Neo-Processor Documentation

Last updated: 9/3/17

Created and maintained by: Joshua Petitmermet
Graduate Research Assistant
Oregon State University – Corvallis, OR
joshuapetitmermet@gmail.com

Documentation Contents

NeoProcessor.zip Key Contents

RxEvaluator.py	Script that performs basic prescription analysis, optimally bucking trees and defining the costs, revenues, and composite resistance score for all stands for all stands and treatments.
NeoProcessor.py	Uses the outputs from RxEvaluator to perform landscape level analysis and optimization using a modified Great Deluge Algorithm.

A Word on Documentation Structure

The documentation for each program is separated into four sections:

- The Model Flow section describes inputs, outputs, assumptions, and the rules the model uses to generate moves and find solutions.
- The User Control section describes variables and settings used to quickly and easily modify individual runs.
- The Object List section describes how data is stored in the model, including their units and formatting. Lastly,
- The Function List describes how objects and variables are used by the individual functions that, taken together, comprise each program.

The Model Flow section is intended to explain *what* each program does to someone who is familiar with heuristics and other optimization techniques but might be unfamiliar with Python or other similar programming languages. By contrast, the User Control, Object List and Function List describe *how* each program does what it does and is meant to guide those who might want to examine, use, or modify the code on their own.

Throughout the User Control, Object List, and Function List, important names are color coded:

- Green names are used when referencing objects such as fragments or cut trees.
- Blue names are used when referencing functions such as gendib or gensol.
- Orange names are used when referencing variables. This includes singular values such as flood, list names like frl, or dictionary names like rxf.

All objects and functions are defined in their respective sections in order of use, with the first objects/functions called within the program being the first defined.

A Word on Lists and Dictionaries

While a working knowledge of the Python programming language is not necessary to understand *what* each program does, two things are critical to understanding *how* they do what they do:

1. Lists are ordered sets of information with data stored at indexes. Individual values in a list are referenced by their index, with the first value in the list at [0] (index zero), the second at [1] (index one) and so on. Most objects in RxEvaluator and Neo-Processor are lists with information stored at specific indexes.

2. Dictionaries are sets of information that work on key/value pairs. When a given key is called within a dictionary it returns the value associated with that key. Most objects in RxEvaluator and Neo-Processor are stored in *nested* dictionaries. For example, fragment outputs are stored in rxf [standid] [rx] [ac], where rxf is a dictionary keyed by standid (the unique identifier for the stand the fragment is a part of), standid is also a dictionary keyed by rx (the

prescription code), and rx is also a dictionary keyed by ac (the number of acres in the fragment).

A Word on Species Codes and Material Definitions

Throughout Neo-Processor two distinct systems of species identification are used:

- FIA/FVS species codes, and
- Tree Eater species codes.

FVS species codes will generally correspond to FIA species codes, save for when species occur in a stand without being present in the variant assigned to that stand. When that happens FVS assigns the tree in question to the other softwood code (298) or other hardwood code (998) as appropriate. Tree Eater species codes are used to identify and (in some cases) group commercial species for analysis, using the same prices and taper equations for each species in the group.

Likewise, throughout Neo-Processor two different types of material are produced and tracked:

- Saw-quality material is made up of the bole wood of commercial species that can be cut into logs between eight and twenty eight feet in length and between six and twenty four inches in top-end diameter inside bark.
- Biochar feedstock is made up of the bole wood of all species that can be cut into logs of eight feet or more in length with a top-end diameter inside bark of four inches or greater.

Neo-Processor does not track the material from tops of less than four inches in diameter or branches. Neo-Processor was designed explicitly for use with cut-to-length systems where such material is generally used to make brush mats and masticated, compacted, and/or mixed with topsoil as a result.

Rx Evaluator Model Flow

Rx Evaluator is the first of two programs designed to substitute for the Processor module in BIOSUM, with the other being Neo-Processor. The decision was made to split Rx Evaluator and Neo-Processor to save on overall processing time. Rx Evaluator is a custom script written in Python 2.7 and executed in the PythonWin integrated development environment. Rx Evaluator requires the user to provide values for five different user controls (as described in the Rx Evaluator User Controls) and a number of supplemental inputs as tab-delimited text files.

Model Flow: Inputs and Initialization

Rx Evaluator requires a host of inputs, some are provided along with the script, and others must be supplied by the user. Both types of inputs are stored externally as tab-delimited text files. The provided inputs are:

- The parameters for calculating diameter inside bark, stored as dibparms.txt in the rxe_parameters subfolder

- The parameters for calculating green weights, stored as gwtparms.txt in the rxe_parameters subfolder
- The parameters for calculating predicted volume mortality, stored as mrtparms.txt in the rxe_parameters subfolder, and
- The Westside Scribner board foot volumes for logs between eight and twenty eight feet in length and between zero and twenty four inches in top-end diameter inside bark, stored as scribtable.txt in the rxe_parameters subfolder.

The user supplied inputs are:

- The facility capability data, stored as fcap.txt in the rxe_parameters subfolder
- The transport costs from each stand to each facility, stored in tcost.txt in the rxe_parameters subfolder
- The average yarding distance for each stand, stored as yard.txt in the rxe_parameters subfolder, and
- Four files for each prescription to be analyzed, saved as XXXc.txt, XXXl.txt, XXXp.txt, and XXXs.txt in the rxe_FVSinputs subfolder, where XXX is the prescription code. These files are generated from FVS output databases using SQL queries (provided in the Rx Evaluator User Control section).

All treatments use a three digit naming convention, where the first two digits are the treatment number and the last digit is the treatment period. Treatment code 999 is reserved and should always be used for the no-action alternative.

Once all inputs are provided and the script is run, Rx Evaluator performs a transportation analysis, identifying and storing the lowest cost route from each stand to each facility for each material type and (for saw-quality material) Tree Eater species code. After the lowest cost routes are identified, each prescription is evaluated in turn by passing through three different modules, Tree Eater, Accountant, and Fire Eater.

Model Flow: Tree Eater

The Tree Eater module uses the cut list for a given prescription to perform three key tasks:

- Optimally bucking each tree in the cut list capable of producing saw-quality material
- Calculating the volume of biochar feedstock produced by each tree in the cut list, and
- Using the outputs from that bucking and calculation to determine the expected yields per acre for each stand treated by that prescription.

By default, Tree Eater has six species codes comprised of twelve individual species, selected from the species present in the Upper Klamath basin. Species were selected for inclusion in Tree Eater by only if they met each of three criteria:

- Being a species acceptable to local mills for producing traditional, high value wood products (not pulp or biochar)
- Being a species likely to be removed during a treatment aimed at fuels reduction, and
- Being a species with available price and sort data.

Table NPD.1 – Default Tree Eater Species Codes

In FVS / FIA	
FIA Code	Common Name
11	Pacific silver fir
15	White fir
17	Grand fir
19	Subalpine fir
20	Red fir
21	Shasta fir
64	Western juniper
81	Incense cedar
93	Engelmann spruce
101	Whitebark pine
108	Lodgepole pine
116	Jeffrey pine
117	Sugar pine
119	Western white pine
122	Ponderosa pine
202	Douglas-fir
264	Mountain hemlock
321 / 998	Rocky mountain maple
431	Chinkapin
475	Mountain mahogany
746	Quaking aspen
763	Chokecherry
815	Oregon white oak
818	California black oak

Species Status for Bucking	
	Commercial Species
	Non-commercial Species

In Tree Eater	
TE Code	Common Name
1	Douglas-fir
2	Incense cedar
3	Jeffrey pine
3	Ponderosa pine
4	Sugar pine
5	Lodgepole pine
6	Pacific silver fir
6	White fir
6	Grand fir
6	Subalpine fir
6	Red fir
6	Shasta fir

By default, price and sort information is built to reflect the Klamath Unit average of the four most recent quarters available on the State of Oregon's Open Data website at the time of Neo-Processor's design, specifically quarters three and four of 2015 and quarters one and two of 2016. Five of the six Tree Eater species are priced with the same diameter class breakdown, but lodgepole pine has only a single "camp run" value. A camp run log is any log of greater than cull quality (Bell and Dillworth, 1988).

Table NPD.2 – Default Log Prices ($/MBF) by Sort and Species Aggregate

TE Code	Aggregate	Sort	2015.3	2015.4	2016.1	2016.2	Ave.
1	Douglas-fir	6"-8"	$465.00	$475.00	$490.00	$495.00	$481.25
		8"-14"	$510.00	$520.00	$525.00	$500.00	$513.75
		14"-22"	$520.00	$535.00	$530.00	$520.00	$526.25
		22"+	$525.00	$545.00	$535.00	$525.00	$532.50
2	Incense cedar	6"-8"	$650.00	$600.00	$650.00	$640.00	$635.00

		8"-14"	$650.00	$600.00	$650.00	$640.00	$635.00
		14"-22"	$650.00	$600.00	$650.00	$640.00	$635.00
		22"+	$650.00	$600.00	$650.00	$640.00	$635.00
3	Ponderosa pine	6"-8"	$285.00	$270.00	$265.00	$310.00	$282.50
		8"-14"	$320.00	$315.00	$305.00	$325.00	$316.25
		14"-22"	$355.00	$350.00	$335.00	$340.00	$345.00
		22"+	$390.00	$385.00	$365.00	$370.00	$377.50
4	Sugar pine	6"-8"	$285.00	$280.00	$275.00	$260.00	$275.00
		8"-14"	$305.00	$295.00	$300.00	$280.00	$295.00
		14"-22"	$335.00	$360.00	$320.00	$305.00	$330.00
		22"+	$365.00	$335.00	$345.00	$335.00	$345.00
5	Lodgepole pine	CR	$325.00	$320.00	$360.00	$365.00	$342.50
6	True fir	6"-8"	$380.00	$385.00	$370.00	$375.00	$377.50
		8"-14"	$410.00	$390.00	$380.00	$385.00	$391.25
		14"-22"	$415.00	$405.00	$410.00	$410.00	$410.00
		22" +	$420.00	$415.00	$415.00	$415.00	$416.25

Tree Eater's optimal bucking process uses a forward reaching algorithm (Denardo 2012) to find the best log or combination of logs that could come from any given tree. It does this by breaking each tree down into one foot increments called "nodes." Starting from an assumed one foot stump, Tree Eater moves up the tree node by node, identifying and evaluating all logs that could originate from each node and saving the value those logs to the last node of that log (the trim). As Tree Eater moves up the tree, it begins comparing different paths to the same node, saving the best it finds each time. After each node capable of creating a log has been evaluated, the highest value node will contain the optimal value for the tree.

Tree Eater only allows logs of eight, twelve, sixteen, twenty, twenty four, or twenty eight feet in length and assumes one additional foot per log of trim. By default, Tree Eater only generates nodes until the diameter inside bark for a node drops below four inches and will only attempt to create saw-quality logs while the top-end diameter inside bark is six inches or greater (this does not include trim).

To illustrate this process, assume Tree Eater finds a tree in the cut list with a valid Tree Eater species code, a diameter inside bark that falls below six inches twenty feet up the bole, and falls below four inches twenty nine feet up the bole. It would generate twenty eight nodes, and evaluate them from the bottom up. From the first node (in yellow) three logs (in brown) with one foot of trim (t) would be possible, of eight feet, twelve feet, and sixteen feet in length.

Height (ft)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Node Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Log 1										T										
Log 2														T						
Log 3																		T		

Let's further assume that an eight foot log is worth five dollars, a twelve foot log is worth seven dollars, and a sixteen foot log is worth nine dollars, and save the value of those logs to their last node (the trim).

Height (ft)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Node Value	0	0	0	0	0	0	0	0	0	5	0	0	0	7	0	0	0	9	0	0
Log 1										T										
Log 2														T						
Log 3																		T		

Once those values are saved Tree Eater moves onto the next node and the process repeats.

Height (ft)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Node Value	0	0	0	0	0	0	0	0	0	5	5	0	0	7	7	0	0	9	9	0
Log 1											T									
Log 2															T					
Log 3																			T	

Since the diameter inside bark drops below six inches at node twenty, node three is the last node with three possible logs.

Height (ft)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Node Value	0	0	0	0	0	0	0	0	0	5	5	5	0	7	7	7	0	9	9	9
Log 1												T								
Log 2																T				
Log 3																				T

At node five, cells begin to overlap, testing different paths to the same node against each other.

Height (ft)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Node Value	0	0	0	0	0	0	0	0	0	5	5	5	5	7	7	7	7	9	9	9
Path Value					0									5				7		
Log 1														T						
Log 2																		T		

In this case, the potential logs ending at node fourteen and eighteen are rejected because the model has already found a higher value path to those nodes. This changes at node ten.

Height (ft)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Node Value	0	0	0	0	0	0	0	0	0	5	5	5	5	7	7	7	7	9	9	9
Path Value										5									10	
Log 1																			T	

At node ten the path value is no longer just the value of the current log, it's the value of the current log and the previous log. The two eight foot logs earn more than the one sixteen foot log, so this new value is saved. The same thing happens at node eleven, and since node eleven is the last node capable of making a saw-quality log, bucking stops and the best value for the tree ($10) is saved.

Height (ft)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Node Value	0	0	0	0	0	0	0	0	0	5	5	5	5	7	7	7	7	9	10	10

Once the optimal value for the tree is found, the model determines how much (if any) biochar feedstock can be produced from the remainder. It does this by identifying the lowest node containing the optimal value and measuring the distance from that point to the last node. If that distance is eight feet or more, a feedstock log of that length is produced.

To go back to the previous example, there are two nodes after bucking (nineteen and twenty) that contain the optimal value. Node nineteen is the lowest. The diameter inside bark falls below four inches at twenty nine feet up the bole, so node twenty eight is the last node in the tree.

Height (ft)	17	18	19	20	21	22	23	24	25	26	27	28
Node Value	7	9	10	10	0	0	0	0	0	0	0	0
Feedstock Log												

The distance between the lowest optimal node and the last node is nine feet, so a nine foot feedstock log is created. For trees of non-commercial species, the entire volume of the tree to a four inch top is used as biochar feedstock. For trees of commercial species that are incapable of making a saw-quality log but capable of making a feedstock log, the entire volume of the tree to a four inch top is used as biochar feedstock.

As each cut tree is processed, Tree Eater calculates the per acre revenue, saw-quality volume and green weight, and biochar feedstock volume and weight associated with that cut tree and adds those yields to running totals for that tree's stand. After the last cut tree is processed, those per acre yields are passed on to the Accountant module.

Model Flow: Accountant

The Accountant module takes in the per acre yields from Tree Eater and uses them to calculate per acre costs as a function of the machine time required to handle the volume removed. It does this by calculating the cost of the harvest system and the cost to load and transport all saw-quality material and biochar feedstock produced by the treatment. Accountant assumes a tethered cut-to-length harvest system will be used for all treatments, with a harvester felling and bucking and a forwarder yarding and loading the material onto trucks for transport.

Harvester time is calculated as:

$$ht = 1.43tw + 0.25hdi$$

Where ht is the harvester time required in productive machine minutes, tw is the total weight of material handled in green tonnes, and hdi is the distance traveled in meters. hdi is held constant at 531 meters per acre and was calculated as two times the linear distance required to cover one acre with a corridor fifty feet wide. This may overestimate the actual distance required on flat ground where opportunistic wandering may allow the harvester to move from section to section without backtracking. It may also underestimate the actual distance on short, steep corridors where it fails to account for travel time between corridors. The harvester time in minutes is then converted to hours and multiplied by either $278.95 per productive machine hour for tethered operations or $212.47 per productive machine hour for untethered operations, producing a harvester cost per acre.

Forwarding time is calculated as:

$$ft = 0.05fdi + 2.19sac + 4.06pac \text{ on tethered ground, and}$$

$$ft = 0.01fdi + 2.19sac + 4.06pac \text{ on untethered ground}$$

Where ft is the forwarding time required in productive machine minutes, sac is the weight of sawlog material in green tonnes, pac is the weight of feedstock material in green tonnes, and fdi is the distance traveled by the forwarder in meters. fdi is calculated by multiplying the number of bunks per acre by the two times the average yarding distance. The number of bunks per acre is calculated by dividing the total green tonnes per acre by twelve and rounding to the nearest whole number. fdi is also subject to a minimum distance of 473 meters per acre, calculated as two times the linear distance required to cover one acre with a corridor sixty feet wide, with an additional thirty meters added per bunk to account for out of corridor travel to a pile. The forwarding time in minutes is then converted to hours and multiplied by either $198.11 per productive machine hour for tethered operations or $172.22 per productive machine hour for untethered operations, producing a forwarding cost per acre.

Truck loading time is calculated as:

$$lt = (sac / lsp) + (pac / lpp)$$

Where lt is the time spent loading in productive machine hours, lsp is the loading rate for saw material in green tonnes per hour and lpp is the loading rate for feedstock material in green tonnes per hour. Loading cost is then calculated as the loading time multiplied by $213.22 ($172.22 per productive machine hour for the forwarder plus $46 per productive hour for the idle truck).

The cost to transport material of a given type and species from each stand to each appropriate facility is calculated externally using the ArcGIS Network Analyst tool. Accountant takes those costs (in round trip dollars per green ton) and uses them to calculate the transport cost for each stand on a piece by piece basis, identifying the minimum cost route for each piece of saw-quality material and/or biochar feedstock and summing them to build the transportation costs for each stand.

Lastly, Accountant sums the per acre harvester, forwarding, loading, and transportation costs into a single per acre treatment cost. It is important to note that fixed costs per entry (such as those associated with mobilization) and any additional per acre costs (such as those associated with post

thinning prescribed burns) are not included in Rx Evaluator, but can be added and managed in Neo-Processor.

Model Flow: Fire Eater

After all costs, revenues, and yields have been calculated for a prescription, it passes through one final module before the final results are exported. Fire Eater calculates the combined resistance score (Jain *et al*. 2017) for each stand under each prescription. The composite resistance score is comprised of four components:

- Canopy base height / crown connectivity
- Canopy bulk density
- Basal area of fire resistant species, and
- Predicted volume mortality.

Each component is scored on a scale of zero to three points and those scores are summed to provide a composite resistance score between zero and twelve.

The canopy base height / crown connectivity is calculated in one of two ways. For stands with a single stratum (as defined by FVS defaults) Fire Eater uses the crown base height output from the FVS structure class table. For multi-strata stands, Fire Eater uses canopy connectivity, calculated as the distance between tallest tree of the lowest stratum and the lowest tree of the next-to-lowest stratum. Canopy base height and crown connectivity are both scored in the same fashion:

- Zero points for heights / distances of less than seven feet
- One point for heights / distances between seven and twenty feet
- Two points for heights / distances between twenty and thirty feet, and
- Three points for heights / distances of thirty feet or greater.

Canopy bulk density is taken directly from the FVS Potfire tables and is scored as follows:

- Zero points for densities of 0.15 kilograms per cubic meter or greater
- One point for densities between 0.1 and 0.15 kilograms per cubic meter
- Two points for densities between 0.05 and 0.1 kilograms per cubic meter, and
- Three points for densities of 0.05 kilograms per cubic meter or less

The basal area of resistant species is calculated as percent of total (expressed as a fractional value between zero and one). In Rx Evaluator six species are considered to be fire resistant:

- Red fir
- Shasta fir
- Jeffrey pine
- Ponderosa pine
- Sugar pine, and
- Douglas-fir

Total basal area and resistant basal area are calculated using FVS tree list records, with the tree list TPA adjusted to reflect TPA removed during treatment. The DBH of each tree is multiplied by the

Forester's Constant to get the basal area for that tree and then added to the running total for the stand and (depending on species) the running total for resistant species in the stand. Resistant basal area fraction (RBA) is scored by assigning:

- Zero points for an RBA of less than 0.25
- One point for an RBA between 0.25 and 0.5
- Two points for an RBA between 0.5 and 0.75, and
- Three points for an RBA of 0.75 or greater.

Predicted volume mortality is the estimated mortality in the event of a fire with six to eight foot flame lengths, as a percent of the total volume. Predicted volume mortality is calculated using parameters provided by the authors of Jain *et al* (2017) by multiplying the volume for each tree by the appropriate predicted mortality parameter, summing the predicted volume mortality, and dividing it by the total volume.

Table NPD.4 – Predicted Mortality Parameters by Species and DBH Class

	1-4.9"	5-9.9"	10-14.9"	15-20.9"	21-29.9"	30-39.9"	40-999"
White fir	0.99	0.99	0.65	0.46	0.14	0.09	0.065
Grand fir	1	0.9675	0.94	0.7325	0.47	0.47	0.47
Subalpine fir	1	0.9675	0.94	0.7325	0.47	0.47	0.47
Incense cedar	0.95	0.95	0.95	0.5	0.11	0.025	0.03
Engelmann spruce	0.99	0.99	0.955	0.73	0.72	0.655	0.655
Red fir	0.85	0.535	0.295	0.1425	0.09	0.09	0.09
Shasta fir	0.85	0.535	0.295	0.1425	0.09	0.09	0.09
Whitebark pine	1	0.99	0.9075	0.775	0.37	0.18	0.18
Lodgepole pine	1	0.99	0.9075	0.775	0.37	0.18	0.18
Sugar pine	1	1	0.905	0.56	0.2	0.16	0.125
Jeffrey pine	0.8	0.8	0.66	0.4475	0.2025	0.0725	0.06
Ponderosa pine	0.8	0.8	0.66	0.4475	0.2025	0.0725	0.06
Douglas-fir	0.98	0.98	0.705	0.54	0.275	0.305	0.215
Quaking aspen	1	1	1	1	1	1	1
Western juniper	1	0.991	0.951	0.839	0.708	0.596	0.47
Rocky mountain maple	1	0.991	0.951	0.839	0.708	0.596	0.47
Chinkapin	1	0.991	0.951	0.839	0.708	0.596	0.47
Mountain mahogany	1	0.991	0.951	0.839	0.708	0.596	0.47
Chokecherry	1	0.991	0.951	0.839	0.708	0.596	0.47
Oregon white oak	1	0.991	0.951	0.839	0.708	0.596	0.47
California black oak	1	0.991	0.951	0.839	0.708	0.596	0.47
Pacific silver fir	0.997	0.991	0.868	0.607	0.332	0.157	0.093
Western white pine	0.997	0.991	0.868	0.607	0.332	0.157	0.093
Mountain hemlock	0.997	0.991	0.868	0.607	0.332	0.157	0.093

After calculation, the predicted volume mortality is scored with:

- Zero points for predicted volume mortality of greater than 0.98
- One point for predicted volume mortality between 0.98 and 0.7
- Two points for predicted volume mortality between 0.7 and 0.4, and
- Three points for predicted volume mortality of 0.4 or less.

Composite resistance scores are calculated for each of five years of interest, the four treatment years (one, six, eleven, and sixteen) and the last year in the planning horizon (year twenty). In years where treatment occurs the composite resistance score is always calculated for post-treatment conditions.

Once the yields, revenues, costs, and scores are all calculated, Rx Evaluator packages them into an ordered list, with one such list for every valid stand/prescription combination. Rx Evaluator then exports those lists to a tab delimited text file (rxoutputs.txt) to be imported to and analyzed by Neo-Processor.

Rx Evaluator User Controls

Rx Evaluator has five built in user control variables for quick iteration and fine tuning of output generation (with Rx Evaluator). These variables are found on lines 865 to 874 of the Rx Evaluator script.

Assumed Bark Moisture Content and Assumed Wood Moisture Content

Variable name: mcbark and mcwood
Default value: 0
Appropriate inputs: See below

The assumed moisture content controls are used for calculating the green weight in tons of harvested material and, by extension, the transportation costs for that material. All moisture contents are calculated on a dry weight basis. These controls have three modes of operation:

- When set to zero, all green weights are calculated using the default values included in the FIA species reference tables.
- When set to a negative number, all green weights are calculated as the FIA defaults minus the number specified. The model defaults to a moisture content of 10% if this control would reduce it to less than 10%.
- When set to a positive number, the moisture content for all species is set to that value.

See the calcbarkwt and calcwoodwt functions for a complete description of the dry volume to wet weight equations used by Rx Evaluator.

Slope Breakpoint Control

Variable name: slopebp
Default value: 35
Appropriate inputs: any positive integer

The slope breakpoint control is used to determine whether the tethered or untethered harvest model is invoked for a given stand. When the slope for the stand is equal to or greater than the slope breakpoint, tethered calculations are used. See the coststst function for a complete description of the harvest time and cost equations used by Rx Evaluator.

Prescription Control List

Variable name: rxlist
Default value: NA
Appropriate inputs: any number of valid prescription codes as a comma separated list

The prescription control list holds the codes for each prescription being analyzed in the current run. It is highly recommended that the user break the list into separate lines (using a backslash as a line break) for easy inspection.

Facility Disable List

Variable name: fdisl
Default value: []
Appropriate inputs: any number of facility codes as a comma separated list

The facility disable list is a control designed to let the user remove facilities from consideration without removing those facilities from existing input files. It is important to note that this does not actually prevent facilities from being selected, instead setting their transport costs to $999.90 per green ton. This should prevent facility selection in the vast majority of circumstances, but could create anomalous results if most or all of the facilities that handle a given species are disabled in this way.

SQL Queries

Rx Evaluator relies on queried FVS tables to generate key inputs, those queries are listed here for convenience. The results of each query must be saved to an appropriately named text file and stored in the rxe_FVSinputs subfolder.

