft|pfd). If left empty takes the result␣
↪types available for all simulations
resTypes = ppr|pft|pfv

# data result type for runout analysis (ppr, pft, pfv)
runoutResType = ppr

# limit value for evaluation of runout (depends on the runoutResType chosen)
thresholdValue = 1

# contour levels value for the difference plot (depends on the runoutResType chosen)
# use | delimiter (for ppr 1|3|5|10, for pft 0.1|0.25|0.5|0.75|1)
contourLevels = 1|3|5|10

# max of runoutResType difference for contour lines plus capped difference in␣
↪runoutResType plot (for example 1 (pft), 5 (ppr))
diffLim = 1

# percentile to display when analyzing the peak values along profile
# for example 5 (corresponding to 5%) will lead to the interval [2.5, 97.5]%
percentile = 5
# chose interpolation method between 'nearest' and 'bilinear'
interpMethod = bilinear

# threshold distance [m]. When looking for the beta point make sure at least
# dsMin meters after the beta point also have an angle bellow 10°
dsMin = 30

# computational module that was used to produce avalanche simulations (to locate␣
↪peakFiles)
anaMod = com1DFA


# two computational module that were used to produce avalanche simulations (to locate␣
↪peakFiles) for comparison separated by |
comModules =

# if a computation module is benchmark, specify the test name (we assume that the␣
↪testName folder is in AvaFrame/benchmarks/)
testName =

# parameter used for ordering the simulations - multiple possible; (e.g.relTh|deltaTh)
varParList =
# True if ascending ordering of simulations for varPar, False if descending order
ascendingOrder = True
```

(continues on next page)

(continued from previous page)

```
# parameter value (first parameter in varParList) that should be used as reference␣
↪simulation (e.g. 1.0)
referenceSimValue =
# OR directly set reference simulation by its name (name of simulation result file or␣
↪parts of it that definitively
# identify one particular simulation)
referenceSimName =
# unit of desired result parameter (e.g. m)
unit =

#----------------------------------------
## Uncomment this section FILTER in your local copy of the ini file and add filter␣
↪parameter and parameter values
## see the example provided below for release thickness
#[FILTER]
## define parameter and corresponding values from the simulation configuration to filter␣
↪simulations
## multiple parameters are possible, just add them in a new line each
##relTh = 0.75|0.9|0.8


[PLOTS]
# if extraPlots true additional analysis plots are created
extraPlots = True
# for one to one comparison of two result variables or derived quantities with option to␣
↪distinguish scenarios using scenario
# comparison result variables options: resTypeFieldMax (or Min or Mean),␣
↪maxresTypeCrossMax, sRunout, deltaSXY, zRelease,
# zRunout, deltaH, relMass, finalMass, entMass, thalwegTravelAngle
# comparison result variable 1
compResType1 = pfvFieldMax|deltaSXY
# comparison result variable 2
compResType2 = pftFieldMax|runoutAngle
# scenario parameter name used to colorcode comparison plots
scenarioName =
# interval of cross max values of pft to create bars in profile plot
barInterval = 25
# threshold of velocity to compute alpha angle
velocityThreshold = 1.



# plot save results flags------------------------
[FLAGS]
# Which value to plot in resultVisu
# 1 = mean pressure data
# 2 = growth index
# 3 = max pressure data
typeFlag = 3
# number of simulations above which a (additional) density plot is created
nDensityPlot = 100
```

(continues on next page)

(continued from previous page)

```
# Mass analysis
flagMass = True

#-----------------------------------------------------------
```

5.3 ana4Stats: Statistical analysis tools

5.3.1 probAna - Probability maps

probAna is used to derive simple probability maps for a set of simulations for one avalanche track. These maps show for each point in space the fraction of simulations where a chosen parameter has exceeded a given threshold. For example, it is possible to compute the probability map for an avalanche with respect to a peak pressure threshold of 1kPa, but it is also possible to chose other result variables and threshold values.

A set of multiple avalanche simulations is required to generate these maps. The simulations can be generated with *com1DFA* using a parameter variation, different release-, entrainment- or resistance scenarios. An example run script is given by `runAna4ProbAna`: where firstly, *com1DFA* is used to perform avalanche simulations varying parameters set in the `ana4Stats/probAnaCfg.ini` file. Using these simulations, a probability map is generated. The output is a raster file (.asc) with values ranging from 0-1. 0 meaning that no simulation exceeded the threshold in this point in space. 1 on the contrary means that all simulations exceeded the threshold. Details on this function, as for example required inputs can be found in: *ana4Stats.probAna*.

5.3.1.1 To run - via QGis Connector

Since version 1.6 it is possible to generate probability maps via the QGis connector. This is provided in the experimental folder of the QGis processing plugin via **Probability run**. A standard setup is used in which mu and release thickness are varied. Please note: *samosAT* friction is used instead of the default *samosATAuto* one.

The input requirements are the same as for *com1DFA* with one important difference: an additional *ci95* attribute is needed for the release shapes. This describes the confidence intervall for the release thickness, using the same units. I.e. if you are using a release thickness of 1.5m, you need to provide the ci in [m] as well, for example ci=0.3m. The release thickness is then automatically varied within the release thickness plus/minus the confidence interval.

5.3.1.2 To run - example run scripts

In `runProbAnaCom1DFA.py`, avalanche simulations are performed with the settings defined in the configuration file of *com1DFA* and in addition a parameter variation is performed according to the parameters set in `ana4Stats/probAnaCfg.ini` in the section PROBRUN. The parameters to be varied are set in **varParList**, the type of variation in **variationType** (options are: percent, range, rangefromci) and the value of the variation in **variationValue**. Then there are two sampling strategies to choose from, for performing the parameter variation: (1) a latin hypercube sample of all the parameters to be varied (provided in varParList) is generated and simulations are performed using sets of parameters drawn from this sample. Using sampling strategy (2) all the parameters set in *PROBRUN* are varied on at a time, i.e. simulations are performed for the standard settings of all parameters, except the one parameter to be varied, subsequently the other variations are performed. One probability map is created for all the different simulations and in case of sampling strategy (2), also one map per parameter that is varied once at a time, is created in addition. In order to run this example:

- first go to `AvaFrame/avaframe`
- copy `avaframeCfg.ini` to `local_avaframeCfg.ini` and set your desired avalancheDir

- copy `ana4Stats/probAnaCfg.ini` to `ana4Stats/local_probAnaCfg.ini` and optionally adjust variation settings
- uncomment `'FILTER'` section in `local_probAnaCfg.ini` and insert filter parameters if you want to first filter simulations
- run:

```
python3 runAna4ProbAna.py
```

Another example on how to generate probability maps for avalanche simulations performed with *com1DFA* is given in `runScripts.runProbAna`, where for *avaHockeyChannel* simulations are performed with varying release thickness values ranging from 0.75 to 1.75 meters in steps of 0.05 meters. The resulting simulations are then used to generate the probability map with *out3Plot.statsPlots.plotProbMap()*. There is also the option to filter the simulations further - using the function *in3Utils.fileHandlerUtils.getFilterDict()* which generates a parameter dictionary for filtering according to the filter criteria set in the configuration file (`ana4Stats/probAnaCfg.ini`) of the *ana4Stats.probAna* function. In order to run this example:

- first go to `AvaFrame/avaframe`
- copy `ana4Stats/probAnaCfg.ini` to `ana4Stats/local_probAnaCfg.ini`
- uncomment `'FILTER'` section in `local_probAnaCfg.ini` and insert filter parameters if you want to first filter simulations
- run:

```
python3 runScripts/runProbAna.py
```

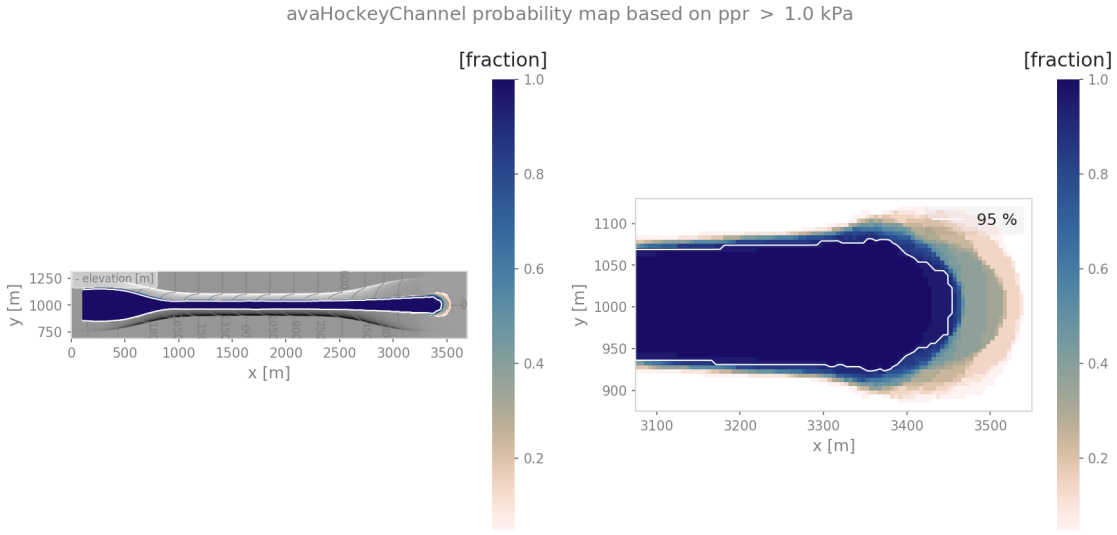


Fig. 5.25: Probability map example.

5.3.1.3 Theory

This point-wise probability is expressed by the relative frequency of avalanche peak flow field exceeding a certain threshold for a set of deterministic avalanche simulations derived from a range of input parameters (see [HBB19]).

5.3.2 getStats

In *ana4Stats.getStats*, functions that help to compute statistical properties of simulation results are gathered. *ana4Stats.getStats.extractMaxValues()* can be used to determine the maximum peak values of the simulation results. These values can then be plotted using the functions in *out3Plot.statsPlots* in order to visualise the statistics of a set of avalanche simulations. For further details on the specific functions, have a look at: *ana4Stats.getStats*.

5.3.2.1 To run

An example on how to use these statistical functions is given in `runScripts.runStatsExample`, where for `avaHockeyChannel` simulations are performed for two different release area scenarios and the release thickness is varied from 0.75 to 1.75 meters in steps of 0.05 meters. The resulting simulations are then analysed using the *ana4Stats.getStats.extractMaxValues()* function and plots are generated using the plotting routines from *out3Plot.statsPlots* . If in the configuration file `ana4Stats/getStats.ini` the flag aimec is set to True, additionally an *ana3AIMEC: Aimec* analysis is performed.

- first go to `AvaFrame/avaframe`
- copy `ana4Stats/getStats.ini` to `ana4Stats/local_getStatsCfg.ini`
- uncomment `'FILTER'` section in `ana4Stats/local_getStatsCfg.ini` and insert filter parameters if you want to first filter simulations
- run:

```
python3 runScripts/runStatsExample.py
```

5.4 ana5Utils

5.4.1 distanceTimeAnalysis: Visualizing the temporal evolution of flow variables

With the functions gathered in this module, flow variables of avalanche simulation results can be visualized in a distance versus time diagram, the so called **thalweg-time diagram**. The **tt-diagram** provides a way to identify main features of the temporal evolution of flow variables along the avalanche *path*. This is based on the ideas presented in [FFGS13] and [RK20], where avalanche simulation results have been transformed into the radar coordinate system to facilitate direct comparison, combined with the attempt to analyze simulation results in an avalanche path dependent coordinate system ([Fis13]). In addition to the **tt-diagram**, *ana5Utils.distanceTimeAnalysis* also offers the possibility to produce simulated **range-time diagrams** of the flow variables with respect to a radar field of view. With this, simulation results can be directly compared to radar measurements (for example moving-target-identification (MTI) images from [KMS18]) in terms of front position and inferred approach velocity. The colorcoding of the simulated **range-time** diagrams show the average values of the chosen flow parameter (e.g. flow thickness (FT), flow velocity (FV)) at specified range gates. This colorcoding is not directly comparable to the MTI intensity given in the range-time diagram from radar measurements.

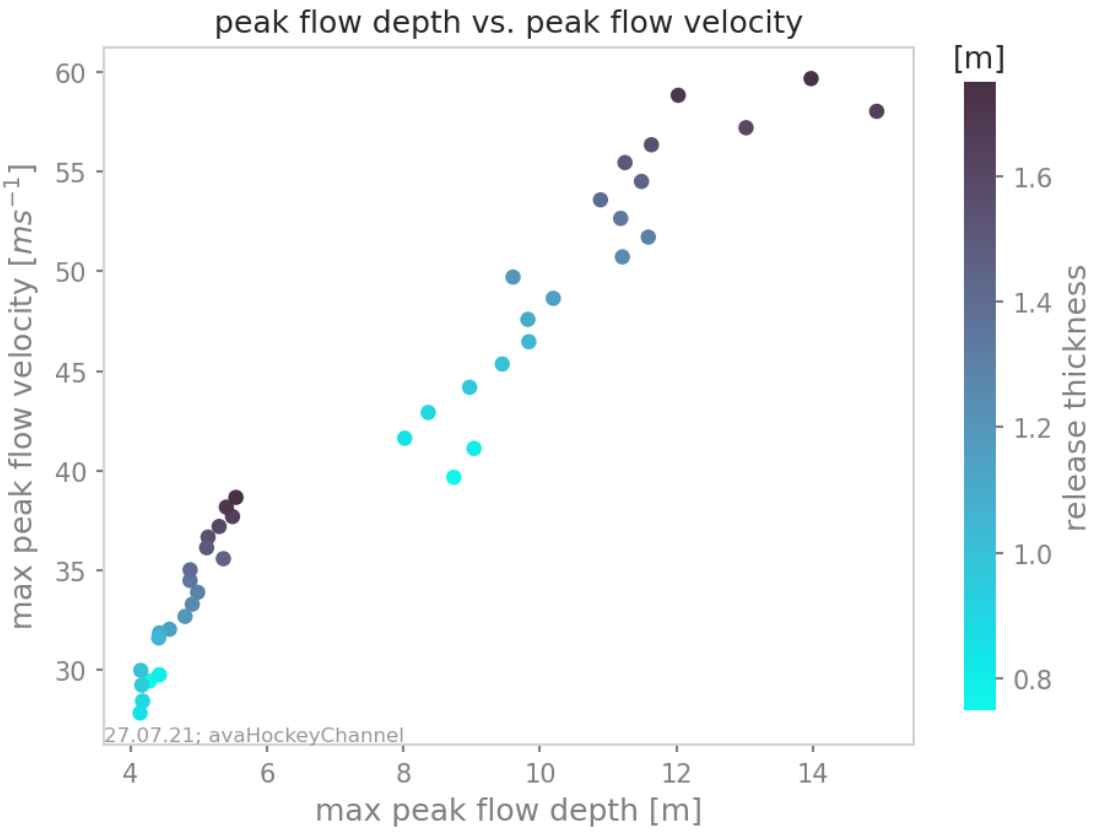


Fig. 5.26: Scatter plot of the hockey example with color-coded release thickness values.

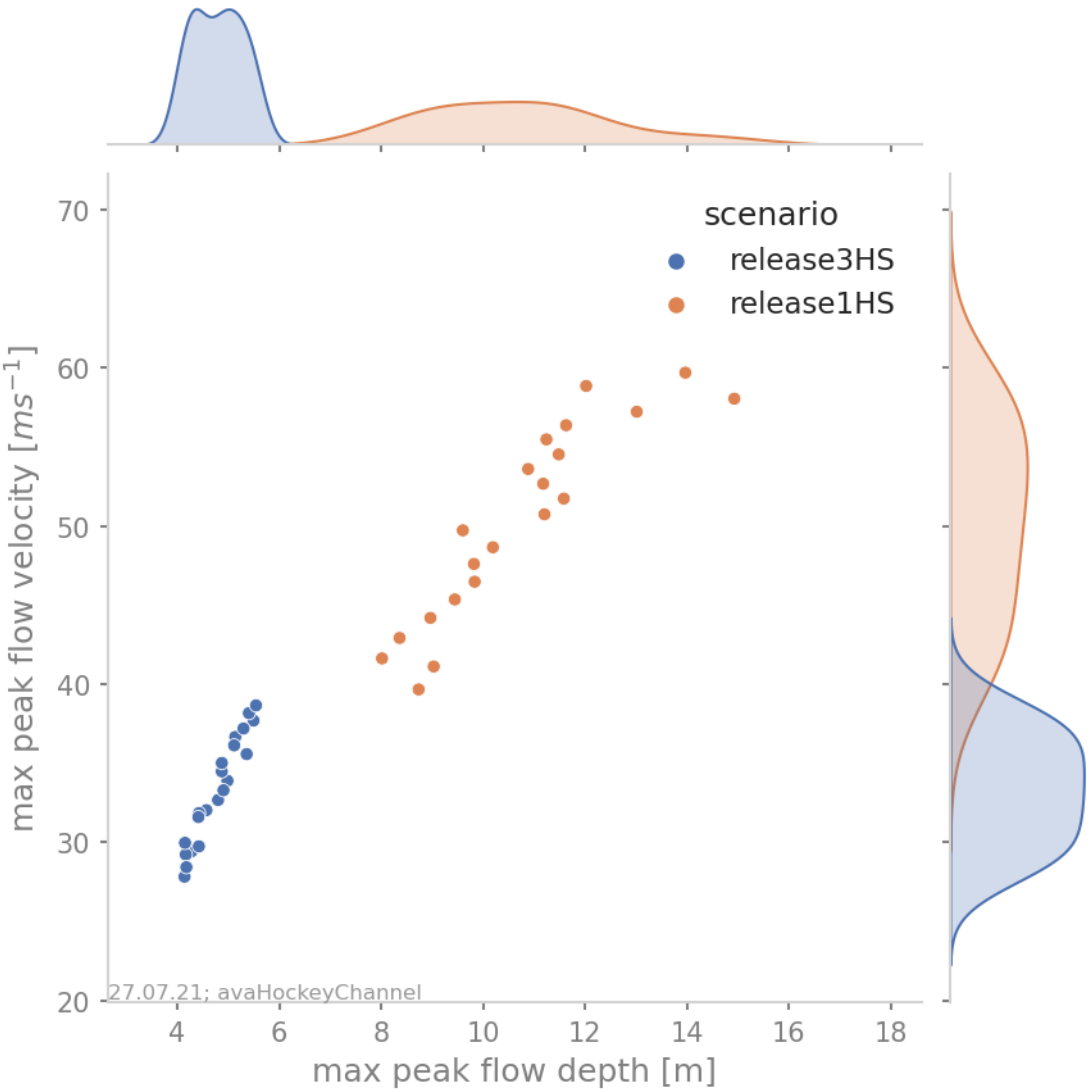


Fig. 5.27: Scatter plot of the hockey example including a marginal kde plot and color coded with release area scenario.

Note: The data processing for the **tt-diagram** and the **range-time diagram** can be done *during run time* of *com1DFA*, or as a postprocessing step. However, the second option requires first saving and then reading all the required time steps of the flow variable fields, which is much more computationally expensive compared to the first option.

5.4.1.1 To run

During run-time of *com1DFA*:

- in your local copy of `com1DFA/com1DFACfg.ini` in [VISUALISATION] set *createRangeTimeDiagram* to True and choose if you want a *TTdiagram* by setting this flag to True or in the case of a simulated range-time diagram to False
- in your local copy of `ana5Utils/distanceTimeAnalysisCfg.ini` you can adjust the default settings for the generation of the diagrams
- run `runCom1DFA.py` to calculate mtiInfo dictionary (saved as pickle in `avalancheDir/Outputs/com1DFA/distanceTimeAnalysis/mtiInfo_simHash.p`) that contains the required data for producing the **tt-diagram** or **range-time diagram**
- run `runScripts.runThalwegTimeDiagram.py` or `runScripts.runRangeTimeDiagram.py` and set the *preProcessedData* flag to *True*

As a postprocessing step:

- first you need to run *com1DFA* to produce fields of the desired flow variable (e.g. FT, FV) of sufficient temporal resolution (every second), for this in your local copy of *com1DFACfg.ini* add e.g. FT to the *resType* and change the *tSteps* to *0:1*
- have a look at `runScripts.runThalwegTimeDiagram.py` and `runScripts.runRangeTimeDiagram.py`
- in your local copy of `ana5Utils/distanceTimeAnalysisCfg.ini` you can adjust the default settings for the generation of the diagrams

The resulting figures are saved to `avalancheDirectory/Outputs/ana5Utils`.

Note: The **tt-diagram** requires info on an avalanche path (see *ana3AIMEC: Aimec*). The simulated **range-time diagram** requires info on the coordinates of the radar location (x0, y0), a point in the direction of the field of view (x1, y1), the aperture angle and the width of the range gates. The maximum approach velocity is indicated in the distance-time diagrams with a red star and is computed as the ratio of the distance traveled by the front and the respective time needed for a time step difference of at least *minVelTimeStep* which is set to 2 seconds as default. The approach velocity is a projection on the horizontal plane since the distance traveled by the front is also measured in this same plane.

5.4.1.2 Theory

5.4.1.3 Thalweg-time diagram

First, the flow variable result field is transformed into a path-following coordinate system, of which the centerline is the avalanche *path*. For this step, functions from *ana3AIMEC* are used. The distance of the avalanche front to the *start of runout area point* is determined using a user defined threshold of the flow variable. The front positions defined with this method for all the time steps are shown as black dots in the **tt-diagram**. The mean values of the flow variable are computed at cross profiles along the avalanche path for each time step and included in the **tt-diagram** as colored field. When computing the mean values, all the area where the flow variable is bigger than zero is taken into account. For this analysis, all available flow variables can be chosen, but the interpretation of the tt-diagram structures and the corresponding meaning of avalanche front may be different for flow thickness or flow velocity.

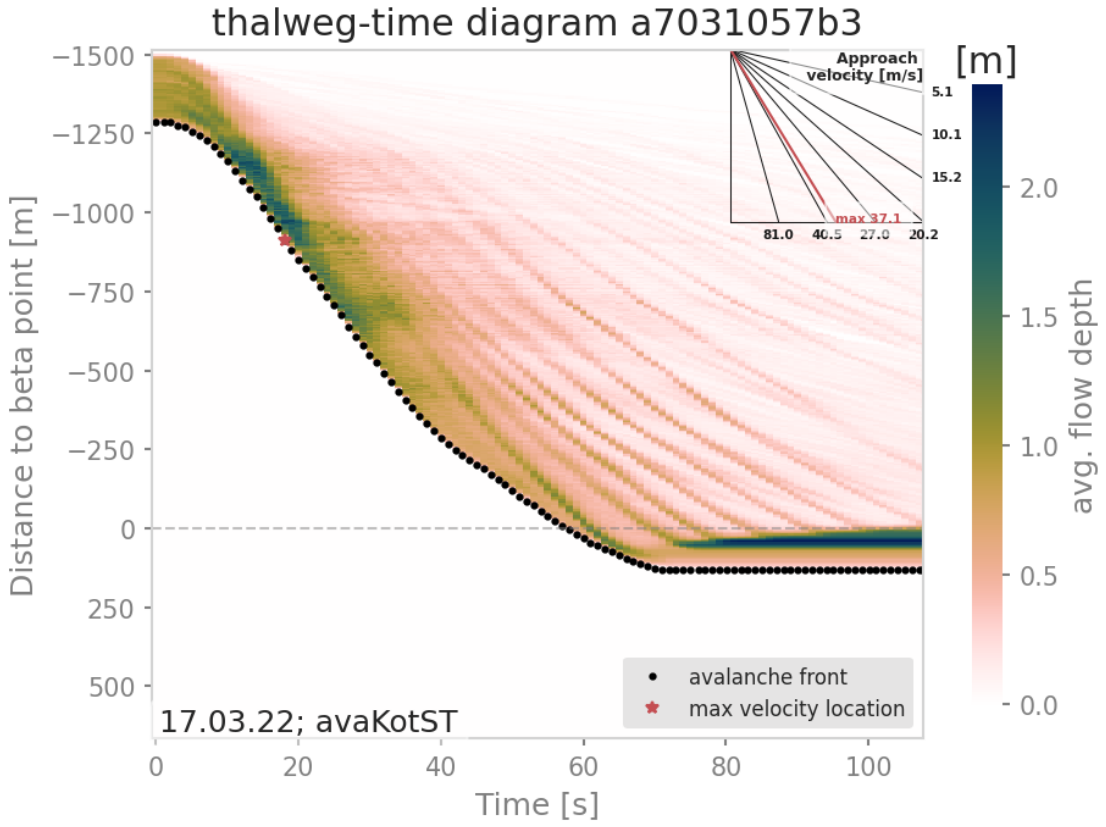


Fig. 5.28: Thalweg-time diagram example: The y-axis contains the distance from the beta point along the avalanche path (projected on the horizontal plane), e.g. the thalweg. Dots represent the avalanche front with the slope being the approach velocity. Red star marks the maximal approach velocity (this approach velocity is also projected on the horizontal plane).

5.4.1.4 Simulated Range-Time diagram

The radar's field of view is determined using its location, a point in the direction of the field of view and the horizontal (azimuth) aperture angle of the antenna. The elevation or vertical aperture angle is not yet included. The line-of-sight distance of every grid point in the simulation results to the radar location is computed. The simulation results which lie outside the radar's field of view are masked. The distance of the avalanche front with respect to the radar location is determined for a user defined threshold in the flow variable and the average values of the result field for each range gate along the radar's line of sight are computed. This data is plotted in a range-time diagram, where the black dots indicate the avalanche front, and the colored field indicates the mean values of the flow variable for the range gates at each time step.

5.4.2 Automated path generation

Computational modules like $\alpha\beta$ (*com2AB: Alpha Beta Model*) or analysis modules like the Thalweg-time diagram (*ana5Utils*) or Aimec (*ana3AIMEC: Aimec*) require an avalanche path and split point as input. This avalanche path and split point are usually created manually based on an expert opinion. The objective of this module is to automatically generate an avalanche path from a dense flow avalanche (DFA) simulation and placing a split point. The path is generated from the center of mass position of the dense material, so it is called the mass averaged path. It is extended towards the top of the release area and at the bottom. Therefore it covers the entire length of the avalanche with some buffer in the runout area. The split point is extracted from the parabola that is fitted on to the avalanche path profile.

5.4.2.1 Input

The automatic path generation needs dense flow simulation results as input. These can be flow mass and flow thickness or particles for multiple time steps. com1DFA provides these already in the correct way.

We provide `runComputeDFAPath`, in which two options exist:

1. DFA simulation results already exist: in this case, you want to provide these as inputs to the path generation function. The flag `runDFAModule` in `runComputeDFAPath` is set to `False`. You need to provide the avalanche directory in your `local_avaframeCfg.ini` file. This avalanche directory should already have `Outputs/com1DFA` with one or multiple simulation results. The simulation DEM is also required.
2. No DFA simulation results exist: use com1DFA to generate the simulation results before generating a path. Change the `runDFAModule` flag in `runComputeDFAPath` to True. The default configuration for com1DFA is read. `tSteps` are adjusted, `resType` and `simTypeList` are modified before running com1DFA.

5.4.2.2 Outputs

A mass averaged path is produced for each com1DFA simulation. The path is/are saved in `avalancheDir/Outputs/DFAPath`

5.4.2.3 To run automated path

- go to `AvaFrame/avaframe`
- copy `ana5Utils/DFAPathGenerationCfg.ini` to `ana5Utils/local_DFAPathGenerationCfg.ini` and edit (if not, default values are used)
- run:

