ws3 Documentation

Release 0.1a1

Gregory Paradis

Jan 04, 2018

CONTENTS:

CHAPTER ONE

OVERVIEW

This chapter describes the `ws3` Python software package. `ws3` (short for *Wood Supply Simulation System*) is an open-source software package that is designed to model *wood supply planning problems* (WSPP), in the context of sustainable forest management.

1.1 Background

The WSPP basically consists of determining the location, nature, and timing of forest managment activities (i.e., *actions*) for a given forest, typically over multiple planning periods (often spanning a planning horizon of 100 years or more). This is a very complex problem, so in practice the WSPP process is typically supported by complex software models that simulate an alternating sequence of *actions* and *growth* for each time step, starting from an initial forest inventory.

Wood supply models require complex input datasets. WSM input data can be divided into *static* and *dynamic* components.

Static WSM input data types include initial forest inventory, growth and yield curves, action definitions, transition definitions, and a schedule of prescribed activities to simulate. Dynamic WSM input data may include a combination of heuristic and optimization-based processes to automatically derive a dynamic activity schedule (which gets layered on top of the static activity schedule).

The forest inventory data is typically aggregated into a manageable number of *strata* (i.e., *development types*), which simplifies the modelling. Each development type is linked to *growth and yield* functions describing the change in key attributes attributes (e.g., species-wise standing timber volume, number of merchantable stems per unit area, wildlife habitat suitability index value, etc.) expressed as a function of statum age. Each development type may also be associated with one or more *actions*, which can yield *output products* (e.g., species-wise assortments of raw timber products, cost, treated area, etc.). Applying an action to a development type induces a *state transition* (i.e., applying an action may modify one or more stratification variables, effectively transitioning the treated area to a different development type).

Given a set of static inputs, a given WSM can be used to simulate a number of *scenarios*. Generally, scenarios differ only in terms of the dynamic activity schedule that is simulated. Comparing output from several scenarios is the basic mechanism by which forest managers derive insight from wood supply models.

There are two basic approached that can be used (independently, or in combination) to generate the dynamic activity schedules for each scenario.

The simplest approach, which we call the *heuristic* activity scheduling method, involves defining period-wise targets for a single key output (e.g., total harvest volume) along with a set of rules that determines the order in which actions are applied to eligible development types. At each time step, the model iteratively applies actions according to the rules until the output target value is met, or it runs out of eligible area. At this point, the model simulates one time-step worth of growth, and the process repeats until the end of the planning horizon.

A slightly more complex approach, which we call the *optimization* activity scheduling method, involves defining an optimization problem (i.e., an objective function and constraints), and solving this problem to optimality (using one of several available third-party mathematical solver software packages).

Although the optimization approach is more powerful than the heuristic approach for modelling harvesting and other anthopic activites, an optimization approach is not appropriate for modelling strongly-stochastic disturbance processes (e.g., wildfire, insect invasions, blowdown). Thus, a hybrid heuristic-optimization approach may be best when modelling a combination of anthopic and natural disturbance processes.

1.2 Package Design and Implementation

The `ws3` package is implemented using the Python programming language. `ws3` is basically an aspatial wood supply model, which applies actions to development types, simulates growth, and tracks inventory area at each time step. Aspatial models output aspatial activity schedules—each line of the output schedule specifies the stratification variable values (which constitute a unique key into the list of development types), the time step, the action code, and the area treated.

Because the model is aspatial, the area treated on a given line of the output schedule may not be spatially contiguous (i.e., the area may be geographically dispersed throughout the landscape). Furthermore, in the common case where only a subset of development type area is treated in a given time step, the aspatial model provides not information regarding which subset of available area is treated (and, conversely, not treated). Some applications (e.g., linking to spatially-explicit or highly–spatially-referenced models) require a spatially-explicit activity schedule. `ws3` includes a *spatial disturbance allocator* sub-module, which contains functions that can map aspatial multi-period action schedules onto a rasterized spatial representation of the forest.

`ws3` uses a scripted Python interface to control the model, which provides maximum flexibility and makes it very easy to automate modelling workflows. This ensures reproducible methodologies, and makes it relatively easy to link `ws3` models to other software packages to form complex modelling pipelines. The scripted interface also makes it relatively easy to implement custom data-importing functions, which makes it easier to import existing data from a variety of ad-hoc sources without the need to recompile the data into a standard `ws3`-specific format (i.e., importing functions can be implemented such that the conversion process is fully automated and applied to raw input data *on the fly*). Similarly, users can easily implment custom functions to re-format `ws3` output data *on the fly* (either for static serialization to disk, or to be piped live into another process).

