FEAST

Release 3.1

Chandler Kemp and Clay Bell

Nov 12, 2020

CONTENTS:

CHAPTER

ONE

FEAST MODULES

1.1 MEET_1_importer

gas_comp_to_dict (*gc*, *delta_t*)

converts a gas_comp DataFrame to a dict specifying the start time, top time, emission rate, location, emission type and emission ID for every emitter in gas_comp. If an emission changes its emission rate, it is recorded as stopping and restarting at the time of the change. LDAR programs will treat the emission correctly due to the emission ID that persists after the change in emission rate.

Parameters

- **gc** – a gas_comp DataFrame as created by the function load_gas_comp_file
- **delta_t** – The time resolution of the gas_comp file (seconds)

Returns the dict containing emitter data from the gas_comp file

gascomp_reader (*path_to_gas_comp*, *feast_delta_t*, *duration*, *comps_per_site*, *rep_cost_path*, *met_data_path=None*, *met_start_hr=None*)

Read a MEET 1.0 GasComp result file and return a feast Time object and GasField object specifying all of the emissions to occur in a FEAST simulation

Parameters

- **path_to_gas_comp** – path to a gas_comp file
- **feast_delta_t** – time resolution of the FEAST simulation (days). FEAST will represent emissions with the precision of the original MEET simulation, but will invoke LDAR events as though they occur instantaneously with the time resolution specified here.
- **duration** – duration of the simulation to run in FEAST (days)
- **comps_per_site** – A dict of form {loc: number of components}. This will specify the number of components to be inspected for leaks at every site
- **rep_cost_path** – Path to a repair cost data file. The FEAST will randomly assign these repair costs to leaks simulated in MEET.
- **met_data_path** – Path to a meteorological data file (if left as None, FEAST will run without met data)
- **met_start_hr** – FEAST will rotate the meteorological data to begin at the hour specified by an integer here.

Return time_obj a FEAST Time object to specify a simulation

Return gas_field a FEAST GasField object

gc_dat_to_gas_field(*feast_delta_t*, *duration*, *comps_per_site*, *rep_cost_path*, *met_data_path=None*, *met_start_hr=None*, *size=0*, *loc=0*, *start=0*, *stop=0*, *emtype=0*, *em_id=None*)

Ports data from a data_dict returned by the funcation gas_comp_to_dict into a FEAST GasField

Parameters

- **feast_delta_t** – FEAST time resolution (days)
- **duration** – FEAST simulation duration (days)
- **comps_per_site** – A dict of form {loc: number of components}. This will specify the number of components to be inspected for leaks at every site
- **rep_cost_path** – Path to a repair cost data file. The FEAST will randomly assign these repair costs to leaks simulated in MEET
- **met_data_path** – Path to a meteorological data file (if left as None, FEAST will run without met data)
- **met_start_hr** – FEAST will rotate the meteorological data to begin at the hour specified by an integer here.
- **size** – an array of emission sizes (g/s)
- **loc** – an array of site ids
- **start** – an array of emission start times (days)
- **stop** – an array of emission stop times (days)
- **emtype** – an array of emission types (reparable or no)
- **em_id** – an array of emission IDs

Returns a FEAST Time object

Returns a FEAST GasField object

load_gas_comp_file(*path_to_gas_comp*)

Loads a gas_comp.csv file generated by MEET 1.0

Parameters **path_to_gas_comp** – a path (str) to a gas_comp file

Returns a DataFrame of the gas_comp file

Returns the time resolution of meet simulation (seconds)

1.2 input_data_classes

This module defines all classes used to store input data.

class DataFile(*notes='No notes provided'*, *raw_file_name=None*, *data_prep_file=None*)

DataFile is an abstract super class that all data file types inherit from FEAST.

Parameters

- **notes** – A string containing notes on the object created
- **raw_file_name** – path (or list of paths) to a raw input file(s)
- **data_prep_file** – path to the file used to process input data and create the DataFile object

class LeakData(*notes='No notes provided'*, *raw_file_name=None*, *data_prep_file=None*, *leak_sizes=None*)

LeakData is designed to store all leak size data from a reference. It accommodates multiple detection methods within a single instance.

