OpenCLSim Documentation

Release 1.2.3

Mark van Koningsveld

Oct 19, 2020

Contents:

OpenCLSim is a python package for rule driven scheduling of cyclic activities for in-depth comparison of alternative operating strategies

Welcome to OpenCLSim documentation! Please check the contents below for information on installation, getting started and actual example code. If you want to dive straight into the code you can check out our GitHub page or the working examples presented in Jupyter Notebooks.

CHAPTER 1

Installation

1.1 Stable release

To install OpenCLSim, run this command in your terminal:

```
# Use pip to install OpenCLSim
pip install openclsim
```

This is the preferred method to install OpenCLSim, as it will always install the most recent stable release.

If you do not pip installed, this Python installation guide can guide you through the process.

1.2 From sources

The sources for OpenCLSim can be downloaded from the Github repo.

You can either clone the public repository:

```
# Use git to clone OpenCLSim
git clone git://github.com/TUDelft-CITG/OpenCLSim
```

Or download the tarball:

```
# Use curl to obtain the tarball
curl  -OL https://github.com/TUDelft-CITG/OpenCLSim/tarball/master
```

Once you have a copy of the source, you can install it with:

```
# Use python to install
python setup.py install
```

CHAPTER 2

Usage

2.1 Import required components

To use OpenCLSim in a project you have to import the following three components:

```
# Import openclsim for the logistical components
import openclsim.model as model
import openclsim.core as core

# Import simpy for the simulation environment
import simpy
```

2.2 Using Mixins and Metaclasses

The Open Complex Logistics Simulation package is developed with the goal of reusable and generic components in mind. A new class can be instatiated by combining mixins from the *openclsim.core*, such as presented below. The following lines of code demonstrate how a containervessel can be defined:

```
# Define the core components
# A generic class for an object that can move and transport material
ContainerVessel = type('ContainerVessel',
                       (core.Identifiable,             # Give it a name and unique UUID
                        core.Log,                      # Allow logging of all discrete␣
↪events
                        core.ContainerDependentMovable,# It can transport an amount
                        core.HasResource,              # Add information on serving␣
↪equipment
                        ),
                       {})

# The next step is to define all the required parameters for the defined metaclass
```

(continues on next page)

(continued from previous page)

```
# For more realistic simulation you might want to have speed dependent on the filling␣
↪degree
v_full  = 8      # meters per second
v_empty = 5      # meters per second

def variable_speed(v_empty, v_full):
    return lambda x: x * (v_full - v_empty) + v_empty

# Other variables
data_vessel = {
            "env": simpy.Environment(),                    # The simpy environment
            "name": "Vessel 01",                           # Name
            "geometry": shapely.geometry.Point(0, 0),      # The lat, lon coordinates
            "capacity": 5_000,                             # Capacity of the vessel
            "compute_v": variable_speed(v_empty, v_full), # Variable speed
            }

# Create an object based on the metaclass and vessel data
vessel_01 = ContainerVessel(**data_vessel)
```

For more elaboration and examples please check the examples documentation. In-depth Jupyter Notebooks can also be used.

CHAPTER 3

Examples

This small example guide will cover the basic start-up and the three main elements of the OpenClSim package:

- Start-Up (Minimal set-up)
- Locations (Sites or stockpiles)
- Resources (Processors and transporters)
- Activities (Rule-based operations)

Once the elements above are explained some small simulations examples are presented.

3.1 Start-Up

The first part of every OpenClSim simulation is to import the required libraries and to initiate the simulation environment.

3.1.1 Required Libraries

Depending on the simulation it might be required to import additional libraries. The minimal set-up of an OpenCLSim project has the following import statements:

```
# Import openclsim for the logistical components
import openclsim.model as model
import openclsim.core as core

# Import simpy for the simulation environment
import simpy
```

3.1.2 Simulation Environment

OpenClSim continues on the SimPy discrete event simulation package. Some components are modified, such as the resources and container objects, but the simulation environment is pure SimPy. Starting the simulation environment can be done with the following line of code. For more information in SimPy environment please refer to the SimPy documentation.

```
# Start the SimPy environment
env = simpy.Environment()
```

3.2 Locations

Basic processes do not require a location but more comprehensive simulations do. Locations are added to an OpenCLSim environment so that it becomes possible to track all events in both time and space. If a site is initiated with a container it can store materials as well. Adding a processor allows the location to load or unload as well.

3.2.1 Basic Location

The code below illustrates how a basic location can be created using OpenClSim. Such a location can be used to add information on events in space, such as tracking movement events or creating paths to follow.

```
# Import the library required to add coordinates
import shapely.geometry

# Create a location class
Location = type(
    "Location",
    (
        core.Identifiable,  # Give it a name and unique UUID
        core.Log,           # To keep track of all events
        core.HasResource,   # Add information on the number of resources
        core.Locatable,     # Add coordinates to extract distance information
    ),
    {},
)

location_data = {
    "env": env,                                 # The SimPy environment
    "name": "Location 01",                      # Name of the location
    "geometry": shapely.geometry.Point(0, 0), # The lat, lon coordinates
}

location_01 = Location(**location_data)
```

3.2.2 Storage Location

The code below illustrates how a location can be created that is capable of storing an amount. Such a location can be used by the OpenClSim.model activities as origin or destination.

```
# Import the library required to add coordinates
import shapely.geometry
```

(continues on next page)

(continued from previous page)

```
# Create a location class
StorageLocation = type(
    "StorageLocation",
    (
        core.Identifiable,  # Give it a name and unique UUID
        core.Log,           # To keep track of all events
        core.HasResource,   # Add information on the number of resources
        core.Locatable,     # Add coordinates to extract distance information
        core.HasContainer,  # Add information on storage capacity
    ),
    {},
)

location_data = {
    "env": env,                                # The SimPy environment
    "name": "Location 02",                     # Name of the location
    "geometry": shapely.geometry.Point(0, 0), # The lat, lon coordinates
    "capacity": 10_000,                        # The maximum number of units
    "level": 10_000,                           # The number of units in the location
}

location_02 = StorageLocation(**location_data)
```