Although we recommend using Jupyter Notebooks as an interactive interface to `ws3` (the package was specifically designed with an interactive notebook interface in mind), `ws3` functions can also be imported and run in fully scripted workflow (e.g., non-interactive batch processes can be run in a massively-parallelled workflow on high-performance–computing resources, if available). The ability to mix interactive and massively-paralleled non-interactive workflows is a unique feature of `ws3`.

`ws3` is a complex and flexible collection of functional software units. The following sections describe some of the main classes and functions in the package, and describe some common use cases, and link to sample notebooks that implement these use cases.

1.3 Overview of Main Classes and Functions

This section describes some of the main classes and functions that make up.

The `ForestModel` class is the core class in the package. This class encapsulates all the information used to simulate scenarios from a given dataset (i.e., stratified intial inventory, growth and yield functions, action eligibility, transition matrix, action schedule, etc.), as well as a large collection of functions to import and export data, generate activity schedules, and simulate application of these schedules (i.e., run scenarios).

At the heart of the `ForestModel` class is a list of `DevelopentType` instances. Each `DevelopmentType` instance encapsulates information about one development type (i.e., a forest stratum, which is an aggregate of smaller *stands* that make up the raw forest inventory input data). The `DevelopmentType` class also stores a list of operable *actions*, maps *state variable transitions* to these actions, stores growth and yield functions, and knows how to *grow itself* when time is incremented during a simulation.

1.4 Common Use Case and Sample Notebooks

In this section, we assume an interactive Jupyter Notebook environment is used to interface with `ws3`.

A typical use case starts with creating an instance of the `ForestModel` class. Then, we need to load data into this instance, define one or more scenarios (using a mix of heuristic and optimization approaches), run the scenarios, and export output data to a format suitable for analysis (or link to the next model in a larger modelling pipeline).

The first step in typical workflow is to run a mix of standard `ws3` and custom data-importing functions. These functions import data from various sources, *on-the-fly* reformat this data to be compatible with `ws3`, and load the reformated data into the `ForestModel` instance using standard methods. For example, `ws3` includes functions to import legacy Woodstock[1] model data (including LANDSCAPE, CONSTANTS, AREAS, YIELDS, LIFESPAN, ACTIONS, TRANSITIONS, and SCHEDULE section data), as well as functions to import and rasterize vector stand inventory data.

For example, one might define the following custom Python function in a Jupyter Notebook, to import data formatted for Woodstock.:

```
def instantiate_forestmodel(model_name, model_path, horizon,
                            period_length, max_age, add_null_action=True):
    fm = ForestModel(model_name=model_name,
                     model_path=model_path,
                     horizon=horizon,
                     period_length=period_length,
                     max_age=max_age)
    fm.import_landscape_section()
    fm.import_areas_section()
    fm.import_yields_section()
    fm.import_actions_section()
    fm.add_null_action()
    fm.import_transitions_section()
    fm.reset_actions()
    fm.initialize_areas()
    fm.grow()
    return fm
```

The next step in a typical workflow is to define one or more scenarios. Assuming that we are using an optimization approach to harvest scheduling, we need to define an objective function (e.g., maximize total harvest volume) and constraints (e.g., species-wise volume and area even-flow constraints, ending standing inventory constraints, periodic minimum late-seral-stage area constraints)[2], build the optimization model matrix, solve the model to optimality[3].

[1] Woodstock software is part of Remsoft Solution Suite.

[2] `ws3` currently implements functions to formulate and solve *Model I* wood supply optimization problems—however, the package was deliberately designed to make it easy to transparently switch between *Model I*, *Model II* and *Model III* formulations without affecting the rest of the modelling workflow. `ws3` currently has placeholder function stubs for *Model II* and *Model III* formulations, which will be implemented in later versions as the need arises. For more information on wood supply model formulations, see Chapter 16 of the Handbook of Operations Research in Natural Resources.