Cut List Query

Text file: XXXc.txt
Query: SELECT FVS_CutList.Species, FVS_CutList.DBH, FVS_CutList.Ht, FVS_CutList.TruncHt, FVS_CutList.PctCr, FVS_CutList.MDefect, FVS_CutList.TPA, FVS_CutList.MCuFt, FVS_CutList.StandID, FVS_CutList.TreeId, FVS_CutList.Year
FROM FVS_CutList;

Tree List Query

Text file: XXXl.txt
Query: SELECT FVS_TreeList.Species, FVS_TreeList.DBH, FVS_TreeList.Ht, FVS_TreeList.TruncHt, FVS_TreeList.PctCr, FVS_TreeList.MDefect, FVS_TreeList.TPA, FVS_TreeList.MCuFt, FVS_TreeList.StandID, FVS_TreeList.TreeId, FVS_TreeList.Year
FROM FVS_TreeList;

Potfire Query

Text file: XXXp.txt
Query: SELECT FVS_PotFire.StandID, FVS_PotFire.Year, FVS_PotFire.Canopy_Ht, FVS_PotFire.Canopy_Density
FROM FVS_PotFire;

Structure Class Query (Fuels Treatments Only)

Text file: XXXs.txt
Query: SELECT DISTINCT (FVS_StrClass.StandID) AS Expr1, FVS_StrClass.Year, FVS_StrClass.Number_of_Strata, FVS_StrClass.Stratum_1_Crown_Base, FVS_StrClass.Stratum_1_Sm_Ht, FVS_StrClass.Stratum_2_Lg_Ht, FVS_StrClass.Stratum_2_Sm_Ht, FVS_StrClass.Stratum_3_Lg_Ht
FROM FVS_StrClass
WHERE (((FVS_StrClass.Removal_Code)=1) AND ((FVS_StrClass.Year)<21));

Structure Class Query (No-Action Alternative)

Text file: 999s.txt
Query: SELECT DISTINCT (FVS_StrClass.StandID) AS Expr1, FVS_StrClass.Year, FVS_StrClass.Number_of_Strata, FVS_StrClass.Stratum_1_Crown_Base, FVS_StrClass.Stratum_1_Sm_Ht, FVS_StrClass.Stratum_2_Lg_Ht, FVS_StrClass.Stratum_2_Sm_Ht, FVS_StrClass.Stratum_3_Lg_Ht
FROM FVS_StrClass
WHERE (((FVS_StrClass.Year)<21));

Rx Evaluator Object List

The objects used by Rx Evaluator can all be classified as external objects, or internal objects. External objects are stored in tab-delimited text files and read into the model during the “Dictionary Loading” phase of operation. Internal objects are created by external objects interacting with functions and user inputs.

Object List: External Objects

External Trees

Source: Queried FVS tree list and cut list files
Stored externally in: XXXc.txt and XXXl.txt, where XXX is a valid prescription code
Stored internally in: NA

Field	Units	Index	Format	Description
Species	NA	0	Int	FVS tree species code
DBH	in	1	Float	Diameter at breast height in inches
Height	ft	2	Float	Height of the tree in feet (estimated for broken tops)
TrunHt	ft	3	Float	Truncated height of a tree with a broken top in feet
PctCr	%	4	Float	Percent crown ration
Mdefect	%	5	Float	Percent defect in merchantable volume
TPA	NA	6	Float	Trees per acre associated with the tree record
MCuFt	ft3	7	Float	Merchantable volume in cubic feet
StandID	NA	8	Str	Stand identifier associated with the tree
TreeID	NA	9	Int	Tree identifier
Year	NA	10	Int	FVS model year

Before they can become cut trees or leave trees, all trees in Rx Evaluator start in the cut list or tree list of FVS outputs. Those lists are queried to generate these external tree records and stored in the XXXc.txt and XXXl.txt respectively, where XXX is a valid prescription code. It is important to note that the merchantable cubic foot volume is measured to a four inch top and not the FVS default six inch top.

Potfire Values

Source: Queried FVS Potfire tables
Stored externally in: XXXp.txt, where XXX is a valid prescription code
Stored internally in: prx [rx] [standid] [year]

Field	Units	Index	Format	Description
Rx	NA	0	Int	Prescription code
StandID	NA	1	Str	Stand identifier associated with the record
Year	NA	2	Int	FVS model year
CBH	ft	3	Float	Height to crown base in feet (defunct)
CBD	kg/m^3	4	Float	Crown bulk density in kilograms per cubic meter

Potfire values contain the crown base height and crown bulk density as calculated by the FVS Potfire extension. The height to crown base is vestigial from earlier model versions and no longer used in Rx Evaluator.

Structure Class Inputs

Source: Queried FVS structure class tables
Stored externally in: XXXs.txt, where XXX is a valid prescription code
Stored internally in: hrx [rx] [standid] [year]

Field	Units	Index	Format	Description
Stand	NA	0	Str	Stand identifier associated with the record
Year	NA	1	Int	FVS model year
#Strata	NA	2	Int	The current number of strata in the stand
HCB	ft	3	Int	The height to crown base in feet
SmHt1	ft	4	Int	The height of the shortest tree in stratum one in feet
LgHt2	ft	5	Int	The height of the tallest tree in stratum two in feet
SmHt2	ft	6	Int	The height of the shortest tree in stratum two in feet
LgHt3	ft	7	Int	The height of the largest tree in stratum three in feet

The structure class inputs are used to calculate the structure class values and, by extension, the composite resistance scores for each stand under each prescription. See structure class values and dictionary loading for a full description of how these values are used.

Scribner Values

Source: Bell and Dillworth (1988)
Stored externally in: scribtable.txt
Stored internally in: slist [length] [topdib]

Scriber values hold the Revised Scribner board foot volumes for logs between eight and twenty eight feet in length (in four foot increments) and one and twenty four inches in top diameter inside bark (in one inch increments). These values are used during bucking to determine the volume of individual logs in each cut tree. The length index used to store scribner values is in four foot increments, such that an eight foot log is at [2], a twelve foot log is at [3], and so on. By default, twenty four foot and twenty eight foot logs are not considered during bucking.

DIB Parameters

Source: Garber and Maguire (2003), Hann (2016)
Stored externally in: dibparms.txt
Stored internally in: mdib [tesp]

DIB parameters are an ordered list of parameters used for calculating the diameter inside bark during bucking. The parameters for lodgepole pine are taken from the corrected Garber and Maguire (2003) and the parameters for all other species are taken from Hann (2016). The parameters are stored internally in a list (mdib) at the index appropriate for their tree eater species code (tesp). See gendib for a full description of the equations used.

Green Weight Parameters

Source: Queried FIA species reference table
Stored externally in: gwtparms.txt
Stored internally in: gwp [sp]

Field	Units	Index	Format	Description
BARK_VOL_PCT	%	0	Float	The volume of bark as a percent of wood volume
WOOD_SPGR	NA	1	Float	The specific gravity of wood
BARK_SPGR	NA	2	Float	The specific gravity of bark
MC_WOOD	%	3	Float	The default moisture content of wood
MC_BARK	%	4	Float	The default moisture content of bark

Green weight parameters are taken directly from the FIA species reference table and used to convert our harvest volumes into green ton weights for transport. Both the wood and bark specific gravities use a reference point of 62.4 pounds per cubic foot and both wood and bark moisture contents are measured on a dry weight basis. Green weight parameters are stored in a dictionary (gwp) keyed by FIA species code (sp). See calcbarkwt and calcwoodwt for a full description of the equations used.

Mortality Parameters

Source: Jain et al (2017)
Stored externally in: mrtparms.txt
Stored internally in: mrt [sp]

The mortality parameters are used for calculating the predicted volume mortality in the event of a fire as a function of tree size and species. Mortality parameters are stored in a dictionary (mrt) keyed by FVS species code (sp) with the predicted mortality proportion for trees 1-4.9” in DBH at [0], 5-9.9” at [1], 10-14.9” at [2], 15-20.9” at [3], 21-29.9” at [4], 30-39.9” at [5], and 40”+ at [6].

Facility Capability Data

Source: See below
Stored externally in: fcap.txt
Stored internally in: NA

Field	Units	Index	Format	Description
psite_id	NA	0	Int	Unique facility identifier
Name	NA	1	String	Facility name
Products	NA	2	String	Facility products
Feedstock	NA	3	String	Facility feedstock
PSME	NA	4	Int	Binary variable determining if facility accepts PSME
CADE	NA	5	Int	Binary variable determining if facility accepts CADE
PIPO	NA	6	Int	Binary variable determining if facility accepts PIPO

PILA	NA	7	Int	Binary variable determining if facility accepts PILA
PICO	NA	8	Int	Binary variable determining if facility accepts PICO
TFIR	NA	9	Int	Binary variable determining if facility accepts TFIR
City	NA	10	String	City associated with facility
State	NA	11	String	State associated with facility

The facility capability data is a list of existing mills and potential biochar processing facilities developed in collaboration with William Hollamon and David Smith of OSU. This data is used in the model to build species accept lists. Feedstock will always be a "P" or an "S", with the latter indicating a facility that accepts saw-quality material and the former indicating one that accepts biochar feedstock. The binary variables at [4] to [9] have a one if the mill accepts that species / species group and a zero if they do not.

Travel Cost Data

Source: See below
Stored externally in: tcost.txt
Stored internally in: travel [standid] [material] [fid]

The travel cost data is the round trip cost per green ton for the optimal route from each stand (standid) to each facility (fid). This cost data was calculated using the network analyst tool in ArcMap 10.4.1, the USGS National Transportation Datasets for Oregon and California, and truck capacity and prices provided by John Sessions of OSU. Travel cost data is stored in a dictionary keyed by standid, the material being moved (with a zero for biochar feedstock and a one for saw quality material), and the fid.

Yarding Data

Source: Queried BIOSUM master database
Stored externally in: yard.txt
Stored internally in: yard [standid]

The yarding data is the percent slope and average yarding distance in feet for each stand (standid) in the project area. The slope is taken directly from FIA measurements and the average yarding distance is calculated as the distance from fuzzed plot location to the nearest road. Yarding data is stored in a dictionary keyed by standid.

Object List: Internal Objects

Species Accept Lists

Source: Generated during Dictionary Loading
Stored in: mcap [fid]

Species accept lists store what material and species each facility accepts. Material is stored at as a zero or one at [0], with a zero indicating that the facility accepts biochar feedstock and a one indicating that the facility accepts saw-quality material. Indexes [1] through [6] indicate whether or not the facility accepts a given species / species group, with PSME at [1], CADE at [2], PIPO at [3], PILA at [4], PICO at [5], and TFIR at [6]. All species accept lists are stored in a dictionary keyed by facility ID (fid).

Optimal Route Costs

Source: Generated by evaltravel
Stored in: tcost [standid]

Optimal route costs store the cost to send material from a given stand to the lowest cost facility capable of handling that material, with the cost for biochar feedstock at [0], saw-quality PSME at [1], saw-quality CADE at [2], saw-quality PIPO at [3], saw-quality PILA at [4], saw quality PICO at [5], and saw quality TFIR at [6]. All optimal route costs are stored in a route cost dictionary (tcost) keyed by stand (standid).

Cut Trees

Source: Generated by incut
Stored in: trx [rx] [standid] [treeid]

Field	Units	Index	Format	Description
Species	NA	0	Int	FVS Species code
DBH	in	1	Float	Diameter at breast height in inches
Height	ft	2	Float	Height of the tree in feet (estimated for broken tops)
TrunHt	ft	3	Float	Truncated height of a tree with a broken top, in feet
PctCr	%	4	Float	Percent crown ratio
Mdefect	%	5	Float	Percent defect in merchantable volume
TPA	NA	6	Float	Trees per acre associated with the cut tree record
MCuFt	ft3	7	Float	Merchantable volume in cubic feet
StandID	NA	8	String	Stand identifier associated with the tree
TESC	NA	9	Int	Tree Eater species code
TreeID	NA	10	Int	Tree identifier
HCB	ft	11	Float	Height to crown base in feet
YearCut	NA	12	Int	Model year the tree was cut
Rx	NA	13	Int	Prescription associated with the cut tree record
Tprice	$	14	Float	The total price of the tree in dollars
Spulp	ft^3	15	Float	The volume of rejected material in cubic feet
LogPulp	ft^3	16	Float	The volume of biochar feedstock in cubic feet
MerchVol	ft^3	17	Float	The volume of saw-quality material in cubic feet
LogPulpGT	gt	18	Float	The green weight of biochar feedstock in tons
MerchGT	gt	19	Float	The green weight of saw-quality material in tons

Cut trees are used to store all the data associated with trees harvested during a fuels treatment. All values are calculated for a single tree and stored in a dictionary (trx) keyed by prescription (rx), stand (standid) and tree (treeid). See evaltree and the associated sub-functions for a full description of how individual cut tree values are calculated.

DIB Lists

Source: Generated by dibtree
Stored in: NA

When cut trees are being bucked the model creates a DIB list for the tree being processed, an ordered list containing the diameter inside bark at one foot increments. The model assumes a one foot stump and a minimum log length of eight feet, so the first nine indexes ([0] through [8]) of each dib list are left blank to save on processing time. By default, DIB lists are not stored after the tree in question has been bucked.

Node Lists

Source: Generated by gennodes
Stored in: NA

Like DIB lists, node lists are created while a cut tree is being bucked. Where dib lists store the diameter inside bark at a given height, node lists store the value of the optimal path to a given height. Upon creation, the first nine entries of a node list are empty, but once bucking is complete the best total value of the tree is saved to [0] for easy retrieval. By default, node lists are not stored after the tree in question has been bucked.

Per Acre Outputs

Source: Generated by incut
Stored in: orx [rx] [standid]

Field	Units	Index	Format	Description
Rx	NA	0	Int	Prescription associated with the outputs
StandID	NA	1	String	Stand identifier
Tprice	$	2	Float	Gross revenue in dollars
MerchVol	ft^3	3	Float	Total saw-quality material produced in cubic feet
LogPulp	ft^3	4	Float	Total biochar feedstock produced in cubic feet
MerchGT	gt	5	Float	Total weight of saw-quality material in green tons
PulpGT	gt	6	Float	Total weight of biochar feedstock in green tons

Per acre outputs store all the yields garnered from treating a single acre of a given stand (standid) with a given prescription (rx). All per acre outputs are stored in a dictionary (orx) keyed by rx and standid.

Leave Trees

Source: Generated by intree
Stored in: lrx [rx] [standid] [treeid] [year]

Field	Units	Index	Format	Description
Species	NA	0	Int	FVS species code
DBH	in	1	Float	Diameter at breast height in inches
TPA	NA	2	Float	Remaining trees per acre
StandID	NA	3	Str	Stand identifier associated with the tree record
TreeID	NA	4	Int	Tree identifier
Year	NA	5	Int	FVS model year
MCuFt	ft^3	6	Float	Merchantable volume in cubic feet
BA	ft^2	7	Int	Basal area in square feet
Rx	NA	8	Int	Prescription associated with tree record

Leave trees store information of interest for tree records that are not wholly cut during treatment. Remaining trees per acre is calculated after treatment for each year of interest (years one, six, eleven, sixteen, and twenty) as the FVS tree list TPA minus the FVS cut list TPA (if any). Basal area is calculated for a single tree. Leave trees are stored in a dictionary (lrx) keyed by prescription (rx), stand (standid), tree (treeid), and year of record (year).

Per Acre Leave

Source: Generated by intree
Stored in: srx [rx] [standid] [year]

Field	Units	Index	Format	Description
Rx	NA	0	Int	Prescription associated with the record
StandID	NA	1	Str	Stand identifier associated with the record
Year	NA	2	Int	FVS model year
TBA	ft^2	3	Float	Basal area in square feet
RBA	ft^2	4	Float	Basal area of fire resistant species in square feet
CBH	ft	5	Float	Height to crown base / crown connectivity in feet
CBD	kg/m^3	6	Float	Crown bulk density in kilograms per cubic meter
TVOL	ft^3	7	Float	Total volume in cubic feet
PMV	ft^3	8	Float	Predicted volume mortality in cubic feet

Per acre leave stores the information about what is left behind on one acre of a given stand with a given prescription. Per acre leaves are calculated for each stand for all years of interest (years one, six, eleven, sixteen, and twenty) and used primarily for calculating composite resistance scores. It's important to note that the CBH field is only the height to crown base for single strata stands, for multi-strata stands it is the distance in feet from the top of the lowest stratum to the bottom of the next to lowest stratum. Per acre leaves are stored in a dictionary (srx) keyed by prescription (rx), stand (standid) and model year (year).

Per Acre Costs

Source: Generated by coststst
Stored in: crx [rx] [standid]

Field	Units	Index	Format	Description
Rx	NA	0	Int	Prescription associated with the record
StandID	NA	1	Str	Stand identifier associated with the record
Tcost	$	2	Float	The total cost of treatment
Fcost	$	3	Float	Forwarder costs incurred during treatment in dollars
Hcost	$	4	Float	Harvester costs incurred during treatment in dollars
Lcost	$	5	Float	Loading costs incurred during treatment in dollars
Alt	NA	6	Int	See below
Mcost	$	7	Float	Transport costs incurred during treatment in dollars

Per acre costs store the costs incurred by assigning one acre of a given stand to a given prescription. Forwarder, harvester, and loading costs are all calculated using the cost data and productivity models created from Pilot Project observations. Transport costs are calculated as the sum of costs incurred in sending all biochar feedstock and saw-quality material to the lowest cost facility appropriate for that material.

The alt field is a binary variable used to generate warnings when no transportable saw-quality or biochar feedstock is produced by a treatment, with zero indicating that volume was produced for transport and a one indicating that no transportable volume was produced. Due to productivity model design Rx Evaluator may significantly underestimate costs in scenarios where no transportable material is produced.

Per acre costs are stored in a dictionary (crx) keyed by prescription (rx) and stand (standid).

Rx Outputs

Source: Generated by scorerx
Stored in: optout [rx] [standid]

Field	Units	Index	Format	Description
Rx	NA	0	Int	Prescription associated with the record

StandID	NA	1	Str	Stand identifier associated with the record
Grev	$/ac	2	Float	The per acre gross revenue in dollars
Cost	$/ac	3	Float	The per acre treatment cost in dollars
MerchT	gt/ac	4	Float	The per acre green ton weight of saw-quality material
PulpT	gt/ac	5	Float	The per acre green ton weight of biochar feedstock
CRS1	NA	6	Int	The composite resistance score for year one
CRS6	NA	7	Int	The composite resistance score for year six
CRS11	NA	8	Int	The composite resistance score for year eleven
CRS16	NA	9	Int	The composite resistance score for year sixteen
CRS20	NA	10	Int	The composite resistance score for year twenty
Mfac	NA	11	Int	Lowest cost mill site (defunct)
Pfac	NA	12	Int	Lowest cost biochar processing site (defunct)

Rx Outputs are the final product of Rx Evaluator, and the information that is imported by Neo-Processor for optimization. The Mfac and Pfac fields are vestigial from an earlier version of the software and should always return "999". Rx Outputs are stored in a dictionary (optout) keyed by prescription (rx) and stand (standid) and printed out to rxoutputs.txt.

Rx Evaluator Function List

Parameter Loading

While not a true function, parameter loading is a critical step in how the model operates. When the Rx Evaluator is run the first thing that happens is that it populates key dictionaries with internal and external data. The parameters loaded in order are:

- Scribner values, from scribtable.txt
- Log prices, by species and top diameter, stored internally
- Species equivalence, determining which FVS species codes correspond to which Tree Eater species code, stored internally
- Parameters for calculating diameter inside bark, from dibparms.txt
- Parameters for calculating green weight, from gwtparms.txt
- Parameters for calculating predicted volume mortality, from mrtparms.txt
- Facility capability data, from fcap.txt
- Travel cost data, from tcost.txt, and
- Yarding data (distances), from yard.txt.

disroute

The disroute function is called any time when the model is run with one or more values in the Facility Disable List (fdisl). For each facility in the disable list it cycles through the travel cost dictionary

(travel) and sets the route cost from each stand to that facility to $999.90 per green ton. As noted in the Facility Disable List description, this does not truly remove a facility from consideration and could result in anomalous outputs if all the facilities handling a given species are turned off at once.

The function modifies travel costs in place, and does not return an object.

evaltravel

Sub-functions: minroute

The evaltravel function creates and populates a route cost dictionary (tcost) with optimal route costs appropriate for each stand. It does this by cycling through each stand in the travel cost dictionary (travel) and determining the minimum cost facility for biochar feedstock and each saw-quality species / species group in that stand. The function returns the completed route cost dictionary keyed by stand (st).

minroute

The minroute function takes a stand of interest (st), the facility list (facl), travel cost data for the stand of interest, the species accept list dictionary (mcap), and a species of interest (sp) to find the lowest cost route from that stand to a facility capable of accepting the species of interest. The function returns the cost of that route as min.

evaltree

Sub-functions: gendib, dibtree, pricelog, gennodes, evalnode

The evaltree function is the core of the Tree Eater section of Rx Evaluator, finding the optimal combination of saw logs to maximize harvest value for a given tree. It does this by:

- Generating a DIB list for the tree with dibtree
- Generating an empty node list for the tree with gennodes, and
- Using evalnode to evaluate each node in the node list from [1] (the one foot stump) to the last node capable of creating a log (the node eight feet below the maximum height).

The process operates as a forward reaching algorithm (Denardo 2012), working its way up each node and saving the best results found (see evalnode for a full description of evaluation methodology). The function returns the completed node list with the dollar value of the optimal path for the tree of interest at [0].

gendib

The gendib function uses a tree of interest, a height of interest, and the appropriate DIB parameters to calculate the diameter inside bark according to the taper equations from Hann (2016) and Garber and Maguire (2003). It is important to note that the Garber and Maguire equations are in metric,

so the model converts the key equation inputs to centimeters (diameter) and meters (height), performs the equations, and then converts the result back to inches.

The function returns the diameter inside bark at the height of interest as dib.

dibtree

Sub-functions: gendib

The dibtree function takes a tree of interest and the appropriate DIB parameters and creates a DIB list for the tree. It does this by:

- Generating a list populated with zeros from [0] through [8]
- Using gendib to calculate the diameter inside bark nine feet up the tree, appending it to the DIB list, and
- Generating and appending new diameter inside bark values at one foot increments up the tree until either gendib returns a diameter of less than four inches or the top of the tree is reached (for broken topped trees).

The function returns the completed DIB list as diblist.

pricelog

The pricelog function takes the length, species, and top diameter inside bark of a log as uses them to price a log with those characteristics. It does this by:

- Looking up the board foot volume (bf) of a log with that length and top diameter (topdib, rounded to the nearest inch)
- Converting that volume to thousand board foot volume (mbf)
- Identifying the price bracket for the log according to topdib, and
- Multiplying the volume in thousand board feet by the price per thousand board feet for the appropriate species and price bracket.

Prices were calculated as the average of Klamath Unit pond values for the four most recent quarters available on the State of Oregon's Open Data website (https://data.oregon.gov/Natural-Resources/Log-Prices/4v4m-wr5p), specifically: quarters three and four of 2015 and quarters one and two of 2016. The function returns the price of the log as a float (p).

gennodes

The gennodes function takes a maximum height (either the top height of a tree or the height one foot before the top diameter drops below four inches) and generates node list of that length populated with zero at every index. The function returns the node list as nodes.

evalnode

Sub-functions: pricelog

The evalnode function takes a node list and a node of interest and identifies and evaluates all potential logs that could originate from that node. For every potential log length (length) stored in the log length list (llist) the function:

- Checks that the log would not violate the maximum height of the tree (maxht), and if it does not
- Identifies the target node (tnode) as the height of the node of interest (cnode) plus the length of the log and one foot of trim
- Checks to see if the top diameter of the log is six inches or greater, and
    - If it is, the value of the log is calculated with pricelog and added to the value of the best path going through the node of interest (nodes [cnode]) as the new path value (pathval)
    - If it is not, the value of the best path going through the node of interest is brought forward as the new path value
- Lastly, the new path value is checked against the current value in the target node (nodes [tnode]), overwriting that value if the new path value is higher.

The function returns the updated node list as nodes.

evalpulp

The evalpulp function takes the node list for an evaluated tree and determines the volume and characteristics of all non-saw bole material to a four inch top. It does this by:

- Identifying the optimal value of the tree
- Finding the lowest height where that value occurs (optht)
- Calculating the distance between optht and the maximum height for the tree (maxht) as pulplen, and
- Calculating the volume between optht and the maxht as a tapered cylinder.

The function returns a list (pulp) with the volume at [0] if pulplen is at least eight feet long or at [1] if the pulplen is less than eight feet long.

calcbarkwt and calcwoodwt

The calcbarkwt and calcwoodwt functions take a volume, species code (spcd), and assumed moisture content and uses them in conjunction with green weight parameters to convert volumes to weights in green pounds. They do this using:

(1 + bmc / 100) * (bvp / (100 + bvp)) * bsg * 62.4 * volume = bark biomass in green pounds, and

(1 + wmc / 100) * (1 - (bvp / (100 + bvp)) * wsg * 62.4 * volume = wood biomass in green pounds

Where:

- bmc and wmc are the assumed moisture content for bark and wood respectively, on a dry weight basis as set by the assumed moisture content user controls
- bvp (gwp [spcd] [0]) is the bark volume percent, the volume of bark as a percent of the total wood volume, taken from the FIADB species reference table
- bsg (gwp [spcd] [2]) and wsg (gwp [spcd] [1]) are the green volume to dry weight specific gravity for bark and wood respectively, taken from the FIADB species reference table
- 62.4 is the reference density for the specific gravities (in pounds per cubic foot), and
- volume is the cubic foot volume being converted.

Both functions internally label assumed moisture content mc, but Rx Evaluator runs them using one of two different global variables (mcbark and mcwood). As mentioned in the user controls section, if either function would have an assumed moisture content of less than 10%, that function will default back to a 10% assumed moisture content. The functions return wet weight in pounds as barkwt and woodwt respectively.

incut

Sub-functions: evaltree, calcbarkwt, calcwoodwt

The incut function reads in all the external trees in the cut list for a given treatment and uses them to generate a dictionary of all cut trees for that treatment and the per acre outputs for all stands treated. It does this by loading the XXXc.txt file, reading through it line by line, and for each line:

- Generating a temporary list (tlist) of the values on that line
- Setting the truncated height (d) equal to the total height (c) if it has a value of zero
- Checking if the FVS species code (a) has a corresponding Tree Eater species code and, if not, setting the Tree Eater species code (j) to zero
- Calculating the height to crown base (l)
- Generating a partial cut tree record (tree)
- Determining if the partial cut tree record is suitable for bucking and either
    - Bucking it with evaltree and appending the outputs to the partial cut tree, or
    - Assuming the entire volume (to a four inch top) will go to biochar feedstock and appending the relevant values to the partial cut tree
- Calculating and green weight of wood and bark with calcbarkwt and calcwoodwt, converting them to tons, and appending those values to complete the cut tree record.

Once the cut tree record is complete the function creates a per acre output object for the stand the tree is in (if one doesn't already exist), and adds the per acre revenue, volumes, and weights of the cut tree to the running totals for the stand. After each tree in the cut list has been processed in this way, the function returns a tuple containing a dictionary of cut trees (trx) at [0] and a dictionary of per acre outputs (orx) at [1].

loadprx

The loadprx function reads in the Potfire values for every stand under a given prescription from the XXXp.txt file, returning those values as a dictionary (prx) keyed by stand (a) and year (b). The function does not store values for stands that are not treated under the prescription being loaded, unless that prescription is the no action alternative (code 999).

loadhrx

The loadhrx function reads in the structure class inputs for every stand under a given prescription from the XXXs.txt file, and calculating the appropriate canopy connectivity metric for that stand. For stands with one or less strata (c <= 1), this is the height to crown base (d). For two strata stands (c = 2) this is the distance between the bottom of stratum one (tlist [4]) and the top of stratum two (tlist [5]). For three strata stands (c = 3) this is the distance between the bottom of stratum two (tlist [6]) and the top of stratum three (tlist [7]).

The function returns a structure class input dictionary (hrx) keyed stand (a) and year (b). Like loadprx, loadhrx does not store values for stands that are not treated under the prescription being loaded, unless that prescription is the no action alternative (code 999).

intree

Sub-functions: calcmort

The intree function loads all the external trees for a prescription from the XXXl.txt file and uses them to generate leave tree and per acre leave records for all stands treated under the prescription (or all stands when loading the no-action alternative). It does this by reading through the file line by line and for each line:

- Checking if the stand the tree is in was treated by checking the cut tree dictionary (trx) for that stand (d)
- Checking if the year of the record (f) is one of the years of interest (years one, six, eleven, sixteen, and twenty)
- Calculating the remaining TPA (c) as the tree list TPA minus the cut list TPA (if any) for that tree
- If the tree is in a stand of interest, a year of interest, and has a non-zero remaining TPA, the basal area (h) for that tree is calculated and a leave tree record is generated (treeyear).
- If a leave tree was generated, a per acre leave is created for the stand that tree is in (if one does not already exist) and the values for that tree are added to the running totals in that per acre leave.

Once all tree records in XXXl.txt have been processed, the function returns a tuple containing the leave tree dictionary (lrx) keyed by stand (d), tree (e), and year (f) at [0] and the per acre leave dictionary (srx) keyed by stand and year at [1].

calcmort

The calcmort function is used by intree to calculate the predicted volume mortality for each leave tree record to add that value to the per acre leave. It does this by looking up the mortality parameter appropriate for that tree (by species and DBH) and multiplying the volume of the tree by that parameter. The function returns the predicted volume mortality in cubic feet as mort.

coststt

The coststt function takes the per acre output for a stand under a given treatment and calculates the per acre cost as a function of harvesting and transportation costs. It does this by:

- Converting key inputs to metric equivalents
- Estimating the number of bunks per acre given the per acre material loading
- Calculating the productive machine minutes required for the harvester (ht), forwarding (ft), and loading (lt) and the associated costs for each (hc, fc, and lc respectively)
- Calculating the transport cost (mc) for each cut tree, and
- Summing all costs into a per acre total and compiling all the costs into a per acre cost

The bunk count per acre is estimated by dividing the total weight of material removed per acre (in green tonnes) by twelve, and rounding the result to the nearest whole number.

Harvester time is calculated as:

$$ht = 1.43tw + 0.25hdi$$

Where ht is the harvester time required in productive machine minutes, tw is the total weight of material handled in green tonnes, and hdi is the distance traveled in meters. hdi is held constant at 531 meters per acre and was calculated as two times the linear distance required to cover one acre with a corridor fifty feet wide. This may overestimate the actual distance required on flat ground where opportunistic wandering may allow the harvester to move from section to section without backtracking. It may also underestimate the actual distance on short, steep corridors where it fails to account for travel time between corridors.

Harvester cost in dollars per acre is calculated as:

$$hc = ht / 60 * 278.95 \text{ on tethered ground, and}$$

$$hc = ht / 60 * 212.47 \text{ on untethered ground}$$

Forwarder time is calculated as:

$$ft = 0.05fdi + 2.19sac + 4.06pac \text{ on tethered ground, and}$$

$$ft = 0.01fdi + 2.19sac + 4.06pac \text{ on untethered ground}$$

Where ft is the forwarding time required in productive machine minutes, sac is the weight of sawlog material in green tonnes, pac is the weight of feedstock material in green tonnes, and fdi is the distance traveled by the forwarder in meters. fdi is calculated by multiplying the number of bunks per acre by the two times the average yarding distance. fdi is also subject to a minimum distance of 473 meters per acre, calculated as two times the linear distance required to cover one acre with a corridor

sixty feet wide, with an additional thirty meters added per bunk to account for out of corridor travel to a pile.

Forwarder cost in dollars per acre is calculated as:

fc = ft / 60 * 198.11 on tethered ground, and

fc = ft / 60 * 172.22 on untethered ground

Loading time is calculated as:

lt = (sac / lsp) + (pac / lpp)

Where lt is the time spent loading in productive machine hours, lsp is the loading rate for saw material in green tonnes per hour and lpp is the loading rate for feedstock material in green tonnes per hour. Loading cost is then calculated as the loading time multiplied by $213.22 ($172 per productive machine hour for the forwarder plus $46 per hour for the idle truck).

Per acre transport costs (mc) are constructed by retrieving and summing the optimal route costs for each portion of each cut tree on the stand. All costs are then summed into a single total cost (tc) and used to construct a per acre cost for the stand. The function returns the per acre cost as a list.

costrx

Sub-functions: costst

The costrx function uses the costst function to generate a per acre cost for each stand treated by a given prescription. It does this by taking the per acre output dictionary for that prescription (orx) and running the costst function on every stand found in it. Once complete, the function reports the number of per acre costs generated and the number of stands that failed to produce transportable material (as having incurred “brushing costs”). If a significant number of stands are failing to produce transportable material it is recommended that those stands be removed from the analysis. This can be done before running Rx Evaluator by removing them from the stand list during the FVS runs of problem prescriptions, or in Neo-Processor by adding a modest additional cost per acre and limiting the selection criteria to self-paying prescriptions only.

The function returns a per acre cost dictionary as crx.

scorest

The scorest function takes a per acre leave (pal) and uses the information in it to generate the composite resistance score for that per acre leave. It does this by generating an empty list and appending the appropriate scores as follows:

- Crown base height / crown connectivity (pal [5])is scored by giving zero points for heights / distances of less than seven feet, one point for seven to twenty feet, two points for twenty to thirty feet, and three points for heights / distances of thirty feet or greater.

- Crown bulk density (pal [6]) is scored by giving zero points for bulk densities of greater than 0.15 kilograms per cubic meter, one point for between 0.15 and 0.10, two points for between 0.05 and 0.10, and three points for bulk densities of less than 0.05 kilograms per cubic meter.
- Resistant basal area portion (rbp) is scored by giving zero points if 25% or less of the total basal area being in resistant species, one point if between 25% and 50% is in resistant species, two points if between 50% and 75% is in resistant species, and three points if more than 75% of the total basal area is in resistant species.
- The predicted volume mortality is score by giving zero points if less than 2% of standing volume is predicted to survive a fire with six to eight foot flame lengths, one point if between 2% and 30% is predicted to survive, two points if between 30% and 60% is predicted to survive, and three points if more than 60% of standing volume is predicted to survive a fire with six to eight foot flame lengths.

Once all four categories have been scored, those scores are summed and the function returns a tuple containing the list of scores (score) at [0] and the sum of those scores (crs) at [1].

scorerx

Sub-functions: scorest

The scorerx function generates the Rx outputs for a given prescription. It does this cycling through every stand with a per acre leave in the per acre leave dictionary for that prescription (srx [rx]) and building the Rx output by:

- Generating a partial Rx output (opt) with revenues, costs, and weight of materials produced for the prescription and stand being processed
- Generating and appending the composite resistance scores (crs) for each year of interest (one, six, eleven, sixteen, and twenty) in order, and
- Appending two “999” values to the end of the Rx output.

The two 999 values are from a vestigial function in an earlier version of Rx Evaluator. The appending of composite resistance scores uses a try/except functionality to accommodate stands that are clearcut (and, by extension, have no leave tree records or per acre leaves). Stands that are clearcut are assigned a composite resistance score as though 100% of basal area was in resistant species, the crown base height was 100 feet, a crown bulk density of zero, and zero predicted volume mortality in the event of a fire.

The function returns a dictionary of Rx outputs (optout) keyed by stand.

loadblock

Sub-functions: incut, costrx, loadprx, loadhrx, intree, scorerx

The loadblock function is used to automate all the other functions in Rx Evaluator into a single, easily iterable process. Given a valid prescription code, assumed bark and wood moisture contents, and slope breakpoint, the function:

- Loads external trees from the cut list and generates cut trees and per acre outputs with incut
- Calculates per acre costs with costrx
- Loads Potfire values and structure class inputs with loadprx and loadhrx respectively
- Loads external trees from the tree list and generates leave trees and per acre leaves with intree, and
- Generates Rx outputs for the prescription with scorerx.

Like scorerx, loadblock returns a dictionary of Rx outputs as optout.

Neo-Processor Model Flow

Where Rx Evaluator performs tree level optimization and outputs stand level data, Neo-Processor takes that stand level data and uses it to perform landscape level optimization. Like Rx Evaluator, Neo-Processor is a custom script written in Python 2.7 and executed within the PythonWin integrated development environment. Neo-Processor requires the user to set values for eighteen user control variables (as described in the User Controls section) and supply two external files:

- A tab delimited text file containing the unique stand identifier and area (in acres) of each stand, named acres.txt, and
- The output file from Rx Evaluator, named rxoutputs.txt.

In the process of running, Neo-Processor passes through three distinct phases, initialization, optimization, and outputting.

Model Flow: Initialization

Initialization begins by importing the stand level data from acres.txt and rxoutputs.txt. This data is stored internally on a stand by stand basis. Those stands are then sorted into two groups, those with at least one valid fuels treatment, and those without. To save on processing time, stands with no valid fuels treatments are categorically excluded from optimization. Within the project area 354 of 825 stands (representing 45% of total federal forested acres) are excluded in this way.

Stands within the Upper Klamath are highly heterogeneous in terms of size, ranging from 3 to 7000 acres with an average size of over 2000 acres per stand. This means that if we were to optimize the landscape on a stand by stand basis that size and heterogeneity would serve to create an unrealistically discrete solution space. To resolve this, Neo-Processor uses fragments and not whole stands as the primary unit for decision-making.

Fragment size is specified by the user prior to running Neo-Processor. Then, during initialization the acre count of each stand is assessed one by one. If the acre count of the stand is less than the fragment size, that stand is used to create a single fragment. If the acre count is larger than the fragment size a fragment of that size is created and the acre count for the stand is reduced by that number of acres. This process is iterative, repeating until the number of acres remaining is less than the fragment size, and the remaining acres made into one final fragment. To illustrate, if you took a stand of

3500 acres and a fragment size of 1000 acres, that stand would create three fragments of 1000 acres each, and one fragment of the remaining 500 acres.

It must be noted that this system has to potential to effectively price out a small number of acres in each stand. For example, if you took a stand of 3002 acres and a fragment size of 1000 acres, you would create three fragments of 1000 acres each, and one fragment of 2 acres. That two acre fragment would be unlikely to ever breakeven due to the application of fixed costs per entry. This is problematic in theory, but has virtually no effect on the quality of solutions in practice, the few dozen acres adversely affected dwarfed by the hundreds of thousands of acres under consideration.

Once all stands have been turning into fragments, Neo-Processor finds and evaluates every unique fragment that could exist. A fragment is considered unique if no other fragment shares the same parent stand, prescription, and fragment size. No stand should produce more than two unique fragments per valid prescription, with one of the fragment size, and one of less than the fragment size. For each unique fragment, the outputs for treating that fragment are calculated including:

- Gross revenue
- Treatment cost
- Sawlog material produced
- Biochar feedstock produced
- The composite resistance score for years one, six, eleven, sixteen, and twenty, and
- Any additional per acre and/or fixed costs specified by the user

Fixed costs are defined in part by an average entry size (specified by the user) and will scale up when fragments are larger than the average entry size. Fixed costs do not scale down for fragments smaller than the average entry size to better represent costs that are largely insensitive to treatment size, such as those associated with mobilization.

Once all fragments are created and evaluated, an initial solution is generated through one of two mechanisms:

- If there is no even flow constraint and no limits on the number of acres treated in any period, a fully random initial solution is generated.
- If there is an even flow constraint or a limit on the number of acres treated in any period, a partially random solution is generated.

Fully random solutions are generated by moving through each fragment in order and assigning it one of the prescriptions appropriate for that fragment's parent stand, chosen at random. Partially random solutions are generated by

- Assigning a user-defined percent of fragments (selected randomly) to a prescription appropriate for that fragment's parent stand, chosen at random. If an assignment would violate the limitation on acres treated, that fragment is assigned the no-action alternative instead.
- The remaining fragments are used to generate a randomized list. The model then
    - Finds the treatment period with the lowest yield of interest (if even flow is enabled) or the period with the lowest number of acres treated (if even flow is disabled).

    - Cycles through the list until it finds a fragment with a treatment capable of improving that minimum
    - Assigns that fragment a prescription that improves the minimum (selecting one at random if more than one treatment is capable of doing so)
    - Updates the yield of interest and/or number of acres treated, and
    - Removes the fragment from the list. If prescription assignment would violate the limit on acres treated, that fragment is ignored as though it had no valid prescriptions capable of improving the minimum.
- This process repeats until either all fragments have been assigned a treatment, or no fragments are capable of improving the minimum. When no fragments are capable of improving the minimum, all remaining fragments are assigned the no-action alternative.

Early Neo-Processor designs struggled to effectively apply even flow constraints due to the extremely lop-sided nature of the solution space. Many stands are inoperable in early periods, but grow into operability over the course of the planning horizon. This operability in-growth combined with the fact that late-period prescriptions are removing more material results in fully random solutions that are highly skewed towards later periods in terms of acres treated and volumes produced.

Trial runs with fully random solutions and even flow constraints often spent half or more of their total move count simply seeking feasibility, resulting in very poor quality solutions. By contrast, partially random solutions virtually always start within acceptable bounds, resulting in much higher quality solutions.

Model Flow: Optimization

With the outputs for each fragment calculated and the initial solution generated, the model begins the actual work of optimization. This process can be broken into three distinct phases: move generation, move evaluation, and move implementation.

During move generation a random fragment is selected, the model looks up and copies the list of all prescriptions that could be applied to that fragment, removes the current prescription from the list, and selects one of the remaining prescriptions at random. Once both prescriptions are identified, a move effect is created. The move effect is an ordered list containing the revenue, cost, and biomass produced by both treatments, and the net effect on composite resistance score of moving from the old treatment to the new treatment in years one, six, eleven, sixteen, and twenty.

Move evaluation analyzes that move effect list. In order, it tests the move effect for:

- Violations of the limits on acres treated per period
- Violations of the even flow constraint, and
- The quality of the objective function.

Moves that would cause a violation in the number of acres treated are always rejected. Moves that would cause a violation in the even flow constraint are rejected unless the new solution is considered closer to feasibility than the current solution. Even flow is measured within the model as the percent of deviation in any period from the average for all periods in a given yield of interest (such as

net revenue, acres treated, or biochar feedstock produced). For example, if the average net revenue for all periods was $10,000 and the even flow constraint was set to 30% on net revenue, any move that resulted in a period with a net revenue higher than $13,000 or less than $7,000 would be rejected as infeasible unless the current solution was also infeasible. When both the current and new solution are infeasible, the squared deviation from the average is measured for each period and summed for both solutions. Then, the new solution will be accepted if it has a lower sum of squared deviations than the current solution.

The quality of the objective function is tested using a modified Great Deluge algorithm (Dueck 1993). Moves that do not cause a violation and are an improvement on the objective function are automatically accepted. Moves that do not cause a violation and reduce the objective function may also be accepted as an allowable disimprovement. What constitutes an allowable disimprovement changes over the course of each run according to the following rules:

- An allowable disimprovement may be no worse than the current objective function minus the “flood” value.
- The flood value is set to zero during solution generation and defaults back to zero if it would ever become negative.
- If the flood value is zero and an improvement is found, the flood value is set to 99% of the value of that improvement.
- If the flood value is a non-zero number, an improvement is found, and the improvement is less than or equal to ten times the current flood value, the current flood value is reduced by 50% of the value of the improvement. Lastly,
- If the flood value is a non-zero number, an improvement is found, and that improvement is more than ten times the current flood value, the flood value is set to 99% of the value of that improvement.

This resetting flood system is designed to force the model to become less and less accepting of disimprovement when repeatedly making small gains and disallow backtracking (where the model cycles back and forth between functionally identical moves), but still allow it to effectively explore the solution space when making gains in leaps and bounds.

Neo-Processor’s move evaluation also includes a “noise generation” function, designed to prevent the model from becoming trapped in local maxima for extended periods of time when limitations on the number of acres treated are enabled. If enough moves are rejected for violating the limitation on acres treated, noise generation is triggered. When noise generation is triggered, fragments are randomly selected and set to the no-action alternative until 10% of all fragments have been reset in this way.

When a move is accepted, it is immediately implemented. The fragment subject to the move is updated to the new prescription and the running totals for all yields of interest, the objective function, and the number of acres treated are all updated using the information from the move effect list. If the objective function value is better than the best objective function value ever seen or the solution is closer to feasibility than any solution ever seen, that solution is saved away for future use.

Model Flow: Outputting

Neo-Processor attempts a set number of moves in each run. The total number of moves is determined by the user as a function of the number of fragments being analyzed. The number of moves required to find a high quality solution varies by fragment size and the nature of the constraints applied, with more limiting constraints generally requiring more moves per fragment.

After running through the entire move count for a given run, Neo-Processor loads the best solution ever seen and reports the following:

- The total number of valid stand / prescription combinations
- The total number of acres in the project area, the number of acres broken into fragments, and the number of acres represented by all fragments
- The number of unique fragments identified and evaluated
- The type of initial solution generated (fully random or partially random)
- The final flood value
- The best objective function value ever found
- The number of improvements, accepted disimprovements, and rejected disimprovements
- The number of moves rejected due to infeasibility and the number of times noise generation was triggered (if any)
- The number of acres assigned to each treatment by the best solution ever found
- The number of acres treated in each period by the best solution ever found
- The net revenue, gross revenue, sawlog material, and biochar feedstock produced in each period by the best solution ever found
- The costs incurred in each period by the best solution ever found
- The composite resistance scores for years one, six, eleven, sixteen, and twenty for the entire project area under the best solution ever found and the no-action alternative, and
- The composite resistance scores for years one, six, eleven, sixteen, and twenty for operable acres only (those with a valid fuels treatment) under the best solution ever found and the no-action alternative.

In addition, the model outputs two tab delimited text files. One contains a list of all fragments with their parent stand identifier, acre count, and prescription under the best solution ever found. The other reports the total weight of material delivered to each facility in each period.

Neo-Processor User Controls

Neo-Processor includes eighteen user control variables meant to allow the user to quickly and easily fine tune model runs without significantly altering the program itself. These variables are found in a single code block from line 957 to line 974.

Initial Solution Control

Variable name: init
Default value: 0

Appropriate inputs: 0, 1, 2, 3, 999

The initial solution control serves two purposes: determining how initial solutions are generated with gensol, and determining whether or not the model disables prescriptions in the disable list (disl).

- If init is set to 0, the solution is randomly generated from all potential prescriptions
- If init is set to 1, the solution is randomly generated from all potential prescriptions except those in the disabled list
- If init is set to 2, the solution is randomly generated from all self-paying prescriptions
- If init is set to 3, the solution is randomly generated from all self-paying prescriptions save those in the disabled list
- If init is set to 999, each fragment is automatically assigned the no action alternative. This is generally only used for performing diagnostics and isolating errors.

While genlimsol does not directly use the initial solution control, it must still be set to one or three to disable prescriptions when using limited solutions.

Disable List

Variable name: disl
Default value: []
Appropriate inputs: any valid treatment code or combination of treatment codes

The disable list allows the user to remove treatments from consideration by entering any number of treatment codes in the list to disable them. Treatment codes must be entered between the square brackets with a comma separating each one. Note: this does not remove the treatments from Rx Outputs or Fragment Outputs, it only prevents the model from selecting the disabled prescriptions during initial solution generation and move generation.

Moves per Fragment Control

Variable name: mpf
Default value: 1000
Appropriate inputs: any positive integer

The moves per fragment control works with the fragment size control to determine how many total moves the model makes before reporting the best solution found. There is no effective limit on the number of moves the model can make, but higher move counts will increase running time. Moves per fragment should be modified in tandem with fragment size.

Fragment Size Control

Variable name: fragsize
Default value: 1000
Appropriate inputs: any non-negative number

The fragment size control determines the maximum fragment size in acres. Smaller fragment sizes allow for a more granular solution space when using even flow and/or area control constraints, but will increase running time. Moves per fragment and fragment size should be modified in tandem.

Biochar Feedstock Price Control

Variable name: biop
Default value: 0.0
Appropriate inputs: any non-negative number

The biochar feedstock price control determines the price in dollars per green ton associated with the sale of biochar feedstock.

Additional Variable Cost Control

Variable name: adac
Default value: 0.0
Appropriate inputs: any non-negative number

The additional variable cost control is used to apply additional dollar per acre costs to each fuels treatment in the analysis. This can be used to account for site prep, piling, burning, or any other cost that can be reasonably expressed on a per acre basis.

Entry Size Control

Variable name: ents
Default value: 100.0
Appropriate inputs: any non-negative number

The entry size control determines the assumed average entry size (in acres) for use in calculating fixed treatment costs.

Fixed Cost Control

Variable name: fixc
Default value: NA
Appropriate inputs: any non-negative number

The fixed cost control is used to apply a fixed dollars per entry cost to each fuels treatment. This includes but is not limited to mobilization costs. Fixed costs scale up if the fragment size is larger than the entry size, but do not scale down for fragments smaller than the entry size.

Objective Function Control

Variable name: obj

Default value: NA
Appropriate inputs: 0, 1, 2, 3

The objective function control determines what is being maximized for the current model run. It has four built in settings:

0 – Maximizing total net revenue over the planning horizon.

1 – Maximizing total gross revenue over the planning horizon.

2 – Maximizing total biomass removed over the planning horizon.

3 – Maximizing total composite resistance score over the planning horizon.

Self-Paying Mode Control

Variable name: spmode
Default value: 0
Appropriate inputs: 0, 1

The self-paying mode control determines whether or not the model can select revenue negative treatments when seeking a solution. Setting spmode to zero allows all treatments, setting it to one allows self-paying treatments only.

Even Flow Mode Control

Variable name: emode
Default value: 0
Appropriate inputs: 0, 1, 2, 3, 4, 5

The even flow mode control determines whether or not the even flow constraint is enforced and what it is enforced on. Neo-Processor has six built in settings:

0 – Even flow constraint is disabled.

1 – Even flow constraint is applied to net revenue.

2 – Even flow constraint is applied to gross revenue.

3 – Even flow constraint is applied to sawlog volume.

4 – Even flow constraint is applied to biochar feedstock volume.

5 – Even flow constraint is applied total acres treated.

Even Flow Control

Variable name: even
Default value: 0
Appropriate inputs: any positive integer between one and one hundred

Where even flow mode determines whether or not the even flow constraint is enforced, the even flow control determines how limiting that constraint is. When even flow is enabled, moves are rejected as infeasible if they would result in the yield at the period of interest being more than even percent away from the average yield for all periods, unless that move would result in a solution with less total deviation from the average than the current solution.

Acre Limitation Mode Control

Variable name: almode
Default value: 0
Appropriate inputs: 0, 1

The acre limitation mode control determines whether or not a limit is placed on the maximum number of acres treated per planning period. Setting almode to zero allows any number of acres to be treated in any period, while setting it to one enforces the limits as defined in the actlim variable.

Acre Limitation Control

Variable name: actlim
Default value: ["l", -1, -1, -1, -1]
Appropriate inputs: -1, 0, or any positive number

Just as even determines how limiting the even flow constraint is, actlim determines how limiting the acre limitation constraint is. The control itself is an ordered list (see results in the object list for a full description) that allows the user to determine how many acres may be treated by the model in any given period. For example, to limit the model to 50,000 acres in the first period and 75,000 in each other period, you would enter ["l", 50000, 75000, 75000, 75000]. If a value of -1 is entered in each field, the model will allow any number of acres to be treated even if almode is set to one.

Random Fraction Control

Variable name: ranf
Default value: 50
Appropriate inputs: any positive integer between 0 and 100

The random fraction control is used when generating the initial solution on runs subject to even flow or acre limitation constraints. It does this by determining what percent of fragments are assigned a prescription randomly and, by extension, what percent are "guided" (see genlimsol, randomlim, and guidedlim for a full description). The default value of 50 will work for most model runs, but the more limiting your constraints the lower the random fraction should be to ensure the highest quality initial solution.

Noise Control

Variable name: noise

Default value: 10.0
Appropriate inputs: any positive number between 1 and 100

The noise control determines what percent of fragments are set to the no-action alternative when the gennoise function is triggered. Noise generation is only used when the acre limitation constraint is enabled, and is designed to prevent the model from becoming trapped in local maxima for extended periods of time.

Noise Trigger Control

Variable name: noiset
Default value: 2000
Appropriate inputs: any positive integer

The noise trigger control determines how many moves must be rejected due to the acre limitation constraint before noise generation is triggered. This number is highly sensitive to the granularity of the solution space, with more granular solution spaces benefitting from high trigger values (to fully explore the local maxima before noise generation) and less granular solution spaces benefitting from lower trigger values (to spend less total time at any given local maxima and more time finding different local maxima). It is highly recommended to test a wide variety of noise trigger values for any given solution space.

Neo-Processor Object List

Rx Output

Source: Imported from Rx Evaluator
Stored In: rxe [standid] [rx]

Rx Outputs are ordered lists describing the per acre outputs of each valid stand / treatment combination. Rx Outputs are imported unaltered from Rx Evaluator and used only to calculate Fragment Outputs. Rx Outputs are stored in nested dictionaries keyed by stand identifier [standid] and prescription [rx].

Fragment

Source: Generated by genfrags
Stored in: frags [fid]

Field	Units	Index	Format	Description
FID	NA	0	Int	Fragment identifier
StandID	NA	1	Str	Stand identifier

Acres	ac	2	Float	Acres in fragment
Rx	NA	3	Int	Current prescription assigned to fragment

Fragments are ordered lists that serve as the primary decision-making unit for Neo-Processor. Maximum fragment size is set by the user prior to each run with smaller fragments providing a more granular solution space at the cost of an increase in time to solve.

Fragment Output

Source: Calculated by evalfrags
Stored in: rxf [standid] [rx] [ac]

Field	Units	Index	Format	Description
Rx	NA	0	Int	Treatment code
StandID	NA	1	Str	Stand identifier
Grev	$	2	Float	Gross revenue from treating fragment
Cost	$	3	Float	Cost of treating fragment
MerchT	gt	4	Float	Green tons of sawlog quality material
PulpT	gt	5	Float	Green tons of biochar feedstock
CRS1	NA	6	Int	Composite resistance score for year 1
CRS6	NA	7	Int	Composite resistance score for year 6
CRS11	NA	8	Int	Composite resistance score for year 11
CRS16	NA	9	Int	Composite resistance score for year 16
CRS20	NA	10	Int	Composite resistance score for year 20
Mfac	NA	11	Int	Lowest cost sawlog facility
Pfac	NA	12	Int	Lowest cost biochar facility
AddC	$	13	Float	User specified per acre and per entry costs

Fragment outputs are ordered lists describing the per fragment yields for each unique stand / treatment / fragment size combination. Fragment Outputs are used by multiple functions to evaluate a solution or the effects of move.

Move Effect

Source: generated by genmove
Stored in: mef

Field	Units	Index	Format	Description
NRx	NA	0	Int	The new prescription for the fragment
StandID	NA	1	Str	The stand identifier for the fragment being moved
FID	NA	2	Int	The fragment identifier for the fragment being moved
Ope	NA	3	Int	The current treatment period for the fragment

OG	$	4	Float	The revenue generated by the current treatment
OC	$	5	Float	The costs incurred by the current treatment
OM	gt	6	Float	The sawlog material produced by the current treatment
OP	gt	7	Float	The biochar feedstock produced by the current treatment
Npe	NA	8	Int	The new treatment period
NG	$	9	Float	The revenue generated by the new treatment
NC	$	10	Float	The costs incurred by the new treatment
NM	gt	11	Float	The sawlog material produced by the new treatment
NP	gt	12	Float	The biochar feedstock produced by the new treatment
se1	NA	13	Float	The effect of moving treatments on year one CRS
se6	NA	14	Float	The effect of moving treatments on year six CRS
se11	NA	15	Float	The effect of moving treatments on year eleven CRS
se16	NA	16	Float	The effect of moving treatments on year sixteen CRS
se20	NA	17	Float	The effect of moving treatments on year twenty CRS

A move effect is an ordered list containing all the information necessary to evaluate and make a move. When moving to or from the no action alternative the appropriate period of treatment is set to zero. The model generates and stores move effects one at a time, saving each to the mef variable.

Result

Source: see below
Stored in: see below

Per Period Result	Stored As	Identifier	Calculated By	Updated By
Total CRS	cfs	NA	calccfs	updatecfs
Operable CRS	ocfs	NA	calcopcfs	NA
Gross revenue	grev	"r"	calcecn	updateyield
Costs	cost	"c"	calcecn	updateyield
Sawlog material	mton	"m"	calcton	updateyield
Biochar feedstock	pton	"p"	calcton	updateyield
Net revenue	nrev	"n"	calcnrv	updateyield
Acres treated	act	"a"	calcact	updateact
Acre limits	actlim	“l”	NA	NA
Delivered material	deld	“dX”	calcdelivered	NA

Results are ordered lists containing the per-period yields for the current solution, with a unique identifier at [0] and the yield for each period at indexes [1] through [4]. Composite resistance score results do not contain an identifier, instead holding the CRS for each year of interest in order. The total CRS result holds the average CRS for all acres in the project area, while the operable CRS result holds the average CRS for acres with a valid fuel reduction treatment.

The actlim result is a special user-defined result that prevents the model from harvesting more than the number of acres specified for each period.

Unlike the other results, the delivered material result is a dictionary of results keyed by facility number. Delivered materials are only calculated for the best solution ever found as part of the end of run analysis. Each delivered material result is assigned a unique identifier of "dX" where X is the appropriate facility number.

Solution

Source: generated by savesol
Stored in: bsol

Solutions are ordered lists storing the prescription for each fragment at the index corresponding to its fragment identifier (i.e., the prescription for fragment 0 is stored at [0], fragment 1 at [1] and so on). The model only generates and saves a solution object when a new best solution is found, saving it as bsol.

Neo-Processor Function List

Output Loading

While not truly a compartmentalized function, output loading is the first thing Neo-Processor does and is required for the rest of the program to operate. Output Loading starts by creating a host of dictionaries and lists:

rxe – the dictionary used to store Rx Outputs.

rxl – a list of all treatment codes in Rx Outputs.

fcl – a list of all facility codes in Rx Outputs.

etd – a dictionary of all possible treatments for each stand, keyed by the stand identifier.

spd – a dictionary of all self-paying treatments for each stand, keyed by stand identifier.

per – a dictionary of all treatments in Rx Outputs, keyed by treatment period.

Once those are created, output loading opens the text file containing the Rx Outputs from RxEvaluator and for each line in the file:

- Converts the line into a temporary list (tlist)
- Saves the prescription as rx
- Adds the prescription to rxl if it is not present
- Adds the prescription to per if it not present and is not the no action alternative
- Saves the stand identifier as st

- Adds the stand identifier to rxe and etd if it is not present
- Adds the prescription to etd if it is not present
- Saves the per acre revenue and per acre costs as c and d respectively
- Adds the prescription to spd if it is not present and per acre revenues are greater than or equal to per acre costs.
- Saves the remaining Rx Output values to variables e through m
- Adds the sawlog facility and biochar facility to the facility list if they are not present
- Lastly, it combines rx, st, and variables c through m into a list and saves them to rxe [st] [rx].

After every line in the text file has been processed, Neo-Processor prints the number of stands and stand / prescription combinations encountered.

genfrags

The genfrags function is used to import the area of each stand and split each stand into fragments where appropriate. It begins by generating a set of variables:

frags – the dictionary used to store fragments

nas – a dictionary used to store stands with no valid fuels reduction treatments

frl – a list of all fragments

fcount – the current number of fragments processed

It then opens a text file containing the stand identifier and acre count for each stand on separate lines. For each line in the file it

- Converts the line to a temporary list (tlist)
- Saves the stand identifier as st
- Calculates the number of possible treatments for the stand (net) and saves the total number of acres as the current number of acres (cac)
- If the stand only has one possible treatment (the no action alternative) the stand is added to the no-action dictionary (nas) and not broken into fragments.
- If the stand has more than one possible treatment and the current acre count is greater than the maximum fragment size (fragsize), a fragment of the maximum size is generated and saved to the fragment dictionary (frags) and fragment list (frl) with the current fragment count (fcount) as its unique identifier. Then the current acre count is reduced and the fragment count increased.
- This process repeats until the current acre count is less than the maximum fragment size, and the remaining acres are used to generate one last fragment.

Once each line in the file has been processed Neo-Processor prints the total number of acres loaded, the number of acres with more than one prescription, and the number of acres with only one prescription (the no action alternative). The function returns a list containing the fragment dictionary at [0], the no action dictionary at [1], the fragment list at [2], the fragment count at [3], the number of

operable acres (oac) at [4], the number of inoperable acres (nac) at [5], and the total number of acres (tac) at [6].

evalfrags

The evalfrags function identifies all potential treatments for each unique fragment and calculates the fragment outputs for each of them. It does this by creating an empty fragment output dictionary (rxf) and cycling through each fragment in the fragment dictionary (frags). For each fragment it looks up all potential prescriptions in the potential prescriptions dictionary (etd). For each prescription found for each fragment:

- If the fragment's stand identifier (st) is not present in the fragment output dictionary (rxf), it is added as a dictionary
- If the current prescription (rx) is not present in rxf [st], it is also added as a dictionary.
- If the fragment's acre count (ac) is not present in rxf [st] [rx],
    - The per acre yields are looked up for that stand / prescription combination and are multiplied by the acre count of the fragment
    - The composite resistance scores for that stand / prescription combination are looked up, multiplied by the acre count of the fragment, and divided by the total number of acres in the project area
    - The facility for sawlog material and biochar feedstock are identified
    - If the entry size (ents) is less than the acre count of the fragment fixed costs (fixc) are scaled to the number of entries required (ac / ents * fixc). Fixed costs are not scaled if the entry size is larger than the acre count of the fragment
    - Additional per acre costs (adac) are multiplied by acre count of the fragment and added to the fixed costs
    - The resulting yields, scores, facilities, and costs are built into a fragment output list and stored as rxf [st] [rx] [ac].

In general, the model should find two unique fragments per stand / treatment combination. The only exception to this occurs when the acre counts for one or more stands are evenly divisible by the maximum fragment size. The evalfrags function returns the populated fragment output dictionary.

pricebio

The pricebio function is called when the user specifies a non-zero price for biochar feedstock (biop). The function updates each fragment output in the fragment output dictionary (rxf) by multiplying the biochar feedstock produced by biop and adding that value to the existing gross revenue. The function modifies each fragment output in place and does not return an object.

disablerx

If any prescriptions are in the disabled list (disl) and the initial solution control (init) is set to one or three, the disablerx function is called. The function works by cycling through each stand in the potential prescription dictionary (etd) or self-paying dictionary (spd) and removing any prescriptions on the disabled list. This prevents those prescriptions from being selected during initial solution generation or move generation for the current run.

The function prints the number of prescriptions disabled. If all fuels treatments for one or more stands are disabled (leaving only the no action alternative), the function will also print the number of stands and acres that no longer have a valid treatment. The disablerx function modifies its targets in place and does not return an object.

gensol

The gensol function is used for generating fully random initial solutions suitable for runs without even flow or area control constraints. Like genlimsol, gensol does not generate a solution object, instead randomly selecting a prescription for each fragment and modifying that fragment in place. The function has several different settings controlled by the initial solution control (init):

- If init is set to 0, the solution is randomly generated from all potential prescriptions
- If init is set to 1, the solution is randomly generated from all potential prescriptions except those in the disabled list
- If init is set to 2, the solution is randomly generated from all self-paying prescriptions
- If init is set to 3, the solution is randomly generated from all self-paying prescriptions save those in the disabled list
- If init is set to 999, each fragment is automatically assigned the no action alternative. This is generally only used for performing diagnostics and isolating errors.

genlimsol

Sub-functions: randomlim, genptd, guidedlim, updatelim

The genlimsol function is used for generating initial solutions suitable for runs with even flow and/or area control constraints (referred to as “limited solutions”). Like gensol, genlimsol does not generate a solution object, instead modifying each fragment in place. Unlike gensol, this function has two distinct phases of operation:

- The random phase, during which a percent of the total solution is generated by random assignment with the randomlim function. How many fragments are randomly assigned is managed with the random fraction control (ranf). And
- The guided phase, during which the rest of the solution is generated by the guidedlim function. In the guided phase the model preferentially selects prescriptions to improve feasibility. If no feasibility improving moves can be found, the remaining fragments are set to the no-action alternative.

Throughout both phases the function uses a randomized fragment list (ranl) and assignment count (count) to ensure that each fragment is assigned a prescription. The function prints the number of attempted random assignments, the number of random assignments rejected for violating area control constraints, the number of guided assignments, and the number of fragments set to the no action alternative.

randomlim

Sub-functions: updatelim

The randomlim function is used to generate the random portion of a limited solution. It brings in the randomized fragment list (ranl) from genlimsol and generates two empty results, one for acres treated (act) and one for the result subject to the even flow constraint (eyield), if any. It then moves through the randomized fragment list assigning a random prescription to each one until it has filled the percent of total fragments set by the random fraction control.

The act and eyield results are updated with every move by the updatelim function. If a prescription assignment would violate the acre limitation constraint in any period, that prescription is instead set to the no-action alternative.

The randomlim function returns a tuple containing:

- A list with act at [0], the number of assignments made (count) at [1], and the number of assignments rejected and set to the no-action alternative (vioac) at [2]. And,
- The eyield result, or an "x" if the even flow constraint is disabled.

updatelim

The updatelim function is used to keep a running tally of results while generating a limited solution. It does this by looking up the appropriate value in the fragment output dictionary (rxf) and adding it to the appropriate period of the result being updated.

guidedlim

Sub-functions: updatelim, genptd

The guidedlim function is used to build the guided portion of a limited solution and utilizes the act, eyield, and count values returned by randomlim. It begins by slicing the random fragment list (ranl) to include only those fragments that were not assigned a prescription by randomlim, and then randomizing this truncated list. The genptd function is then called to produce a dictionary of treatments keyed by stand identifier and treatment period (ptd).

Once ptd is generated the model finds the eyield period with the lowest value (or the act period with the lowest value if the even flow constraint is disabled) and cycles through the truncated random fragment list until it finds a fragment with a valid prescription that can increase that minimum value,

that prescription is assigned to that fragment, and that fragment is removed from the randomized fragment list. If that fragment is associated with more than one treatment capable of increasing the minimum value, one of those treatments is assigned at random.

If a prescription assignment would violate the acre limitation constraint, that fragment is ignored for that assignment as though it had no valid treatments. If the model cycles through the entire random fragment list and fails to find a fragment capable of improving the minimum all remaining fragments are assigned the no-action alternative.

The function returns a list with act at [0], the number of successful guided moves (gmove) at [1], the number of fragments set to the no action alternative (nmove) at [2], and the eyield (or "x") at [3].

genptd

The genptd function is used to generate a dictionary keyed by stand identifier and treatment period for use in generating limited solutions. It does this by creating an empty period of application dictionary (ptd) and cycling through every fragment in the fragment dictionary (frags). For each fragment it

- Adds the stand identifier (st) associated with that fragment to ptd if it is not present
- Looks up st in the prescription dictionary (rxd) and for each fuels treatment there it
    - Looks up the period of application for that treatment
    - Adds that period (rxp) to ptd [st] as a list if it is not present, and
    - Appends the treatment code to ptd [st] [rxp].

After each fragment has been checked, the function returns the completed dictionary.

calccfs and calcopcfs

The calccfs function is used to calculate the composite resistance score for years one, six, eleven, sixteen, and twenty for the current state of each fragment in the fragment dictionary (frags) and the stands in the no-action dictionary (nas). It does this by generating an empty result (cfs) and cycling through each no-action stand and fragment, calculating their contribution to the average composite resistance score for the entire project area in each year of interest, and adding those contributions to the running totals for each year. Once every no-action stand and fragment is accounted for, the function returns cfs.

The function can also be set evaluate the no-action alternative for the project area by setting mode to 999. The calcopcfs function operates in fundamentally the same way, but only looks at fragments, not no-action stands.

calcecn, calcton, and calcact

The calcecn function calculates the economic outputs for the current state of each fragment in the fragment dictionary (frags). It does this by generating two empty results (grev and cost) and cycling through each fragment, looking up its associated revenue and costs in the fragment output dictionary (rxf) and adding them to the running total. Once each fragment is accounted for the function returns a tuple containing grev at [0] and cost at [1].

The calcton function operates in fundamentally the same way, but tallies the green ton weight of sawlog material and biochar feedstock, returning their results as mton at [0] and pton at [1] respectively. The calcact function uses the same methods as calcton and calcecn, but returns the acres treated result (act) as a list, not as part of a tuple.

calcnrv

The calcnrv function calculates the net revenue for the current state of each fragment in the fragment dictionary (frags) by generating an empty result (nrev) and adding the revenue minus the cost (taken from grev and cost respectively) for each period. The function returns the completed result.

calccov

The calccov function calculates the current objective function value by summing the result (or results) appropriate for the current objective function. See obj in the user controls section for full description of objective function settings. Returns the summed value as cov.

genmove

The genmove function is used to generate move effects for evaluation. It does this by

- Selecting a random fragment from the fragment list (frl)
- Identifying the current prescription (crx) for that fragment
- Generating a list of valid prescriptions (prx) for that fragment, removing the current prescription, and selecting one of the remaining prescriptions at random as the new prescription (nrx)
- Generating a new move effect (mef) with the new prescription at [0], stand identifier (st) at [1], fragment identifier (fr) at [2], and old treatment period (ope) at [3]
- Appending the revenue, costs, sawlog material, and biochar feedstock associated with the old prescription to mef
- Appending the new treatment period (npe), revenue, cost, sawlog material, and biochar feedstock associated with the new prescription to mef, and
- Calculating the net effect of the new treatment on the composite resistance score for each year of interest (se1 through se5) and appending those effects.

The function returns the completed move effect as a list.

evalmove

Sub-functions: testactl, testeven, genbounds, testbounds, and calcdev

The evalmove function is the linchpin of the Neo-Processor program, determining whether moves created by genmove are accepted or rejected. The function uses a modified Great Deluge selection criteria, seeking improvement but capable of accepting limited disimprovement to better explore the solution space. The function is built around four key variables, the acceptance of the move (accept), the state of move assessment (fin), the tolerance for disimprovement (flood), and the improvement found (imp). The function works in three distinct stages:

- Stage one checks for violations of using the testactl and testeven functions. If the move is found to be in violation it will be assigned an accept value of -3 (rejected for violating the acre limitation constraint), -2 (rejected for violating the even flow constraint) or -1 (accepted for moving the model closer to meeting even flow constraints) and fin will be set to 1, passing the move through the other stages without further assessment.
- If fin is still set to 0, stage two evaluates the effect of the move on the objective function on its own merits. If the move would improve the objective function value, accept is set to 1 (accepted for improving the objective function), and the size of the improvement is saved as imp. If the effect of the move is no worse than the current value of the objective function minus the current flood value, accept is set to 2 (accepted as an allowable disimprovement). If the effect of the move is worse than the objective function minus the current flood value, accept remains 0 (move rejected).
- If an improvement is found, stage three is triggered, updating the flood value.

The flood value starts each run set to zero and defaults back to zero if it ever becomes negative. If the flood value is zero and an improvement is found, the flood value is set to 99% of that improvement. If the flood value is a non-zero number and an improvement is found, one of two things happens, either

- The flood value is reduced by 50% of the found improvement, or
- If the improvement found is more than ten times the current flood value, the flood value is set to 99% of the value of that improvement.

This design ensures that the model can adequately explore the solution space while improving in leaps and bounds, but will become more and more intolerant of disimprovement while fine tuning a solution.

The function returns a tuple containing the accept value at [0] and the flood value at [1].

testactl

The testactl function is used by the evalmove function to determine whether or not the move would cause the solution to become infeasible. It does this by adding the number of acres subject to the move (ac) to the current number of acres being treated in the new treatment period act [np] and comparing that number to the limitation imposed on that period (actlim [np]). If the move would

generate a violation, accept is set to -3 to reject the move and fin is set to 1 to prevent the model from analyzing the move further.

The function returns a tuple containing accept at [0] and fin at [1].

testeven

Sub-functions, genbounds, testbounds, calcdev

The testeven function is used by the evalmove function to assess the effect of the move on the even flow constraint. Even flow in Neo-Processor is measured as the deviation of a result in any given period from the average of that result for all periods. If the move would cause the result for the old treatment period (oy) or the result for the new treatment period (ny) to be more than even percent away from the average, fin is set to zero and accept is set to -2. When this occurs the model performs a deviation test, calculating the sum of squared deviations of the new solution (ndev), comparing it to the sum of squared deviations of the old solution (odev), and setting accept to -1 if the new deviation is lower than the old.

Like testactl, testeven returns a tuple containing accept at [0] and fin at [1].

genbounds

The genbounds function finds the average (ave) for the result subject to the even flow constraint (cyield) and uses that average to calculate the minimum acceptable value for any given period (lb) and the maximum value acceptable for any given period (ub).

The function returns a tuple containing the upper bound at [0] and the lower bound at [1].

testbounds

The testbounds function compares the result in a period of interest (tp) to the upper and lower bounds calculated by genbounds. If that result of interest (cyield [tp]) is more extreme than either bound, fin is set to 1 to trigger a deviation test and prevent the model from further assessing the move.

The function returns the fin value of zero (not in violation) or one (in violation).

calcdev

The calcdev function calculates the total deviation of each period in a result to the average value for that result. It does this by finding the average (ave), taking the absolute value of the average minus the value for each period (individually, stored in variables dis1 through dis4), squaring each of those deviations and summing them.

The function returns that sum of squared deviations as dev.

updatecov, updateyield, updatecfs, and updateact

The update functions are called when a move is accepted to update key values of interest:

- The objective function value (updatecov)
- The results for gross revenue, net revenue, cost, sawlog material, and biochar feedstock (updateyield)
- The result for acres treated (updateact), and
- The result for composite resistance score (updatecfs).

Non-CRS values are updated by subtracting the values for the old treatment (unless the old treatment was the no-action alternative) and adding the values for the new treatment (unless the new treatment is the no-action alternative). CRS values are updated by adding the move effect variables se1 through se5 to the appropriate year in the cfs result. The values for all of these changes are taken directly from the accepted move effect. Each of these functions returns the updated result as a list, save for updatecov which returns the updated objective function value as a float.

savesol

When a move is accepted that generates a better objective function value than the best solution ever found, the savesol function is used to save the prescription for each fragment to a new solution. When constraints are enabled savesol will also be called to overwrite infeasible solutions with feasible solutions or solutions closer to feasibility regardless of the objective function value.

Regardless of the triggering event, savesol will always return the new solution as bsol.

loadsol

When Neo-Processor has run through its total move count, the best solution ever found (bsol) is loaded for further analysis using the loadsol function. The function moves through every fragment in the fragment dictionary (frags) and reassigns them to the prescription they had under the best solution ever found.

The function modifies each fragment in place and does not return an object.

calcdelivered

After Neo-Processor has run through its total move count and loaded the best solution ever found, the calcdelivered function is called to calculate the total biomass delivered in each period on a facility by facility basis. It does this by taking the sorted facility list (fcl) and creating a dictionary of results (deld) keyed by facility number. Once the dictionary and associated results are created, the function cycles through every fragment in the fragment dictionary (frags) and for each one

- Saves the stand identifier as st, fragment size as ac, and prescription as rx
- If any sawlog material is produced by that prescription, it looks up the sawlog facility and adds that biomass to the running total for that facility. And
- If any biochar feedstock is produced by that prescription, it looks up the biochar processing facility and adds that biomass to the running total for that facility.

After cycling through each fragment the function returns the completed delivered material dictionary (deld).

printout

The printout function is a general purpose function used for rapid diagnostic work. The function takes a list (olist) and file handle (fhandle) and prints each index in the list to the file in the file handle as a single tab delimited line followed by a new line character (\n). The function does not return an object or close the file handle.

References

Bell, J.F. and Dilworth, J.R., 1988. Log scaling and timber cruising. OSU Book Stores. Inc., Corvallis, Oregon.

Denardo, Eric V. *Dynamic programming: models and applications*. Courier Corporation, 2012.

Dueck, Gunter. "New optimization heuristics: The great deluge algorithm and the record-to-record travel." *Journal of Computational physics* 104.1 (1993): 86-92.

Garber, Sean M., and Douglas A. Maguire. "Modeling stem taper of three central Oregon species using nonlinear mixed effects models and autoregressive error structures." *Forest Ecology and Management* 179.1 (2003): 507-522.

Hann, David W. "Revised volume and taper equations for six major conifer species in southwest Oregon." *Department of Forest Engineering, Resources, and Management. Oregon State University, Corvallis, Oregon. Forest Biometrics Research Paper 2* (2016).

Jain, T.J; J.S. Fried; R.F. Keefe; S. Loreno; C. Bell. 2017. Evaluating cost-effectiveness of multi-purpose fuel treatments in western mixed-conifer forests considering hazard, risk, longevity and co-benefits: Final Project Report to Joint Fire Sciences Program. [draft in preparation].