ANUGA

Release 3.1.9

Stephen Roberts, Ole Nielsen, Gareth Davies

Jun 26, 2022

CONTENTS:

ANUGA (pronounced “AHnooGAH”) is open-source software for the simulation of the shallow water equation, in particular it can be used to model tsunamis and floods.

ANUGA is a python 3 package with some C and Cython extensions (and an optional fortran extension).

ANUGA is developed at Geoscience Australia, Mathematical Sciences Institute at the Australian National University and volunteers.

Althought ANUGA was created in a collaboration by Geoscience Australia and Mathematical Sciences Institute at the Australian National University, it is now developed and maintained by a community of volunteers.

Note: This project is under active development.

CHAPTER

ONE

BACKGROUND

Modelling the effects on the built environment of natural hazards such as riverine flooding, storm surges and tsunami is critical for understanding their economic and social impact on our urban communities. Geoscience Australia and the Australian National University are developing a hydrodynamic inundation modelling tool called ANUGA to help simulate the impact of these hazards.

The core of ANUGA is the fluid dynamics object, called `anuga.Domain`, which is based on a finite-volume method for solving the Shallow Water Wave Equation. The study area is represented by a mesh of triangular cells. By solving the governing equation within each cell, water depth and horizontal momentum are tracked over time.

A major capability of ANUGA is that it can model the process of wetting and drying as water enters and leaves an area. This means that it is suitable for simulating water flow onto a beach or dry land and around structures such as buildings. ANUGA is also capable of modelling hydraulic jumps due to the ability of the finite-volume method to accommodate discontinuities in the solution and the bed (using the latest algorithms

To set up a particular scenario the user specifies the geometry (bathymetry and topography), the initial water level (stage), boundary conditions such as tide, and any operators that may drive the system such as rainfall, abstraction of water, erosion, culverts See section *Operators* for details of operators available in ANUGA.

The built-in mesh generator, called `graphical_mesh_generator`, allows the user to set up the geometry of the problem interactively and to identify boundary segments and regions using symbolic tags. These tags may then be used to set the actual boundary conditions and attributes for different regions (e.g. the Manning friction coefficient) for each simulation.

Most ANUGA components are written in the object-oriented programming language Python. Software written in Python can be produced quickly and can be readily adapted to changing requirements throughout its lifetime. Computationally intensive components are written for efficiency in `C` routines working directly with Python `numpy` structures.

The visualisation tool developed for ANUGA is based on OpenSceneGraph, an Open Source Software (OSS) component allowing high level interaction with sophisticated graphics primitives. See cite{nielsen2005} for more background on ANUGA.

CHAPTER

TWO

INSTALLATION

Contents

2.1 Introduction

ANUGA is a python package with some C extensions (and an optional fortran extension). This version of ANUGA is run and tested using python 3.7 - 3.9

2.2 Dependencies

ANUGA requires python 3.X (X>6) and the following python packages:

```
numpy scipy matplotlib pytest cython netcdf4 dill future gdal \
pyproj pymetis triangle Pmw mpi4py pytz ipython  meshpy Pmw pymetis utm
```

ANUGA is developed on Ubuntu and so we recommend Ubuntu as your production environment (though ANUGA can be installed on MacOS and Windows using *Miniconda* or *MiniForge*)

2.3 Ubuntu Install with MiniForge3

A clean way to install the dependencies for ANUGA is to use Anaconda, or Miniconda Python distributions by Continuum Analytics.

Using a *conda* installation has the advantage of allowing you to create multiple python environments and is particularly useful if you want to keep multiple versions of ANUGA

Indeed the most stable install is via the *conda-forge* channel which is easily available using the Miniforge. In particular the installation of the *gdal* and *mpi4py* modules are more stable.

These conda environments do not require administrative rights to your computer and do not interfere with the Python installed in your system.

Install the latest version of *Miniforge* from https://github.com/conda-forge/miniforge or use, for instance, *wget* to download the latest version via:

```
wget -O Miniforge3.sh "https://github.com/conda-forge/miniforge/releases/latest/download/
↪Miniforge3-$(uname)-$(uname -m).sh"
bash Miniforge3.sh
```

If you don't have *wget* you can install it via:

```
sudo apt-get update -q
sudo apt-get install wget git
```

Once *Miniforge* is installed, we can now create an environment to run `anuga'.

Create a conda environment *anuga_env* (or what ever name you like):

```
conda update conda
conda create -n anuga_env python=3.8 anuga mpi4py
conda activate anuga_env
```

Note we have also installed *mpi4py* to allow anuga to run in parallel. On some systems you may need to manually install *mpi4py* to match the version of *mpi* you are using.

This has setup a *conda* environment for *anuga* using python 3.8. (*anuga* has be tested on 3.7, 3.8. 3.9.)

We are now ready to use `anuga'.

You can test your installation via:

```
python -c "import anuga; anuga.test()"
```

2.4 Ubuntu Install with MiniForge3 and pip

Once you have a python environment it is also possible to install *anuga* via *pip*:

```
pip install anuga
```

You might need to run this command twice to push *pip* to install all the dependencies. And indeed you will need to install *gdal* and *mpi4py* manually.

You can test your installation via:

```
python -c "import anuga; anuga.test()"
```

2.5 Ubuntu Install with MiniForge3 from github

Alternatively you can the most current version of *anuga* ` from GitHub

```
git clone https://github.com/anuga-community/anuga_core.git
cd anuga_core
pip install -e .
python runtests.py
```

Remember, to use ANUGA you will have to activate the *anuga_env* environment via the command:

```
conda activate anuga_env`
```

You might even like to set this up in your *.bashrc* file.

2.6 Installing GDAl on Ubuntu using apt and pip

ANUGA can be installed using *pip*, but a complication arise when installing the *gdal* package.

First set up a python virtual environment and activate via:

```
python3 -m venv anuga_env
course anuga_env/bin/activate
```

Now we first need to install the *gdal* python package. First install the gdal library, via:

```
sudo apt-get install -y gdal-bin libgdal-dev
```

We need to ascertain the version of *gdal* installed using the following command:

```
ogrinfo --version
```

THe version of *gdal* to install via *pip* should match the version of the library. For instance on Ubuntu 20.04 the previous command produces:

```
GDAL 3.0.4, released 2020/01/28
```

So in this case we install the *gdal* python package as follows

```
pip install gdal==3.0.4
```

Now we complete the installation of *anuga* simply by:

```
pip install anuga
```

If you obtain errors from *pip* regarding "not installing dependencies", it seems that that can be fixed by just running the *pip install anuga* again

2.7 Installing on Ubuntu using apt and pip

You can install the *anuga* dependencies via a combination of the standard ubuntu `apt` method and python pip install.

From your home directory run the following commands which will download anuga to a directory *anuga_core*, install dependencies, install anuga and run the unit tests:

```
git clone https://github.com/anuga-community/anuga_core.git
sudo bash anuga_core/tools/install_ubuntu_20_04.sh
```

Note: Part of the bash shell will run as sudo so will ask for a password. If you like you can run the package installs manually, run the commands in the script `anuga_core/tools/install_ubuntu_20._04.sh`.

This script also creates a python3 virtual environment *anuga_env*. You should activate this virtual environment when working with *anuga*, via the command:

```
source ~/anuga_env/bin/activate
```

You might like to add this command to your *.bashrc* file to automatically activate this python environment.

2.7.1 Updating

From time to time you might like to update your version of anuga to the latest version on github. You can do this by going to the *anuga_core* directory and *pulling* the latest version and then reinstalling via the following commands:

```
cd anuga_core
git pull
pip install -e .
```

And finally check the newinstallation by running the unit tests via: .. code-block:: bash

python runtests.py -n

2.8 Windows 10 Install using 'Ubuntu on Windows'

Starting from Windows 10, it is possible to run an Ubuntu Bash console from Windows. This can greatly simplify the install for Windows users. You'll still need administrator access though. First install an ubuntu 20_04 subsystem. Then just use your preferred ubuntu install described above.

2.9 Windows Installation using MiniForge

We have installed *anuga* on *windows* using miniforge.

You can download MiniForge manually from the MiniForge site https://github.com/conda-forge/miniforge:

Alternatively you can download and install miniforge via CLI commands:

Run the following powershell instruction to download miniforge.

```
Start-FileDownload "https://github.com/conda-forge/miniforge/releases/latest/download/
↪Miniforge3-Windows-x86_64.exe" C:\Miniforge.exe; echo "Finished downloading miniforge"
```

From a standard *cmd* prompt then install miniconda via:

```
C:\Miniconda.exe /S /D=C:\Py
C:\Py\Scripts\activate.bat
```

Install conda-forge packages:

```
conda create -n anuga_env python=3.8  anuga mpi4py
conda activate anuga_env
```

You can test your installation via:

```
python -c "import anuga; anuga.test()"
```

CHAPTER

THREE

EXAMPLES

3.1 Simple Script Example

Here we discuss the structure and operation of a script called `runup.py` (which is available in the `examples` directory of `anuga_core`.

This example carries out the solution of the shallow-water wave equation in the simple case of a configuration comprising a flat bed, sloping at a fixed angle in one direction and having a constant depth across each line in the perpendicular direction.

The example demonstrates the basic ideas involved in setting up a complex scenario. In general the user specifies the geometry (bathymetry and topography), the initial water level, boundary conditions such as tide, and any forcing terms that may drive the system such as rainfall, abstraction of water, wind stress or atmospheric pressure gradients. Frictional resistance from the different terrains in the model is represented by predefined forcing terms. In this example, the boundary is reflective on three sides and a time dependent wave on one side.

The present example represents a simple scenario and does not include any forcing terms, nor is the data taken from a file as it would typically be.

The conserved quantities involved in the problem are stage (absolute height of water surface), x-momentum and y-momentum. Other quantities involved in the computation are the friction and elevation.

Water depth can be obtained through the equation:

```
depth = stage - elevation
```

3.1.1 Outline of the Program

In outline, `runup.py` performs the following steps:

- Sets up a triangular mesh.
- Sets certain parameters governing the mode of operation of the model, specifying, for instance, where to store the model output.
- Inputs various quantities describing physical measurements, such as the elevation, to be specified at each mesh point (vertex).
- Sets up the boundary conditions.
- Carries out the evolution of the model through a series of time steps and outputs the results, providing a results file that can be viewed.

3.1.2 The Code

For reference we include below the complete code listing for `runup.py`. Subsequent paragraphs provide a 'commentary' that describes each step of the program and explains it significance.

```
"""Simple water flow example using ANUGA

Water driven up a linear slope and time varying boundary,
similar to a beach environment
"""

#------------------------------------------------------------------------------
# Import necessary modules
#------------------------------------------------------------------------------
import anuga

from math import sin, pi, exp

#------------------------------------------------------------------------------
# Setup computational domain
#------------------------------------------------------------------------------
domain = anuga.rectangular_cross_domain(10, 10)     # Create domain

#------------------------------------------------------------------------------
# Setup initial conditions
#------------------------------------------------------------------------------
def topography(x, y):
    return -x/2                             # linear bed slope
    #return x*(-(2.0-x)*.5)                 # curved bed slope

domain.set_quantity('elevation', topography) # Use function for elevation
domain.set_quantity('friction', 0.1)         # Constant friction
domain.set_quantity('stage', -0.4)           # Constant negative initial stage

#------------------------------------------------------------------------------
# Setup boundary conditions
#------------------------------------------------------------------------------
Br = anuga.Reflective_boundary(domain)      # Solid reflective wall
Bw = anuga.Time_boundary(domain=domain,     # Time dependent boundary
                   function=lambda t: [(0.1*sin(t*2*pi)-0.3)*exp(-t), 0.0, 0.0])

# Associate boundary tags with boundary objects
domain.set_boundary({'left': Br, 'right': Bw, 'top': Br, 'bottom': Br})

#------------------------------------------------------------------------------
# Evolve system through time
#------------------------------------------------------------------------------
for t in domain.evolve(yieldstep=0.1, finaltime=10.0):
    print (domain.timestepping_statistics())
```

3.1.3 Establishing the Domain

The very first thing to do is import the various modules, of which the anuga module is the most important:

```
import anuga
```

Then we need to set up the triangular mesh to be used for the scenario. This is carried out through the statement:

```
domain = anuga.rectangular_cross_domain(10, 5, len1=10.0, len2=5.0)
```

The above assignment sets up a 10×5 rectangular mesh, triangulated in a regular way with boundary tags `left`, `right`, `top` or `bottom`.

It is also possible to set up a domain from "first principles" using `points`, `vertices` and `boundary` via the assignment:

```
points, vertices, boundary = anuga.rectangular_cross(10, 5, len1=10.0, len2=5.0)
domain = anuga.Domain(points, vertices, boundary)
```

where:

- `points` is a list giving the coordinates of each mesh point,
- `vertices` is a list specifying the three vertices of each triangle, and
- `boundary` is a dictionary that stores the edges on the boundary and associates with each a symbolic tag. The edges are represented as pairs (i, j) where i refers to the triangle id and j to the edge id of that triangle. Edge ids are enumerated from 0 to 2 based on the id of the vertex opposite.

This creates an instance of the `Domain` class, which represents the domain of the simulation. Specific options are set at this point, including the basename for the output file and the directory to be used for data:

```
domain.set_name('runup')
domain.set_datadir('.')
```

In addition, the following statement could be used to state that quantities `stage`, `xmomentum` and `ymomentum`` are to be stored at every timestep and `elevation` only once at the beginning of the simulation:

```
domain.set_quantities_to_be_stored({'stage': 2, 'xmomentum': 2, 'ymomentum': 2,
↪'elevation': 1})
```

However, this is not necessary, as the above is the default behaviour.

3.1.4 Initial Conditions

The next task is to specify a number of quantities that we wish to set for each mesh point. The class `{Domain` has a method `set_quantity`, used to specify these quantities. It is a flexible method that allows the user to set quantities in a variety of ways – using constants, functions, numeric arrays, expressions involving other quantities, or arbitrary data points with associated values, all of which can be passed as arguments. All quantities can be initialised using `set_quantity`. For a conserved quantity (such as `stage, xmomentum, ymomentum`) this is called an *initial condition*. However, other quantities that aren't updated by the evolution procedure are also assigned values using the same interface. The code in the present example demonstrates a number of forms in which we can invoke `set_quantity`.

Elevation

The elevation, or height of the bed, is set using a function defined through the statements below, which is specific to this example and specifies a particularly simple initial configuration for demonstration purposes:

```
def topography(x, y):
    return -x/2
```

This simply associates an elevation with each point `(x, y)` of the plane. It specifies that the bed slopes linearly in the `x` direction, with slope $-\frac{1}{2}$, and is constant in the `y` direction.

Once the function `topography`` is specified, the quantity `elevation` is assigned through the statement:

```
domain.set_quantity('elevation', topography)
```

NOTE: If using function to set `elevation` it must be vector compatible. For example, using square root will not work.

Friction

The assignment of the friction quantity (a forcing term) demonstrates another way we can use `set_quantity` to set quantities – namely, assign them to a constant numerical value:

```
domain.set_quantity('friction', 0.1)
```

This specifies that the Manning friction coefficient is set to 0.1 at every mesh point.

Stage

The stage (the height of the water surface) is related to the elevation and the depth at any time by the equation:

```
stage = elevation + depth
```

For this example, we simply assign a constant value to `stage`, using the statement:

```
domain.set_quantity('stage', -0.4)
```

which specifies that the surface level is set to a height of -0.4, i.e. 0.4 units (metres) below the zero level.

Although it is not necessary for this example, it may be useful to digress here and mention a variant to this requirement, which allows us to illustrate another way to use `set_quantity` – namely, incorporating an expression involving other quantities. Suppose, instead of setting a constant value for the stage, we wished to specify a constant value for the *depth*. For such a case we need to specify that `stage` is everywhere obtained by adding that value to the value already specified for `elevation`. We would do this by means of the statements:

```
h = 0.05    # Constant depth
domain.set_quantity('stage', expression='elevation + %f' % h)
```

That is, the value of `stage` is set to `h = 0.05` plus the value of `elevation` already defined.

The reader will probably appreciate that this capability to incorporate expressions into statements using `set_quantity` greatly expands its power.

3.1.5 Boundary Conditions

The boundary conditions are specified as follows:

```
Br = anuga.Reflective_boundary(domain)
Bw = anuga.Time_boundary(domain=domain, f=lambda t: [(0.1*sin(t*2*pi)-0.3)*exp(-t), 0.0,␣
↪0.0])
```

The effect of these statements is to set up a selection of different alternative boundary conditions and store them in variables that can be assigned as needed. Each boundary condition specifies the behaviour at a boundary in terms of the behaviour in neighbouring elements. The boundary conditions introduced here may be briefly described as follows:

- Reflective boundary: Returns same `stage` as in its neighbour volume but momentum vector reversed 180 degrees (reflected). Specific to the shallow water equation as it works with the momentum quantities assumed to be the second and third conserved quantities. A reflective boundary condition models a solid wall.
- Time boundary: Set a boundary varying with time.

Before describing how these boundary conditions are assigned, we recall that a mesh is specified using three variables `points`, `vertices` and `boundary`. In the code we are discussing, these three variables are returned by the function `rectangular_cross`. The example given in Section ref{sec:realdataexample} illustrates another way of assigning the values, by means of the function `create_domain_from_regions`.

These variables store the data determining the mesh as follows. (You may find that the example given in Section ref{sec:meshexample} helps to clarify the following discussion, even though that example is a *non-rectangular* mesh.):

- `points`` stores a list of 2-tuples giving the coordinates of the mesh points.
- `vertices` stores a list of 3-tuples of numbers, representing vertices of triangles in the mesh. In this list, the triangle whose vertices are `points[i]}, :code:`points[j]`, `points[k]` is represented by the 3-tuple `(i, j, k)`.
- The variable `boundary` is a Python dictionary that not only stores the edges that make up the boundary but also assigns symbolic tags to these edges to distinguish different parts of the boundary. An edge with endpoints `points[i]` and `points[j]` is represented by the 2-tuple `(i, j)`. The keys for the dictionary are the 2-tuples `(i, j)` corresponding to boundary edges in the mesh, and the values are the tags are used to label them. In the present example, the value `boundary[(i, j)]` assigned to `(i, j)]` is one of the four tags `left`, `right`, `top` or `bottom`, depending on whether the boundary edge represented by `(i, j)` occurs at the left, right, top or bottom of the rectangle bounding the mesh. The function `rectangular_cross` automatically assigns these tags to the boundary edges when it generates the mesh.

The tags provide the means to assign different boundary conditions to an edge depending on which part of the boundary it belongs to. (In Section Real Example we describe an example that uses different boundary tags – in general, the possible tags are entirely selectable by the user when generating the mesh and not limited to 'left', 'right', 'top' and 'bottom' as in this example.) All segments in bounding polygon must be tagged. If a tag is not supplied, the default tag name `exterior` will be assigned by ANUGA.

Using the boundary objects described above, we assign a boundary condition to each part of the boundary by means of a statement like:

```
domain.set_boundary({'left': Br, 'right': Bw, 'top': Br, 'bottom': Br})
```

It is critical that all tags are associated with a boundary condition in this statement. If not the program will halt with a statement like:

```
Traceback (most recent call last):
File "mesh_test.py", line 114, in ?
    domain.set_boundary({'west': Bi, 'east': Bo, 'north': Br, 'south': Br})
```

(continues on next page)

(continued from previous page)

```
File "X:\inundation\sandpits\onielsen\anuga_core\source\anuga\
    abstract_2d_finite_volumes\domain.py", line 505, in set_boundary
    raise msg
ERROR (domain.py): Tag "exterior" has not been bound to a boundary object.
All boundary tags defined in domain must appear in the supplied dictionary.
The tags are: ['ocean', 'east', 'north', 'exterior', 'south']
```

The command `set_boundary` stipulates that, in the current example, the right boundary varies with time, as defined by the lambda function, while the other boundaries are all reflective.

3.1.6 Evolution

The final statement:

```
for t in domain.evolve(yieldstep=0.1, duration=10.0):
    print domain.timestepping_statistics()
```

causes `domain` we have just setup to *evolve*, over a series of steps indicated by the values of `yieldstep` and `duration`, which can be altered as required (an alternative to `duration` is `finaltime`) The value of `yieldstep` provides the time interval between successive yields to the evolve loop. Behind the scenes more *inner* time steps are generally taken.

By default, the current state of the evolution is stored a each yield step.

Time between output can also be controlled by the argument `outputstep` which needs to an integer multiple of the `yieldstep`

3.1.7 Output

The output is a NetCDF file with the extension `.sww`. It contains stage and momentum information and can be used with the ANUGA viewer `anuga_viewer` to generate a visual display.

3.1.8 Exploring the Model Output

The following figures are screenshots from the anuga viewer visualisation tool `anuga_viewer`.

The first figure shows the domain with water surface as specified by the initial condition, $t = 0$.

The second figure shows the flow at time $t = 2.3$ and the last figure show the flow at time $t = 4$ where the system has been evolved and the wave is encroaching on the previously dry bed.

Online documentation is available for the `anuga_viewer`

3.2 Simple Notebook Example

Here we introduce the idea of creating a domain which contains the mesh and quantities needed to run the simulation, and encapsulates the methods for setting up the initial conditions, the boundary conditions and the method for evolving the solution.