[3] `ws3` currently uses the Gurobi solver to solve the linear programming (LP) problems to optimality. We chose Gurobi because it is one of the top two solvers currently available (along with the CPLEX solver), has a simple and flexible policy for requesting unlimited licences for free use in research projects, has elegant Python bindings, and we like the technical documentation. However, we deliberately used a modular design, which allows us to transparently switch to a different solver in `ws3` without affecting the rest of the workflow—this design will make it easy to implement an interface to addional solvers in future releases.

CHAPTER

TWO

WS3 PACKAGE MODULES

2.1 common module

This module contains definitions for global attributes, functions, and classes that might be used anywhere in the package.

Attributes: HORIZON_DEFAULT (int): Default value for ‘’. PERIOD_LENGTH_DEFAULT (int): Default number of years per period. MIN_AGE_DEFAULT (int): Default value for *core.Curve.xmin*. MAX_AGE_DEFAULT (int): Default value for *core.Curve.xmax*. CURVE_EPSILON_DEFAULT (float): Defalut value for *core.Curve.epsilon*.

AREA_EPSILON_DEFAULT = 0.01

class common.**Node**(*nid*, *data=None*, *parent=None*)

Bases: `object`

add_child(*child*)

children()

data(*key=None*)

is_leaf()

is_root()

parent()

class common.**Tree**(*period=1*)

Bases: `object`

add_node(*data*, *parent=None*)

children(*nid*)

grow(*data*)

leaves()

node(*nid*)

nodes()

path(*leaf=None*)

paths()

root()

ungrow()

common.**clean_vector_data**(*src_path*, *dst_path*, *dst_name*, *prop_names*, *clean=True*, *tolerance=10.0*, *preserve_topology=True*, *logfn='clean_stand_shapefile.log'*, *max_records=None*, *theme0=None*, *prop_types=None*, *driver='ESRI Shapefile'*, *dst_epsg=None*)

common.**harv_cost**(*piece_size*, *is_finalcut*, *is_toleranthw*, *partialcut_extracare=False*, *A=1.97*, *B=0.405*, *C=0.169*, *D=0.164*, *E=0.202*, *F=13.6*, *G=8.83*, *K=0.0*, *rv=False*)

Returns harvest cost, given piece size, treatment type (final cut or not), stand type (tolerant hardwood or not), partialcut "extra care" flag, and a series of regression coefficients (A, B, C, D, E, F, G, K, all with defaults [extracted from MERIS technical documentation; also see Sebastien Lacroix, BMMB]). Assumes that variables are deterministic.

common.**harv_cost_rv**(*tv_mu*, *tv_sigma*, *N_mu*, *N_sigma*, *psr*, *is_finalcut*, *is_toleranthw*, *partialcut_extracare=False*, *tv_min=50.0*, *N_min=200.0*, *ps_min=0.05*, *E_fromintegral=False*, *e=0.01*, *n=1000*)

Returns harvest cost, given piece size, treatment type (final cut or not), stand type (tolerant hardwood or not), partialcut "extra care" flag, and a series of regression coefficients (A, B, C, D, E, F, G, K, all with defaults [extracted from MERIS technical documentation; also see Sebastien Lacroix, BMMB]). Assumes that variables are random variates (returns expected value of function, using PaCAL packages to model random variates, assuming normal distribution for all three variables). Can use either PaCAL numerical integration (sssslow!), or custom numerical integration using Monte Carlo sampling (default).

common.**harv_cost_wec**(*piece_size*, *is_finalcut*, *is_toleranthw*, *sigma*, *nsigmas=3*, ***kwargs*)

Estimate harvest cost with error correction. :float piece_size: mean piece size :bool is_finalcut: True if harvest treatment is final cut, False otherwise :bool is_toleranthw: True if tolerant hardwood cover type, False otherwise :float sigma: standard deviation of piece size estimator :int nsigmas: number of standard deviations to model on either side of the mean (default 3) :float binw: width of bins for weighted numerical integration, in multiples of sigma (default 1.0)

common.**hash_dt**(*dt*, *dtype='uint32'*, *nbytes=4*)

common.**is_num**(*s*)

Returns True if s is a number.

common.**piece_size_ratio**(*treatment_type*, *cover_type*, *piece_size_ratios*)

Returns piece size ratio. Assume Action.is_harvest in [0, 1, 2, 3] Assume cover_type in ['r', 'm', 'f'] Return vr/vp ratio, where

> vr is mean piece size of harvested stems, and vp is mean piece size of stand before harvesting.