3.2.3 Processing Storage Location

The code below illustrates how a location can be created that is capable of storing an amount. Additional to the storage location, a processing- and storage location can be used as both the origin and loader or destination and unloader in a OpenClSim.model activity.

```
# Import the library required to add coordinates
import shapely.geometry

# Create a location class
ProcessingStorageLocation = type(
    "ProcessingStorageLocation",
    (
        core.Identifiable,  # Give it a name and unique UUID
        core.Log,           # To keep track of all events
        core.HasResource,   # Add information on the number of resources
        core.Locatable,     # Add coordinates to extract distance information
        core.HasContainer,  # Add information on storage capacity
        core.Processor,     # Add information on processing
    ),
    {},
)

# Create a processing function
processing_rate = lambda x: x

location_data = {
    "env": env,                                # The SimPy environment
    "name": "Location 03",                     # Name of the location
    "geometry": shapely.geometry.Point(0, 1), # The lat, lon coordinates
    "capacity": 10_000,                        # The maximum number of units
```

(continues on next page)

(continued from previous page)

```
    "level": 0,                                 # The number of units in the location
    "loading_func": processing_rate,            # Loading rate of 1 unit per 1 unit time
    "unloading_func": processing_rate,          # Unloading rate of 1 unit per 1 unit␣
↪time
}

location_03 = ProcessingStorageLocation(**location_data)
```

Optionally a *OpenCLSim.core.Log* mixin can be added to all locations to keep track of all the events that are taking place.

3.3 Resources

OpenCLSim resources can be used to process and transport units. The OpenCLSim.model activity class requires a loader, an unloader and a mover, this are examples of resources. A resource will always interact with another resource in an OpenClSim.model activity, but it is possible to initiate a simpy process to keep track of a single resource.

3.3.1 Processing Resource

An example of a processing resource is a harbour crane, it processes units from a storage location to a transporting resource or vice versa. In the OpenClSim.model activity such a processing resource could be selected as the loader or unloader. The example code is presented below.

```
# Create a resource
ProcessingResource = type(
    "ProcessingResource",
    (
        core.Identifiable,  # Give it a name and unique UUID
        core.Log,           # To keep track of all events
        core.HasResource,   # Add information on the number of resources
        core.Locatable,     # Add coordinates to extract distance information
        core.Processor,     # Add information on processing
    ),
    {},
)

# The next step is to define all the required parameters for the defined metaclass
# Create a processing function
processing_rate = lambda x: x

resource_data = {
    "env": env,                         # The SimPy environment
    "name": "Resource 01",              # Name of the location
    "geometry": location_01.geometry,   # The lat, lon coordinates
    "loading_func": processing_rate,    # Loading rate of 1 unit per 1 unit time
    "unloading_func": processing_rate,  # Unloading rate of 1 unit per 1 unit time
}

# Create an object based on the metaclass and vessel data
resource_01 = ProcessingResource(**resource_data)
```

3.3.2 Transporting Resource

A harbour crane will service transporting resources. To continue with the harbour crane example, basically any vessel is a transporting resource because it is capable of moving units from location A to location B. In the OpenClSim.model activity such a processing resource could be selected as the mover.

```
# Create a resource
TransportingResource = type(
    "TransportingResource",
    (
        core.Identifiable,              # Give it a name and unique UUID
        core.Log,                       # To keep track of all events
        core.HasResource,               # Add information on the number of resources
        core.ContainerDependentMovable, # It can transport an amount
    ),
    {},
)

# The next step is to define all the required parameters for the defined metaclass
# For more realistic simulation you might want to have speed dependent on the filling␣
↪degree
v_full = 8  # meters per second
v_empty = 5  # meters per second

def variable_speed(v_empty, v_full):
    return lambda x: x * (v_full - v_empty) + v_empty

# Other variables
resource_data = {
    "env": env,                                     # The SimPy environment
    "name": "Resource 02",                          # Name of the location
    "geometry": location_01.geometry,               # The lat, lon coordinates
    "capacity": 5_000,                              # Capacity of the vessel
    "compute_v": variable_speed(v_empty, v_full), # Variable speed
}

# Create an object based on the metaclass and vessel data
resource_02 = TransportingResource(**resource_data)
```

3.3.3 Transporting Processing Resource

Finally, some resources are capable of both processing and moving units. Examples are dredging vessels or container vessels with deck cranes. These specific vessels have the unique property that they can act as the loader, unloader and mover in the OpenClSim.model activity.

```
# Create a resource
TransportingProcessingResource = type(
    "TransportingProcessingResource",
    (
        core.Identifiable,              # Give it a name and unique UUID
        core.Log,                       # To keep track of all events
        core.HasResource,               # Add information on the number of resources
        core.ContainerDependentMovable, # It can transport an amount
        core.Processor,                 # Add information on processing
    ),
    {},
```

(continues on next page)

(continued from previous page)

```
)

# The next step is to define all the required parameters for the defined metaclass
# For more realistic simulation you might want to have speed dependent on the filling␣
↪degree
v_full = 8  # meters per second
v_empty = 5  # meters per second

def variable_speed(v_empty, v_full):
    return lambda x: x * (v_full - v_empty) + v_empty

# Create a processing function
processing_rate = lambda x: x

# Other variables
resource_data = {
    "env": env,                                     # The SimPy environment
    "name": "Resource 03",                          # Name of the location
    "geometry": location_01.geometry,               # The lat, lon coordinates
    "capacity": 5_000,                              # Capacity of the vessel
    "compute_v": variable_speed(v_empty, v_full), # Variable speed
    "loading_func": processing_rate,                # Loading rate of 1 unit per 1 unit␣
↪time
    "unloading_func": processing_rate,              # Unloading rate of 1 unit per 1␣
↪unit time
}

# Create an object based on the metaclass and vessel data
resource_03 = TransportingProcessingResource(**resource_data)
```

3.4 Simulations

The code below will start the simulation if SimPy processes are added to the environment. These SimPy processes can be added using a combination of SimPy and OpenCLSim, or by using OpenCLSim activities.

```
env.run()
```

3.4.1 SimPy processes

A SimPy process can be initiated using the code below. The code below will instruct Resource 02, which was a TransportingResource, to sail from Location 01 (at Lat, Long (0, 0)) to Location 02 (at Lat, Long (0, 1)). The simulation will stop as soon as Resource 02 is at Location 02.

```
# Create the process function
def move_resource(mover, destination):

    # the is_at function is part of core.Movable
    while not mover.is_at(destination):

      # the move function is part of core.Movable
      yield from mover.move(destination)
```

(continues on next page)

(continued from previous page)

```
# Add to the SimPy environment
env.process(move_resource(resource_02, location_03))

# Run the simulation
env.run()
```

3.4.2 Unconditional Activities

Activities are at the core of what OpenCLSim adds to SimPy, an activity is a collection of SimPy Processes. These activities schedule cyclic events, which could be production or logistical processes and, but the current OpenCLSim.model.activity assumes the following cycle:

- Loading
- Transporting
- Unloading
- Transporting

This cycle is repeated until a certain condition is met. Between the individual components of the cycle waiting events can occur due to arising queues, equipment failure or weather events. The minimal input for an activity is listed below.

- Origin
- Destination
- Loader
- Mover
- Unloader

If no additional input is provided, the cyclic process will be repeated until either the origin is empty or the destination is full. The example activity below will stop after two cycles because the origin will be empty and the destination will be full.

```
# Define the activity
activity_01 = model.Activity(
    env=env,                    # The simpy environment defined in the first cel
    name="Activity 01",         # Name of the activity
    origin=location_02,         # Location 02 was filled with 10_000 units
    destination=location_03,    # Location 03 was empty
    loader=resource_03,         # Resource 03 could load
    mover=resource_03,          # Resource 03 could move
    unloader=resource_03,       # Resource 03 could unload
)

# Run the simulation
env.run()
```

3.4.3 Conditional Activities

Additionally, start and stop events can be added to the activity. The process will only start as soon as a start event (or a list of start events) is completed and it will stop as soon as the stop event (or a list of stop events) are completed. These can be any SimPy event, such as a time-out, but OpenClSim provides some additional events as well, such as empty-

or full events. The activity in the example below will start as soon as the previous activity is finished, but not sooner than 2 days after the simulation is started.

```
# Activity starts after both
#  - Activity 01 is finished
#  - A minimum of 2 days after the simulation starts
start_event = [activity_01.main_process, env.timeout(2 * 24 * 3600)]

# Define the activity
activity_02 = model.Activity(
    env=env,                    # The simpy environment defined in the first cel
    name="Activity 02",         # Name of the activity
    origin=location_03,         # Location 03 will be filled
    destination=location_02,    # Location 02 will be empty
    loader=resource_03,         # Resource 03 could load
    mover=resource_03,          # Resource 03 could move
    unloader=resource_03,       # Resource 03 could unload
    start_event=start_event,    # Start Event
)

# Run the simulation
env.run()
```

CHAPTER 4

OpenCLSim

This page lists all functions and classes available in the OpenCLSim.model and OpenCLSim.core modules. For examples on how to use these submodules please check out the Examples page, information on installing OpenCLSim can be found on the Installation page.

4.1 Submodules

The main components are the Model module and the Core module. All of their components are listed below.

4.2 openclsim.model module

class openclsim.model.**Activity**(*origin*, *destination*, *loader*, *mover*, *unloader*, *start_event=None*, *stop_event=None*, *show=False*, **args*, ***kwargs*)

Bases: *openclsim.core.Identifiable*, *openclsim.core.Log*

The Activity Class forms a specific class for a single activity within a simulation. It deals with a single origin container, destination container and a single combination of equipment to move substances from the origin to the destination. It will initiate and suspend processes according to a number of specified conditions. To run an activity after it has been initialized call env.run() on the Simpy environment with which it was initialized.

To check when a transportation of substances can take place, the Activity class uses three different condition arguments: start_condition, stop_condition and condition. These condition arguments should all be given a condition object which has a satisfied method returning a boolean value. True if the condition is satisfied, False otherwise.

origin: object inheriting from HasContainer, HasResource, Locatable, Identifiable and Log destination: object inheriting from HasContainer, HasResource, Locatable, Identifiable and Log loader: object which will get units from 'origin' Container and put them into 'mover' Container

should inherit from Processor, HasResource, Identifiable and Log after the simulation is complete, its log will contain entries for each time it started loading and stopped loading

mover: moves to 'origin' if it is not already there, is loaded, then moves to 'destination' and is unloaded should inherit from Movable, HasContainer, HasResource, Identifiable and Log after the simulation is complete, its log will contain entries for each time it started moving, stopped moving, started loading / unloading and stopped loading / unloading

unloader: gets amount from 'mover' Container and puts it into 'destination' Container should inherit from Processor, HasResource, Identifiable and Log after the simulation is complete, its log will contain entries for each time it started unloading and stopped unloading

start_event: the activity will start as soon as this event is triggered by default will be to start immediately

stop_event: the activity will stop (terminate) as soon as this event is triggered by default will be an event triggered when the destination container becomes full or the source container becomes empty

class openclsim.model.**Simulation**(*sites*, *equipment*, *activities*, **args*, ***kwargs*)
Bases: *openclsim.core.Identifiable*, *openclsim.core.Log*

The Simulation Class can be used to set up a full simulation using configuration dictionaries (json).

sites: a list of dictionaries specifying which site objects should be constructed equipment: a list of dictionaries specifying which equipment objects should be constructed activities: list of dictionaries specifying which activities should be performed during the simulation

Each of the values the sites and equipment lists, are a dictionary specifying "id", "name", "type" and "properties". Here "id" can be used to refer to this site / equipment in other parts of the configuration, "name" is used to initialize the objects name (required by core.Identifiable). The "type" must be a list of mixin class names which will be used to construct a dynamic class for the object. For example: ["HasStorage", "HasResource", "Locatable"]. The core.Identifiable and core.Log class will always be added automatically by the Simulation class. The "properties" must be a dictionary which is used to construct the arguments for initializing the object. For example, if "HasContainer" is included in the "type" list, the "properties" dictionary must include a "capacity" which has the value that will be passed to the constructor of HasContainer. In this case, the "properties" dictionary can also optionally specify the "level".

Each of the values of the activities list, is a dictionary specifying an "id", "type", and other fields depending on the type. The supported types are "move", "single_run", "sequential", "conditional", and "delayed". For a "move" type activity, the dictionary should also contain a "mover", "destination" and can optionally contain a "moverProperties" dictionary containing an "engineOrder". For a "single_run" type activity, the dictionary should also contain an "origin", "destination", "loader", "mover", "unloader" and can optionally contain a "moverProperties" dictionary containing an "engineOrder" and/or "load". For a "sequential" type activity, the dictionary should also contain "activities". This is a list off activities (dictionaries as before) which will be performed until sequentially in the order in which they appear in the list. For a "conditional" type activity, the dictionary should also contain a "condition" and "activities", where the "activities" is another list of activities which will be performed until the event corresponding with the condition occurs. For a "delayed" type activity, the dictionary should also contain a "condition" and "activities", where the "activities" is another list of activities which will be performed after the event corresponding with the condition occurs.

The "condition" of a "conditional" or "delayed" type activity is a dictionary containing an "operator" and one other field depending on the type. The operator can be "is_full", "is_empty", "is_done", "any_of" and "all_of". For the "is_full" operator, the dictionary should contain an "operand" which must be the id of the object (site or equipment) of which the container should be full for the event to occur. For the "is_empty" operator, the dictionary should contain an "operand" which must be the id of the object (site or equipment) of which the container should be empty for the event to occur. For the "is_done" operator, the dictionary should contain an "operand" which must the the id of an activity which should be finished for the event to occur. To instantiate such an event, the operand activity must already be instantiated. The Simulation class takes care of instantiating its activities in an order which ensures this is the case. However, if there is no such order because activities contain "is_done" conditions which circularly reference each other, a ValueError will be raised. For the "any_of" operator, the dictionary should contain "conditions", a list of (sub)conditions of which any must occur for the

event to occur. For the "all_of" operator, the dictionary should contain "conditions", a list of (sub)conditions which all must occur for the event to occur.

static get_as_feature_collection(*id*, *features*)

get_condition_event(*condition*)

get_level_event_operand(*condition*)

get_logging()

static get_mover_properties_kwargs(*activity*)

get_process_control(*activity*, *stop_reservation_waiting_event=None*)

get_sub_condition_events(*condition*)

openclsim.model.**add_object_properties**(*new_object*, *properties*)

openclsim.model.**conditional_process**(*activity_log*, *env*, *stop_event*, *sub_processes*)

Returns a generator which can be added as a process to a simpy.Environment. In the process the given sub_processes will be executed until the given stop_event occurs. If the stop_event occurs during the execution of the sub_processes, the conditional process will first complete all sub_processes (which are executed sequentially in the order in which they are given), before finishing its own process.

activity_log: the core.Log object in which log_entries about the activities progress will be added. env: the simpy.Environment in which the process will be run stop_event: a simpy.Event object, when this event occurs, the conditional process will finish executing its current

run of its sub_processes and then finish

sub_processes: an Iterable of methods which will be called with the activity_log and env parameters and should

return a generator which could be added as a process to a simpy.Environment the sub_processes will be executed sequentially, in the order in which they are given as long as the stop_event has not occurred.

openclsim.model.**delayed_process**(*activity_log*, *env*, *start_event*, *sub_processes*)

"Returns a generator which can be added as a process to a simpy.Environment. In the process the given sub_processes will be executed after the given start_event occurs.

activity_log: the core.Log object in which log_entries about the activities progress will be added. env: the simpy.Environment in which the process will be run start_event: a simpy.Event object, when this event occurs the delayed process will start executing its sub_processes sub_processes: an Iterable of methods which will be called with the activity_log and env parameters and should

return a generator which could be added as a process to a simpy.Environment the sub_processes will be executed sequentially, in the order in which they are given after the start_event occurs

openclsim.model.**energy_use_processing**(*duration_seconds*, *constant_hourly_use*)

openclsim.model.**energy_use_sailing**(*distance*, *current_speed*, *filling_degree*, *speed_max_full*, *speed_max_empty*, *propulsion_power_max*, *boardnet_power*)

openclsim.model.**get_class_from_type_list**(*class_name*, *type_list*)

openclsim.model.**get_compute_function**(*table_entry_list*, *x_key*, *y_key*)

openclsim.model.**get_kwargs_from_properties**(*environment*, *name*, *properties*, *sites*)

openclsim.model.**get_loading_func**(*property*)

Returns a loading_func based on the given input property. Input can be a flat rate or a table defining the rate depending on the level. In the second case, note that by definition the rate is the derivative of the level with respect to time. Therefore d level / dt = f(level), from which we can obtain that the time taken for loading can be calculated by integrating 1 / f(level) from current_level to desired_level.

openclsim.model.**get_spill_condition_kwargs**(*environment*, *condition_dict*)

openclsim.model.**get_unloading_func**(*property*)

Returns an unloading_funct based on the given input property. Input can be a flat rate or a table defining the rate depending on the level. In the second case, note that by definition the rate is -1 times the derivative of the level with respect to time. Therefore d level / dt = - f(level), from which we can obtain the the time taken for unloading can be calculated by integrating 1 / f(level) from desired_level to current_level.

openclsim.model.**move_process**(*activity_log*, *env*, *mover*, *destination*, *engine_order=1.0*)

Returns a generator which can be added as a process to a simpy.Environment. In the process, a move will be made by the mover, moving it to the destination.

activity_log: the core.Log object in which log_entries about the activities progress will be added. env: the simpy.Environment in which the process will be run mover: moves from its current position to the destination

should inherit from core.Movable

destination: the location the mover will move to should inherit from core.Locatable

engine_order: optional parameter specifying at what percentage of the maximum speed the mover should sail. for example, engine_order=0.5 corresponds to sailing at 50% of max speed

openclsim.model.**sequential_process**(*activity_log*, *env*, *sub_processes*)

Returns a generator which can be added as a process to a simpy.Environment. In the process the given sub_processes will be executed sequentially in the order in which they are given.

activity_log: the core.Log object in which log_entries about the activities progress will be added. env: the simpy.Environment in which the process will be run sub_processes: an Iterable of methods which will be called with the activity_log and env parameters and should

return a generator which could be added as a process to a simpy.Environment the sub_processes will be executed sequentially, in the order in which they are given

openclsim.model.**single_run_process**(*activity_log*, *env*, *origin*, *destination*, *loader*, *mover*, *unloader*, *engine_order=1.0*, *filling=1.0*, *stop_reservation_waiting_event=None*, *verbose=False*)

Returns a generator which can be added as a process to a simpy.Environment. In the process, a single run will be made by the given mover, transporting content from the origin to the destination.

activity_log: the core.Log object in which log_entries about the activities progress will be added. env: the simpy.Environment in which the process will be run origin: object inheriting from HasContainer, HasResource, Locatable, Identifiable and Log destination: object inheriting from HasContainer, HasResource, Locatable, Identifiable and Log loader: object which will get units from 'origin' Container and put them into 'mover' Container

should inherit from Processor, HasResource, Identifiable and Log after the simulation is complete, its log will contain entries for each time it started loading and stopped loading

mover: moves to 'origin' if it is not already there, is loaded, then moves to 'destination' and is unloaded should inherit from Movable, HasContainer, HasResource, Identifiable and Log after the simulation is complete, its log will contain entries for each time it started moving, stopped moving, started loading / unloading and stopped loading / unloading

unloader: gets amount from 'mover' Container and puts it into 'destination' Container should inherit from Processor, HasResource, Identifiable and Log after the simulation is complete, its log will contain entries for each time it started unloading and stopped unloading

engine_order: optional parameter specifying at what percentage of the maximum speed the mover should sail. for example, engine_order=0.5 corresponds to sailing at 50% of max speed

filling: optional parameter specifying at what percentage of the maximum capacity the mover should be loaded.
for example, filling=0.5 corresponds to loading the mover up to 50% of its capacity

stop_reservation_waiting_event: a simpy.Event, if there is no content available in the origin, or no space available
in the destination, instead of performing a single run, the process will wait for new content or space to become available. If a stop_reservation_waiting_event is passed, this event will be combined through a simpy.AnyOf event with the event occurring when new content or space becomes available. This can be used to prevent waiting for new content or space indefinitely when we know it will not become available.

verbose: optional boolean indicating whether additional debug prints should be given.

openclsim.model.**string_to_class**(*text*)

4.3 openclsim.core module

Main module.

class openclsim.core.**ContainerDependentMovable**(*compute_v*, **args*, ***kwargs*)
Bases: *openclsim.core.Movable*, *openclsim.core.HasContainer*

ContainerDependentMovable class

Used for objects that move with a speed dependent on the container level compute_v: a function, given the fraction the container is filled (in [0,1]), returns the current speed

current_speed

determine_amount(*origins*, *destinations*, *loader*, *unloader*, *filling=1*)
Determine the maximum amount that can be carried

determine_schedule(*amount*, *all_amounts*, *origins*, *destinations*)
Define a strategy for passing through the origins and destinations Implemented is FIFO: First origins will start and first destinations will start.

class openclsim.core.**ContainerDependentRouteable**(**args*, ***kwargs*)
Bases: *openclsim.core.ContainerDependentMovable*, *openclsim.core.Routeable*

ContainerDependentRouteable class

Used for objects that move with a speed dependent on the container level compute_v: a function, given the fraction the container is filled (in [0,1]), returns the current speed

current_speed

energy_use(*distance*, *speed*)
Determine the energy use

log_energy_use(*energy*)

class openclsim.core.**DebugArgs**(**args*, ***kwargs*)
Bases: object

Object that logs if leftover args are passed onto it.

class openclsim.core.**DictEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)
Bases: json.encoder.JSONEncoder

serialize a simpy openclsim object to json

default(*o*)
Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

class openclsim.core.**EnergyUse**(*energy_use_sailing=None*, *energy_use_loading=None*, *energy_use_unloading=None*, **args*, ***kwargs*)
Bases: *openclsim.core.SimpyObject*

EnergyUse class

energy_use_sailing: function that specifies the fuel use during sailing activity - input should be time energy_use_loading: function that specifies the fuel use during loading activity - input should be time energy_use_unloading: function that specifies the fuel use during unloading activity - input should be time

Example function could be as follows. The energy use of the loading event is equal to: duration * power_use.

def energy_use_loading(power_use): return lambda x: x * power_use

class openclsim.core.**EventsContainer**(**args*, ***kwargs*)
Bases: `simpy.resources.container.Container`

empty_event

full_event

get(*amount*)
Request to get *amount* of matter from the *container*. The request will be triggered once there is enough matter available in the *container*.

Raise a `ValueError` if `amount <= 0`.

get_available(*amount*)

get_callback(*event*)

get_empty_event(*start_event=False*)

get_full_event(*start_event=False*)

put(*amount*)
Request to put *amount* of matter into the *container*. The request will be triggered once there is enough space in the *container* available.

Raise a `ValueError` if `amount <= 0`.

put_available(*amount*)

put_callback(*event*)

class openclsim.core.**HasContainer**(*capacity*, *level=0*, **args*, ***kwargs*)
Bases: *openclsim.core.SimpyObject*

Container class

capacity: amount the container can hold level: amount the container holds initially container: a simpy object that can hold stuff

class openclsim.core.**HasCosts**(*dayrate=None*, *weekrate=None*, *mobilisation=None*, *demobilisation=None*, **args*, ***kwargs*)

Bases: object

Add cost properties to objects

cost

class openclsim.core.**HasDepthRestriction**(*compute_draught*, *ukc*, *waves=None*, *filling=None*, *min_filling=None*, *max_filling=None*, **args*, ***kwargs*)

Bases: object

HasDepthRestriction class

Used to add depth limits to vessels draught: should be a lambda function with input variable container.volume waves: list with wave_heights ukc: list with ukc, corresponding to wave_heights

filling: filling degree [%] min_filling: minimal filling degree [%] max_filling: max filling degree [%]

calc_depth_restrictions(*location*, *processor*)

calc_required_depth(*draught*, *wave_height*)

check_depth_restriction(*location*, *fill_degree*, *duration*)

check_optimal_filling(*loader*, *unloader*, *origin*, *destination*)

current_draught

viable_time_windows(*fill_degree*, *duration*, *location*)

class openclsim.core.**HasPlume**(*sigma_d=0.015*, *sigma_o=0.1*, *sigma_p=0.05*, *f_sett=0.5*, *f_trap=0.01*, **args*, ***kwargs*)

Bases: *openclsim.core.SimpyObject*

Using values from Becker [2014], https://www.sciencedirect.com/science/article/pii/S0301479714005143.

The values are slightly modified, there is no differences in dragead / bucket drip / cutterhead within this class sigma_d = source term fraction due to dredging sigma_o = source term fraction due to overflow sigma_p = source term fraction due to placement f_sett = fraction of fines that settle within the hopper f_trap = fraction of fines that are trapped within the hopper

class openclsim.core.**HasResource**(*nr_resources=1*, **args*, ***kwargs*)

Bases: *openclsim.core.SimpyObject*

HasProcessingLimit class

Adds a limited Simpy resource which should be requested before the object is used for processing.

class openclsim.core.**HasSoil**(**args*, ***kwargs*)

Bases: object

Add soil properties to an object

soil = list of SoilLayer objects

add_layer(*soillayer*)
Add a layer based on a SoilLayer object.

add_layers(*soillayers*)
Add a list layers based on a SoilLayer object.

get_properties(*amount*)
Get the soil properties for a certain amount

get_soil(*volume*)
Remove soil from self.

put_soil(*soillayer*)
Add soil to self.

Add a layer based on a SoilLayer object.

total_volume()
Determine the total volume of soil.

weighted_average(*layers*, *volumes*)
Create a new SoilLayer object based on the weighted average parameters of extracted layers.

len(layers) should be len(volumes)

class openclsim.core.**HasSpill**(**args*, ***kwargs*)
Bases: *openclsim.core.SimpyObject*

Using relations from Becker [2014], https://www.sciencedirect.com/science/article/pii/S0301479714005143.

spillDredging(*processor*, *mover*, *density*, *fines*, *volume*, *dredging_duration*, *overflow_duration=0*)
Calculate the spill due to the dredging activity

density = the density of the dredged material fines = the percentage of fines in the dredged material volume = the dredged volume dredging_duration = duration of the dredging event overflow_duration = duration of the dredging event whilst overflowing

m_t = total mass of dredged fines per cycle m_d = total mass of spilled fines during one dredging event m_h = total mass of dredged fines that enter the hopper

m_o = total mass of fine material that leaves the hopper during overflow m_op = total mass of fines that are released during overflow that end in dredging plume m_r = total mass of fines that remain within the hopper

spillPlacement(*processor*, *mover*)
Calculate the spill due to the placement activity

class openclsim.core.**HasSpillCondition**(*conditions*, **args*, ***kwargs*)
Bases: *openclsim.core.SimpyObject*

Condition to stop dredging if certain spill limits are exceeded

limit = limit of kilograms spilled material start = start of the condition end = end of the condition

check_conditions(*spill*)

class openclsim.core.**HasWeather**(*dataframe*, *timestep=10*, *bed=None*, *waveheight_column='Hm0 [m]'*, *waveperiod_column='Tp [s]'*, *waterlevel_column='Tide [m]'*, **args*, ***kwargs*)
Bases: object

HasWeather class

Used to add weather conditions to a project site name: name of .csv file in folder

year: name of the year column month: name of the month column day: name of the day column

timestep: size of timestep to interpolate between datapoints (minutes) bed: level of the seabed / riverbed with respect to CD (meters)

class openclsim.core.**HasWorkabilityCriteria**(*criteria*, **args*, ***kwargs*)
Bases: object

HasWorkabilityCriteria class

Used to add workability criteria

calc_work_restrictions(*location*)

check_weather_restriction(*location*, *event_name*)

class openclsim.core.**Identifiable**(*name*, *ID=None*, **args*, ***kwargs*)
Bases: object

Something that has a name and id

name: a name id: a unique id generated with uuid

class openclsim.core.**LoadingFunction**(*loading_rate*, *load_manoeuvring=0*, **args*, ***kwargs*)
Bases: object

Create a loading function and add it a processor. This is a generic and easy to read function, you can create your own LoadingFunction class and add this as a mixin.

loading_rate: the rate at which units are loaded per second load_manoeuvring: the time it takes to manoeuvring in minutes

loading(*origin*, *destination*, *amount*)
Determine the duration based on an amount that is given as input with processing. The origin an destination are also part of the input, because other functions might be dependent on the location.

class openclsim.core.**LoadingSubcycle**(*loading_subcycle*, **args*, ***kwargs*)
Bases: object

loading_subcycle: pandas dataframe with at least the columns EventName (str) and Duration (int or float in minutes)

class openclsim.core.**Locatable**(*geometry*, **args*, ***kwargs*)
Bases: object

Something with a geometry (geojson format)

geometry: can be a point as well as a polygon

is_at(*locatable*, *tolerance=100*)

class openclsim.core.**Log**(**args*, ***kwargs*)
Bases: *openclsim.core.SimpyObject*

Log class

log: log message [format: 'start activity' or 'stop activity'] t: timestamp value: a value can be logged as well geometry: value from locatable (lat, lon)

get_log_as_json()

log_entry(*log*, *t*, *value*, *geometry_log*, *ActivityID*)
Log

class openclsim.core.**Movable**(*v=1*, **args*, ***kwargs*)
Bases: *openclsim.core.SimpyObject*, *openclsim.core.Locatable*

Movable class

Used for object that can move with a fixed speed geometry: point used to track its current location v: speed

current_speed

energy_use(*distance*, *speed*)
Determine the energy use

log_sailing(*event*)
Log the start or stop of the sailing event

move(*destination*, *engine_order=1.0*)
determine distance between origin and destination. Yield the time it takes to travel based on flow properties and load factor of the flow.

sailing_duration(*origin*, *destination*, *engine_order*, *verbose=True*)
Determine the sailing duration

class openclsim.core.**Processor**(**args*, ***kwargs*)
Bases: *openclsim.core.SimpyObject*

Processor class

Adds the loading and unloading components and checks for possible downtime.

If the processor class is used to allow "loading" or "unloading" the mixins "LoadingFunction" and "UnloadingFunction" should be added as well. If no functions are used a subcycle should be used, which is possible with the mixins "LoadingSubcycle" and "UnloadingSubcycle".

addSpill(*origin*, *destination*, *amount*, *duration*)
duration: duration of the activity in seconds origin: origin of the moved volume (the computed amount) destination: destination of the moved volume (the computed amount)

There are three options:

1. Processor is also origin, destination could have spill requirements
2. Processor is also destination, origin could have spill requirements
3. Processor is neither destination, nor origin, but both could have spill requirements

Result of this function is possible waiting, spill is added later on and does not depend on possible requirements

checkSpill(*mover*, *site*, *amount*)
duration: duration of the activity in seconds origin: origin of the moved volume (the computed amount) destination: destination of the moved volume (the computed amount)

There are three options:

1. Processor is also origin, destination could have spill requirements
2. Processor is also destination, origin could have spill requirements
3. Processor is neither destination, nor origin, but both could have spill requirements

Result of this function is possible waiting, spill is added later on and does not depend on possible requirements

checkTide(*mover*, *site*, *desired_level*, *amount*, *duration*)

checkWeather(*processor*, *site*, *event_name*)

check_possible_downtime(*mover*, *site*, *duration*, *desired_level*, *amount*, *event_name*)

check_possible_shift(*origin*, *destination*, *amount*, *activity*)
Check if all the material is available

If the amount is not available in the origin or in the destination yield a put or get. Time will move forward until the amount can be retrieved from the origin or placed into the destination.

computeEnergy(*duration*, *origin*, *destination*)
duration: duration of the activity in seconds origin: origin of the moved volume (the computed amount) destination: destination of the moved volume (the computed amount)

There are three options:

1. Processor is also origin, destination could consume energy
2. Processor is also destination, origin could consume energy
3. Processor is neither destination, nor origin, but both could consume energy

process(*mover*, *desired_level*, *site*)
Moves content from ship to the site or from the site to the ship to ensure that the ship's container reaches the desired level. Yields the time it takes to process.

shiftSoil(*origin*, *destination*, *amount*)
origin: origin of the moved volume (the computed amount) destination: destination of the moved volume (the computed amount) amount: the volume of soil that is moved

Can only occur if both the origin and the destination have soil objects (mix-ins)

class openclsim.core.**ReservationContainer**(**args*, ***kwargs*)
Bases: simpy.resources.container.Container

reserve_get(*amount*)

reserve_get_available

reserve_put(*amount*)

reserve_put_available

class openclsim.core.**Routeable**(**args*, ***kwargs*)
Bases: *openclsim.core.Movable*

Moving folling a certain path

determine_route(*origin*, *destination*)
Determine the fastest sailing route based on distance

determine_speed(*node_from*, *node_to*)
Determine the sailing speed based on edge properties

energy_use(*distance*, *speed*)
Determine the energy use

log_energy_use(*energy*)

sailing_duration(*origin*, *destination*, *engine_order*, *verbose=True*)
Determine the sailing duration based on the properties of the sailing route

class openclsim.core.**SimpyObject**(*env*, **args*, ***kwargs*)
Bases: object

General object which can be extended by any class requiring a simpy environment

env: a simpy Environment

class openclsim.core.**SoilLayer**(*layer*, *volume*, *material*, *density*, *fines*, **args*, ***kwargs*)
Bases: object

Create a soillayer

layer = layer number, 0 to n, with 0 the layer at the surface material = name of the dredged material density = density of the dredged material fines = fraction of total that is fine material

class openclsim.core.**SpillCondition**(*spill_limit*, *start*, *end*, **args*, ***kwargs*)
Bases: object

Condition to stop dredging if certain spill limits are exceeded

limit = limit of kilograms spilled material start = start of the condition end = end of the condition

class openclsim.core.**UnloadingFunction**(*unloading_rate*, *unload_manoeuvring=0*, **args*, ***kwargs*)
Bases: object

Create an unloading function and add it a processor. This is a generic and easy to read function, you can create your own LoadingFunction class and add this as a mixin.

unloading_rate: the rate at which units are loaded per second unload_manoeuvring: the time it takes to manoeuvring in minutes

unloading(*origin*, *destination*, *amount*)
Determine the duration based on an amount that is given as input with processing. The origin an destination are also part of the input, because other functions might be dependent on the location.

class openclsim.core.**UnloadingSubcycle**(*unloading_subcycle*, **args*, ***kwargs*)
Bases: object

unloading_subcycle: pandas dataframe with at least the columns EventName (str) and Duration (int or float in minutes)

class openclsim.core.**WorkabilityCriterion**(*event_name*, *condition*, *minimum=-inf*, *maximum=inf*, *window_length=datetime.timedelta(seconds=3600)*, **args*, ***kwargs*)
Bases: object

WorkabilityCriterion class

Used to add limits to vessels (and therefore acitivities) event_name: name of the event for which this criterion applies condition: column name of the metocean data (Hs, Tp, etc.) minimum: minimum value maximum: maximum value window_length: minimal length of the window (minutes)

openclsim.core.**serialize**(*obj*)

4.4 openclsim.server module

openclsim.server.**csv**()

openclsim.server.**demo_plot**()
demo plot

openclsim.server.**energy_plot**()
return a plot with the cumulative energy use

openclsim.server.**energy_use_plot_from_json**(*jsonFile*)
Create a Gantt chart, based on a json input file

openclsim.server.**equipment_plot**()
return a planning

openclsim.server.**equipment_plot_from_json**(*jsonFile*)
Create a Gantt chart, based on a json input file

openclsim.server.**interrupt_processes**(*event*, *activities*, *env*)

openclsim.server.**main**()

openclsim.server.**save_simulation**(*config*, *simulation*, *tmp_path=''*)
Save the given simulation. The config is used to produce an md5 hash of its text representation. This hash is used as a prefix for the files which are written. This ensures that simulations with the same config are written to the same files (although it is not a completely foolproof method, for example changing an equipment or location name, does not alter the simulation result, but does alter the config file). The optional tmp_path parameter should only be used for unit tests.

openclsim.server.**simulate**()
run a simulation

openclsim.server.**simulate_from_json**(*config*, *tmp_path='static'*)
Create a simulation and run it, based on a json input file. The optional tmp_path parameter should only be used for unit tests.

openclsim.server.**update_end_time**(*event*, *env*)

4.5 Module contents

Top-level package for OpenCLSim.

CHAPTER 5

OpenCLSim API

A flask server is part of the OpenCLSim package. This allows using the python code from OpenCLSim from a separate front-end.

5.1 Starting the Flask Server

The example code below lets you start the Flask server from the windows command line, for other operation systems please check the Flask Documentation.

```
# Set Flask app
set FLASK_APP=openclsim/server.py

# Set Flask environment
set FLASK_ENV=development

# Run Flask
flask run
```

5.2 Using the Flask Server

You can send json strings to the Flask Server using the methods presented in the server module.

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/TUDelft-CITG/OpenCLSim/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

OpenCLSim could always use more documentation, whether as part of the official OpenCLSim docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/TUDelft-CITG/OpenCLSim/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *OpenCLSim* for local development.

1. Fork the *OpenCLSim* repository on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/OpenCLSim.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv openclsim
$ cd openclsim/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 openclsim tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. The style of OpenCLSim is according to Black. Format your code using Black witht the following lines of code:

```
$ black openclsim
$ black tests
```

   You can install black using pip.

7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.4, 3.5 and 3.6, and for PyPy. Check CircleCI and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ py.test tests.test_openclsim
```

To make the documentation pages:

```
$ make docs # for linux/osx
```

For windows:

```
$ del docs\openclsim.rst
$ del docs\modules.rst
$ sphinx-apidoc -o docs/ openclsim
$ cd docs
$ make html
$ start explorer _build\html\index.html
```

6.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 7

Credits

7.1 Development Lead

- Mark van Koningsveld
- Joris den Uijl
- Fedor Baart
- Anne Hommelberg

7.2 Contributors

Various MSc projects

- Joris den Uijl, 2018. **Integrating engineering knowledge in logistical optimisation: development of a concept evaluation tool.** MSc thesis. Delft University of Technology, Civil Engineering and Geosciences, Hydraulic Engineering. Delft, the Netherlands.
- Vibeke van der Bilt, 2019. **Assessing emission performance of dredging projects.** MSc thesis. Delft University of Technology, Civil Engineering and Geosciences, Hydraulic Engineering - Ports and Waterways. Delft, the Netherlands.
- Pieter van Halem, 2019. **Route optimization in dynamic currents. Navigation system for the North Sea and Wadden Sea.** MSc thesis. Delft University of Technology, Civil Engineering and Geosciences, Environmental Fluid Mechanics. Delft, the Netherlands.
- Servaas Kievits, 2019. **A framework for the impact assessment of low discharges on the performance of inland waterway transport.** MSc thesis. Delft University of Technology, Civil Engineering and Geosciences, Hydraulic Engineering - Ports and Waterways. Delft, the Netherlands.

Ongoing PhD work

- Frederik Vinke, 2019. **Climate proofing the inland water transport system in the Netherlands.** PhD thesis. Delft University of Technology, Civil Engineering and Geosciences, Hydraulic Engineering - Ports and Waterways. Delft, the Netherlands.

CHAPTER 8

History

8.1 1.2.3 (2020-05-07)

Improved documentation and readme.

8.2 1.2.2 (2020-04-10)

Fixed a bug raised in GitHub issue #89.

8.3 1.2.1 (2020-03-27)

Minor bug fixes.

8.4 1.2.0 (2020-01-27)

- Major updates to the Movable class
- You can now enter multiple origins and destinations in one activity
- Optimisation of the schedule is possible by enhanding the Movable

8.5 1.1.1 (2019-12-11)

- Minor bug fixes

8.6 1.1.0 (2019-08-30)

- More generic Movable class
- More generic Routeable class
- Easier to implement own functions and adjustments

8.7 1.0.1 (2019-07-26)

- Small bug fixes

8.8 1.0.0 (2019-07-10)

- First formal release

8.9 0.3.0 (2019-06-20)

- First release to PyPI and rename to OpenCLSim

8.10 v0.2.0 (2019-02-14)

- Second tag on GitHub

8.11 v0.1.0 (2018-08-01)

- First tag on GitHub

CHAPTER 9

Version conventions

This package is being developed continuously. Branch protection is turned on for the master branch. Useful new features and bugfixes can be developed in a separate branch or fork. Pull requests can be made to integrate updates into the master branch. To keep track of versions, every change to the master branch will receive a version tag. This page outlines the version tags' naming convention.

Each change to the master branch is stamped with a unique version identifier. We use sequence based version identifiers, that consist of a sequence of three numbers: the first number is a major change identifier, followed by a minor change idenfier and finally a maintenance identifier. This leads to version identifiers of the form:

major.minor.maintenance (example: 1.2.2)

The following guideline gives an idea what types of changes are considered major changes, minor changes and maintenance:

- Major changes (typically breaking changes) -> major + 1
- Minor changes (typically adding of new features) -> minor + 1
- Maintenance (typically bug fixes and updates in documentation -> maintenance + 1

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

F

G

H

I

L

M

O

P

R

S

T

U

V

W