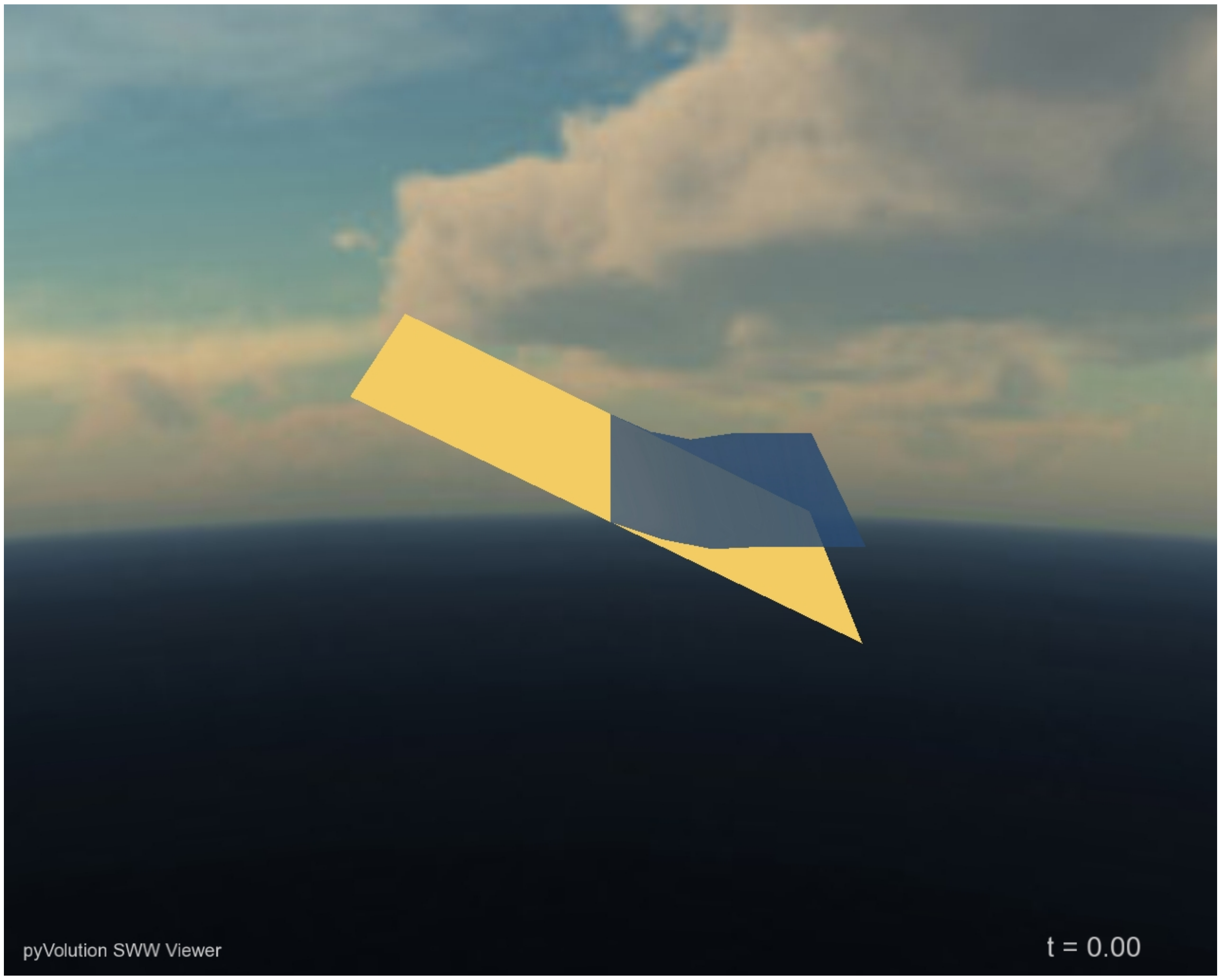


Fig. 1: Runup example viewed at time 0.0 with the ANUGA viewer

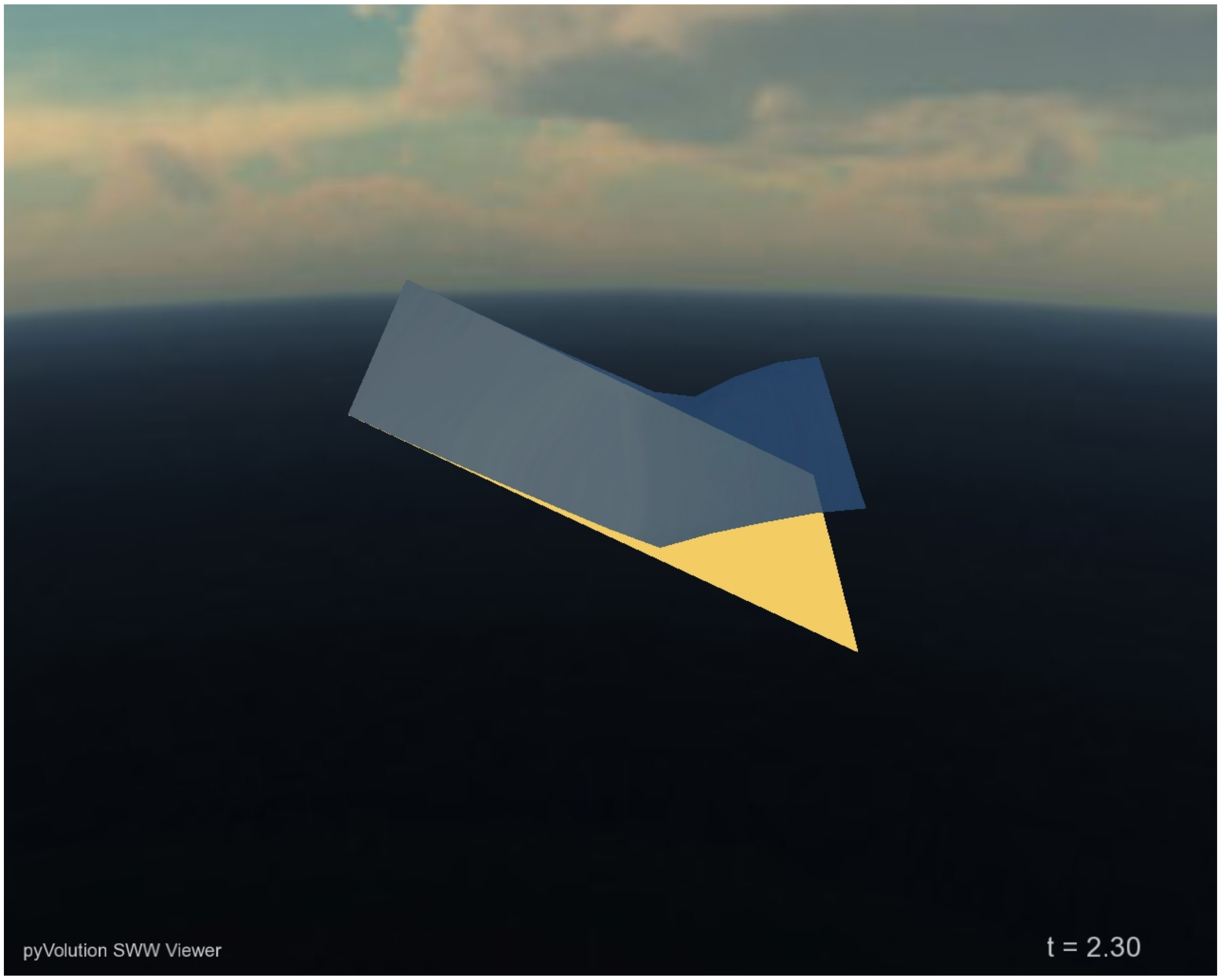


Fig. 2: Runup example viewed at time 2.3 with the ANUGA viewer

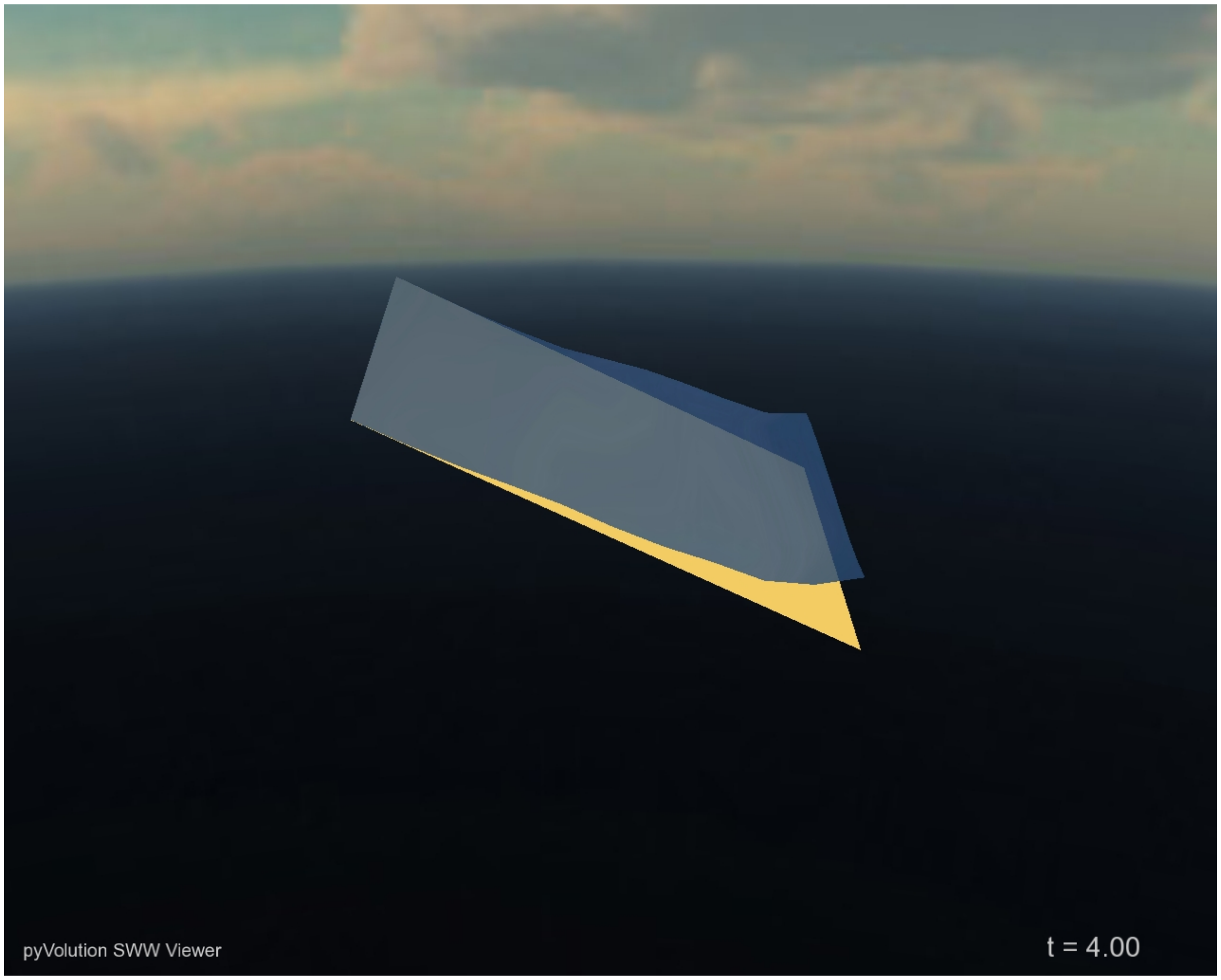


Fig. 3: Runup example viewed time 4.0 with the ANUGA viewer

3.2.1 Setup Notebook for Visualisation and Animation

We are using the format of a jupyter notebook. As such we need to setup inline matplotlib plotting and animation.

[10]:
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

# Allow inline jshtml animations
from matplotlib import rc
rc('animation', html='jshtml')
```

3.2.2 Import ANUGA

We assume that anuga has been installed. If so we can import anuga.

[11]:
```
import anuga
```

3.2.3 Create an ANUGA domain

A Domain is the core object which contains the mesh and the quantities for the particular problem. Here we create a simple rectangular Domain. We set the name to `domain1` which will be used when storing the simulation output to a sww file called `domain1.sww`.

[12]:
```
domain1 = anuga.rectangular_cross_domain(40, 20, len1=20.0, len2=10.0)

domain1.set_name('domain1')
domain1.set_store_vertices_smoothly(False)
```

3.2.4 Plot Mesh

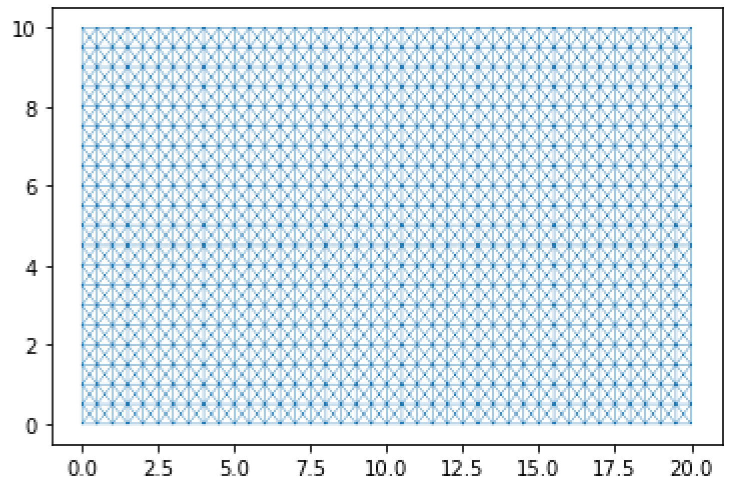
Let's look at the mesh. We will use some code derived form the clawpack project to simplify plotting and animation of the output from our simulations. This is available via the animate module loaded from anuga.

The `Domain_plotter` class provides a plotting wrapper around our standard anuga `Domain`, providing simple access to the centroid values of our evolution quantities, `stage`, `depth`, `elev`, `xmon` and `ymon` and the triangulation `triang`.

Note: This visualisation is recommended for smaller domains (maybe up to 10,000 triangles). We have an `anuga-viewer` for larger domains.

[13]:
```
dplotter1 = anuga.Domain_plotter(domain1)
plt.triplot(dplotter1.triang, linewidth = 0.4);
```

```
Figure files for each frame will be stored in _plot
```



3.2.5 Setup Initial Conditions

We have to setup the values of various quantities associated with the domain. In particular we need to setup the elevation the elevation of the bed or the bathymetry. In this case we will do this using a function.

[14]:
```
def topography(x, y):

    z = -x/10

    N = len(x)

    minx = np.floor(np.max(x)/4)
    wallx1 = np.min(x[(x >= minx)])
    wallx2 = np.min(x[(x > wallx1 + 0.25)])

    minx = np.floor(np.max(x)/2)
    wallx3 = np.min(x[(x >= minx)])
    wallx4 = np.min(x[(x > wallx3 + 0.25)])

    minx = np.floor(3*np.max(x)/4)
    wallx5 = np.min(x[(x >= minx)])
    wallx6 = np.min(x[(x > wallx5 + 0.25)])

    dist = 0.4 * (np.max(y) - np.min(y))

    for i in range(N):
        if wallx1 <= x[i] <= wallx2:
            if (y[i] < dist):
                z[i] += 1

        if wallx3 <= x[i] <= wallx4:
            if (y[i] > np.max(y) - dist):
                z[i] += 1

        if wallx5 <= x[i] <= wallx6:
```

(continues on next page)

(continued from previous page)

```
            if (y[i] < dist):
                z[i] += 1

    return z
```

3.2.6 Set Quantities

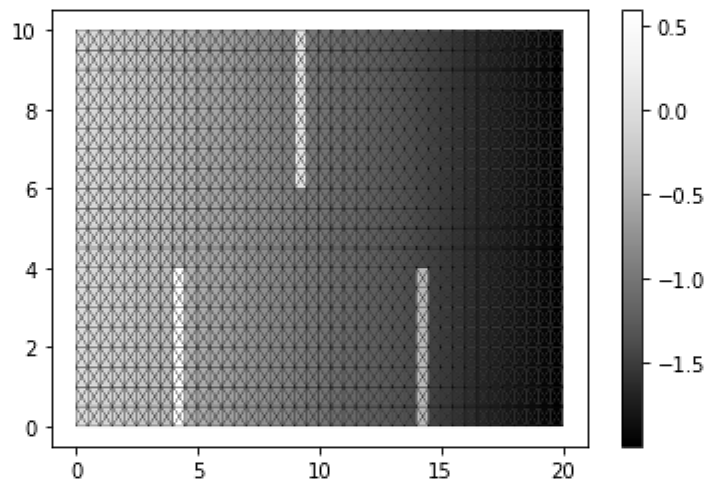
Now we set the elevation, stage and friction using the domain.set_quantity function.

[15]:
```
domain1.set_quantity('elevation', topography, location='centroids')         # Use␣
↪function for elevation
domain1.set_quantity('friction', 0.01, location='centroids')                # Constant␣
↪friction
domain1.set_quantity('stage', expression='elevation', location='centroids') # Dry Bed
```

3.2.7 View Elevation

Let's use the matplotlib function tripcolor to plot the elevation quantity. We access the domain mesh and elevation quantitiy via the dplotter interface.

[16]:
```
plt.tripcolor(dplotter1.triang,
              facecolors = dplotter1.elev,
              edgecolors='k',
              cmap='Greys_r')
plt.colorbar();
```



Notice that we have been very careful to match up the defintion of the topography via the function topography with the resolution of the mesh.

3.2.8 Setup Boundary Conditions

The rectangular domain has 4 tagged boundaries, left, top, right and bottom. We need to set boundary conditons for each of these tagged boundaries. We can set `Dirichlet_boundary` type BC with specified values of stage, and x and y "momentum". Another common BC is `Reflective_boundary` which mimics a wall.

[17]:
```
Bi = anuga.Dirichlet_boundary([0.4, 0, 0])         # Inflow
Bo = anuga.Dirichlet_boundary([-2, 0, 0])          # Outflow
Br = anuga.Reflective_boundary(domain1)            # Solid reflective wall

domain1.set_boundary({'left': Bi, 'right': Bo, 'top': Br, 'bottom': Br})
```

3.2.9 Run the Evolution

We evolve using a for statement, which evolves the quantities using the ANUGA shallow water wave solver. The calculation yields every `yieldstep` seconds, for a given `duration` (or until a specified `finaltime`).

[18]:
```
for t in domain1.evolve(yieldstep=2, duration=40):

    #dplotter.plot_depth_frame()
    dplotter1.save_depth_frame(vmin=0.0,vmax=1.0)

    domain1.print_timestepping_statistics()


# Read in the png files stored during the evolve loop
dplotter1.make_depth_animation()
```

```
Time = 0.0000 (sec), steps=0 (18s)
Time = 2.0000 (sec), delta t in [0.01779464, 0.03749219] (s), steps=93 (0s)
Time = 4.0000 (sec), delta t in [0.01523410, 0.01780455] (s), steps=123 (0s)
Time = 6.0000 (sec), delta t in [0.01509139, 0.01543878] (s), steps=132 (0s)
Time = 8.0000 (sec), delta t in [0.01543945, 0.01589701] (s), steps=129 (0s)
Time = 10.0000 (sec), delta t in [0.01510457, 0.01595656] (s), steps=129 (0s)
Time = 12.0000 (sec), delta t in [0.01448747, 0.01510270] (s), steps=136 (0s)
Time = 14.0000 (sec), delta t in [0.01416889, 0.01448641] (s), steps=140 (0s)
Time = 16.0000 (sec), delta t in [0.01390842, 0.01416679] (s), steps=143 (0s)
Time = 18.0000 (sec), delta t in [0.01381293, 0.01390783] (s), steps=145 (0s)
Time = 20.0000 (sec), delta t in [0.01356459, 0.01381284] (s), steps=147 (0s)
Time = 22.0000 (sec), delta t in [0.01337491, 0.01356424] (s), steps=149 (0s)
Time = 24.0000 (sec), delta t in [0.01312175, 0.01337337] (s), steps=152 (0s)
Time = 26.0000 (sec), delta t in [0.01302523, 0.01317617] (s), steps=153 (0s)
Time = 28.0000 (sec), delta t in [0.01288636, 0.01302421] (s), steps=155 (0s)
Time = 30.0000 (sec), delta t in [0.01274763, 0.01288612] (s), steps=156 (0s)
Time = 32.0000 (sec), delta t in [0.01265408, 0.01274647] (s), steps=158 (0s)
Time = 34.0000 (sec), delta t in [0.01260016, 0.01266082] (s), steps=159 (0s)
Time = 36.0000 (sec), delta t in [0.01259445, 0.01261115] (s), steps=159 (0s)
Time = 38.0000 (sec), delta t in [0.01257706, 0.01260399] (s), steps=159 (0s)
Time = 40.0000 (sec), delta t in [0.01254139, 0.01258247] (s), steps=160 (0s)
```

[18]:
```
<matplotlib.animation.FuncAnimation at 0x7f3b6d9ea670>
```

[ ]:

3.3 Simple Example using Create from Regions

To take advantage of the ability of triangular meshes to match complex geometries, we need to be able to create a domain with a complicated boundary. As such in this notebook we investigate the use of the procedure `create_domain_from_regions`. We then set up the initial conditions, the boundary conditions and the method for evolving the solution.

3.3.1 Setup Notebook for Visualisation and Animation

We are using the format of a jupyter notebook. As such we need to setup inline matplotlib plotting and animation.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

# Allow inline jshtml animations
from matplotlib import rc
rc('animation', html='jshtml')
```

3.3.2 Import ANUGA

We assume that anuga has been installed. If so we can import anuga.

[2]:
```
import anuga
```

3.3.3 Create an ANUGA domain with create_domain_from_regions

ANUGA is based on triangles and so the mesh can conform to interesting geometrical structures. In our example the steps define an interesting geometry. Let's conform our mesh to the steps.

We will use the construction function `anuga.create_domain_from_regions`. This function needs at least a polygon which defines the boundary of the region, and a tagging of the sections of the boundry polygon, which will allow us to specify specific boundary conditions associated with the tagged sections of the boundary.

In our previous example the function `rectangular_cross_domain` created a mesh with 4 tagged boundary sections, corresponding to the tags `left`, `right`, `top` and `bottom`.

We wil do the same, but this time using the function `anuga.create_domain_from_regions`.

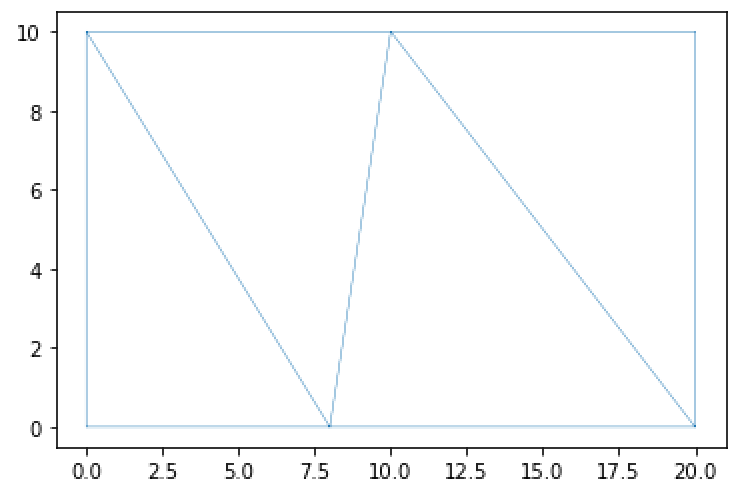
[3]:
```
bounding_polygon = [[0.0, 0.0],
                    [20.0, 0.0],
                    [20.0, 10.0],
                    [0.0, 10.0]]

boundary_tags={'bottom': [0],
                'right': [1],
                'top': [2],
                'left': [3]}
```

(continues on next page)

(continued from previous page)

```
domain2 = anuga.create_domain_from_regions(bounding_polygon, boundary_tags)

# Plot the resulting mesh
dplotter2 = anuga.Domain_plotter(domain2)
plt.triplot(dplotter2.triang, linewidth = 0.4);
```

```
Figure files for each frame will be stored in _plot
```

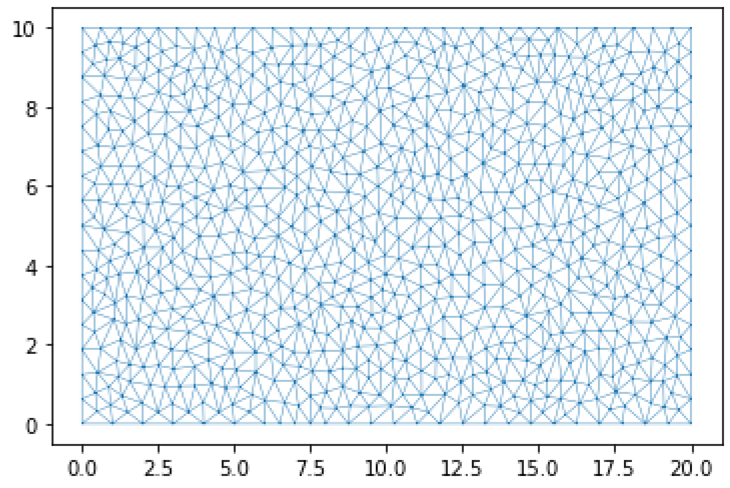


3.3.4 Mesh size

Obviously the mesh is too coarse. We can force the mesh size to be smaller by using the argument maximum_triangle_size.