common.**rasterize_stands**(*shp_path*, *tif_path*, *theme_cols*, *age_col*, *age_divisor=1.0*, *d=100.0*, *dtype='uint32'*, *compress='lzw'*, *round_coords=True*, *value_func=<function <lambda>>*, *verbose=False*)

common.**reproject**(*f*, *srs_crs*, *dst_crs*)

common.**reproject_vector_data**(*src_path*, *snk_path*, *snk_epsg*, *driver='ESRI Shapefile'*)

common.**sylv_cred**(*P*, *vr*, *vp*, *formula*)

Returns sylviculture credit ($ per hectare), given P (volume harvested per hectare), vr (mean piece size of harvested stems), vp (mean piece size of stand before harvesting), and formula index (1 to 7). Assumes that variables (P, vr, vp) are deterministic.

common.**sylv_cred_formula**(*treatment_type*, *cover_type*)

Returns sylviculture credit formula index, given treatment type and cover type.

common.**sylv_cred_rv**(*P_mu*, *P_sigma*, *tv_mu*, *tv_sigma*, *N_mu*, *N_sigma*, *psr*, *treatment_type=None*, *cover_type=None*, *formula=None*, *P_min=20.0*, *tv_min=50.0*, *N_min=200.0*, *ps_min=0.05*, *E_fromintegral=False*, *e=0.01*, *n=1000*)

Returns sylviculture credit ($ per hectare), given P (volume harvested per hectare), vr (mean piece size of harvested stems), vp (mean piece size of stand before harvesting), and formula index (1 to 7). Assumes that variables (P, vr, vp) are random variates (returns expected value of function, using PaCAL packages to model random variates, assuming normal distribution for all three variables). Can use either PaCAL numerical integration (sssslow!), or custom numerical integration using Monte Carlo sampling (default).

`common.`**`timed`**(*func*)

`common.`**`warp_raster`**(*src*, *dst_path*, *dst_crs={'init': 'EPSG:4326'}*)

2.2 core module

class `core.`**`Curve`**(*label=None*, *id=None*, *is_volume=False*, *points=None*, *type='a'*, *is_special=False*, *period_length=10*, *xmin=0*, *xmax=1000*, *epsilon=0.01*, *simplify=True*)

Bases: `object`

Describes change in state over time (between treatments)

`add_points`(*points*, *simplify=True*, *compile_y=False*)

`cai`()

`lookup`(*y*, *from_right=False*, *roundx=False*)

`mai`()

`points`()

`range`(*lo=None*, *hi=None*, *as_bounds=False*, *left_range=True*)

left_range True: ub lookup from left (default) left_range False: ub lookup from right (widest possible range)

`simplify`(*points=None*, *autotune=True*, *compile_y=False*, *verbose=False*)

`y`(*compile_y=False*)

`ytp`()

class `core.`**`Interpolator`**(*points*)

Bases: `object`

Interpolates x and y values from sparse curve point list.

`lookup`(*y*, *from_right=False*)

`points`()

2.3 forest module

This module implements functions for building and running wood supply simulation models.

The `ForestModel` and `DevelopmentType` classes constitute the core functional units of this module, and of the `ws3` package in general.

class `forest.`**`Action`**(*code*, *targetage=None*, *descr="*, *lockexempt=False*, *components=None*, *partial=None*, *is_harvest=0*)

Bases: `object`

Encapsulates data for an action.

class `forest.`**`DevelopmentType`**(*key*, *parent*)

Bases: `object`

Encapsulates Forest development type data (curves, age, area), and provides methods to operate on the data.

The key is basically the fully expanded mask (expressed as a tuple of values). The parent is a reference to the ForestModel object in which self is embedded.

add_ycomp(*ytype*, *yname*, *ycomp*, *first_match=True*)

area(*period*, *age=None*, *area=None*, *delta=True*)
If area not specified, returns area inventory for period (optionally age), else sets area for period and age. If delta switch active (default True), area value is interpreted as an increment on current inventory.

compile_action(*acode*, *verbose=False*)
Compile action, given action code. This mostly involves resolving operability expression strings into lower and upper operability limits, defined as (alo, ahi) age pair for each period. Deletes action from self if not operable in any period.

compile_actions(*verbose=False*)
Compile all actions.

grow(*start_period=1*, *cascade=True*)
Grow self (default starting period 1, and cascading to end of planning horizon).

initialize_areas()
Copy initial inventory to period-1 inventory.

is_operable(*acode*, *period*, *age=None*, *verbose=False*)
Test hypothetical operability. Does not imply that there is any operable area in current inventory.

operable_ages(*acode*, *period*)
Finds list of ages at which self is operable, given an action code and period index.

operable_area(*acode*, *period*, *age=None*, *cleanup=True*)
Returns 0 if inoperable or no current inventory, operable area given action code and period (and optionally age) index otherwise. If cleanup switch activated (default True) and age specified, deletes the ageclass from the inventory dict if operable area is less than self.parent.area_epsilon.

reset_areas(*period=None*)
Reset areas dict.

resolve_condition(*yname*, *lo*, *hi*)
Find lower and upper ages that correspond to lo and hi values of yname (interpreted as first occurence of yield value, reading curve from left and right, respectively).

ycomp(*yname*, *silent_fail=True*)

ycomps()
Returns list of yield component keys.

class forest.**ForestModel**(*model_name*, *model_path*, *horizon=30*, *period_length=10*, *max_age=1000*, *area_epsilon=0.01*, *curve_epsilon=0.01*)
Bases: `object`

This is the core class of the ws3 package. Includes methods import data from various sources, simulate growth and apply actions. The model can be used in either a (prescriptive) simulation-based approach or a (descriptive) optimization-based approach.

This class encapsulates all the information used to simulate scenarios from a given dataset (i.e., stratified intial inventory, growth and yield functions, action eligibility, transition matrix, action schedule, etc.), as well as a large collection of functions to import and export data, generate activity schedules, and simulate application of these schedules (i.e., run scenarios).

At the heart of the `ForestModel` class is a list of `DevelopentType` instances. Each `DevelopmentType` instance encapsulates information about one development type (i.e., a forest stratum, which is an aggregate of smaller *stands* that make up the raw forest inventory input data). The `DevelopmentType` class also stores a list of operable *actions*, maps *state variable transitions* to these actions, stores growth and yield functions, and knows how to *grow itself* when time is incremented during a simulation.

A typical use case starts with creating an instance of the `ForestModel` class. Then, we need to load data into this instance, define one or more scenarios (using a mix of heuristic and optimization approaches), run the scenarios, and export output data to a format suitable for analysis (or link to the next model in a larger modelling pipeline).

`add_null_action`(*acode='null'*, *minage=None*, *maxage=None*)

`add_problem`(*name*, *coeff_funcs*, *cflw_e*, *cgen_data=None*, *solver='gurobi'*, *formulation=1*, *z_coeff_key='z'*, *acodes=None*)

`add_theme`(*name*, *basecodes=[]*, *aggs={}*)

`age_class_distribution`(*period*, *mask=None*)
Returns age class distribution (dict of areas, keys on age).

`apply_action`(*dtype_key*, *acode*, *period*, *age*, *area*, *override_operability=False*, *fuzzy_age=True*, *recourse_enabled=True*, *areaselector=None*, *compile_t_ycomps=False*, *compile_c_ycomps=False*, *verbose=False*)
Applies action, given action code, development type, period, age, area. Can optionally override operability limits, optionally use fuzzy age (i.e., attempt to apply action to proximal age class if specified age is not operable), optionally use default AreaSelector to patch missing area (if recourse enabled). Applying an action is a rather complex process, involving testing for operability (JIT-compiling operability expression as required), checking that valid transitions are defined, checking that area is available (possibly using fuzzy age and area selector functions to find missing area), generate list of target development types (from source development type and transition expressions [which may need to be JIT-compiled]), creating new development types (as needed), doing the area accounting correctly (without creating or destroying any area), and compiling the products from the action (which gets a bit complicated in the case of partial cuts…).

Returns (errorcode, missing_area, target_dt) triplet, where errorcode is an error code, missing_area is the missing area, and target_dt is a list of (dtk, tprop, targetage) triplets (one triplet per target development type).

`apply_schedule`(*schedule*, *max_period=None*, *verbose=False*, *fail_on_missingarea=False*, *force_integral_area=False*, *override_operability=False*, *fuzzy_age=True*, *recourse_enabled=True*, *areaselector=None*, *compile_t_ycomps=False*, *compile_c_ycomps=False*)
Assumes schedule in format returned by import_schedule_section(). That is: list of (dtype_key, age, area, acode, period, etype) tuples. Also assumes that actions in list are sorted by applied period.

`commit_actions`(*period=1*, *repair_future_actions=False*, *verbose=False*)
Commits applied actions (i.e., apply transitions and grow, default starting at period 1). By default, will attempt to repair broken (infeasible) future actions, attempting to replace infeasiblea operated area using default AreaSelector.

`compile_product`(*period*, *expr*, *acode=None*, *dtype_keys=None*, *age=None*, *coeff=False*, *verbose=False*)
Compiles products from applied actions in given period. Parses string expression, which resolves to a single coefficient. Operated area can be filtered on action code, development type key list, and age. Result is product of sum of filtered area and coefficient.

`compile_schedule`(*problem*, *formulation=1*, *skip_null='null'*)
Compiles a `ws3`-compatible schedule data object from a solved `ws3.opt.Problem` instance. This is just a dispatcher function—the actual compilation is done by a formulation-specific function (assumes *Model I* formulation if not specified).

`create_dtype_fromkey`(*key*)
Creates a new development type, given a key (checks for existing, auto-assigns yield compompontents,

auto-assign actions and transitions, checks for operability (filed under inoperable if applicable).

dt (*dtype_key*)
Returns development type, given key (returns None on invalid key).

grow (*start_period=1*, *cascade=True*)
Simulates growth (default startint at period 1 and cascading to the end of the planning horizon).

import_actions_section (*filename_suffix='act'*, *mask_func=None*, *nthemes=None*)
Imports ACTIONS section from a Forest model.

import_areas_section (*model_path=None*, *model_name=None*, *filename_suffix='are'*, *import_empty=False*)
Imports AREAS section from a Forest model.

import_constants_section (*filename_suffix='con'*)
Imports CONSTANTS section from a Forest model.

import_control_section (*filename_suffix='run'*)
Imports CONTROL section from a Forest model. .. warning:: Not implemented yet.

import_graphics_section (*filename_suffix='gra'*)
Imports GRAPHICS section from a Forest model. .. warning:: Not implemented yet.

import_landscape_section (*filename_suffix='lan'*, *ti_offset=0*)
Imports LANDSCAPE section from a Forest model.

import_lifespan_section (*filename_suffix='lif'*)
Imports LIFESPAN section from a Forest model. .. warning:: Not implemented yet.

import_optimize_section (*filename_suffix='opt'*)
Imports OPTIMIZE section from a Forest model. .. warning:: Not implemented yet.

import_outputs_section (*filename_suffix='out'*)
Imports OUTPUTS section from a Forest model.

import_schedule_section (*filename_suffix='seq'*, *replace_commas=True*, *filename_prefix=None*)
Imports SCHEDULE section from a Forest model.

import_transitions_section (*filename_suffix='trn'*, *mask_func=None*, *nthemes=None*)
Imports TRANSITIONS section from a Forest model.

import_yields_section (*filename_suffix='yld'*, *mask_func=None*, *verbose=False*)
Imports YIELDS section from a Forest model.

initialize_areas ()
Copies areas from period 0 to period 1.

inventory (*period*, *yname=None*, *age=None*, *mask=None*, *dtype_keys=None*)
Flexible method that compiles inventory at given period. Unit of return data defaults to area if yname not given, but takes on unit of yield component otherwise. Can be constrained by age and development type mask.

is_harvest (*acode*)
Returns True if acode corresponds to a harvesting action.

match_mask (*mask*, *key*)
Returns True if key matches mask.

operable_area (*acode*, *period*, *age=None*)
Returns total operable area, given action code and period (and optionally age).

operable_dtypes (*acode*, *period*, *mask=None*)
Returns dict (keyed on development type key, values are lists of operable ages).

operated_area (*acode*, *period*, *dtype_key=None*, *age=None*)
Compiles operated area, given action code and period (and optionally list of development type keys or age).

piece_size (*dtype_key*, *age*)
Returns piece size, given development type key and age.

register_curve (*curve*)
Add curve to global curve hash map (uses result of Curve.points() to construct hash key).

repair_actions (*period*, *areaselector=None*)
Attempts to repair the action schedule for given period, using an AreaSelector object (defaults to class-default areaselector, which is a simple greedy oldest-first selector).

reset_actions (*period=None*, *acode=None*)
Resets actions (default resets all periods, all actions, unless period or acode specified).

resolve_append (*dtk*, *expr*)

resolve_condition (*condition*, *dtype_key=None*)
Evaluate @AGE or @YLD condition. Returns list of ages.

resolve_replace (*dtk*, *expr*)

resolve_tappend (*dt*, *tappend*)

resolve_targetage (*dtk*, *tyield*, *sage*, *tage*, *acode*, *verbose=False*)

resolve_tmask (*dt*, *tmask*, *treplace*, *tappend*)
Returns new developement type key (tuple of values, one per theme), given developement type and (treplace, tappend) expressions.

resolve_treplace (*dt*, *treplace*)

sylv_cred_formula (*treatment_type*, *cover_type*)

theme_basecodes (*theme_index*)
Return list of base codes, given theme index.

tree ()

unmask (*mask*)
Iteratively filter list of development type keys using mask values. Accepts Forest-style string masks to facilitate cut-and-paste testing.

class forest.**GreedyAreaSelector** (*parent*)
Bases: `object`

Default AreaSelector implementation. Selects areas for treatment from oldest age classes.

operate (*period*, *acode*, *target_area*, *mask=None*, *commit_actions=True*, *verbose=False*)
Greedily operate on oldest operable age classes. Returns missing area (i.e., difference between target and operated areas).

class forest.**Output** (*parent*, *code=None*, *expression=None*, *factor=(1.0, 1)*, *description=''*, *theme_index=-1*, *is_basic=False*, *is_level=False*)
Bases: `object`

Encapsulates data and methods to operate on aggregate outputs from the model. Emulates behaviour of Forest outputs. .. warning:: Behaviour of Forest outputs is quite complex. This class needs more work before it is used in a production setting (i.e., resolution of some complex output cases is buggy).

2.4 opt module

This module implements functions for formulating and solving optimization problems. The notation is very generic (i.e., refers to variables, constraints, problems, solutions, etc.). All the wood-supply-problem–specific references are implemented in the `forest` module.

The `Problem` class is the main functional unit here. It encapsulates optimization problem data (i.e., variables, constraints, objective function, and optimal solution), as well as methods to operate on this data (i.e., methods to build and solve the problem, and report on the optimal solution).

Note that we implemented a modular design that decouples the implementation from the choice of solver. Currently, only bindings to the Gurobi solver are implemented, although bindings to other solvers can easilty be added (we will add more binding in later releases, as the need arises).

class `opt.`**`Constraint`**(*name*, *coeffs*, *sense*, *rhs*)

Bases: `object`

Encapsulates data describing a constraint in an optimization problem. This includes a constraint name (should be unique within a problem, although the user is responsible for enforcing this condition), a vector of coefficient values (length of vector should match the number of variables in the problem, although the user is responsible for enforcing this condition), a sense (should be one of `SENSE_EQ`, `SENSE_GEQ`, or `SENSE_LEQ`), and a right-hand-side value.

class `opt.`**`Problem`**(*name*, *sense=-1*, *solver='gurobi'*)

Bases: `object`

This is the main class of the `opt` module—it encapsulates optimization problem data (i.e., variables, constraints, objective function, optimal solution, and choice of solver), as well as methods to operate on this data (i.e., methods to build and solve the problem, and report on the optimal solution).

`add_constraint`(*name*, *coeffs*, *sense*, *rhs*, *validate=False*)

Adds a constraint to the problem. The constraint name should be unique within the problem (user is responsible for enforcing this condition). Constraint coeffients should be provided as a `dict`, keyed on variable names—length of constraint coefficient `dict` should match number of variables in the problem (user is responsible for enforcing this condition). Constraint sense should be one of `SENSE_EQ`, `SENSE_GEQ`, or `SENSE_LEQ`.

Note that calling this method resets the value of the optimal solution to `None`.

`add_var`(*name*, *vtype*, *lb=0.0*, *ub=inf*)

Adds a variable to the problem. The variable name should be unique within the problem (user is responsible for enforcing this condition). Variable type should be one of `VTYPE_CONTINUOUS`, `VTYPE_INTEGER`, or `VTYPE_BINARY`. Variable value bounds default to zero for the lower bound and positive infinity for the upper bound.

Note that calling this method resets the value of the optimal solution to `None`.

`constraint_names`()

Returns a list of constraint names.

`name`()

Returns problem name.

`sense`(*val=None*)

Returns (or sets) objective function sense. Value should be one of `SENSE_MINIMIZE` or `SENSE_MAXIMIZE`.

`solution`()

Returns a `dict` of variable values, keyed on variable names.

solve(*validate=False*)
Solves the optimization problem. Dispatches to a solver-specific method (only Gurobi bindings are implemented at this time).

solved()
Returns `True` if the problem has been solved, `False` otherwise.

solver(*val*)
Sets the solver (defaults to `` `SOLVER_GUROBI` `` in the class constructor). Note that only Gurobi solver bindings are implemented at this time.

var(*name*)
Returns a `Variable` instance, given a variable name.

var_names()
Return a list of variable names.

z(*coeffs=None*, *validate=False*)
Returns the objective function value if `coeffs` is not provided (triggers an exception if problem has not been solved yet), or updates the objective function coefficient vector (resets the value of the optimal solution to `None`).

class opt.**Variable**(*name*, *vtype*, *lb=0.0*, *ub=inf*, *val=None*)
Bases: `object`

Encapsulates data describing a variable in an optimization problem. This includes a variable name (should be unique within a problem, although the user is responsible for enforcing this condition), a variable type (should be one of `VTYPE_CONTINUOUS`, `VTYPE_INTEGER`, or `VTYPE_BINARY`), variable value bound (lower bound defaults to zero, upper bound defaults to positive infinity), and variable value (defaults to `None`).

2.5 `spatial` module

class spatial.**ForestRaster**(*hdt_map*, *hdt_func*, *src_path*, *snk_path*, *acodes*, *horizon*, *base_year*, *period_length=1*, *tif_compress='lzw'*, *tif_dtype='uint8'*, *piggyback_acodes=None*)
Bases: `object`

The `ForestRaster` class can be used to allocate an aspatial disturbance schedule (for example, an optimal solution to a wood supply problem generated by an instance of the `forest.ForestModel` class) to a rasterized representation of the forest inventory.

param hdt_map A dictionary mapping hash values to development types. The rasterized forest inventory is stored in a 2-layer GeoTIFF file. Pixel values for the first layer represent the *theme* values (i.e., the stratification variables used to stratify the forest inventory into development types). The value of the `hdt_map` parameter is used to *expand* hash value back into a tuple of theme values.

type hdt_map dict

param hdt_func A function that accepts a tuple of theme values, and returns a hash value. Must be the same function used to encode the rasterized forest inventory (see documentation of the `hdt_map` parameter, above).

param src_path Filesystem path pointing to the input GeoTIFF file (i.e., the rasterized forest inventory). Note that this file will be used as a model for the output GeoTIFF files (i.e., pixel matrix height and width, coordinate reference system, compression parameters, etc.).

param snk_path Filesystem path pointing to a directory where the output GeoTIFF files. The output GeoTIFF files are automatically created inside the class constructor method (one GeoTIFF file for each combination of disturbance type and year. If the disturbance schedule is from a `ForestModel` instance that uses multi-year periods, then the `ForestRaster` class automatically disaggregates the periodic solution into annual time steps.

param acodes List of disturbance codes.

param horizon Length of planning horizon (expressed as a number of periods).

param base_year Base year for numbering of annual time steps.

param period_length Length of planning period in the `ForestModel` instance used to generate the disturbance schedule.

param tiff_compress GeoTIFF compression mode (uses LZW lossless compression by default).

param tif_dtype Data type for output GeoTIFF files (defaults to `rasterio.uint8`, i.e., an 8-byte unsigned integer).

param piggyback_acodes A dictionary of list of tuples, describing piggyback disturbance parameters. By *piggyback* disturbance, we mean a disturbance that was not explicitly scheduled by the `ForestModel` instance, but rather is modelled as a (randomly-selected) subset of one of the explicitly modelled disturbances.

For example, if we want to model that 85% of pixels disturbed using the *clearcut* disturbance are disturbed by a piggybacked *slashburn* disturbance, we would pass

```
piggyback_acodes={'clearcut':[('slashburn', 0.85)]}.
```

allocate_schedule(*forestmodel*, *da=0*, *fudge=1.0*, *mask=None*, *verbose=False*)

cleanup()

commit()

grow()

transition_cells_random(*from_dtk*, *from_age*, *to_dtk*, *to_age*, *tarea*, *acode*, *dy*, *da=0*, *fudge=1.0*, *verbose=False*)

CHAPTER THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX