ANUGA v1.0 User Manual

Release 1.0beta_7013

Geoscience Australia and the Australian National University

Wednesday 13th May, 2009, Ten minutes past Four in the afternoon

Geoscience Australia
Email: ole.nielsen@ga.gov.au

This work was produced at Geoscience Australia and the Australian National University funded by the Commonwealth of Australia. Neither the Australian Government, the Australian National University, Geoscience Australia nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the Australian Government, Geoscience Australia or the Australian National University. The views and opinions of authors expressed herein do not necessarily state or reflect those of the Australian Government, Geoscience Australia or the Australian National University, and shall not be used for advertising or product endorsement purposes.

This document does not convey a warranty, express or implied, of merchantability or fitness for a particular purpose.

ANUGA v1.0

Manual typeset with LaTeX

Credits:

- **ANUGA** v1.0 was developed and is maintained by Stephen Roberts, Ole Nielsen, Duncan Gray and Jane Sexton.

License:

- **ANUGA** v1.0 is freely available and distributed under the terms of the GNU General Public Licence.

Acknowledgments:

- John Jakeman, Rudy van Drie, Ted Rigby, Joaquim Luis, Nils Goseberg, William Power, Petar Milevski, Trevor Dhu, Linda Stals, Matt Hardy, Jack Kelly and Christopher Zoppou who contributed to this project at various times.
- A stand alone visualiser (anuga_viewer) based on Open-scene-graph was developed by Darran Edmundson.

- The mesh generator engine was written by Jonathan Richard Shewchuk and made freely available under the following license. See source code `triangle.c` for more details on the origins of this code. The license reads

```
/*****************************************************************************/
/*                                                                           */
/*      888888888        ,o,                          / 888                  */
/*         888    88o88o  "    o8888o  88o8888o o88888o 888  o88888o        */
/*         888    888    888       88b 888  888 888 888 888 d888  88b        */
/*         888    888    888  o88^o888 888  888 "88888" 888 8888oo888        */
/*         888    888    888 C888  888 888  888  /      888 q888             */
/*         888    888    888  "88o^888 888  888 Cb      888  "88oooo"        */
/*                                              "8oo8D                       */
/*                                                                           */
/*  A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator.      */
/*  (triangle.c)                                                             */
/*                                                                           */
/*  Version 1.6                                                              */
/*  July 28, 2005                                                            */
/*                                                                           */
/*  Copyright 1993, 1995, 1997, 1998, 2002, 2005                             */
/*  Jonathan Richard Shewchuk                                                */
/*  2360 Woolsey #H                                                          */
/*  Berkeley, California  94705-1927                                         */
/*  jrs@cs.berkeley.edu                                                      */
/*                                                                           */
/*  This program may be freely redistributed under the condition that the    */
/*    copyright notices (including this entire header and the copyright      */
/*    notice printed when the ‘-h’ switch is selected) are not removed, and  */
/*    no compensation is received.  Private, research, and institutional     */
/*    use is free.  You may distribute modified versions of this code UNDER  */
/*    THE CONDITION THAT THIS CODE AND ANY MODIFICATIONS MADE TO IT IN THE   */
/*    SAME FILE REMAIN UNDER COPYRIGHT OF THE ORIGINAL AUTHOR, BOTH SOURCE   */
/*    AND OBJECT CODE ARE MADE FREELY AVAILABLE WITHOUT CHARGE, AND CLEAR    */
/*    NOTICE IS GIVEN OF THE MODIFICATIONS.  Distribution of this code as    */
/*    part of a commercial system is permissible ONLY BY DIRECT ARRANGEMENT  */
/*    WITH THE AUTHOR.  (If you are not directly supplying this code to a    */
/*    customer, and you are instead telling them how they can obtain it for  */
/*    free, then you are not required to make any arrangement with me.)      */
/*****************************************************************************/
```

- Pmw is a toolkit for building high-level compound widgets in Python using the Tkinter module. Parts of Pmw have been encorpoated into the graphical mesh generator. The license for Pmw reads

Abstract

ANUGA v1.0 is a hydrodynamic modelling tool that allows users to model realistic flow problems in complex 2D geometries. Examples include dam breaks or the effects of natural hazards such as riverine flooding, storm surges and tsunami.

The user must specify a study area represented by a mesh of triangular cells, the topography and bathymetry, frictional resistance, initial values for water level (called *stage* within **ANUGA** v1.0), boundary conditions and forces such as rainfall, stream flows, windstress or pressure gradients if applicable.

ANUGA v1.0 tracks the evolution of water depth and horizontal momentum within each cell over time by solving the shallow water wave equation governing equation using a finite-volume method.

ANUGA v1.0 also incorporates a mesh generator that allows the user to set up the geometry of the problem interactively as well as tools for interpolation and surface fitting, and a number of auxiliary tools for visualising and interrogating the model output.

Most **ANUGA** v1.0 components are written in the object-oriented programming language Python and most users will interact with **ANUGA** v1.0 by writing small Python programs based on the **ANUGA** v1.0 library functions. Computationally intensive components are written for efficiency in C routines working directly with the Numerical Python structures.

CONTENTS

CHAPTER

ONE

Introduction

1.1 Purpose

The purpose of this user manual is to introduce the new user to the inundation software, describe what it can do and give step-by-step instructions for setting up and running hydrodynamic simulations. The stable release of **ANUGA** v1.0 and this manual are available on sourceforge at http://sourceforge.net/projects/anuga. A snapshot of work in progress is available through the **ANUGA** v1.0 software repository at https://datamining.anu.edu.au/svn/ga/anuga_core where the more adventurous reader might like to go.

1.2 Scope

This manual covers only what is needed to operate the software after installation and configuration. It does not includes instructions for installing the software or detailed API documentation, both of which will be covered in separate publications and by documentation in the source code.

1.3 Audience

Readers are assumed to be familiar with the Python Programming language and its object oriented approach. Python tutorials include http://docs.python.org/tut, http://www.sthurlow.com/python, and http://datamining.anu.edu.au/~ole/work/teaching/ctac2006/exercise1.pdf.

Readers also need to have a general understanding of scientific modelling, as well as enough programming experience to adapt the code to different requirements.

CHAPTER

TWO

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** v1.0 to help simulate the impact of these hazards.

The core of **ANUGA** v1.0 is the fluid dynamics module, called `shallow_water`, 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** v1.0 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** v1.0 is also capable of modelling hydraulic jumps due to the ability of the finite-volume method to accommodate discontinuities in the solution/footnoteWhile **ANUGA** v1.0 works with discontinuities in the conserved quantities stage, xmomentum and ymomentum, it does not allow discontinuities in the bed elevation.

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 forcing terms that may drive the system such as rain_fall, abstraction of water, wind stress or atmospheric pressure gradients. Gravity and frictional resistance from the different terrains in the model are represented by predefined forcing terms. See section 5.5 for details on forcing terms 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** v1.0 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 the Numerical Python structures. The animation tool developed for **ANUGA** v1.0 is based on OpenSceneGraph, an Open Source Software (OSS) component allowing high level interaction with sophisticated graphics primitives. See [nielsen2005] for more background on **ANUGA** v1.0 .

CHAPTER

THREE

Restrictions and limitations on **ANUGA** v1.0

Although a powerful and flexible tool for hydrodynamic modelling, **ANUGA** v1.0 has a number of limitations that any potential user need to be aware of. They are

- The mathematical model is the 2D shallow water wave equation. As such it cannot resolve vertical convection and consequently not breaking waves or 3D turbulence (e.g. vorticity).
- All spatial coordinates are assumed to be UTM (meters). As such, ANUGA is unsuitable for modelling flows in areas larger than one UTM zone (6 degrees wide).
- Fluid is assumed to be inviscid - i.e. no kinematic viscosity included.
- The finite volume is a very robust and flexible numerical technique, but it is not the fastest method around. If the geometry is sufficiently simple and if there is no need for wetting or drying, a finite-difference method may be able to solve the problem faster than **ANUGA** v1.0 .
- Frictional resistance is implemented using Manning's formula, but **ANUGA** v1.0 has not yet been fully validated in regard to bottom roughness

CHAPTER

FOUR

Getting Started

This section is designed to assist the reader to get started with **ANUGA** v1.0 by working through some examples. Two examples are discussed; the first is a simple example to illustrate many of the concepts, and the second is a more realistic example.

4.1 A Simple Example

4.1.1 Overview

What follows is a discussion of the structure and operation of a script called 'runup.py'.

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 rain_fall, 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
```

4.1.2 Outline of the Program

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

1. Sets up a triangular mesh.
2. Sets certain parameters governing the mode of operation of the model-specifying, for instance, where to store the model output.
3. Inputs various quantities describing physical measurements, such as the elevation, to be specified at each mesh point (vertex).

4. Sets up the boundary conditions.
5. Carries out the evolution of the model through a series of time steps and outputs the results, providing a results file that can be visualised.

4.1.3 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
#--------------------------------------------------------------------------

from anuga.abstract_2d_finite_volumes.mesh_factory import rectangular_cross
from anuga.shallow_water import Domain
from anuga.shallow_water import Reflective_boundary
from anuga.shallow_water import Dirichlet_boundary
from anuga.shallow_water import Time_boundary
from anuga.shallow_water import Transmissive_boundary

from math import sin, pi, exp

#--------------------------------------------------------------------------
# Setup computational domain
#--------------------------------------------------------------------------

points, vertices, boundary = rectangular_cross(10, 10) # Basic mesh

domain = Domain(points, vertices, boundary) # Create domain
domain.set_name('runup')                    # Output to file runup.sww
domain.set_datadir('.')                     # Use current directory for output


#--------------------------------------------------------------------------
# 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', -.4)            # Constant negative initial stage


#--------------------------------------------------------------------------
# Setup boundary conditions
#--------------------------------------------------------------------------
```

```
Br = Reflective_boundary(domain)      # Solid reflective wall
Bt = Transmissive_boundary(domain)    # Continue all values on boundary
Bd = Dirichlet_boundary([-0.2,0.,0.]) # Constant boundary values
Bw = Time_boundary(domain=domain,     # Time dependent boundary
                   f=lambda t: [(.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()
```

4.1.4 Establishing the Mesh

The first task is to set up the triangular mesh to be used for the scenario. This is carried out through the statement:

```
points, vertices, boundary = rectangular_cross(10, 10)
```

The function `rectangular_cross` is imported from a module `mesh_factory` defined elsewhere. (**ANUGA** v1.0 also contains several other schemes that can be used for setting up meshes, but we shall not discuss these.) The above assignment sets up a 10×10 rectangular mesh, triangulated in a regular way. The assignment

```
points, vertices, boundary = rectangular_cross(m, n)
```

returns:

- a list `points` giving the coordinates of each mesh point,
- a list `vertices` specifying the three vertices of each triangle, and
- a dictionary `boundary` that stores the edges on the boundary and associates each with one of the symbolic tags `'left'`, `'right'`, `'top'` or `'bottom'`.

(For more details on symbolic tags, see page 12.)

An example of a general unstructured mesh and the associated data structures `points`, `vertices` and `boundary` is given in Section 5.1.

4.1.5 Initialising the Domain

These variables are then used to set up a data structure `domain`, through the assignment:

```
domain = Domain(points, vertices, boundary)
```

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 now specifies that the quantities `stage`, `xmomentum` and `ymomentum` are to be stored:

```
domain.set_quantities_to_be_stored(['stage', 'xmomentum',
'ymomentum'])
```

4.1.6 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 equation 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`.

4.1.6.1 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 f(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 `f` is specified, the quantity `elevation` is assigned through the simple statement:

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

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

4.1.6.2 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.

4.1.6.3 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', -.4)
```

which specifies that the surface level is set to a height of -0.4, i.e. 0.4 units (m) 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.) See Section 5.3 for more details.

4.1.7 Boundary Conditions

The boundary conditions are specified as follows:

```
Br = Reflective_boundary(domain)

Bt = Transmissive_boundary(domain)

Bd = Dirichlet_boundary([0.2,0.,0.])

Bw = Time_boundary(domain=domain,
            f=lambda t: [(0.1*sin(t*2*pi)-0.3), 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 as present 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.

- **Transmissive boundary** Returns same conserved quantities as those present in its neighbour volume. This is one way of modelling outflow from a domain, but it should be used with caution if flow is not steady state as replication of momentum at the boundary may cause numerical instabilities propagating into the domain and eventually causing ANUGA to crash. If this occurs, consider using e.g. a Dirichlet boundary condition with a stage value less than the elevation at the boundary.
- **Dirichlet boundary** Specifies constant values for stage, x-momentum and y-momentum at the boundary.
- **Time boundary** Like a Dirichlet boundary but with behaviour 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`; however, the example given in Section 4.6 illustrates another way of assigning the values, by means of the function `create_mesh_from_regions`.

These variables store the data determining the mesh as follows. (You may find that the example given in Section 5.1 helps to clarify the following discussion, even though that example is a *non-rectangular* mesh.)

- The variable `points` stores a list of 2-tuples giving the coordinates of the mesh points.
- The variable `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]`, `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` 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 4.6 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 assoctiated with a boundary conditing 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})
  File "X:\inundation\sandpits\onielsen\anuga_core\source\anuga\abstract_2d_finite_volumes\doma
    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.

The reader may wish to experiment by varying the choice of boundary types for one or more of the boundaries. (In the case of `Bd` and `Bw`, the three arguments in each case represent the `stage`, x-momentum and y-momentum, respectively.)

```
Bw = Time_boundary(domain=domain,
                   f=lambda t: [(0.1*sin(t*2*pi)-0.3), 0.0, 0.0])
```

4.1.8 Evolution

The final statement

```
for t in domain.evolve(yieldstep = 0.1, duration = 4.0):
    print domain.timestepping_statistics()
```

causes the configuration of the domain to 'evolve', over a series of steps indicated by the values of `yieldstep` and `duration`, which can be altered as required. The value of `yieldstep` controls the time interval between successive model outputs. Behind the scenes more time steps are generally taken.

4.1.9 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 `animate` (see Section A.2) visualisation package to generate a visual display. See Section 6.1 (page 61) for more on NetCDF and other file formats.

The following is a listing of the screen output seen by the user when this example is run:

```
$ python runup.py
Time = 0.0000, steps=0 (0)
Time = 0.1000, delta t in [0.01373568, 0.01683588], steps=7 (7)
Time = 0.2000, delta t in [0.01203520, 0.01357912], steps=8 (8)
Time = 0.3000, delta t in [0.01144234, 0.01193508], steps=9 (9)
Time = 0.4000, delta t in [0.01141301, 0.01152065], steps=9 (9)
...
```

4.2 How to Run the Code

The code can be run in various ways:

- from a Windows or Unix command line as in `python runup.py`
- within the Python IDLE environment
- within emacs
- within Windows, by double-clicking the `runup.py` file.

4.3 Exploring the Model Output

The following figures are screenshots from the **ANUGA** v1.0 visualisation tool `animate`. Figure 4.1 shows the domain with water surface as specified by the initial condition, $t = 0$. Figure 4.2 shows later snapshots for $t = 2.3$ and $t = 4$ where the system has been evolved and the wave is encroaching on the previously dry bed. All figures are screenshots from an interactive animation tool called animate which is part of **ANUGA** v1.0 and distributed as in the package anuga_viewer. Animate is described in more detail is Section A.2.

Figure 4.1: Runup example viewed with the ANUGA viewer

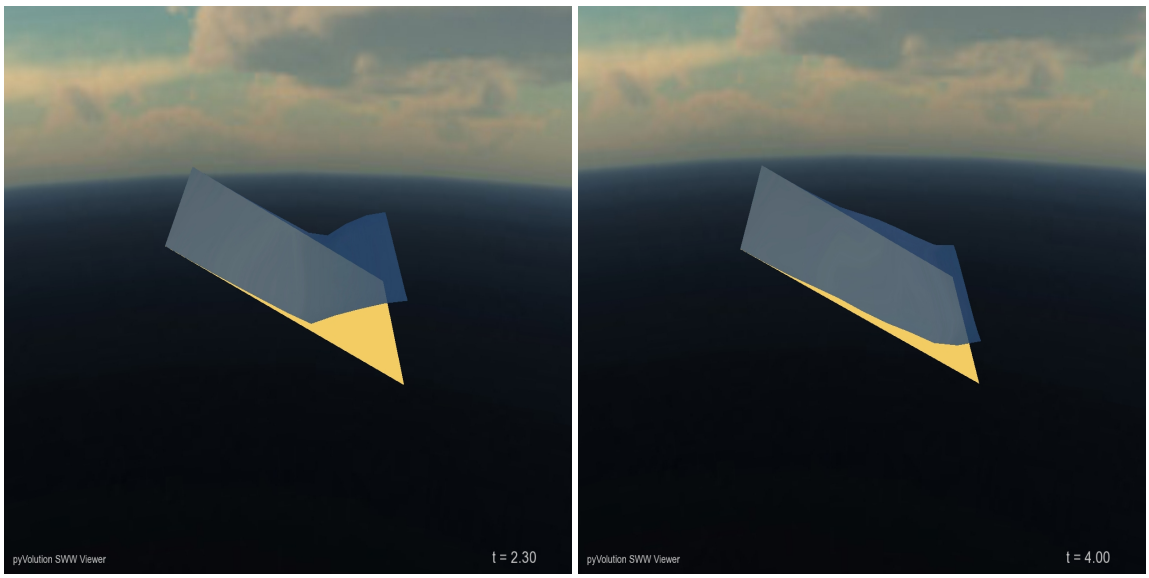

Figure 4.2: Runup example viewed with ANGUA viewer

4.4 A slightly more complex example

4.4.1 Overview

The next example is about waterflow in a channel with varying boundary conditions and more complex topograhies. These examples build on the concepts introduced through the ‘runup.py’ in Section 4.1. The example will be built up through three progressively more complex scripts.

4.4.2 Overview

As in the case of ‘runup.py’, the actions carried out by the program can be organised according to this outline:

1. Set up a triangular mesh.
2. Set certain parameters governing the mode of operation of the model—specifying, for instance, where to store the model output.
3. Set up initial conditions for various quantities such as the elevation, to be specified at each mesh point (vertex).
4. Set up the boundary conditions.
5. Carry out the evolution of the model through a series of time steps and output the results, providing a results file that can be visualised.