[4]:
```
domain2 = anuga.create_domain_from_regions(bounding_polygon,
                               boundary_tags,
                               maximum_triangle_area = 0.2,
                               )

# Plot the resulting mesh
dplotter2 = anuga.Domain_plotter(domain2)
plt.triplot(dplotter2.triang, linewidth = 0.4);
```

```
Figure files for each frame will be stored in _plot
```



3.3.5 More Complicated Boundary

In the first example we created the steps using a discontinuous elevation. We can mimic that behaviour by explicitly cutting out the triangles associated with the steps. This leads to a more complicated boundary polygon.

Note that we need to be careful about associating boundary polygon sections with the approriate tagged boundary.

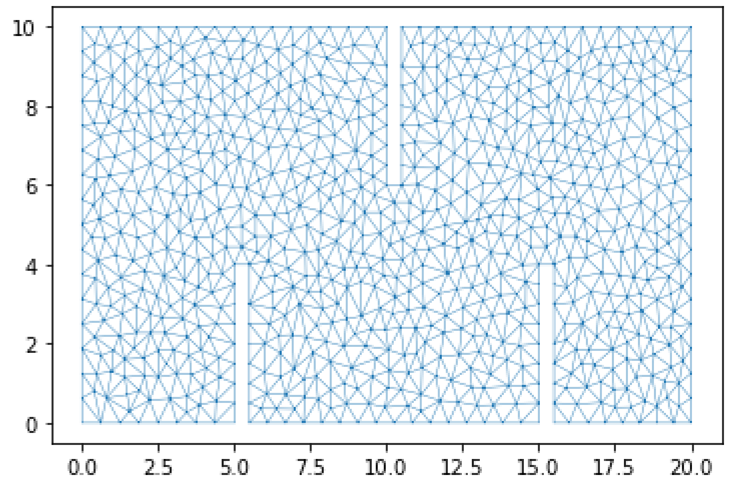
We now have 7 tagged bounday regions. These 7 regions will need to be associated with appropriate boundary conditions.

[5]:
```
bounding_polygon = [[0.0, 0.0],
                    [5.0, 0.0], [5.0, 4.0], [5.5, 4.0], [5.5, 0.0],
                    [15.0, 0.0], [15.0, 4.0], [15.5, 4.0], [15.5, 0.0],
                    [20.0, 0.0],
                    [20.0, 10.0],
                    [10.5, 10.0], [10.5, 6.0], [10, 6.0], [10, 10.0],
                    [0.0, 10.0]]

boundary_tags={'bottom': [0,4,8],
               'right': [9],
               'top': [10,14],
               'left': [15],
               'wall1': [1,2,3],
               'wall2': [5,6,7],
               'wall3': [11,12,13]
              }


domain2 = anuga.create_domain_from_regions(bounding_polygon,
                                           boundary_tags,
                                           maximum_triangle_area = 0.2,)

domain2.set_name('domain2')
domain2.set_store_vertices_smoothly(False)

# Plot the resulting mesh
```
(continues on next page)

(continued from previous page)

```
dplotter2 = anuga.Domain_plotter(domain2)
plt.triplot(dplotter2.triang, linewidth = 0.4);
```

```
Figure files for each frame will be stored in _plot
```



3.3.6 Initial Conditions and Boundary Conditions

As before we setup the inital values for our elevation, friction and stage. And associated Dirichlet BC on the left and right boundary regions and reflective everywhere else.

Notice that in this case the definition of `elevation` is very simple.

[6]:
```
#Initial Conditions
domain2.set_quantity('elevation', lambda x,y : -x/10, location='centroids')  # Use␣
↪function for elevation
domain2.set_quantity('friction', 0.01, location='centroids')                # Constant␣
↪friction
domain2.set_quantity('stage', expression='elevation', location='centroids') # Dry Bed

# Boundary Conditions
Bi = anuga.Dirichlet_boundary([0.4, 0, 0])          # Inflow
Bo = anuga.Dirichlet_boundary([-2, 0, 0])           # Inflow
Br = anuga.Reflective_boundary(domain2)             # Solid reflective wall

domain2.set_boundary({'left': Bi, 'right': Bo, 'top': Br, 'bottom': Br, 'wall1': Br,
↪'wall2': Br, 'wall3': Br})
```

3.3.7 Evolve

Now we can evolve. With this implementation the step walls are infinitely high and so we will not get a flow over the top of 2nd lower step.

[7]:
```
for t in domain2.evolve(yieldstep=2, duration=40):

    #dplotter.plot_depth_frame()
    dplotter2.save_depth_frame(vmin=0.0, vmax=1.0)

    domain2.print_timestepping_statistics()


# Read in the png files stored during the evolve loop
dplotter2.make_depth_animation()
```

```
Time = 0.0000 (sec), steps=0 (7s)
Time = 2.0000 (sec), delta t in [0.01905513, 0.04796679] (s), steps=87 (0s)
Time = 4.0000 (sec), delta t in [0.01744624, 0.01945916] (s), steps=109 (0s)
Time = 6.0000 (sec), delta t in [0.01382385, 0.01743857] (s), steps=134 (0s)
Time = 8.0000 (sec), delta t in [0.01615668, 0.01741694] (s), steps=121 (0s)
Time = 10.0000 (sec), delta t in [0.01595034, 0.01723757] (s), steps=123 (0s)
Time = 12.0000 (sec), delta t in [0.01510446, 0.01647925] (s), steps=128 (0s)
Time = 14.0000 (sec), delta t in [0.01386254, 0.01509676] (s), steps=137 (0s)
Time = 16.0000 (sec), delta t in [0.01301787, 0.01385327] (s), steps=151 (0s)
Time = 18.0000 (sec), delta t in [0.01246696, 0.01301680] (s), steps=158 (0s)
Time = 20.0000 (sec), delta t in [0.01228814, 0.01246448] (s), steps=162 (0s)
Time = 22.0000 (sec), delta t in [0.01209191, 0.01229133] (s), steps=165 (0s)
Time = 24.0000 (sec), delta t in [0.01206389, 0.01210745] (s), steps=166 (0s)
Time = 26.0000 (sec), delta t in [0.01202287, 0.01212743] (s), steps=166 (0s)
Time = 28.0000 (sec), delta t in [0.01195973, 0.01202261] (s), steps=167 (0s)
Time = 30.0000 (sec), delta t in [0.01194640, 0.01205201] (s), steps=167 (0s)
Time = 32.0000 (sec), delta t in [0.01180743, 0.01194597] (s), steps=169 (0s)
Time = 34.0000 (sec), delta t in [0.01177425, 0.01180725] (s), steps=170 (0s)
Time = 36.0000 (sec), delta t in [0.01173895, 0.01177404] (s), steps=171 (0s)
Time = 38.0000 (sec), delta t in [0.01172513, 0.01173895] (s), steps=171 (0s)
Time = 40.0000 (sec), delta t in [0.01171070, 0.01172510] (s), steps=171 (0s)
```

[7]:
```
<matplotlib.animation.FuncAnimation at 0x7f94a72c0df0>
```

3.4 Example of Creating Domains with River Walls

An alternative method to simulate walls (or levees) is to use `riverWalls`. Think of `riverWalls` as infinitely thin walls. To set these up we need to build our mesh with `breaklines` to define where the wall will occur and also how to apply them during the evolution by setting up a `riverWall` operator.

First setup the mesh.

We setup a dictionary to contain the x,y,z information of each of the river walls in our simulation. In this case 3 river walls associated with wall1 to wall3.

Look carefully at the mesh produced and notice the straight lines in the mesh at the location of the walls.

3.4.1 Setup Notebook for Visualisation and Animation

We are using the format of a jupyter notebook. As such we need to setup inline matplotlib plotting and animation.

[13]:
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

# Allow inline jshtml animations
from matplotlib import rc
rc('animation', html='jshtml')
```

3.4.2 Import ANUGA

We assume that anuga has been installed. If so we can import anuga.

[14]:
```
import anuga
```

3.4.3 Create an ANUGA domain with create_domain_from_regions

ANUGA is based on triangles and so the mesh can conform to interesting geometrical structures. In our example the steps define an interesting geometry. Let's conform our mesh to the steps.

We will use the construction function `anuga.create_domain_from_regions`. This function needs at least a polygon which defines the boundary of the region, and a tagging of the sections of the boundry polygon, which will allow us to specify specific boundary conditions associated with the tagged sections of the boundary.

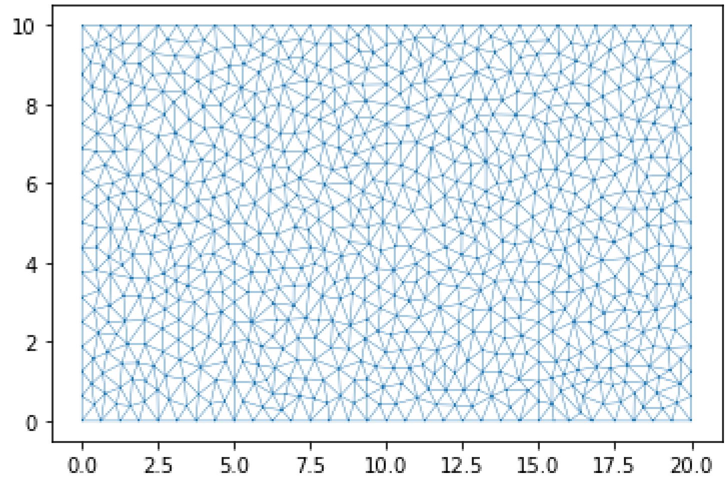
We wil do this using the function `anuga.create_domain_from_regions`. In addition we aline the mesh with our `riverwalls` which will represent the position of our three walls.

[15]:
```
bounding_polygon = [[0.0, 0.0],
                    [20.0, 0.0],
                    [20.0, 10.0],
                    [0.0, 10.0]]

boundary_tags={'bottom': [0],
               'right': [1],
               'top': [2],
               'left': [3]
              }


riverWalls = { 'wall1': [[5.0,0.0,   0.5], [5.0,4.0,  0.5]],
               'wall2': [[15.0,0.0, -0.5], [15.0,4.0,-0.5]],
               'wall3': [[10.0,10.0, 0.0], [10.0,6.0, 0.0]]
             }

#bline = [[[0.1,5.0,0.0],[19.9,5.0,0.0]]]

domain3 = anuga.create_domain_from_regions(bounding_polygon,
                                           boundary_tags,
```
(continues on next page)

(continued from previous page)

```
                                          maximum_triangle_area = 0.2,
                                          breaklines = riverWalls.values())

domain3.set_name('domain3')
domain3.set_store_vertices_smoothly(False)

# Plot the resulting Mesh
dplotter3 = anuga.Domain_plotter(domain3)
plt.triplot(dplotter3.triang, linewidth = 0.4);
```

```
Figure files for each frame will be stored in _plot
```



Note: Look closely at the mesh and you will see three straight lines in the mesh generated by the breaklines. THis use of breakline can be very useful to build structures into the mesh (such as valley floors, buildings, and of course in this case riverwalls (or levees)).

3.4.4 Initial and Boundary Conditions and River walls

[16]:
```
#Initial Conditions
domain3.set_quantity('elevation', lambda x,y : -x/10, location='centroids') # Use␣
↪function for elevation
domain3.set_quantity('friction', 0.01, location='centroids')                # Constant␣
↪friction
domain3.set_quantity('stage', expression='elevation', location='centroids') # Dry Bed


# Boundary Conditions
Bi = anuga.Dirichlet_boundary([0.4, 0, 0])         # Inflow
Bo = anuga.Dirichlet_boundary([-2, 0, 0])          # Inflow
Br = anuga.Reflective_boundary(domain2)            # Solid reflective wall

domain3.set_boundary({'left': Bi, 'right': Bo, 'top': Br, 'bottom': Br})

# Setup RiverWall
domain3.riverwallData.create_riverwalls(riverWalls, verbose=False)
```

3.4.5 Evolve

Notice that we have setup the river walls to be only 1 metre high. So we would expect some overtopping of the 2nd lower step.

[17]:
```
for t in domain3.evolve(yieldstep=2, duration=40):

    #dplotter.plot_depth_frame()
    dplotter3.save_depth_frame(vmin=0.0, vmax=1.0)

    domain3.print_timestepping_statistics()


# Read in the png files stored during the evolve loop
dplotter3.make_depth_animation()
```

```
Time = 0.0000 (sec), steps=0 (5s)
Time = 2.0000 (sec), delta t in [0.01813369, 0.04322357] (s), steps=86 (0s)
Time = 4.0000 (sec), delta t in [0.01584979, 0.01883231] (s), steps=118 (0s)
Time = 6.0000 (sec), delta t in [0.01583185, 0.01738707] (s), steps=121 (0s)
Time = 8.0000 (sec), delta t in [0.01625105, 0.01686416] (s), steps=122 (0s)
Time = 10.0000 (sec), delta t in [0.01662976, 0.01887900] (s), steps=111 (0s)
Time = 12.0000 (sec), delta t in [0.01762579, 0.01855303] (s), steps=112 (0s)
Time = 14.0000 (sec), delta t in [0.01705865, 0.01766205] (s), steps=116 (0s)
Time = 16.0000 (sec), delta t in [0.01673190, 0.01716414] (s), steps=118 (0s)
Time = 18.0000 (sec), delta t in [0.01581448, 0.01672762] (s), steps=124 (0s)
Time = 20.0000 (sec), delta t in [0.01558134, 0.01581339] (s), steps=128 (0s)
Time = 22.0000 (sec), delta t in [0.01517773, 0.01559634] (s), steps=131 (0s)
Time = 24.0000 (sec), delta t in [0.01509483, 0.01517755] (s), steps=133 (0s)
Time = 26.0000 (sec), delta t in [0.01510280, 0.01513743] (s), steps=133 (0s)
Time = 28.0000 (sec), delta t in [0.01495640, 0.01513585] (s), steps=133 (0s)
Time = 30.0000 (sec), delta t in [0.01495971, 0.01499768] (s), steps=134 (0s)
Time = 32.0000 (sec), delta t in [0.01485341, 0.01497427] (s), steps=135 (0s)
Time = 34.0000 (sec), delta t in [0.01483037, 0.01486285] (s), steps=135 (0s)
Time = 36.0000 (sec), delta t in [0.01484865, 0.01489936] (s), steps=135 (0s)
Time = 38.0000 (sec), delta t in [0.01477187, 0.01487359] (s), steps=135 (0s)
Time = 40.0000 (sec), delta t in [0.01476090, 0.01477261] (s), steps=136 (0s)
```

[17]:
```
<matplotlib.animation.FuncAnimation at 0x7f94a750cee0>
```

3.5 Merewether Flood Case Study Example

Here we look at a case study of a flood in the community of Merewether near Newcastle NSW. We will add a flow using an `Inlet_operator` and extract flow details at various points by interagating the `sww` file which is produced during the ANUGA run.

This example is based on the the validation test merewether, provided in the ANUGA distribution.

3.5.1 Setup Notebook for Visualisation and Animation

We are using the format of a jupyter notebook. As such we need to setup inline matplotlib plotting and animation.

[41]:
```
import numpy as np
import os
import matplotlib.pyplot as plt

%matplotlib inline

# Allow inline jshtml animations
from matplotlib import rc
rc('animation', html='jshtml')
```

3.5.2 Import ANUGA

We assume that anuga has been installed. If so we can import anuga.

[42]:
```
import anuga
```

3.5.3 Read in Data

We have included some topography data and extent data in our anuga-clinic notebook repository.

Let's read that in and create a mesh associated with it.

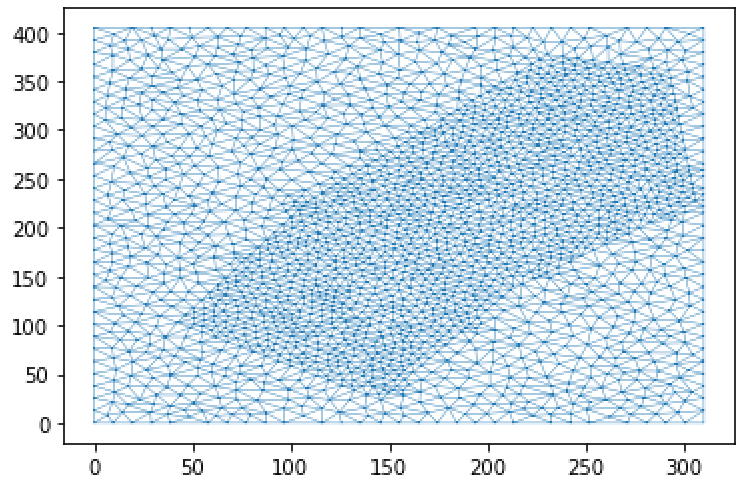
[43]:
```
data_dir = '/home/anuga/anuga-clinic/data/merewether'

# Polygon defining broad area of interest
bounding_polygon = anuga.read_polygon(os.path.join(data_dir,'extent.csv'))


# Polygon defining particular area of interest
merewether_polygon = anuga.read_polygon(os.path.join(data_dir,'merewether.csv'))


# Elevation Data
topography_file = os.path.join(data_dir,'topography1.asc')


# Resolution for most of the mesh
base_resolution = 80.0  # m^2

# Resolution in particular area of interest
merewether_resolution = 25.0 # m^2

interior_regions = [[merewether_polygon, merewether_resolution]]
```

3.5.4 Create and View Domain

Note that we use a `base_resolution` to ensure a reasonable refinement over the whole region, and we use `interior_regions` to refine the mesh in the area of interest. In this case we pass a list of `polygon, resolution` pairs.

[44]:
```
domain = anuga.create_domain_from_regions(
            bounding_polygon,
            boundary_tags={'south': [0],
                           'east':  [1],
                           'north':    [2],
                           'west':   [3]},
            maximum_triangle_area=base_resolution,
            interior_regions=interior_regions)


domain.set_name('merewether1') # Name of sww file
dplotter = anuga.Domain_plotter(domain)
plt.triplot(dplotter.triang, linewidth = 0.4);
```

```
Figure files for each frame will be stored in _plot
```



3.5.5 Setup Initial Conditions

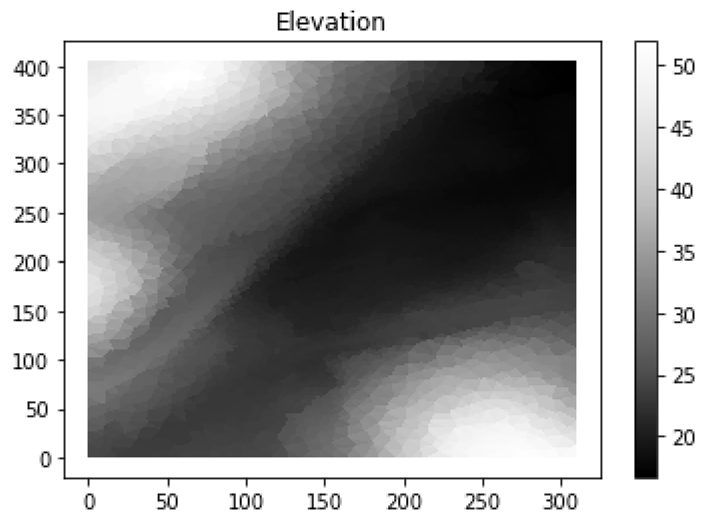
We have to setup the values of various quantities associated with the domain. In particular we need to setup the `elevation` the elevation of the bed or the bathymetry. In this case we will do this using the DEM file `topography1.asc` .

[45]:
```
domain.set_quantity('elevation', filename=topography_file, location='centroids') # Use␣
↪function for elevation
domain.set_quantity('friction', 0.01, location='centroids')                      #␣
↪Constant friction
domain.set_quantity('stage', expression='elevation', location='centroids')       # Dry␣
↪Bed

plt.tripcolor(dplotter.triang,
              facecolors = dplotter.elev,
```

(continues on next page)

(continued from previous page)

```
               cmap='Greys_r')
plt.colorbar();
plt.title("Elevation");
```



3.5.6 Setup Boundary Conditions

The rectangular domain has 4 tagged boundaries, left, top, right and bottom. We need to set boundary conditons for each of these tagged boundaries. We can set Transmissive type BC on the outflow boundaries and reflective on the others.

[46]:
```
Br = anuga.Reflective_boundary(domain)
Bt = anuga.Transmissive_boundary(domain)

domain.set_boundary({'south':   Br,
                     'east':    Bt, # outflow
                     'north':   Bt, # outflow
                     'west':    Br})
```

3.5.7 Setup Inflow

We need some water to flow. The easiest way to input a specified amount of water is via an `Inlet_operator` where we can specify a discharge `Q`.

[47]:
```
# Setup inlet flow
center = (382270.0, 6354285.0)
radius = 10.0
region0 = anuga.Region(domain, center=center, radius=radius)
fixed_inflow = anuga.Inlet_operator(domain, region0 , Q=19.7)
```

3.5.8 Run the Evolution

We evolve using a `for` statement, which evolves the quantities using the shallow water wave solver. The calculation `yields` every `yieldstep` seconds, up to a given `duration`.

[48]:
```
for t in domain.evolve(yieldstep=20, duration=300):

    #dplotter.plot_depth_frame()
    dplotter.save_depth_frame(vmin=0.0, vmax=1.0)

    domain.print_timestepping_statistics()


# Read in the png files stored during the evolve loop
dplotter.make_depth_animation()
```

```
Time = 0.0000 (sec), steps=0 (16s)
Time = 20.0000 (sec), delta t = 1000.00000000 (s), steps=1 (0s)
Time = 40.0000 (sec), delta t in [0.31097778, 0.54228760] (s), steps=57 (0s)
Time = 60.0000 (sec), delta t in [0.19738976, 0.34302832] (s), steps=78 (0s)
Time = 80.0000 (sec), delta t in [0.19746572, 0.20781137] (s), steps=99 (0s)
Time = 100.0000 (sec), delta t in [0.19296721, 0.21299123] (s), steps=99 (0s)
Time = 120.0000 (sec), delta t in [0.18074822, 0.19291818] (s), steps=107 (0s)
Time = 140.0000 (sec), delta t in [0.16806655, 0.18068880] (s), steps=116 (0s)
Time = 160.0000 (sec), delta t in [0.15436803, 0.16804974] (s), steps=123 (0s)
Time = 180.0000 (sec), delta t in [0.15154953, 0.15428113] (s), steps=132 (0s)
Time = 200.0000 (sec), delta t in [0.15069124, 0.15217552] (s), steps=133 (0s)
Time = 220.0000 (sec), delta t in [0.14955219, 0.15068963] (s), steps=134 (0s)
Time = 240.0000 (sec), delta t in [0.14931204, 0.14955927] (s), steps=134 (0s)
Time = 260.0000 (sec), delta t in [0.14956369, 0.14977760] (s), steps=134 (0s)
Time = 280.0000 (sec), delta t in [0.14947921, 0.14972423] (s), steps=134 (0s)
Time = 300.0000 (sec), delta t in [0.14941875, 0.14947894] (s), steps=134 (0s)
```

[48]:
```
<matplotlib.animation.FuncAnimation at 0x7f94a5c96af0>
```

3.5.9 SWW File

The evolve loop saves the quantites at the end of each yield step to an `sww` file, with name domain name + extension sww. In this case the sww file is `merewether1.sww`.

An sww file can be viewed via our 3D anuga-viewer application, via the crayfish plugin for QGIS, or simply read back into python using netcdf commands.

For this clinic we have provided a wrapper called an SWW_plotter to provide easy acces to the saved quantities, `stage`, `elev`, `depth`, `xmom`, `ymom`, `xvel`, `yvel`, `speed` which are all time slices of centroid values, and a time variable.

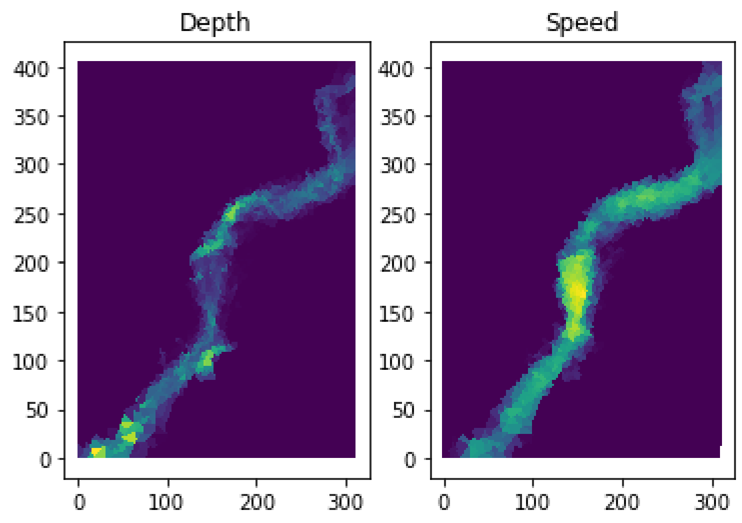
[49]:
```
# Create a wrapper for contents of sww file
swwfile = 'merewether1.sww'
splotter = anuga.SWW_plotter(swwfile)


# Plot Depth and Speed at the last time slice
plt.subplot(121)
splotter.triang.set_mask(None)
```

(continues on next page)

(continued from previous page)

```
plt.tripcolor(splotter.triang,
              facecolors = splotter.depth[-1,:],
              cmap='viridis')

plt.title("Depth")


plt.subplot(122)
splotter.triang.set_mask(None)
plt.tripcolor(splotter.triang,
              facecolors = splotter.speed[-1,:],
              cmap='viridis')

plt.title("Speed");
```

```
Figure files for each frame will be stored in _plot
```



3.5.10 Comparison

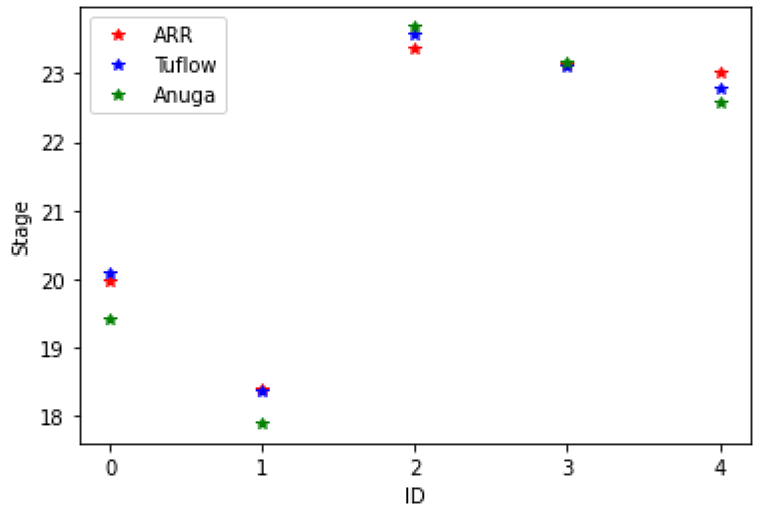
The data file `ObservationPoints.csv` contains some comparison depth data from Australian Rain and Runoff. Let's plot the depth for our simulation against the comparison data.

```
[50]: point_observations = np.genfromtxt(
          os.path.join(data_dir,'ObservationPoints.csv'),
          delimiter=",",skip_header=1)

      # Convert to absolute corrdinates
      xc = splotter.xc + splotter.xllcorner
      yc = splotter.yc + splotter.yllcorner

      nearest_points = []
      for row in point_observations:
          nearest_points.append(np.argmin( (xc-row[0])**2 + (yc-row[1])**2 ))
```

(continues on next page)

(continued from previous page)

```
loc_id = point_observations[:,2]

fig, ax = plt.subplots()
ax.plot(loc_id, point_observations[:,4], '*r', label='ARR')
ax.plot(loc_id, point_observations[:,5], '*b', label='Tuflow')
ax.plot(loc_id, splotter.stage[-1,nearest_points], '*g', label='Anuga')

plt.xticks(range(0,5))
plt.xlabel('ID')
plt.ylabel('Stage')
ax.legend()

plt.show()
```



3.5.11 Flow with Houses

We have polygonal data which specifies the location of a number of structures (homes) in our study. We can consider the flow in which those houses are cut out of the simulation.

First we read in the house polygonal data. To maintain a small mesh size we will only read in structures with an area grester than 60 m^2.

[51]:
```
# Read in house polygons from data directory and retain those of area > 60 m^2

import glob
house_files = glob.glob(os.path.join(data_dir,'house*.csv'))

house_polygons = []
for hf in house_files:
  house_poly = anuga.read_polygon(hf)
  poly_area = anuga.polygon_area(house_poly)

  # Leave out some small houses
```

(continues on next page)

(continued from previous page)

```
  if poly_area > 60:
    house_polygons.append(house_poly)
```

3.5.12 Create Domain

To incorporate the housing information, we will cutout the polygons representing the houses. This is done by passing the list of house polygons to the `interior_holes` argument of the `anuga.create_domain_from_regions` procedure.

This will produce a new tagged boundary region called `interior`. We will have to assign a boundsry condition to this new boundary region.

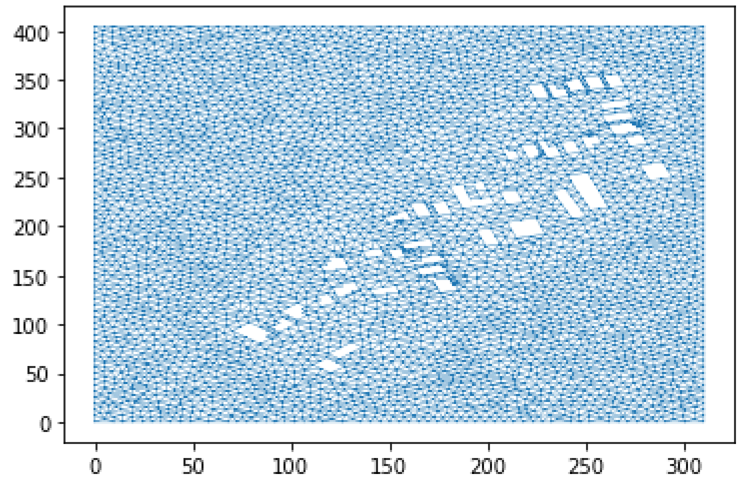
[52]:
```
# Resolution for most of the mesh
base_resolution = 20.0  # m^2

# Resolution in particular area of interest
merewether_resolution = 10.0 # m^2

domain = anuga.create_domain_from_regions(
            bounding_polygon,
            boundary_tags={'bottom': [0],
                           'right':  [1],
                           'top':    [2],
                           'left':   [3]},
            maximum_triangle_area=base_resolution,
            interior_holes=house_polygons,
            interior_regions=interior_regions)


domain.set_name('merewether2') # Name of sww file
domain.set_low_froude(1)

# Setup Initial Conditions
domain.set_quantity('elevation', filename=topography_file, location='centroids') # Use␣
↪function for elevation
domain.set_quantity('friction', 0.01, location='centroids')                     #␣
↪Constant friction
domain.set_quantity('stage', expression='elevation', location='centroids')      # Dry␣
↪Bed

# Setup BC
Br = anuga.Reflective_boundary(domain)
Bt = anuga.Transmissive_boundary(domain)


# NOTE: We need to assign a BC to the interior boundary region.
domain.set_boundary({'bottom':   Br,
                     'right':    Bt, # outflow
                     'top':      Bt, # outflow
                     'left':     Br,
                     'interior': Br})
```

(continues on next page)

(continued from previous page)

```
# Setup inlet flow
center = (382270.0, 6354285.0)
radius = 10.0
region0 = anuga.Region(domain, center=center, radius=radius)
fixed_inflow = anuga.Inlet_operator(domain, region0 , Q=19.7)


dplotter = anuga.Domain_plotter(domain)
plt.triplot(dplotter.triang, linewidth = 0.4);
```

```
Figure files for each frame will be stored in _plot
```



3.5.13 Evolve

[53]:
```
for t in domain.evolve(yieldstep=20, duration=300):

    #dplotter.plot_depth_frame()
    dplotter.save_depth_frame(vmin=0.0, vmax=1.0)

    domain.print_timestepping_statistics()


# Read in the png files stored during the evolve loop
dplotter.make_depth_animation()
```

```
Time = 0.0000 (sec), steps=0 (2s)
Time = 20.0000 (sec), delta t = 1000.00000000 (s), steps=1 (0s)
Time = 40.0000 (sec), delta t in [0.13714328, 0.20383256] (s), steps=124 (0s)
Time = 60.0000 (sec), delta t in [0.13454129, 0.15865474] (s), steps=137 (0s)
Time = 80.0000 (sec), delta t in [0.14031636, 0.15568732] (s), steps=134 (0s)
Time = 100.0000 (sec), delta t in [0.14006036, 0.14969987] (s), steps=140 (0s)
Time = 120.0000 (sec), delta t in [0.10431407, 0.14241125] (s), steps=160 (0s)
Time = 140.0000 (sec), delta t in [0.09577573, 0.10772380] (s), steps=200 (0s)
Time = 160.0000 (sec), delta t in [0.09570496, 0.10573006] (s), steps=199 (0s)
```

(continues on next page)

(continued from previous page)

```
Time = 180.0000 (sec), delta t in [0.09334587, 0.09569365] (s), steps=213 (0s)
Time = 200.0000 (sec), delta t in [0.09276761, 0.09334463] (s), steps=216 (0s)
Time = 220.0000 (sec), delta t in [0.09262018, 0.09276757] (s), steps=216 (0s)
Time = 240.0000 (sec), delta t in [0.09257185, 0.09261972] (s), steps=217 (0s)
Time = 260.0000 (sec), delta t in [0.09261584, 0.09269557] (s), steps=216 (0s)
Time = 280.0000 (sec), delta t in [0.09268672, 0.09269568] (s), steps=216 (0s)
Time = 300.0000 (sec), delta t in [0.09267530, 0.09268670] (s), steps=216 (0s)
```

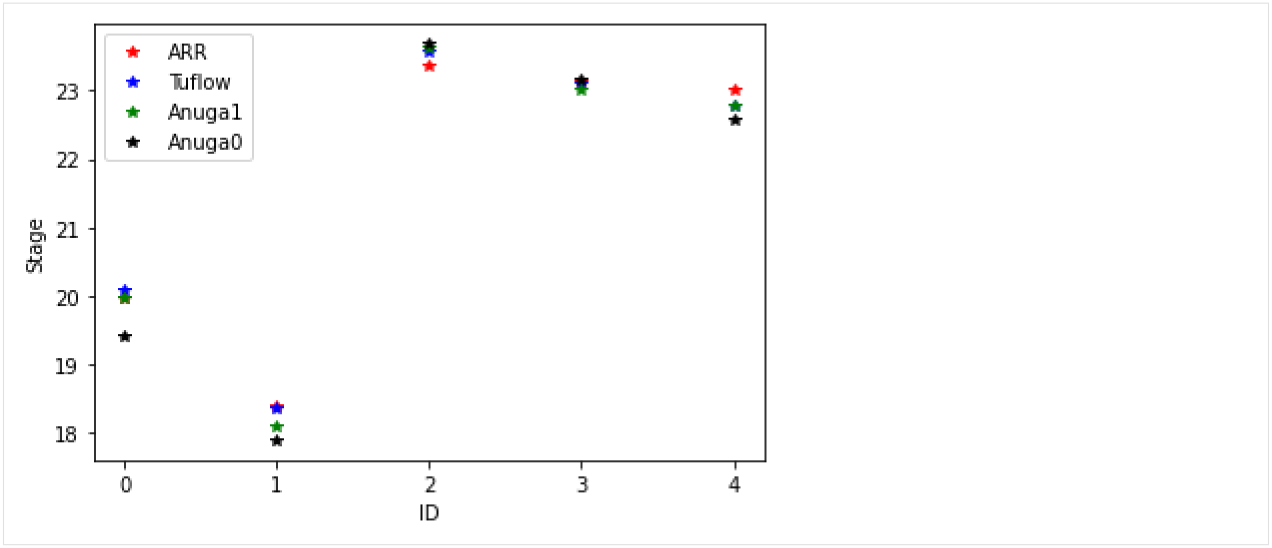
[53]:
```
<matplotlib.animation.FuncAnimation at 0x7f94a29c9f40>
```

3.5.14 Read in SWW File and Compare

Perhaps not conclusive, but with the houses the anuga results, especially for id point 0, are much closer to the comparison results. Note that we are running with a very coarse mesh for this case study.

[54]:
```
# Create a wrapper for contents of sww file
swwfile2 = 'merewether2.sww'
splotter2 = anuga.SWW_plotter(swwfile2)

# Convert to absolute corrdinates
xc = splotter2.xc + splotter2.xllcorner
yc = splotter2.yc + splotter2.yllcorner

nearest_points_2 = []
for row in point_observations:
    nearest_points_2.append(np.argmin( (xc-row[0])**2 + (yc-row[1])**2 ))

loc_id = point_observations[:,2]

fig, ax = plt.subplots()
ax.plot(loc_id, point_observations[:,4], '*r', label='ARR')
ax.plot(loc_id, point_observations[:,5], '*b', label='Tuflow')
ax.plot(loc_id, splotter2.stage[-1,nearest_points_2], '*g', label='Anuga1')
ax.plot(loc_id, splotter.stage[-1,nearest_points], '*k', label='Anuga0')


plt.xticks(range(0,5))
plt.xlabel('ID')
plt.ylabel('Stage')
ax.legend()

plt.show()
```

```
Figure files for each frame will be stored in _plot
```



3.6 Tsunami runup example

Validation of the AnuGA implementation of the shallow water wave equation. This script sets up Okushiri Island benchmark as published at the Third International Workshop on Long-Wave Runup Models.

The validation data is available from our anuga-clinic repository and the original data is available online where a detailed description of the problem is also available.

3.6.1 Setup Notebook for Visualisation and Animation

We are using the format of a jupyter notebook. As such we need to setup inline matplotlib plotting and animation.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

# Allow inline jshtml animations
from matplotlib import rc
rc('animation', html='jshtml')
```

3.6.2 Import ANUGA

We assume that anuga has been installed. If so we can import anuga.

[2]:
```
import anuga
```

3.6.3 The Code

This code is taken from run_okishuri.py.

First we define the location of our data files, which specify:

(1) the incoming tsunami wave,

(2) the bathymetry file,

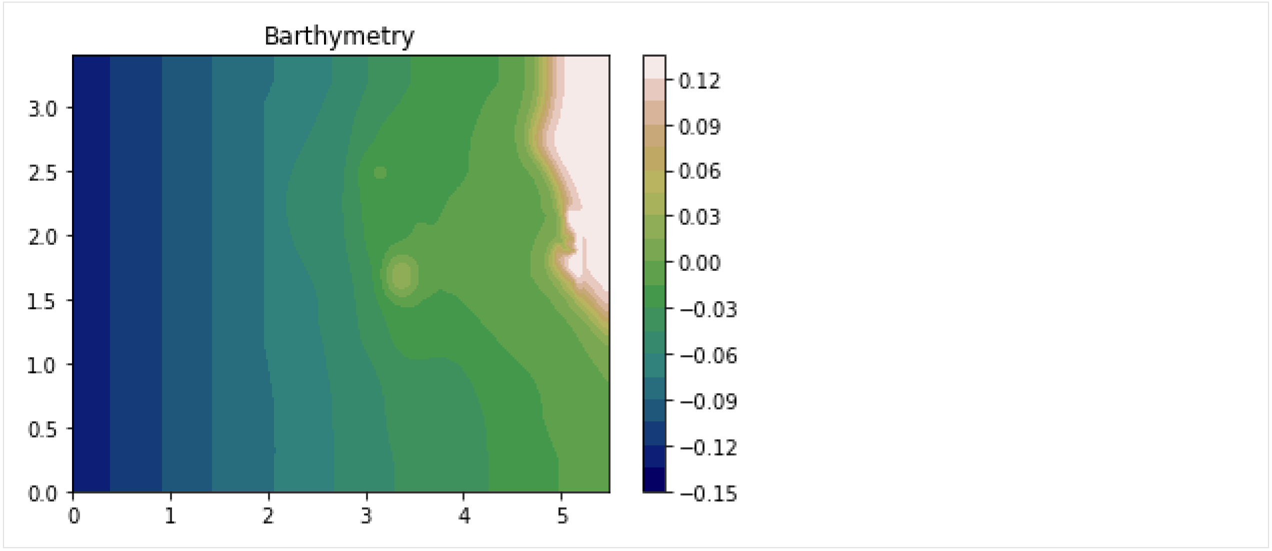
(3) the measured stage height at 3 gauge locations.

[3]:
```
# Incoming boundary wave (m)
boundary_filename = '/home/anuga/anuga-clinic/data/Okushiri/Benchmark_2_input.txt'

# Digital Elevation Model (x,y,z) (m)
bathymetry_filename = '/home/anuga/anuga-clinic/data/Okushiri/Benchmark_2_Bathymetry.xya'

# Observed timeseries (cm)
gauge_filename = '/home/anuga/anuga-clinic/data/Okushiri/output_ch5-7-9.txt'
```

3.6.4 Load Barthymetry Data

Using in the barthymetry data provided from the workshop. We need to reshape the data and form a raster (x,y,Z).

[4]:
```
xya = np.loadtxt(bathymetry_filename, skiprows=1, delimiter=',')

X = xya[:,0].reshape(393,244)
Y = xya[:,1].reshape(393,244)
Z = xya[:,2].reshape(393,244)


plt.contourf(X,Y,Z, 20, cmap=plt.get_cmap('gist_earth'));
plt.title('Barthymetry')
plt.colorbar();

# Create raster tuple
x = X[:,0]
y = Y[0,:]
Zr = np.flipud(Z.T)

raster = x,y,Zr
```



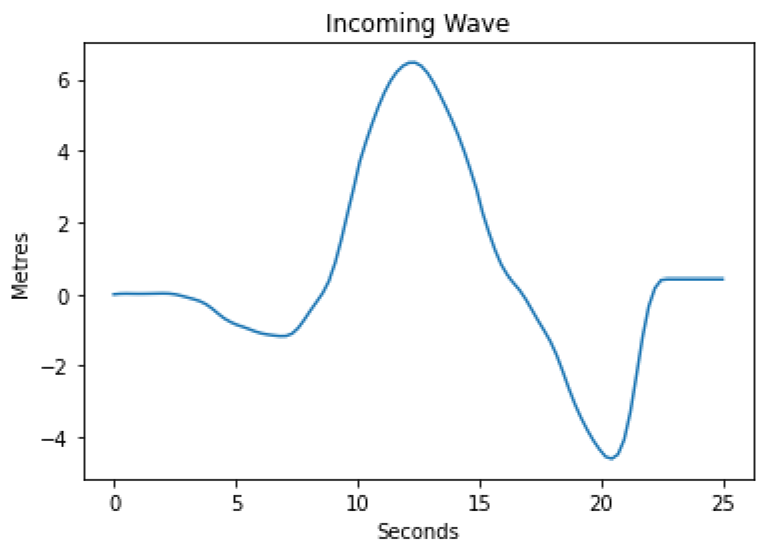
3.6.5 Load Incoming Wave

We will apply an incoming wave on the left boundary. So first we load the data from the `boundary_filename` file.

From the data we form an interpolation functon called `wave_function`, which will be used to specify the boundary condition on the left.

And we also plot the function. The units of the data in the file are metres, and the scale of the experimental setup is 1 in 400.

[5]:
```
bdry = np.loadtxt(boundary_filename, skiprows=1)

bdry_t = bdry[:,0]
bdry_v = bdry[:,1]


import scipy.interpolate
wave_function = scipy.interpolate.interp1d(bdry_t, bdry_v, kind='zero', fill_value=
↪'extrapolate')

t = np.linspace(0.0,25.0, 100)

plt.plot(t,wave_function(t)*400);
plt.xlabel('Seconds')
plt.ylabel('Metres')
plt.title('Incoming Wave');
```



3.6.6 Setup Domain

We define the domain for our simulation. This object encapsulates the mesh for our problem, which is defined by setting up a bounding polygon and associated tagged boundary. We use the base_resolution variable to set the maximum area of the triangles of our mesh.

At the end we use matplotlib to visualise the mesh associated with the domain.

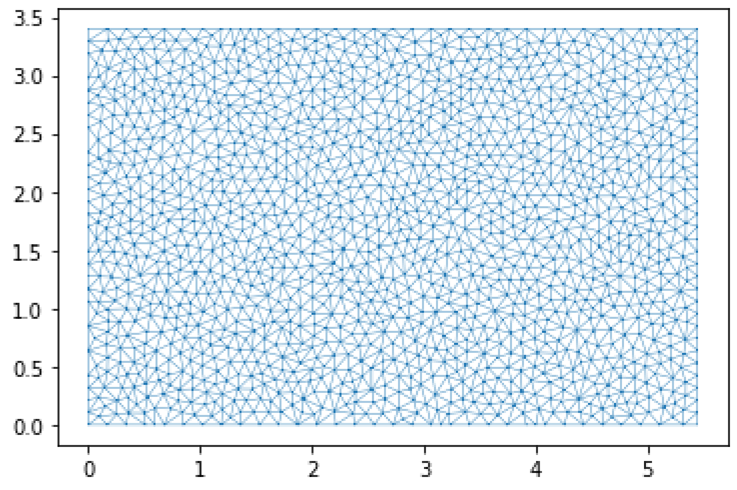
[6]:
```
base_resolution = 0.01
#base_resolution = 0.0005

# Basic geometry and bounding polygon
xleft   = 0
xright  = 5.448
ybottom = 0
ytop    = 3.402

point_sw = [xleft, ybottom]
point_se = [xright, ybottom]
point_nw = [xleft, ytop]
point_ne = [xright, ytop]

bounding_polygon = [point_se,
                    point_ne,
                    point_nw,
                    point_sw]

domain = anuga.create_domain_from_regions(bounding_polygon,
                                 boundary_tags={'wall': [0, 1, 3],
                                                'wave': [2]},
                                 maximum_triangle_area=base_resolution,
                                 use_cache=False,
                                 verbose=False)
```

(continues on next page)

(continued from previous page)

```
domain.set_name('okushiri')  # Name of output sww file
domain.set_minimum_storable_height(0.001) # Don't store w-z < 0.001m
domain.set_flow_algorithm('DE1')

print ('Number of Elements ', domain.number_of_elements)

dplotter = anuga.Domain_plotter(domain, min_depth=0.001)
plt.triplot(dplotter.triang, linewidth = 0.4);
```

```
Number of Elements  2884
Figure files for each frame will be stored in _plot
```

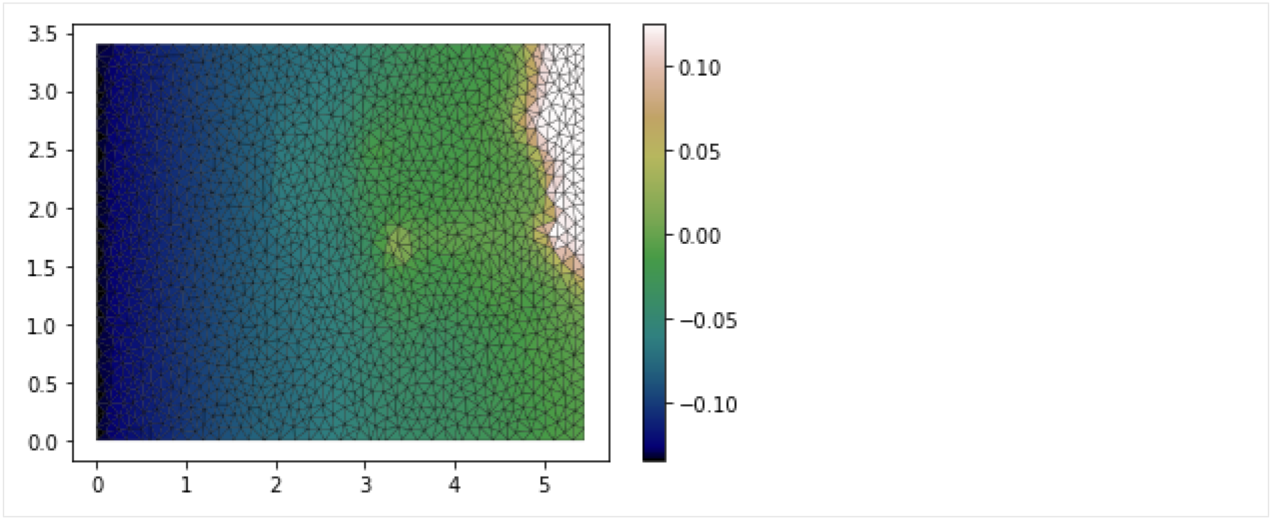


3.6.7 Setup Quantities

We use the `raster` created earlier to set the `quantity` called `elevation`. We also set the `stage` and the Mannings `friction`.

We also visualise the `elevation` quantity.

[7]:
```
domain.set_quantity('elevation',raster=raster, location='centroids')
domain.set_quantity('stage', 0.0)
domain.set_quantity('friction', 0.0025)


plt.tripcolor(dplotter.triang,
              facecolors = dplotter.elev,
              edgecolors='k',
              cmap='gist_earth')
plt.colorbar();
```



3.6.8 Setup Boundary Conditions

Excuse the verbose boundary type name `Transmissive_n_momentum_zero_t_momentum_set_stage_boundary`, but we use that to set the incoming wave boundary condition.

On the other boundaries we will have just reflective boundaries.

[8]:
```
Bts = anuga.Transmissive_n_momentum_zero_t_momentum_set_stage_boundary(domain, wave_
↪function)

Br = anuga.Reflective_boundary(domain)

domain.set_boundary({'wave': Bts, 'wall': Br})
```

3.6.9 Setup Interogation Variables

We will record the stage at the 3 gauge locations and at the Monai valley.

[9]:
```
yieldstep = 0.5
finaltime = 25.0

nt = int(finaltime/yieldstep)+1


# area for gulleys
x1 = 4.85
x2 = 5.25
y1 = 2.05
y2 = 1.85

# indices in gulley area
x = domain.centroid_coordinates[:,0]
y = domain.centroid_coordinates[:,1]
v = np.sqrt( (x-x1)**2 + (y-y1)**2 ) + np.sqrt( (x-x2)**2 + (y-y2)**2 ) < 0.5
```

(continues on next page)

(continued from previous page)

```
# Gauge and bounday locations
gauge = [[4.521, 1.196],  [4.521, 1.696],  [4.521, 2.196]] #Ch 5-7-9
bdyloc = [0.00001, 2.5]
g5_id = domain.get_triangle_containing_point(gauge[0])
g7_id = domain.get_triangle_containing_point(gauge[1])
g9_id = domain.get_triangle_containing_point(gauge[2])
bc_id = domain.get_triangle_containing_point(bdyloc)

# Arrays to store data
meanstage = np.nan*np.ones((nt,))
g5 = np.nan*np.ones((nt,))
g7 = np.nan*np.ones((nt,))
g9 = np.nan*np.ones((nt,))
bc = np.nan*np.ones((nt,))
```

3.6.10 Evolve

[10]:
```
Stage = domain.quantities['stage'].centroid_values
stage_qoi = Stage[v]

k = 0
for t in domain.evolve(yieldstep=yieldstep, finaltime=finaltime):

    domain.print_timestepping_statistics()

    # stage
    stage_qoi = Stage[v]
    meanstage[k] = np.mean(stage_qoi)
    g5[k] = Stage[g5_id]
    g7[k] = Stage[g7_id]
    g9[k] = Stage[g9_id]
    bc[k] = Stage[bc_id]

    k = k+1

    #dplotter.save_depth_frame()


# Read in the png files stored during the evolve loop
#dplotter.make_depth_animation()
```

```
Time = 0.0000 (sec), steps=0 (17s)
Time = 0.5000 (sec), delta t in [0.01553114, 0.01553127] (s), steps=33 (0s)
Time = 1.0000 (sec), delta t in [0.01552392, 0.01553122] (s), steps=33 (0s)
Time = 1.5000 (sec), delta t in [0.01552381, 0.01552585] (s), steps=33 (0s)
Time = 2.0000 (sec), delta t in [0.01552456, 0.01552599] (s), steps=33 (0s)
Time = 2.5000 (sec), delta t in [0.01552109, 0.01552454] (s), steps=33 (0s)
Time = 3.0000 (sec), delta t in [0.01552098, 0.01553040] (s), steps=33 (0s)
Time = 3.5000 (sec), delta t in [0.01551921, 0.01553066] (s), steps=33 (0s)
```

(continues on next page)

(continued from previous page)

```
Time = 4.0000 (sec), delta t in [0.01550540, 0.01551913] (s), steps=33 (0s)
Time = 4.5000 (sec), delta t in [0.01548068, 0.01550529] (s), steps=33 (0s)
Time = 5.0000 (sec), delta t in [0.01544420, 0.01548039] (s), steps=33 (0s)
Time = 5.5000 (sec), delta t in [0.01542374, 0.01544386] (s), steps=33 (0s)
Time = 6.0000 (sec), delta t in [0.01541583, 0.01542361] (s), steps=33 (0s)
Time = 6.5000 (sec), delta t in [0.01540816, 0.01541572] (s), steps=33 (0s)
Time = 7.0000 (sec), delta t in [0.01540731, 0.01540954] (s), steps=33 (0s)
Time = 7.5000 (sec), delta t in [0.01540962, 0.01541778] (s), steps=33 (0s)
Time = 8.0000 (sec), delta t in [0.01541798, 0.01545159] (s), steps=33 (0s)
Time = 8.5000 (sec), delta t in [0.01545233, 0.01553221] (s), steps=33 (0s)
Time = 9.0000 (sec), delta t in [0.01548376, 0.01558391] (s), steps=33 (0s)
Time = 9.5000 (sec), delta t in [0.01514249, 0.01548252] (s), steps=33 (0s)
Time = 10.0000 (sec), delta t in [0.01455953, 0.01513366] (s), steps=34 (0s)
Time = 10.5000 (sec), delta t in [0.01398266, 0.01454743] (s), steps=36 (0s)
Time = 11.0000 (sec), delta t in [0.01358727, 0.01398170] (s), steps=37 (0s)
Time = 11.5000 (sec), delta t in [0.01329014, 0.01358455] (s), steps=38 (0s)
Time = 12.0000 (sec), delta t in [0.01311287, 0.01328876] (s), steps=38 (0s)
Time = 12.5000 (sec), delta t in [0.01304909, 0.01311013] (s), steps=39 (0s)
Time = 13.0000 (sec), delta t in [0.01304870, 0.01310310] (s), steps=39 (0s)
Time = 13.5000 (sec), delta t in [0.01310395, 0.01328468] (s), steps=38 (0s)
Time = 14.0000 (sec), delta t in [0.01329067, 0.01357083] (s), steps=38 (0s)
Time = 14.5000 (sec), delta t in [0.01357297, 0.01390371] (s), steps=37 (0s)
Time = 15.0000 (sec), delta t in [0.01390790, 0.01432648] (s), steps=36 (0s)
Time = 15.5000 (sec), delta t in [0.01433271, 0.01485796] (s), steps=35 (0s)
Time = 16.0000 (sec), delta t in [0.01486278, 0.01510085] (s), steps=34 (0s)
Time = 16.5000 (sec), delta t in [0.01475513, 0.01495640] (s), steps=34 (0s)
Time = 17.0000 (sec), delta t in [0.01462906, 0.01475218] (s), steps=35 (0s)
Time = 17.5000 (sec), delta t in [0.01456646, 0.01462895] (s), steps=35 (0s)
Time = 18.0000 (sec), delta t in [0.01453566, 0.01456620] (s), steps=35 (0s)
Time = 18.5000 (sec), delta t in [0.01453288, 0.01454775] (s), steps=35 (0s)
Time = 19.0000 (sec), delta t in [0.01454807, 0.01456682] (s), steps=35 (0s)
Time = 19.5000 (sec), delta t in [0.01456703, 0.01462551] (s), steps=35 (0s)
Time = 20.0000 (sec), delta t in [0.01462621, 0.01472820] (s), steps=35 (0s)
Time = 20.5000 (sec), delta t in [0.01467696, 0.01482861] (s), steps=34 (0s)
Time = 21.0000 (sec), delta t in [0.01177207, 0.01485123] (s), steps=37 (0s)
Time = 21.5000 (sec), delta t in [0.01176240, 0.01269140] (s), steps=41 (0s)
Time = 22.0000 (sec), delta t in [0.01270128, 0.01311657] (s), steps=39 (0s)
Time = 22.5000 (sec), delta t in [0.01312089, 0.01345244] (s), steps=38 (0s)
Time = 23.0000 (sec), delta t in [0.01345847, 0.01374983] (s), steps=37 (0s)
Time = 23.5000 (sec), delta t in [0.01362081, 0.01378886] (s), steps=37 (0s)
Time = 24.0000 (sec), delta t in [0.01377684, 0.01398414] (s), steps=37 (0s)
Time = 24.5000 (sec), delta t in [0.01398436, 0.01431604] (s), steps=36 (0s)
Time = 25.0000 (sec), delta t in [0.01431817, 0.01446696] (s), steps=35 (0s)
```

3.6.11 Animation Using swwfile

Read in the sww file and then iterate through the time slices to produce an animation.

[11]:
```
swwplotter = anuga.SWW_plotter('okushiri.sww', min_depth = 0.001)

n = len(swwplotter.time)

for k in range(n):
  swwplotter.save_stage_frame(frame=k, vmin=-0.02, vmax = 0.1)

swwplotter.make_stage_animation()
```

```
Figure files for each frame will be stored in _plot
```

[11]:
```
<matplotlib.animation.FuncAnimation at 0x7f74354d9e80>
```

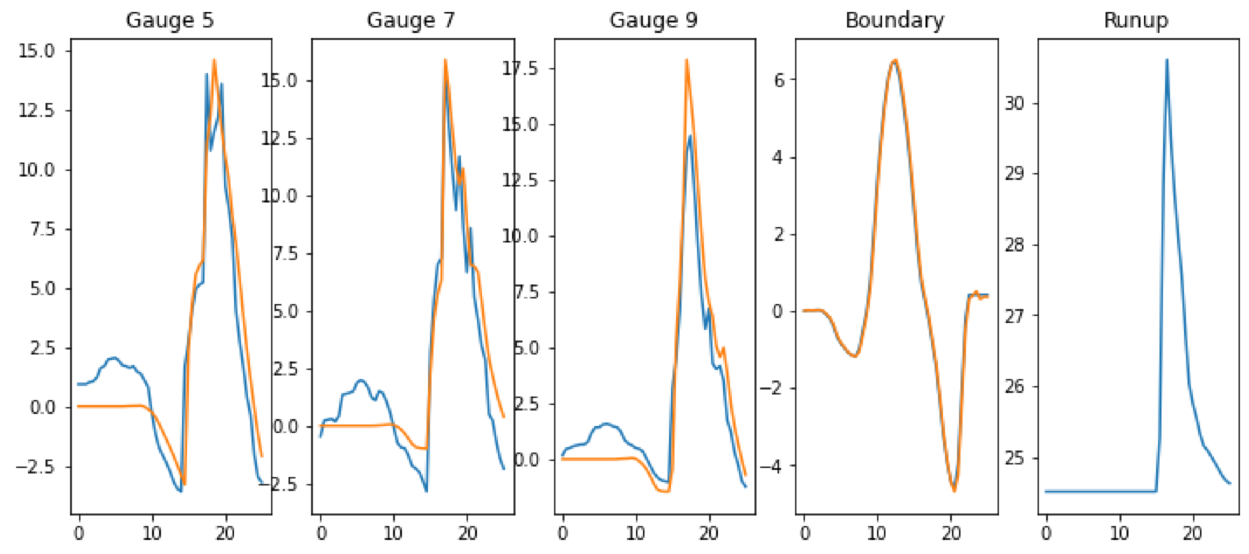
3.6.12 View Time Series

[12]:
```
old_figsize = plt.rcParams['figure.figsize']

plt.rcParams['figure.figsize'] = [12, 5]

gauge = np.loadtxt(gauge_filename, skiprows=1)

gauge_t = gauge[:,0]
gauge_5 = gauge[:,1]
gauge_7 = gauge[:,2]
gauge_9 = gauge[:,3]

nt = int(finaltime/yieldstep)+1

import scipy
gauge_5_f = scipy.interpolate.interp1d(gauge_t, gauge_5, kind='zero', fill_value=
↪'extrapolate')
gauge_7_f = scipy.interpolate.interp1d(gauge_t, gauge_7, kind='zero', fill_value=
↪'extrapolate')
gauge_9_f = scipy.interpolate.interp1d(gauge_t, gauge_9, kind='zero', fill_value=
↪'extrapolate')

t = np.linspace(0.0,finaltime, nt)

tt= np.linspace(0.0,finaltime, nt)

plt.subplot(1,5,1)
plt.plot(t,gauge_5_f(t)*4)
plt.plot(tt,g5*400)
plt.title('Gauge 5')

plt.subplot(1,5,2)
plt.plot(t,gauge_7_f(t)*4)
plt.plot(tt,g7*400)
plt.title('Gauge 7')
```

(continues on next page)

(continued from previous page)

```
plt.subplot(1,5,3)
plt.plot(t,gauge_9_f(t)*4)
plt.plot(tt,g9*400)
plt.title('Gauge 9')

plt.subplot(1,5,4)
plt.plot(t,wave_function(t)*400)
plt.plot(tt,bc*400)
plt.title('Boundary')

plt.subplot(1,5,5)
plt.plot(tt,meanstage*400)
plt.title('Runup');


plt.rcParams['figure.figsize'] = old_figsize
```



CHAPTER

FOUR

CREATING A DOMAIN

anuga.create_domain_from_regions(...[, ...])	Create domain from bounding polygons and resolutions.

4.1 anuga.create_domain_from_regions

anuga.**create_domain_from_regions**(*bounding_polygon*, *boundary_tags*, *maximum_triangle_area=None*, *mesh_filename=None*, *interior_regions=None*, *interior_holes=None*, *hole_tags=None*, *poly_geo_reference=None*, *mesh_geo_reference=None*, *breaklines=None*, *regionPtArea=None*, *minimum_triangle_angle=28.0*, *fail_if_polygons_outside=True*, *use_cache=False*, *verbose=False*)

Create domain from bounding polygons and resolutions.

Parameters

- **bounding_polygon** – list of points in Eastings and Northings, relative to the zone stated in poly_geo_reference if specified. Otherwise points are just x, y coordinates with no particular association to any location.
- **boundary_tags** – dictionary of symbolic tags. For every tag there is a list of indices referring to segments associated with that tag. If a segment is omitted it will be assigned the default tag ‘’.
- **maximum_triangle_area** – maximal area per triangle for the bounding polygon, excluding the interior regions.
- **Interior_regions** – list of tuples consisting of (polygon, resolution) for each region to be separately refined. Do not have polygon lines cross or be on-top of each other. Also do not have polygon close to each other. NOTE: If a interior_region is outside the bounding_polygon it should throw an error
- **interior_holes** – list of polygons for each hole. These polygons do not need to be closed, but their points must be specified in a counter-clockwise order.
- **hole_tags** – list of tag segment dictionaries. This function does not allow segments to share points - use underlying pmesh functionality for that
- **poly_geo_reference** – geo_reference of the bounding polygon and the interior polygons. If none, assume absolute. Please pass one though, since absolute references have a zone.
- **mesh_geo_reference** – geo_reference of the mesh to be created. If none is given one will be automatically generated. It was use the lower left hand corner of bounding_polygon (absolute) as the x and y values for the geo_ref.

- `breaklines` – list of polygons. These lines will be preserved by the triangulation algorithm - useful for coastlines, walls, etc. The polygons are not closed.
- `regionPtArea` – list of 3-tuples specifing a point with max area for region containing point
- `fail_if_polygons_outside` – If True (the default) Exception in thrown where interior polygons fall outside bounding polygon. If False, these will be ignored and execution continued.

Returns
shallow water domain instance

Note: Interior regions should be fully nested, as overlaps may cause unintended resolutions.

CHAPTER FIVE

INITIAL CONDITIONS

`Domain.set_quantity`(name, *args, **kwargs)	Set values for named quantity

5.1 anuga.Domain.set_quantity

Domain.**set_quantity**(*name*, **args*, ***kwargs*)

Set values for named quantity

We have to do something special for ‘elevation’ otherwise pass through to generic set_quantity

CHAPTER

SIX

BOUNDARIES

This being worked on. You can find the material in the pdf file anuga_user_manual.pdf in the doc section of anuga_core.

CHAPTER

SEVEN

EVOLVE

Running a ANUGA model involves five basic steps:

- *Creating a domain*
- *Setting up the initial conditions*
- *Settting up the boundary condition*
- *Setting up any stuctures or operators*
- *Evolving the model*

Here we describe the last step, how to run (evolve) the model for a specified amount of time.

7.1 Evolving the Model

In addition to evolving the model, it would good to be able to interact with the evolving model. This is all provided by the `evolve` method of the *Domain* object.

Suppose we have created and set up a Domain by completing the first 4 basic steps. For example here is such a setup for a domain object called *domain*:

```
>>> domain = anuga.rectangular_cross_domain(10,5)
>>> domain.set_quantity('elevation', function = lambda x,y : x/10)
>>> domain.set_quantity('stage', expression = "elevation + 0.2" )
>>> Br = anuga.Reflective_boundary(domain)
>>> domain.set_boundary({'left' : Br, 'right' : Br, 'top' : Br, 'bottom' : Br})
```

To evolve the model we would use the domain's evolve method, using the following code:

```
>>> for t in domain.evolve(yieldstep=1.0, finaltime=10.0):
>>>    pass
```

This will run the model from *time=0* to the *finaltime = 10.0*. The method will "yield" to the for loop every *yieldstep = 1*. By default the state of the simulation will be saved to a file (by default named *domain.sww*) every *yieldstep*, in this case every 1 second of simulation time.