```
python3 runScripts/runComputeDFAPath.py
```

5.4.2.4 Theory automated path

Mass average path

Any DFA simulation should be able to produce information about mass distribution for different time steps of the simulation (either flow thickness, mass, velocities… rasters or particles). This information is used to compute time dependent mass average quantities such as position (center of mass), velocity… For a flow quantity $\mathbf{a}(\mathbf{x}, t)$, the associated mass averaged quantity is defined by:

$$\bar{\mathbf{a}}(t) = \int_V \rho \mathbf{a}(\mathbf{x}, t)\, dV \approx \sum_k m_k \mathbf{a}_k(t)$$

where m_k respectively $\mathbf{a}_k(t)$ defines the mass respectively flow quantity of particle or raster cell k. Applying the mass averaging to (x, y, z) gives the mass average path profile.

Note: The mass average path profiles does not necessarily lie on the topography

It is also possible to compute the mass averaged velocity squared $\overline{\mathbf{u^2}}(t)$, kinetic energy $\overline{\frac{1}{2}m\mathbf{u^2}}(t)$ or travel distance s (which are used in the *Energy line test*).

The path is resampled at `nCellsResample` x cellsize and is extended towards the release area top to produce meaningful results when used in the com2AB module. Since results from the $\alpha\beta$ analysis depend on the path profile start, moving the starting point of the profile will shift the α upwards or downwards and affect the runout value.

Extending path towards the top (release)

There are two options available to extend the mass averaged path profile in the release area (`extTopOption` in the configuration file):

0. Extend the path up to the highest point in the release (highest particle or highest cell depending on which particles or rasters are available).
1. Extend the path towards the point that will lead to the longest runout. This point does not necessarily coincide with the highest point in the release area and corresponds to the point for which $(\Delta z - \Delta s \tan \alpha)$ is maximum. α corresponds to the angle of the runout line going from first to last point of the mass averaged line. Δz and Δs represent the vertical and horizontal distance between a point in the release and the first point of the mass averaged path profile.

Extending path towards the bottom (runout)

It is also necessary to extend the profile in the runout area. This is done by finding the direction of the path given by the few last points in the path in (x,y) (all points at a distance `nCellsMinExtend` x cellSize < distance < `nCellsMaxExtend` x cellSize)) and extending in this direction for a given percentage (`factBottomExt`) of the total length of the path s.

Split point generation

A parabolic curve is fitted to the avalanche path profile extracted from the DFA simulation (non-extended profile), where the first and last point of the parabolic profile match the avalanche path profile. To find the best fitting parabolic profile, an additional constraint is needed. Two options are available: the default one (`fitOption``= 0`) minimises the distance between the parabolic profile and the avalanche path profile. The second option (``fitOption``= 1) matches the end slope of the parabola to the profile. This parabolic fit determines the split point location. It is the first point for which the slope is lower than the ``slopeSplitPoint` angle. This point is then projected on the avalanche path profile.

Resampling

If the center of mass positions are derived in an equal time interval from the simulations, derived points will not be spaced equally due to variations in flow velocity. Especially in the release and runout area, lower velocites result in a denser spacing of extracted centers of mass, which can cause a crossing of grid lines that are drawn perpendicularly to the thalweg over the width of the domain. In order to reduce these overlaps, a the resampling function *in3Utils.geoTrans.prepareLine()* can be used, where the thalweg is generated based on a spline of degree k scipy splprep and a user defined approximate distance between points along the spline.

5.5 out1Peak: Peak plots

5.5.1 Plot all peak fields

out1Peak.outPlotAllPeak.plotAllPeakFields() generates one plot for each peak field in `Outputs/modName/peakFiles`, constrained to existing data including a buffer specified in `out3Plot/plotUtilsCfg.ini`. These peak fields represent the peak values of the simulation result parameters (*dynamic peak pressure*, *peak flow thickness*, *peak velocity*), and modName corresponds to the name of the computational module that has been used to perform the simualtions. Details on this function, as for example required inputs, can be found in *out1Peak.outPlotAllPeak.plotAllPeakFields()*.

5.5.2 Plot all fields

out1Peak.outPlotAllPeak.plotAllFields() generates one plot for each simulation result field provided in the specified input directory. This function is designed to work for result fields that follow a certain naming convention in order to provide full functionality: *releaseAreaName_simulationType_modelType_simulationIdentifier_resultType.asc*

One plot for each field is saved using the name of the input field. By default, the plot is constrained to where there is data, however this can be turned off by setting `constrainData=False` in the inputs. Details on this function, as for example required inputs, can be found in `out1Peak.outPlotAllPeak.plotAllField()`.

5.6 out3Plot: Plots

This module gathers various functions for creating visualisations of simulation results, analysis results, etc. Details on these functions can be found in *out3Plot*. General plot settings can be found in `out3Plot.plotUtils` and the respective configuration file `out3Plot/plotUtilsCfg.ini`.

In the following sections, some of these plotting functions are described in more detail.

5.6.1 plotUtils

`out3Plot.plotUtils` gathers the general settings used for creating plots as well as small utility functions.

5.6.1.1 makeColorMap

When plotting raster data, `out3Plot.plotUtils.makeColorMap()` is useful for deriving a suitable colormap. As inputs, this function requires:

```
makeColorMap(colormapDict, minimum level, maximum level, continuous flag)
```

By setting the continuous flag, there is the option to return a continuous or discrete colormap. Within this function, several default colormaps are available, as for example the ones used to visualize flow thickness, flow velocity or pressure. These can be initialised by:

```
from avaframe.out3Plot import plotUtils
plotUtils.cmapThickness
```

A colormap dictionary is loaded with information about the colormap, specific colors and levels corresponding to the default option of visualising pressure in com1DFA. If the flag *continuous* is set to *True*, this function returns a colormap object and a norm object fitted to the minimum and maximim values provided as input parameters. If the *continuous* flag is set to *False*, it will return a colormap object and a norm object representing the discrete levels defined in the dictionary. The maximum level provided in the inputs will be added as maximum level of the colorscale. See the two example plots below, where the predefined levels correspond to 1, 10, 25, and 50 kPa and the maximum level provided as input is 320 kPa.

In order to define customized discrete colorbars, there are different options. Firstly, provide a colorbar dictionary that follows the structure in the example dictionary:

```
from cmcrameri import cm as cmapCameri
colormapDict = {'cmap': cmapCameri.hawaii.reversed(),
                'colors': ["#B0F4FA", "#75C165", "#A96C00", "#8B0069"],
                'levels': [1.0, 10.0, 25.0, 50.0]}
```

It is also possible to provide key colors only. In this case the number of levels will match the number of colors and are equally distributed between the provided maximum and minimum value. If neither colors nor levels are provided in the dictionary, a default of six levels will be used. Another option is to just provide a colormap object instead of a dictionary. Here the *continuous* flag will be ignored and a colormap object as well as a norm object fitted to the maximum and minimum value will be returned.

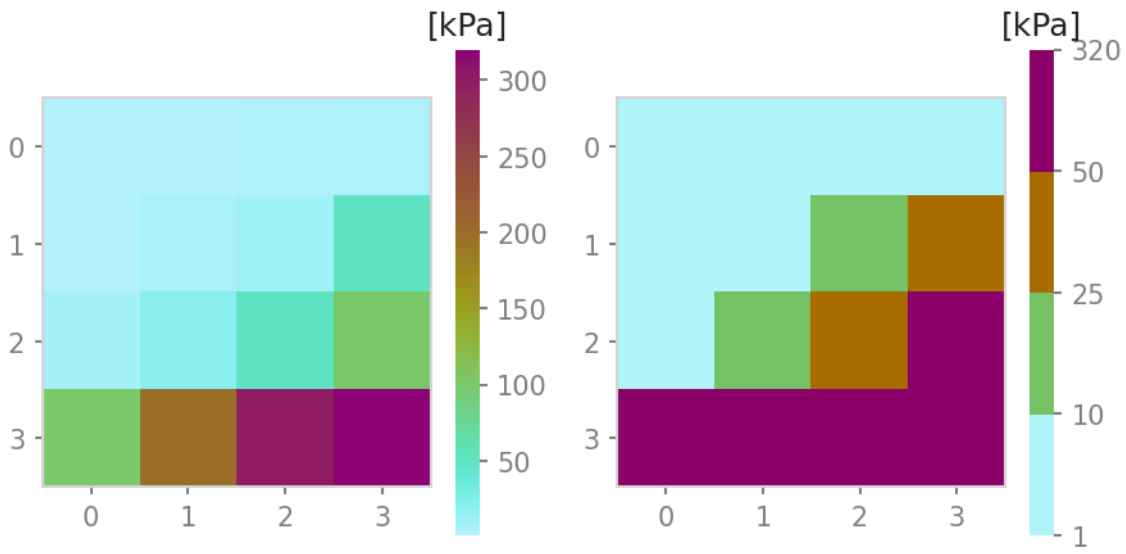


Fig. 5.29: left panel: continuous colormap. right panel: default discrete colormap used for pressure in com1DFA

5.6.2 outQuickPlot

out3Plot.outQuickPlot is used to generate plots of raster datasets, as for example the simulation results of *com1DFA*.

5.6.2.1 generatePlot

out3Plot.outQuickPlot.generatePlot() creates two plots, one plot with four panels, first dataset, second dataset, the absolute difference of the two datasets and the absolute difference capped to a smaller range of differences (ppr: +- 100kPa, pft: +-1m, pfv:+- 10ms-1). The difference plots also include an insert showing the histogram and the cumulative density function of the differences. The second plot shows a cross- and along profile cut of the two datasets. In addition to the plots, a dictionary is returned with information on the plot paths, as well as the statistical measures of the difference plots, such as mean, max and min difference. Details on the required inputs for this function can be found in *out3Plot.outQuickPlot.generatePlot()*.

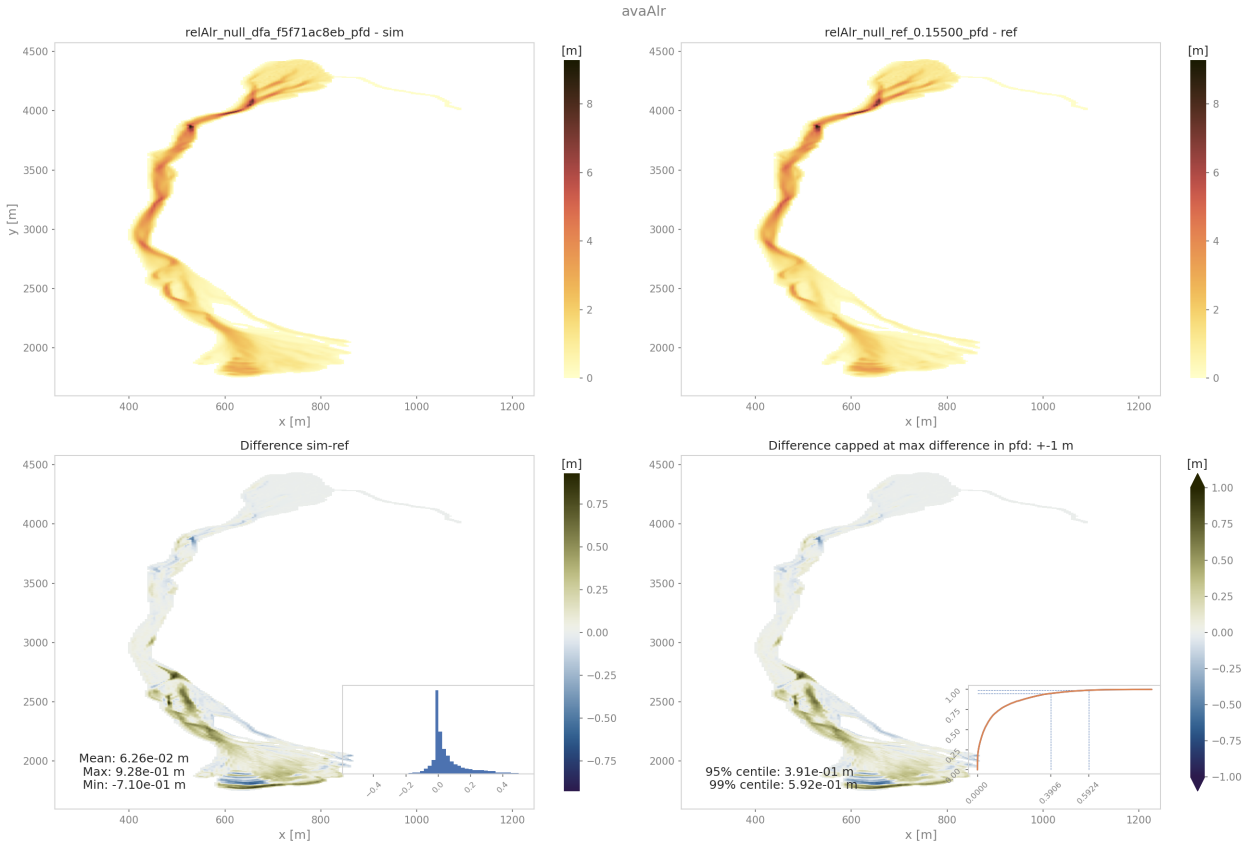


Fig. 5.30: Output plot from generatePlot on peak flow thickness results

5.6.2.2 quickPlotBench

out3Plot.outQuickPlot.quickPlotBench() calls *out3Plot.outQuickPlot.generatePlot()* to generate all comparison plots between the results of two simulations. This requires information on simulation names and paths to the simulation results and the desired result type. For further details have a look at *out3Plot.outQuickPlot.quickPlotBench()*.

5.6.2.3 quickPlotSimple

out3Plot.outQuickPlot.quickPlotSimple() is a bit more general, as it calls *out3Plot.outQuickPlot.generatePlot()* to generate the comparison plots between of two raster datasets of identical shape in a given input directory, without requiring further information. For further details have a look at *out3Plot.outQuickPlot.quickPlotSimple()*.

To run

An example on how to create the difference plots for two raster datasets of identical shape is provided in `runScript/runQuickPlotSimple`

- first go to `AvaFrame/avaframe`
- copy `avaframeCfg.ini` to `local_avaframeCfg.ini` and set your avalanche directory and the flag `showPlot`
- specifiy input directory, default is `data/NameOfAvalanche/Work/simplePlot`
- run:

```
python3 runScripts/runQuickPlotSimple.py
```

5.6.2.4 generateOnePlot

out3Plot.outQuickPlot.generateOnePlot() creates one plot of a single raster dataset. The first panel shows the dataset and the second panel shows a cross- or along profile of the dataset. The function returns a list with the file path of the generated plot. For further details have a look at *out3Plot.outQuickPlot.generateOnePlot()*.

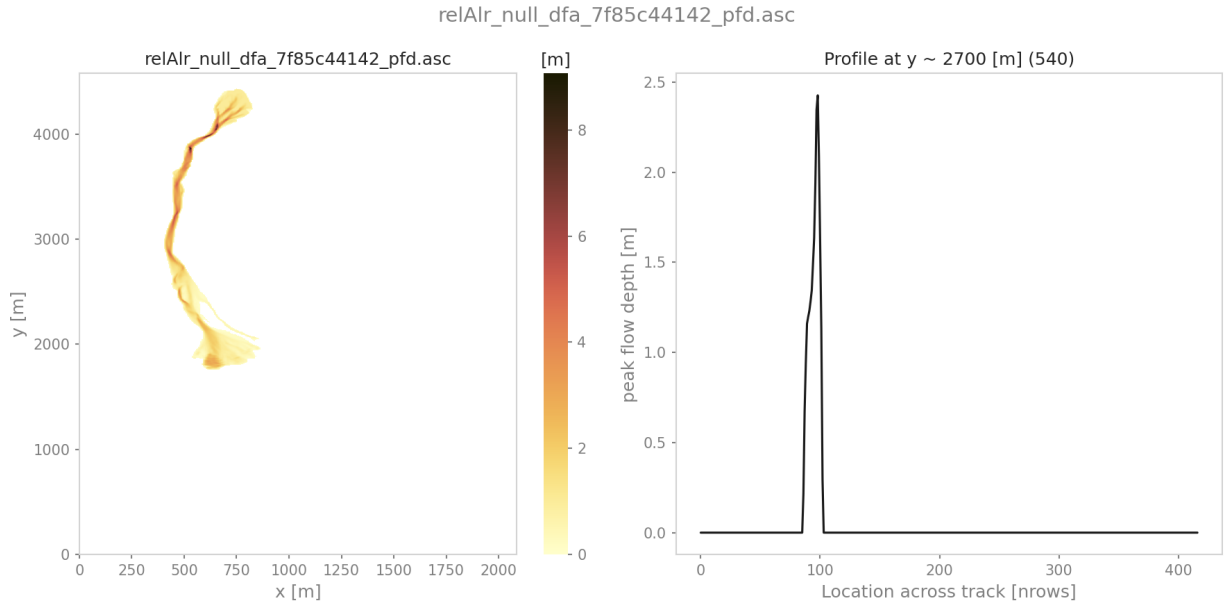


Fig. 5.31: Output plot from generatePlotOne on peak flow thickness results

5.6.2.5 quickPlotOne

out3Plot.outQuickPlot.quickPlotOne() calls *out3Plot.outQuickPlot.generateOnePlot()* to generate the plot corresponding to the input data. For information on the required inputs have a look at *out3Plot.outQuickPlot.quickPlotOne()*.

To run quickPlotOne

An example on how to create this plot from a given input directory or from the default one `data/NameOfAvalanche/Work/simplePlot`, is provided in `runScript/runQuickPlotOne`

- first go to `AvaFrame/avaframe`
- copy `avaframeCfg.ini` to `local_avaframeCfg.ini` and set your avalanche directory and the flag `showPlot`
- copy `out3Plot/outQuickPlotCfg.ini` to `out3Plot/local_outQuickPlotCfg.ini` and optionally specify input directory
- run:

```
python3 runScripts/runQuickPlotOne.py
```

5.6.3 in1DataPlots

out3Plot.in1DataPlots can be used to plot a sample and its characteristics derived with *in1Data.computeFromDistribution*, such as: cumulative distribution function (CDF), bar plot of sample values, probability density function (PDF) of the sample, comparison plot of empirical- and desired CDF and comparison of empirical- and desired PDF.

5.6.4 statsPlots

out3Plot.statsPlots can be used to create scatter plots using a peak dictionary where information on two result parameters of avalanche simulations is saved. This peak dictionary can be created using the function *ana4Stats.getStats.extractMaxValues()* of *ana4Stats.getStats*. This can be used to visualize results of avalanche simulations where a parameter variation has been used or for e.g. in the case of different release area scenarios. If a parameter variation was used to derive the simulation results, the plots indicate the parameter values in color. If the input data includes information about the 'scenario' that was used, for example different release scenarios, the plots use different colors for each scenario. There is also the option to add a kde (kernel density estimation) plot for each result parameter as marginal plots. An example on how these plotting functions are used and exemplary plots can be found in *getStats*

Additionally, a plotting function for visualising probability maps is provided by *out3Plot.statsPlots.plotProbMap()*, where probability maps can be plotted including contour lines. An example on how these plotting function is used and an exemplary plot can be found in moduleAna4Stats:probAna.

5.6.4.1 plotValuesScatter

out3Plot.statsPlots.plotValuesScatter() produces a scatter plot of result type 1 vs result type 2 with color indicating values of the varied parameter.

5.6.4.2 plotValuesScatterHist

out3Plot.statsPlots.plotValuesScatterHist() produces a scatter plot with marginal kde plots of result type 1 vs result type 2 with color indicating different scenarios (optional).

5.6.4.3 plotHistCDFDiff

out3Plot.statsPlots.plotHistCDFDiff() generates the histogram plot and CDF plot of a input dataset.

5.6.5 particle analysis plots

`out3Plot.particleAnalysisPlots` can be used to create plots of particle properties for a *com1DFA* simulation, where particles refer to two-dimensional numerical columns (see *Discretization*). The particle properties can be analyzed over time or transformed into a thalweg following coordinate system using *ana3AIMEC*. Additional functions to compute velocity envelopes, i.e. the min and max values of the particle properties over time but also along the thalweg are used (see `out3Plot.outParticleAnalysis.velocityEnvelope()` and `out3Plot.outParticleAnalysis.velocityEnvelopeThalweg()`). The provided run script `runScripts/runParticleAnalysis.py`, provides an example of calling *com1DFA* to perform an avalanche simulation and then perform the respective particle analysis including a coordinate transformation and producing the final plots, which are examplary shown here:

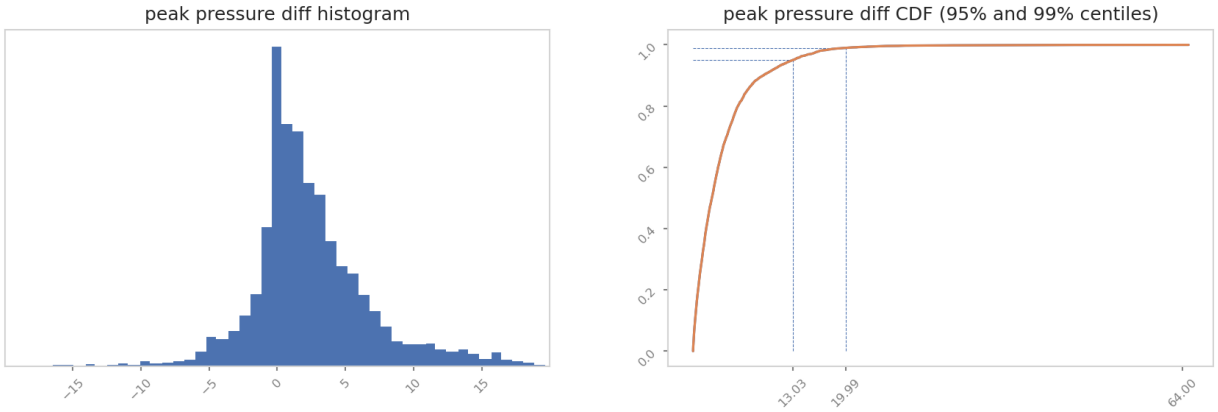


Fig. 5.32: Output plot from plotHistCDFDiff on peak pressure results from two simulations of avaAlr

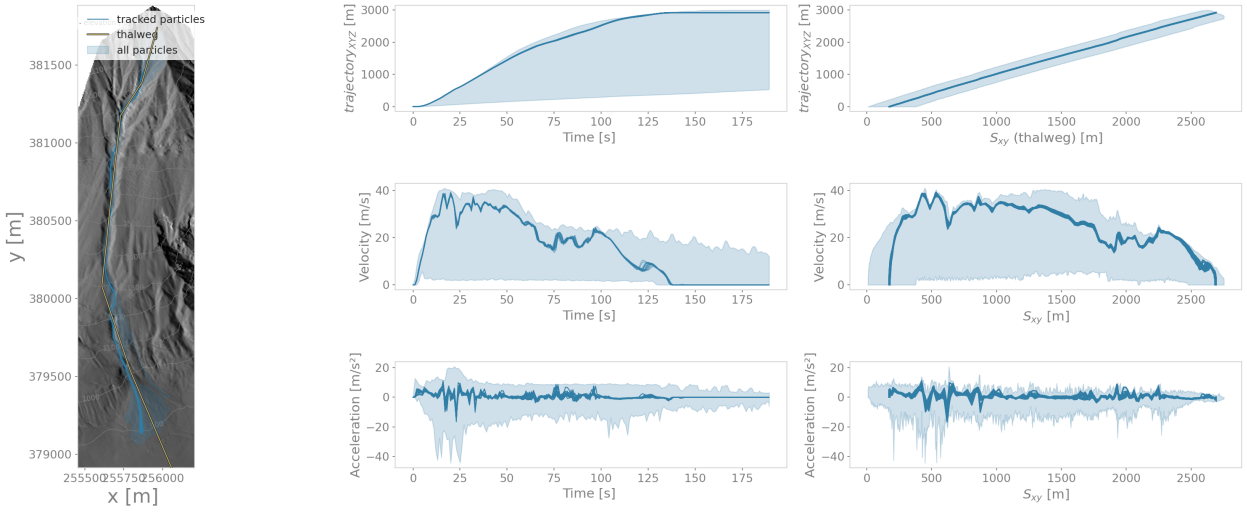


Fig. 5.33: Particle properties summary plot showing a map view of the affected area (by all particles) based on the peak flow velocity field, the tracked particles trajectories in dark blue with the superimposed thalweg line. The panels in the middle row show the evolution of particles' trajectory lengths, velocity and acceleration over time. In the rightmost panels, the particle data has been transformed into a thalweg following coordinate system and particle properties are shown along the thalweg coordinate S_{XY}. The light blue area extends from the min value found for all particles to the max value and the blue solid lines indicate the values of the tracked particles.

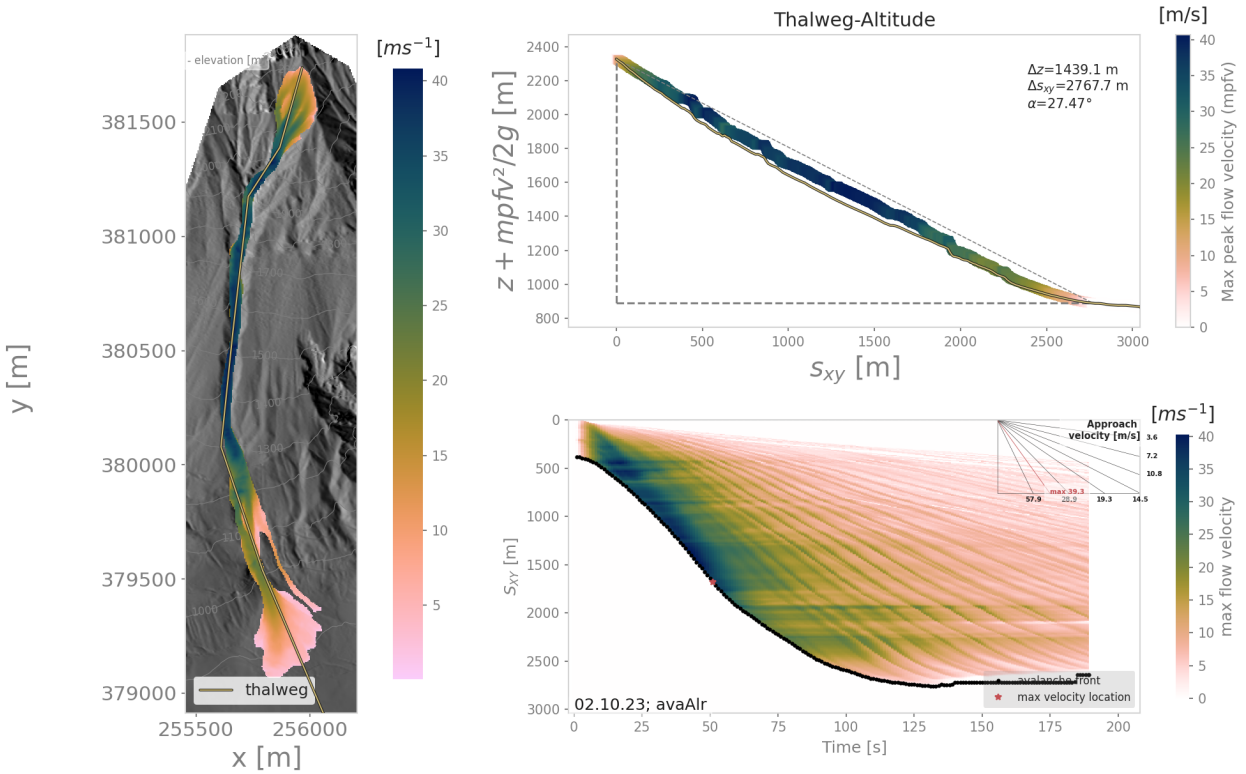


Fig. 5.34: The left panel shows a map view of the peak flow velocity field for the avalanche simulation with superimposed thalweg line. The top right plot shows the thalweg profile and the 'velocity altitude', i.e. the thalweg elevation plus the peak flow velocity cross max values along the thalweg to the power of two divided by two times the gravity acceleration, these values are then colored using the peak flow velocity cross max values. In the legend, the runout length ΔS_{xy}, altitude difference Δz and corresponding runout anlge α, measured using the peak flow velocity field and a threhsold of 1 ms^{-1} (default setting) are provided. The lower right panel shows the thalweg-time diagram for the respective simulation.

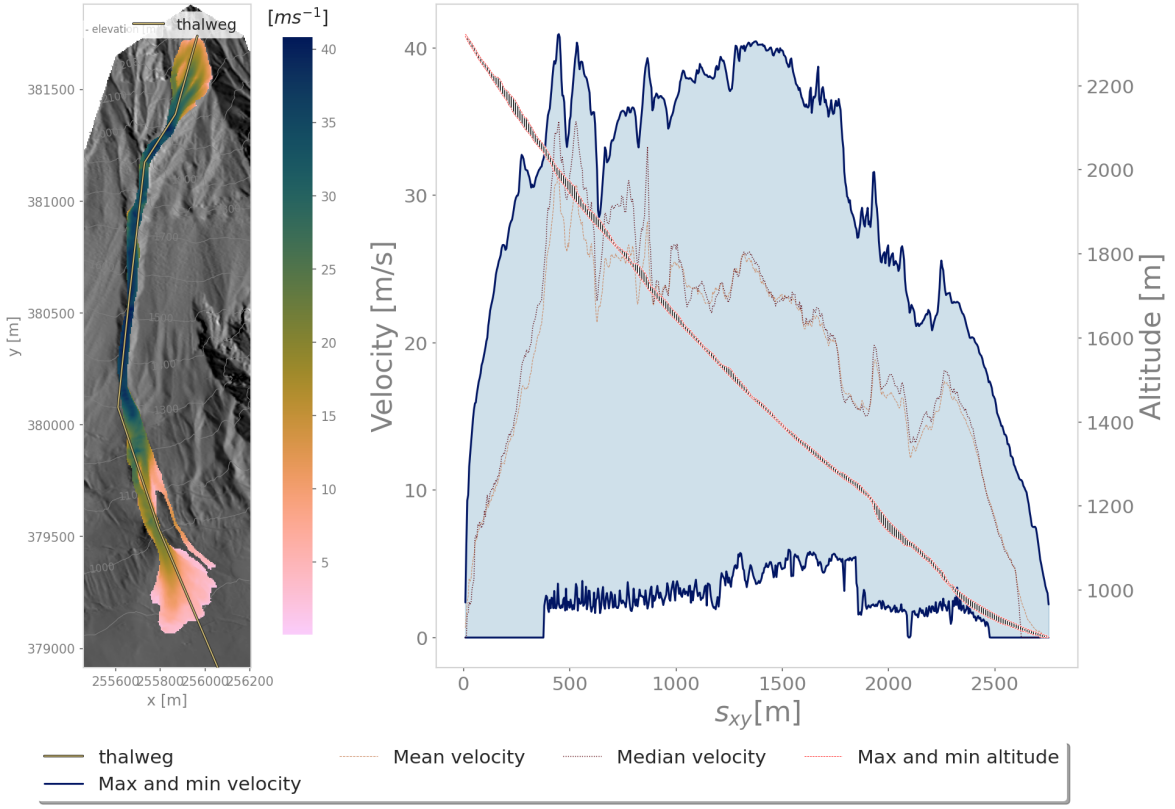


Fig. 5.35: The left panel shows a map view of the peak flow velocity field for the avalanche simulation with superimposed thalweg line. The right panel shows the velocity envelope for all particles, the mean and median particle velocity and the spread of altitudes covered by the particle positions along the thalweg coordinate.

Note: In order to create the presented plots, in addtion to the particle data also result fields of peak flow velocity are required.

5.6.5.1 To run

- first go to `AvaFrame/avaframe`
- copy `avaframeCfg.ini` to `local_avaframeCfg.ini` and set your desired avalanche directory name
- create an avalanche directory with required input files - for this task you can use *Initialize Project*
- in `AvaFrame/avaframe/out3Plot`, copy `outParticleAnalysisCfg.ini` to `local_outParticleAnalysisCfg.ini` and set your desired configuration for the analysis, the avalanche simulation run in com1DFA_override, the aimec analysis in ana3AIMEC_override and the range time diagram in distanceTimeAnalysis_override section
- run:

```
python3 runScripts/runParticleAnalysis.py
```

5.7 log2Reports: Create Reports

5.7.1 Generate Report

log2Report.generateReport creates a markdown style report corresponding to the data provided in the python input dictionary. The report is structured in blocks according to the types provided in the dictionary. Currently, the report generation supports the following types:

- title - creates a title header
- avaName - includes a line with the avalanche directory
- simName - includes a line with Simulation name in the header
- time - includes a line with the date
- list - creates a table with two columns where the keys and values are listed and the dictionary name is set as title
- columns - creates a table with two rows and one column for each key and value pair and the dictionary name is set as title
- image - creates a block with the dictionary name as title and prints the key as heading with the prefix *Figure* and plots the image provided as value
- text - creates a text block with the dictionary name as title and prints the key as header with the prefix *Topic* and prints the text provided as value

For the input dictionary, a certain structure is required as illustrated for the following example:

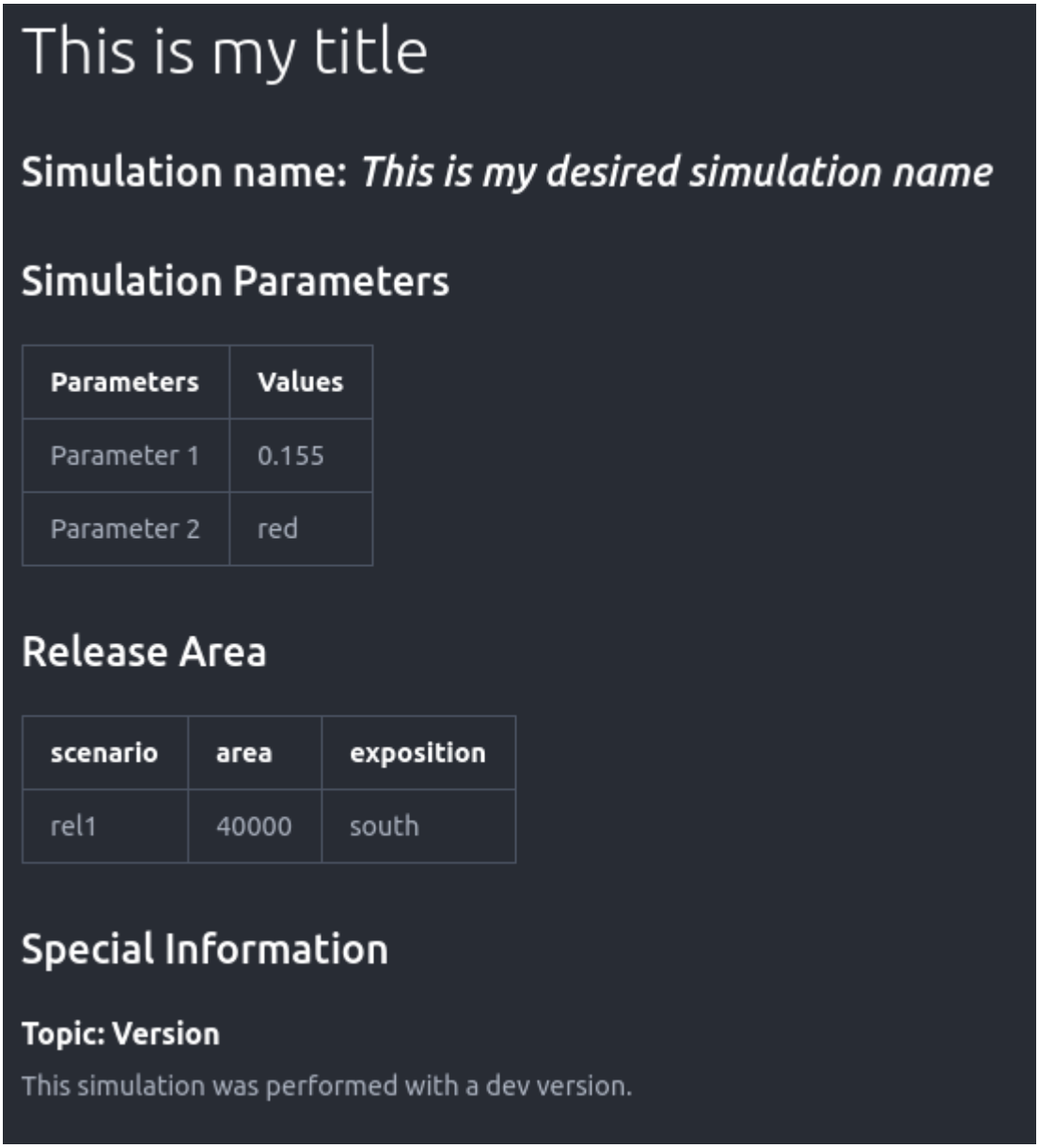
```
reportDict = {'title Block' : {'type' : 'title', 'title' : 'This is my title'},
              'Simulation Name' : {'type' : 'simName', 'name' : 'This is my desired␣
↪simulation name'},
              'Simulation Parameters' : {'type' : 'list', 'Parameter 1' : 0.155,
↪'Parameter 2' : 'red'},
              'Release Area' : {'type' : 'columns', 'scenario' : 'rel1', 'area' : '40000
```

(continues on next page)

(continued from previous page)

```
↪', 'exposition' : 'south'},
             'Special Information' : {'type' : 'text', 'Version' : 'This simulation was␣
↪performed with a dev version.'}
             }
```

where the keys and values do not have to follow a predefined form, only the key *type* has to be provided. This example dictionary results in a report of the form:



This is my title

Simulation name: *This is my desired simulation name*

Simulation Parameters

Parameters	Values
Parameter 1	0.155
Parameter 2	red

Release Area

scenario	area	exposition
rel1	40000	south

Special Information

Topic: Version

This simulation was performed with a dev version.

If a separate plotDictionary shall be included, in addition the key *simName* is required so that it can be appended to the correct simulation dictionary.

The default setting is to produce one report file, *fullSimulationReport.md*, however if one wants to receive on report for each *simName*, set the flag *reportOneFile = False* in `avaframeCfg.ini` or preferabley your local copy

local_com1DFACfg.ini. For further details have a look at *log2Report.generateReport* .

5.7.2 Generate Compare Report

log2Report.generateCompareReport creates a markdown style report where the simulation results are compared to the benchmark results. The report is structured as follows:

- name of avalanche
- simulation name
- text block with info on test
- table listing simulation parameter name, value (reference) and value (simulation) -> if value of simulation run differs from reference value, it is highlighted in red
- table listing aimec analysis measure, value (reference) and value (simulation) -> if value of simulation run differs from reference value, it is highlighted in red
- block with plots including a header with plot title -> if differences exceed threshold a Warning is printed for each plot

For this purpose, python dictionaries are used to read the required input data. In order to produce the structure described above, the dictionaries require a certain structure too, for example:

```
dictionary = {
              '*simName*': 'name of simulation',
              '*Simulation Parameters*': {
              '*type*': '*list*',
              'name of parameter': 'value of parameter'
              ...
                                        },
              '*Test Info*': {
              '*type*': '*text*',
              'Title of text box': 'text, text',
              ...
                             },
              '*Simulation Results*': {
              'desired plot title': '*file path*',
              ...
                                    }
                             },
```

where the required **keys** are indicated by the asterisk sign. The settings can be found in the respective configuration file log2Report/genereateCompareReportCfg.ini. For further details visit *log2Report.generateCompareReport*.

An example of creating a comparison report can be found in runStandardTestsCom1DFA.py. There, all the standard tests are run sequentially and a report is created where the com1DFA simulation results and the benchmark results are compared.

CHAPTER

SIX

THEORY

6.1 com1DFA DFA-Kernel theory

Warning: This theory has not been fully reviewed yet. Read its content with a critical mind.

6.1.1 Governing Equations for the Dense Flow Avalanche

The governing equations of the dense flow avalanche are derived from the incompressible mass and momentum balance on a Lagrange control volume ([Zwi00, ZKS03]).

6.1.1.1 Mass balance:

$$\frac{d}{dt}\int\limits_{V(t)} \rho_0 \,\mathrm{d}V = \rho_0 \frac{dV(t)}{dt} = \oint\limits_{\partial V(t)} q^{\text{ent}} \,\mathrm{d}A \tag{6.1}$$

Where q^{ent} represents the snow entrainment rate.

6.1.1.2 Momentum balance:

$$\rho_0 \frac{d}{dt}\int\limits_{V(t)} u_i \,\mathrm{d}V = \oint\limits_{\partial V(t)} \sigma_{ij}^{\text{tot}} n_j \,\mathrm{d}A + \rho_0 \int\limits_{V(t)} g_i \,\mathrm{d}V, \quad i = (1,2,3) \tag{6.2}$$

We introduce the volume average of a quantity $P(\mathbf{x}, t)$:

$$\overline{P}(\mathbf{x}, t) = \frac{1}{V(t)} \int\limits_{V(t)} P(\mathbf{x}, t) \,\mathrm{d}V$$

and split the area integral into :

$$\oint\limits_{\partial V(t)} \sigma_{ij}^{\text{tot}} n_j \,\mathrm{d}A = \oint\limits_{\partial V(t)} \sigma_{ij} n_j \,\mathrm{d}A + F_i^{\text{ent}} + F_i^{\text{res}}, \quad i = (1,2,3)$$

F_i^{ent} represents the force required to break the entrained snow from the ground and to compress it (since the dense-flow bulk density is usually larger than the density of the entrained snow, i.e. $\rho_{\text{ent}} < \rho$) and F_i^{res} represents the resistance force due to obstacles (for example trees). This leads to in Eq.6.2:

$$\rho_0 \frac{dV(t)\overline{u}_i}{dt} = \rho_0 V \frac{d\overline{u}_i}{dt} + \rho_0 \overline{u}_i \frac{dV}{dt} = \oint\limits_{\partial V(t)} \sigma_{ij} n_j \,\mathrm{d}A + \rho_0 V g_i + F_i^{\text{ent}} + F_i^{\text{res}}, \quad i = (1,2,3)$$

Using the mass balance equation Eq.6.1, we get:

$$\rho_0 V \frac{d\overline{u}_i}{dt} = \oint\limits_{\partial V(t)} \sigma_{ij} n_j \, \mathrm{d}A + \rho_0 V g_i + F_i^{\text{ent}} + F_i^{\text{res}} - \overline{u}_i \oint\limits_{\partial V(t)} q^{\text{ent}} \, \mathrm{d}A, \quad i = (1,2,3) \tag{6.3}$$

6.1.1.3 Boundary conditions:

The free surface is defined by :

$$F_s(\mathbf{x}, t) = z - s(x, y, t) = 0$$

The bottom surface is defined by :

$$F_b(\mathbf{x}) = z - b(x, y) = 0$$

The boundary conditions at the free surface and bottom of the flow read:

$$\begin{cases} \dfrac{dF_s}{dt} = \dfrac{\partial F_s}{\partial t} + u_i \dfrac{\partial F_s}{\partial x_i} = 0 & \text{at } F_s(\mathbf{x}, t) = 0 \quad \text{Kinematic BC (Material boundary)} \\ \sigma_{ij} n_j = 0 & \text{at } F_s(\mathbf{x}, t) = 0 \quad \text{Dynamic BC (Traction free surface)} \\ u_i n_i = 0 & \text{at } F_b(\mathbf{x}, t) = 0 \quad \text{Kinematic BC (No detachment)} \\ \tau_i^{(b)} = f(\sigma^{(b)}, \overline{u}, \overline{h}, \rho_0, t, \mathbf{x}) & \text{at } F_b(\mathbf{x}, t) = 0 \quad \text{Dynamic BC (Chosen friction law)} \end{cases} \tag{6.4}$$

$\sigma_i^{(b)} = (\sigma_{kl} n_l n_k) n_i$ represents the normal stress at the bottom and $\tau_i^{(b)} = \sigma_{ij} n_j - \sigma_i^{(b)}$ represents the shear stress at the bottom surface. f describes the chosen friction model and are described in *Friction Model*. The normals at the free surface ($n_i^{(s)}$) and bottom surface ($n_i^{(b)}$) are:

$$n_i^{(s,b)} = \frac{\partial F_{s,b}}{\partial x_i} \left(\frac{\partial F_{s,b}}{\partial x_j} \frac{\partial F_{s,b}}{\partial x_j} \right)^{-1/2}$$

6.1.1.4 Choice of the coordinate system:

The previous equations will be developed in the orthonormal coordinate system $(B, \mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3})$, further referenced as Natural Coordinate System (NCS). In this NCS, $\mathbf{v_1}$ is aligned with the velocity vector at the bottom and $\mathbf{v_3}$ with the normal to the slope, i.e.:

$$\mathbf{v_1} = \frac{\mathbf{u}}{\|\mathbf{u}\|}, \quad \mathbf{v_2} = \mathbf{v_3} \wedge \mathbf{v_1}, \quad \mathbf{v_3} = \mathbf{n^{(b)}}$$

The origin B of the NCS is attached to the slope. This choice leads to:

$$n_i^{(b)} = \delta_{i3}, \quad \left. \frac{\partial b}{\partial x_i} \right|_{\mathbf{0}} = 0 \quad \text{for} \quad i = (1,2), \quad \text{and} \quad u_2^{(b)} = u_3^{(b)} = 0$$

6.1.1.5 Thickness averaged equations:

In this NCS and considering a prism-like Control volume, the volume content $V(t) = A_b(t)\overline{h}$ is obtained by multiplication of the basal area of the prism, A_b, with the averaged value of the flow thickness,

$$\overline{h} = \frac{1}{A_b(t)} \int\limits_{A_b(t)} [s(\mathbf{x}) - b(\mathbf{x})] \, \mathrm{d}A = \frac{1}{A_b(t)} \int\limits_{A_b(t)} h(\mathbf{x}) \, \mathrm{d}A, \qquad \overline{u}_i = \frac{1}{V(t)} \int\limits_{V(t)} u_i(\mathbf{x}) \, \mathrm{d}V \tag{6.5}$$

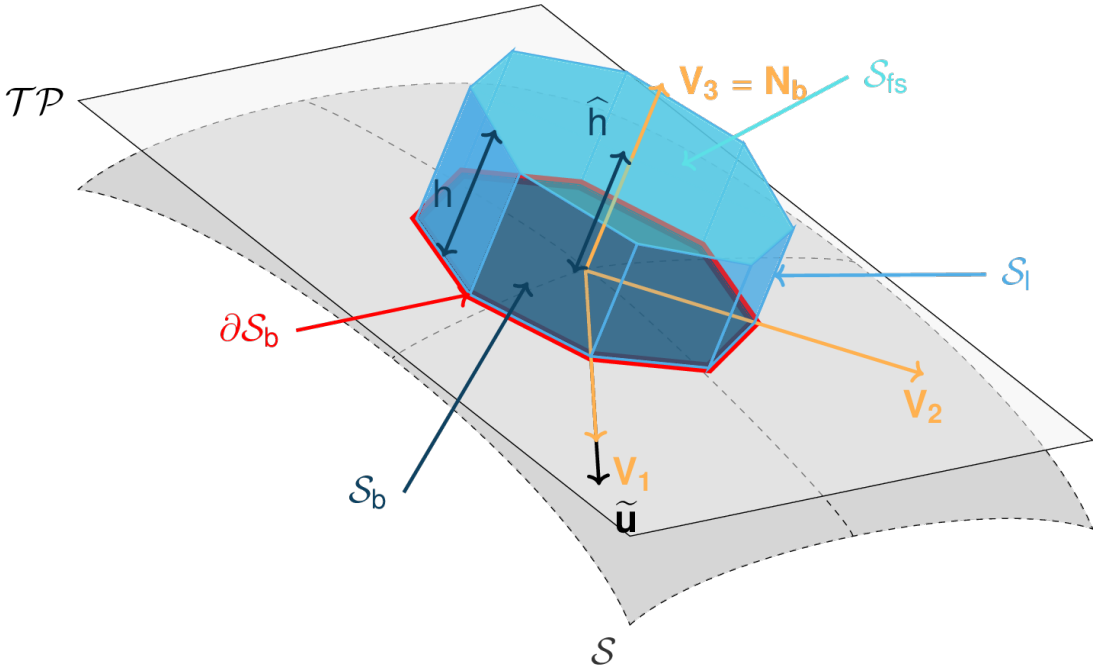


Fig. 6.1: Small Lagrangian prism-like Control volume

Entrainment:

The snow entrainment is either due to plowing at the front of the avalanche or to erosion at the bottom. The entrainment rate at the front $\dot{q}^{\mathrm{plo}}$ can be expressed as a function of the properties of the entrained snow (density ρ_{ent} and snow thickness h_{ent}), the velocity of the avalanche at the front $\overline{\mathbf{u}}$ and length w_f of the front (measured perpendicularly to the flow velocity $\overline{\mathbf{u}}$). It obviously only happens on the front of the avalanche:

$$\oint\limits_{\partial V(t)} \dot{q}^{\mathrm{plo}} \, \mathrm{d}A = \int\limits_{l_{\mathrm{front}}} \int_b^s \dot{q}^{\mathrm{plo}} \, \mathrm{d}l \, \mathrm{d}z = \rho_{\mathrm{ent}} \, w_f \, h_{\mathrm{ent}} \, \|\overline{\mathbf{u}}\| \tag{6.6}$$

The entrainment rate at the bottom $\dot{q}^{\mathrm{ero}}$ can be expressed as a function of the bottom area A_b of the control volume, the velocity of the avalanche $\overline{\mathbf{u}}$, the bottom shear stress $\tau^{(b)}$ and the specific erosion energy e_b:

$$\oint\limits_{\partial V(t)} \dot{q}^{\mathrm{ero}} \, \mathrm{d}A = \int\limits_{A_b} \dot{q}^{\mathrm{ero}} \, \mathrm{d}A = A_b \, \frac{\tau^{(b)}}{e_b} \, \|\overline{\mathbf{u}}\| \tag{6.7}$$

This leads in the mass balance Eq.6.1 to :

$$\frac{\mathrm{d}V(t)}{\mathrm{d}t} = \frac{\mathrm{d}(A_b\overline{h})}{\mathrm{d}t} = \frac{\rho_{\mathrm{ent}}}{\rho_0} \, w_f \, h_{\mathrm{ent}} \, \|\overline{\mathbf{u}}\| + \frac{A_b}{\rho_0} \, \frac{\tau^{(b)}}{e_b} \, \|\overline{\mathbf{u}}\| \tag{6.8}$$

The force F_i^{ent} required to break the entrained snow from the ground and to compress it is expressed as a function of the required breaking energy per fracture surface unit e_s ($J.m^{-2}$), the deformation energy per entrained mass element e_d ($J.kg^{-1}$) and the entrained snow thickness ([FFGS13, SFF+08, Sam07]):

$$F_i^{\text{ent}} = -w_f \, (e_s + q^{\text{ent}} \, e_d), \tag{6.9}$$

where q^{ent} refers to the entrainable mass per surface area ($kg.m^{-2}$) defined by $q^{\text{ent}} := \rho^{\text{ent}} h^{\text{ent}}$ which depending on whether entrainment is due to ploughing or erision, is derived using the integral of $\dot{q}^{\text{plo}}$, or respectively $\dot{q}^{\text{ero}}$, over time.

Resistance:

The force F_i^{res} due to obstacles is expressed as a function of the characteristic diameter $\overline{d}$ and height h_{res} of the obstacles, the spacing s_{res} between the obstacles and an empirical coefficient c_w (see Fig. 6.2). The effective height h^{eff} is defined as $\min(\overline{h}, h_{res})$:

$$F_i^{\text{res}} = -(\frac{1}{2} \, \overline{d} \, c_w / s_{\text{res}}^2) \, \rho_0 \, A \, h^{\text{eff}} \, \overline{u}^2 \, \frac{\overline{u}_i}{||\overline{u}||}$$

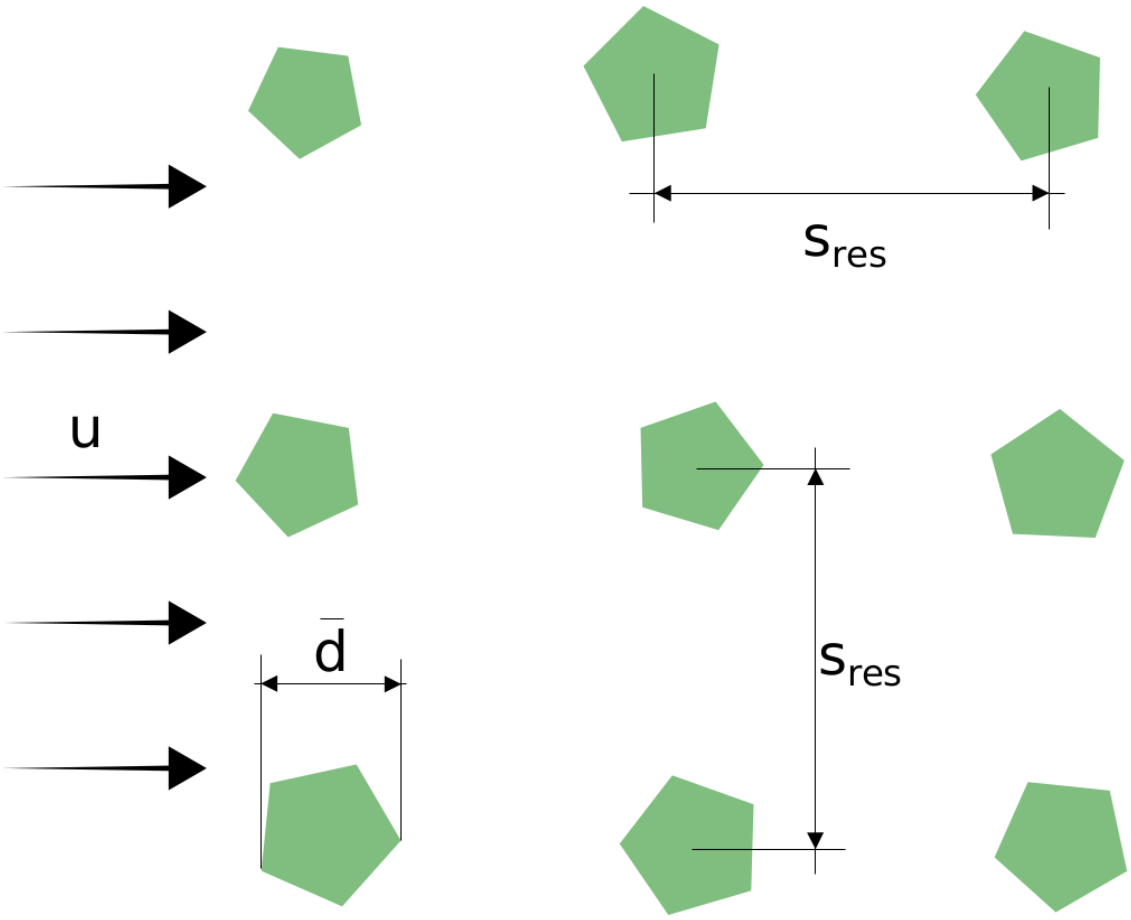


Fig. 6.2: Resistance force due to obstacles (from [FK13])

Surface integral forces:

The surface integral is split in three terms, an integral over A_b the bottom $x_3 = b(x_1, x_2)$, A_s the top $x_3 = s(x_1, x_2, t)$ and A_h the lateral surface. Introducing the boundary conditions Eq.6.4 leads to:

$$\begin{aligned}\oint_{\partial V(t)} \sigma_{ij} n_j \,\mathrm{d}A &= \int_{A_b} \underbrace{\sigma_{ij}\, n_j^{(b)}}_{-\sigma_{i3}} \,\mathrm{d}A + \int_{A_s} \underbrace{\sigma_{ij}\, n_j^{(s)}}_{0} \,\mathrm{d}A + \int_{A_h} \sigma_{ij}\, n_j \,\mathrm{d}A \\ &= -A_b \overline{\sigma}_{i3}^{(b)} + \oint_{\partial A_b} \left(\int_b^s \sigma_{ij}\, n_j \,\mathrm{d}x_3 \right) \mathrm{d}l\end{aligned}$$

Which simplifies the momentum balance Eq.6.3 to:

$$\begin{aligned}\rho_0 V \frac{d\overline{u}_i}{dt} = \oint_{\partial A_b} \left(\int_b^s \sigma_{ij}\, n_j \,\mathrm{d}x_3 \right) \mathrm{d}l - A_b \overline{\sigma}_{i3}^{(b)} + \rho_0 V g_i + F_i^{\text{ent}} + F_i^{\text{res}} - \overline{u}_i \oint_{\partial V(t)} q^{\text{ent}} \,\mathrm{d}A, \\ i = (1, 2, 3)\end{aligned} \tag{6.10}$$

The momentum balance in direction x_3 (normal to the slope) is used to obtain a relation for the vertical distribution of the stress tensor ([Sam07]). Due to the choice of coordinate system and because of the kinematic boundary condition at the bottom, the left side of Eq.6.10 can be expressed as a function of the velocity $\overline{u}_1$ in direction x_1 and the curvature of the terrain in this same direction $\frac{\partial^2 b}{\partial x_1^2}$ ([Zwi00]):

$$\rho\, A_b\, \overline{h}\, \frac{\mathrm{d}\overline{u}_3}{\mathrm{d}t} = \rho\, A_b\, \overline{h}\, \frac{\partial^2 b}{\partial x_1^2}\, \overline{u}_1^2,$$

rearranging the terms in the momentum equation leads to:

$$\overline{\sigma}_{33}(x_3) = \rho_0\, (s - x_3) \left(g_3 - \frac{\partial^2 b}{\partial x_1^2}\, \overline{u}_1^2 \right) + \frac{1}{A_b} \oint_{\partial A_b} \left(\int_{x_3}^s \sigma_{3j}\, n_j \,\mathrm{d}x_3 \right) \mathrm{d}l \tag{6.11}$$

6.1.1.6 Non-dimensional Equations

The previous equations Eq.6.10 and Eq.6.11 can be further simplified by introducing a scaling based on the characteristic values of the physical quantities describing the avalanche. The characteristic length L, the thickness H, the acceleration due to gravity g and the characteristic radius of curvature of the terrain R are the chosen quantities. From those values, it is possible to form two non dimensional parameters that describe the flow:

- Aspect ratio: $\varepsilon = H/L$
- Curvature: $\lambda = L/R$

The different properties involved are then expressed in terms of characteristic quantities L, H, g, ρ_0 and R (see Fig. 6.3):

$$\begin{aligned} x_i &= L\, x_i^* \\ (dx_3, h, \overline{h}) &= H\, (dx_3^*, h^*, \overline{h}^*) \\ A_b &= L^2\, A_b^* \\ t &= \sqrt{L/\mathrm{g}}\, t^* \\ \overline{u_i} &= \sqrt{\mathrm{g} L}\, \overline{u_i}^* \\ \mathrm{g}_i &= \mathrm{g}\, \mathrm{g}_i^* \\ \frac{\partial^2 b}{\partial {x_1}^2} &= \frac{1}{R}\, \frac{\partial^2 b^*}{\partial {x_1}^{*2}} \end{aligned}$$

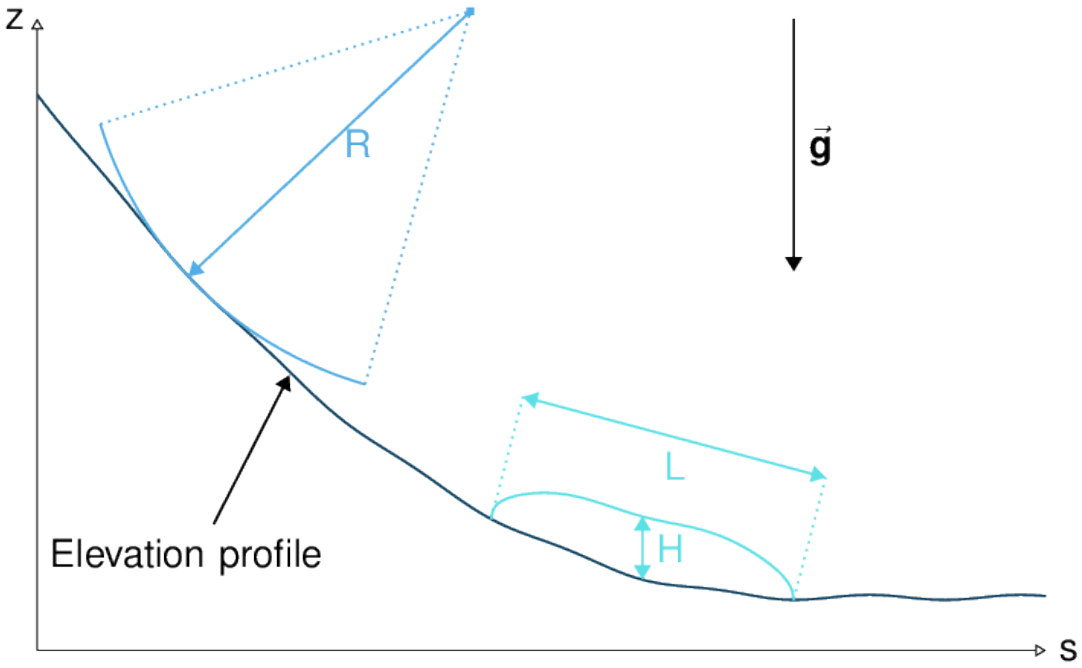


Fig. 6.3: Characteristic size of the avalanche along its path (from [Zwi00], modified)

The normal part of the stress tensor is directly related to the hydro-static pressure:

$$\sigma_{ii} = \rho_0 \, \mathrm{g} \, H \, \sigma^*_{ii}$$

The dimensionless properties are indicated by a superscripted asterisk. Introducing those properties in Eq.6.11, leads to :

$$\overline{\sigma^*}_{33} = \left(g^*_3 - \lambda \frac{\partial^2 b^*}{\partial x_1^{*2}} \, \overline{u}_1^{*2} \right) (s^* - x^*_3) + \underbrace{\varepsilon \oint_{\partial A^*_b} \left(\int_{x^*_3}^{s^*} \sigma^*_{31} \, \mathrm{d}x^*_3 \right) \mathrm{d}l^*}_{O(\varepsilon)} . \tag{6.12}$$

The height, H of dense flow avalanches is assumed to be small compared to its length, L. Meaning that the equations are examined in the limit $\varepsilon \ll 1$. It is then possible to neglect the last term in Eq.6.12 which leads to (after reinserting the dimensions):

$$\overline{\sigma}_{33}(x_3) = \rho_0 \underbrace{\left(g_3 - \overline{u_1}^2 \frac{\partial^2 b}{\partial x_1^2} \right)}_{g_{\text{eff}}} \left[\overline{h} - x_3 \right] \tag{6.13}$$

And at the bottom of the avalanche, with $x_3 = 0$, the normal stress can be expressed as:

$$\overline{\sigma}^{(b)}_{33} = \rho_0 \left(g_3 - \overline{u_1}^2 \frac{\partial^2 b}{\partial x_1^2} \right) \overline{h} \tag{6.14}$$

Calculating the surface integral in equation Eq.6.10 requires to express the other components of the stress tensor. Here again a magnitude consideration between the shear stresses $\sigma_{12} = \sigma_{21}$ and σ_{13}. The shear stresses are based on a generalized Newtonian law of materials, which controls the influence of normal stress and the rate of deformation through the viscosity.

$$\tau_{ij} = \eta \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right), \; i \neq j$$

Because ∂x_1 and ∂x_2 are of the order of L, whereas ∂x_3 is of the order of H, it follows that:

$$O\left(\frac{\sigma_{12}}{\sigma_{13}}\right) = \frac{H}{L} = \varepsilon \ll 1$$

and thus $\sigma_{12} = \sigma_{21}$ is negligible compared to σ_{13}. σ_{13} is expressed using the bottom friction law $\tau_i^{(b)} = f(\sigma^{(b)}, \overline{u}, \overline{h}, \rho_0, t, \mathbf{x})$ introduced in Eq.6.4.

In addition, a relation linking the horizontal normal stresses, σ_{ii}, $i = (1, 2)$, to the vertical pressure distribution given by Eq.6.14 is introduced. In complete analogy to the arguments used by Savage and Hutter ([SH89]) the horizontal normal stresses are given as:

$$\sigma_{ii} = K_{(i)}\, \sigma_{33}$$

Where $K_{(i)}$ are the earth pressure coefficients (cf. [Sal04, ZKS03]):

$$\sigma_{11} = K_{x\ akt/pass}\, \sigma_{33}$$
$$\sigma_{22} = K_{y\ akt/pass}^{(x\ akt/pass)}\, \sigma_{33}$$

With the above specifications, the integral of the stresses over the flow height is simplified in equation Eq.6.10 to:

$$\int_b^s \sigma_{ij}\, \mathrm{d}x_3 = \int_b^s K_{(i)}\, \sigma_{33}\, \mathrm{d}x_3 = K_{(i)}\, \frac{\overline{h}\, \sigma^{(b)}}{2}$$

and the momentum balance can be written:

$$\begin{aligned}\rho_0\, A\, \overline{h}\, \frac{\mathrm{d}\overline{u}_i}{\mathrm{d}t} =& \rho_0\, A\, \overline{h}\, g_i + \underbrace{K_{(i)} \oint_{\partial A} \left(\frac{\overline{h}\, \sigma^{(b)}}{2}\right) n_i\, \mathrm{d}l}_{F_i^{\text{lat}}} \underbrace{-\delta_{i1}\, A\, \tau^{(b)}}_{F_i^{\text{bot}}} \underbrace{-\rho_0\, A\, h_{\text{eff}}\, C_{\text{res}}\, \overline{\mathbf{u}}^2\, \frac{\overline{u_i}}{\|\overline{\mathbf{u}}\|}}_{F_i^{\text{res}}} \\ & - \overline{u_i}\, \rho_0\, \frac{\mathrm{d}\left(A\, \overline{h}\right)}{\mathrm{d}t} + F_i^{\text{ent}}\end{aligned} \tag{6.15}$$

with

$$C_{\text{res}} = \frac{1}{2}\, \overline{d}\, \frac{c_w}{s_{\text{res}}^2}.$$

The mass balance Eq.6.8 remains unchanged:

$$\frac{\mathrm{d}V(t)}{\mathrm{d}t} = \frac{\mathrm{d}\left(A_b \overline{h}\right)}{\mathrm{d}t} = \frac{\rho_{\text{ent}}}{\rho_0}\, w_f\, h_{\text{ent}}\, \|\overline{\mathbf{u}}\| + \frac{A_b}{\rho_0}\, \frac{\tau^{(b)}}{e_b}\, \|\overline{\mathbf{u}}\| \tag{6.16}$$

The unknown $\overline{u}_1$, $\overline{u}_2$ and $\overline{h}$ satisfy Eq.6.14, Eq.6.15 and Eq.6.16. In equation Eq.6.15 the bottom shear stress $\tau^{(b)}$ remains unknown, and and a constitutive equation has to be introduced in order to completely solve the equations.

6.1.1.7 Friction Model

The problem can be solved by introducing a constitutive equation which describes the basal shear stress tensor $\tau^{(b)}$ as a function of the flow state of the avalanche.

$$\tau_i^{(b)} = f(\sigma^{(b)}, \overline{u}, \overline{h}, \rho_0, t, \mathbf{x}) \tag{6.17}$$

With

$\sigma^{(b)}$	normal component of the stress tensor
$\overline{u}$	average velocity
$\overline{h}$	average flow thickness
ρ_0	density
t	time
$\mathbf{x}$	position vector

Several friction models already implemented in the simulation tool are described here.

Mohr-Coulomb friction model

The Mohr-Coulomb friction model describes the friction interaction between twos solids. The bottom shear stress simply reads:

$$\tau^{(b)} = \tan\delta\, \sigma^{(b)}$$

$\tan\delta = \mu$ is the friction coefficient (and δ the friction angle). The bottom shear stress linearly increases with the normal stress component $\sigma^{(b)}$ ([BSG99, Sam07, WHP04, Zwi00]).

With this friction model, an avalanche starts to flow if the slope inclination is steeper than the friction angle δ. In the case of an infinite slope of constant inclination, the avalanche velocity would increase indefinitely. This is unrealistic to model snow avalanches because it leads to over prediction of the flow velocity. The Mohr-Coulomb friction model is on the other hand well suited to model granular flow. Because of its relative simplicity, this friction model is also very convenient to derive analytic solutions and validate the numerical implementation.

Chezy friction model

The Chezy friction model describes viscous friction interaction. The bottom shear stress then reads:

$$\tau^{(b)} = c_{\text{dyn}}\, \rho_0\, \bar{u}^2$$

c_{dyn} is the viscous friction coefficient. The bottom shear stress is a quadratic function of the velocity. ([BSG99, Sam07, WHP04, Zwi00]).

This model enables to reach more realistic velocities for avalanche simulations. The draw back is that the avalanche doesn't stop flowing before the slope inclination approaches zero. This implies that the avalanche flows to the lowest local point.

Voellmy friction model

Anton Voellmy was a Swiss engineer interested in avalanche dynamics [Voe55]. He first had the idea to combine both the Mohr-Coulomb and the Chezy model by summing them up in order to take advantage of both. This leads to the following friction law:

$$\tau^{(b)} = \tan\delta\, \sigma^{(b)} + \frac{g}{\xi}\, \rho_0\, \bar{u}^2$$

where ξ is the turbulent friction term. This model is described as Voellmy-Fluid [Sal04, Sam07].

VoellmyMinShear friction model

In order to increase the friction force and make the avalanche flow stop on steeper slopes than with the Voellmy friction relation, a minimum shear stress can be added to the Voellmy friction relation. This minimum value defines a shear stress under which the snowpack doesn't move, and induces a strong flow deceleration. This expression of the basal layer friction model also resembles the one used in the swiss RAMMS model, where the Voellmy model is modified by adding a yield stress supposed to account for the snow cohesion (https://ramms.slf.ch/en/modules/debrisflow/theory/friction-parameters.html).

$$\tau^{(b)} = \tau_0 + \tan\delta\,\sigma^{(b)} + \frac{g}{\xi}\,\rho_0\,\bar{u}^2$$

SamosAT friction model

SamosAT friction model is a modification of some more classical models such as Voellmy model *Voellmy friction model*. The basal shear stress tensor $\tau^{(b)}$ is expressed as ([Sam07]):

$$\tau^{(b)} = \tau_0 + \tan\delta\,\left(1 + \frac{R_s^0}{R_s^0 + R_s}\right)\,\sigma^{(b)} + \frac{\rho_0\,\overline{u}^2}{\left(\frac{1}{\kappa}\,\ln\frac{\overline{h}}{R} + B\right)^2}$$

With

τ_0	minimum shear stress
R_s	relation between friction and normal pressure (fluidization factor)
R	empirical constant
R_s^0	empirical constant
B	empirical constant
κ	empirical constant

The minimum shear stress τ_0 defines a lower limit below which no flow takes place with the condition $\rho_0\,\overline{h}\,g\,\sin\alpha > \tau_0$. α being the slope. τ_0 is independent of the flow thickness, which leeds to a strong avalanche deceleration, especially for avalanches with low flow heights. R_s is expressed as $R_s = \frac{\rho_0\,\overline{u}^2}{\sigma^{(b)}}$. Together with the empirical parameter R_s^0 the term $\frac{R_s^0}{R_s^0+R_s}$ defines the Coulomb basal friction. Therefore lower avalanche speeds lead to a higher bed friction, making avalanche flow stop already at steeper slopes α, than without this effect. This effect is intended to avoid lateral creep of the avalanche mass ([SG09]).

The default configuration also provides two additional calibrations for small- (< 25.000 m^3 release volume) and medium-sized (< 60.000 m^3 release volume) avalanches. A further constraint is the altitude of runout below 1500m msl for both.

Wet snow friction type

Note: This is an experimental option to account for wet snow conditions, still under development and not yet tested. Also the parameters are not yet calibrated.

In addition, com1DFA provides an optional friction model implementation to account for wet snow conditions. This approach is based on the Voellmy friction model but with an enthalpy dependent friction parameter.

$$\tau^{(b)} = \mu\,\sigma^{(b)} + c_{\text{dyn}}\,\rho_0\,\bar{u}^2$$

where,

$$\mu = \mu_0 \, \exp(-enthalpy/enthRef)$$

The total specific enthalpy of the particles is initialized based on their initial temperature, specific heat capacity, altitude and their velocity (which is zero for the initial time step). Throughout the computation, the particles specific enthalpy is then computed following:

$$enthalpy = totalEnthalpy - g\,z - 0.5\,\bar{u}^2$$

6.1.1.8 Dam

The dam is described by a crown line, that is to say a series of x, y, z points describing the crown of the dam (the dam wall is located on the left side of the line), by the slope of the dam wall (slope measured from the horizontal, β) and a restitution coefficient (describing if we consider more elastic or inelastic collisions between the particles and the dam wall, varying between 0 and 1).

The geometrical description of the dam is given on the figure Fig. 6.4. The dam crown line ($\mathbf{x}_{\text{crown}}$) is projected onto the topography, which provides us with the dam center line ($\mathbf{x}_{\text{center}}$). We compute the tangent vector to the center line ($\mathbf{t_f}$). From this tangent vector and the dam slope, it is possible to compute the wall tangent vector ($\mathbf{t_w}$). Knowing the wall tangent vector and height, it is possible to determine normal vector to the wall ($\mathbf{n_w}$) and the foot line which is the intersection between the dam wall and the topography ($\mathbf{x}_{\text{foot}}$).

When the dam fills up (flow thickness increases), the foot line is modified ($\mathbf{x}_{\text{foot}}^{\text{filled}} = \mathbf{x}_{\text{foot}} + \frac{h_v}{2}\mathbf{e_z}$). The normal and tangent vectors to the dam wall are readjusted accordingly.

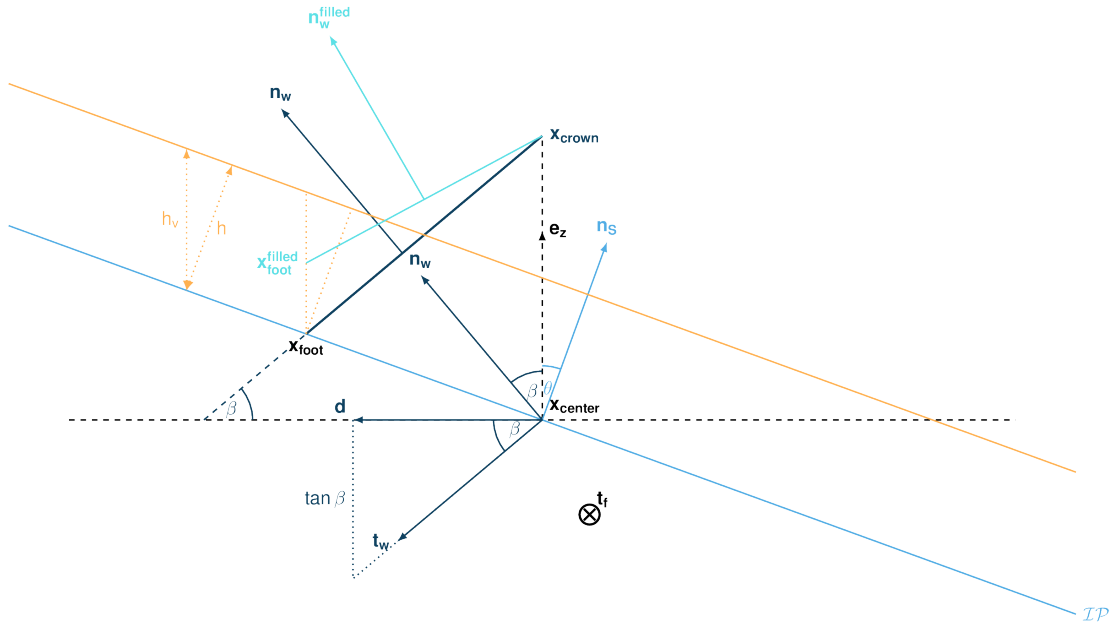


Fig. 6.4: Side view of the dam (cut view). $\mathbf{x}_{\text{crown}}$ describes the crown of the dam, $\mathbf{x}_{\text{center}}$ is the vertical projection of the crown on the topography (here the light blue line represents the topography). The tangent vector to the center line ($\mathbf{t_f}$) is computed from the center line points. The tangent vector to the center line with the dam slope angle enable to compute the tangent ($\mathbf{t_w}$) and normal ($\mathbf{n_w}$) vector to the dam wall. Finally, this normal vector is adjusted depending on the snow thickness at the dam location (filling of the dam , $\mathbf{n_w}^{\text{filled}}$)

In the initialization of the simulation, the dam tangent vector to the center line ($\mathbf{t_f}$), foot line ($\mathbf{x}_{\text{foot}}$) and normal vector to the wall ($\mathbf{n_w}$) are computed. The grid cells crossed by the dam as well as their neighbor cells are memorized (tagged as dam cells).

6.2 com1DFA DFA-Kernel numerics

Warning: This theory has not been fully reviewed yet.

The numerical method used in com1DFA mixes particle methods and mesh methods. Mass and momentum are tracked using particles but flow thickness is tracked using the mesh. The mesh is also used to access topographic information (surface elevation, normal vector) as well as for displaying results.

Mass Eq.6.16 and momentum Eq.6.15 balance equations as well as basal normal stress Eq.6.14 are computed numerically using a SPH method (**S**moothed **P**article **H**ydrodynamics) ([Mon92]) for the variables $\overline{\mathbf{u}} = (\overline{u}_1, \overline{u}_2)$ and $\overline{h}$ by discretization of the released avalanche volume in a large number of mass elements. SPH in general, is a mesh-less numerical method for solving partial differential equations. The SPH algorithm discretizes the numerical problem within a domain using particles ([Sam07, SG09]), which interact with each-other in a defined zone of influence. Some of the advantages of the SPH method are that free surface flows, material boundaries and moving boundary conditions are considered implicitly. In addition, large deformations can be modeled due to the fact that the method is not based on a mesh. From a numerical point of view, the SPH method itself is relatively robust.

6.2.1 Discretization

6.2.1.1 Space discretization

The domain is discretized in particles. The following properties are assigned to each particle p_k: a mass m_{p_k}, a thickness h_{p_k}, a density $\rho_{p_k} = \rho_0$ and a velocity $\mathbf{u}_{p_k} = (u_{p_k,1}, u_{p_k,2})$ (**those quantities are thickness averaged, note that we dropped the overline from** Eq.6.5 **for simplicity reasons**). In the following paragraphs, i and j indexes refer to the different directions in the NCS, whereas k and l indexes refer to particles.

The quantities velocity, mass and flow thickness are also defined on the fixed mesh. It is possible to navigate from particle property to mesh property using the interpolation methods described in *Mesh and interpolation*.

6.2.1.2 Time step

A fixed time step can be used or an adaptive time step that depends on the sph kernel radius as well as the particle size.

6.2.2 Mesh and interpolation

Here is a description of the mesh and the interpolation method that is used to switch from particle to mesh values and the other way around.

6.2.2.1 Mesh

For practical reasons, a 2D rectilinear mesh (grid) is used. Indeed the topographic input information is read from 2D raster files (with N_y and N_x rows and columns) which correspond exactly to a 2D rectilinear mesh. Moreover, as we will see in the following sections, 2D rectilinear meshes are very convenient for interpolations as well as for particle tracking. The 2D rectilinear mesh is composed of N_y and N_x rows and columns of square cells (of side length csz) and $N_y + 1$ and $N_x + 1$ rows and columns of vertices as described in Fig. 6.5. Each cell has a center and four vertices. The data read from the raster file is assigned to the cell centers. Note that although this is a 2D mesh, as we use a terrain-following coordinate system to perform our computations, this 2D mesh is oriented in 3D space and hence the projected side length corresponds to csz, whereas the actual side length and hence also the *Cell area*, depend on the local slope, expressed by the *Cell normals*.

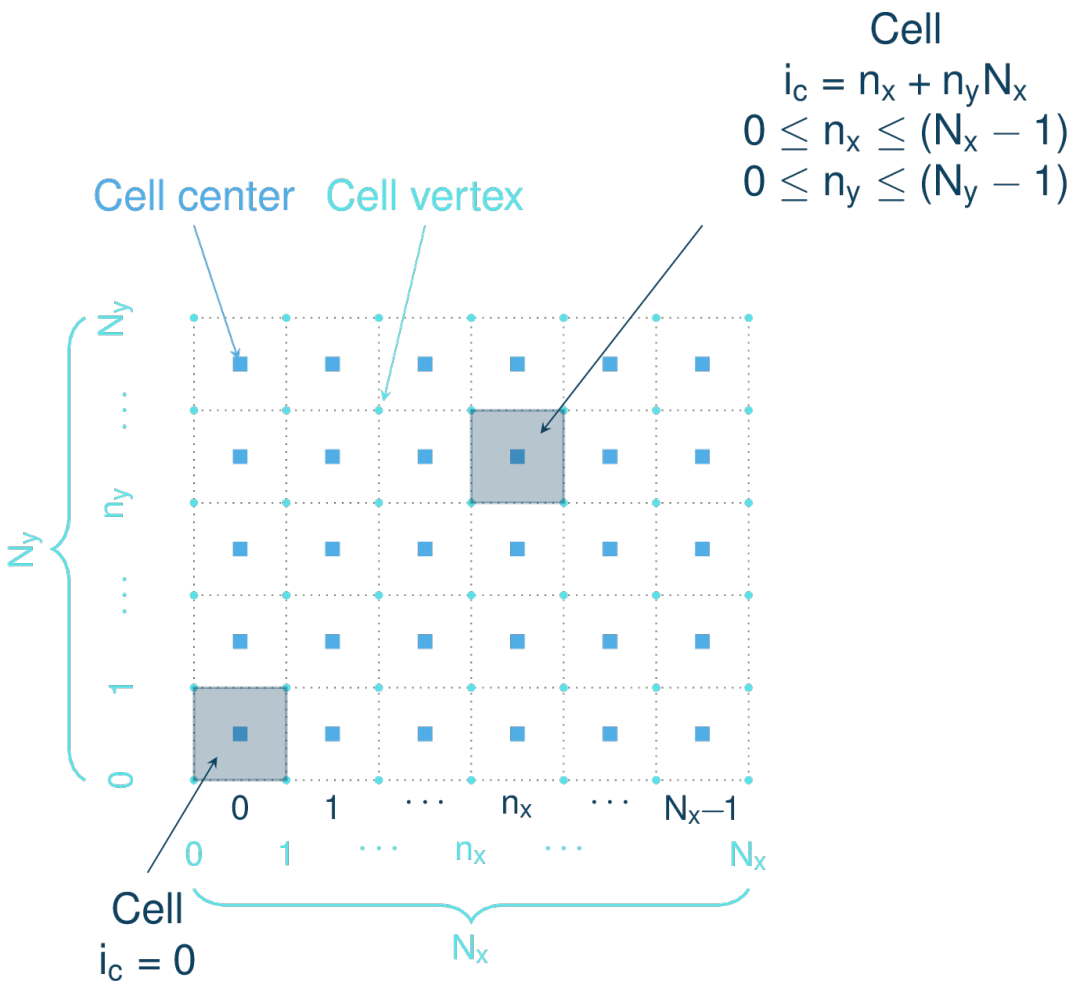


Fig. 6.5: Rectangular grid

Cell normals

There are many different methods available for computing normal vectors on a 2D rectilinear mesh. Several options are available in com1DFA.

The first one consists in computing the cross product of the diagonal vectors between four cell centers. This defines the normal vector at the vertices. It is then possible to interpolate the normal vector at the cell centers from the ones at the vertices.

The other methods use the plane defined by different adjacent triangles to a cell center. Each triangle has a normal and the cell center normal is the average of the triangles normal vectors.

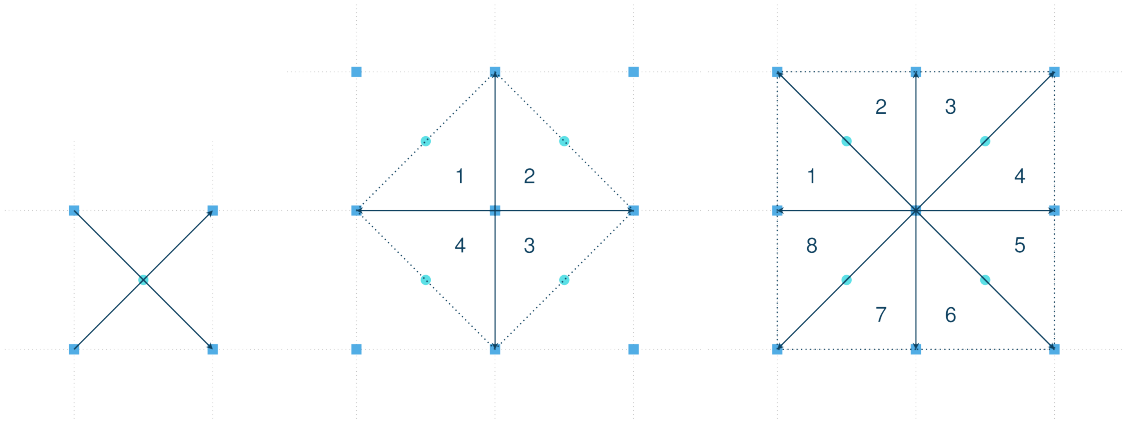


Fig. 6.6: Grid normal computation

Cell area

The cell area can be deduced from the grid cellsize and the cell normal. A cell is a plane ($z = ax + by + c$) of same normal as the cell center:

$$\mathbf{n} = \frac{1}{\sqrt{1 + a^2 + b^2}} \begin{vmatrix} -a \\ -b \\ 1 \end{vmatrix}$$

Surface integration over the cell extent leads to the area of the cell:

$$A_{cell} = \iint_S \mathrm{d}S = \int_0^{csz} \int_0^{csz} \sqrt{1 + \frac{\partial z}{\partial x}^2 + \frac{\partial z}{\partial y}^2}\, \mathrm{d}x\, \mathrm{d}y = csz^2 \sqrt{1 + \frac{\partial z}{\partial x}^2 + \frac{\partial z}{\partial y}^2} = \frac{csz^2}{n_z}$$

6.2.2.2 Interpolation

In the DFA kernel, mass, flow thickness and flow velocity can be defined at particle location or on the mesh. We need a method to be able to go from particle properties to mesh (field) values and from mesh values to particle properties.

Mesh to particle

On a 2D rectilinear mesh, scalar and vector fields defined at cell centers can be evaluated anywhere within the mesh using a bilinear interpolation between mesh cell centers. Evaluating a vector field simply consists in evaluating the three components as scalar fields.

The bilinear interpolation consists in successive linear interpolations in both x and y direction using the four nearest cell centers, two linear interpolations in the first direction (in our case in the y direction in order to evaluated f_{0v} and f_{1v}) followed by a second linear interpolation in the second direction (x in our case to finally evaluate f_{uv}) as shown on Fig. 6.7:

$$\begin{aligned} f_{0v} =& (1-v)f_{00} + vf_{01} \\ f_{1v} =& (1-v)f_{10} + vf_{11} \end{aligned}$$

and

$$\begin{aligned} f_{uv} =& (1-u)f_{0v} + uf_{1v} \\ =& (1-u)(1-v)f_{00} + (1-u)vf_{01} + u(1-v)f_{10} + uvf_{11} \\ =& w_{00}f_{00} + w_{01}f_{01} + w_{10}f_{10} + w_{11}f_{11} \end{aligned}$$

the w are the bilinear weights. The example given here is for a unit cell. For no unit cells, the u and v simply have to be normalized by the cell size.

Particles to mesh

Going from particle property to mesh value is also based on bilinear interpolation and weights but requires a bit more care in order to conserve mass and momentum balance. Flow thickness and velocity fields are determined on the mesh using, as intermediate step, mass and momentum fields. First, mass and momentum mesh fields can be evaluated by summing particles mass and momentum. This can be donne using the bilinear weights w defined in the previous paragraph (here f represents the mass or momentum and f_{uv} is the particle value. f_{nm} , $n, m \in \{0, 1\} \times \{0, 1\}$, are the cell center values):

$$\begin{aligned} f_{00} =& w_{00}f_{uv} \\ f_{01} =& w_{01}f_{uv} \\ f_{10} =& w_{10}f_{uv} \\ f_{11} =& w_{11}f_{uv} \end{aligned}$$

The contribution of each particle to the different mesh points is summed up to finally give the mesh value. This method ensures that the total mass and momentum of the particles is preserved (the mass and momentum on the mesh will sum up to the same total). Flow thickness and velocity mesh fields can then be deduced from the mass and momentum fields and the cell area (actual area of each grid cell, not the projected area).

6.2.3 Neighbor search

The lateral pressure forces are computed via the SPH flow thickness gradient. This method is based on particle interactions within a certain neighborhood, meaning that it is necessary to keep track of all the particles within the neighborhood of each particle. Computing the gradient of the flow thickness at a particle location, requires to find all the particles in its surrounding. Considering the number of particles and their density, computing the gradient ends up in computing a lot of interactions and represents the most computationally expensive part of the dense flow avalanche simulation. It is therefore important that the neighbor search is fast and efficient. [IOS+14] describe different rectilinear mesh neighbor search methods. In com1DFA, the simplest method is used. The idea is to locate each particle in a cell, this way, it is possible to keep track of the particles in each cell. To find the neighbors of a particle, one only needs to read the cell in which the particle is located (dark blue cell in Fig. 6.8) , find the direct adjacent cells in all directions (light blue cells) and simply read all particles within those cells. This is very easily achieved on rectilinear meshes because locating a particle in a cell is straightforward and finding the adjacent cells is also easily done.

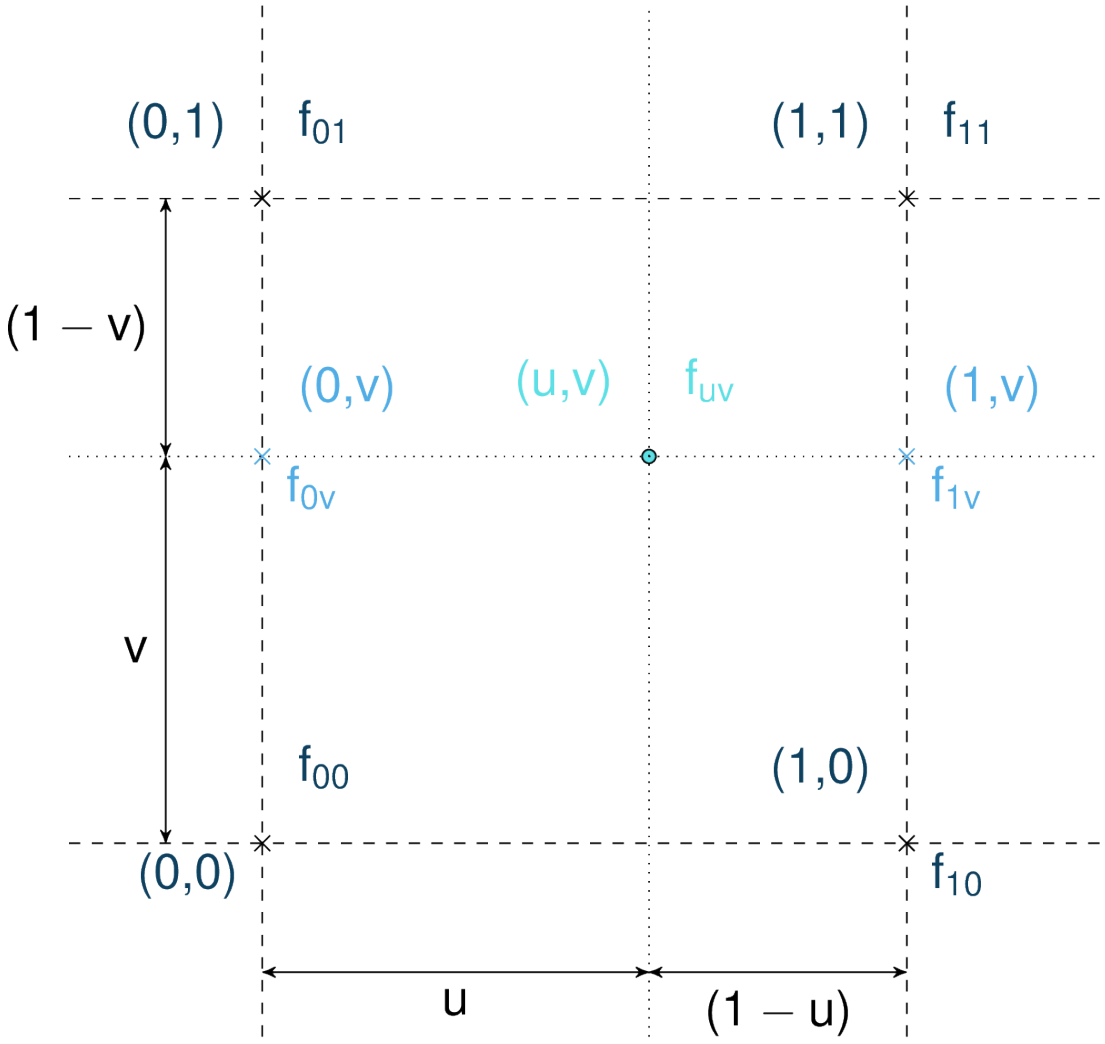


Fig. 6.7: Bilinear interpolation in a unit mesh (cell size is 1).

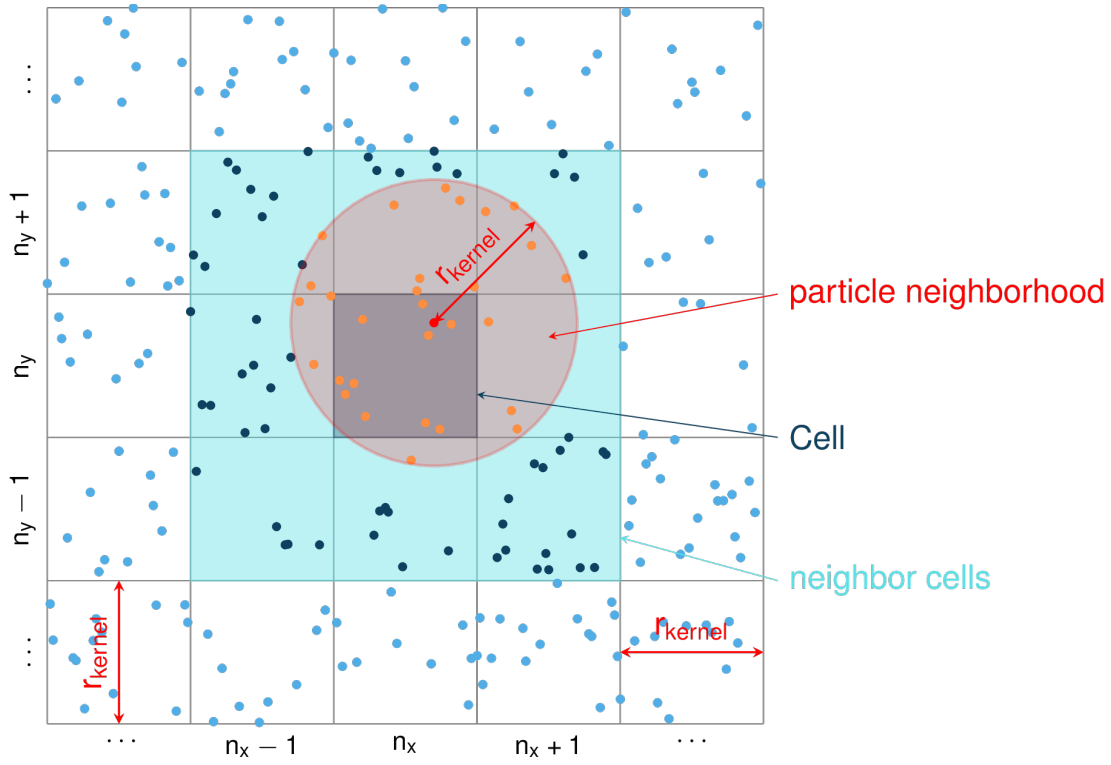


Fig. 6.8: Support mesh for neighbor search: if the cell side is bigger than the kernel length r_{kernel} (red circle in the picture), the neighbors for any particle in any given cell (dark blue square) can be found in the direct neighborhood of the cell itself (light blue squares)

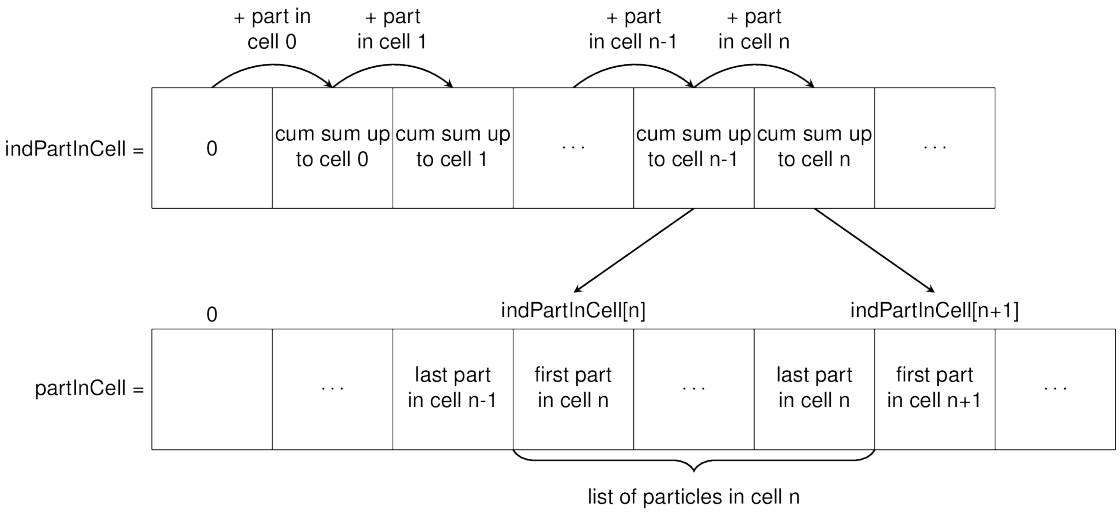


Fig. 6.9: The particles are located in the cells using two arrays. indPartInCell of size number of cells + 1 which keeps track of the number of particles in each cell and partInCell of size number of particles + 1 which lists the particles contained in the cells.

6.2.4 SPH gradient

SPH method can be used to solve thickness integrated equations where a 2D (respectively 3D) equation is reduced to a 1D (respectively 2D) one. This is used in ocean engineering to solve shallow water equations (SWE) in open or closed channels for example. In all these applications, whether it is 1D or 2D SPH, the fluid is most of the time, assumed to move on a horizontal plane (bed elevation is set to a constant). In the case of avalanche flow, the "bed" is sloped and irregular. The aim is to adapt the SPH method to apply it to thickness integrated equations on a 2D surface living in a 3D world.

6.2.4.1 Method

The SPH method is used to express a quantity (the flow thickness in our case) and its gradient at a certain particle location as a weighted sum of its neighbors properties. The principle of the method is well described in [LL10]. In the case of thickness integrated equations (for example SWE), a scalar function f and its gradient can be expressed as following:

$$\begin{aligned} f_k &= \sum_l f_l A_l \, W_{kl} \\ \nabla f_k &= -\sum_l f_l A_l \, \nabla W_{kl} \end{aligned} \tag{6.18}$$

Which gives for the flow thickness:

$$\begin{aligned} \overline{h}_k &= \frac{1}{\rho_0} \sum_l m_l \, W_{kl} \\ \nabla \overline{h}_k &= -\frac{1}{\rho_0} \sum_l m_l \, \nabla W_{kl} \end{aligned} \tag{6.19}$$

Where W represents the SPH-Kernel function.

The computation of its gradient depends on the coordinate system used.

Standard method

Let us start with the computation of the gradient of a scalar function $f\colon \mathbb{R}^2 \to \mathbb{R}$ on a horizontal plane. Let $P_k = \mathbf{x}_k = (x_{k,1}, x_{k,2})$ and $Q_l = \mathbf{x}_l = (x_{l,1}, x_{l,2})$ be two points in $\mathbb{R}^2$ defined by their coordinates in the Cartesian coordinate system $(P_k, \mathbf{e_1}, \mathbf{e_2})$. $\mathbf{r}_{kl} = \mathbf{x}_k - \mathbf{x}_l$ is the vector going from Q_l to P_k and $r_{kl} = \|\mathbf{r}_{kl}\|$ the length of this vector. Now consider the kernel function W:

$$\begin{aligned} W\colon \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R} &\to \mathbb{R} \\ (P_k, Q_l, r_0) &\mapsto W(P_k, Q_l, r_0) \end{aligned} \quad , r_0 \in \mathbb{R} \text{ is the smoothing kernel length}$$

In the case of the spiky kernel, W reads (2D case):

$$\begin{aligned} W_{kl} =& W(\mathbf{x_k}, \mathbf{x_l}, r_0) = W(\mathbf{x_k} - \mathbf{x_l}, r_0) = W(\mathbf{r_{kl}}, r_0) \\ =& \frac{10}{\pi r_0^5} \begin{cases} (r_0 - \|\mathbf{r_{kl}}\|)^3, & 0 \le \|\mathbf{r_{kl}}\| \le r_0 \\ 0, & r_0 < \|\mathbf{r_{kl}}\| \end{cases} \end{aligned} \tag{6.20}$$

$\|\mathbf{r_{kl}}\| = \|\mathbf{x_k} - \mathbf{x_l}\|$ represents the distance between particle k and l and r_0 the smoothing length.

Using the chain rule to express the gradient of W in the Cartesian coordinate system (x_1, x_2) leads to:

$$\nabla W_{kl} = \frac{\partial W}{\partial r} . \nabla r, \quad r = \|\mathbf{r}\| = \sqrt{(x_{k,1} - x_{l,1})^2 + (x_{k,2} - x_{l,2})^2} \tag{6.21}$$

with,

$$\frac{\partial W}{\partial r} = -3\frac{10}{\pi r_0^5}\begin{cases}(r_0 - \|\mathbf{r_{kl}}\|)^2, & 0 \leq \|\mathbf{r_{kl}}\| \leq r_0\\ 0, & r_0 < \|\mathbf{r_{kl}}\|\end{cases}$$

and

$$\frac{\partial r}{\partial x_{k,i}} = \frac{(x_{k,i} - x_{l,i})}{\sqrt{(x_{k,1} - x_{l,1})^2 + (x_{k,2} - x_{l,2})^2}}, \quad i = \{1, 2\}$$

which leads to the following expression for the gradient:

$$\nabla W_{kl} = -3\frac{10}{\pi r_0^5}\begin{cases}(r_0 - \|\mathbf{r_{kl}}\|)^2\dfrac{\mathbf{r_{kl}}}{r_{kl}}, & 0 \leq \|\mathbf{r_{kl}}\| \leq r_0\\ 0, & r_0 < \|\mathbf{r_{kl}}\|\end{cases} \tag{6.22}$$

The gradient of f is then simply:

$$\nabla f_k = -\sum_l f_l A_l \nabla W_{kl} \tag{6.23}$$

2.5D SPH method

We now want to express a function f and its gradient on a potentially curved surface and express this gradient in the 3 dimensional Cartesian coordinate system $(P_k, \mathbf{e_1}, \mathbf{e_2}, \mathbf{e_3})$.

Let us consider a smooth surface $\mathcal{S}$ and two points $P_k = \mathbf{x}_k = (x_{k,1}, x_{k,2}, x_{k,3})$ and $Q_l = \mathbf{x}_l = (x_{l,1}, x_{l,2}, x_{l,3})$ on $\mathcal{S}$. We can define $\mathcal{TP}$ the tangent plane to $\mathcal{S}$ in P_k. If $\mathbf{u}_k$ is the (none zero) velocity of the particle at P_k, it is possible to define the local orthonormal coordinate system $(P_k, \mathbf{V_1}, \mathbf{V_2}, \mathbf{V_3} = \mathbf{n})$ with $\mathbf{V_1} = \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}$ and $\mathbf{n}$ the normal to $\mathcal{S}$ at P_k. Locally, $\mathcal{S}$ can be assimilated to $\mathcal{TP}$ and Q_l to its projection Q'_l on $\mathcal{TP}$. The vector $\mathbf{r'}_{kl} = \mathbf{x}_k - \mathbf{x'}_l$ going from Q'_l to P_k lies in $\mathcal{TP}$ and can be express in the plane local basis:

$$\mathbf{r'}_{kl} = \mathbf{x}_k - \mathbf{x'}_l = v_{kl,1}\mathbf{V_1} + v_{kl,2}\mathbf{V_2}$$

It is important to define f properly and the gradient that will be calculated:

$$\begin{aligned} &f\colon \mathcal{TP} \subset \mathbb{R}^3 \to \mathbb{R}\\ &\quad (x_1, x_2, x_3) \mapsto f(x_1, x_2, x_3) = f(x_1(v_1, v_2), x_2(v_1, v_2)) = \tilde{f}(v_1, v_2)\end{aligned}$$

Indeed, since (x_1, x_2, x_3) lies in $\mathcal{TP}$, x_3 is not independent of (x_1, x_2):

$$\begin{aligned} &\tilde{f}\colon \mathcal{TP} \subset \mathbb{R}^2 \to \mathbb{R}\\ &\quad (v_1, v_2) \mapsto \tilde{f}(v_1, v_2) = \tilde{f}(v_1(x_1, x_2), v_2(x_1, x_2)) = f(x_1, x_2, x_3)\end{aligned}$$

The target is the gradient of $\tilde{f}$ in terms of the $\mathcal{TP}$ variables (v_1, v_2). Let us call this gradient $\nabla_{\mathcal{TP}}$. It is then possible to apply the *Standard method* to compute this gradient:

$$\nabla_{\mathcal{TP}} W_{kl} = \frac{\partial W}{\partial r}.\nabla_{\mathcal{TP}} r, \quad r = \|\mathbf{r}\| = \sqrt{v_{kl,1}^2 + v_{kl,2}^2} \tag{6.24}$$

Which leads to:

$$\nabla_{\mathcal{TP}} W_{kl} = -3\frac{10}{\pi r_0^5}\frac{(r_0 - \|\mathbf{r'_{kl}}\|)^2}{r'_{kl}}\begin{cases}v_{kl,1}\mathbf{V_1} + v_{kl,2}\mathbf{V_2}, & 0 \leq \|\mathbf{r'_{kl}}\| \leq r_0\\ 0, & r_0 < \|\mathbf{r'_{kl}}\|\end{cases} \tag{6.25}$$

$$\nabla_{\mathcal{TP}} \tilde{f}_k = -\sum_l \tilde{f}_l A_l \nabla W_{kl} \tag{6.26}$$

This gradient can now be expressed in the Cartesian coordinate system. It is clear that the change of coordinate system was not needed:

$$\nabla_{\mathcal{TP}} W_{kl} = -3\frac{10}{\pi r_0^5}\frac{(r_0 - \|\mathbf{r'_{kl}}\|)^2}{r'_{kl}} \begin{cases} r_{kl,1}\mathbf{e_1} + r_{kl,2}\mathbf{e_2} + r_{kl,3}\mathbf{e_3}, & 0 \leq \|\mathbf{r'_{kl}}\| \leq r_0 \\ 0, & r_0 < \|\mathbf{r'_{kl}}\| \end{cases}$$

The advantage of computing the gradient in the local coordinate system is if the components (in flow direction or in cross flow direction) need to be treated differently.

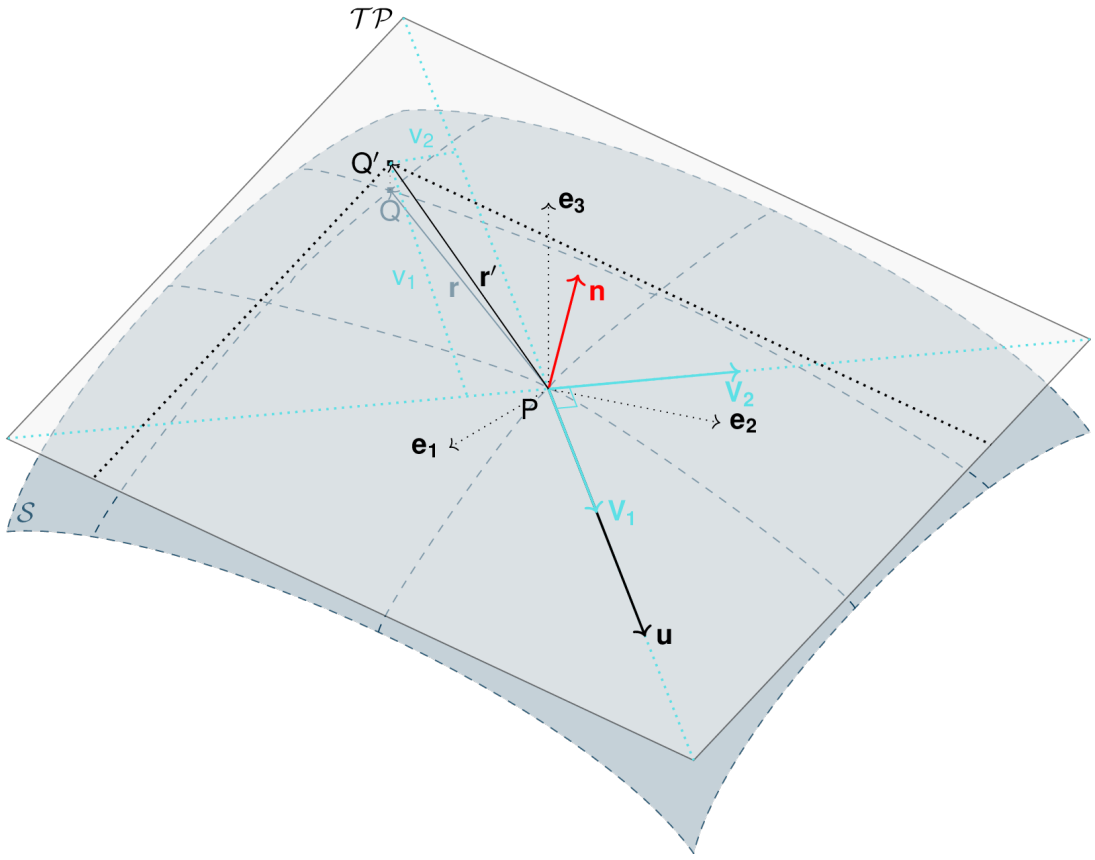


Fig. 6.10: Tangent plane and local coordinate system used to apply the SPH method

6.2.5 Particle splitting and merging

There are two different approaches treating splitting of particles in com1DFA. The first one only deals with splitting of particles with too much mass('split only'). The second approach, "split/merge" approach aims at keeping a stable amount of particles within a given range. This is done in order to guaranty a sufficient accuracy of the sph flow thickness gradient computation.

6.2.5.1 Split (default)

If the `splitOption` is set to 0, particles are split because of snow entrainment. In this case, particles that entrain snow grow, i.e. their mass increases. At one point the mass of the particles is considered to be too big and this particle is split in two. The splitting operation happens if the mass of the particle exceeds a threshold value ($mPart > massPerPart \times thresholdMassSplit$), where `thresholdMassSplit` is specified in the configuration file and `massPerPart` depends on the chosen `massPerParticleDeterminationMethod` as defined here: *Initialize particles*. When a particle is split a new child particle is created with the same properties as the parent apart from mass and position. Both parent and child get half of the parent mass. The parent and child's position are adjusted: the first / second is placed forward / backward in the direction of the velocity vector at a distance $distSplitPart \times rPart$ of the initial parent position. Particles are considered to have a circular basal surface $A = \frac{m}{\rho} = \pi r^2$.

6.2.5.2 Split and merge

If the `splitOption` is set to 1 particles are split or merged in order to keep the particle count as constant as possible within the kernel radius. Assessing the number of particles within one kernel radius is done based on the particle area. Particles are assumed to be cylindrical, i.e the base is a circle. For particle `k` we have $A_k = \frac{m_k}{\rho}$. The area of the support domain of the sph kernel function is πr_0^2. The aim is to keep `nPPK` particles within the kernel radius. The particles are split if the estimated number of particles per kernel radius $\frac{\pi r_0^2}{A_k}$ falls below a given value of $n_{PPK}^{min} = C_{n_{PPK}}^{min} n_{PPK}$. Particles are split using the same method as in *Split (default)*. Similarly, particles are merged if the estimated number of particles per kernel radius exceeds a given value $n_{PPK}^{max} = C_{n_{PPK}}^{max} n_{PPK}$. In this case particles are merged with their closest neighbor. The new position and velocity is the mass averaged one. The new mass is the sum. Here, two coefficients `C_{n_{PPK}}^{min}` and `C_{n_{PPK}}^{max}` were introduced. A good balance needs to be found for the coefficients so that the particles are not constantly split or merged but also not too seldom. The split and merge steps happen only once per time step and per particle.

6.2.6 Artificial viscosity

Two options are available to add viscosity to stabilize the numerics. The first option consists in adding artificial viscosity (`viscOption` = 1). The second option attempts to adapt the Lax-Friedrich scheme (usually applied to meshes) to the particle method (`viscOption` = 2). Finally, `viscOption` = 0 deactivates any viscosity force.

6.2.6.1 SAMOS Artificial viscosity

In *Governing Equations for the Dense Flow Avalanche*, the governing equations for the DFA were derived and all first order or smaller terms where neglected. Among those terms is the lateral shear stress. This term leads toward the homogenization of the velocity field. It means that two neighbor elements of fluid should have similar velocities. The aim behind adding artificial viscosity is to take this phenomena into account. The following viscosity force is added:

$$\begin{aligned}\mathbf{F_{viscosity}} &= -\frac{1}{2}\rho C_{Lat}\|\mathbf{du}\|^2 A_{Lat}\frac{\mathbf{du}}{\|\mathbf{du}\|} \\ &= -\frac{1}{2}\rho C_{Lat}\|\mathbf{du}\| A_{Lat}\mathbf{du}\end{aligned}$$

Where the velocity difference reads $\mathbf{du} = \mathbf{u} - \bar{\mathbf{u}}$ ($\bar{\mathbf{u}}$ is the mesh velocity interpolated at the particle position). C_{Lat} is a coefficient that rules the viscous force. It would be the equivalent of C_{Drag} in the case of the drag force. The C_{Lat} is a numerical parameter that depends on the mesh size. Its value is set to 100 and should be discussed and further tested.

Adding the viscous force

The viscous force acting on particle k reads:

$$\begin{aligned}\mathbf{F_k^{viscosity}} &= -\frac{1}{2}\rho C_{Lat}\|\mathbf{du}_k^{old}\|A_{Lat}\mathbf{du}_k^{new} \\ &= -\frac{1}{2}\rho C_{Lat}\|\mathbf{u}_k^{old} - \bar{\mathbf{u}}_k^{old}\|A_{Lat}(\mathbf{u}_k^{new} - \bar{\mathbf{u}}_k^{old})\end{aligned}$$

Updating the velocity is done in two steps. First adding the explicit term related to the mean mesh velocity and then the implicit term which leads to:

$$\mathbf{u}_k^{new} = \frac{\mathbf{u}_k^{old} - C_{vis}\bar{\mathbf{u}}_k^{old}}{1 + C_{vis}}$$

With $C_{vis} = \frac{1}{2}\rho C_{Lat}\|\mathbf{du}_k^{old}\|A_{Lat}\frac{dt}{m}$

6.2.6.2 Ata Artificial viscosity

An upwind method based on Lax-Friedrichs scheme

Shallow Water Equations are well known for being hyperbolic transport equations. They have the particularity of carrying discontinuities or shocks which will cause numerical instabilities.

A decentering in time allows to better capture the discontinuities. This can be done in the manner of the Lax-Friedrich scheme as described in [AS05], which is formally the same as adding a viscous force. Implementing it for the SPH method, this viscous force applied on a given particle k can be expressed as follows:

$\mathbf{u}_{kl} = \mathbf{u}_k - \mathbf{u}_l$ is the relative velocity between particle k and l, $\mathbf{r}_{kl} = \mathbf{x}_k - \mathbf{x}_l$ is the vector going from particles l to particle k and $\lambda_{kl} = \frac{c_k + c_l}{2}$ with $c_k = \sqrt{gh_l}$ the wave speed. The λ_{kl} is obtained by turning expressions related to time and spatial discretization parameters into an expression on maximal speed between both particles in the Lax Friedrich scheme.

Due to the expression of the viscosity force, it makes sense to compute it at the same place where the SPH pressure force are computed (for this reason, the `viscOption` = 2 corresponding to the "Ata" viscosity option is only available in combination with the `sphOption` = 2).

6.2.7 Forces discretization

6.2.7.1 Lateral force

The SPH method is introduced when expressing the flow thickness gradient for each particle as a weighted sum of its neighbors ([LL10, Sam07]). From now on the p for particles in p_k is dropped (same applies for p_l).

The lateral pressure forces on each particle are calculated from the compression forces on the boundary of the particle. The boundary is approximated as a square with the base side length $\Delta s = \sqrt{A_p}$ and the respective flow height. This leads to (subscript $|_{.,i}$ stands for the component in the i^{th} direction, $i = 1, 2$):

$$F_{k,i}^{\text{lat}} = K_{(i)} \oint_{\partial A_k} \left(\int_b^s \sigma_{33}\, n_i \, \mathrm{d}x_3 \right) \mathrm{d}l$$

From equation Eq.6.15

$$F_{k,i}^{\text{lat}} = K_{(i)} \frac{\Delta s}{2} \left((\overline{h}\,\overline{\sigma}_{33}^{(b)})_{x_i - \frac{\Delta s}{2}} - (\overline{h}\,\overline{\sigma}_{33}^{(b)})_{x_i + \frac{\Delta s}{2}} \right) = K_{(i)} \frac{\Delta s^2}{2} \left. \frac{d\,\overline{h}\,\overline{\sigma}^{(b)}}{d\,x_i} \right|_k$$

The product of the average flow thickness $\overline{h}$ and the basal normal pressure $\overline{\sigma}_{33}^{(b)}$ reads (using equation Eq.6.14 and dropping the curvature acceleration term):

$$\overline{h}\,\overline{\sigma}^{(b)} = \overline{h}^2\,\rho_0 \left(g_3 - \overline{u_1}^2 \frac{\partial^2 b}{\partial x_1^2} \right) \approx \overline{h}^2\,\rho_0\,g_3$$

Which leads to, using the relation Eq.6.18:

$$F_{k,i}^{\text{lat}} = K_{(i)}\,\rho_0\,g_3\,A_k\,\overline{h}_k \,.\, \left. \frac{d\,\overline{h}}{d\,x_i} \right|_k = -K_{(i)}\,m_i\,g_3 \,.\, \frac{1}{\rho_0} \sum_l m_l \left. \frac{d\,W_{kl}}{d\,x_i} \right|_l \tag{6.27}$$

6.2.7.2 Bottom friction force

The bottom friction forces on each particle depend on the chose friction model. Using the SamosAT friction model (using equation Eq.6.14 for the expression of $\sigma_k^{(b)}$) the formulation of the bottom friction forec reads:

$$F_{k,i}^{\text{bot}} = -\frac{\overline{u}_{k,i}}{\|\mathbf{u}_k\|} A_k\,\tau_k^{(b)} = -\delta_{k1}\,A_k \left(\tau_0 + \tan\delta \left(1 + \frac{R_s^0}{R_s^0 + R_s} \right) \sigma_k^{(b)} + \frac{\rho_0\,\overline{\mathbf{u}}_k^2}{\left(\frac{1}{\kappa} \ln \frac{\overline{h}}{R} + B \right)^2} \right) \tag{6.28}$$

6.2.7.3 Added resistance force

The resistance force on each particle reads (where h_k^{eff} is a function of the average flow thickness $\overline{h}_k$):

$$F_{k,i}^{\text{res}} = -\rho_0\,A_k\,h_k^{\text{eff}}\,C_{\text{res}}\,\|\overline{\mathbf{u}}_k\|^2\,\frac{\overline{u}_{k,i}}{|\overline{\mathbf{u}}_k\|} \tag{6.29}$$

Both the bottom friction and resistance force are friction forces. The expression above represent the maximal friction force that can be added. This maximal force is added if the particles are flowing. If not, the friction force equals the driving forces. See [MCVB+03] for more information.

6.2.7.4 Entrainment force

The term $-\overline{u_i}\,\rho_0\,\frac{\mathrm{d}(A\,\overline{h})}{\mathrm{d}t}$ related to the entrained mass in Eq.6.3 now reads:

$$-\overline{u}_{k,i}\,\rho_0\,\frac{\mathrm{d}}{\mathrm{d}t}\,\left(A_k\,\overline{h}_k\right) = -\overline{u}_{k,i}\,A_k^{\text{ent}}\,q_k^{\text{ent}}$$

The mass of entrained snow for each particle depends on the type of entrainment involved (plowing or erosion) and reads:

$$\rho_0\,\frac{\mathrm{d}}{\mathrm{d}t}\,\left(A_k\,\overline{h}_k\right) = \frac{\mathrm{d}\,m_k}{\mathrm{d}t} = A_k^{\text{ent}}\,q_k^{\text{ent}}$$

with

$$A_k^{\text{plo}} = w_f\,h_k^{\text{ent}} = \sqrt{\frac{m_k}{\rho_0\,\overline{h}_k}}\,h_k^{\text{ent}} \qquad \text{and} \quad q_k^{\text{plo}} = \rho_{\text{ent}}\,\|\overline{\mathbf{u}}_k\| \qquad \text{for plowing}$$

$$A_k^{\text{ero}} = A_k = \frac{m_k}{\rho_0\,\overline{h}_k} \qquad \text{and} \quad q_k^{\text{ero}} = \frac{\tau_k^{(b)}}{e_b}\,\|\overline{\mathbf{u}}_k\| \qquad \text{for erosion}$$

Finaly, the entrainment force reads:

$$F_{k,i}^{\text{ent}} = -w_f\,(e_s + q_k^{\text{ent}}\,e_d)$$

6.2.8 Adding forces

The different components are added following an operator splitting method. This means particle velocities are updated successively with the different forces.

6.2.8.1 Adding artificial viscosity

If the viscosity option (`viscOption`) is set to 1, artificial viscosity is added first, as described in *Artificial viscosity* (this is the default option). With `viscOption` set to 0, no viscosity is added. Finally, if `viscOption` is set to 2, artificial viscosity is added during SPH force computation. (TODO add link to description)

6.2.8.2 Adding entrainment

Entrainment is taken into account by first adding the component representing the loss of momentum due to acceleration of the entrained mass $-\overline{u}_{k,i}\, A_k^{\text{ent}}\, q_k^{\text{ent}}$. Second by adding the force due to the need to break and compact the entrained mass ($F_{k,i}^{\text{ent}}$) as described in *Entrainment force*.

6.2.8.3 Adding driving forces

The driving forces -gravity force and lateral forces- are taken into account next. The velocity is updated explicitly.

6.2.8.4 Adding friction forces

Both the bottom friction and resistance forces act against the flow. Two methods are available to add these forces in com1DFA.

An implicit method:

$$\mathbf{u}_k^{new} = \frac{\mathbf{u}_k^{old}}{1 + \frac{C_k^{\text{fric}} \Delta t}{m_k}}$$

where $F_{k,i}^{\text{fric}} = C_k^{\text{fric}} u_{k,i}^{new} = F_{k,i}^{\text{res}} + F_{k,i}^{\text{bot}}$ (the two forces are described in *Bottom friction force* and *Added resistance force*).

This implicit method has a few draw-backs. First the flow does not start properly if the friction angle δ is too close to the slope angle. Second, the flow never properly stops, even if the particles physically should, i.e. particles keep oscillating back and force around their end position.

An explicit method:

The method based on [MCVB+03] addresses these two issues. The idea is that the friction forces only modify the magnitude of velocity and not the direction. This means dissipation, so the friction force can not become a driving force. Moreover, the friction force magnitude depends on the particle state, i.e. if it is flowing or at rest. The friction force is expressed:

$$\mathbf{F}_k^{\text{fric}} = -\|\mathbf{F}_k^{\text{fric}}\| \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}$$

with:

$$\|\mathbf{F}_k^{\text{fric}}\| \leq \|\mathbf{F}_k^{\text{fric}}\|_{max}$$

If the velocity of the particle **k** reads $\mathbf{u}_k^{old}$ after adding the driving forces, adding the fiction force leads to :

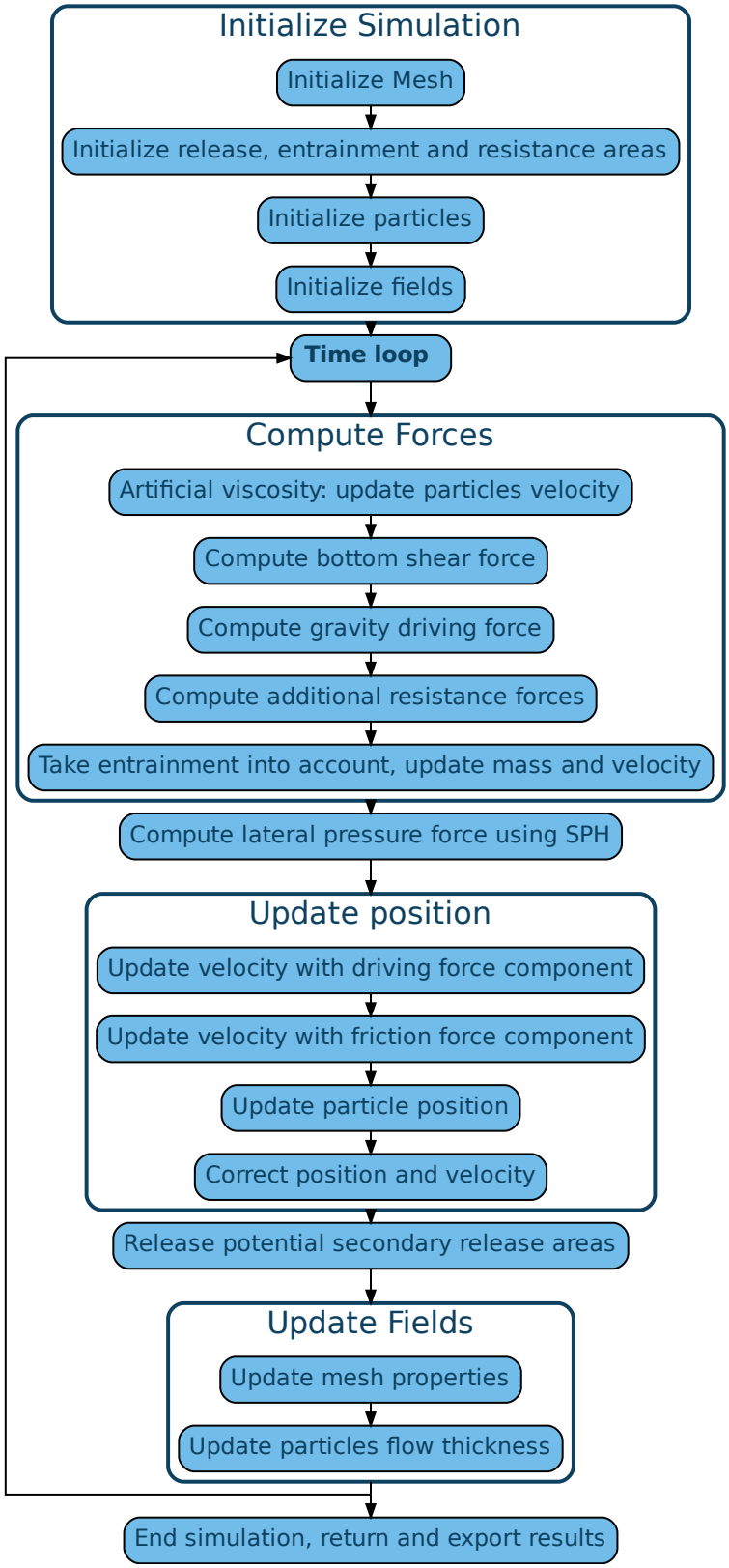
$$\mathbf{u}_k = \mathbf{u}_k^{old}(1 - \frac{\Delta t}{m}\frac{\|\mathbf{F}_k^{\text{fric}}\|}{\|\mathbf{u}_k^{old}\|}), \quad \|\mathbf{F}_k^{\text{fric}}\| = \|\mathbf{F}_k^{\text{fric}}\|_{max}$$

at the condition that $1 \geq \frac{\Delta t}{m}\frac{\|\mathbf{F}_k^{\text{fric}}\|_{max}}{\|\mathbf{u}_k^{old}\|}$. If on the contrary $1 \leq \frac{\Delta t}{m}\frac{\|\mathbf{F}_k^{\text{fric}}\|_{max}}{\|\mathbf{u}_k^{old}\|}$, the friction would change the velocity direction which is nonphysical. In this case, the particle will stop before the end of the time step. This allows the particles to start and stop flowing properly.

6.3 com1DFA Algorithm and workflow

6.3.1 Algorithm graph

The following graph describes the Dense Flow Avalanche simulation workflow (the different nodes are clickable and link to the detailed documentation)


Initialize Simulation
Initialize Mesh
Initialize release, entrainment and resistance areas
Initialize particles
Initialize fields
Time loop
Compute Forces
Artificial viscosity: update particles velocity
Compute bottom shear force
Compute gravity driving force
Compute additional resistance forces
Take entrainment into account, update mass and velocity
Compute lateral pressure force using SPH
Update position
Update velocity with driving force component
Update velocity with friction force component
Update particle position
Correct position and velocity
Release potential secondary release areas
Update Fields
Update mesh properties
Update particles flow thickness
End simulation, return and export results

6.3.2 Initialization:

At the beginning of the simulation, the avalanche folder and the configuration are read (configuration:Configuration). Input data is fetched according to the chosen configuration. Mesh, particles and fields are subsequently initialized.

6.3.2.1 Initialize Mesh

Read DEM ascii file provided in the Input folder (only one DEM ascii file allowed). If the DEM cell size is different from the :`meshCellSize` specified in the configuration from more than `meshCellSizeThreshold` [m] the DEM is remeshed (`in3Trans.geoTrans.remesh()`).

Prepare DEM for simulation, compute surface normals vector field, cell area (*Mesh*).

This is done in the `com1DFA.com1DFA.initializeMesh()` function.

Go back to *Algorithm graph*

6.3.2.2 Initialize release, entrainment and resistance areas

Read and check shapefiles according to the configuration (check consistency between what is required by the configuration file and what is available in the `Inputs` folder). Convert shapefile features (polygons) to rasters (*`in3Utils.geoTrans.prepareArea()`*). Check consistency of rasters according to the following rules:

- multiple release features in the release and secondary release shapefiles are allowed but they should not overlap. If they do, simulation terminates with an error message.
- secondary release and entrainment rasters should no overlap between each other or with the main release. If they do, the overlapping part is removed. Order of priority is: main release, secondary release, entrainment area.

Go back to *Algorithm graph*

6.3.2.3 Initialize Dam

If the dam option is activated (which is the case in the default configuration `dam` is `True`) AND the dam input shapefile is provided, the dam is initialized and will be take into account for in the DFA computation.

6.3.2.4 Initialize particles

Particles are initialized according to the release raster extracted from the release shapefile and the mass per particle determination method (`massPerParticleDeterminationMethod`) specified in the configuration. The mass per particle determination method can be chosen between:

- MPPDIR= mass per particle direct. The `massPerPart` value is taken from the configuration and is the same for all cells.
- MPPDH= mass per particles through release thickness. The `massPerPart` value is computed using the release thickness per particle `deltaTh` value given in the configuration and the area of the release mesh cell: massPerPart $= \rho \times$ cellArea $\times$ deltaTh.
- MPPKR= mass per particles through number of particles per kernel radius. There is no `massPerPart` since it can vary from one cell to another depending on the release thickness of the cells. The aim of this method is to ensure a constant density of particles within the snow domain (`nPPK` particles per kernel radius is the target value). This is related to the SPH method used for computing the flow thickness gradient. It requires a sufficient number of particles to properly approximate the flow thickness gradient. It makes the most sense to combine the MPPKR particle initialization method with the *splitOption* 1. In this combination, the particles will be merged or split to keep a constant density of particles per kernel radius (*Particle splitting and merging*).

Note: If MPPDIR is used, consider adapting the mass per particle value when changing the mesh cell size from the default. This is important because, when using MPPDIR, the total number of particles is independent of the cell size. Hence, reducing the cell size results in less particles per cell, whereas when using MPPDH, the number of particles per cell is fixed (considering the respective release thickness and deltaTh value). Reducing the cell size will increase the total number of particles but not the number of particles per cell. Finally, using the MPPKR method, the number of particles per cell is independent from both cell size and release thickness (`nPPK` particles per kernel radius is the target value).

The number of particles placed in each release cell `nPartPerCell` is computed according to the `massPerPart` or `nPPK` depending on the `massPerParticleDeterminationMethod` chosen and the area and/or release thickness of the cell. The number should be an integer meaning that the float is rounded up or down with a probability corresponding to the decimal part (i.e. 5.7 will be rounded to 6 with a probability of 0.7 and 5 with a probability of 0.3). This ensures a better match with the desired `massPerPart` value.

There are then different ways to place the particles in the cells. This is decided by the `initPartDistType` parameter in the configuration file:

- `random` initialization: The `nPartPerCell` particles are placed randomly within each release cell. This initialization method allows steps of ones in the choice of `nPartPerCell`, which enables to avoid large jumps in the mass of the particles or the number of particles per Cell between one release cell to another. Due to the random initialization process, some particles cluster can appear. For operational applications, this does not seem to have a significant impact since the particles will redistribute in the first few time steps due to the pressure gradient. For some specific research applications (e.g. the dam break test), clusters might disturb the results. In this case, applying an initialization reprocessing can help (see `iniStep` in the configuration file). The random number generator is controlled by a random seed which ensures the possibility to reproduce the DFA results.
- `uniform` initialization: The release (square) cell is divided into (square) subcells (as many as `nPartPerCell`). The particles are placed in the center of each subcell. This method allows only steps of 1, 4, 9, 16, ... due to the square subdivision of the cells. This can lead, in some cases, to significant variations in the number of particles per cell or in the mass of the particles. This can then lead to spurious numerical artifacts. The particles are aligned with the grid, which can also lead to some spurious numerical artifacts.
- `semirandom` initialization: This method uses the uniform division of the release cell into subcells but the particles are placed randomly within each subcell. This method has the same disadvantage as the `uniform` method, it allows only steps of 1, 4, 9, 16, ... due to the square subdivision of the cells. This can lead, in some cases, to significant variations in the number of particles per cell or in the mass of the particles. But due to the random position of particles in the subcells particles are not aligned anymore with the grid.
- `triangular` initialization: The particles are initialized along a regular triangular mesh within the release area. This allows a regular distribution of particles with steps of one (like in the case of `random` initialization). The disadvantages are that the same triangle size is used within the whole release leading in some cases to significant differences in the mass of the particles. The same comment applies if multiple release features are used. Finally, this method only works for constant release thickness.

  Note: This initialization option is meant to be used when the snow slide option is activated (`cohesion` activated). The behavior in the standard case (`cohesion` deactivated) has not been tested.

Particle properties

Other particles properties are also initialized here:

- `x`, `y`, `z` - coordinates
- `m` - mass of particle [kg]
- `h` - flow thickness [m]
- `ux`, `uy`, `uz` - velocity components [ms-1]
- `uAcc` - approximation for particle acceleration between each computational time step (simply computed by (velocityMagnitude_t0 - velocityMagnitude_t1) / dt)
- `trajectoryLengthXY` - traveled length of particle accumulated over time in xy plane
- `trajectoryLengthXYCor` - trajectoryLengthXY corrected with the angle difference between the slope and the normal
- `trajectoryLengthXYZ` - traveled length of a particle accumulated over time in xyz
- `travelAngle` - travel angle computed using arctan((z0-z)/trajectoryLengthXY)
- `ID, parentID` - particle IDs and parentID required if splitting, merging
- `t`- corresponding time step

For more details, see `com1DFA.com1DFA.initializeParticles()`.

Go back to *Algorithm graph*

6.3.2.5 Initialize fields

All fields (mesh values defined as a raster) are initialized. Flow velocity, pressure, peak flow velocity and peak pressures are set to zero. Flow thickness and peak flow thickness are set according to the initial particle distribution. See `com1DFA.com1DFA.initializeFields()`

Go back to *Algorithm graph*

6.3.3 Time scheme and iterations:

The mass and momentum equations described in *Governing Equations for the Dense Flow Avalanche* are solved numerically in time using an operator splitting method. The different forces involved are sequentially added to update the velocity (see *Adding forces*). Position is then updated using a centered Euler scheme. The time step can either be fixed or dynamically computed using the Courant–Friedrichs–Lewy (CFL) condition (in the second case one must set `cflTimeStepping` to `True` and set the desired CFL coefficient).

Go back to *Algorithm graph*

6.3.4 Compute Forces:

This section gives an overview of the different steps to compute the forces acting on the snow particles. Those forces are separated in several terms: A gravity driving fore (F_{drive}), a friction force (F_{fric}), an entrainment force (related to the entrained mass of snow) and an artificial viscous force. Those forces are computed by the two following functions `com1DFA.DFAfunctionsCython.computeForceC()` and `com1DFA.DFAfunctionsCython.computeForceSPHC()`.

Go back to *Algorithm graph*

6.3.4.1 Artificial viscosity

This viscous friction force is artificially added to the numerical computation. The aim of this force is to stabilize the simulation and prevent neighbor particles to have too significantly different velocities. Physically, this force also makes sense and corresponds to some second order forces that were neglected (lateral shear stress) as explained in *Artificial viscosity*. This force is controlled by the `subgridMixingFactor` in the configuration file. Setting this parameter to 0 deactivates the artificial viscosity term. The default value (set to 100) does not have any physical foundation yet. Future work will help defining this parameter in a more physical way. Remember that the artificial viscosity is dependent on the grid cell size.

The velocity is updated immediately after using an explicit/implicit formulation.

Go back to *Algorithm graph*

6.3.4.2 Compute friction forces

The friction force encompasses all forces that oppose the motion of the particles. One of those forces is the bottom shear force. The other is an optional resistance force. Both components are added to the F_{fric} force term.

This force accounts for the friction between the snow particles and the bottom surface. The expression of the bottom shear stress depends on the friction model chosen but can be written in the following general form, $\tau_i^{(b)} = f(\sigma^{(b)}, \overline{u}, \overline{h}, \rho_0, t, \mathbf{x})$. The friction model is set by the `frictModel` value and the corresponding parameters can be set in the configuration file. More details about the different friction models are given in *Friction Model*. Be aware that the normal stress on the bottom surface $\sigma^{(b)}$ is composed of the normal component of the gravity force and the curvature acceleration term as shown in Eq.6.14. It is possible to deactivate the curvature acceleration component of the shear stress by setting the `curvAcceleration` coefficient to 0 in the configuration file.

An additional friction force called resistance can be added. This force aims to model the added resistance due to the specificity of the terrain on which the avalanche evolves, for example due to forests. To add a resistance force, one must provide a resistance shapefile in the `Inputs/RES` folder and switch the `simType` to `res`, `entres` or `available` to take this resistance area into account. Then, during the simulation, all particles flowing through this resistance area will undergo an extra resistance force. More details about how this force is computed and the different parameters chosen are found in *Resistance*.

Go back to *Algorithm graph*

6.3.4.3 Compute driving force

This force takes into account the gravity force, which is the driving force of the snow motion. The expression of this force is rather simple, it represents the tangential (tangent to the surface) part of the gravity force (the normal part of the force is accounted for in the friction term).

Go back to *Algorithm graph*

6.3.4.4 Take entrainment into account

Snow entrainment can be added to the simulation. One must provide an entrainment shapefile in `Inputs/ENT` and set the `simType` to `ent`, `entres` or `available` (see *Initialize release, entrainment and resistance areas*). In the entrainment areas defined by the entrainment shapefile, particles can entrain mass through erosion or plowing. In both mechanisms, one must account for three things:

- change of mass due to the entrainment
- change of momentum - entrained snow was accelerated from rest to the speed of the avalanche
- loss of momentum due to the plowing or erosion processes -entrained mass bounds with the ground needs to be broken

These three terms are further detailed in *Entrainment*. The parameters used to compute these processes can be set in the configuration file.

In the numerics, the mass is updated according to the entrainment model in `com1DFA.DFAfunctionsCython.computeEntMassAndForce()`. The velocity is updated immediately after using an implicit formulation.

Go back to *Algorithm graph*

6.3.4.5 Compute lateral pressure forces

The lateral pressure forces (F_{SPH}) are related to the gradient of the flow thickness (*Forces discretization*). This gradient is computed using a smoothed particle hydrodynamic method (*SPH gradient*).

Go back to *Algorithm graph*

6.3.5 Update position

Driving force, lateral pressure force and friction forces are subsequently used to update the velocity. Then the particle position is updated using a centered Euler scheme. These steps are done in `com1DFA.DFAfunctionsCython.updatePositionC()`.

6.3.5.1 Take gravity and lateral pressure forces into account

F_{drive} and F_{SPH} are summed up and taken into account to update the velocity. This is done via an explicit method.

6.3.5.2 Take friction into account

F_{fric} is taken into account to update the velocity. This is done via an implicit method.

6.3.5.3 Update particle position

The particles position is updated using the new velocity and a centered Euler scheme:

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + dt0.5(\mathbf{u}_{\text{old}} + \mathbf{u}_{\text{new}})$$

6.3.5.4 Take dam interaction into account

If the dam option is activated, the interaction between the particles and the dam is taken into account. During the computation of the DFA simulation, at each time step, if a particle enters one of the dam cells, the dam algorithm is called. The first step is to check if the particle crosses the dam during the time step. If not, the particle position and velocity are updated as if there was no dam. If yes, the intersection point between the particle trajectory and the dam line is computed. The dam properties are interpolated at the intersection point (dam tangent and normal vectors). The wall tangent and normal vectors are updated taking the flow thickness into account. The particle position and velocity are updated taking the dam into account.

Let $\mathbf{x}_{\text{foot}}$ and $\mathbf{x}_{\text{new}}$ be the intersection point with the dam and the new particle position vector if there was no dam. $\mathbf{u}_{\text{new}}$ is the new particle velocity with no dam. First the position $\mathbf{x}_{\text{b}}$ after elastic bouncing of the particle on the dam is computed:

$$\mathbf{x}_{\text{b}} = \mathbf{x}_{\text{new}} - 2\left\{\mathbf{n}_{\mathbf{w}}^{\text{filled}} \cdot (\mathbf{x}_{\text{new}} - \mathbf{x}_{\text{foot}})\right\}\mathbf{n}_{\mathbf{w}}^{\text{filled}}$$

Which gives the direction $\mathbf{e}_{\text{b}}$:

$$\mathbf{e}_{\text{b}} = \mathbf{x}_{\text{b}} - \mathbf{x}_{\text{foot}}$$

Next, the restitution coefficient is accounted for which leads to the new velocity:

$$\mathbf{x}_{\text{new}}^{\star} = \mathbf{x}_{\text{foot}} - \alpha_{\text{rest}}\mathbf{e}_{\text{b}} = \alpha_{\text{rest}}\mathbf{x}_{\text{b}} + (1 - \alpha_{\text{rest}})\mathbf{x}_{\text{foot}})$$

The velocity vector $\mathbf{u}_{\text{new}}^{\star}$ after the dam interaction reads:

$$\mathbf{u}_{\text{new}}^{\star} = (\mathbf{u}_{\text{new}} - 2\left\{\mathbf{n}_{\mathbf{w}}^{\text{filled}} \cdot \mathbf{u}_{\text{new}}\right\}\mathbf{n}_{\mathbf{w}}^{\text{filled}})\alpha_{\text{rest}}$$

Finally, the new velocity and position are re-projected onto the topography.

6.3.5.5 Correction step:

The particles z coordinate it readjusted so that the particles lie on the surface of the slope. There are two reasons why the particles might not lie on the surface anymore after updating their position according to the computed velocities:

- 1) because of the inaccuracy related to the time and space discretization. This can lead to a particle position being slightly above or under the surface. We want to correct this inaccuracy and therefore reproject the particle on the surface using its x and y coordinates.
- 2) because of the curvature of the slope and the particle velocity, particles can become detached from the ground in - in this case, the particle is located above the surface. In the current state, the com1DFA kernel does not allow this. If a particle becomes detached, the particle is also reprojected onto the surface using its x and y coordinates.

Similarly, the particles velocity is corrected in order to ensure that it lies in the tangent plane to the surface (the velocity vector magnitude is preserved, only the direction is changed).

The way the particles position is reprojected onto the surface does not allow both the velocity magnitude and the particle displacement to match perfectly. This is amplified by highly curved topographies or abrupt changes in slope.

Go back to *Algorithm graph*

6.3.6 Add secondary release area

If a secondary release area is provided, the flow thickness field from the previous time step is used to release a potential secondary release area. To do so, the flow thickness field is compared to the secondary release area rasters. If they overlap, the secondary release area is triggered and the secondary release particles are initialized and added to the flowing particles.

Go back to *Algorithm graph*

6.3.7 Update fields

This steps are done in `com1DFA.DFAfunctionsCython.updateFieldsC()`.

The mesh values are updated with the particles properties using *particles to mesh interpolation* methods. This is used to compute flow thickness, flow velocity and pressure fields from the particle properties.

6.3.7.1 Update particles flow thickness

The mesh flow thickness is finally used to update the particle flow thickness value using *mesh to particle interpolation* methods.

Go back to *Algorithm graph*

6.3.8 Simulation outputs

At the end of the simulation, the result fields are exported to .asc files in `com1DFA.com1DFA.exportFields()` and the information gathered in an info dictionary is used to create a markdown report (*`log2Report.generateReport.writeReport()`*). Further details on the available outputs can be found in *Output*.

Go back to *Algorithm graph*

6.4 com4FlowPy Theory

![Image](img/Motivation_2d.png)

Fig. 1: Definition of angles and geometric measures for the calculation of Z^δ, where s is the projected distance along the path and z(s) the corresponding altitude.

The model equations that determine the run out in three dimensional terrain are mainly motivated with respect to simple, geometric, two dimensional concepts [0,3,4] in conjunction with ideas existing algorithms for flow routing in three dimensional terrain [1,2], controlling the main routing and final stopping of the flow.

Figure 1 summarizes the basic concept of a constant run out angle (alpha) with the corresponding geometric relations in two dimensions along a possible process path.

![tan_alpha](img/tan_alpha.png)

The local travel angle gamma is defined by the altitude difference and projected distance along the path, from the release point to the current location.

![tan_gamma](img/tan_gamma.png)

The angle delta is the difference between the local travel angle gamma and the runout angle alpha and is related to Z^δ, so when delta equals zero or gamma equals alpha, the maximum runout distance is reached.

![z_alpha](img/z_alpha.png)

Z^α can be interpreted as dissipation energy.

![z_gamma](img/z_gamma.png)

Z^γ is the altitude difference between the starting point and the current calculation step at the projected distance s.

Z^δ is the difference between Z^γ and Z^α, so when Z^δ is lower or equal zero the stopping criterion is met and the flow stops. Z^δ is associated to the process magnitude and can be interpreted as the kinetic energy or velocity of the process.

![z_delta](img/z_delta.png)

The major drawback of implementing the geometric runout angle concepts is that they require a predefined flow path in two dimensional terrain. To allow for an enhanced routing in three dimensional terrain without prior knowledge of the flow path we combine these concepts [4] with extensions of existing algorithms [1,2,3] that are described in the following sections.

Spatial Input and Iterative Calculation Steps on the Path

In nature a GMF has one or more release areas that span over single or multiple release cells. Flow-Py computes the so called path, which is defined as the spatial extent of the routing from each release cell. Each release area (single raster cell in release area layer) has it's own unique path (collection of raster cells), and a location on the terrain (a single raster cell) can belong to many paths. Flow-Py identifies the path with spatial iterations starting with a release area raster cell and only iterating over cells which receive routing flux. The corresponding functions are implemented in the code in the flow_class.calc_distribution() function.

To route on the surface of the three dimensional terrain, operating on a quadrilateral grid, we implement the geometric concepts that have been sketched in the model motivation utilizing the following cell definitions:

![grid_overview](img/Neighbours.png)

Fig. 2: Definition of parent, base, child and neighbors, as well as the indexing around the base.

Each path calculation starts with a release cell and operates on the raster, requiring the definition of parent, base, child and neighbor cells (see Fig. 2). The base cell is the cell being calculated on the current spatial iteration step. The 8 raster cells surrounding the base cell are called neighbor cells (n, i) which have the potential to be parents (supplying flux to base cell), or a child (receive flux from the base cell). In 2d the base cell corresponds to the cell/location at the distance s along the path in Fig. 1.

Every base has at least one parent cell, except in the first calculation step from the release cell, where we start our calculation, this would be at s = s₀ in Fig. 1.

During an iteration step a raster cell from the iteration list is identified as the current base cell. The routing flux is calculated across the base cell from the parent cell to possible child cells. The goal is to keep the spatial iteration steps to a minimum, which is achieved by only adding neighbor cells to the iteration list that have flux routed to them from the base cell and do not meet either of the stopping conditions. These cells are called child cells. Child cells that are not already on the iteration list are added to the list and flow_class python object is created for the raster cell. The child cells flow_class has the parent added to it as a source for routing flux. By being added to the iteration list the cell has been recognized as being part of the GMF path and will be the base cell for a future iteration step.

When the iteration list is empty and all potential children fulfill one of the stopping criteria:

- Z^δ has to be smaller than zero: Z^δ < 0,
- Routing Flux has to be smaller than the flux cut off: R_i < R_{Stop},

the path calculation is finished. The required information is saved from the cell class to the summarizing output raster files. Then the calculation starts again for the next release cell and respective flow path. The spatial extent and magnitude for all release cells are summarized in the output raster files, which represent the overlay of all paths.

Every path is independent from the other, but depending on the information we want to extract, we save the highest values (e.g. Z^δ) or sums (e.g.Cell Counts) of different paths to the output raster file.

Z^δ

For each base cell in a path we solve the equations (6,7 and 8) for every neighbor n, if Z_{bn}^δ is higher than zero, this neighbor is defined as a potential child of this base, and routing in this direction is possible.

![z_delta_i](img/z_delta_array.png)

Here S_{bn} is the projected distance between the base and the neighbor.

As Z_{bn}^δ can be interpreted as process magnitude (and kinetic energy or velocity respectively) it is possible to limit this value to a maximum. In comparison to process based modeling approaches this would correspond to maximum velocity induced by a velocity dependent turbulent friction term.

![z_delta_max](img/z_delta_max.png)

The projected path lengths, or total travel distance to one of the neighbors (S_n) equals the path length to the base (S_b) plus the path from base to the neighbor (S_{bn}), which reads:

![S_bn](img/S_bn.png)

As there are many possibilities for the path from the starting point to the actual cell or base, the shortest path is taken into account, corresponding to the highest Z^δ in the base. If Z^δ_{max} is set to infinity, or as in the code to 8848 m (= Mount Everest), we can calculate the shortest path from the starting point to the base and yields the total projected travel distance:

![S_n_eq1](img/S_n_eq1.png)

This equations determine the routing and corresponding run out distance for the process, the next steps demonstrate how spreading is handled on the surface of the three dimensional terrain.

Persistence based routing

The persistence contribution P_i aims to reproduce the behavior of inertia, and takes the change in flow direction into account [3]. The direction contribution is scaled with the process magnitude Z^δ_{parent}, such that the direction from a parent cell with higher process magnitude has more effect on the path routing and direction.

The direction contributions D_n are defined by the cosine of the angle between parent, base and child/neighbor minus pi:

Therefore the direction contribution limits the maximum number of potential children to three, getting input via the persistence function from one parent.

In the first calculation step, at the release or start cell no parent cells are defined and the persistence is set to one. So the first calculation step is solely determined by the terrain contribution.

Terrain based routing

The terrain based routing is solely dependent on the slope angle phi. The exponent exp allows to control the divergence of the spreading. The Holmgren (1994) algorithm [1] is used in different kind of models and works well for avalanches but also rockfall or soil slides. For avalanches an exponent of 8 shows good results. To reach a single flow in step terrain (rockfall, soil slides, steepest descend), an exponent of 75 is considered.

Holmgrem Algorithm from 1994 [1]

To overcome the challenge of routing in flat or uphill terrain, we adapted the slope angle phi for the normalized terrain contribution to:

Routing Flux

The routing flux summarizes the persistence and terrain contributions according to Eq.(16):

where i is the direction and n are the neighbors from 1 to 8. R_i is then the routing flux in direction i. R_b is the flux in the base, for a release cell or starting cell the flux of the base equals one. The result of Eq. (16) is a 3 x 3 array with assigned flux values. A normalization stage is then required to bring the sum of the R_i's to the value of R_b. This aims at avoiding loss of flux [2].

Flow Chart / Overview

In Fig. 3 the algorithm of the computational implementation is sketched, including function and files names with respect to the code in the repository.

The file main.py handles the input for the computation and splits the release layer in tiles and saves them in a release list. Then the main.py starts one process per tile, which calls the flow_core.py and starts the calculation for one release cell and the corresponding path. The number of processes is depending on the hardware setting (CPU and RAM). Whenever a new cell is created flow_core.py calls flow_class.py and makes a new instance of this class, which is saved in the path. When the calculation in flow_core.py is finished it returns the path to main.py which saves the result to the output rasters.

Fig.3: Flow chart of the Flow-Py computational process and an overview of the files and what they manage.

References

[0] [Heim, A. (1932).]: Bergstürze und Menschenleben

[1] [Holmgren, P. (1994).](https://www.researchgate.net/publication/229484151_Multiple_flow_direction_algorithms_for_runoff_modelling_in_grid_based_elevation_models_An_empirical_evaluation) Multiple flow direction algorithms for runoff modelling in grid based elevation models: an empirical evaluation. Hydrological Processes, 8:327–334.

[2] [Horton, P., Jaboyedoff, M., Rudaz, B., and Zimmermann, M. (2013).](https://nhess.copernicus.org/articles/13/869/2013/nhess-13-869-2013.pdf) Flow-R, a model for susceptibility mapping of debris flows and other gravitational hazards at a regional scale. Natural Hazards and Earth System Science, 13:869–885.

[3] [Gamma, P. (1999).](https://www.researchgate.net/publication/34432465_dfwalk-Ein_Murgang-Simulationsprogramm_zur_Gefahrenzonierung) dfwalk - Ein Murgang-Simulationsprogramm zur Gefahrenzonierung. PhD thesis, Universität Bern.

[4] [Huber, A., Fischer, J. T., Kofler, A., and Kleemayr, K. (2016).] Using spatially distributed statistical models for avalanche runout estimation. In International Snow Science Workshop, Breckenridge, Colorado, USA - 2016.

Contact and acknowledgment

For Questions contact: Michael Neuhauser, Austrian Research Centre for Forest: Michael.Neuhauser@bfw.gv.at
Christopher D'Amboise, Austrian Research Centre for Forest: Christopher.DAmboise@bfw.gv.at

This study was carried out in the framework of the GreenRisk4Alps project ASP635, funded by the European Regional Development Fund through the Interreg Alpine Space programme. Additional financial support from the AvaRange (www.AvaRange.org, international cooperation project "AvaRange - Particle Tracking in Snow Avalanches" supported by the German Research Foundation (DFG) and the Austrian Science Fund (FWF, project number I 4274-N29) and the AvaFrame (www.AvaFrame.org, AvaFrame - The open Avalanche Framework is a cooperation between the Austrian Research Centre for Forests (Bundesforschungszentrum für Wald; BFW) and Austrian Avalanche and Torrent Service (Wildbach- und Lawinenverbauung; WLV) in conjunction with the Federal Ministry Republic of Austria: Agriculture, Regions and Tourism (BMLRT)) projects are greatly acknowledged.

Citation:

Michael Neuhauser, Christopher D'Amboise, Michaela Teich, Andreas Kofler, Andreas Huber, Reinhard Fromm, & Jan Thomas Fischer. (2021, June 24). Flow-Py: routing and stopping of gravitational mass flows (Version 1.0). Zenodo. http://doi.org/10.5281/zenodo.5027275

CHAPTER

SEVEN

REFERENCES

7.1 Data sources

Here is a list of external data we used, with links and sources if available:

- Release , entrainment and resistance geodata for the realistic test cases **Gar, Mal, Hit, Alr, Kot, Wog, Mal** was kindly provided by the Austrian Torrent and Avalanche Service (https://die-wildbach.at; via FZGL, Fachbereich Lawinen; Nov 2020).
- Topography data for for the realistic test cases **Gar, Mal, Hit, Alr, Kot, Wog, Mal** was obtained through *Open data Austria*

  Datenquelle: Land Tirol - data.tirol.gv.at

  https://www.data.gv.at/katalog/dataset/land-tirol_tirolgelnde

7.2 Glossary

Note: This is still under construction

Note: This glossary has the main purpose to introduce and consolidate an unambiguous language and definitions with respect to avalanche dynamics, avalanche simulations and modelling for the digital toolbox AvaFrame and particularly the Avalanche Modelling Atlas (AMA). Terms are organized in alphabetical order and the terminology builds on the UNESCO Avalanche atlas [DeQuervain+81] and relates to existing guidelines provided and used by Avalanche Associations and Warning Services (Canadian Avalanche Association [CAA16] , American Avalanche Association (AAA) [AmericanAAssociation16] or the European Avalanche Warning Services (EAWS)). The glossary has the goal to follow common conventions used in publications and related projects.

alpha point
alpha angle
: The alpha point is the point of the furthest reach of an avalanche (visible deposition or affected area). Thealpha angleis the angle between the line connecting the release point and the alpha point and the horizontally projected

travel length along the *thalweg* (see *travel length* and *runout angle*). The alpha angle is also termed theangle of reachor the*runout angle*.

beta point
beta angle
The boundary between the *transition* and *deposition* zone (start of deposition) is often called beta point and associated to a certain slope threshold of the *thalweg*. The corresponding *travel angle* is referred to as beta angle. For major avalanche *path* (s) that may produce avalanches of *size* 4-5, the beta point is associated to the 10° Point (or beta_10), i.e. the point where the slope angle of the *thalweg* decreases below 10°. For avalanche *path* (s) with *event* (s) of avalanche *size* 1-3 the beta point may accordingly be associated to larger *thalweg* slope angles (i.e. up to 30°, see start of *transition*).

coordinate transformation
Coordinate transformations refer to the operation of changing coordinates, e.g. between a fixed, Eularian, global coordinate system with east and north orientation, to an avalanche *path* dependent coordinate system along the *thalweg*, or even a Langrangian coordinate system, moving along particle trajectories.

cycle
An avalanche cycle describes a series of *event* (s) that occur across a region over a relatively short time span.

danger scale
The avalanche danger scale refers to the avalanche hazard and is an inherrent part of avalanche warning.

dense flow
Dense flow is a *form of movement* in the *transition* zone of the avalanche. Dense flow avalanches (DFA) are flowing along the ground and are rather associated to warm flow. Mixed types of movement are often observed, combining different flow regimes and their partial or complete transitions, e.g. 'mixed flow and powder avalanches' or 'flow avalanche with powder component', towards the evolution of a fluidized layer in the avalanche flow.

density
Release, *entrainment*, flow, or deposition density. Important quantity relating mass and volume, influencing impact pressure and particular friction relations.

deposition
see *zone of deposition*

depth
Release, *entrainment*, flow, or deposition depth refers to the extent of the avalanche measured in the direction of gravity.

entrainment
Entrainment describes the process of mass intake during the avalanche flow.

event
see *scenario*

flow variables
Flow variables include *thickness*, *velocity*, or *density* and are determined by the form of movement. Flow models that are implemented usually calculate the spatio-temporal evolution of these variables and where the maximum over the whole flow or computational duration, I.e., their peak values are the most used results. The flow variables are used to derive other variables such as impact pressure or kinetic energy of the flow.

form of movement
Is an avalanche criterion in the zone of *transition* and has *powder snow* or *dense flow* as characteristics.

manner of starting
Is an avalanche criterion in the zone of *origin* and has the possible characteristics loose, slab, or gliding.

origin
see *zone of origin*

path
: The avalanche path summarizes the total catchment and is divided into different zones (zone of *origin*, *transition*, *deposition*) with different criteria and characteristics. An inherent property of the avalanche path is the *thalweg* and the associated avalanche *event*.

powder snow
: Powder snow avalanches (PSA) refer to the *form of movement* in the zone of *transition*, referring to the dust or suspension cloud in avalanches. PSA are associated with cold, dry cohesionless snow. Mixed types of movement are often observed, combining different flow regimes and their transitions, e.g., 'powder avalanche with flow component'.

projection
: The projection refers to the projection of coordinates within a coordinate system, I.e. the projection of the *runout point* (as furthest reach of the avalalanche) to the *thalweg*. Or projecting a 3D travel length (xyz) to a 2D travel length measured only along xy.

release area
: Potential release areas are located in the zone of *origin*. Each documented *event* or simulation *scenario* is associated to one or more primary and/or secondary release areas, that can further be described by the *manner of starting*.

return period
: Return periods are related to return levels describing the *size* or magnitude of design or recorded *event* (s) on a respective avalanche *path*. The return level is often determined by the run out length of historically documented avalanche *event* (s) accompanied with return period estimates, which are associated to the occurrence probability.

runout area
runout angle
runout length
runout point
: Runout lengths and angles are intricately linked to the *alpha point*, utilizing the *projection* to the *thalweg*. In the same manner as *travel length*, run out lengths are measured as horizontally projected lengths along the *thalweg*, from the uppermost point of the *release area* to furthest reach of the runout area. The runout may refer to visible deposition (associated to dense flow), damages or the impacted and affected area (associated to air blast or *powder snow*) in the zone of *deposition* and is usually defined via flow *thickness*, velocity, kinetic energy or impact pressure thresholds.

scenario
: One or multiple avalanche events or corresponding simulation scenarios are associated to a certain avalanche *path* and have distinct criteria and characteristics such as avalanche *size*, *release area* (s), or *runout area*. These properties are morphologically connected to the different zones (*origin*, *transition*, *deposition*) of an avalanche *path* and allow to define other associated properties, such as *alpha angle* or *runout length* that are defined in combination with the avalanche *thalweg*. Besides observed and documented avalanche events, design events of particular *return period* (s) are of particular interest for engineering applications.

size
: Avalanche size refers to the magnitude or intensity of an *event*, classified by destructive potential, *runout length* and dimension according to the EAWS size classification, which is closely related to the CAA destructive size.

terrain classification
: Terrain may be classified according to the Avalanche Terrain Exposure Scale (ATES) into simple (low angle or primarily forested terrain with some openings that may involve the *deposition* zone of infrequent avalanche *path* (s)), challenging (well defined avalanche *path* (s), starting zones, or terrain traps), and complex (exposure to multiple overlapping avalanche *path* (s), large expanses of steep, open terrain, multiple starting zones, and terrain traps below).

thalweg
: The thalweg is defined by the main flow direction of an avalanche *path* of one or multiple, i.e. not regarding

a specific *event* or *scenario*. avalanche *event* (s). Technically it is the two-dimensional terrain representation, displaying the terrain altitude along the horizontally projected *travel length*.

thickness
Release, *entrainment*, flow, or deposition thickness refers to the extent (distance) of the avalanche measured perpendicular to the slope.

trajectory length
Used in com1DFA particle dictionaries, where the trajectory length is computed as the distance traveled by a particle from one time step to the next and then accumulated over time. Three different trajectory lengths are computed (1) trajectoryLengthXY - computed in the x, y plane, (2) trajectoryLengthXYZ - also taking the slope of the topography into account and (3) trajectoryLengthXYCor - same as trajectoryLengthXY but corrected for the potential angle difference of the slope and the normal.

transition
see *zone of transition*

travel angle
travel length
Travel lengths are measured as horizontally projected travel length (s_{XY}) along the *thalweg* and are associated with the corresponding travel angle, measured between the line connecting the current location with the uppermost point of the release and the horizontal plane. Alternatively, the surface parallel travel length (s_{XYZ}) may be defined as the three-dimensional length travelled by the avalanche.

velocity
Flow velocities are usually measured in a surface parallel direction. Alternatively approach velocities are measured along the line of sight.

wet snow
A wet snow avalanche (WSA) implies the presence of liquid water within the avalanche and is usually associated to *dense flow* type of movement in the *transition* zone of the avalanche.

zone of deposition
The zone of deposition is where the *runout area* of the avalanche is located and where the avalanche stops due to frictional energy dissipation. The boundary with the *transition* zone (start of deposition) is often called *beta point*.

zone of origin
The zone of origin delineates the area, in which typical *release area* (s) are located, and an avalanche's appearance is characterized by the *manner of starting*. The uppermost possible point is referred to as start of origin.

zone of transition
The zone of transition is the area between zone of *origin* and zone of *deposition* along the *thalweg*. The *form of movement* is linked to the *flow variables*. The start of transition links the zone of *origin* and transition and is usually associated with a slope inclination of about 28-30°.

BIBLIOGRAPHY

[AS05] R. Ata and A. Soulaïmani. A stabilized sph method for inviscid shallow water flows. *International Journal for Numerical Methods in Fluids*, 47:139–159, 2005.

[BDL83] S. Bakkehøi, U. Domaas, and K. Lied. Calculation of snow avalanche runout distance. *Annals of Glaciology*, 4:24–29, 1983.

[BSG99] P. Bartelt, B. Salm, and U. Gruber. Calculating dense-snow avalanche runout using a voellmy-fluid model with active/passive longitudinal straining. *Journal of Glaciology*, 45(150):242–254, 1999.

[CAA16] Canadian Avalanche Association CAA. Observation guidelines and recording standards for weather snowpack and avalanches. Technical Report, Canadian Avalanche Association, Revelstoke, BC, Canada, 2016.

[FM13] Gloria Faccanoni and Anne Mangeney. Exact solution for granular flows. *International Journal for Numerical and Analytical Methods in Geomechanics*, 37(10):1408–1433, 2013.

[FFGS13] J. T. Fischer, R. Fromm, P. Gauer, and B. Sovilla. Evaluation of probabilistic snow avalanche simulation ensembles with doppler radar observations. *Cold Regions Science and Technology*, 2013. doi:10.1016/j.coldregions.2013.09.011.

[Fis13] J.-T. Fischer. A novel approach to evaluate and compare computational snow avalanche simulation. *Natural Hazards and Earth System Science*, 13(6):1655–1667, 2013. URL: http://www.nat-hazards-earth-syst-sci.net/13/1655/2013/, doi:10.5194/nhess-13-1655-2013.

[FK13] J.-T. Fischer and A. Kofler. Samosat cosica. Technical Report, Federal Research and Training Centre for Forests, Natural Hazards and Landscape, Innsbruck, Austria, 2013.

[HSSN93] Columban Hutter, M. Siegel, Stuart Savage, and Y. Nohguchi. Two-dimensional spreading of a granular avalanche down an inclined plane part i. theory. *Acta Mechanica*, 100:37–68, 01 1993. doi:10.1007/BF01176861.

[HBB19] David M Hyman, Andrea Bevilacqua, and Marcus I Bursik. Statistical theory of probabilistic hazard maps: a probability distribution for the hazard boundary location. *Natural Hazards and Earth System Sciences*, 19(7):1347–1363, 2019.

[IOS+14] Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. Sph fluids in computer graphics. In Sylvain Lefebvre and Michela Spagnuolo, editors, *Eurographics 2014 - State of the Art Reports*. The Eurographics Association, 2014. doi:10.2312/egst.20141034.

[KMS18] A Köhler, JN McElwaine, and B Sovilla. Geodar data and the flow regimes of snow avalanches. *Journal of geophysical research: earth surface*, 123(6):1272–1294, 2018.

[LB80] K. Lied and K. Bakkehøi. Empirical calculations of snow–avalanche run–out distance based on topographic parameters. *Journal of Glaciology*, 26(94):165–177, 1980. doi:10.3189/S0022143000010704.

[LL10] M. Liu and G.R. Liu. Smoothed particle hydrodynamics (sph): an overview and recent developments. *Archives of Computational Methods in Engineering*, 17:25–76, 03 2010. doi:10.1007/s11831-010-9040-7.

[MCVB+03] Anne Mangeney-Castelnau, Jean-Pierre Vilotte, Marie-Odile Bristeau, Benoit Perthame, François Bouchut, Chiara Simeoni, and Sudhakar Yerneni. Numerical modeling of avalanches based on saint venant equations using a kinetic scheme. *Journal of Geophysical Research: Solid Earth*, 108:2527–2544, 11 2003. doi:10.1029/2002JB002024.

[Mon92] J.J. Monaghan. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30:543–574, 1992.

[RK20] Matthias Rauter and Anselm Köhler. Constraints on entrainment and deposition models in avalanche simulations from high-resolution radar data. *Geosciences*, 2020. URL: https://www.mdpi.com/2076-3263/10/1/9, doi:10.3390/geosciences10010009.

[SFF+08] R. Sailer, W. Fellin, R. Fromm, P. Jörg, L. Rammer, P. Sampl, and A. Schaffhauser. Snow avalanche mass-balance calculation and simulation-model verification. *Annals of Glaciology*, 48(1):183–192, 2008.

[Sal04] B. Salm. A short and personal history of snow avalanche dynamics. *Cold Regions Science and Technology*, 39(2-3):83–92, 2004.

[Sam07] P. Sampl. Samosat modelltheorie und numerik. Technical Report, AVL List GMBH, 2007.

[SG09] P. Sampl and M. Granig. Avalanche simulation with samos-at. In *Proceedings of the International Snow Science Workshop, Davos*. 2009.

[SH89] S. B. Savage and K. Hutter. The motion of a finite mass of granular material down a rough incline. *Journal of Fluid Mechanics*, 199(1):177–215, 1989.

[Voe55] A. Voellmy. Über die zerstörungskraft von lawinen. *Schweizerische Bauzeitung*, Sonderdruck aus dem 73. Jahrgang(12, 15, 17, 19 und 37):1–25, 1955.

[Wag16] P.M. Wagner. Kalibrierung des --modells für das ermitteln der auslauflänge von kleinen und mittleren lawinen. Master's thesis, Institut für Alpine Naturgefahren (IAN), BOKU-Universität für Bodenkultur, 2016.

[WHP04] Y. Wang, K. Hutter, and S.P. Pudasaini. The savage-hutter theory: a system of partial differential equations for avalanche flows of snow, debris, and mud. *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, 84(8):507–527, 2004.

[Zwi00] T. Zwinger. *Dynamik einer Trockenschneelawine auf beliebig geformten Berghangen*. PhD Thesis, Technischen Universitaet Wien, 2000.

[ZKS03] T. Zwinger, A. Kluwick, and P. Sampl. Numerical simulation of dry-snow avalanche flow over natural terrain. *Dynamic Response of Granular and Porous Materials under Large and Catastrophic Deformations, Hutter, K. and Kirchner, N., Springer Verlag*, 11:161–194, 2003.

[AmericanAAssociation16] AAA American Avalanche Association. Snow, weather and avalanches: observation guidelines for avalanche programs in the united states. Technical Report, American Avalanche Association, 2016.

[DeQuervain+81] R De Quervain and others. *Avalanche atlas*. Unesco, Paris, 1981. URL: http://unesdoc.unesco.org/images/0004/000480/048004MB.pdf.

PYTHON MODULE INDEX

INDEX

A

B

C

D

E

F

G

N

O

P

Q

R

S

T

U

V

W

Z