4.4.3 The Code

Here is the code for the first version of the channel flow ‘channel1.py’:

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

Water flowing down a channel
"""

#------------------------------------------------------------------------------
# Import necessary modules
#------------------------------------------------------------------------------
from anuga.abstract_2d_finite_volumes.mesh_factory import rectangular_cross
from anuga.shallow_water import Domain
from anuga.shallow_water import Reflective_boundary
from anuga.shallow_water import Dirichlet_boundary


#------------------------------------------------------------------------------
# Setup computational domain
#------------------------------------------------------------------------------
points, vertices, boundary = rectangular_cross(10, 5,
                                               len1=10.0, len2=5.0) # Mesh

domain = Domain(points, vertices, boundary)  # Create domain
domain.set_name('channel1')                  # Output name


#------------------------------------------------------------------------------
# Setup initial conditions
#------------------------------------------------------------------------------
```

```
def topography(x,y):
    return -x/10                            # linear bed slope

domain.set_quantity('elevation', topography) # Use function for elevation
domain.set_quantity('friction', 0.01)        # Constant friction
domain.set_quantity('stage',                 # Dry bed
                    expression='elevation + 0.0')


#-----------------------------------------------------------------------------
# Setup boundary conditions
#-----------------------------------------------------------------------------
Bi = Dirichlet_boundary([0.4, 0, 0])         # Inflow
Br = Reflective_boundary(domain)             # Solid reflective wall

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


#-----------------------------------------------------------------------------
# Evolve system through time
#-----------------------------------------------------------------------------
for t in domain.evolve(yieldstep = 0.2, finaltime = 40.0):
    print domain.timestepping_statistics()
```

In discussing the details of this example, we follow the outline given above, discussing each major step of the code in turn.

4.4.4 Establishing the Mesh

In this example we use a similar simple structured triangular mesh as in `runup.py` for simplicity, but this time we will use a symmetric one and also change the physical extent of the domain. The assignment

```
    points, vertices, boundary = rectangular_cross(m, n,
                                                   len1=length, len2=width)
```

returns a m x n mesh similar to the one used in the previous example, except that now the extent in the x and y directions are given by the value of `length` and `width` respectively.

Defining m and n in terms of the extent as in this example provides a convenient way of controlling the resolution: By defining dx and dy to be the desired size of each hypothenuse in the mesh we can write the mesh generation as follows:

```
length = 10.
width = 5.
dx = dy = 1           # Resolution: Length of subdivisions on both axes

points, vertices, boundary = rectangular_cross(int(length/dx), int(width/dy),
                                               len1=length, len2=width)
```

which yields a mesh of length=10m, width=5m with 1m spacings. To increase the resolution, as we will later in this example, one merely decrease the values of dx and dy.

The rest of this script is as in the previous example.

4.5 Model Output

The following figure is a screenshot from the **ANUGA** v1.0 visualisation tool `animate` of output from this example.

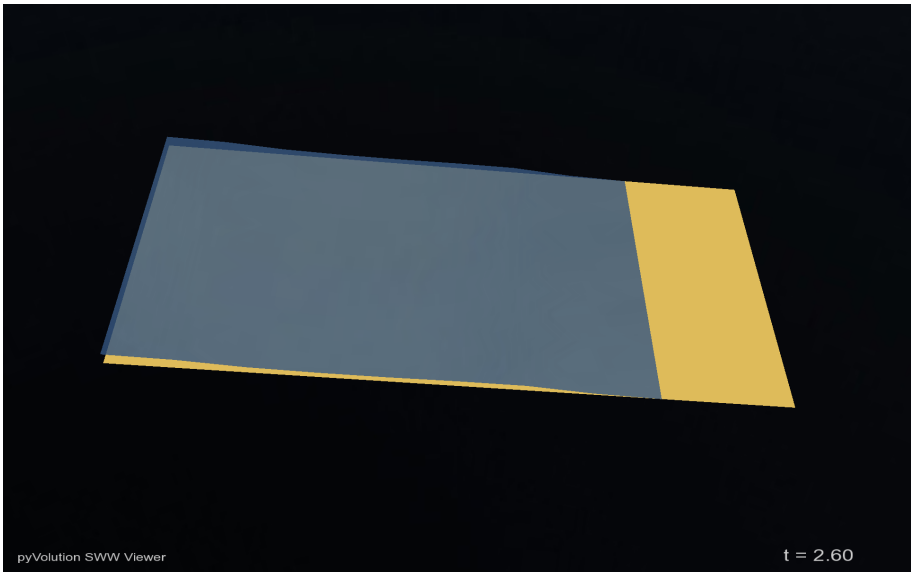

Figure 4.3: Simple channel example viewed with the ANUGA viewer.

4.5.1 Changing boundary conditions on the fly

Here is the code for the second version of the channel flow 'channel2.py':

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

Water flowing down a channel with changing boundary conditions
"""

#------------------------------------------------------------------------------
# Import necessary modules
#------------------------------------------------------------------------------
from anuga.abstract_2d_finite_volumes.mesh_factory import rectangular_cross
from anuga.shallow_water import Domain
from anuga.shallow_water import Reflective_boundary
from anuga.shallow_water import Dirichlet_boundary
from anuga.shallow_water import Time_boundary


#------------------------------------------------------------------------------
# Setup computational domain
#------------------------------------------------------------------------------
length = 10.
width = 5.
dx = dy = 1           # Resolution: Length of subdivisions on both axes

points, vertices, boundary = rectangular_cross(int(length/dx), int(width/dy),
                                               len1=length, len2=width)
```

```
domain = Domain(points, vertices, boundary)
domain.set_name('channel2')                 # Output name


#-------------------------------------------------------------------------
# Setup initial conditions
#-------------------------------------------------------------------------
def topography(x,y):
    return -x/10                            # linear bed slope

domain.set_quantity('elevation', topography) # Use function for elevation
domain.set_quantity('friction', 0.01)        # Constant friction
domain.set_quantity('stage',
                    expression='elevation')  # Dry initial condition


#-------------------------------------------------------------------------
# Setup boundary conditions
#-------------------------------------------------------------------------
Bi = Dirichlet_boundary([0.4, 0, 0])         # Inflow
Br = Reflective_boundary(domain)             # Solid reflective wall
Bo = Dirichlet_boundary([-5, 0, 0])          # Outflow

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


#-------------------------------------------------------------------------
# Evolve system through time
#-------------------------------------------------------------------------
for t in domain.evolve(yieldstep = 0.2, finaltime = 40.0):
    print domain.timestepping_statistics()

    if domain.get_quantity('stage').\
           get_values(interpolation_points=[[10, 2.5]]) > 0:
        print 'Stage > 0: Changing to outflow boundary'
        domain.set_boundary({'right': Bo})
```

This example differs from the first version in that a constant outflow boundary condition has been defined

```
    Bo = Dirichlet_boundary([-5, 0, 0]) # Outflow
```

and that it is applied to the right hand side boundary when the water level there exceeds 0m.

```
for t in domain.evolve(yieldstep = 0.2, finaltime = 40.0):
    domain.write_time()

    if domain.get_quantity('stage').get_values(interpolation_points=[[10, 2.5]]) > 0:
        print 'Stage > 0: Changing to outflow boundary'
        domain.set_boundary({'right': Bo})
```

The if statement in the timestepping loop (evolve) gets the quantity `stage` and obtain the interpolated value at the point (10m, 2.5m) which is on the right boundary. If the stage exceeds 0m a message is printed and the old boundary

condition at tag 'right' is replaced by the outflow boundary using the method

```
domain.set_boundary({'right': Bo})
```

This type of dynamically varying boundary could for example be used to model the breakdown of a sluice door when water exceeds a certain level.

4.5.2 Output

The text output from this example looks like this

```
...
Time = 15.4000, delta t in [0.03789902, 0.03789916], steps=6 (6)
Time = 15.6000, delta t in [0.03789896, 0.03789908], steps=6 (6)
Time = 15.8000, delta t in [0.03789891, 0.03789903], steps=6 (6)
Stage > 0: Changing to outflow boundary
Time = 16.0000, delta t in [0.02709050, 0.03789898], steps=6 (6)
Time = 16.2000, delta t in [0.03789892, 0.03789904], steps=6 (6)
...
```

4.5.3 Flow through more complex topograhies

Here is the code for the third version of the channel flow 'channel3.py':

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

Water flowing down a channel with more complex topography
"""

#------------------------------------------------------------------------------
# Import necessary modules
#------------------------------------------------------------------------------
from anuga.abstract_2d_finite_volumes.mesh_factory import rectangular_cross
from anuga.shallow_water import Domain
from anuga.shallow_water import Reflective_boundary
from anuga.shallow_water import Dirichlet_boundary
from anuga.shallow_water import Time_boundary


#------------------------------------------------------------------------------
# Setup computational domain
#------------------------------------------------------------------------------
length = 40.
width = 5.
dx = dy = .1           # Resolution: Length of subdivisions on both axes

points, vertices, boundary = rectangular_cross(int(length/dx), int(width/dy),
                                               len1=length, len2=width)
domain = Domain(points, vertices, boundary)
domain.set_name('channel3')                 # Output name
print domain.statistics()
```

```
#--------------------------------------------------------------------------
# Setup initial conditions
#--------------------------------------------------------------------------
def topography(x,y):
    """Complex topography defined by a function of vectors x and y
    """

    z = -x/10

    N = len(x)
    for i in range(N):

        # Step
        if 10 < x[i] < 12:
            z[i] += 0.4 - 0.05*y[i]

        # Constriction
        if 27 < x[i] < 29 and y[i] > 3:
            z[i] += 2

        # Pole
        if (x[i] - 34)**2 + (y[i] - 2)**2 < 0.4**2:
            z[i] += 2

    return z


domain.set_quantity('elevation', topography)  # Use function for elevation
domain.set_quantity('friction', 0.01)         # Constant friction
domain.set_quantity('stage',
                    expression='elevation')   # Dry initial condition


#--------------------------------------------------------------------------
# Setup boundary conditions
#--------------------------------------------------------------------------
Bi = Dirichlet_boundary([0.4, 0, 0])          # Inflow
Br = Reflective_boundary(domain)              # Solid reflective wall
Bo = Dirichlet_boundary([-5, 0, 0])           # Outflow

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


#--------------------------------------------------------------------------
# Evolve system through time
#--------------------------------------------------------------------------
for t in domain.evolve(yieldstep = 0.1, finaltime = 16.0):
    print domain.timestepping_statistics()


    if domain.get_quantity('stage').\
           get_values(interpolation_points=[[10, 2.5]]) > 0:
        print 'Stage > 0: Changing to outflow boundary'
        domain.set_boundary({'right': Bo})
```

This example differs from the first two versions in that the topography contains obstacles.

This is accomplished here by defining the function `topography` as follows

```
def topography(x,y):
    """Complex topography defined by a function of vectors x and y
    """

    z = -x/10

    N = len(x)
    for i in range(N):

        # Step
        if 10 < x[i] < 12:
            z[i] += 0.4 - 0.05*y[i]

        # Constriction
        if 27 < x[i] < 29 and y[i] > 3:
            z[i] += 2

        # Pole
        if (x[i] - 34)**2 + (y[i] - 2)**2 < 0.4**2:
            z[i] += 2

    return z
```

In addition, changing the resolution to dx=dy=0.1 creates a finer mesh resolving the new featurse better.

A screenshot of this model at time == 15s is

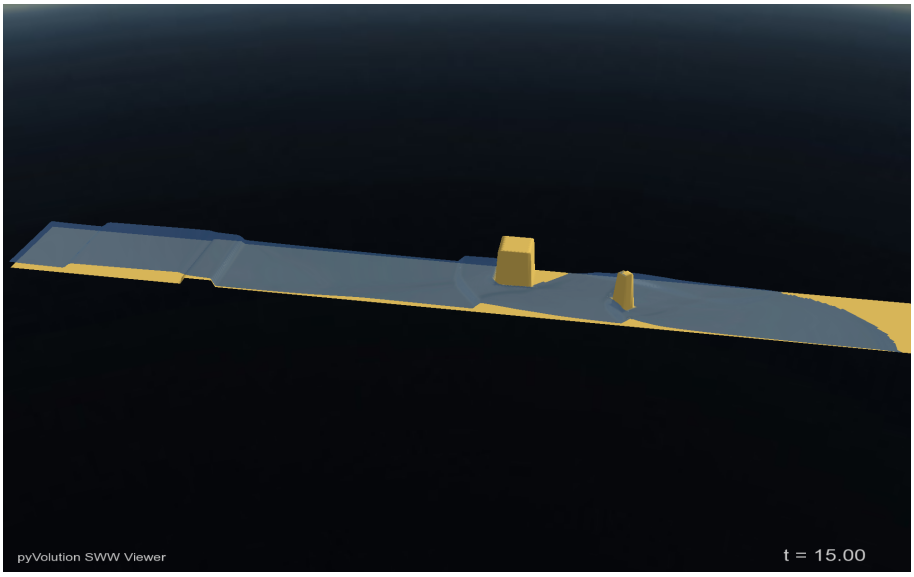

Figure 4.4: More complex flow in a channel

4.6 An Example with Real Data

The following discussion builds on the concepts introduced through the 'runup.py' example and introduces a second example, 'runcairns.py'. This refers to a **hypothetical** scenario using real-life data, in which the domain of interest surrounds the Cairns region. Two scenarios are given; firstly, a hypothetical tsunami wave is generated by a submarine mass failure situated on the edge of the continental shelf, and secondly, a fixed wave of given amplitude and period is introduced through the boundary.

Each scenario has been designed to generate a tsunami which will inundate the Cairns region. To achieve this, suitably large parameters were chosen and were not based on any known tsunami sources or realistic amplitudes.

4.6.1 Overview

As in the case of 'runup.py', the actions carried out by the program can be organised according to this outline:

1. Set up a triangular mesh.
2. Set certain parameters governing the mode of operation of the model—specifying, for instance, where to store the model output.
3. Input various quantities describing physical measurements, such as the elevation, to be specified at each mesh point (vertex).
4. Set up the boundary conditions.
5. Carry out the evolution of the model through a series of time steps and output the results, providing a results file that can be visualised.

4.6.2 The Code

Here is the code for 'runcairns.py':

```
"""Script for running a tsunami inundation scenario for Cairns, QLD Australia.

Source data such as elevation and boundary data is assumed to be available in
directories specified by project.py
The output sww file is stored in directory named after the scenario, i.e
slide or fixed_wave.

The scenario is defined by a triangular mesh created from project.polygon,
the elevation data and a tsunami wave generated by a submarine mass failure.

Geoscience Australia, 2004-present
"""

#-------------------------------------------------------------------------------
# Import necessary modules
#-------------------------------------------------------------------------------

# Standard modules
import os
import time
import sys

# Related major packages
```

```
from anuga.interface import create_domain_from_regions
from anuga.interface import Reflective_boundary
from anuga.interface import Dirichlet_boundary
from anuga.interface import Time_boundary
from anuga.interface import File_boundary
from anuga.interface import Transmissive_stage_zero_momentum_boundary

from anuga.shallow_water.data_manager import convert_dem_from_ascii2netcdf
from anuga.shallow_water.data_manager import dem2pts

from anuga.shallow_water.smf import slide_tsunami

# Application specific imports
import project                 # Definition of file names and polygons


#-------------------------------------------------------------------------------
# Preparation of topographic data
# Convert ASC 2 DEM 2 PTS using source data and store result in source data
#-------------------------------------------------------------------------------

# Create DEM from asc data
convert_dem_from_ascii2netcdf(project.demname, use_cache=True, verbose=True)

# Create pts file for onshore DEM
dem2pts(project.demname, use_cache=True, verbose=True)


#-------------------------------------------------------------------------------
# Create the triangular mesh and domain based on
# overall clipping polygon with a tagged
# boundary and interior regions as defined in project.py
#-------------------------------------------------------------------------------

domain = create_domain_from_regions(project.bounding_polygon,
                                    boundary_tags={'top': [0],
                                                   'ocean_east': [1],
                                                   'bottom': [2],
                                                   'onshore': [3]},
                                    maximum_triangle_area=project.default_res,
                                    mesh_filename=project.meshname,
                                    interior_regions=project.interior_regions,
                                    use_cache=True,
                                    verbose=True)


# Print some stats about mesh and domain
print 'Number of triangles = ', len(domain)
print 'The extent is ', domain.get_extent()
print domain.statistics()

#-------------------------------------------------------------------------------
# Setup parameters of computational domain
#-------------------------------------------------------------------------------


domain.set_name('cairns_' + project.scenario) # Name of sww file
domain.set_datadir('.')                       # Store sww output here
domain.set_minimum_storable_height(0.01)      # Store only depth > 1cm
```

```
#--------------------------------------------------------------------------
# Setup initial conditions
#--------------------------------------------------------------------------

tide = 0.0
domain.set_quantity('stage', tide)
domain.set_quantity('friction', 0.0)
domain.set_quantity('elevation',
                    filename=project.demname + '.pts',
                    use_cache=True,
                    verbose=True,
                    alpha=0.1)


#--------------------------------------------------------------------------
# Setup information for slide scenario (to be applied 1 min into simulation
#--------------------------------------------------------------------------

if project.scenario == 'slide':
    # Function for submarine slide
    tsunami_source = slide_tsunami(length=35000.0,
                                   depth=project.slide_depth,
                                   slope=6.0,
                                   thickness=500.0,
                                   x0=project.slide_origin[0],
                                   y0=project.slide_origin[1],
                                   alpha=0.0,
                                   domain=domain,
                                   verbose=True)


#--------------------------------------------------------------------------
# Setup boundary conditions
#--------------------------------------------------------------------------

print 'Available boundary tags', domain.get_boundary_tags()


Bd = Dirichlet_boundary([tide,0,0]) # Mean water level
Bs = Transmissive_stage_zero_momentum_boundary(domain) # Neutral boundary


if project.scenario == 'fixed_wave':
    # Huge 50m wave starting after 60 seconds and lasting 1 hour.
    Bw = Time_boundary(domain=domain,
                       function=lambda t: [(60<t<3660)*50, 0, 0])

    domain.set_boundary({'ocean_east': Bw,
                         'bottom': Bs,
                         'onshore': Bd,
                         'top': Bs})

if project.scenario == 'slide':
    # Boundary conditions for slide scenario
    domain.set_boundary({'ocean_east': Bd,
                         'bottom': Bd,
                         'onshore': Bd,
```

```
                          'top': Bd})


#--------------------------------------------------------------------------
# Evolve system through time
#--------------------------------------------------------------------------

import time
t0 = time.time()

from Numeric import allclose
from anuga.abstract_2d_finite_volumes.quantity import Quantity

if project.scenario == 'slide':

    for t in domain.evolve(yieldstep=10, finaltime=60):
        print domain.timestepping_statistics()
        print domain.boundary_statistics(tags='ocean_east')

    # Add slide
    thisstagestep = domain.get_quantity('stage')
    if allclose(t, 60):
        slide = Quantity(domain)
        slide.set_values(tsunami_source)
        domain.set_quantity('stage', slide + thisstagestep)

    for t in domain.evolve(yieldstep=10, finaltime=5000,
                           skip_initial_step = True):
        print domain.timestepping_statistics()
        print domain.boundary_statistics(tags='ocean_east')


if project.scenario == 'fixed_wave':

    # Save every two mins leading up to wave approaching land
    for t in domain.evolve(yieldstep=120, finaltime=5000):
        print domain.timestepping_statistics()
        print domain.boundary_statistics(tags='ocean_east')

    # Save every 30 secs as wave starts inundating ashore
    for t in domain.evolve(yieldstep=10, finaltime=10000,
                           skip_initial_step=True):
        print domain.timestepping_statistics()
        print domain.boundary_statistics(tags='ocean_east')

print 'That took %.2f seconds' %(time.time()-t0)
```

In discussing the details of this example, we follow the outline given above, discussing each major step of the code in turn.

4.6.3 Establishing the Mesh

One obvious way that the present example differs from 'runup.py' is in the use of a more complex method to create the mesh. Instead of imposing a mesh structure on a rectangular grid, the technique used for this example involves building mesh structures inside polygons specified by the user, using a mesh-generator.

In its simplest form, the mesh-generator creates the mesh within a single polygon whose vertices are at geographical locations specified by the user. The user specifies the *resolution*—that is, the maximal area of a triangle used for

triangulation—and a triangular mesh is created inside the polygon using a mesh generation engine. On any given platform, the same mesh will be returned.

Boundary tags are not restricted to `'left'`, `'bottom'`, `'right'` and `'top'`, as in the case of 'runup.py'. Instead the user specifies a list of tags appropriate to the configuration being modelled.

In addition, the mesh-generator provides a way to adapt to geographic or other features in the landscape, whose presence may require an increase in resolution. This is done by allowing the user to specify a number of *interior polygons*, each with a specified resolution. It is also possible to specify one or more 'holes'—that is, areas bounded by polygons in which no triangulation is required.

In its general form, the mesh-generator takes for its input a bounding polygon and (optionally) a list of interior polygons. The user specifies resolutions, both for the bounding polygon and for each of the interior polygons. Given this data, the mesh-generator first creates a triangular mesh with varying resolution.

The function used to implement this process is `create_mesh_from_regions`. Its arguments include the bounding polygon and its resolution, a list of boundary tags, and a list of pairs `[polygon, resolution]`, specifying the interior polygons and their resolutions.

The resulting mesh is output to a *mesh file*. This term is used to describe a file of a specific format used to store the data specifying a mesh. (There are in fact two possible formats for such a file: it can either be a binary file, with extension `.msh`, or an ASCII file, with extension `.tsh`. In the present case, the binary file format `.msh` is used. See Section 6.1 (page 61) for more on file formats.)

In practice, the details of the polygons used are read from a separate file 'project.py'. Here is a complete listing of 'project.py':

```
"""Common filenames and locations for topographic data, meshes and outputs.
"""

from anuga.utilities.polygon import read_polygon, plot_polygons, \
                                    polygon_area, is_inside_polygon


#------------------------------------------------------------------------------
# Define scenario as either slide or fixed_wave.
#------------------------------------------------------------------------------
#scenario = 'slide'
scenario = 'fixed_wave'


#------------------------------------------------------------------------------
# Filenames
#------------------------------------------------------------------------------
demname = 'cairns'
meshname = demname + '.msh'

# Filename for locations where timeseries are to be produced
gauge_filename = 'gauges.csv'


#------------------------------------------------------------------------------
# Domain definitions
#------------------------------------------------------------------------------

# bounding polygon for study area
bounding_polygon = read_polygon('extent.csv')

A = polygon_area(bounding_polygon)/1000000.0
print 'Area of bounding polygon = %.2f km^2' % A
```

```
#-------------------------------------------------------------------------------
# Interior region definitions
#-------------------------------------------------------------------------------

# Read interior polygons
poly_cairns = read_polygon('cairns.csv')
poly_island0 = read_polygon('islands.csv')
poly_island1 = read_polygon('islands1.csv')
poly_island2 = read_polygon('islands2.csv')
poly_island3 = read_polygon('islands3.csv')
poly_shallow = read_polygon('shallow.csv')

# Optionally plot points making up these polygons
#plot_polygons([bounding_polygon,poly_cairns,poly_island0,poly_island1,\
#               poly_island2,poly_island3,poly_shallow],\
#               style='boundingpoly',verbose=False)



# Define resolutions (max area per triangle) for each polygon
default_res = 10000000 # Background resolution
islands_res = 100000
cairns_res = 100000
shallow_res = 500000

# Define list of interior regions with associated resolutions
interior_regions = [[poly_cairns, cairns_res],
                    [poly_island0, islands_res],
                    [poly_island1, islands_res],
                    [poly_island2, islands_res],
                    [poly_island3, islands_res],
                    [poly_shallow, shallow_res]]