As the *evolve* construct provides a *for* loop (via the python *yield* construct) it is possible to include extra code within the loop. A typical *evolve* loop can provide some printed feedback using the `print_timestepping_statistics` method, i.e.,

```
>>> for t in domain.evolve(yieldstep=1.0, finaltime=10.0):
>>>    domain.print_timestepping_statistics()
Time = 0.0000 (sec), steps=0 (33s)
Time = 1.0000 (sec), delta t in [0.00858871, 0.01071429] (s), steps=111 (0s)
Time = 2.0000 (sec), delta t in [0.00832529, 0.00994060] (s), steps=110 (0s)
Time = 3.0000 (sec), delta t in [0.00901413, 0.00993095] (s), steps=106 (0s)
Time = 4.0000 (sec), delta t in [0.00863985, 0.00963487] (s), steps=109 (0s)
Time = 5.0000 (sec), delta t in [0.00887345, 0.00990731] (s), steps=106 (0s)
Time = 6.0000 (sec), delta t in [0.00934142, 0.00988233] (s), steps=104 (0s)
Time = 7.0000 (sec), delta t in [0.00904828, 0.00970252] (s), steps=107 (0s)
Time = 8.0000 (sec), delta t in [0.00917360, 0.00985509] (s), steps=106 (0s)
Time = 9.0000 (sec), delta t in [0.00925747, 0.00984041] (s), steps=104 (0s)
Time = 10.0000 (sec), delta t in [0.00927581, 0.00973202] (s), steps=106 (0s)
```

During the evolution the yieldsteps are fixed but to maintain stability of the simulation, the underlying computation uses inner evolve timesteps which are generally much smaller than the yieldstep. The number of these inner evolve timesteps are reported as steps and the range of the sizes of these evolve timesteps are reported as the delta t.

7.2 Duration instead of finaltime

It can also be convenient to evolve for a specific *duration*. In this case we replace the *finaltime* argument with *duration*. I.e. let us continue the evolution for 7 seconds with yieldstep now set to 2 seconds.

```
>>> for t in domain.evolve(yieldstep=2.0, duration=7.0):
>>>    domain.print_timestepping_statistics()
Time = 12.0000 (sec), delta t in [0.00932516, 0.00982159] (s), steps=209 (63s)
Time = 14.0000 (sec), delta t in [0.00941363, 0.00981322] (s), steps=210 (0s)
Time = 16.0000 (sec), delta t in [0.00944121, 0.00979934] (s), steps=208 (0s)
Time = 17.0000 (sec), delta t in [0.00945517, 0.00978655] (s), steps=105 (0s)
```

7.3 Outputstep

Sometimes it is necessary to interact with the evolution using a small *yieldstep* (such as controlling a hydraulic structure). In this case the *sww* file stored at each yieldstep can become prohibitively large.

Instead you can save the state every *outputstep* time interval, while still interacting every *yieldstep* interval.

For instance. let us continue the evolution, but now with a smaller yieldstep of 0.5 seconds, but with output to *domain.sww* every 2 seconds.

```
>>> for t in domain.evolve(yieldstep=0.5, outputstep=2.0, duration=4.0):
>>>     domain.print_timestepping_statistics()
Time = 17.5000 (sec), delta t in [0.00964414, 0.00977317] (s), steps=52 (650s)
Time = 18.0000 (sec), delta t in [0.00946685, 0.00972477] (s), steps=53 (0s)
Time = 18.5000 (sec), delta t in [0.00953534, 0.00965620] (s), steps=53 (0s)
Time = 19.0000 (sec), delta t in [0.00955560, 0.00976215] (s), steps=52 (0s)
Time = 19.5000 (sec), delta t in [0.00947717, 0.00955428] (s), steps=53 (0s)
Time = 20.0000 (sec), delta t in [0.00955552, 0.00966630] (s), steps=53 (0s)
Time = 20.5000 (sec), delta t in [0.00951811, 0.00975266] (s), steps=52 (0s)
Time = 21.0000 (sec), delta t in [0.00948645, 0.00957223] (s), steps=53 (0s)
```

Typical situations could be *yieldstep = 1.0* and *outputstep=300*. In this case the *sww* file will be 300 times smaller than just using *yieldstep* for output.

7.4 Start Time

By default the evolution starts at time 0.0. To set another start time, simply use `set_starttime` before the evolve loop, i.e.

```
>>> domain.set_starttime(-3600*24)
```

to set the start time one day in the past (from ANUGA's zero time). This can be used to allow the model to "burn in" before starting the evolution proper.

7.5 Start times with DateTime and Timezones

To work with dates, times and timezones we can use the python module *datetime*. to setup a date and time (and timezone) associated with ANUGA's starttime time.

Once again let's suppose we have setup a domain via:

```
>>> import anuga
>>> from datetime import datetime
>>> domain = anuga.rectangular_cross_domain(10,5)
>>> domain.set_quantity('elevation', function = lambda x,y : x/10)
>>> domain.set_quantity('stage', expression = "elevation + 0.2" )
>>> Br = anuga.Reflective_boundary(domain)
>>> domain.set_boundary({'left' : Br, 'right' : Br, 'top' : Br, 'bottom' : Br})
```

By default ANUGA uses a UTC as the default timezone for the domain. We can change it via `set_timezone`

```
>>> domain.set_timezone('Australia/Sydney')
```

Suppose we want to start the model at 18:45 on the 21st July 2021. Use the *datetime* module to setup this date, and the set the start time, as follows:

```
>>> from datetime import datetime
>>> starttime = datetime(2021, 7, 21, 18, 45)
>>> domain.set_starttime(starttime)
```

Suppose we want to evolve until 19:00 on the 21st July 2021. Use *datetime* to setup this *finaltime*:

```
>>> finaltime = datetime(2021, 7, 21, 19, 0)
```

And now evolve the model. Note the use of the *datetime* = *True* argument for the `print_timestepping_statisitics` procedure.

```
>>> for t in domain.evolve(yieldstep=300, finaltime=finaltime):
>>>   domain.print_timestepping_statistics(datetime=True)
DateTime: 2021-07-21 18:45:00+1000, steps=0 (0s)
DateTime: 2021-07-21 18:50:00+1000, delta t in [0.00832571, 0.01071429] (s), steps=31233␣
↪(10s)
```

(continues on next page)

(continued from previous page)

```
DateTime: 2021-07-21 18:55:00+1000, delta t in [0.00959070, 0.00964172] (s), steps=31205␣
↪(10s)
DateTime: 2021-07-21 19:00:00+1000, delta t in [0.00959070, 0.00964172] (s), steps=31205␣
↪(10s)
```

7.6 Default zero time

We use unix timestamp as our underlying absolute time. So *time = 0* corresponds to Jan 1st 1970 UTC.

For instance going back to an earlier example which uses the default timezone (UTC) and 0 start time.

(Compare the output with *datetime = True* and *datetime = False* in the `print_timestepping_statistics` procedure.)

```
>>> import anuga
>>> domain = anuga.rectangular_cross_domain(10,5)
>>> domain.set_quantity('elevation', function = lambda x,y : x/10)
>>> domain.set_quantity('stage', expression = "elevation + 0.2" )
>>> Br = anuga.Reflective_boundary(domain)
>>> domain.set_boundary({'left' : Br, 'right' : Br, 'top' : Br, 'bottom' : Br})
>>>
>>> for t in domain.evolve(yieldstep=1, finaltime=5):
>>>   domain.print_timestepping_statistics(datetime=True)
DateTime: 1970-01-01 00:00:00+0000, steps=0 (10s)
DateTime: 1970-01-01 00:00:01+0000, delta t in [0.00858871, 0.01071429] (s), steps=111␣
↪(0s)
DateTime: 1970-01-01 00:00:02+0000, delta t in [0.00832529, 0.00994060] (s), steps=110␣
↪(0s)
DateTime: 1970-01-01 00:00:03+0000, delta t in [0.00901413, 0.00993095] (s), steps=106␣
↪(0s)
DateTime: 1970-01-01 00:00:04+0000, delta t in [0.00863985, 0.00963487] (s), steps=109␣
↪(0s)
DateTime: 1970-01-01 00:00:05+0000, delta t in [0.00887345, 0.00990731] (s), steps=106␣
↪(0s)
```

Note that the date is 1st Jan 1970, starting at time 0:00, incrementing by 1 sec and the UTC offset is +0000 (ie the timezone is UTC).

7.7 Useful Domain methods

`anuga.Domain.evolve`([yieldstep, outputstep, ...])	Evolve method from Domain class.
`anuga.Domain.print_timestepping_statistics`(...)	Print time stepping statistics
`anuga.Domain.set_starttime`([timestamp])	Set the starttime for the evolution
`anuga.Domain.set_timezone`([tz])	Set timezone for domain

7.7.1 anuga.Domain.evolve

`Domain.`**`evolve`**(*yieldstep=None*, *outputstep=None*, *finaltime=None*, *duration=None*, *skip_initial_step=False*)

Evolve method from Domain class.

Parameters

- **`yieldstep`** (*`float`*) – yield every yieldstep time period
- **`outputstep`** (*`float`*) – Output to sww file every outputstep time period. outputstep should be an integer multiple of yieldstep.
- **`finaltime`** (*`float`*) – evolve until finaltime (can be a float or a datetime object)
- **`duration`** (*`float`*) – evolve for a time of length duration
- **`skip_inital_step`** (*`boolean`*) – Can be used to restart a simulation (not often used).

If outputstep is None, the output to sww file happens every yieldstep. If yieldstep is None then simply evolve to finaltime or for a duration.

7.7.2 anuga.Domain.print_timestepping_statistics

`Domain.`**`print_timestepping_statistics`**(**args*, ***kwargs*)

Print time stepping statistics

Parameters

- **`time_units`** – 'sec', 'min', 'hr', 'day'
- **`datetime`** (*`bool`*) – flag to use timestamp or datetime
- **`track_speed`** – Optional boolean keyword track_speeds decides whether to report location of smallest timestep as well as a histogram and percentile report.
- **`relative_time`** (*`bool`*) – Flag to report relative time instead of absolute time
- **`triangle_id`** (*`int`*) – Can be used to specify a particular triangle rather than the one with the largest speed.

7.7.3 anuga.Domain.set_starttime

`Domain.`**`set_starttime`**(*timestamp=0.0*)

Set the starttime for the evolution

Parameters
`time` – Either a float or a datetime object

Essentially we use unix time as our absolute time. So time = 0 corresponds to Jan 1st 1970 UTC

Use naive datetime which will be localized to the domain timezone or or use pytz.timezone.localize to set timezone of datetime. Don't use the tzinfo argument of datetime to set timezone as this does not work!

Example:

Without setting timezone for the *domain* and the *starttime* then time calculations are all based on UTC. Note the timestamp, which is time in seconds from 1st Jan 1970 UTC.

```
>>> import pytz
>>> import anuga
>>> from datetime import datetime
>>>
>>> domain = anuga.rectangular_cross_domain(10,10)
>>> dt = datetime(2021,3,21,18,30)
>>> domain.set_starttime(dt)
>>> print(domain.get_datetime(), 'TZ', domain.get_timezone(), 'Timestamp: ', domain.
↪get_time())
2021-03-21 18:30:00+00:00 TZ UTC Timestamp:  1616351400.0
```

Example:

Setting timezone for the *domain*, then naive *datetime* will be localizes to the *domain* timezone. Note the timestamp, which is time in seconds from 1st Jan 1970 UTC.

```
>>> import pytz
>>> import anuga
>>> from datetime import datetime
>>>
>>> domain = anuga.rectangular_cross_domain(10,10)
>>> AEST = pytz.timezone('Australia/Sydney')
>>> domain.set_timezone(AEST)
>>>
>>> dt = datetime(2021,3,21,18,30)
>>> domain.set_starttime(dt)
>>> print(domain.get_datetime(), 'TZ', domain.get_timezone(), 'Timestamp: ', domain.
↪get_time())
2021-03-21 18:30:00+11:00 TZ Australia/Sydney Timestamp:  1616311800.0
```

Example:

Setting timezone for the *domain*, and setting the timezone for the *datetime*. Note the timestamp, which is time in seconds from 1st Jan 1970 UTC is the same as teh previous example.

```
>>> import pytz
>>> import anuga
>>> from datetime import datetime
>>>
>>> domain = anuga.rectangular_cross_domain(10,10)
>>>
>>> ACST = pytz.timezone('Australia/Adelaide')
>>> domain.set_timezone(ACST)
>>>
>>> AEST = pytz.timezone('Australia/Sydney')
>>> dt = AEST.localize(datetime(2021,3,21,18,30))
>>>
>>> domain.set_starttime(dt)
>>> print(domain.get_datetime(), 'TZ', domain.get_timezone(), 'Timestamp: ', domain.
↪get_time())
2021-03-21 18:00:00+10:30 TZ Australia/Adelaide Timestamp:  1616311800.0
```

7.7.4 anuga.Domain.set_timezone

`Domain.set_timezone(tz=None)`

Set timezone for domain

Parameters

`tz` – either a timezone object or string

We recommend using the timezone provided by the pytz modules. Default is pytz.utc

Example: Set default timezone UTC

```
>>> domain.set_timezone()
```

Example: Set timezone using pytz string

```
>>> domain.set_timezone('Australia/Syndey')
```

Example: Set timezone using pytz timezone

```
>>> new_tz = pytz.timezone('Australia/Sydney')
>>> domain.set_timezone(new_tz)
```

CHAPTER

EIGHT

OPERATORS

This being worked on. You can find the material in the pdf file anuga_user_manual.pdf in the doc section of anuga_core.

CHAPTER NINE

STRUCTURES

This being worked on. You can find the material in the pdf file anuga_user_manual.pdf in the doc section of anuga_core.

CHAPTER

TEN

REFERENCE

10.1 Introduction

After starting Python, import the anuga module with

```
>>> import anuga
```

To save repetition, in the documentation we assume that ANUGA has been imported this way.

If importing ANUGA fails, it means that Python cannot find the installed module. Check your installation and your PYTHONPATH.

The following Domain class is available:

Domain
This class initializes a domain object.

Initialize a ANUGA Model with

```
>>> domain = anuga.Domain()
```

Once a Domain is initialized, there are several options available to setup the domain (initial conditions, boundary conditions, operators) and run the model (evolve).

anuga	ANUGA models the effect of tsunamis and flooding upon a terrain mesh.

10.1.1 anuga

ANUGA models the effect of tsunamis and flooding upon a terrain mesh. In typical usage, a Domain class is created for a particular piece of terrain. Boundary conditions are specified for the domain, such as inflow and outflow, and then the simulation is run.

This is the public API to ANUGA. It provides a toolkit of often-used modules, which can be used directly by including the following line in the user's code:

```
>>> import anuga
```

This usage pattern abstracts away the internal heirarchy of the ANUGA system, allowing the user to concentrate on writing simulations without searching through the ANUGA source tree for the functions that they need.

Also, it isolates the user from "under-the-hood" refactorings.

Functions

`get_args()`	Explicitly parse the argument list using standard anuga arguments

10.2 Creating a Domain

10.3 Classes Associated with the Domain

anuga.Domain([coordinates, vertices, ...])	Object which encapulates the shallow water model
anuga.Quantity(domain[, vertex_values, ...])	Class Quantity - Implements values at each triangular element
anuga.Region(domain[, indices, polygon, ...])	Object which defines a region within the domain

10.3.1 anuga.Domain

class anuga.**Domain**(*coordinates=None*, *vertices=None*, *boundary=None*, *tagged_elements=None*, *geo_reference=None*, *use_inscribed_circle=False*, *mesh_filename=None*, *use_cache=False*, *verbose=False*, *conserved_quantities=None*, *evolved_quantities=None*, *other_quantities=None*, *full_send_dict=None*, *ghost_recv_dict=None*, *starttime=0*, *processor=0*, *numproc=1*, *number_of_full_nodes=None*, *number_of_full_triangles=None*, *ghost_layer_width=2*, ***kwargs*)

Object which encapulates the shallow water model

This class is a specialization of class Generic_Domain from module generic_domain.py consisting of methods specific to the Shallow Water Wave Equation

Shallow Water Wave Equation

$$U_t + E_x + G_y = S$$

where

$$U = [w, uh, vh]^T$$

$$E = [uh, u^2h + gh^2/2, uvh]$$

$$G = [vh, uvh, v^2h + gh^2/2]$$

S represents source terms forcing the system (e.g. gravity, friction, wind stress, . . .)

and _t, _x, _y denote the derivative with respect to t, x and y respectively.

The quantities are

symbol variable name explanation x x horizontal distance from origin [m] y y vertical distance from origin [m] z elevation elevation of bed on which flow is modelled [m] h height water height above z [m] w stage absolute water level, w = z+h [m] u speed in the x direction [m/s] v speed in the y direction [m/s] uh xmomentum momentum in the x direction [m^2/s] vh ymomentum momentum in the y direction [m^2/s]

eta mannings friction coefficient [to appear] nu wind stress coefficient [to appear]

The conserved quantities are w, uh, vh

`__init__`(*coordinates=None*, *vertices=None*, *boundary=None*, *tagged_elements=None*, *geo_reference=None*, *use_inscribed_circle=False*, *mesh_filename=None*, *use_cache=False*, *verbose=False*, *conserved_quantities=None*, *evolved_quantities=None*, *other_quantities=None*, *full_send_dict=None*, *ghost_recv_dict=None*, *starttime=0*, *processor=0*, *numproc=1*, *number_of_full_nodes=None*, *number_of_full_triangles=None*, *ghost_layer_width=2*, ***kwargs*)

Instantiate a shallow water domain.

Parameters

- **`coordinates`** – vertex locations for the mesh
- **`vertices`** – vertex indices defining the triangles of the mesh
- **`boundary`** – boundaries of the mesh

Methods

`__init__`([coordinates, vertices, boundary, ...])	Instantiate a shallow water domain.
`add_quantity`(name, *args, **kwargs)	Add values to a named quantity
`apply_fractional_steps`()	
`apply_protection_against_isolated_degenerate_timesteps`()	
`backup_conserved_quantities`()	
`balance_deep_and_shallow`()	Compute linear combination between stage as computed by gradient-limiters limiting using w, and stage computed by gradient-limiters limiting using h (h-limiter).
`boundary_statistics`([quantities, tags])	Output statistics about boundary forcing at each timestep
`build_tagged_elements_dictionary`(*args, **kwargs)	
`centroid_norm`(quantity, normfunc)	Calculate the norm of the centroid values of a specific quantity, using normfunc.
`check_integrity`()	Run integrity checks on shallow water domain.
`compute_boundary_flows`()	Compute boundary flows at current timestep.
`compute_flux_update_frequency`()	Update the 'flux_update_frequency' and 'update_extrapolate' variables Used to control updating of fluxes / extrapolation for 'local-time-stepping'
`compute_fluxes`()	Compute fluxes and timestep suitable for all volumes in domain.
`compute_forcing_flows`()	Compute flows in and out of domain due to forcing terms.
`compute_forcing_terms`()	If there are any forcing functions driving the system they should be defined in Domain subclass and appended to the list self.forcing_terms
`compute_total_volume`()	Compute total volume (m^3) of water in entire domain
`conserved_values_to_evolved_values`(q_cons, ...)	Needs to be overridden by Domain subclass
`create_quantity_from_expression`(expression)	Create new quantity from other quantities using arbitrary expression.

continues on next page

Table 1 – continued from previous page

distribute_to_vertices_and_edges()	Call correct module function
distribute_using_edge_limiter()	Distribution from centroids to edges specific to the SWW eqn.
distribute_using_vertex_limiter()	Distribution from centroids to vertices specific to the SWW equation.
dump_triangulation([filename])	Get vertex coordinates, partition full and ghost triangles based on self.tri_full_flag
evolve([yieldstep, outputstep, finaltime, ...])	Evolve method from Domain class.
evolve_one_euler_step(yieldstep, finaltime)	One Euler Time Step Q^{n+1} = E(h) Q^n
evolve_one_rk2_step(yieldstep, finaltime)	One 2nd order RK timestep Q^{n+1} = 0.5 Q^n + 0.5 E(h)^2 Q^n
evolve_one_rk3_step(yieldstep, finaltime)	One 3rd order RK timestep Q^(1) = 3/4 Q^n + 1/4 E(h)^2 Q^n (at time t^n + h/2) Q^{n+1} = 1/3 Q^n + 2/3 E(h) Q^(1) (at time t^{n+1})
evolve_to_end([finaltime])	Iterate evolve all the way to the end.
extrapolate_second_order_sw()	Fast version of extrapolation from centroids to edges
get_CFL()	get CFL
get_algorithm_parameters()	Get the standard parameter that are currently set (as a dictionary)
get_area(*args, **kwargs)	
get_areas(*args, **kwargs)	
get_beta()	Get default beta for limiting.
get_boundary_flux_integral()	Compute the boundary flux integral.
get_boundary_polygon(*args, **kwargs)	
get_boundary_tags(*args, **kwargs)	
get_centroid_coordinates(*args, **kwargs)	
get_centroid_transmissive_bc()	Get value of centroid_transmissive_bc flag.
get_cfl()	get CFL
get_compute_fluxes_method()	Get method for computing fluxes.
get_conserved_quantities(vol_id[, vertex, edge])	Get conserved quantities at volume vol_id.
get_datadir()	
get_datetime()	Retrieve datetime corresponding to current timestamp wrt to domain timezone
get_disconnected_triangles(*args, **kwargs)	
get_distribute_to_vertices_and_edges_method()	Get method for distribute_to_vertices_and_edges.
get_edge_midpoint_coordinate(*args, **kwargs)	
get_edge_midpoint_coordinates(*args, **kwargs)	
get_energy_through_cross_section(polyline[, ...])	Obtain average energy head [m] across specified cross section.
get_evolve_max_timestep()	Set default max_timestep for evolving.
get_evolve_min_timestep()	Set default max_timestep for evolving.