Creates a LeakData object

Parameters

- **notes** – a string containing notes on the object created
- **raw_file_name** – path to a raw input file
- **data_prep_file** – path to a script used to build the object from the raw data file
- **leak_sizes** – list of leak sizes. If leaks were detected using multiple methods, leak_sizes must be a dict with one key for each detection method

define_data(*leak_data=None*, *well_counts=None*, *comp_counts=None*, *detect_methods=None*)

Check data formatting and set keys. . . there is exactly one detection method, leak_data may be a list.

Parameters

- **leak_data** – leak_data must be a dict of emission rates if there are multiple detection methods. If there is exactly one detection method, leak_data may be a list.
- **well_counts** – lists the number of wells inspected by each detection method in keys
- **comp_counts** – lists the number of components inspected by each detection method in keys
- **detect_methods** – each detection method should have a unique key associated with it

Returns None

class ProductionData(*site_prod=None*, ***kwargs*)

Stores an array of production rates that may be associated with gas production sites

Parameters

- **prod_dat** – an array of production rates
- **kwargs** – pass through

class RepairData(*notes='No notes provided'*, *raw_file_name=None*)

RepairData is designed to store the costs of repairing leaks from a particular reference andd associated notes.

Parameters

- **notes** – A string containing notes on the object created
- **raw_file_name** – path to a raw input file

define_data(*repair_costs=None*)

Parameters **repair_costs** – list of costs to repair leaks

1.3 Detection Modules

1.3.1 ldar_program

This module defines the LDARProgram class.

class LDARProgram(*time*, *gas_field*, *tech_dict*)

An LDAR program contains one or more detection methods and one or more repair methods. Each LDAR program records the find and repair costs associated with all detection and repair methods in the program. The LDAR program deploys runs the action methods of each detection and repair method contained in the program. The detection and repair methods determine their own behavior at each time step.

Parameters

- **time** – a Time object
- **gas_field** – a GasField object
- **tech_dict** – a dict containing all of the detection methods to be employed by the LDAR program. The dict must have the form {"name": DetectionMethod}. All of the relationships between detection methods and between detection methods and repair methods must be defined by the dispatch_objects specified for each method.

action(*time*, *gas_field*)

Runs the detect method for every tech in tech_dict and runs the repair method :param time: the simulation time object :param gas_field: the simulation gas_field object :return:

1.3.2 abstract_detection_method

class DetectionMethod(*time*, *detection_variables=None*, *op_envelope=None*, *ophrs=None*)

DetectionMethod is an abstract super class that defines the form required for all detection methods

Parameters **time** – a Time object

static check_min_max_condition(*condition*, *params*)

Checks a min-max condition defined by params. Supports float, integer, list and array based min max conditions. If the min-max condition is specified as a min float/integer and max float/integer, the numbers are placed in min and max lists each with length 1. The function returns True if the condition is between the min and max values, False otherwise. If the min and max values are array-like, the function returns true if the condition is between any pair of min-max values.

Parameters

- **condition** – condition to check (must be a number)
- **params** – a dict with 'min' and 'max' keys. The min and max values can be numbers or array-like.

Returns

check_op_envelope(*gas_field*, *time*, *site_index=None*)

Returns the status of the operating envelope. The method supports 8 types of operating envelope conditions:

1. A meteorological condition based on min-max values that apply to the whole field (eg. temperature)
2. A meteorological condition based on min-max values that are site-specific (eg. wind direction)
3. A meteorological condition based on a fail list that applies to the whole field (eg. precipitation type)
4. A meteorological condition based on a fail list that is site specific (possible but not expected)

5. A site condition based on min-max values that apply to the whole field (eg site production)
6. A site condition based on min-max values that are site-specific (possible but not expected)
7. A site condition based on a fail list that applies to the whole field (eg. site type)
8. A site condition based on a fail list that is site specific (possible but not expected).

Parameters

- **gas_field** – A feast GasField object
- **time** – A feast Time object
- **site_index** – Index to a specific site

Return status A string specifying the result of the operating envelope check. Can be one of 4 strings:

1. 'field pass'
2. 'field fail'
3. 'site pass'
4. 'site fail

check_time (*time*)

Determines whether or not the detection method is active during the present time step

Parameters **time** – A Time object

Returns True if check_time passes, False otherwise

choose_sites (*gas_field*, *time*, *n_sites*, *clear_sites=True*)

Identifies sites to survey at this time step

Parameters

- **gas_field** – A GasField object
- **time** – A Time object
- **n_sites** – Max number of sites to survey at this time step
- **clear_sites** – If true, clear sites selected from the queue. If False, leave sites in the queue. Leaving sites in the queue is useful for SiteMonitor type detection methods.

Returns None

static empirical_interpolator (*test_conditions*, *test_results*, *sim_conditions*)

Calculates the probabiity of detection by interpolating the value of test_results between test_conditions.

Parameters

- **test_conditions** – conditions to be interpolated from
- **test_results** – results associated with each condition listed in test_conditions
- **sim_conditions** – Nxk array of current conditions, where N is the number of emissions to consider, and k is the number of conditions

Returns an array of the probabilities of detection (dimension N)

extend_site_queue (*site_inds*)

Add new sites to the site_queue if they are not already in the queue

Parameters `site_inds` – List of indexes to add to the queue

Returns None

`static find_comp_name`(*gas_field*, *sitename*, *comp_index*)
Determines the key for a component based on its index and site

Parameters

- `gas_field` – a GasField object
- `sitename` – name of the site containing the component
- `comp_index` – index of the component to consider

Returns The key for the component identified by comp_index, or -1 if the component is not found.

`static find_site_name`(*gas_field*, *site_index*)
Determines the key for a site based on its index :param gas_field: a GasField object :param site_index: an integer indicating the index of the site to be considered :return: the key for the site identified by site_index, or -1 if the key cannot be found.

`get_current_conditions`(*time*, *gas_field*, *emissions*, *em_id*)
Extracts conditions specified in self.detection_variables

Parameters

- `time` – a Time object
- `gas_field` – a GasField object
- `emissions` – a DataFrame of current emissions
- `em_id` – emission indexes to consider

Return conditions an array (n_emissions, n_variables) of conditions for use in the PoD calculation

1.3.3 comp_survey

This module defines the component level survey based detection class, CompSurvey.

`class CompSurvey`(*time*, *dispatch_object*, *survey_interval*, *survey_speed*, *labor*, *site_queue*, *detection_probability_points*, *detection_probabilities*, *ophrs*, *comp_survey_index=0*, *site_survey_index=0*, *op_env_wait_time=7*, ***kwargs*)
Bases: `feast.DetectionModules.abstract_detection_method.DetectionMethod`

This class specifies a component level, survey based detection method. A component level method identifies the specific component that is the source of the emission at the time of detection. Examples of components include connectors, valves, open ended lines, etc. A survey based method inspects emissions at specific moments in time (as opposed to a monitor method that continuously monitors for emissions). The class has three essential attributes:

1. An operating envelope function to determine if conditions satisfy requirements for the method to be deployed
2. A probability of detection surface function to determine which emissions are detected
3. The ability to call a follow up action

Parameters

- `time` – a Time object

- **dispatch_object** – an object to dispatch for follow-up actions (typically a Repair method)
- **survey_interval** – Time between surveys (float–days)
- **survey_speed** – Speed of surveys (float–components/hr)
- **labor** – Cost of surveys (float–$/hr)
- **site_queue** – Sites to survey (list of ints)
- **detection_probability_points** – Set of conditions at which the probability was measured (array)
- **detection_probabilities** – Set of probabilities that were measured (array)
- **comp_survey_index** – Index of the component to be surveyed next (int)
- **site_survey_index** – Index of the site to be surveyed next (or currently under survey) (int)
- **op_env_wait_time** – Time to wait if operating envelope conditions fail part way through a site before moving to the next site (float–days)
- **ophrs** – range of times of day when the method performs survey (dict–hours). eg: {'begin': 8, 'end': 17}

action (*site_inds=None*, *emit_inds=None*)

Adds sites to the queue for future inspections. This method is expected to be called by another detection method or by an LDAR program.

Parameters

- **site_inds** – List of sites to add to the queue
- **emit_inds** – Not used.

Returns None

detect (*time*, *gas_field*, *emissions*)

The detect method checks that the current time is within operating hours, selects emitters to inspect, determines which emissions are detected and dispatches the follow up action for detected emissions.

Parameters

- **time** – a Time object
- **gas_field** – a GasField object
- **emissions** – a DataFrame of current emissions

detect_prob_curve (*time*, *gas_field*, *em_surveyed*, *emissions*)

This function determines which leaks are found given an array of indexes defined by "cond." The method uses attributes of the DetectionMethod and interpolation.

Parameters

- **time** – a Time object
- **gas_field** – a GasField object
- **em_surveyed** – Array of emission_id to consider
- **emissions** – a DataFrame of current emissions

Return detect the indexes of detected leaks (array of ints)

emitters_surveyed(*time*, *gas_field*, *emissions*)

Determines which emitters are surveyed during the current time step. Accounts for the number of components surveyed per timestep, the number of components at each site, and the component and site at which the survey left off in the previous time step

Parameters

- **time** – a Time object
- **gas_field** – a GasField object
- **emissions** – a DataFrame of current emissions

Return emitter_inds emission_id of emissions to evaluate at this timestep (list of ints)

1.3.4 site_survey

The site_survey module defines the site level level survey based detection class, SiteSurvey.

class SiteSurvey(*time*, *dispatch_object*, *sites_per_day*, *site_cost*, *detection_probability_points*, *detection_probabilities*, *op_envelope=None*, *ophrs=None*, *site_queue=None*, *survey_interval=None*, ***kwargs*)

Bases: *feast.DetectionModules.abstract_detection_method.DetectionMethod*

SiteSurvey specifies a site level, survey based detection method. A site level detection method is sensitive to the total emissions from a site. If emissions are detected, the site is identified as the source of emissions rather than a component on the site. Survey based detection methods search for emissions at a specific moment in time (as opposed to monitor detection methods that continuously scan sites for new emissions). The class has three essential attributes:

1. An operating envelope function to determine if the detection method can be applied
2. A probability of detection surface function to determine which emissions are detected
3. The ability to dispatch a follow up action

Parameters

- **time** – a Time object
- **dispatch_object** – the object that SiteSurvey will pass flagged site indexes to (DetectionMethod or Repair)
- **sites_per_day** – the number of sites that the method can survey in one day (int)
- **site_cost** – the cost per site of the detection method ($/site–float)
- **detection_probability_points** – The conditions at which the detection probability was measured. (NxM array, where N is the number of distinct conditions and M is the number of variables (up to two)).
- **detection_probabilities** – The list of probabilities of detection associated with every point in detection_probability_points (array of shape N, where N is the number of conditions with an associated probability of detection).
- **op_envelope** – The set of conditions underwhich the SiteSurvey may operate. The op_envelope must be passed as a dict with the following form–

  {'parameter name': {'class': int, 'min': list of minimum conditions, 'max': list of maximum conditions}}

Unique minima can be defined for every site in a list if the op_envelope 'class' is site specific. Multiple minima can be defined in a list for a single site if multiple ranges should be considered.

- **ophrs** – The times of day when the SiteSurvey can be deployed. Should be a dict:

  {'begin': hour integer, 'end': hour integer}

- **site_queue** – an ordered list of sites to be surveyed. An LDAR program may update this list.

- **survey_interval** – The time between surveys (int–days)

action (*site_inds=None*, *emit_inds=None*)

Action to add sites to queue. Expected to be called by another detection method or by an LDAR program

Parameters

- **site_inds** – List of sites to add to the queue
- **emit_inds** – Not used.

Returns None

detect (*time*, *gas_field*, *emissions*)

The detection method implements a survey-based detection method model

Parameters

- **time** – an object of type Time (defined in feast_classes)
- **gas_field** – an object of type GasField (defined in feast_classes)
- **emissions** – an Emissions object

Returns None

detect_prob_curve (*time*, *gas_field*, *site_inds*, *emissions*)

This function determines which sites are passed to the dispatch_object by SiteSurvey. The function sums all emissions at a site, determines the probability of detection given the total site emissions and present conditions, then determines whether or not the site is flagged according to the probability.

Parameters

- **time** – Simulation time object
- **gas_field** – Simulation gas_field object
- **site_inds** – The set of sites to be considered
- **emissions** – an object storing all emissions in the simulation

Return detect the indexes of detected leaks

sites_surveyed (*gas_field*, *time*)

Determines which sites are surveyed during the current time step. Accounts for the number of sites surveyed per timestep

Parameters

- **gas_field** –
- **time** –

Return site_inds the indexes of sites to be surveyed during this timestep.

1.3.5 site_monitor

The site_monitor module defines the site monitor detection class, SiteMonitor.

class SiteMonitor (*time*, *dispatch_object*, *time_to_detect_points*, *time_to_detect_days*, *ophrs=None*, *capital=0*, *site_queue=None*, ***kwargs*)

Bases: `feast.DetectionModules.abstract_detection_method.DetectionMethod`

This class specifies a site level continuous monitoring method. A site monitor continuously observes emissions from an entire site and determines when an action should be dispatched at the site. The method has three essential characteristics:

1. A list of the sites where the method applies
2. A time-to-detect surface specified as a list of conditions and associated mean detection times
3. The ability to dispatch a follow up action

Parameters

- **time** – a Time object
- **dispatch_object** – the object to dispatch for follow up actions
- **time_to_detect_points** – The conditions at which the time to detection was measured. (NxM array, where N is the number of distinct conditions and M is the number of variables (up to two)).
- **time_to_detect_days** – The list of probabilities of detection associated with every point in detection_probability_points (array of shape N, where N is the number of conditions with an associated probability of detection).
- **ophrs** – The times of day when the SiteMonitor is operational. Should be a dict:

  {'begin': hour integer, 'end': hour integer}
- **capital** – The total cost of installing the site monitor system in the simulation (float–$)
- **site_queue** – A list of sites where the site monitor system is installed

action (*site_inds=None*, *emit_inds=None*)

Action to add sites to queue. Expected to be called by another detection method or by an LDAR program

Parameters

- **site_inds** – List of sites to add to the queue
- **emit_inds** – Not used.

Returns None

detect (*time*, *gas_field*, *emissions*)

The detection method implements a continuous monitor detection method model

Parameters

- **time** – a Time object
- **gas_field** – a GasField object
- **emissions** – a DataFrame containing emission data to evaluate

Returns None

detect_prob_curve (*time*, *gas_field*, *site_inds*, *emissions*)
Determines which sites are passed to the dispatch method. In this case, the sites to pass are determined by calculating a probability of detection based on the simulation time resolution (time.delta_t) and the mean time to detection

Parameters

- **time** – simulation Time object
- **gas_field** – simulation GasField object
- **site_inds** – the set of sites to be considered
- **emissions** – an object storing all emissions in the simulation

Return detect the indexes of detected leaks

static prob_detection (*time*, *ttd*)
Calculates the probability of detection during a timestep of length time.delta_t and a mean time to detection ttd

Parameters

- **time** – Simulation time object
- **ttd** – mean time to detection (float–days)

Returns the probability of detection during this timestep

1.3.6 repair

This module defines the Repair class. Repair may be called by detection objects as follow up actions.

class Repair (*repair_delay=0*, *name=None*)
Bases: `object`

Defines a repair process. A repair process determines when emissions are ended by an LDAR program and the associated costs.

Parameters **repair_delay** – The time between when an emission is passed to Repair and when it is removed from the simulation (float–days)

action (*site_inds=None*, *emit_inds=None*)
adds emissions to the to_repair queue.

Parameters

- **site_inds** – not used
- **emit_inds** – A list of emission indexes to repair

Returns None

repair (*time*, *emissions*)
Adjusts the emission end time based on the current time and the repair delay time If the null emission end time comes before the repair time, the end time is not changed

Parameters

- **time** – a Time object
- **emissions** – an Emission object

Returns None

1.4 EmissionSimModules

1.4.1 emission

A class for storing emission properties and functions for modifying emission proporeties throughout a simulation are defined in this module.

class Emission(*flux=()*, *reparable=True*, *site_index=()*, *comp_index=()*, *start_time=0*, *end_time=inf*, *repair_cost=()*, *emission_id=None*)

Bases: `object`

Stores all properties of all emissions that exist at a particular instant in a simulation.

Parameters

- **flux** – An array of emission rates (array of floats–gram/second)
- **reparable** – An array of True/False values to indicate whether or not an emission is reparable
- **site_index** – An array indicating the index of the site that contains every emission
- **comp_index** – An array indicating the index of the component that is the source of each emission
- **start_time** – An array specifying the time when every emission begins
- **end_time** – An array specifying the time when every emission will end (days)
- **emission_id** –
- **repair_cost** – An array storing the cost of repairing every emission ($)

em_rate_in_range(*t0*, *t1*, *reparable=None*)

Returns the sum of emissions that existed between t0 and t1 integrated over the time period :param t0: beginning of interval (days) :param t1: end of interval (days) :param reparable: boolean condition. If set, only returns emissions with a matching reparable property :return: Average emission rate between t1 and t0 (g/s)

extend(**args*)

Extends the existing emissions data frame with all of the entries in args :param args: a list of Emission objects :return:

get_current_emissions(*time*)

Returns all emissions that exist at time.current_time :param time: a Time object :return: a DataFrame of current emissions

get_emissions_in_range(*t0*, *t1*, *reparable=None*)

Returns all emissions that existed between t0 and t1 :param t0: beginning of interval (days) :param t1: end of interval (days) :param reparable: boolean condition. If set, only returns emissions with a matching reparable property :return: a DataFrame of all emissions that existed at any time in the interval t0:t1

bootstrap_emission_maker(*n_em_in*, *comp_name*, *site*, *time*, *start_time=None*, *reparable=True*)

Create leaks using a bootstrap method.

Parameters

- **n_em_in** – number of leaks to generate
- **comp_name** – key to a Component object in site.comp_dict
- **site** – a Site object
- **time** – a Time object

- **start_time** – the times at which each emission begins
- **reparable** – Specifies whether emissions should be reparable or not (boolean)

comp_indexes_fcn (*site*, *comp_name*, *n_inds*)
Returns an array of indexes to associate with new emissions

Parameters

- **site** – a EmissionSimModules.simulation_classes.Site object
- **comp_name** – name of a component contained in Site.comp_dict
- **n_inds** – Integer of indexes to generate

Returns An array of indexes in the range specified for the relevant component

emission_objects_generator (*dist_type*, *emission_data_path*, *custom_emission_maker=None*)
emission_objects_generator is a parent function that will be called to initialize gas fields

Parameters

- **dist_type** – Type of leak distribution to be used
- **leak_data_path** – Path to a leak data file

permitted_emission (*n_emit*, *sizes*, *duration*, *time*, *site*, *comp_name*, *start_time*)
Creates an emission object specifying new permitted emissions

Parameters

- **n_emit** – number of emissions to create
- **sizes** – a list of leak sizes from which to specify the emission rate
- **duration** – a float defining the duration of the emission
- **time** – a Time object
- **site** – a Site object
- **comp_name** – Name of the component to be considered from within site.comp_dict
- **start_times** – array of times at which emissions start

Returns an Emission object

1.4.2 infrastructure_classes

This module stores component, gasfield and site classes to represent infrastructure in a simulation

class Component (*repair_cost_path=None, emission_data_path=None, base_reparable=None, custom_emission_maker=None, emission_production_rate=0, emission_per_comp=None, episodic_emission_sizes=[0], episodic_emission_per_day=0, episodic_emission_duration=0, vent_sizes=[0], vent_period=inf, vent_starts=array([], dtype=float64), vent_duration=0, name='default', null_repair_rate=None, dist_type='bootstrap'*)
Bases: `object`

A class to store parameters defining a component (for example, name, leak production rate, leak size distribution, etc)

Parameters

- **repair_cost_path** – path to a repair cost data file

- **emission_data_path** – path to an emission data file
- **base_reparable** – Defines whether emissions generated are reparable with a boolean true/false
- **custom_emission_maker** – Optional custom defined function for creating new emissions
- **emission_production_rate** – The rate at which new emissions are created (emissions per day per component)
- **emission_per_comp** – The number of emissions expected per component (must be less than 1) If emission_per_comp is left as None, then emission_per_comp is set equal to the emissions per component recorded in the file at emission_data_path.
- **episodic_emission_sizes** – A list of emission sizes to draw from for episodic emissions (g/s)
- **episodic_emission_per_day** – The average frequency at which episodic emissions occur (1/days)
- **episodic_emission_duration** – The duration of episodic emissions (days)
- **vent_sizes** – A list of emission sizes for periodic emissions (g/s)
- **vent_period** – The time between emissions (days)
- **vent_duration** – the time that a periodic vent persits (days)
- **vent_starts** – the time at which the first periodic vent occurs at each component in the simulation
- **name** – A name for the instance of Component
- **null_repair_rate** – the rate at which fugitive emissions are repaired. If None, a steady state assumption is enforced based on emission_production_rate and emission_per_comp.
- **dist_type** – The type of distribution to be used in determining emission rates for new emissions

class GasField(*time=None*, *sites=None*, *emissions=None*, *met_data_path=None*)

Bases: `object`

GasField accommodates all data that defines a gas field at the beginning of a simulation.

Parameters

- **time** – A FEAST time object
- **sites** – a dict of sites like this: {'name': {'number': n_sites, 'parameters': site_object}}
- **emissions** – A FEAST emission object to be used during the simulations
- **met_data_path** – A path to a met data file

emerging_emissions(*time*)

Defines emissions that emerge during a simulation :param time: :return:

static emission_maker(*n_leaks*, *new_leaks*, *comp_name*, *n_comp*, *time*, *site*, *n_episodic=None*)

Updates an Emission object with new values returned by emission_size_maker and assigns unique indexes to them

Parameters

- **n_leaks** – number of new leaks to create
- **new_leaks** – a leak object to extend

- **comp_name** – name of a component object included in site.comp_dict
- **n_comp** – the number of components to model
- **time** – a time object
- **site** – a site object
- **n_episodic** – number of episodic emissions to create
- **start_time** – time at which the new emissions begin emitting

Returns None

emission_size_maker (*time*)
Creates a new set of leaks based on attributes of the gas field :param time: a time object (the parameter delta_t is used) :return new_leaks: the new leak object

get_met (*time*, *parameter_names*, *interp_modes='mean'*, *ophrs=None*)
Return the relevant meteorological condition, accounting for discrepancies between simulation time resolution and data time resolution

Parameters

- **time** – time object
- **parameter_names** – specify a list of meteorological conditions to return
- **interp_modes** – can be a list of strings: mean, median, max or min
- **ophrs** – Hours to consider when interpolating met data should be of form {'begin': 5, 'end':17}

Return met_conds dict of meteorological conditions

initialize_emissions (*time*)
Create emissions that exist at the beginning of the simulation

Parameters time –

Return initial_emissions

met_data_maker (*start_hr=0*)
Creates a dict to store met data derived from a Typical Meteorological Year file. The data may be rotated so that the simulation begins at any hour in the TMY file. :param start_hr: The hour at which the simulation should begin. :return: None

set_indexes ()
Counts components for each site and assigns appropriate indexes

class Site (*name='default'*, *comp_dict=None*, *prod_dat=None*)
Bases: `object`

A class to store the number and type of components associated with a site.

Parameters

- **name** – The name of the site object (a string)
- **comp_dict** – A dict of components at the site, for example: {'name': {'number': 650, 'parameters': Component()}}
- **prod_dat** –

1.4.3 result_classes

result_classes defines classes that are used to store event counts and continuous variable data for saving.

class ResultAggregate(*units=None*, *time_value=None*)

Bases: `object`

A super class designed to store aggregate results during a simulation. Time and value pairs are stored in a list.

append_entry(*time_value*)

Add a new entry to the ResultAggregate object

Parameters **time_value** – an ordered pair following this pattern: [time, value]

Returns None

get_vals(*t_start=0*, *t_end=inf*)

Returns all values associated with times between t_start and t_end.

Parameters

- **t_start** – Time to begin the sum
- **t_end** – time to end the sum

Returns All values associated with time

class ResultContinuous(***kwargs*)

Bases: *feast.EmissionSimModules.result_classes.ResultAggregate*

Designed to store continuous rates that endure between consecutive time recordings as opposed to discrete variables that occur at a specific time. For example, emission rate can be recorded as a continuous data type.

get_time_integrated(*start_time=0*, *end_time=None*, *unit_factor=1*)

Calculates the integral of value over the time period start_time:end_time

Parameters

- **start_time** – Beginning of the integration period
- **end_time** – End of the integration period
- **unit_factor** – A factor that may be used to ensure that the units of value are consistent with the units of time. For example, if time is measured in days and emissions are measured in g/s, a conversion factor of 3600 * 24 should be used to convert gram/second*days to grams.

Returns The integrated value

class ResultDiscrete(***kwargs*)

Bases: *feast.EmissionSimModules.result_classes.ResultAggregate*

Designed to store discrete values associated with specific times, as opposed to continuous rates that persist between consecutive data points. For example, the number of sites surveyed can be recorded as a discrete data type.

get_cumulative_vals(*t_start=0*, *t_end=inf*)

Returns a cumulative sum of the attribute "value"

Parameters

- **t_start** – Time to begin the cumulative sum
- **t_end** – time to end the cumulative sum

Returns Array of times in between t_start and t_end, cumulative sum of the attribute "value"

get_sum_val (*t_start=0*, *t_end=inf*)
Returns the sum of values between t_start and t_end

Parameters

- **t_start** – Time to begin the sum
- **t_end** – time to end the sum

Returns sum of values between t_start and t_end

1.4.4 simulation_classes

simulation_classes stores the classes used to represent time, results and financial settings in simulations.

class Scenario (*time*, *gas_field*, *ldar_program_dict*)
Bases: `object`

A class to store all data specifying a scenario and the methods to run and save a realization

Parameters

- **time** – Time object
- **gas_field** – GasField object
- **ldar_program_dict** – dict of detection methods and associated data

check_timestep ()
Prints a warning if time.delta_t is greater than the duration of some permitted emissions

Parameters

- **gas_field** – a GasField object
- **time** – a Time object

Returns None

run (*dir_out='Results'*, *display_status=True*, *save_method='json'*)
run generates a single realization of a scenario.

Parameters

- **dir_out** – path to a directory in which to save results (string)
- **display_status** – if True, display a status update whenever 10% of the time steps are completed

Returns None

save (*dir_out*, *method='json'*)
Save results to a file

Parameters

- **dir_out** – Name of directory in which to save output file.
- **method** – Specifies how results should be saved

class Time (*delta_t=1*, *end_time=365*, *current_time=0*)
Bases: `object`

Instances of the time class store all time related information during a simulation

Parameters

- **delta_t** – length of one timestep (days)
- **end_time** – length of the simulation (days)
- **current_time** – current time in a simulation (days)

PYTHON MODULE INDEX

f