#-------------------------------------------------------------------------------
# Data for exporting ascii grid
#-------------------------------------------------------------------------------
eastingmin = 363000
eastingmax = 418000
northingmin = 8026600
northingmax = 8145700

#-------------------------------------------------------------------------------
# Data for landslide
#-------------------------------------------------------------------------------
slide_origin = [451871, 8128376] # Assume to be on continental shelf
slide_depth = 500.
```

Figure 4.5 illustrates the landscape of the region for the Cairns example. Understanding the landscape is important in determining the location and resolution of interior polygons. The supporting data is found in the ASCII grid, `cairns.asc`, which has been sourced from the publically available Australian Bathymetry and Topography Grid

2005, [grid250]. The required resolution for inundation modelling will depend on the underlying topography and bathymetry; as the terrain becomes more complex, the desired resolution would decrease to the order of tens of metres.

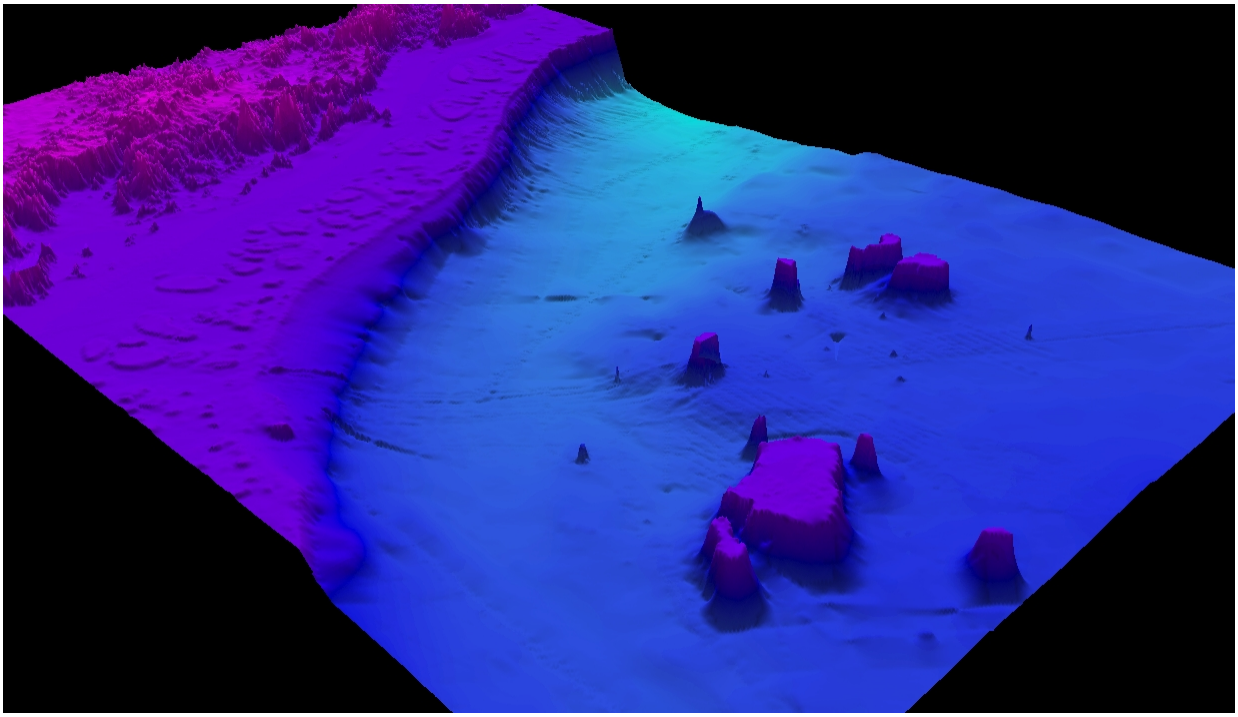

Figure 4.5: Landscape of the Cairns scenario.

The following statements are used to read in the specific polygons from `project.cairns` and assign a defined resolution to each polygon.

```
islands_res = 100000
cairns_res = 100000
shallow_res = 500000
interior_regions = [[project.poly_cairns, cairns_res],
                    [project.poly_island0, islands_res],
                    [project.poly_island1, islands_res],
                    [project.poly_island2, islands_res],
                    [project.poly_island3, islands_res],
                    [project.poly_shallow, shallow_res]]
```

Figure 4.6 illustrates the polygons used for the Cairns scenario.

The statement

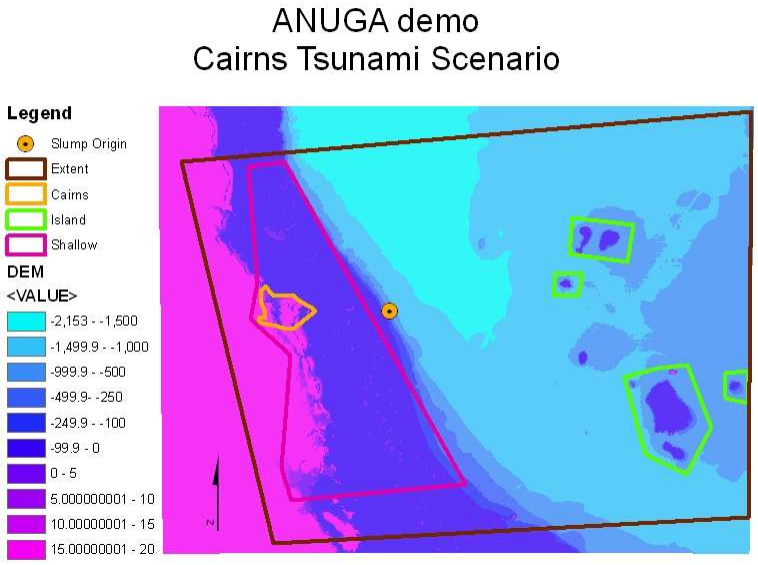

Figure 4.6: Interior and bounding polygons for the Cairns example.

```
remainder_res = 10000000
create_mesh_from_regions(project.bounding_polygon,
                         boundary_tags={'top': [0],
                                        'ocean_east': [1],
                                        'bottom': [2],
                                        'onshore': [3]},
                         maximum_triangle_area=remainder_res,
                         filename=meshname,
                         interior_regions=interior_regions,
                         use_cache=True,
                         verbose=True)
```

is then used to create the mesh, taking the bounding polygon to be the polygon `bounding_polygon` specified in 'project.py'. The argument `boundary_tags` assigns a dictionary, whose keys are the names of the boundary tags used for the bounding polygon—`'top'`, `'ocean_east'`, `'bottom'`, and `'onshore'`— and whose values identify the indices of the segments associated with each of these tags. The polygon may be arranged either clock-wise or counter clock-wise and the indices refer to edges in the order they appear: Edge 0 connects vertex 0 and vertex 1, edge 1 connects vertex 1 and 2; and so forth. (Here, the values associated with each boundary tag are one-element lists, but they can have as many indices as there are edges) If polygons intersect, or edges coincide (or are even very close) the resolution may be undefined in some regions. Use the underlying mesh interface for such cases. See Section 5. If a segment is omitted in the tags definition an Exception is raised.

Note that every point on each polygon defining the mesh will be used as vertices in triangles. Consequently, polygons with points very close together will cause triangles with very small areas to be generated irrespective of the requested resolution. Make sure points on polygons are spaced to be no closer than the smallest resolution requested.

4.6.4 Initialising the Domain

As with 'runup.py', once we have created the mesh, the next step is to create the data structure `domain`. We did this for 'runup.py' by inputting lists of points and triangles and specifying the boundary tags directly. However, in the present case, we use a method that works directly with the mesh file `meshname`, as follows:

```
domain = Domain(meshname, use_cache=True, verbose=True)
```

Providing a filename instead of the lists used in ‘runup.py’ above causes `Domain` to convert a mesh file `meshname` into an instance of `Domain`, allowing us to use methods like `set_quantity` to set quantities and to apply other operations.

The following statements specify a basename and data directory, and identify quantities to be stored. For the first two, values are taken from ‘project.py’.

```
domain.set_name(project.basename)
domain.set_datadir(project.outputdir)
domain.set_quantities_to_be_stored(['stage', 'xmomentum',
    'ymomentum'])
```

4.6.5 Initial Conditions

Quantities for ‘runcairns.py’ are set using similar methods to those in ‘runup.py’. However, in this case, many of the values are read from the auxiliary file ‘project.py’ or, in the case of `elevation`, from an ancillary points file.

4.6.5.1 Stage

For the scenario we are modelling in this case, we use a callable object `tsunami_source`, assigned by means of a function `slide_tsunami`. This is similar to how we set elevation in ‘runup.py’ using a function—however, in this case the function is both more complex and more interesting.

The function returns the water displacement for all `x` and `y` in the domain. The water displacement is a double Gaussian function that depends on the characteristics of the slide (length, width, thickness, slope, etc), its location (origin) and the depth at that location. For this example, we choose to apply the slide function at a specified time into the simulation. **Note, the parameters used in this example have been deliberately chosen to generate a suitably large amplitude tsunami which would inundate the Cairns region.**

4.6.5.2 Friction

We assign the friction exactly as we did for ‘runup.py’:

```
domain.set_quantity('friction', 0.0)
```

4.6.5.3 Elevation

The elevation is specified by reading data from a file:

```
domain.set_quantity('elevation',
                    filename = project.dem_name + '.pts',
                    use_cache = True,
                    verbose = True)
```

4.6.6 Boundary Conditions

Setting boundaries follows a similar pattern to the one used for ‘runup.py’, except that in this case we need to associate a boundary type with each of the boundary tag names introduced when we established the mesh. In place of the four boundary types introduced for ‘runup.py’, we use the reflective boundary for each of the eight tagged segments defined by `create_mesh_from_regions`:

```
Bd = Dirichlet_boundary([0.0,0.0,0.0])
domain.set_boundary( {'ocean_east': Bd, 'bottom': Bd, 'onshore': Bd,
                      'top': Bd} )
```

4.6.7 Evolution

With the basics established, the running of the ‘evolve’ step is very similar to the corresponding step in ‘runup.py’. For the slide scenario, the simulation is run for 5000 seconds with the output stored every ten seconds. For this example, we choose to apply the slide at 60 seconds into the simulation.

```
import time t0 = time.time()


for t in domain.evolve(yieldstep = 10, finaltime = 60):
        domain.write_time()
        domain.write_boundary_statistics(tags = 'ocean_east')

    # add slide
    thisstagestep = domain.get_quantity('stage')
    if allclose(t, 60):
        slide = Quantity(domain)
        slide.set_values(tsunami_source)
        domain.set_quantity('stage', slide + thisstagestep)

    for t in domain.evolve(yieldstep = 10, finaltime = 5000,
                           skip_initial_step = True):
        domain.write_time()
    domain.write_boundary_statistics(tags = 'ocean_east')
```

For the fixed wave scenario, the simulation is run to 10000 seconds, with the first half of the simulation stored at two minute intervals, and the second half of the simulation stored at ten second intervals. This functionality is especially convenient as it allows the detailed parts of the simulation to be viewed at higher time resolution.

```
# save every two mins leading up to wave approaching land
    for t in domain.evolve(yieldstep = 120, finaltime = 5000):
        domain.write_time()
        domain.write_boundary_statistics(tags = 'ocean_east')

    # save every 30 secs as wave starts inundating ashore
    for t in domain.evolve(yieldstep = 10, finaltime = 10000,
                           skip_initial_step = True):
        domain.write_time()
        domain.write_boundary_statistics(tags = 'ocean_east')
```

4.7 Exploring the Model Output

Now that the scenario has been run, the user can view the output in a number of ways. As described earlier, the user may run animate to view a three-dimensional representation of the simulation.

The user may also be interested in a maximum inundation map. This simply shows the maximum water depth over the domain and is achieved with the function sww2dem (described in Section 6.1.9). 'ExportResults.py' demonstrates how this function can be used:

```
import project, os
import sys

from anuga.shallow_water.data_manager import sww2dem
from anuga.abstract_2d_finite_volumes.util import start_screen_catcher
from os import sep

scenario = 'slide'
#scenario = 'fixed_wave'

name = scenario + sep + scenario + 'source'

print 'output dir:', name
which_var = 4
if which_var == 0:  # Stage
    outname = name + '_stage'
    quantityname = 'stage'

if which_var == 1:  # Absolute Momentum
    outname = name + '_momentum'
    quantityname = '(xmomentum**2 + ymomentum**2)**0.5'  #Absolute momentum

if which_var == 2:  # Depth
    outname = name + '_depth'
    quantityname = 'stage-elevation'  #Depth

if which_var == 3:  # Speed
    outname = name + '_speed'
    quantityname = '(xmomentum**2 + ymomentum**2)**0.5/(stage-elevation+1.e-30)'  #Speed

if which_var == 4:  # Elevation
    outname = name + '_elevation'
    quantityname = 'elevation'  #Elevation
```

```
print 'start sww2dem'

sww2dem(name, basename_out = outname,
            quantity = quantityname,
            cellsize = 100,
            easting_min = project.eastingmin,
            easting_max = project.eastingmax,
            northing_min = project.northingmin,
            northing_max = project.northingmax,
            reduction = max,
            verbose = True,
            format = 'ers')
```

The script generates an maximum water depth ASCII grid at a defined resolution (here 100 m^2) which can then be viewed in a GIS environment, for example. The parameters used in the function are defined in 'project.py'. Figures 4.7 and 4.8 show the maximum water depth within the defined region for the slide and fixed wave scenario respectively. **Note, these inundation maps have been based on purely hypothetical scenarios and were designed explicitly for demonstration purposes only.** The user could develop a maximum absolute momentum or other expressions which can be derived from the quantities. It must be noted here that depth is more meaningful when the elevation is positive (`depth` = `stage` − `elevation`) as it describes the water height above the available elevation. When the elevation is negative, depth is meauring the water height from the sea floor. With this in mind, maximum inundation maps are typically "clipped" to the coastline. However, the data input here did not contain a coastline.

The user may also be interested in interrogating the solution at a particular spatial location to understand the behaviour of the system through time. To do this, the user must first define the locations of interest. A number of locations have been identified for the Cairns scenario, as shown in Figure 4.9.

These locations must be stored in either a .csv or .txt file. The corresponding .csv file for the gauges shown in Figure 4.9 is 'gauges.csv'

```
easting,northing,name,elevation
367622.63,8128196.42,Cairns,0
360245.11,8142280.78,Trinity Beach,0
386133.51,8131751.05,Cairns Headland,0
430250,8128812.23,Elford Reef,0
367771.61,8133933.82,Cairns Airport,0
```

Header information has been included to identify the location in terms of eastings and northings, and each gauge is given a name. The elevation column can be zero here. This information is then passed to the function `sww2csv_gauges` (shown in 'GetTimeseries.py' which generates the csv files for each point location. The csv files can then be used in `csv2timeseries_graphs` to create the timeseries plot for each desired quantity. `csv2timeseries_graphs` relies on `pylab` to be installed which is not part of the standard `anuga` release, however it can be downloaded and installed from `http://matplotlib.sourceforge.net/`

```
"""
Generate time series of nominated "gauges" read from project.gauge_filename. This
is done by first running sww2csv_gauges on two different directories to make
'csv' files. Then running csv2timeseries_graphs detailing the two directories
containing the csv file and produces one set of graphs in the 'output_dir' containing
the details at the gauges for both these sww files.

Note, this script will only work if pylab is installed on the platform

"""

from os import sep
```

```
import project
from anuga.abstract_2d_finite_volumes.util import sww2csv_gauges,csv2timeseries_graphs

sww2csv_gauges('slide'+sep+'slidesource.sww',
                project.gauge_filename,
                quantities = ['stage','speed','depth','elevation'],
                verbose=True)

sww2csv_gauges('fixed_wave'+sep+'fixed_wavesource.sww',
               project.gauge_filename,
               quantities = ['stage', 'speed','depth','elevation'],
               verbose=True)

try:
    import pylab
    csv2timeseries_graphs(directories_dic={'slide'+sep:['Slide',0, 0],
                                           'fixed_wave'+sep:['Fixed Wave',0,0]},
                          output_dir='fixed_wave'+sep,
                          base_name='gauge_',
                          plot_numbers='',
                          quantities=['stage','speed','depth'],
                          extra_plot_name='',
                          assess_all_csv_files=True,
                          create_latex=False,
                          verbose=True)
except ImportError:
    #ANUGA does not rely on pylab to work
    print 'must have pylab installed to generate plots'
```

Here, the time series for the quantities stage, depth and speed will be generated for each gauge defined in the gauge file. As described earlier, depth is more meaningful for onshore gauges, and stage is more appropriate for offshore gauges.

As an example output, Figure 4.10 shows the time series for the quantity stage for the Elford Reef location for each scenario (the elevation at this location is negative, therefore stage is the more appropriate quantity to plot). Note the large negative stage value when the slide was introduced. This is due to the double gaussian form of the initial surface displacement of the slide. By contrast, the time series for depth is shown for the onshore location of the Cairns Airport in Figure 4.11.

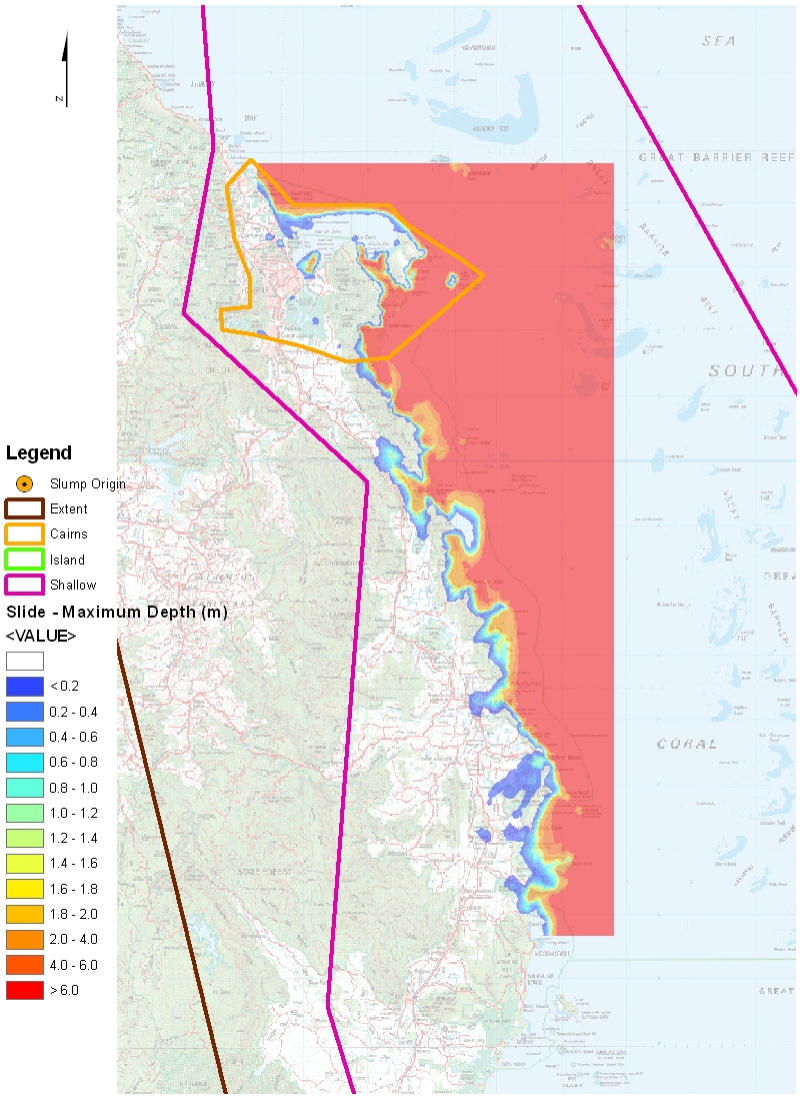

Figure 4.7: Maximum inundation map for the Cairns slide scenario. **Note, this inundation map has been based on a purely hypothetical scenario which was designed explictiy for demonstration purposes only.**

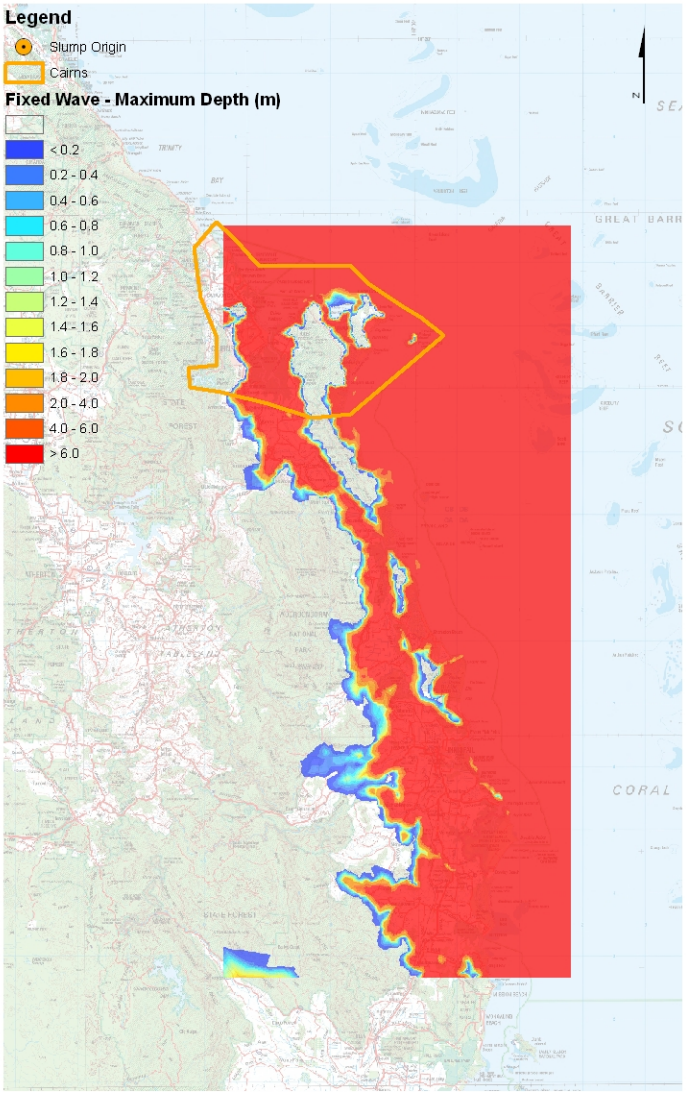

Figure 4.8: Maximum inundation map for the Cairns fixed wave scenario. **Note, this inundaiton map has been based on a purely hypothetical scenario which was designed explictiy for demonstration purposes only.**

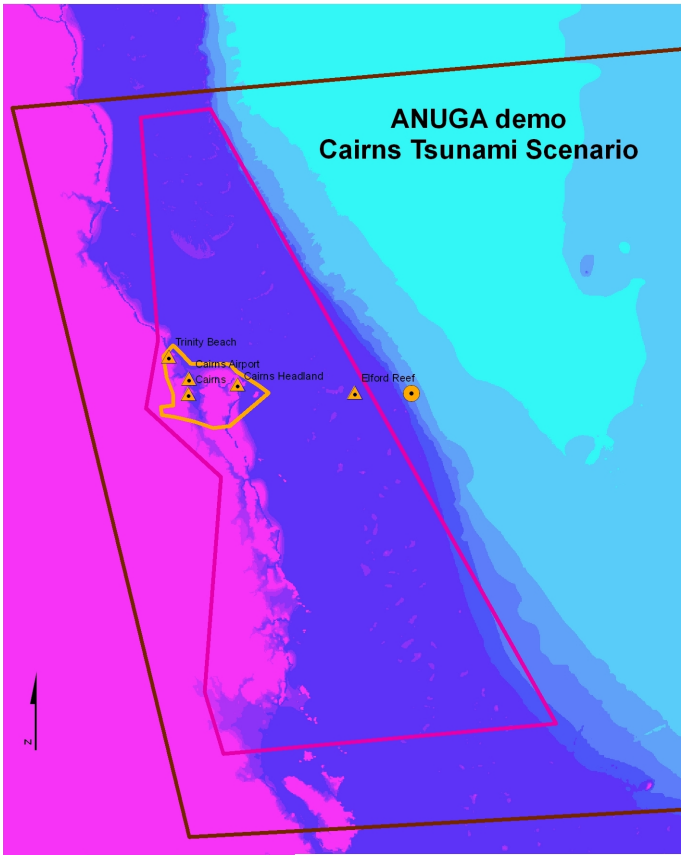

Figure 4.9: Point locations to show time series information for the Cairns scenario.

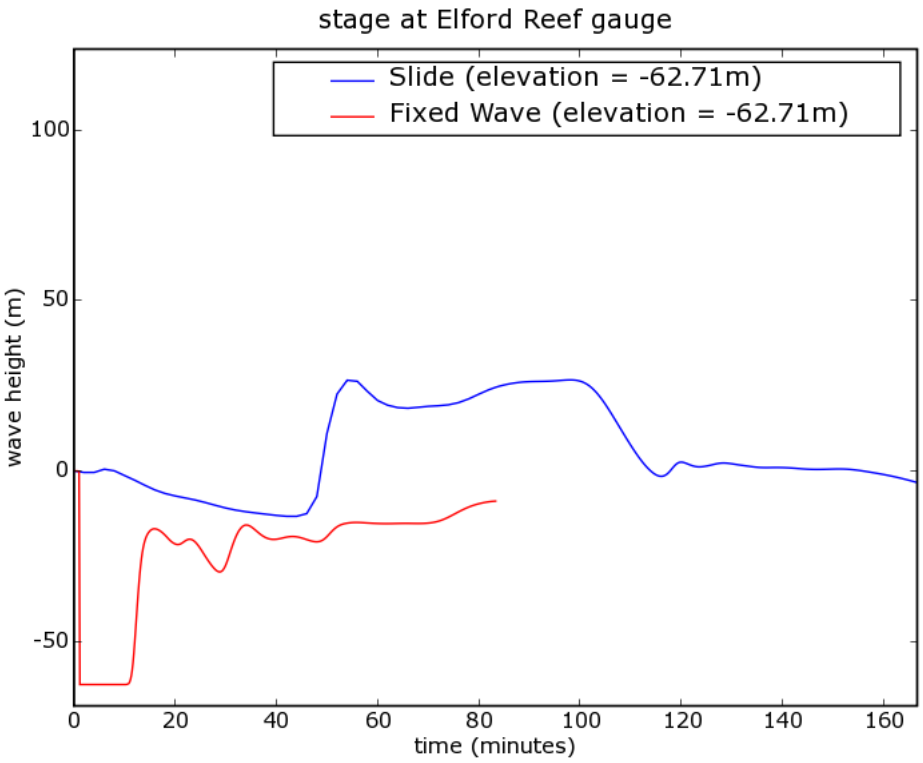

Figure 4.10: Time series information of the quantity stage for the Elford Reef location for the fixed wave and slide scenario.

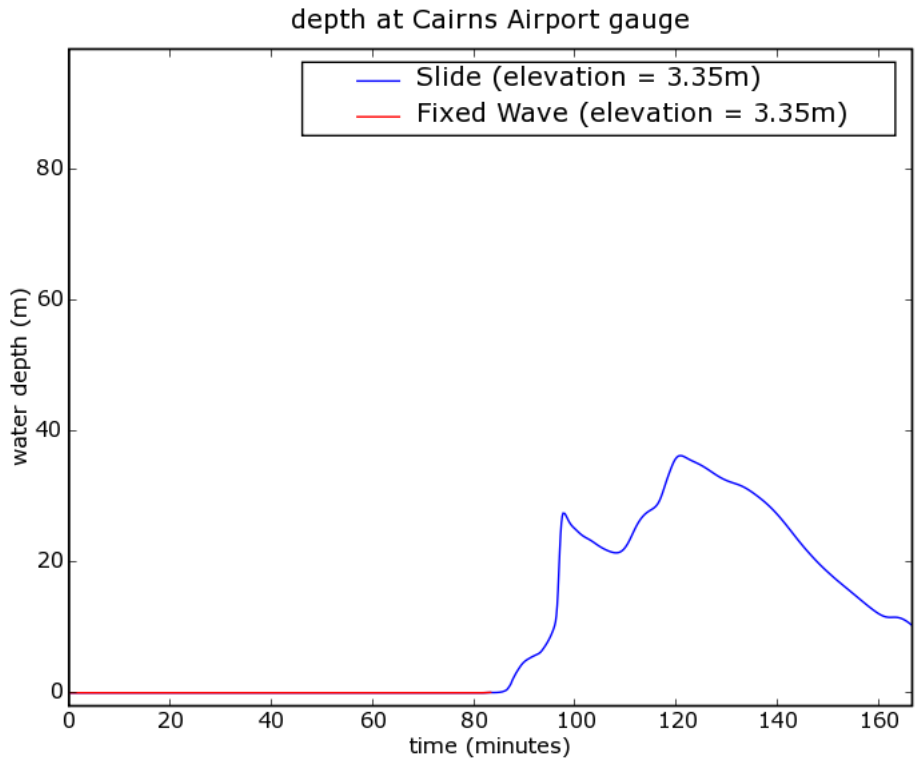

Figure 4.11: Time series information of the quantity depth for the Cairns Airport location for the slide and fixed wave scenario.

CHAPTER

FIVE

ANUGA V1.0 Public Interface

This chapter gives an overview of the features of **ANUGA** v1.0 available to the user at the public interface. These are grouped under the following headings, which correspond to the outline of the examples described in Chapter 4:

- Establishing the Mesh: Section 5.1
- Initialising the Domain: Section 5.2
- Specifying the Quantities: Section **??**
- Initial Conditions: Section 5.3
- Boundary Conditions: Section 5.4
- Forcing Terms: Section 5.5
- Evolution: Section 5.6

The listings are intended merely to give the reader an idea of what each feature is, where to find it and how it can be used—they do not give full specifications; for these the reader may consult the code. The code for every function or class contains a documentation string, or 'docstring', that specifies the precise syntax for its use. This appears immediately after the line introducing the code, between two sets of triple quotes.

Each listing also describes the location of the module in which the code for the feature being described can be found. All modules are in the folder 'inundation' or one of its subfolders, and the location of each module is described relative to 'inundation'. Rather than using pathnames, whose syntax depends on the operating system, we use the format adopted for importing the function or class for use in Python code. For example, suppose we wish to specify that the function `create_mesh_from_regions` is in a module called `mesh_interface` in a subfolder of `inundation` called `pmesh`. In Linux or Unix syntax, the pathname of the file containing the function, relative to 'inundation', would be

```
pmesh/mesh_interface.py
```

while in Windows syntax it would be

```
pmesh\mesh_interface.py
```

Rather than using either of these forms, in this chapter we specify the location simply as `pmesh.mesh_interface`, in keeping with the usage in the Python statement for importing the function, namely:

```
from pmesh.mesh_interface import create_mesh_from_regions
```

Each listing details the full set of parameters for the class or function; however, the description is generally limited to the most important parameters and the reader is again referred to the code for more details.

The following parameters are common to many functions and classes and are omitted from the descriptions given below:

use_cache	Specifies whether caching is to be used for improved performance. See Section A.1 for details on the underlying caching
verbose	If `True`, provides detailed terminal output to the user

5.1 Mesh Generation

Before discussing the part of the interface relating to mesh generation, we begin with a description of a simple example of a mesh and use it to describe how mesh data is stored.

Figure 5.1 represents a very simple mesh comprising just 11 points and 10 triangles.

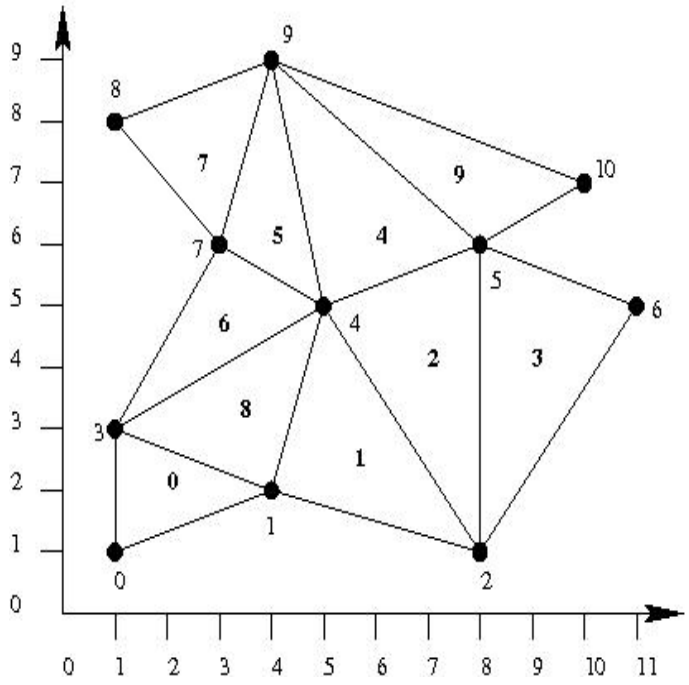

Figure 5.1: A simple mesh

The variables `points`, `triangles` and `boundary` represent the data displayed in Figure 5.1 as follows. The list `points` stores the coordinates of the points, and may be displayed schematically as in Table 5.1.

The list `triangles` specifies the triangles that make up the mesh. It does this by specifying, for each triangle, the indices (the numbers shown in the first column above) that correspond to the three points at the triangles vertices, taken in an anti-clockwise order around the triangle. Thus, in the example shown in Figure 5.1, the variable `triangles` contains the entries shown in Table 5.2. The starting point is arbitrary so triangle $(0, 1, 3)$ is considered the same as $(1, 3, 0)$ and $(3, 0, 1)$.

Finally, the variable `boundary` identifies the boundary triangles and associates a tag with each.

pmesh.meshinterface]pmesh.mesh_interface

`create_mesh_from_regions`(*bounding_polygon, boundary_tags, maximum_triangle_area, filename=None, interior_regions=None, poly_geo_reference=None, mesh_geo_reference=None, minimum_triangle_angle=28.0*)

index	x	y
0	1	1
1	4	2
2	8	1
3	1	3
4	5	5
5	8	6
6	11	5
7	3	6
8	1	8
9	4	9
10	10	7

Table 5.1: Point coordinates for mesh in Figure 5.1

index	`points`		
0	0	1	3
1	1	2	4
2	2	5	4
3	2	6	5
4	4	5	9
5	4	9	7
6	3	4	7
7	7	9	8
8	1	4	3
9	5	10	9

Table 5.2: Triangles for mesh in Figure 5.1

Module: `pmesh.mesh_interface`

This function allows a user to initiate the automatic creation of a mesh inside a specified polygon (input `bounding_polygon`). Among the parameters that can be set are the *resolution* (maximal area for any triangle in the mesh) and the minimal angle allowable in any triangle. The user can specify a number of internal polygons within each of which the resolution of the mesh can be specified. `interior_regions` is a paired list containing the interior polygon and its resolution. Additionally, the user specifies a list of boundary tags, one for each edge of the bounding polygon.

WARNING. Note that the dictionary structure used for the parameter `boundary_tags` is different from that used for the variable `boundary` that occurs in the specification of a mesh. In the case of `boundary`, the tags are the *values* of the dictionary, whereas in the case of `boundary_tags`, the tags are the *keys* and the *value* corresponding to a particular tag is a list of numbers identifying boundary edges labelled with that tag. Because of this, it is theoretically possible to assign the same edge to more than one tag. However, an attempt to do this will cause an error.

WARNING. Do not have polygon lines cross or be on-top of each other. This can result in regions of unspecified resolutions. Do not have polygon close to each other. This can result in the area between the polygons having small triangles. For more control over the mesh outline use the methods described below.

5.1.1 Advanced mesh generation

For more control over the creation of the mesh outline, use the methods of the class `Mesh`.

class `Mesh`(*userSegments=None, userVertices=None, holes=None, regions=None*)
Module: `pmesh.mesh`

A class used to build a mesh outline and generate a two-dimensional triangular mesh. The mesh outline is used to describe features on the mesh, such as the mesh boundary. Many of this classes methods are used to build a mesh outline, such as `add_vertices` and `add_region_from_polygon`.

5.1.1.1 Key Methods of Class Mesh

`add_hole`(*x,y*)
Module: `pmesh.mesh`, Class: `Mesh`

This method is used to build the mesh outline. It defines a hole, when the boundary of the hole has already been defined, by selecting a point within the boundary.

`add_hole_from_polygon`(*self, polygon, tags=None*)
Module: `pmesh.mesh`, Class: `Mesh`

This method is used to add a 'hole' within a region —that is, to define a interior region where the triangular mesh will not be generated—to a `Mesh` instance. The region boundary is described by the polygon passed in. Additionally, the user specifies a list of boundary tags, one for each edge of the bounding polygon.

`add_points_and_segments`(*self, points, segments, segment_tags=None*)
Module: `pmesh.mesh`, Class: `Mesh`

This method is used to build the mesh outline. It adds points and segments connecting the points. Points is a list of points. Segments is a list of segments. Each segment is defined by the start and end of the line by it's point index, e.g. use `segments = [[0,1],[1,2]]` to make a polyline between points 0, 1 and 2. A tag for each segment can optionally be added.

`add_region`(*x,y*)
Module: `pmesh.mesh`, Class: `Mesh`

This method is used to build the mesh outline. It defines a region, when the boundary of the region has already been defined, by selecting a point within the boundary. A region instance is returned. This can be used to set the resolution.

add_region_from_polygon(*self, polygon, segment_tags=None, region_tag=None max_triangle_area=None*)
Module: `pmesh.mesh`, Class: `Mesh`

This method is used to build the mesh outline. It adds a region to a `Mesh` instance. Regions are commonly used to describe an area with an increased density of triangles, by setting `max_triangle_area`. The region boundary is described by the input `polygon`. Additionally, the user specifies a list of segment tags, one for each edge of the bounding polygon. The regional tag is set using `region`.

add_vertices(*point_data*)
Module: `pmesh.mesh`, Class: `Mesh`

Add user vertices. The point_data can be a list of (x,y) values, a numeric array or a geospatial_data instance.

auto_segment(*raw_boundary=raw_boundary, remove_holes=remove_holes, smooth_indents=smooth_indents, expand_pinch=expand_pinch*)
Module: `pmesh.mesh`, Class: `Mesh`

Add segments between some of the user vertices to give the vertices an outline. The outline is an alpha shape. This method is useful since a set of user vertices need to be outlined by segments before generate_mesh is called.

export_mesh_file(*self,ofile*)
Module: `pmesh.mesh`, Class: `Mesh`

This method is used to save the mesh to a file. `ofile` is the name of the mesh file to be written, including the extension. Use the extension `.msh` for the file to be in NetCDF format and `.tsh` for the file to be ASCII format.

generate_mesh(*self, maximum_triangle_area=None, minimum_triangle_angle=28.0, verbose=False*)
Module: `pmesh.mesh`, Class: `Mesh`

This method is used to generate the triangular mesh. The maximal area of any triangle in the mesh can be specified, which is used to control the triangle density, along with the minimum angle in any triangle.

import_ungenerate_file(*self,ofile, tag=None, region_tag=None*)
Module: `pmesh.mesh`, Class: `Mesh`

This method is used to import a polygon file in the ungenerate format, which is used by arcGIS. The polygons from the file are converted to vertices and segments. `ofile` is the name of the polygon file. `tag` is the tag given to all the polygon's segments. `region_tag` is the tag given to all the polygon's segments. If it is a string the one value will be assigned to all regions. If it is a list the first value in the list will be applied to the first polygon etc. If `tag` is not given a value it defaults to None, which means the segement will not effect the water flow, it will only effect the mesh generation.

This function can be used to import building footprints.

5.2 Initialising the Domain

class Domain(*source=None, triangles=None, boundary=None, conserved_quantities=None, other_quantities=None, tagged_elements=None, use_inscribed_circle=False, mesh_filename=None, use_cache=False, verbose=False, full_send_dict=None, ghost_recv_dict=None, processor=0, numproc=1*)
Module: `abstract_2d_finite_volumes.domain`

This class is used to create an instance of a data structure used to store and manipulate data associated with a mesh. The mesh is specified either by assigning the name of a mesh file to `source` or by specifying the points, triangle and boundary of the mesh.

5.2.1 Key Methods of Domain

set_name(*name*)
Module: `abstract_2d_finite_volumes.domain`, page 83

Assigns the name `name` to the domain.

`get_name``()`
Module: `abstract_2d_finite_volumes.domain`

Returns the name assigned to the domain by `set_name`. If no name has been assigned, returns `'domain'`.

`set_datadir``(`*name*`)`
Module: `abstract_2d_finite_volumes.domain`

Specifies the directory used for SWW files, assigning it to the pathname `name`. The default value, before `set_datadir` has been run, is the value `default_datadir` specified in `config.py`.

Since different operating systems use different formats for specifying pathnames, it is necessary to specify path separators using the Python code `os.sep`, rather than the operating-specific ones such as '/' or '\'. For this to work you will need to include the statement `import os` in your code, before the first appearance of `set_datadir`.

For example, to set the data directory to a subdirectory `data` of the directory `project`, you could use the statements:

```
import os
domain.set_datadir{'project' + os.sep + 'data'}
```

`get_datadir``()`
Module: `abstract_2d_finite_volumes.domain`

Returns the data directory set by `set_datadir` or, if `set_datadir` has not been run, returns the value `default_datadir` specified in `config.py`.

`set_minimum_allowed_height``()`
Module: `shallow_water.shallow_water_domain`

Set the minimum depth (in meters) that will be recognised in the numerical scheme (including limiters and flux computations)

Default value is 10^{-3} m, but by setting this to a greater value, e.g. for large scale simulations, the computation time can be significantly reduced.

`set_minimum_storable_height``()`
Module: `shallow_water.shallow_water_domain`

Sets the minimum depth that will be recognised when writing to an sww file. This is useful for removing thin water layers that seems to be caused by friction creep.

`set_maximum_allowed_speed``()`
Module: `shallow_water.shallow_water_domain`

Set the maximum particle speed that is allowed in water shallower than minimum_allowed_height. This is useful for controlling speeds in very thin layers of water and at the same time allow some movement avoiding pooling of water.

`set_time``(`*time=0.0*`)`
Module: `abstract_2d_finite_volumes.domain`

Sets the initial time, in seconds, for the simulation. The default is 0.0.

`set_default_order``(`*n*`)`
Sets the default (spatial) order to the value specified by `n`, which must be either 1 or 2. (Assigning any other value to `n` will cause an error.)

`set_store_vertices_uniquely``(`*flag*`)`
Decide whether vertex values should be stored uniquely as computed in the model or whether they should be reduced to one value per vertex using averaging.

Triangles stored in the sww file can be discontinuous reflecting the internal representation of the finite-volume scheme (this is a feature allowing for arbitrary steepness of the water surface gradient as well as the momentum gradients). However, for visual purposes and also for use with `Field_boundary` (and `File_boundary`) it is often desirable to store triangles with values at each vertex point as the average of the potentially discontinuous numbers found at vertices of different triangles sharing the same vertex location.

Storing one way or the other is controlled in ANUGA through the method `domain.store_vertices_uniquely`. Options are

- `domain.store_vertices_uniquely(True)`: Allow discontinuities in the sww file
- `domain.store_vertices_uniquely(False)`: (Default). Average values to ensure continuity in sww file. The latter also makes for smaller sww files.

Note that when model data in the sww file are averaged (i.e. not stored uniquely), then there will most likely be a small discrepancy between values extracted from the sww file and the same data stored in the model domain. This must be borne in mind when comparing data from the sww files with that of the model internally.

`get_nodes`(*absolute=False*)

Return x,y coordinates of all nodes in mesh.

The nodes are ordered in an Nx2 array where N is the number of nodes. This is the same format they were provided in the constructor i.e. without any duplication.

Boolean keyword argument absolute determines whether coordinates are to be made absolute by taking georeference into account Default is False as many parts of ANUGA expects relative coordinates.

`get_vertex_coordinates`(*absolute=False*)

Return vertex coordinates for all triangles.

Return all vertex coordinates for all triangles as a 3*M x 2 array where the jth vertex of the ith triangle is located in row 3*i+j and M the number of triangles in the mesh.

Boolean keyword argument absolute determines whether coordinates are to be made absolute by taking georeference into account Default is False as many parts of ANUGA expects relative coordinates.

`get_centroid_coordinates`(*absolute=False*)

Return centroid coordinates for all triangles.

Return all centroid coordinates for all triangles as a M x 2 array

Boolean keyword argument absolute determines whether coordinates are to be made absolute by taking georeference into account Default is False as many parts of ANUGA expects relative coordinates.

`get_triangles`(*indices=None*)

Return Mx3 integer array where M is the number of triangles. Each row corresponds to one triangle and the three entries are indices into the mesh nodes which can be obtained using the method get_nodes()

Optional argument, indices is the set of triangle ids of interest.

`get_disconnected_triangles`()

Get mesh based on nodes obtained from get_vertex_coordinates.

Return array Mx3 array of integers where each row corresponds to a triangle. A triangle is a triplet of indices into point coordinates obtained from get_vertex_coordinates and each index appears only once.

This provides a mesh where no triangles share nodes (hence the name disconnected triangles) and different nodes may have the same coordinates.

This version of the mesh is useful for storing meshes with discontinuities at each node and is e.g. used for storing data in sww files.

The triangles created will have the format

```
[[0,1,2],
 [3,4,5],
 [6,7,8],
 ...
 [3*M-3 3*M-2 3*M-1]]
```

5.3 Initial Conditions

In standard usage of partial differential equations, initial conditions refers to the values associated to the system variables (the conserved quantities here) for `time = 0`. In setting up a scenario script as described in Sections 4.1 and 4.6, `set_quantity` is used to define the initial conditions of variables other than the conserved quantities, such as friction. Here, we use the terminology of initial conditions to refer to initial values for variables which need prescription to solve the shallow water wave equation. Further, it must be noted that `set_quantity` does not necessarily have to be used in the initial condition setting; it can be used at any time throughout the simulation.

`set_quantity`(*name, numeric = None, quantity = None, function = None, geospatial_data = None, expression = None, filename = None, attribute_name = None, alpha = None, location = 'vertices', indices = None, verbose = False, use_cache = False*)

Module: `abstract_2d_finite_volumes.domain` (see also `abstract_2d_finite_volumes.quantity.set_values`)

This function is used to assign values to individual quantities for a domain. It is very flexible and can be used with many data types: a statement of the form `domain.set_quantity(name, x)` can be used to define a quantity having the name `name`, where the other argument `x` can be any of the following:

- a number, in which case all vertices in the mesh gets that for the quantity in question.
- a list of numbers or a Numeric array ordered the same way as the mesh vertices.
- a function (e.g. see the samples introduced in Chapter 2)
- an expression composed of other quantities and numbers, arrays, lists (for example, a linear combination of quantities, such as `domain.set_quantity('stage','elevation'+x))`
- the name of a file from which the data can be read. In this case, the optional argument attribute_name will select which attribute to use from the file. If left out, set_quantity will pick one. This is useful in cases where there is only one attribute.
- a geospatial dataset (See Section A.5). Optional argument attribute_name applies here as with files.

Exactly one of the arguments numeric, quantity, function, points, filename must be present.

Set quantity will look at the type of the second argument (`numeric`) and determine what action to take.

Values can also be set using the appropriate keyword arguments. If x is a function, for example, `domain.set_quantity(name, x)`, `domain.set_quantity(name, numeric=x)`, and `domain.set_quantity(name, function=x)` are all equivalent.

Other optional arguments are

- `indices` which is a list of ids of triangles to which set_quantity should apply its assignment of values.
- `location` determines which part of the triangles to assign to. Options are 'vertices' (default), 'edges', 'unique vertices', and 'centroids'. If 'vertices' are use, edge and centroid values are automatically computed as the appropriate averages. This option ensures continuity of the surface. If, on the other hand, 'centroids' is used vertex and edge values will be set to the same value effectively creating a piecewise constant surface with possible discontinuities at the edges.

ANUGA v1.0 provides a number of predefined initial conditions to be used with `set_quantity`. See for example callable object `slump_tsunami` below.

add_quantity(*name, numeric = None, quantity = None, function = None, geospatial_data = None, expression = None, filename = None, attribute_name = None, alpha = None, location = 'vertices', indices = None, verbose = False, use_cache = False*)

Module: `abstract_2d_finite_volumes.domain` (see also `abstract_2d_finite_volumes.domain.set_quantity`)

This function is used to *add* values to individual quantities for a domain. It has the same syntax as `domain.set_quantity(name, x)`.

A typical use of this function is to add structures to an existing elevation model:

```
# Create digital elevation model from points file
domain.set_quantity('elevation',
                    filename = 'elevation_file.pts,
                    verbose = True)

# Add buildings from file
building_polygons, building_heights = csv2building_polygons(building_file)

B = []
for key in building_polygons:
    poly = building_polygons[key]
    elev = building_heights[key]
    B.append((poly, elev))

domain.add_quantity('elevation', Polygon_function(B, default=0.0))
```

set_region(*tag, quantity, X, location='vertices'*)

Module: `abstract_2d_finite_volumes.domain`

(see also `abstract_2d_finite_volumes.quantity.set_values`)

This function is used to assign values to individual quantities given a regional tag. It is similar to `set_quantity`. For example, if in the mesh-generator a regional tag of 'ditch' was used, set_region can be used to set elevation of this region to -10m. X is the constant or function to be applied to the quantity, over the tagged region. Location describes how the values will be applied. Options are 'vertices' (default), 'edges', 'unique vertices', and 'centroids'.

This method can also be called with a list of region objects. This is useful for adding quantities in regions, and having one quantity value based on another quantity. See `abstract_2d_finite_volumes.region` for more details.

slump_tsunami(*length, depth, slope, width=None, thickness=None, x0=0.0, y0=0.0, alpha=0.0, gravity=9.8, gamma=1.85, massco=1, dragco=1, frictionco=0, psi=0, dx=None, kappa=3.0, kappad=0.8, zsmall=0.01, domain=None, verbose=False*)

Module: `shallow_water.smf`

This function returns a callable object representing an initial water displacement generated by a submarine sediment failure. These failures can take the form of a submarine slump or slide. In the case of a slide, use `slide_tsunami` instead.

The arguments include as a minimum, the slump or slide length, the water depth to the centre of sediment mass, and the bathymetric slope. Other slump or slide parameters can be included if they are known.

file_function(*filename, domain = None, quantities = None, interpolation_points = None, verbose = False, use_cache = False*)

Module: `abstract_2d_finite_volumes.util`

Reads the time history of spatial data for specified interpolation points from a NetCDF file (`filename`) and returns a callable object. `filename` could be a `sww` or `sts` file. Returns interpolated values based on the input file using the underlying `interpolation_function`.

`quantities` is either the name of a single quantity to be interpolated or a list of such quantity names. In the second case, the resulting function will return a tuple of values—one for each quantity.

`interpolation_points` is a list of absolute coordinates or a geospatial object for points at which values are sought.

`boundary_polygon` is a list of coordinates specifying the vertices of the boundary. This must be the same polygon as used when calling `create_mesh_from_regions`. This argument can only be used when reading boundary data from the STS format.

The model time stored within the file function can be accessed using the method `f.get_time()`

The underlying algorithm used is as follows:
Given a time series (i.e. a series of values associated with different times), whose values are either just numbers, a set of numbers defined at the vertices of a triangular mesh (such as those stored in SWW files) or a set of numbers defined at a number of points on the boundary (such as those stored in STS files), `Interpolation_function` is used to create a callable object that interpolates a value for an arbitrary time `t` within the model limits and possibly a point `(x, y)` within a mesh region.

The actual time series at which data is available is specified by means of an array `time` of monotonically increasing times. The quantities containing the values to be interpolated are specified in an array—or dictionary of arrays (used in conjunction with the optional argument `quantity_names`) — called `quantities`. The optional arguments `vertex_coordinates` and `triangles` represent the spatial mesh associated with the quantity arrays. If omitted the function must be created using an STS file or a TMS file.

Since, in practice, values need to be computed at specified points, the syntax allows the user to specify, once and for all, a list `interpolation_points` of points at which values are required. In this case, the function may be called using the form `f(t, id)`, where `id` is an index for the list `interpolation_points`.

5.4 Boundary Conditions

ANUGA v1.0 provides a large number of predefined boundary conditions, represented by objects such as `Reflective_boundary(domain)` and `Dirichlet_boundary([0.2, 0.0, 0.0])`, described in the examples in Chapter 2. Alternatively, you may prefer to "roll your own", following the method explained in Section 5.4.2.

These boundary objects may be used with the function `set_boundary` described below to assign boundary conditions according to the tags used to label boundary segments.

`set_boundary`(*boundary_map*)
Module: `abstract_2d_finite_volumes.domain`

This function allows you to assign a boundary object (corresponding to a pre-defined or user-specified boundary condition) to every boundary segment that has been assigned a particular tag.

This is done by specifying a dictionary `boundary_map`, whose values are the boundary objects and whose keys are the symbolic tags.

`get_boundary_tags`()
Module: `abstract_2d_finite_volumes.domain`

Returns a list of the available boundary tags.

5.4.1 Predefined boundary conditions

class `Reflective_boundary`(*Boundary*)
Module: `shallow_water`

Reflective boundary returns same conserved quantities as those present in the neighbouring volume but reflected.

This class is specific to the shallow water equation as it works with the momentum quantities assumed to be the second and third conserved quantities.

class `Transmissive_boundary`(*domain = None*)
Module: `abstract_2d_finite_volumes.generic_boundary_conditions`

A transmissive boundary returns the same conserved quantities as those present in the neighbouring volume.

The underlying domain must be specified when the boundary is instantiated.

class `Dirichlet_boundary`(*conserved_quantities=None*)
Module: `abstract_2d_finite_volumes.generic_boundary_conditions`

A Dirichlet boundary returns constant values for each of conserved quantities. In the example of `Dirichlet_boundary([0.2, 0.0, 0.0])`, the `stage` value at the boundary is 0.2 and the `xmomentum` and `ymomentum` at the boundary are set to 0.0. The list must contain a value for each conserved quantity.

class `Time_boundary`(*domain = None, f = None*)
Module: `abstract_2d_finite_volumes.generic_boundary_conditions`

A time-dependent boundary returns values for the conserved quantities as a function `f(t)` of time. The user must specify the domain to get access to the model time.

Optional argument `default_boundary` can be used to specify another boundary object to be used in case model time exceeds the time availabel in the file used by `File_boundary`. The `default_boundary` could be a simple Dirichlet condition or even another `Time_boundary` typically using data pertaining to another time interval.

class `File_boundary`(*Boundary*)
Module: `abstract_2d_finite_volumes.generic_boundary_conditions`

This method may be used if the user wishes to apply a SWW file, STS file or a time series file (TMS) to a boundary segment or segments. The boundary values are obtained from a file and interpolated to the appropriate segments for each conserved quantity.

Optional argument `default_boundary` can be used to specify another boundary object to be used in case model time exceeds the time availabel in the file used by `File_boundary`. The `default_boundary` could be a simple Dirichlet condition or even another `File_boundary` typically using data pertaining to another time interval.

class `Field_boundary`(*Boundary*)
Module: `shallow_water.shallow_water_domain`

This method works in the same way as `File_boundary` except that it allows for the value of stage to be offset by a constant specified in the keyword argument `mean_stage`.

This functionality allows for models to be run for a range of tides using the same boundary information (from .sts, .sww or .tms files). The tidal value for each run would then be specified in the keyword argument `mean_stage`. If `mean_stage` = 0.0, `Field_boundary` and `File_boundary` behave identically.

Note that if the optional argument `default_boundary` is specified it's stage value will be adjusted by `mean_stage` just like the values obtained from the file.

See `File_boundary` for further details.

class `Transmissive_momentum_set_stage_boundary`(*Boundary*)
Module: `shallow_water`

This boundary returns same momentum conserved quantities as those present in its neighbour volume but sets stage as in a Time_boundary. The underlying domain must be specified when boundary is instantiated

This type of boundary is useful when stage is known at the boundary as a function of time, but momenta (or speeds) aren't.

This class is specific to the shallow water equation as it works with the momentum quantities assumed to be the second and third conserved quantities.

In some circumstances, this boundary condition may cause numerical instabilities for similar reasons as what has been observed with the simple fully transmissive boundary condition (see Page 51). This could for example be the case if a planar wave is reflected out through this boundary.

class `Transmissive_stage_zero_momentum_boundary`(*Boundary*)
Module: `shallow_water`

This boundary returns same stage conserved quantities as those present in its neighbour volume but sets momentum to zero. The underlying domain must be specified when boundary is instantiated

This type of boundary is useful when stage is known at the boundary as a function of time, but momentum should be set to zero. This is for example the case where a boundary is needed in the ocean on the two sides perpendicular to the coast to maintain the wave height of the incoming wave.

This class is specific to the shallow water equation as it works with the momentum quantities assumed to be the second and third conserved quantities.

This boundary condition should not cause the numerical instabilities seen with the transmissive momentum boundary conditions (see Page 51 and Page 51).

class `Dirichlet_discharge_boundary`(*Boundary*)
Module: `shallow_water`

Sets stage (stage0) Sets momentum (wh0) in the inward normal direction.

5.4.2 User-defined boundary conditions

All boundary classes must inherit from the generic boundary class `Boundary` and have a method called `evaluate` which must take as inputs `self, vol_id, edge_id` where self refers to the object itself and vol_id and edge_id are integers referring to particular edges. The method must return a list of three floating point numbers representing values for `stage`, `xmomentum` and `ymomentum`, respectively.

The constructor of a particular boundary class may be used to specify particular values or flags to be used by the `evaluate` method. Please refer to the source code for the existing boundary conditions for examples of how to implement boundary conditions.

5.5 Forcing Terms

ANUGA v1.0 provides a number of predefined forcing functions to be used with simulations. Gravity and friction are always calculated using the elevation and friction quantities, but the user may additionally add forcing terms to the list `domain.forcing_terms` and have them affect the model.

Currently, predefined forcing terms are

`General_forcing`()
Module: `shallow_water.shallow_water_domain`

This is a general class to modify any quantity according to a given rate of change. Other specific forcing terms are based on this class but it can be used by itself as well (e.g. to modify momentum).

The General_forcing class takes as input:

- `domain`: a reference to the domain being evolved
- `quantity_name`: The name of the quantity that will be affected by this forcing term
- `rate`: The rate at which the quantity should change. The parameter `rate` can be eithe a constant or a function of time. Positive values indicate increases, negative values indicate decreases. The parametr `rate` can be `None` at initialisation but must be specified before forcing term is applied (i.e. simulation has started). The default value is 0.0 - i.e. no forcing.

- `center, radius`: Optionally restrict forcing to a circle with given center and radius.
- `polygon`: Optionally restrict forcing to an area enclosed by given polygon.

Note specifying both center, radius and polygon will cause an exception to be thrown. Moreover, if the specified polygon or circle does not lie fully within the mesh boundary an Exception will be thrown.

Example:

```
P = [[x0, y0], [x1, y0], [x1, y1], [x0, y1]] # Square polygon

xmom = General_forcing(domain, 'xmomentum', polygon=P)
ymom = General_forcing(domain, 'ymomentum', polygon=P)

xmom.rate = f
ymom.rate = g

domain.forcing_terms.append(xmom)
domain.forcing_terms.append(ymom)
```

Here, `f`, `g` are assumed to be defined as functions of time providing a time dependent rate of change for xmomentum and ymomentum respectively. P is assumed to be polygon, specified as a list of points.

`Inflow``()`

Module: `shallow_water.shallow_water_domain`

This is a general class for inflow and abstraction of water according to a given rate of change. This class will always modify the `stage` quantity.

Inflow is based on the General_forcing class so the functionality is similar.

The Inflow class takes as input:

- `domain`: a reference to the domain being evolved
- `rate`: The flow rate in m^3/s at which the stage should change. The parameter `rate` can be eithe a constant or a function of time. Positive values indicate inflow, negative values indicate outflow.
  Note: The specified flow will be divided by the area of the inflow region and then applied to update the stage quantity.
- `center, radius`: Optionally restrict forcing to a circle with given center and radius.
- `polygon`: Optionally restrict forcing to an area enclosed by given polygon.

Example:

```
hydrograph = Inflow(center=(320, 300), radius=10,
                    rate=file_function('QPMF_Rot_Sub13.tms'))

domain.forcing_terms.append(hydrograph)
```

Here, `'QPMF_Rot_Sub13.tms'` is assumed to be a NetCDF file in the format `tms` defining a timeseries for a hydrograph.

`Rainfall``()`

Module: `shallow_water.shallow_water_domain`

This is a general class for implementing rainfall over the domain, possibly restricted to a given circle or polygon. This class will always modify the `stage` quantity.

Rainfall is based on the General_forcing class so the functionality is similar.

The Rainfall class takes as input:

- `domain`: a reference to the domain being evolved
- `rate`: Total rain rate over the specified domain. Note: Raingauge Data needs to reflect the time step. For example: if rain gauge is mm read every `dt` seconds, then the input here is as `mm/dt` so 10 mm in 5 minutes becomes 10/(5x60) = 0.0333mm/s.

  This parameter can be either a constant or a function of time. Positive values indicate rain being added (or be used for general infiltration), negative values indicate outflow at the specified rate (presumably this could model evaporation or abstraction).
- `center, radius`: Optionally restrict forcing to a circle with given center and radius.
- `polygon`: Optionally restrict forcing to an area enclosed by given polygon.

Example:

```
catchmentrainfall = Rainfall(rate=file_function('Q100_2hr_Rain.tms'))
domain.forcing_terms.append(catchmentrainfall)
```

Here, `'Q100_2hr_Rain.tms'` is assumed to be a NetCDF file in the format `tms` defining a timeseries for the rainfall.

`Culvert_flow``()`

Module: `culver_flows.culvert_class`

This is a general class for implementing flow through a culvert. This class modifies the quantities `stage, xmomentum, ymomentum` in areas at both ends of the culvert.

The Culvert_flow forcing term uses `Inflow` and General_forcing to update the quantities. The flow drection is determined on-the-fly so openings are referenced simple as opening0 and opening1 with either being able to take the role as Inflow and Outflow.

The Culvert_flow class takes as input:

- `domain`: a reference to the domain being evolved
- `label`: Short text naming the culvert
- `description`: Text describing it
- `end_point0`: Coordinates of one opening
- `end_point1`: Coordinates of other opening
- `width`:
- `height`:
- `diameter`:
- `manning`: Mannings Roughness for Culvert
- `invert_level0`: Invert level if not the same as the Elevation on the Domain
- `invert_level1`: Invert level if not the same as the Elevation on the Domain
- `culvert_routine`: Function specifying the calculation of flow based on energy difference between the two openings (see below)
- `number_of_barrels`: Number of identical pipes in the culvert. Default == 1.

The user can specify different culvert routines. Hower ANUGA currently provides only one, namely the `boyd_generalised_culvert_model` as used in the example below.

Example:

```
from anuga.culvert_flows.culvert_class import Culvert_flow
from anuga.culvert_flows.culvert_routines import boyd_generalised_culvert_model

culvert1 = Culvert_flow(domain,
                        label='Culvert No. 1',
                        description='This culvert is a test unit 1.2m Wide by 0.75m High
                        end_point0=[9.0, 2.5],
                        end_point1=[13.0, 2.5],
                        width=1.20,height=0.75,
                        culvert_routine=boyd_generalised_culvert_model,
number_of_barrels=1,
                        verbose=True)

culvert2 = Culvert_flow(domain,
                        label='Culvert No. 2',
                        description='This culvert is a circular test with d=1.2m',
                        end_point0=[9.0, 1.5],
                        end_point1=[30.0, 3.5],
                        diameter=1.20,
                        invert_level0=7,
                        culvert_routine=boyd_generalised_culvert_model,
number_of_barrels=1,
                        verbose=True)

domain.forcing_terms.append(culvert1)
domain.forcing_terms.append(culvert2)
```

5.6 Evolution

evolve(*yieldstep = None, finaltime = None, duration = None, skip_initial_step = False*)

Module: `abstract_2d_finite_volumes.domain`

This function (a method of `domain`) is invoked once all the preliminaries have been completed, and causes the model to progress through successive steps in its evolution, storing results and outputting statistics whenever a user-specified period `yieldstep` is completed (generally during this period the model will evolve through several steps internally as the method forces the water speed to be calculated on successive new cells).

The code specified by the user in the block following the evolve statement is only executed once every `yieldstep` even though ANUGA typically will take many more internal steps behind the scenes.

The user specifies the total time period over which the evolution is to take place, by specifying values (in seconds) for either `duration` or `finaltime`, as well as the interval in seconds after which results are to be stored and statistics output.

You can include `evolve` in a statement of the type:

```
for t in domain.evolve(yieldstep, finaltime):
    <Do something with domain and t>
```

5.6.1 Diagnostics

statistics()
Module: abstract_2d_finite_volumes.domain

timestepping_statistics()
Module: abstract_2d_finite_volumes.domain

Returns a string of the following type for each timestep:

```
Time = 0.9000, delta t in [0.00598964, 0.01177388], steps=12 (12)
```

Here the numbers in `steps=12 (12)` indicate the number of steps taken and the number of first-order steps, respectively.

The optional keyword argument `track_speeds=True` will generate a histogram of speeds generated by each triangle. The speeds relate to the size of the timesteps used by ANUGA and this diagnostics may help pinpoint problem areas where excessive speeds are generated.

boundary_statistics(*quantities = None, tags = None*)
Module: abstract_2d_finite_volumes.domain

Returns a string of the following type when `quantities = 'stage'` and `tags = ['top', 'bottom']`:

```
Boundary values at time 0.5000:
   top:
       stage in [ -0.25821218,  -0.02499998]
   bottom:
       stage in [ -0.27098821,  -0.02499974]
```

get_quantity(*name, location='vertices', indices = None*)
Module: abstract_2d_finite_volumes.domain

This function returns a Quantity object Q. Access to it's values should be done through Q.get_values documented on Page 57.

set_quantities_to_be_monitored()
Module: abstract_2d_finite_volumes.domain

Selects quantities and derived quantities for which extrema attained at internal timesteps will be collected.

Information can be tracked in the evolve loop by printing `quantity_statistics` and collected data will be stored in the sww file.

Optional parameters `polygon` and `time_interval` may be specified to restrict the extremum computation.

quantity_statistics()
Module: abstract_2d_finite_volumes.domain

Reports on extrema attained by selected quantities.

Returns a string of the following type for each timestep:

```
Monitored quantities at time 1.0000:
  stage-elevation:
    values since time = 0.00 in [0.00000000, 0.30000000]
    minimum attained at time = 0.00000000, location = (0.16666667, 0.33333333)
    maximum attained at time = 0.00000000, location = (0.83333333, 0.16666667)
  ymomentum:
    values since time = 0.00 in [0.00000000, 0.06241221]
    minimum attained at time = 0.00000000, location = (0.33333333, 0.16666667)
    maximum attained at time = 0.22472667, location = (0.83333333, 0.66666667)
  xmomentum:
    values since time = 0.00 in [-0.06062178, 0.47886313]
    minimum attained at time = 0.00000000, location = (0.16666667, 0.33333333)
    maximum attained at time = 0.35103646, location = (0.83333333, 0.16666667)
```

The quantities (and derived quantities) listed here must be selected at model initialisation using the method `domain.set_quantities_to_be_monitored`.

The optional keyword argument `precision='%.4f'` will determine the precision used for floating point values in the output. This diagnostics helps track extrema attained by the selected quantities at every internal timestep.

These values are also stored in the sww file for post processing.

`get_values`(*location='vertices', indices = None*)
Module: `abstract_2d_finite_volumes.quantity`

Extract values for quantity as a Numeric array.

```
Inputs:
        interpolation_points: List of x, y coordinates where value is
                              sought (using interpolation). If points
                              are given, values of location and indices
                              are ignored. Assume either absolute UTM
                              coordinates or geospatial data object.

        location: Where values are to be stored.
                  Permissible options are: vertices, edges, centroids
                  and unique vertices. Default is 'vertices'
```

The returned values will have the leading dimension equal to length of the indices list or N (all values) if indices is None.

In case of location == 'centroids' the dimension of returned values will be a list or a Numerical array of length N, N being the number of elements.

In case of location == 'vertices' or 'edges' the dimension of returned values will be of dimension Nx3

In case of location == 'unique vertices' the average value at each vertex will be returned and the dimension of returned values will be a 1d array of length "number of vertices"

Indices is the set of element ids that the operation applies to.

The values will be stored in elements following their internal ordering.

`set_values`(*location='vertices', indices = None*)
Module: `abstract_2d_finite_volumes.quantity`

Assign values to a quantity object. This method works the same way as `set_quantity` except that it doesn't take a quantity name as the first argument. The reason to use `set_values` is for example to assign values to a new quantity that has been created but which is not part of the domain's predefined quantities.

The method `set_values` is always called by `set_quantity` behind the scenes.

`get_integral``()`
Module: `abstract_2d_finite_volumes.quantity`

Return computed integral over entire domain for this quantity

`get_maximum_value``(`*indices = None*`)`
Module: `abstract_2d_finite_volumes.quantity`

Return maximum value of quantity (on centroids)

Optional argument indices is the set of element ids that the operation applies to. If omitted all elements are considered.

We do not seek the maximum at vertices as each vertex can have multiple values - one for each triangle sharing it.

`get_maximum_location``(`*indices = None*`)`
Module: `abstract_2d_finite_volumes.quantity`

Return location of maximum value of quantity (on centroids)

Optional argument indices is the set of element ids that the operation applies to.

We do not seek the maximum at vertices as each vertex can have multiple values - one for each triangle sharing it.

If there are multiple cells with same maximum value, the first cell encountered in the triangle array is returned.

`get_wet_elements``(`*indices=None*`)`
Module: `shallow_water.shallow_water_domain`

Return indices for elements where h $>$ minimum_allowed_height Optional argument indices is the set of element ids that the operation applies to.

`get_maximum_inundation_elevation``(`*indices=None*`)`
Module: `shallow_water.shallow_water_domain`

Return highest elevation where h > 0.
Optional argument indices is the set of element ids that the operation applies to.

Example to find maximum runup elevation:
z = domain.get_maximum_inundation_elevation()

`get_maximum_inundation_location``(`*indices=None*`)`
Module: `shallow_water.shallow_water_domain`

Return location (x,y) of highest elevation where h > 0.
Optional argument indices is the set of element ids that the operation applies to.

Example to find maximum runup location:
x, y = domain.get_maximum_inundation_location()

5.7 Queries of SWW model output files

After a model has been run, it is often useful to extract various information from the sww output file (see Section 6.1.3). This is typically more convenient than using the diagnostics described in Section 5.6.1 which rely on the model running - something that can be very time consuming. The sww files are easy and quick to read and offer much information about the model results such as runup heights, time histories of selected quantities, flow through cross sections and much more.

get_maximum_inundation_elevation(*filename, polygon=None, time_interval=None, verbose=False*)
Module: shallow_water.data_manager

Return highest elevation where depth is positive ($h > 0$)

Example to find maximum runup elevation:
max_runup = get_maximum_inundation_elevation(filename, polygon=None, time_interval=None, verbose=False)

filename is a NetCDF sww file containing ANUGA model output. Optional arguments polygon and time_interval restricts the maximum runup calculation to a points that lie within the specified polygon and time interval.

If no inundation is found within polygon and time_interval the return value is None signifying "No Runup" or "Everything is dry".

See doc string for general function get_maximum_inundation_data for details.

get_maximum_inundation_location(*filename, polygon=None, time_interval=None, verbose=False*)
Module: shallow_water.data_manager

Return location (x,y) of highest elevation where depth is positive ($h > 0$)

Example to find maximum runup location:
max_runup_location = get_maximum_inundation_location(filename, polygon=None, time_interval=None, verbose=False)

filename is a NetCDF sww file containing ANUGA model output. Optional arguments polygon and time_interval restricts the maximum runup calculation to a points that lie within the specified polygon and time interval.

If no inundation is found within polygon and time_interval the return value is None signifying "No Runup" or "Everything is dry".

See doc string for general function get_maximum_inundation_data for details.

sww2timeseries(*swwfiles, gauge_filename, production_dirs, report = None, reportname = None, plot_quantity = None, generate_fig = False, surface = None, time_min = None, time_max = None, time_thinning = 1, time_unit = None, title_on = None, use_cache = False, verbose = False*)
Module: anuga.abstract_2d_finite_volumes.util

Return csv files for the location in the gauge_filename and can also return plots of them

See doc string for general function sww2timeseries for details.

get_flow_through_cross_section(*filename, cross_section, verbose=False*)
Module: shallow_water.data_manager

Obtain flow $[m^3/s]$ perpendicular to specified cross section.

Inputs:

- filename: Name of sww file containing ANUGA model output.
- polyline: Representation of desired cross section - it may contain multiple sections allowing for complex shapes. Assume absolute UTM coordinates.

Output:

- time: All stored times in sww file
- Q: Hydrograph of total flow across given segments for all stored times.

The normal flow is computed for each triangle intersected by the polyline and added up. Multiple segments at different angles are specified the normal flows may partially cancel each other.

Example to find flow through cross section:

```
cross_section = [[x, 0], [x, width]]
time, Q = get_flow_through_cross_section(filename,
                                         cross_section,
                                         verbose=False)
```

See doc string for general function get_maximum_inundation_data for details.

5.8 Other

domain.create_quantity_from_expression(*string*)

Handy for creating derived quantities on-the-fly as for example

```
Depth = domain.create_quantity_from_expression('stage-elevation')

exp = '(xmomentum*xmomentum + ymomentum*ymomentum)**0.5')
Absolute_momentum = domain.create_quantity_from_expression(exp)
```

CHAPTER

SIX

ANUGA V1.0 System Architecture

6.1 File Formats

ANUGA v1.0 makes use of a number of different file formats. The following table lists all these formats, which are described in more detail in the paragraphs below.

Extension	Description
`.sww`	NetCDF format for storing model output with mesh information `f(t,x,y)`
`.sts`	NetCDF format for storing model ouput `f(t,x,y)` without any mesh information
`.tms`	NetCDF format for storing time series `f(t)`
`.csv/.txt`	ASCII format called points csv for storing arbitrary points and associated attributes
`.pts`	NetCDF format for storing arbitrary points and associated attributes
`.asc`	ASCII format of regular DEMs as output from ArcView
`.prj`	Associated ArcView file giving more metadata for `.asc` format
`.ers`	ERMapper header format of regular DEMs for ArcView
`.dem`	NetCDF representation of regular DEM data
`.tsh`	ASCII format for storing meshes and associated boundary and region info
`.msh`	NetCDF format for storing meshes and associated boundary and region info
`.nc`	Native ferret NetCDF format
`.geo`	Houdinis ASCII geometry format (?)

The above table shows the file extensions used to identify the formats of files. However, typically, in referring to a format we capitalise the extension and omit the initial full stop—thus, we refer, for example, to ‘SWW files’ or ‘PRJ files’.

A typical dataflow can be described as follows:

6.1.1 Manually Created Files

ASC, PRJ Digital elevation models (gridded)
NC Model outputs for use as boundary conditions (e.g. from MOST)

6.1.2 Automatically Created Files

ASC, PRJ → DEM → PTS	Convert DEMs to native `.pts` file
NC → SWW	Convert MOST boundary files to boundary `.sww`
PTS + TSH → TSH with elevation	Least squares fit
TSH → SWW	Convert TSH to `.sww`-viewable using `animate`
TSH + Boundary SWW → SWW	Simulation using **`ANUGA`** `v1.0`
Polygonal mesh outline →	TSH or MSH

6.1.3 SWW, STS and TMS Formats

The SWW, STS and TMS formats are all NetCDF formats, and are of key importance for **ANUGA** v1.0 .

An SWW file is used for storing **ANUGA** v1.0 output and therefore pertains to a set of points and a set of times at which a model is evaluated. It contains, in addition to dimension information, the following variables:

- `x` and `y`: coordinates of the points, represented as Numeric arrays
- `elevation`, a Numeric array storing bed-elevations
- `volumes`, a list specifying the points at the vertices of each of the triangles
- `time`, a Numeric array containing times for model evaluation

The contents of an SWW file may be viewed using the anuga viewer `animate`, which creates an on-screen geometric representation. See section A.2 (page 76) in Appendix A for more on `animate`.

Alternatively, there are tools, such as `ncdump`, that allow you to convert an NetCDF file into a readable format such as the Class Definition Language (CDL). The following is an excerpt from a CDL representation of the output file 'runup.sww' generated from running the simple example 'runup.py' of Chapter 4:

```
netcdf bedslope {
dimensions:
    number_of_volumes = 200 ;
    number_of_vertices = 3 ;
    number_of_points = 121 ;
    number_of_timesteps = UNLIMITED ; // (41 currently)
variables:
    float x(number_of_points) ;
    float y(number_of_points) ;
    float elevation(number_of_points) ;
    float z(number_of_points) ;
    int volumes(number_of_volumes, number_of_vertices) ;
    float time(number_of_timesteps) ;
    float stage(number_of_timesteps, number_of_points) ;
    float xmomentum(number_of_timesteps, number_of_points) ;
    float ymomentum(number_of_timesteps, number_of_points) ;

// global attributes:
        :institution = "Geoscience Australia" ;
        :description = "Output from pyvolution suitable for plotting" ;
        :smoothing = "Yes" ;
        :order = 1 ;
        :starttime = 0 ;
        :xllcorner = 0. ;
        :yllcorner = 0. ;
```

```
        :zone = 56 ;
        :false_easting = 500000 ;
        :false_northing = 10000000 ;
        :datum = "wgs84" ;
        :projection = "UTM" ;
        :units = "m" ;
data:

 x = 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,
    0.1, 0.1, 0.1, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,
    0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.4, 0.4, 0.4,
    0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
    0.5, 0.5, 0.5, 0.5, 0.5, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6,
    0.6, 0.6, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.8,
    0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.9, 0.9, 0.9, 0.9,
    0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ;

 y = 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 0, 0.1, 0.2, 0.3,
    0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
    0.8, 0.9, 1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 0, 0.1,
    0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 0, 0.1, 0.2, 0.3, 0.4, 0.5,
    0.6, 0.7, 0.8, 0.9, 1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1,
    0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 0, 0.1, 0.2, 0.3, 0.4,
    0.5, 0.6, 0.7, 0.8, 0.9, 1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,
    0.9, 1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1 ;

 elevation = 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -0.05, -0.05, -0.05, -0.05,
    -0.05, -0.05, -0.05, -0.05, -0.05, -0.05, -0.05, -0.1, -0.1, -0.1, -0.1,
    -0.1, -0.1, -0.1, -0.1, -0.1, -0.1, -0.1, -0.15, -0.15, -0.15, -0.15,
    -0.15, -0.15, -0.15, -0.15, -0.15, -0.15, -0.15, -0.2, -0.2, -0.2, -0.2,
    -0.2, -0.2, -0.2, -0.2, -0.2, -0.2, -0.2, -0.25, -0.25, -0.25, -0.25,
    -0.25, -0.25, -0.25, -0.25, -0.25, -0.25, -0.25, -0.3, -0.3, -0.3, -0.3,
    -0.3, -0.3, -0.3, -0.3, -0.3, -0.3, -0.3, -0.35, -0.35, -0.35, -0.35,
    -0.35, -0.35, -0.35, -0.35, -0.35, -0.35, -0.35, -0.4, -0.4, -0.4, -0.4,
    -0.4, -0.4, -0.4, -0.4, -0.4, -0.4, -0.4, -0.45, -0.45, -0.45, -0.45,
    -0.45, -0.45, -0.45, -0.45, -0.45, -0.45, -0.45, -0.5, -0.5, -0.5, -0.5,
    -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5 ;

 z = 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -0.05, -0.05, -0.05, -0.05, -0.05,
    -0.05, -0.05, -0.05, -0.05, -0.05, -0.05, -0.1, -0.1, -0.1, -0.1, -0.1,
    -0.1, -0.1, -0.1, -0.1, -0.1, -0.1, -0.15, -0.15, -0.15, -0.15, -0.15,
    -0.15, -0.15, -0.15, -0.15, -0.15, -0.15, -0.2, -0.2, -0.2, -0.2, -0.2,
    -0.2, -0.2, -0.2, -0.2, -0.2, -0.2, -0.25, -0.25, -0.25, -0.25, -0.25,
    -0.25, -0.25, -0.25, -0.25, -0.25, -0.25, -0.3, -0.3, -0.3, -0.3, -0.3,
    -0.3, -0.3, -0.3, -0.3, -0.3, -0.3, -0.35, -0.35, -0.35, -0.35, -0.35,
    -0.35, -0.35, -0.35, -0.35, -0.35, -0.35, -0.4, -0.4, -0.4, -0.4, -0.4,
    -0.4, -0.4, -0.4, -0.4, -0.4, -0.4, -0.45, -0.45, -0.45, -0.45, -0.45,
    -0.45, -0.45, -0.45, -0.45, -0.45, -0.45, -0.5, -0.5, -0.5, -0.5, -0.5,
    -0.5, -0.5, -0.5, -0.5, -0.5, -0.5 ;

 volumes =
  11, 12, 0,
  1, 0, 12,
  12, 13, 1,
  2, 1, 13,
  13, 14, 2,
  3, 2, 14,
  14, 15, 3,
  4, 3, 15,
```

```
 ...,
 109, 108, 120 ;

time = 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.1, 1.2, 1.3,
   1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8,
   2.9, 3, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4 ;

stage =
 0.025, 0.02777778, 0.02777778, 0.02777778, 0.02777778, 0.02777778,
   0.02777778, 0.02777778, 0.02777778, 0.02777778, 0.03333334, -0.01111111,
   5.782412e-019, 5.782412e-019, 5.782412e-019, 5.782412e-019,
   5.782412e-019, 5.782412e-019, 5.782412e-019, 5.782412e-019,
   ...
   -0.1654055, -0.1558442, -0.1504084 ;

xmomentum =
 _, _, _, _, _, ,... _, _ ;

ymomentum =
 _, _, _, _, _, ,... _, _ ;
}
```

The SWW format is used not only for output but also serves as input for functions such as `file_boundary` and `file_function`, described in Chapter 5.

An STS file is used for storing a set of points and and associated set of times. It contains, in addition to dimension information, the following variables:

- `x` and `y`: coordinates of the points, represented as Numeric arrays
- `permutation`: Original indices of the points as specified by the optional `ordering_file` (see the function `urs2sts` in Section 6.1.9).
- `elevation`, a Numeric array storing bed-elevations
- `time`, a Numeric array containing times for model evaluation

The only difference between the STS format and the SWW format is the former does not contain a list specifying the points at the vertices of each of the triangles (`volumes`). Consequenlty information (arrays) stored within an STS file such as `elevation` can be accessed in exactly the same way as it would be extraced from an SWW file.

A TMS file is used to store time series data that is independent of position.

6.1.4 Mesh File Formats

A mesh file is a file that has a specific format suited to triangular meshes and their outlines. A mesh file can have one of two formats: it can be either a TSH file, which is an ASCII file, or an MSH file, which is a NetCDF file. A mesh file can be generated from the function `create_mesh_from_regions` (see Section 5.1) and used to initialise a domain.

A mesh file can define the outline of the mesh—the vertices and line segments that enclose the region in which the mesh is created—and the triangular mesh itself, which is specified by listing the triangles and their vertices, and the segments, which are those sides of the triangles that are associated with boundary conditions.

In addition, a mesh file may contain ‘holes’ and/or ‘regions’. A hole represents an area where no mesh is to be created, while a region is a labelled area used for defining properties of a mesh, such as friction values. A hole or region is specified by a point and bounded by a number of segments that enclose that point.

A mesh file can also contain a georeference, which describes an offset to be applied to x and y values — e.g. to the vertices.

6.1.5 Formats for Storing Arbitrary Points and Attributes

A CSV/TXT file is used to store data representing arbitrary numerical attributes associated with a set of points.

The format for an CSV/TXT file is:

first line: `[column names]`
other lines: `[x value], [y value], [attributes]`

for example:

```
x, y, elevation, friction
0.6, 0.7, 4.9, 0.3
1.9, 2.8, 5, 0.3
2.7, 2.4, 5.2, 0.3
```

The delimiter is a comma. The first two columns are assumed to be x, y coordinates.

A PTS file is a NetCDF representation of the data held in an points CSV file. If the data is associated with a set of N points, then the data is stored using an $N \times 2$ Numeric array of float variables for the points and an $N \times 1$ Numeric array for each attribute.

6.1.6 ArcView Formats

Files of the three formats ASC, PRJ and ERS are all associated with data from ArcView.

An ASC file is an ASCII representation of DEM output from ArcView. It contains a header with the following format:

```
ncols         753
nrows         766
xllcorner     314036.58727982
yllcorner     6224951.2960092
cellsize      100
NODATA_value  -9999
```

The remainder of the file contains the elevation data for each grid point in the grid defined by the above information.

A PRJ file is an ArcView file used in conjunction with an ASC file to represent metadata for a DEM.

6.1.7 DEM Format

A DEM file in **ANUGA** v1.0 is a NetCDF representation of regular DEM data.

6.1.8 Other Formats

6.1.9 Basic File Conversions

`sww2dem`(*basename_in, basename_out = None, quantity = None, timestep = None, reduction = None, cellsize = 10, number_of_decimal_places = None, NODATA_value = -9999, easting_min = None, easting_max = None, northing_min = None, northing_max = None, expand_search = False, verbose = False, origin = None, datum = 'WGS84', format = 'ers'*)

Module: `shallow_water.data_manager`

Takes data from an SWW file `basename_in` and converts it to DEM format (ASC or ERS) of a desired grid size `cellsize` in metres. The user can select how many the decimal places the output data can be written to using `number_of_decimal_places`, with the default being 3. The easting and northing values are used if

the user wished to determine the output within a specified rectangular area. The `reduction` input refers to a function to reduce the quantities over all time step of the SWW file, example, maximum.

`dem2pts`(*basename_in, basename_out=None, easting_min=None, easting_max=None, northing_min=None, northing_max=None, use_cache=False, verbose=False*)
Module: `shallow_water.data_manager`

Takes DEM data (a NetCDF file representation of data from a regular Digital Elevation Model) and converts it to PTS format.

`urs2sts`(*basename_in, basename_out=None, weights=None, verbose=False, origin=None,mean_stage=0.0, zscale=1.0, ordering_filename=None*)
Module: `shallow_water.data_manager`

Takes urs data in (timeseries data in mux2 format) and converts it to STS format. The optional filename `ordering_filename` specifies the permutation of indices of points to select along with their longitudes and latitudes. This permutation will also be stored in the STS file. If absent, all points are taken and the permutation will be trivial, i.e. $0, 1, \ldots, N-1$, where N is the total number of points stored.

`csv2building_polygons`(*file_name, floor_height=3*)
Module: `shallow_water.data_manager`

Convert CSV files of the form:

```
easting,northing,id,floors
422664.22,870785.46,2,0
422672.48,870780.14,2,0
422668.17,870772.62,2,0
422660.35,870777.17,2,0
422664.22,870785.46,2,0
422661.30,871215.06,3,1
422667.50,871215.70,3,1
422668.30,871204.86,3,1
422662.21,871204.33,3,1
422661.30,871215.06,3,1
```

to a dictionary of polygons with id as key. The associated number of floors are converted to m above MSL and returned as a separate dictionary also keyed by id.

Optional parameter floor_height is the height of each building story.

These can e.g. be converted to a Polygon_function for use with add_quantity as shown on page 49.

CHAPTER

SEVEN

ANUGA V1.0 mathematical background

7.1 Introduction

This chapter outlines the mathematics underpinning **ANUGA** v1.0 .

7.2 Model

The shallow water wave equations are a system of differential conservation equations which describe the flow of a thin layer of fluid over terrain. The form of the equations are:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{E}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y} = \mathbf{S}$$

where $\mathbf{U} = \left[\begin{array}{ccc} h & uh & vh \end{array} \right]^T$ is the vector of conserved quantities; water depth h, x-momentum uh and y-momentum vh. Other quantities entering the system are bed elevation z and stage (absolute water level) w, where the relation $w = z + h$ holds true at all times. The fluxes in the x and y directions, $\mathbf{E}$ and $\mathbf{G}$ are given by

$$\mathbf{E} = \left[\begin{array}{c} uh \\ u^2h + gh^2/2 \\ uvh \end{array} \right] \text{ and } \mathbf{G} = \left[\begin{array}{c} vh \\ vuh \\ v^2h + gh^2/2 \end{array} \right]$$

and the source term (which includes gravity and friction) is given by

$$\mathbf{S} = \left[\begin{array}{c} 0 \\ -gh(z_x + S_{fx}) \\ -gh(z_y + S_{fy}) \end{array} \right]$$

where S_f is the bed friction. The friction term is modelled using Manning's resistance law

$$S_{fx} = \frac{u\eta^2\sqrt{u^2+v^2}}{h^{4/3}} \text{ and } S_{fy} = \frac{v\eta^2\sqrt{u^2+v^2}}{h^{4/3}}$$

in which η is the Manning resistance coefficient. The model does not currently include consideration of kinematic viscosity or dispersion.

As demonstrated in our papers, [ZR1999, nielsen2005] these equations and their implementation in **ANUGA** v1.0 provide a reliable model of general flows associated with inundation such as dam breaks and tsunamis.

7.3 Finite Volume Method

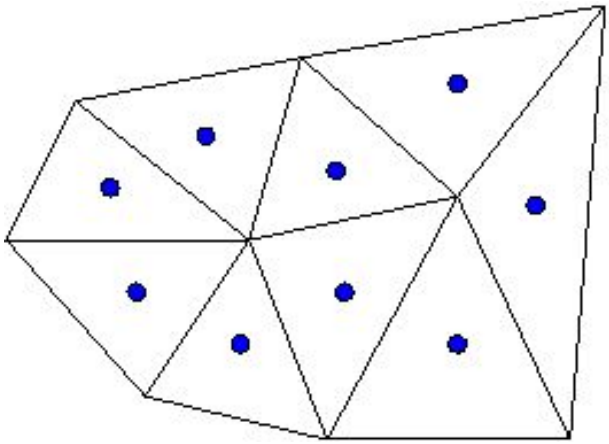

Figure 7.1: Triangular mesh used in our finite volume method. Conserved quantities h, uh and vh are associated with the centroid of each triangular cell.

We use a finite-volume method for solving the shallow water wave equations [ZR1999]. The study area is represented by a mesh of triangular cells as in Figure 7.1 in which the conserved quantities of water depth h, and horizontal momentum (uh, vh), in each volume are to be determined. The size of the triangles may be varied within the mesh to allow greater resolution in regions of particular interest.

The equations constituting the finite-volume method are obtained by integrating the differential conservation equations over each triangular cell of the mesh. Introducing some notation we use i to refer to the ith triangular cell T_i, and $\mathcal{N}(i)$ to the set of indices referring to the cells neighbouring the ith cell. Then A_i is the area of the ith triangular cell and l_{ij} is the length of the edge between the ith and jth cells.

By applying the divergence theorem we obtain for each volume an equation which describes the rate of change of the average of the conserved quantities within each cell, in terms of the fluxes across the edges of the cells and the effect of the source terms. In particular, rate equations associated with each cell have the form

$$\frac{d\mathbf{U}_i}{dt} + \frac{1}{A_i} \sum_{j \in \mathcal{N}(i)} \mathbf{H}_{ij} l_{ij} = \mathbf{S}_i$$

where

- $\mathbf{U}_i$ the vector of conserved quantities averaged over the ith cell,
- $\mathbf{S}_i$ is the source term associated with the ith cell, and
- $\mathbf{H}_{ij}$ is the outward normal flux of material across the *ij*th edge.

The flux $\mathbf{H}_{ij}$ is evaluated using a numerical flux function $\mathbf{H}(\cdot, \cdot; \cdot)$ which is consistent with the shallow water flux in the sense that for all conservation vectors $\mathbf{U}$ and normal vectors $\mathbf{n}$

$$H(\mathbf{U}, \mathbf{U}; \mathbf{n}) = \mathbf{E}(\mathbf{U})n_1 + \mathbf{G}(\mathbf{U})n_2.$$

Then

$$\mathbf{H}_{ij} = \mathbf{H}(\mathbf{U}_i(m_{ij}), \mathbf{U}_j(m_{ij}); \mathbf{n}_{ij})$$

where m_{ij} is the midpoint of the *ij*th edge and $\mathbf{n}_{ij}$ is the outward pointing normal, with respect to the ith cell, on the *ij*th edge. The function $\mathbf{U}_i(x)$ for $x \in T_i$ is obtained from the vector $\mathbf{U}_k$ of conserved average values for the ith and neighbouring cells.

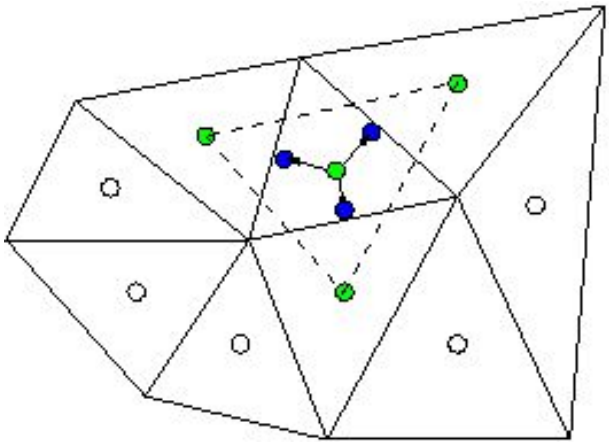

Figure 7.2: From the values of the conserved quantities at the centroid of the cell and its neighbouring cells, a discontinuous piecewise linear reconstruction of the conserved quantities is obtained.

We use a second order reconstruction to produce a piece-wise linear function construction of the conserved quantities for all $x \in T_i$ for each cell (see Figure 7.2. This function is allowed to be discontinuous across the edges of the cells, but the slope of this function is limited to avoid artificially introduced oscillations.

Godunov's method (see [Toro1999]) involves calculating the numerical flux function $\mathbf{H}(\cdot, \cdot; \cdot)$ by exactly solving the corresponding one dimensional Riemann problem normal to the edge. We use the central-upwind scheme of [1] to calculate an approximation of the flux across each edge.

In the computations presented in this paper we use an explicit Euler time stepping method with variable timestepping adapted to the observed CFL condition:

$$\Delta t = \min_{k,i=[0,1,2]} \min \left(\frac{r_k}{v_{k,i}}, \frac{r_{n_{k,i}}}{v_{k,i}} \right) \tag{7.1}$$

where r_k is the radius of the k'th triangle and $v_{k,i}$ is the maximal velocity across edge joining triangle k and it's i'th neighbour, triangle $n_{k,i}$, as calculated by the numerical flux function using the central upwind scheme of [1]. The symbol $r_{n_{k,i}}$ denotes the radius of the i'th neighbour of triangle k. The radii are calculated as radii of the inscribed circles of each triangle.

7.4 Flux limiting

The shallow water equations are solved numerically using a finite volume method on unstructured triangular grid. The upwind central scheme due to Kurganov and Petrova is used as an approximate Riemann solver for the computation of inviscid flux functions. This makes it possible to handle discontinuous solutions.

To alleviate the problems associated with numerical instabilities due to small water depths near a wet/dry boundary we employ a new flux limiter that ensures that unphysical fluxes are never encounted.

Let u and v be the velocity components in the x and y direction, w the absolute water level (stage) and z the bed elevation. The latter are assumed to be relative to the same height datum. The conserved quantities tracked by ANUGA are momentum in the x-direction ($\mu = uh$), momentum in the y-direction ($\nu = vh$) and depth ($h = w - z$).

The flux calculation requires access to the velocity vector (u, v) where each component is obtained as $u = \mu/h$ and $v = \nu/h$ respectively. In the presence of very small water depths, these calculations become numerically unreliable and will typically cause unphysical speeds.

We have employed a flux limiter which replaces the calculations above with the limited approximations.

$$\hat{u} = \frac{\mu}{h + h_0/h}, \hat{v} = \frac{\nu}{h + h_0/h}, \tag{7.2}$$

where h_0 is a regularisation parameter that controls the minimal magnitude of the denominator. Taking the limits we have for $\hat{u}$

$$\lim_{h\to 0} \hat{u} = \lim_{h\to 0} \frac{\mu}{h + h_0/h} = 0$$

and

$$\lim_{h\to \infty} \hat{u} = \lim_{h\to \infty} \frac{\mu}{h + h_0/h} = \frac{\mu}{h} = u$$

with similar results for $\hat{v}$.

The maximal value of $\hat{u}$ is attained when $h + h_0/h$ is minimal or (by differentiating the denominator)

$$1 - h_0/h^2 = 0$$

or

$$h_0 = h^2$$

ANUGA has a global parameter H_0 that controls the minimal depth which is considered in the various equations. This parameter is typically set to 10^{-3}. Setting

$$h_0 = H_0^2$$

provides a reasonable balance between accurracy and stability. In fact, setting $h = H_0$ will scale the predicted speed by a factor of 0.5:

$$\left[\frac{\mu}{h + h_0/h}\right]_{h=H_0} = \frac{\mu}{2H_0}$$

In general, for multiples of the minimal depth NH_0 one obtains

$$\left[\frac{\mu}{h + h_0/h}\right]_{h=NH_0} = \frac{\mu}{H_0(1 + 1/N^2)}$$

which converges quadratically to the true value with the multiple N.

7.5 Slope limiting

A multidimensional slope-limiting technique is employed to achieve second-order spatial accuracy and to prevent spurious oscillations. This is using the MinMod limiter and is documented elsewhere.

However close to the bed, the limiter must ensure that no negative depths occur. On the other hand, in deep water, the bed topography should be ignored for the purpose of the limiter.

Let w, z, h be the stage, bed elevation and depth at the centroid and let w_i, z_i, h_i be the stage, bed elevation and depth at vertex i. Define the minimal depth across all vertices as $h_{\min}$ as

$$h_{\min} = \min_i h_i$$

Let $\tilde{w_i}$ be the stage obtained from a gradient limiter limiting on stage only. The corresponding depth is then defined as

$$\tilde{h_i} = \tilde{w_i} - z_i$$

We would use this limiter in deep water which we will define (somewhat boldly) as

$$h_{\min} \geq \epsilon$$

Similarly, let $\bar{w}_i$ be the stage obtained from a gradient limiter limiting on depth respecting the bed slope. The corresponding depth is defined as

$$\bar{h_i} = \bar{w}_i - z_i$$

We introduce the concept of a balanced stage w_i which is obtained as the linear combination

$$w_i = \alpha\tilde{w}_i + (1-\alpha)\bar{w}_i$$

or

$$w_i = z_i + \alpha\tilde{h_i} + (1-\alpha)\bar{h_i}$$

where $\alpha \in [0,1]$.

Since $\tilde{w}_i$ is obtained in 'deep' water where the bedslope is ignored we have immediately that

$$\alpha = 1 \text{ for } h_{\min} \geq \epsilon$$

If $h_{\min} < \epsilon$ we want to use the 'shallow' limiter just enough that no negative depths occur. Formally, we will require that

$$\alpha\tilde{h_i} + (1-\alpha)\bar{h_i} > \epsilon, \forall i$$

or

$$\alpha(\tilde{h_i} - \bar{h_i}) > \epsilon - \bar{h_i}, \forall i \tag{7.3}$$

There are two cases:

1. $\bar{h_i} \leq \tilde{h_i}$: The deep water (limited using stage) vertex is at least as far away from the bed than the shallow water (limited using depth). In this case we won't need any contribution from $\bar{h_i}$ and can accept any α.

   E.g. $\alpha = 1$ reduces Equation 7.3 to

   $$\tilde{h_i} > \epsilon$$

   whereas $\alpha = 0$ yields

   $$\bar{h_i} > \epsilon$$

   all well and good.

2. $\bar{h_i} > \tilde{h_i}$: In this case the the deep water vertex is closer to the bed than the shallow water vertex or even below the bed. In this case we need to find an α that will ensure a positive depth. Rearranging Equation 7.3 and solving for α one obtains the bound

   $$\alpha < \frac{\epsilon - \bar{h_i}}{\tilde{h_i} - \bar{h_i}}, \forall i$$

Ensuring Equation 7.3 holds true for all vertices one arrives at the definition

$$\alpha = \min_i \frac{\bar{h_i} - \epsilon}{\bar{h_i} - \tilde{h_i}}$$

which will guarantee that no vertex 'cuts' through the bed. Finally, should $\bar{h_i} < \epsilon$ and therefore $\alpha < 0$, we suggest setting $\alpha = 0$ and similarly capping α at 1 just in case.

CHAPTER

EIGHT

Basic **ANUGA** v1.0 Assumptions

8.1 Time

Physical model time cannot be earlier than 1 Jan 1970 00:00:00. If one wished to recreate scenarios prior to that date it must be done using some relative time (e.g. 0).

The ANUGA domain has an attribute `starttime` which is used in cases where the simulation should be started later than the beginning of some input data such as those obtained from boundaries or forcing functions (hydrographs, file_boundary etc)

The `file_boundary` may adjust domain.startime in case the input data does not itself start until a later time.

8.2 Spatial data

8.2.1 Projection

All spatial data relates to the WGS84 datum (or GDA94) and assumes a projection into UTM with false easting of 500000 and false northing of 1000000 on the southern hemisphere (0 on the northern hemisphere). All locations must consequently be specified in Cartesian coordinates (eastings, northings) or (x,y) where the unit is metres. Alternative projections can be assumed, but ANUGA does have the concept of a UTM zone that must be the same for all coordinates in the model.

8.2.2 Internal coordinates

It is important to realise that ANUGA for numerical precision uses coordinates that are relative to the lower left node of the rectangle containing the mesh ($x_{\min}$, $y_{\min}$). This origin is referred to internally as xllcorner, yllcorner following the ESRI ascii grid notation. The sww file format also includes xllcorner, yllcorner and any coordinate in the file should be adjusted by adding this origin. See Section 6.1.3.

Throughout the ANUGA interface functions have optional boolean arguments `absolute` which control whether spatial data received is using the internal representation (absolute=False) or the user coordinate set (absolute=True). See e.g. `get_vertex_coordinates` on 47.

DEMs, meshes and boundary conditions can have different origins. However, the internal representation in ANUGA will use the origin of the mesh.

8.2.3 Polygons

When generating a mesh it is assumed that polygons do not cross. Having polygons tht cross can cause the mesh generation to fail or bad meshes being produced.

APPENDIX

A

Supporting Tools

This section describes a number of supporting tools, supplied with **ANUGA** v1.0 , that offer a variety of types of functionality and enhance the basic capabilities of **ANUGA** v1.0 .

A.1 caching

The `cache` function is used to provide supervised caching of function results. A Python function call of the form

```
result = func(arg1,...,argn)
```

can be replaced by

```
from caching import cache
result = cache(func,(arg1,...,argn))
```

which returns the same output but reuses cached results if the function has been computed previously in the same context. `result` and the arguments can be simple types, tuples, list, dictionaries or objects, but not unhashable types such as functions or open file objects. The function `func` may be a member function of an object or a module.

This type of caching is particularly useful for computationally intensive functions with few frequently used combinations of input arguments. Note that if the inputs or output are very large caching may not save time because disc access may dominate the execution time.

If the function definition changes after a result has been cached, this will be detected by examining the functions `bytecode (co_code, co_consts, func_defaults, co_argcount)` and the function will be recomputed. However, caching will not detect changes in modules used by `func`. In this case cache must be cleared manually.

Options are set by means of the function `set_option(key, value)`, where `key` is a key associated with a Python dictionary `options`. This dictionary stores settings such as the name of the directory used, the maximum number of cached files allowed, and so on.

The `cache` function allows the user also to specify a list of dependent files. If any of these have been changed, the function is recomputed and the results stored again.

USAGE:

```
result = cache(func, args, kwargs, dependencies, cachedir, verbose,
               compression, evaluate, test, return_filename)
```

A.2 ANUGA viewer - animate

The output generated by **ANUGA** v1.0 may be viewed by means of the visualisation tool `animate`, which takes the `SWW` file output by **ANUGA** v1.0 and creates a visual representation of the data. Examples may be seen in Figures 4.1 and 4.2. To view an `SWW` file with `animate` in the Windows environment, you can simply drag the icon representing the file over an icon on the desktop for the `animate` executable file (or a shortcut to it), or set up a file association to make files with the extension `.sww` open with `animate`. Alternatively, you can operate `animate` from the command line, in both Windows and Linux environments.

On successful operation, you will see an interactive moving-picture display. You can use keys and the mouse to slow down, speed up or stop the display, change the viewing position or carry out a number of other simple operations. Help is also displayed when you press the `h` key.

The main keys operating the interactive screen are:

`w`	toggle wireframe
space bar	start/stop
up/down arrows	increase/decrease speed
left/right arrows	direction in time *(when running)*
	step through simulation *(when stopped)*
left mouse button	rotate
middle mouse button	pan
right mouse button	zoom

The following table describes how to operate animate from the command line:

Usage: `animate [options] swwfile ...`
Options:

`--display <type>`	`MONITOR | POWERWALL | REALITY_CENTER | HEAD_MOUNTED_DISPLAY`
`--rgba`	Request a RGBA colour buffer visual
`--stencil`	Request a stencil buffer visual
`--stereo`	Use default stereo mode which is `ANAGLYPHIC` if not overridden by environmental variable
`--stereo <mode>`	`ANAGLYPHIC | QUAD_BUFFER | HORIZONTAL_SPLIT | VERTICAL_SPLIT | LEFT_EYE | RIGHT_EYE | ON | OFF`
`-alphamax <float 0-1>`	Maximum transparency clamp value
`-alphamin <float 0-1>`	Transparency value at `hmin`
`-cullangle <float angle 0-90>`	Cull triangles steeper than this value
`-help`	Display this information
`-hmax <float>`	Height above which transparency is set to `alphamax`
`-hmin <float>`	Height below which transparency is set to zero
`-lightpos <float>,<float>,<float>`	x, y, z of bedslope directional light (z is up, default is overhead)

`-loop`	Repeated (looped) playback of `.swm` files
`-movie <dirname>`	Save numbered images to named directory and quit
`-nosky`	Omit background sky
`-scale <float>`	Vertical scale factor
`-texture <file>`	Image to use for bedslope topography
`-tps <rate>`	Timesteps per second
`-version`	Revision number and creation (not compile) date

A.3 utilities/polygons

class `Polygon_function`(*regions, default=0.0, geo_reference=None*)

Module: `utilities.polygon`

Creates a callable object that returns one of a specified list of values when evaluated at a point `x, y`, depending on which polygon, from a specified list of polygons, the point belongs to. The parameter `regions` is a list of pairs `(P, v)`, where each `P` is a polygon and each `v` is either a constant value or a function of coordinates `x` and `y`, specifying the return value for a point inside `P`. The optional parameter `default` may be used to specify a value (or a function) for a point not lying inside any of the specified polygons. When a point lies in more than one polygon, the return value is taken to be the value for whichever of these polygon appears later in the list. The optional parameter geo_reference refers to the status of points that are passed into the function. Typically they will be relative to some origin. In ANUGA, a typical call will take the form:

```
set_quantity('elevation',
             Polygon_function([(P1, v1), (P2, v2)],
                              default=v3,
                              geo_reference=domain.geo_reference))
```

`read_polygon`(*filename*)

Module: `utilities.polygon`

Reads the specified file and returns a polygon. Each line of the file must contain exactly two numbers, separated by a comma, which are interpreted as coordinates of one vertex of the polygon.

`populate_polygon`(*polygon, number_of_points, seed = None, exclude = None*)

Module: `utilities.polygon`

Populates the interior of the specified polygon with the specified number of points, selected by means of a uniform distribution function.

`point_in_polygon`(*polygon, delta=1e-8*)

Module: `utilities.polygon`

Returns a point inside the specified polygon and close to the edge. The distance between the returned point and the nearest point of the polygon is less than $\sqrt{2}$ times the second argument `delta`, which is taken as 10^{-8} by default.

`inside_polygon`(*points, polygon, closed = True, verbose = False*)

Module: `utilities.polygon`

Used to test whether the members of a list of points are inside the specified polygon. Returns a Numeric array comprising the indices of the points in the list that lie inside the polygon. (If none of the points are inside, returns `zeros((0,), 'l')`.) Points on the edges of the polygon are regarded as inside if `closed` is set to `True` or omitted; otherwise they are regarded as outside.

`outside_polygon`(*points, polygon, closed = True, verbose = False*)

Module: `utilities.polygon`

Exactly like `inside_polygon`, but with the words 'inside' and 'outside' interchanged.

is_inside_polygon(*point, polygon, closed=True, verbose=False*)
Module: utilities.polygon

Returns True if point is inside polygon or False otherwise. Points on the edges of the polygon are regarded as inside if closed is set to True or omitted; otherwise they are regarded as outside.

is_outside_polygon(*point, polygon, closed=True, verbose=False*)
Module: utilities.polygon

Exactly like is_outside_polygon, but with the words 'inside' and 'outside' interchanged.

point_on_line(*x, y, x0, y0, x1, y1*)
Module: utilities.polygon

Returns True or False, depending on whether the point with coordinates x, y is on the line passing through the points with coordinates x0, y0 and x1, y1 (extended if necessary at either end).

separate_points_by_polygon(*points, polygon, closed = True, verbose = False*)
separate_points_by_polygon Module: utilities.polygon

polygon_area(*polygon*)
Module: utilities.polygon

Returns area of arbitrary polygon (reference http://mathworld.wolfram.com/PolygonArea.html)

plot_polygons(*polygons, style, figname, verbose = False*)
Module: utilities.polygon

Plots each polygon contained in input polygon list, e.g. polygons = [poly1, poly2, poly3] where poly1 = [[x11,y11],[x12,y12],[x13,y13]] etc. Each polygon can be closed for plotting purposes by assigning the style type to each polygon in the list, e.g. style = ['line','line','line']. The default will be a line type when style = None. The subsequent plot will be saved to figname or defaulted to test_image.png. The function returns a list containing the minimum and maximum of x and y, i.e. [x_min, x_max, y_min, y_max].

A.4 coordinate_transforms

A.5 geospatial_data

This describes a class that represents arbitrary point data in UTM coordinates along with named attribute values.

class Geospatial_data(*data_points = None, attributes = None, geo_reference = None, default_attribute_name = None, file_name = None*)
Module: geospatial_data

This class is used to store a set of data points and associated attributes, allowing these to be manipulated by methods defined for the class.

The data points are specified either by reading them from a NetCDF or CSV file, identified through the parameter file_name, or by providing their x- and y-coordinates in metres, either as a sequence of 2-tuples of floats or as an $M \times 2$ Numeric array of floats, where M is the number of points. Coordinates are interpreted relative to the origin specified by the object geo_reference, which contains data indicating the UTM zone, easting and northing. If geo_reference is not specified, a default is used.

Attributes are specified through the parameter attributes, set either to a list or array of length M or to a dictionary whose keys are the attribute names and whose values are lists or arrays of length M. One of the attributes may be specified as the default attribute, by assigning its name to default_attribute_name. If no value is specified, the default attribute is taken to be the first one.

Note that the Geospatial_data object currently reads entire datasets into memory i.e. no memomry blocking takes place. For this we refer to the set_quantity method which will read .pts and .csv files into **ANUGA** v1.0 using memory blocking allowing large files to be used.

import_points_file(*delimiter = None, verbose = False*)

export_points_file(*ofile, absolute=False*)

get_data_points(*absolute = True, as_lat_long = False*)
If `as_lat_long` is`True` the point information returned will be in Latitudes and Longitudes.

set_attributes(*attributes*)

get_attributes(*attribute_name = None*)

get_all_attributes()

set_default_attribute_name(*default_attribute_name*)

set_geo_reference(*geo_reference*)

add()

clip()
Clip geospatial data by a polygon

Inputs are `polygon` which is either a list of points, an Nx2 array or a Geospatial data object and `closed`(optional) which determines whether points on boundary should be regarded as belonging to the polygon (`closed=True`) or not (`closed=False`). Default is `closed=True`.

Returns new Geospatial data object representing points inside specified polygon.

clip_outside()
Clip geospatial data by a polygon

Inputs are `polygon` which is either a list of points, an Nx2 array or a Geospatial data object and `closed`(optional) which determines whether points on boundary should be regarded as belonging to the polygon (`closed=True`) or not (`closed=False`). Default is `closed=True`.

Returns new Geospatial data object representing points *out*side specified polygon.

split(*factor=0.5, seed_num=None, verbose=False*)
Returns two geospatial_data object, first is the size of the 'factor' smaller the original and the second is the remainder. The two new object are disjoin set of each other.

Points of the two new geospatial_data object are selected RANDOMLY.

Input - the (`factor`) which to split the object, if 0.1 then 10together object will be returned

Output - two geospatial_data objects that are disjoint sets of the original

find_optimal_smoothing_parameter(*data_file, alpha_list=None, mesh_file=None, boundary_poly=None, mesh_resolution=100000, north_boundary=None, south_boundary=None, east_boundary=None, west_boundary=None, plot_name='all_alphas', split_factor=0.1, seed_num=None, cache=False, verbose=False*)
Removes a small random sample of points from 'data_file'. Creates models from resulting points in 'data_file' with different alpha values from 'alpha_list' and cross validates the predicted value to the previously removed point data. Returns the alpha value which has the smallest covariance.

data_file: must not contain points outside the boundaries defined and it either a pts, txt or csv file.

alpha_list: the alpha values to test in a single list

mesh_file: name of the created mesh file or if passed in will read it. NOTE, if there is a mesh file mesh_resolution, north_boundary, south... etc will be ignored.

mesh_resolution: the maximum area size for a triangle

north_boundary... west_boundary: the value of the boundary

plot_name: the name for the plot contain the results

seed_num: the seed to the random number generator

USAGE: convariance_value, alpha = find_optimal_smoothing_parameter(data_file=fileName, alpha_list=[0.0001, 0.01, 1], mesh_file=None, mesh_resolution=3, north_boundary=5, south_boundary=-5, east_boundary=5, west_-boundary=-5, plot_name='all_alphas', seed_num=100000, verbose=False)

OUTPUT: returns the minumum normalised covalance calculate AND the alpha that created it. PLUS writes a plot of the results

NOTE: code will not work if the data_file extent is greater than the boundary_polygon or any of the boundaries, eg north_boundary...west_boundary

A.6 Graphical Mesh Generator GUI

The program `graphical_mesh_generator.py` in the pmesh module allows the user to set up the mesh of the problem interactively. It can be used to build the outline of a mesh or to visualise a mesh automatically generated.

Graphical Mesh Generator will let the user select various modes. The current allowable modes are vertex, segment, hole or region. The mode describes what sort of object is added or selected in response to mouse clicks. When changing modes any prior selected objects become deselected.

In general the left mouse button will add an object and the right mouse button will select an object. A selected object can de deleted by pressing the the middle mouse button (scroll bar).

A.7 alpha_shape

Alpha shapes are used to generate close-fitting boundaries around sets of points. The alpha shape algorithm produces a shape that approximates to the 'shape formed by the points'—or the shape that would be seen by viewing the points from a coarse enough resolution. For the simplest types of point sets, the alpha shape reduces to the more precise notion of the convex hull. However, for many sets of points the convex hull does not provide a close fit and the alpha shape usually fits more closely to the original point set, offering a better approximation to the shape being sought.

In **ANUGA** v1.0 , an alpha shape is used to generate a polygonal boundary around a set of points before mesh generation. The algorithm uses a parameter α that can be adjusted to make the resultant shape resemble the shape suggested by intuition more closely. An alpha shape can serve as an initial boundary approximation that the user can adjust as needed.

The following paragraphs describe the class used to model an alpha shape and some of the important methods and attributes associated with instances of this class.

class `Alpha_Shape`(*points, alpha = None*)
Module: `alpha_shape`

To instantiate this class the user supplies the points from which the alpha shape is to be created (in the form of a list of 2-tuples `[[x1, y1],[x2, y2]...]`, assigned to the parameter `points`) and, optionally, a value for the parameter `alpha`. The alpha shape is then computed and the user can then retrieve details of the boundary through the attributes defined for the class.

`alpha_shape_via_files`(*point_file, boundary_file, alpha= None*)
Module: `alpha_shape`

This function reads points from the specified point file `point_file`, computes the associated alpha shape (either using the specified value for `alpha` or, if no value is specified, automatically setting it to an optimal value) and outputs the boundary to a file named `boundary_file`. This output file lists the coordinates `x, y` of each point in the boundary, using one line per point.

`set_boundary_type`(*self,raw_boundary=True, remove_holes=False, smooth_indents=False, expand_-pinch=False, boundary_points_fraction=0.2*)
Module: `alpha_shape`, Class: `Alpha_Shape`

This function sets flags that govern the operation of the algorithm that computes the boundary, as follows:

`raw_boundary = True` returns raw boundary, i.e. the regular edges of the alpha shape.
`remove_holes = True` removes small holes ('small' is defined by `boundary_points_fraction`)
`smooth_indents = True` removes sharp triangular indents in boundary
`expand_pinch = True` tests for pinch-off and corrects—preventing a boundary vertex from having more than two edges.

`get_boundary``()`
Module: `alpha_shape`, Class: `Alpha_Shape`

Returns a list of tuples representing the boundary of the alpha shape. Each tuple represents a segment in the boundary by providing the indices of its two endpoints.

`write_boundary``(`*file_name*`)`
Module: `alpha_shape`, Class: `Alpha_Shape`

Writes the list of 2-tuples returned by `get_boundary` to the file `file_name`, using one line per tuple.

A.8 Numerical Tools

The following table describes some useful numerical functions that may be found in the module `utilities.numerical_tools`:

`angle(v1, v2=None)`	Angle between two-dimensional vectors `v1` and `v2`, or between `v1` and the x-axis if `v2` is `None`. Value is in range 0 to 2π.
`normal_vector(v)`	Normal vector to `v`.
`mean(x)`	Mean value of a vector `x`.
`cov(x, y=None)`	Covariance of vectors `x` and `y`. If `y` is `None`, returns `cov(x, x)`.
`err(x, y=0, n=2, relative=True)`	Relative error of $\|x-y\|$ to $\|y\|$ (2-norm if `n` = 2 or Max norm if `n` = `None`). If denominator evaluates to zero or if `y` is omitted or if `relative = False`, absolute error is returned.
`norm(x)`	2-norm of `x`.
`corr(x, y=None)`	Correlation of `x` and `y`. If `y` is `None` returns autocorrelation of `x`.
`ensure_numeric(A, typecode = None)`	Returns a Numeric array for any sequence `A`. If `A` is already a Numeric array it will be returned unaltered. Otherwise, an attempt is made to convert it to a Numeric array. (Needed because `array(A)` can cause memory overflow.)
`histogram(a, bins, relative=False)`	Standard histogram. If `relative` is `True`, values will be normalised against the total and thus represent frequencies rather than counts.
`create_bins(data, number_of_bins = None)`	Safely create bins for use with histogram. If `data` contains only one point or is constant, one bin will be created. If `number_of_bins` is omitted, 10 bins will be created.

A.9 Finding the Optimal Alpha Value

The function ???? more to come very soon

APPENDIX

B

Modules available in **ANUGA** v1.0

B.1 abstract_2d_finite_volumes.general_mesh

B.2 abstract_2d_finite_volumes.neighbour_mesh

B.3 abstract_2d_finite_volumes.domain

Generic module for 2D triangular domains for finite-volume computations of conservation laws

B.4 abstract_2d_finite_volumes.quantity

```
Class Quantity - Implements values at each triangular element

To create:

   Quantity(domain, vertex_values)

   domain: Associated domain structure. Required.

   vertex_values: N x 3 array of values at each vertex for each element.
                  Default None

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

B.5 shallow_water

2D triangular domains for finite-volume computations of the shallow water wave equation. This module contains a specialisation of class Domain from module domain.py consisting of methods specific to the Shallow Water Wave Equation

APPENDIX

C

ANUGA Full-scale Validations

C.1 Overview

There are some long-running validations that are not included in the small-scale validations that run when you execute the `validate_all.py` script in the `anuga_validation/automated_validation_test` directory. These validations are not run automatically since they can take a large amount of time and require an internet connection and user input.

C.2 Patong Beach

The Patong Beach validation is run from the `automated_validation_tests/patong_beach_validation` directory. Just execute the `validate_patong.py` script in that directory. This will run a Patong Beach simulation and compare the generated SWW file with a known good copy of that file.

The script attempts to refresh the validation data files from master copies held on the Sourceforge project site. If you don't have an internet connection you may still run the validation, as long as you have the required files.

You may download the validation data files by hand and then run the validation. Just go to the ANUGA Sourceforge project download page at `http://sourceforge.net/project/showfiles.php?group_id=172848` and select the `validation_data` package, `patong-1.0` release. You need the `data.tgz` file and one or more of the `patong.sww.{BASIC|TRIAL|FINAL}.tgz` files.

The BASIC validation is the quickest and the FINAL validation is the slowest. The `validate.py` script will use whatever files you have, BASIC first and FINAL last.

APPENDIX

D

Frequently Asked Questions

The Frequently Asked Questions have been move to the online FAQ at:

https://datamining.anu.edu.au/anuga/wiki/FrequentlyAskedQuestions

APPENDIX

E

Glossary

Term	*Definition*	*Page*
ANUGA v1.0	Name of software (joint development between ANU and GA)	i
bathymetry	offshore elevation	
conserved quantity	conserved (stage, x and y momentum)	
Digital Elevation Model (DEM)	DEMs are digital files consisting of points of elevations, sampled systematically at equally spaced intervals.	
Dirichlet boundary	A boundary condition imposed on a differential equation that specifies the values the solution is to take on the boundary of the domain.	12
edge	A triangular cell within the computational mesh can be depicted as a set of vertices joined by lines (the edges).	
elevation	refers to bathymetry and topography	
evolution	integration of the shallow water wave equations over time	
finite volume method	The method evaluates the terms in the shallow water wave equation as fluxes at the surfaces of each finite volume. Because the flux entering a given volume is identical to that leaving the adjacent volume, these methods are conservative. Another advantage of the finite volume method is that it is easily formulated to allow for unstructured meshes. The method is used in many computational fluid dynamics packages.	
forcing term		
flux	the amount of flow through the volume per unit time	
grid	Evenly spaced mesh	
latitude	The angular distance on a mericlear north and south of the equator, expressed in degrees and minutes.	
longitude	The angular distance east or west, between the meridian of a particular place on Earth and that of the Prime Meridian (located in Greenwich, England) expressed in degrees or time.	
Manning friction coefficient		
mesh	Triangulation of domain	
mesh file	A TSH or MSH file	27
NetCDF		
node	A point at which edges meet	
northing	A rectangular (x,y) coordinate measurement of distance north from a north-south reference line, usually a meridian used as the axis of origin within a map zone or projection. Northing is a UTM (Universal Transverse Mercator) coordinate.	
points file	A PTS or CSV file	

polygon	A sequence of points in the plane. **ANUGA** v1.0 represents a polygon either as a list consisting of Python tuples or lists of length 2 or as an $N \times 2$ Numeric array, where N is the number of points. The unit square, for example, would be represented either as `[ [0,0], [1,0], [1,1], [0,1] ]` or as `array( [0,0], [1,0], [1,1], [0,1] )`. NOTE: For details refer to the module `utilities/polygon.py`.	
resolution	The maximal area of a triangular cell in a mesh	
reflective boundary	Models a solid wall. Returns same conserved quantities as those present in the neighbouring volume but reflected. Specific to the shallow water equation as it works with the momentum quantities assumed to be the second and third conserved quantities.	11
stage		
animate	visualisation tool used with **ANUGA** v1.0	76
time boundary	Returns values for the conserved quantities as a function of time. The user must specify the domain to get access to the model time.	12
topography	onshore elevation	
transmissive boundary		12
vertex	A point at which edges meet.	
xmomentum	conserved quantity (note, two-dimensional SWW equations say only `x` and `y` and NOT `z`)	
ymomentum	conserved quantity	

INDEX

INDEX

BIBLIOGRAPHY

[nielsen2005] *Hydrodynamic modelling of coastal inundation*. Nielsen, O., S. Roberts, D. Gray, A. McPherson and A. Hitchman. In Zerger, A. and Argent, R.M. (eds) MODSIM 2005 International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand, December 2005, pp. 518-523. ISBN: 0-9758400-2-9.
http://www.mssanz.org.au/modsim05/papers/nielsen.pdf

[grid250] Australian Bathymetry and Topography Grid, June 2005. Webster, M.A. and Petkovic, P. Geoscience Australia Record 2005/12. ISBN: 1-920871-46-2.
http://www.ga.gov.au/meta/ANZCW0703008022.html

[ZR1999] Catastrophic Collapse of Water Supply Reservoirs in Urban Areas. C. Zoppou and S. Roberts. *ASCE J. Hydraulic Engineering*, 125(7):686–695, 1999.

[Toro1999] Riemann problems and the waf method for solving the two-dimensional shallow water equations. E. F. Toro. *Philosophical Transactions of the Royal Society, Series A*, 338:43–68, 1992.

[1] Semidiscrete central-upwind schemes for hyperbolic conservation laws and hamilton-jacobi equations. A. Kurganov, S. Noelle, and G. Petrova. *SIAM Journal of Scientific Computing*, 23(3):707–740, 2001.

99