continues on next page

Table 1 – continued from previous page

get_evolve_starttime()	
get_evolved_quantities(vol_id[, vertex, edge])	Get evolved quantities at volume vol_id.
get_extent(*args, **kwargs)	
get_flow_algorithm()	Get method used for timestepping and spatial discretisation
get_flow_through_cross_section(polyline[, ...])	Get the total flow through an arbitrary poly line.
get_fractional_step_volume_integral()	Compute the integrated flows from fractional steps.
get_full_centroid_coordinates(*args, **kwargs)	
get_full_nodes(*args, **kwargs)	
get_full_triangles(*args, **kwargs)	
get_full_vertex_coordinates(*args, **kwargs)	
get_georeference(*args, **kwargs)	
get_global_name()	
get_interpolation_object(*args, **kwargs)	
get_intersecting_segments(*args, **kwargs)	
get_inv_tri_map()	
get_lone_vertices(*args, **kwargs)	
get_maximum_inundation_elevation([indices, ...])	Return highest elevation where h > 0
get_maximum_inundation_location([indices])	Return location of highest elevation where h > 0
get_minimum_allowed_height()	
get_minimum_storable_height()	
get_name()	
get_nodes(*args, **kwargs)	
get_normal(*args, **kwargs)	
get_number_of_full_triangles(*args, **kwargs)	
get_number_of_nodes(*args, **kwargs)	
get_number_of_triangles(*args, **kwargs)	
get_number_of_triangles_per_node(*args, **kwargs)	

continues on next page

Table 1 – continued from previous page

get_quantity(name[, location, indices])	Get pointer to quantity object.
get_quantity_names()	Get a list of all the quantity names that this domain is aware of.
get_radii(*args, **kwargs)	
get_relative_time()	Set internal relative time
get_starttime()	
get_store()	Get whether data saved to sww file.
get_store_centroids()	Get whether data saved to sww file.
get_tagged_elements(*args, **kwargs)	
get_time()	Get the absolute model time (seconds).
get_timestep()	get current timestep (seconds).
get_timestepping_method()	
get_timezone()	Retrieve current domain timezone
get_tri_map()	
get_triangle_containing_point(*args, **kwargs)	
get_triangles(*args, **kwargs)	
get_triangles_and_vertices_per_node(*args, ...)	
get_triangles_inside_polygon(*args, **kwargs)	
get_unique_vertices(*args, **kwargs)	
get_using_discontinuous_elevation()	Return boolean indicating whether algorithm is using dicontinuous elevation
get_vertex_coordinate(*args, **kwargs)	
get_vertex_coordinates(*args, **kwargs)	
get_water_volume()	
get_wet_elements([indices, minimum_height])	Return indices for elements where h > minimum_allowed_height
initialise_storage()	Create and initialise self.writer object for storing data.
log_operator_timestepping_statistics()	
maximum_quantity(name, *args, **kwargs)	max of values to a named quantity
minimum_quantity(name, *args, **kwargs)	min of values to a named quantity
print_algorithm_parameters()	Print the standard parameters that are curently set (as a dictionary)
print_boundary_statistics([quantities, tags])	
print_operator_statistics()	

continues on next page

Table 1 – continued from previous page

print_operator_timestepping_statistics()	
print_statistics(*args, **kwargs)	
print_timestepping_statistics(*args, **kwargs)	Print time stepping statistics
print_volumetric_balance_statistics()	
protect_against_infinitesimal_and_negative_heights()	Clean up the stage and momentum values to ensure non-negative heights
quantity_statistics([precision])	Return string with statistics about quantities for printing or logging
report_cells_with_small_local_timestep([...])	Convenience function to print the locations of cells with a small local timestep.
report_water_volume_statistics([verbose, ...])	Compute the volume, boundary flux integral, fractional step volume integral, and their difference
saxpy_conserved_quantities(a, b)	
set_CFL([cfl])	Set CFL parameter, warn if greater than 2.0
set_beta(beta)	Shorthand to assign one constant value [0,2] to all limiters.
set_betas(beta_w, beta_w_dry, beta_uh, ...)	Assign beta values in the range [0,2] to all limiters.
set_boundary(boundary_map)	Associate boundary objects with tagged boundary segments.
set_centroid_transmissive_bc(flag)	Set behaviour of the transmissive boundary condition, namely calculate the BC using the centroid value of neighbouring cell or the calculated edge value.
set_cfl([cfl])	Set CFL parameter, warn if greater than 2.0
set_checkpointing([checkpoint, ...])	Set up checkpointing.
set_compute_fluxes_method([flag])	Set method for computing fluxes.
set_datadir(name)	
set_default_order(n)	Set default (spatial) order to either 1 or 2.
set_distribute_to_vertices_and_edges_method([flag])	Set method for computing fluxes.
set_evolve_max_timestep(max_timestep)	Set default max_timestep for evolving.
set_evolve_min_timestep(min_timestep)	Set default min_timestep for evolving.
set_evolve_starttime(time)	
set_extrapolate_velocity([flag])	Extrapolation routine uses momentum by default, can change to velocity extrapolation which seems to work better.
set_flow_algorithm([flag])	Set combination of slope limiting and time stepping
set_fractional_step_operator(operator)	
set_georeference(*args, **kwargs)	
set_gravity_method()	Gravity method is determined by the compute_fluxes_method This is now not used, as gravity is combine in the compute_fluxes method

continues on next page

Table 1 – continued from previous page

`set_institution`(institution)	
`set_local_extrapolation_and_flux_updating`([L])	Use local flux and extrapolation updating
`set_low_froude`([low_froude])	For low Froude problems the standard flux calculations can lead to excessive damping.
`set_maximum_allowed_speed`(maximum_allowed_speed)	Set the maximum particle speed that is allowed in water shallower than minimum_allowed_height.
`set_minimum_allowed_height`(...)	Set minimum depth that will be recognised in the numerical scheme.
`set_minimum_storable_height`(...)	Set the minimum depth that will be written to an SWW file.
`set_name`([name, timestamp])	Assign a name to this simulation.
`set_points_file_block_line_size`(...)	
`set_quantities_to_be_monitored`(q[, polygon, ...])	Specify which quantities will be monitored for extrema.
`set_quantities_to_be_stored`(q)	Specify which quantities will be stored in the SWW file.
`set_quantity`(name, *args, **kwargs)	Set values for named quantity
`set_quantity_vertices_dict`(quantity_dict)	Set values for named quantities.
`set_relative_time`([time])	Set internal relative time
`set_sloped_mannings_function`([flag])	Set mannings friction function to use the sloped wetted area.
`set_starttime`([timestamp])	Set the starttime for the evolution
`set_store`([flag])	Set whether data saved to sww file.
`set_store_centroids`([flag])	Set whether centroid data is saved to sww file.
`set_store_vertices_smoothly`([flag, reduction])	Decide whether vertex values should be stored smoothly (one value per vertex) or uniquely as computed in the model (False).
`set_store_vertices_uniquely`([flag, reduction])	Decide whether vertex values should be stored uniquely as computed in the model (True) or whether they should be reduced to one value per vertex using self.reduction (False).
`set_tag_region`(*args, **kwargs)	Set quantities based on a regional tag.
`set_time`([time])	Set the model time (seconds).
`set_timestepping_method`(timestepping_method)	
`set_timezone`([tz])	Set timezone for domain
`set_use_edge_limiter`([flag])	Extrapolation routine uses vertex values by default, for limiting, can change to edge limiting which seems to work better in some cases.
`set_use_kinematic_viscosity`([flag])	
`set_use_optimise_dry_cells`([flag])	Try to optimize calculations where region is dry
`set_using_discontinuous_elevation`([flag])	Set flag to show whether compute flux algorithm is allowing discontinuous elevation.
`set_zone`(zone)	Set zone for domain.
`statistics`(*args, **kwargs)	
`store_timestep`()	Store time dependent quantities and time.

continues on next page

Table 1 – continued from previous page

sww_merge(*args, **kwargs)	Merge all the sub domain sww files into a global sww file
timestepping_statistics([track_speeds, ...])	Return string with time stepping statistics for printing or logging
update_boundary()	Go through list of boundary objects and update boundary values for all conserved quantities on boundary.
update_boundary_old()	Go through list of boundary objects and update boundary values for all conserved quantities on boundary.
update_boundary_old_2()	Go through list of boundary objects and update boundary values for all conserved quantities on boundary.
update_centroids_of_momentum_from_velocity()	Calculate the centroid value of x and y momentum from height and velocities
update_centroids_of_velocities_and_height()	Calculate the centroid values of velocities and height based on the values of the quantities stage and x and y momentum
update_conserved_quantities()	Update vectors of conserved quantities using previously computed fluxes and specified forcing functions.
update_extrema()	Update extrema if requested by set_quantities_to_be_monitored.
update_ghosts([quantities])	We must send the information from the full cells and receive the information for the ghost cells We have a list with ghosts expecting updates
update_other_quantities()	There may be a need to calculates some of the other quantities based on the new values of conserved quantities
update_special_conditions()	
update_timestep(yieldstep, finaltime)	Calculate the next timestep to take
volumetric_balance_statistics()	Create volumetric balance report suitable for printing or logging.
write_boundary_statistics([quantities, tags])	
write_time([track_speeds])	

10.3.2 anuga.Quantity

class anuga.**Quantity**(*domain*, *vertex_values=None*, *name=None*, *register=False*)

Class Quantity - Implements values at each triangular element

__init__(*domain*, *vertex_values=None*, *name=None*, *register=False*)

Create Quantity object

Parameters

- **domain** – Associated domain structure. Required.
- **vertex_values** – N x 3 array of values at each vertex for each element. Default None

- **name** (*str*) – Provides a way to refer to a created quantity
- **register** – Register a quantity

Usage:

```
>>> Quantity(domain, name="newQ", register=True)
```

If vertex_values are None Create array of zeros compatible with domain. Otherwise check that it is compatible with dimensions of domain. Otherwise raise an exception

For Quantities that need to be saved during checkpointing, set register=True. Registered Quantities can be found in the dictionary domain.quantities (note, other Quantities can exist).

Methods

`__init__`(domain[, vertex_values, name, register])	Create Quantity object
`backup_centroid_values`()	
`bound_vertices_below_by_constant`(bound)	
`bound_vertices_below_by_quantity`(quantity)	
`compute_gradients`()	
`compute_local_gradients`()	
`extrapolate_first_order`()	Extrapolate conserved quantities from centroid to vertices and edges for each volume using first order scheme.
`extrapolate_second_order`()	
`extrapolate_second_order_and_limit_by_edge`()	
`extrapolate_second_order_and_limit_by_vertex`()	
`get_beta`()	Get default beta value for limiting
`get_extremum_index`([mode, indices])	Return index for maximum or minimum value of quantity (on centroids)
`get_gradients`()	Provide gradients.
`get_integral`([full_only, region, indices])	Compute the integral of quantity across entire domain, or over a region.
`get_interpolated_values`(interpolation_points)	Get values at interpolation points
`get_maximum_index`([indices])	See get extreme index for details
`get_maximum_location`([indices])	Return location of maximum value of quantity (on centroids)
`get_maximum_value`([indices])	Return maximum value of quantity (on centroids)
`get_minimum_index`([indices])	See get extreme index for details
`get_minimum_location`([indices])	Return location of minimum value of quantity (on centroids)
`get_minimum_value`([indices])	Return minimum value of quantity (on centroids)
`get_name`()	

continues on next page

Table 2 – continued from previous page

get_values([interpolation_points, location, ...])	Get values for quantity
get_vertex_values([xy, smooth, precision])	Return vertex values like an OBJ format i.e. one value per node.
interpolate()	Compute interpolated values at edges and centroid Pre-condition: vertex_values have been set
interpolate_from_edges_to_vertices()	
interpolate_from_vertices_to_edges()	
interpolate_old()	Compute interpolated values at edges and centroid Pre-condition: vertex_values have been set
limit()	
limit_edges_by_all_neighbours()	
limit_edges_by_neighbour()	
limit_vertices_by_all_neighbours()	
maximum(other)	Max of self with anything that could populate a quantity
minimum(other)	Max of self with anything that could populate a quantity
plot_quantity([filename, draw])	
save_centroid_data_to_csv([filename])	
save_data_to_dem([filename])	
save_to_array([cellsize, NODATA_value, ...])	Interpolate quantity to an array
saxpy_centroid_values(a, b)	
set_beta(beta)	Set default beta value for limiting
set_boundary_values([numeric])	Set boundary values
set_boundary_values_from_edges()	Set boundary values by simply extrapolating from the cells
set_name([name])	
set_values([numeric, quantity, function, ...])	Set values for quantity based on different sources.
set_values_from_array(values[, location, ...])	Set values for quantity
set_values_from_constant(X, location, ...)	Set quantity values from specified constant X
set_values_from_file(filename, ...[, ...])	Set quantity based on arbitrary points in a points file using attribute_name selects name of attribute present in file.
set_values_from_function(f[, location, ...])	Set values for quantity using specified function
set_values_from_geospatial_data(...[, ...])	Set values based on geo referenced geospatial data object.
set_values_from_lat_long_grid_file(filename)	Read Digital model from the following ASCII format (.asc or .grd)
set_values_from_points(points, values, ...)	Set quantity values from arbitray data points using fit_interpolate.fit

continues on next page

Table 2 – continued from previous page

`set_values_from_quantity`(q, location, ...)	Set quantity values from specified quantity instance q
`set_values_from_utm_grid_file`(filename[, ...])	Read Digital Elevation model from the following ASCII format (.asc, .grd or .dem)
`set_values_from_utm_raster`(raster[, ...])	
`set_vertex_values`(A[, indices, use_cache, ...])	Set vertex values for all unique vertices based on input array A which has one entry per unique vertex, i.e. one value for each row in array self.domain.nodes.
`smooth_vertex_values`([use_cache, verbose])	Smooths vertex values.
`update`(timestep)	

Attributes

`counter`

10.3.3 anuga.Region

class anuga.**Region**(*domain*, *indices=None*, *polygon=None*, *center=None*, *radius=None*, *line=None*, *poly=None*, *expand_polygon=False*, *verbose=False*)

Object which defines a region within the domain

__init__(*domain*, *indices=None*, *polygon=None*, *center=None*, *radius=None*, *line=None*, *poly=None*, *expand_polygon=False*, *verbose=False*)

Create a Region object

Parameters

- **domain** – Region must be defined wrt a domain
- **indices** – Define the region by triangle IDs
- **polygon** – List of [x,y] points to define region
- **center** – point [x,y] which defines the centre of a circle
- **radius** – radius of a circle which defines a region
- **line** – List of [x,y] points defining a polyline
- **poly** – An old argument which was used to define a polyline or polygon
- **expand_polygon** – If set true, then calculation of intersection of polygon with triangles based on vertices, otherwise based just on centroids
- **verbose** – Set to True for more verbose output

Setup region (defined by indices, polygon or center/radius). Useful in defining where to apply certain operations

Methods

__init__(domain[, indices, polygon, center, ...])	Create a Region object
get_indices([full_only])	
get_type()	
plot_region([filename])	
set_verbose([verbose])	

10.4 Boundary Conditions

anuga.Reflective_boundary([domain])	Reflective boundary condition object
anuga.Dirichlet_boundary([dirichlet_values])	Dirichlet boundary returns constant values for the conserved quantities
anuga.Time_space_boundary([domain, ...])	Time and spatially dependent boundary returns values for the conserved quantities as a function of time and space.
anuga.Flather_external_stage_zero_velocity_boundary([...])	Boundary condition based on a Flather type approach
anuga.Transmissive_n_momentum_zero_t_momentum_set_stage_boundary([...])	Boundary condition object that returns transmissive normal momentum and sets stage
anuga.Transmissive_boundary([domain])	Transmissive boundary returns same conserved quantities as those present in its neighbour volume.
anuga.File_boundary(filename, domain[, ...])	The File_boundary reads values for the conserved quantities from an sww NetCDF file, and returns interpolated values at the midpoints of each associated boundary segment.
anuga.Field_boundary(filename, domain[, ...])	Set boundary from given field.
anuga.Time_stage_zero_momentum_boundary([...])	Time dependent boundary returns values for stage conserved quantities as a function of time.
anuga.Transmissive_stage_zero_momentum_boundary([...])	BC where stage is same as neighbour volume and momentum to zero.
anuga.Transmissive_momentum_set_stage_boundary([...])	Boundary condition object that returns transmissive momentum and sets stage
anuga.Time_boundary([domain, function, ...])	Time dependent boundary returns values for the conserved quantities as a function of time.

10.4.1 anuga.Reflective_boundary

class anuga.**Reflective_boundary**(*domain=None*)

Reflective boundary condition object

Reflective boundary returns same conserved quantities as those present in its neighbour volume but with normal momentum reflected.

__init__(*domain=None*)

Create boundary condition object

Parameters
domain – domain on which to apply BC

Example:

Set all the tagged boundaries to use the Reflective boundaries

```
>>> domain = anuga.rectangular_cross_domain(10, 10)
>>> BC = anuga.Reflective_boundary(domain)
>>> domain.set_boundary({'left': BC, 'right': BC, 'top': BC, 'bottom': BC})
```

Methods

`__init__`([domain])	Create boundary condition object
`evaluate`(vol_id, edge_id)	Calculate BC associated to specified edge
`evaluate_segment`(domain, segment_edges)	Apply BC on the boundary edges defined by segment_edges
`get_boundary_values`([t])	
`get_time`()	

10.4.2 anuga.Dirichlet_boundary

class anuga.**Dirichlet_boundary**(*dirichlet_values=None*)

Dirichlet boundary returns constant values for the conserved quantities

__init__(*dirichlet_values=None*)

Methods

`__init__`([dirichlet_values])	
`evaluate`([vol_id, edge_id])	
`evaluate_segment`(domain, segment_edges)	Evaluate boundary condition at edges of a domain in a list defined by segment_edges
`get_boundary_values`([t])	
`get_time`()	

10.4.3 anuga.Time_space_boundary

class anuga.**Time_space_boundary**(*domain=None*, *function=None*, *default_boundary=None*, *verbose=False*)

Time and spatially dependent boundary returns values for the conserved quantities as a function of time and space. Must specify domain to get access to model time and a function of t,x,y which must return conserved quantities at specified time and location.

Example:

B = Time_space_boundary(domain,
function=lambda t,x,y: [(60<t<3660)*2, 0, 0])

This will produce a boundary condition with is a 2m high square wave starting 60 seconds into the simulation and lasting one hour. Momentum applied will be 0 at all times.

__init__(*domain=None*, *function=None*, *default_boundary=None*, *verbose=False*)

Methods

__init__([domain, function, ...])	
evaluate([vol_id, edge_id])	
evaluate_segment([domain, segment_edges])	Evaluate boundary condition at edges of a domain in a list defined by segment_edges
get_boundary_values([t])	
get_time()	

10.4.4 anuga.Flather_external_stage_zero_velocity_boundary

class anuga.**Flather_external_stage_zero_velocity_boundary**(*domain=None*, *function=None*)

Boundary condition based on a Flather type approach

Setting the external stage with a function, and a zero external velocity,

The idea is similar (but not identical) to that described on page 239 of the following article:

```
Article{blayo05,
Title       = {Revisiting open boundary conditions from the point of view of␣
↪characteristic variables},
Author      = {Blayo, E. and Debreu, L.},
Journal     = {Ocean Modelling},
Year        = {2005},
Pages       = {231-252},
Volume      = {9},
}
```

Approach

1. The external (outside boundary) stage is set with a function, the external velocity is zero, the internal stage and velocity are taken from the domain values.

2. Some 'characteristic like' variables are computed, depending on whether the flow is incoming or outgoing. See Blayo and Debreu (2005)
3. The boundary conserved quantities are computed from these characteristic like variables

This has been useful as a 'weakly reflecting' boundary when the stage should be approximately specified but allowed to adapt to outgoing waves.

`__init__`(*domain=None*, *function=None*)

Create boundary condition object.

Parameters

- **`domain`** – The domain on which to apply boundary condition
- **`function`** – Function to apply on the boundary

Example:

```
def waveform(t):
    return sea_level + normalized_amplitude/cosh(t-25)**2

Bf = Flather_external_stage_zero_velocity_boundary(domain, waveform)
```

Methods

`__init__`([domain, function])	Create boundary condition object.
`evaluate`(vol_id, edge_id)	
`evaluate_segment`(domain, segment_edges)	Applied in vectorized form for speed.
`get_boundary_values`([t])	
`get_time`()	

10.4.5 anuga.Transmissive_n_momentum_zero_t_momentum_set_stage_boundary

class anuga.**`Transmissive_n_momentum_zero_t_momentum_set_stage_boundary`**(*domain=None*, *function=None*, *default_boundary=0.0*)

Bounday condition object that returns transmissive normal momentum and sets stage

Returns the same normal momentum as that present in neighbour volume edge. Zero out the tangential momentum. Sets stage by specifying a function f of time which may either be a vector function or a scalar function

`__init__`(*domain=None*, *function=None*, *default_boundary=0.0*)

Create boundary condition object.

Parameters

- **`domain`** – domain on which to apply BC
- **`function`** – function to set stage
- **`default_boundary`** (*`float`*) –

Example: Set all the tagged boundaries to use the BC

```
>>> domain = anuga.rectangular_cross_domain(10, 10)
>>> def waveform(t):
>>>     return sea_level + normalized_amplitude/cosh(t-25)**2
>>> BC = anuga.Transmissive_n_momentum_zero_t_momentum_set_stage_
↪boundary(domain, waveform)
>>> domain.set_boundary({'left': BC, 'right': BC, 'top': BC, 'bottom': BC})
```

Methods

__init__([domain, function, default_boundary])	Create boundary condition object.
evaluate(vol_id, edge_id)	Transmissive_n_momentum_zero_t_momentum_set_stage_boundary return the edge momentum values of the volume they serve.
evaluate_segment(domain, segment_edges)	Apply BC on the boundary edges defined by segment_edges
get_boundary_values([t])	
get_time()	

10.4.6 anuga.Transmissive_boundary

class anuga.**Transmissive_boundary**(*domain=None*)

Transmissive boundary returns same conserved quantities as those present in its neighbour volume.

Underlying domain must be specified when boundary is instantiated

__init__(*domain=None*)

Methods

__init__([domain])	
evaluate(vol_id, edge_id)	Transmissive boundaries return the edge values of the volume they serve.
evaluate_segment(domain, segment_edges)	Evaluate boundary condition at edges of a domain in a list defined by segment_edges
get_boundary_values([t])	
get_time()	

10.4.7 anuga.File_boundary

`class anuga.File_boundary`(*filename*, *domain*, *time_thinning=1*, *time_limit=None*, *boundary_polygon=None*, *default_boundary=None*, *use_cache=False*, *verbose=False*)

The File_boundary reads values for the conserved quantities from an sww NetCDF file, and returns interpolated values at the midpoints of each associated boundary segment. Time dependency is interpolated linearly.

Assumes that file contains a time series and possibly also spatial info. See docstring for File_function in util.py for details about admissible file formats

File boundary must read and interpolate from *smoothed* version as stored in sww and cannot work with the discontinuous triangles.

Example: Bf = File_boundary('source_file.sww', domain)

Note that the resulting solution history is not exactly the same as if the models were coupled as there is no feedback into the source model.

Optional keyword argument default_boundary must be either None or an instance of class descending from class Boundary. This will be used in case model time exceeds that available in the underlying data.

`__init__`(*filename*, *domain*, *time_thinning=1*, *time_limit=None*, *boundary_polygon=None*, *default_boundary=None*, *use_cache=False*, *verbose=False*)

Methods

`__init__`(filename, domain[, time_thinning, ...])	
`evaluate`([vol_id, edge_id])	Return linearly interpolated values based on domain time at midpoint of segment defined by vol_id and edge_id.
`evaluate_segment`([domain, segment_edges])	Evaluate boundary condition at edges of a domain in a list defined by segment_edges
`get_boundary_values`([t])	
`get_time`()	

10.4.8 anuga.Field_boundary

`class anuga.Field_boundary`(*filename*, *domain*, *mean_stage=0.0*, *time_thinning=1*, *time_limit=None*, *boundary_polygon=None*, *default_boundary=None*, *use_cache=False*, *verbose=False*)

Set boundary from given field.

Given field is represented in an sww file containing values for stage, xmomentum and ymomentum.

Optionally, the user can specify mean_stage to offset the stage provided in the sww file.

This function is a thin wrapper around the generic File_boundary. The difference between the File_boundary and Field_boundary is only that the Field_boundary will allow you to change the level of the stage height when you read in the boundary condition. This is very useful when running different tide heights in the same area as you need only to convert one boundary condition to a SWW file, ideally for tide height of 0 m (saving disk space). Then you can use Field_boundary to read this SWW file and change the stage height (tide) on the fly depending on the scenario.

`__init__`(*filename*, *domain*, *mean_stage=0.0*, *time_thinning=1*, *time_limit=None*, *boundary_polygon=None*, *default_boundary=None*, *use_cache=False*, *verbose=False*)

Constructor

Parameters

- **`filename`** – Name of sww file containing stage and x/ymomentum
- **`domain`** – pointer to shallow water domain for which the boundary applies
- **`mean_stage`** – The mean water level which will be added to stage derived from the boundary condition
- **`time_thinning`** – Will set how many time steps from the sww file read in will be interpolated to the boundary.
- **`default_boundary`** – This will be used in case model time exceeds that available in the underlying data.
- **`time_limit`** –
- **`boundary_polygon`** –
- **`use_cache`** – True if caching is to be used.
- **`verbose`** – True if this method is to be verbose.

For example if the sww file has 1 second time steps and is 24 hours in length it has 86400 time steps. If you set time_thinning to 1 it will read all these steps. If you set it to 100 it will read every 100th step eg only 864 step. This parameter is very useful to increase the speed of a model run that you are setting up and testing.

Methods

`__init__`(filename, domain[, mean_stage, ...])	Constructor
`evaluate`([vol_id, edge_id])	Calculate 'field' boundary results.
`evaluate_segment`([domain, segment_edges])	Evaluate boundary condition at edges of a domain in a list defined by segment_edges
`get_boundary_values`([t])	
`get_time`()	

10.4.9 anuga.Time_stage_zero_momentum_boundary

class anuga.**Time_stage_zero_momentum_boundary**(*domain=None*, *function=None*, *default_boundary=None*, *verbose=False*)

Time dependent boundary returns values for stage conserved quantities as a function of time. Must specify domain to get access to model time and a function of t which must return conserved stage quantities as a function time.

Example:

B = Time_stage_zero_momentum_boundary(domain,
function=lambda t: (60<t<3660)*2)

This will produce a boundary condition with is a 2m high square wave starting 60 seconds into the simulation and lasting one hour. Momentum applied will be 0 at all times.

__init__(*domain=None*, *function=None*, *default_boundary=None*, *verbose=False*)

Methods

__init__([domain, function, ...])	
evaluate([vol_id, edge_id])	
evaluate_segment(domain, segment_edges)	Evaluate boundary condition at edges of a domain in a list defined by segment_edges
get_boundary_values([t])	
get_time()	

10.4.10 anuga.Transmissive_stage_zero_momentum_boundary

class anuga.**Transmissive_stage_zero_momentum_boundary**(*domain=None*)

BC where stage is same as neighbour volume and momentum to zero.

Underlying domain must be specified when boundary is instantiated

__init__(*domain=None*)

Instantiate a Transmissive (zero momentum) boundary.

Methods

__init__([domain])	Instantiate a Transmissive (zero momentum) boundary.
evaluate(vol_id, edge_id)	Calculate transmissive (zero momentum) results.
evaluate_segment([domain, segment_edges])	Evaluate boundary condition at edges of a domain in a list defined by segment_edges
get_boundary_values([t])	
get_time()	

10.4.11 anuga.Transmissive_momentum_set_stage_boundary

class anuga.**Transmissive_momentum_set_stage_boundary**(*domain=None*, *function=None*)

Bounday condition object that returns transmissive momentum and sets stage

Returns same momentum conserved quantities as those present in its neighbour volume. Sets stage by specifying a function f of time which may either be a vector function or a scalar function

__init__(*domain=None*, *function=None*)

Create boundary condition object.

Parameters

- **domain** – domain on which to apply BC
- **function** – function to set stage

Example: Set all the tagged boundaries to use the

```
>>> domain = anuga.rectangular_cross_domain(10, 10)
>>> def waveform(t):
>>>     return sea_level + normalized_amplitude/cosh(t-25)**2
>>> BC = anuga.Transmissive_momentum_set_stage_boundary(domain, waveform)
>>> domain.set_boundary({'left': BC, 'right': BC, 'top': BC, 'bottom': BC})
```

Methods

`__init__`([domain, function])	Create boundary condition object.
`evaluate`(vol_id, edge_id)	Transmissive momentum set stage boundaries return the edge momentum values of the volume they serve.
`evaluate_segment`([domain, segment_edges])	Evaluate boundary condition at edges of a domain in a list defined by segment_edges
`get_boundary_values`([t])	
`get_time`()	

10.4.12 anuga.Time_boundary

class anuga.**Time_boundary**(*domain=None*, *function=None*, *default_boundary=None*, *verbose=False*)

Time dependent boundary returns values for the conserved quantities as a function of time. Must specify domain to get access to model time and a function of t which must return conserved quantities as a function time.

Example:

B = Time_boundary(domain,
function=lambda t: [(60<t<3660)*2, 0, 0])

This will produce a boundary condition with is a 2m high square wave starting 60 seconds into the simulation and lasting one hour. Momentum applied will be 0 at all times.

__init__(*domain=None*, *function=None*, *default_boundary=None*, *verbose=False*)

Methods

`__init__`([domain, function, ...])	
`evaluate`([vol_id, edge_id])	
`evaluate_segment`(domain, segment_edges)	Evaluate boundary condition at edges of a domain in a list defined by segment_edges
`get_boundary_values`([t])	
`get_time`()	

10.5 Structures

Culverts and Bridges

anuga.Inlet_operator(domain, region[, Q, ...])	Inlet Operator - add water to an inlet.
anuga.Boyd_box_operator(domain, losses, width)	Culvert flow - transfer water from one rectangular box to another.
anuga.Boyd_pipe_operator(domain, losses[, ...])	Culvert flow - transfer water from one location to another via a circular pipe culvert.
anuga.Weir_orifice_trapezoid_operator(...[, ...])	Culvert flow - transfer water from one trapezoidal section to another.
anuga.Internal_boundary_operator(domain, ...)	The internal_boundary_function must accept 2 input arguments (hw, tw). It returns Q: - hw will always be the stage (or energy) at the enquiry_point[0] - tw will always be the stage (or energy) at the enquiry_point[1] - If flow is from hw to tw, then Q should be positive, otherwise Q should be negative.

10.5.1 anuga.Inlet_operator

`class anuga.`**`Inlet_operator`**(*domain*, *region*, *Q=0.0*, *velocity=None*, *zero_velocity=False*, *default=0.0*, *description=None*, *label=None*, *logging=False*, *verbose=False*)

Inlet Operator - add water to an inlet. Sets up the geometry of problem

Inherit from this class (and overwrite discharge_routine method for specific subclasses)

Input: domain, Two points

`__init__`(*domain*, *region*, *Q=0.0*, *velocity=None*, *zero_velocity=False*, *default=0.0*, *description=None*, *label=None*, *logging=False*, *verbose=False*)

Inlet Operator - add water to a domain via an inlet.

Parameters

- **`domain`** – Specify domain
- **`region`** – Apply Inlet flow over a region (which can be a Region, Polygon or line)
- **`Q`** – function(t) or scalar discharge (m^3/s)
- **`velocity`** – Optional [u,v] to set velocity of applied discharge
- **`zero_velocity`** – If set to True, velocity of inlet region set to 0
- **`default`** – If outside time domain of the Q function, use this default discharge
- **`description`** – Describe the Inlet_operator
- **`label`** – Give Inlet_operator a label (name)
- **`verbose`** – Provide verbose output

Example:

```
>>> inflow_region  = anuga.Region(domain, center=[0.0,0.0], radius=1.0)
>>> inflow = anuga.Inlet_operator(domain, inflow_region, Q = lambda t : 1 + 0.
↪5*math.sin(t/60))
```

Methods

`__init__`(domain, region[, Q, velocity, ...])	Inlet Operator - add water to a domain via an inlet.
`activate_logging`()	
`get_Q`()	
`get_applied_Q`()	
`get_default`(t[, err_msg])	Call get_default only if exception Modeltime_too_late(msg) has been raised
`get_inlet`()	
`get_time`()	
`get_timestep`()	
`get_total_applied_volume`()	
`log_timestepping_statistics`()	
`parallel_safe`()	By default an operator is not parallel safe
`print_statistics`()	
`print_timestepping_statisitics`()	
`print_timestepping_statistics`()	
`set_Q`(Q)	
`set_default`([default])	Either leave default as None or change it into a function
`set_label`([label])	
`set_logging`([flag])	
`statistics`()	
`timestepping_statistics`()	
`update_Q`(t)	Allowing local modifications of Q

Attributes

`counter`	

10.5.2 anuga.Boyd_box_operator

class anuga.**Boyd_box_operator**(*domain, losses, width, height=None, barrels=1.0, blockage=0.0, z1=0.0, z2=0.0, end_points=None, exchange_lines=None, enquiry_points=None, invert_elevations=None, apron=0.1, manning=0.013, enquiry_gap=0.0, smoothing_timescale=0.0, use_momentum_jet=True, use_velocity_head=True, description=None, label=None, structure_type='boyd_box', logging=False, verbose=False*)

Culvert flow - transfer water from one rectangular box to another. Sets up the geometry of problem

This is the base class for culverts. Inherit from this class (and overwrite compute_discharge method for specific subclasses)

Input: minimum arguments
domain, losses (scalar, list or dictionary of losses), width (= height if height not given)

__init__(*domain, losses, width, height=None, barrels=1.0, blockage=0.0, z1=0.0, z2=0.0, end_points=None, exchange_lines=None, enquiry_points=None, invert_elevations=None, apron=0.1, manning=0.013, enquiry_gap=0.0, smoothing_timescale=0.0, use_momentum_jet=True, use_velocity_head=True, description=None, label=None, structure_type='boyd_box', logging=False, verbose=False*)

Create a box culvert using Boyd flow algorithm

Parameters

- **domain** – Culvert applied to this domain
- **losses** – Losses
- **width** – Width of culvert
- **height** – height of culvert
- **barrels** – Number of barrels
- **blockage** – Set between 0.0 - 1.0 Set to 1.0 to close off culvert
- **z1** – Elevation of end of Culvert
- **z2** – Elevation of other end of Culvert
- **end_points** – [[x1,y1], [x2,y2]] of centre of ends of culvert
- **exchange_lines** – [[[x1,y1], [x2,y2]], [[x1,y1], [x2,y2]]] list of two lines defining ends of culvert
- **enquiry_points** – [[x1,y1], [x2,y2]] location of enquiry points
- **invert_elevations** – [e1, e2] invert elevations of culvert inlets
- **apron** –
- **manning** –
- **enquiry_gap** –

- **smoothing_timescale** –
- **use_momentum_jet** –
- **use_velocity_head** –
- **description** –
- **label** –
- **structure_type** –
- **logging** –
- **verbose** –

Methods

__init__(domain, losses, width[, height, ...])	Create a box culvert using Boyd flow algorithm
activate_logging()	
discharge_routine()	Procedure to determine the inflow and outflow inlets.
get_culvert_apron()	
get_culvert_barrels()	
get_culvert_blockage()	
get_culvert_diameter()	
get_culvert_height()	
get_culvert_length()	
get_culvert_slope()	
get_culvert_width()	
get_culvert_z1()	
get_culvert_z2()	
get_enquiry_depths()	
get_enquiry_elevations()	
get_enquiry_invert_elevations()	
get_enquiry_positions()	
get_enquiry_specific_energys()	
get_enquiry_speeds()	

continues on next page

Table 3 – continued from previous page

get_enquiry_stages()	
get_enquiry_total_energys()	
get_enquiry_velocity_heads()	
get_enquiry_velocitys()	
get_enquiry_water_depths()	
get_enquiry_xmoms()	
get_enquiry_xvelocitys()	
get_enquiry_ymoms()	
get_enquiry_yvelocitys()	
get_inlets()	
get_master_proc()	
get_time()	
get_timestep()	
log_timestepping_statistics()	
parallel_safe()	By default an operator is not parallel safe
print_statistics()	
print_timestepping_statistics()	
set_culvert_barrels(barrels)	
set_culvert_blockage(blockage)	
set_culvert_height(height)	
set_culvert_width(width)	
set_culvert_z1(z1)	
set_culvert_z2(z2)	
set_label([label])	
set_logging([flag])	
statistics()	

continues on next page

Table 3 – continued from previous page

`timestepping_statistics()`	

Attributes

`counter`	

10.5.3 anuga.Boyd_pipe_operator

class anuga.**Boyd_pipe_operator**(*domain*, *losses*, *diameter=None*, *barrels=1.0*, *blockage=0.0*, *z1=0.0*, *z2=0.0*, *end_points=None*, *exchange_lines=None*, *enquiry_points=None*, *invert_elevations=None*, *apron=0.1*, *manning=0.013*, *enquiry_gap=0.2*, *smoothing_timescale=0.0*, *use_momentum_jet=True*, *use_velocity_head=True*, *description=None*, *label=None*, *structure_type='boyd_pipe'*, *logging=False*, *verbose=False*)

Culvert flow - transfer water from one location to another via a circular pipe culvert. Sets up the geometry of problem

This is the base class for culverts. Inherit from this class (and overwrite compute_discharge method for specific subclasses)

Input: Two points, pipe_size (diameter), mannings_rougness,

__init__(*domain*, *losses*, *diameter=None*, *barrels=1.0*, *blockage=0.0*, *z1=0.0*, *z2=0.0*, *end_points=None*, *exchange_lines=None*, *enquiry_points=None*, *invert_elevations=None*, *apron=0.1*, *manning=0.013*, *enquiry_gap=0.2*, *smoothing_timescale=0.0*, *use_momentum_jet=True*, *use_velocity_head=True*, *description=None*, *label=None*, *structure_type='boyd_pipe'*, *logging=False*, *verbose=False*)

exchange_lines define the input lines for each inlet.

If end_points = None, then the culvert_vector is calculated in the directions from the centre of echange_line[0] to centre of exchange_line[1}

If end_points != None, then culvert_vector is unit vector in direction end_point[1] - end_point[0]

Methods

`__init__`(domain, losses[, diameter, ...])	exchange_lines define the input lines for each inlet.
`activate_logging()`	
`discharge_routine()`	Procedure to determine the inflow and outflow inlets.
`get_culvert_apron()`	
`get_culvert_barrels()`	
`get_culvert_blockage()`	

continues on next page

Table 4 – continued from previous page

get_culvert_diameter()
get_culvert_height()
get_culvert_length()
get_culvert_slope()
get_culvert_width()
get_culvert_z1()
get_culvert_z2()
get_enquiry_depths()
get_enquiry_elevations()
get_enquiry_invert_elevations()
get_enquiry_positions()
get_enquiry_specific_energys()
get_enquiry_speeds()
get_enquiry_stages()
get_enquiry_total_energys()
get_enquiry_velocity_heads()
get_enquiry_velocitys()
get_enquiry_water_depths()
get_enquiry_xmoms()
get_enquiry_xvelocitys()
get_enquiry_ymoms()
get_enquiry_yvelocitys()
get_inlets()
get_master_proc()
get_time()

continues on next page

Table 4 – continued from previous page

`get_timestep()`	
`log_timestepping_statistics()`	
`parallel_safe()`	By default an operator is not parallel safe
`print_statistics()`	
`print_timestepping_statistics()`	
`set_culvert_barrels(barrels)`	
`set_culvert_blockage(blockage)`	
`set_culvert_height(height)`	
`set_culvert_width(width)`	
`set_culvert_z1(z1)`	
`set_culvert_z2(z2)`	
`set_label([label])`	
`set_logging([flag])`	
`statistics()`	
`timestepping_statistics()`	

Attributes

`counter`

10.5.4 anuga.Weir_orifice_trapezoid_operator

class anuga.**Weir_orifice_trapezoid_operator**(*domain, losses, width, height=None, barrels=1.0, blockage=0.0, z1=0.0, z2=0.0, end_points=None, exchange_lines=None, enquiry_points=None, invert_elevations=None, apron=0.1, manning=0.013, enquiry_gap=0.0, smoothing_timescale=0.0, use_momentum_jet=True, use_velocity_head=True, description=None, label=None, structure_type='weir_orifice_trapezoid', logging=False, verbose=False*)

Culvert flow - transfer water from one trapezoidal section to another. Sets up the geometry of problem

This is the base class for culverts. Inherit from this class (and overwrite compute_discharge method for specific subclasses)

Input: minimum arguments
domain, losses (scalar, list or dictionary of losses), width (= height if height not given)

`__init__`(*domain*, *losses*, *width*, *height=None*, *barrels=1.0*, *blockage=0.0*, *z1=0.0*, *z2=0.0*, *end_points=None*, *exchange_lines=None*, *enquiry_points=None*, *invert_elevations=None*, *apron=0.1*, *manning=0.013*, *enquiry_gap=0.0*, *smoothing_timescale=0.0*, *use_momentum_jet=True*, *use_velocity_head=True*, *description=None*, *label=None*, *structure_type='weir_orifice_trapezoid'*, *logging=False*, *verbose=False*)

exchange_lines define the input lines for each inlet.

If end_points = None, then the culvert_vector is calculated in the directions from the centre of echange_line[0] to centre of exchange_line[1}

If end_points != None, then culvert_vector is unit vector in direction end_point[1] - end_point[0]

Methods

`__init__`(domain, losses, width[, height, ...])	exchange_lines define the input lines for each inlet.
`activate_logging`()	
`discharge_routine`()	Procedure to determine the inflow and outflow inlets.
`get_culvert_apron`()	
`get_culvert_barrels`()	
`get_culvert_blockage`()	
`get_culvert_diameter`()	
`get_culvert_height`()	
`get_culvert_length`()	
`get_culvert_slope`()	
`get_culvert_width`()	
`get_culvert_z1`()	
`get_culvert_z2`()	
`get_enquiry_depths`()	
`get_enquiry_elevations`()	
`get_enquiry_invert_elevations`()	
`get_enquiry_positions`()	
`get_enquiry_specific_energys`()	

continues on next page

Table 5 – continued from previous page

get_enquiry_speeds()	
get_enquiry_stages()	
get_enquiry_total_energys()	
get_enquiry_velocity_heads()	
get_enquiry_velocitys()	
get_enquiry_water_depths()	
get_enquiry_xmoms()	
get_enquiry_xvelocitys()	
get_enquiry_ymoms()	
get_enquiry_yvelocitys()	
get_inlets()	
get_master_proc()	
get_time()	
get_timestep()	
log_timestepping_statistics()	
parallel_safe()	By default an operator is not parallel safe
print_statistics()	
print_timestepping_statistics()	
set_culvert_barrels(barrels)	
set_culvert_blockage(blockage)	
set_culvert_height(height)	
set_culvert_width(width)	
set_culvert_z1(z1)	
set_culvert_z2(z2)	
set_label([label])	
set_logging([flag])	

continues on next page

Table 5 – continued from previous page

`statistics()`
`timestepping_statistics()`

Attributes

`counter`

10.5.5 anuga.Internal_boundary_operator

class anuga.**Internal_boundary_operator**(*domain*, *internal_boundary_function*, *width=1.0*, *height=1.0*, *end_points=None*, *exchange_lines=None*, *enquiry_points=None*, *invert_elevation=None*, *apron=0.0*, *enquiry_gap=0.0*, *use_velocity_head=False*, *zero_outflow_momentum=False*, *force_constant_inlet_elevations=True*, *smoothing_timescale=0.0*, *compute_discharge_implicitly=True*, *description=None*, *label=None*, *structure_type='internal_boundary'*, *logging=False*, *verbose=True*)

The internal_boundary_function must accept 2 input arguments (hw, tw). It returns Q: - hw will always be the stage (or energy) at the enquiry_point[0] - tw will always be the stage (or energy) at the enquiry_point[1] - If flow is from hw to tw, then Q should be positive, otherwise Q

should be negative

def internal_boundary_function(hw, tw):
# Compute Q here from headwater hw and tailwater hw return(Q)

smoothing_timescale>0. can be used to make Q vary more slowly

__init__(*domain*, *internal_boundary_function*, *width=1.0*, *height=1.0*, *end_points=None*, *exchange_lines=None*, *enquiry_points=None*, *invert_elevation=None*, *apron=0.0*, *enquiry_gap=0.0*, *use_velocity_head=False*, *zero_outflow_momentum=False*, *force_constant_inlet_elevations=True*, *smoothing_timescale=0.0*, *compute_discharge_implicitly=True*, *description=None*, *label=None*, *structure_type='internal_boundary'*, *logging=False*, *verbose=True*)

exchange_lines define the input lines for each inlet.

If end_points = None, then the culvert_vector is calculated in the directions from the centre of echange_line[0] to centre of exchange_line[1}

If end_points != None, then culvert_vector is unit vector in direction end_point[1] - end_point[0]

Methods

`__init__`(domain, internal_boundary_function)	exchange_lines define the input lines for each inlet.
`activate_logging()`	
`discharge_routine()`	Both implicit and explicit methods available The former seems more stable and more accurate (in at least some cases).
`discharge_routine_explicit()`	Procedure to determine the inflow and outflow inlets.
`discharge_routine_implicit()`	Uses semi-implicit discharge estimation:
`get_culvert_apron()`	
`get_culvert_barrels()`	
`get_culvert_blockage()`	
`get_culvert_diameter()`	
`get_culvert_height()`	
`get_culvert_length()`	
`get_culvert_slope()`	
`get_culvert_width()`	
`get_culvert_z1()`	
`get_culvert_z2()`	
`get_enquiry_depths()`	
`get_enquiry_elevations()`	
`get_enquiry_invert_elevations()`	
`get_enquiry_positions()`	
`get_enquiry_specific_energys()`	
`get_enquiry_speeds()`	
`get_enquiry_stages()`	
`get_enquiry_total_energys()`	
`get_enquiry_velocity_heads()`	
`get_enquiry_velocitys()`	

continues on next page

Table 6 – continued from previous page

get_enquiry_water_depths()	
get_enquiry_xmoms()	
get_enquiry_xvelocitys()	
get_enquiry_ymoms()	
get_enquiry_yvelocitys()	
get_inlets()	
get_master_proc()	
get_time()	
get_timestep()	
log_timestepping_statistics()	
parallel_safe()	By default an operator is not parallel safe
print_statistics()	
print_timestepping_statistics()	
set_culvert_barrels(barrels)	
set_culvert_blockage(blockage)	
set_culvert_height(height)	
set_culvert_width(width)	
set_culvert_z1(z1)	
set_culvert_z2(z2)	
set_label([label])	
set_logging([flag])	
statistics()	
timestepping_statistics()	

Attributes

`counter`

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX