InsideOpt Seeker$^{(\mathrm{TM})}$ 1.0.0 Reference Manual

October 2023

Contents

1 Introduction

The Seeker$^{(TM)}$ Library consists of two main classes, Env and Term, plus some few helper classes. This reference manual lists the functions provided by theses classes, as well as some helper classes. For more detailed background on these functions, as well as examples that illustrate their use, please refer to the Seeker$^{(TM)}$ User's Manual. The purpose of this Reference Manual is to provide a quick reference guide to look up functions quickly when developing a new Seeker$^{(TM)}$ optimization model.

2 The Environment Class

2.1 Environment Creation and Termination

- **Env(string)**: Constructor. Creates a deterministic Seeker$^{(TM)}$ environment, provided a valid license file name is given.
- **Env(string, bool stochastic)**: Constructor. Creates a stochastic Seeker$^{(TM)}$ environment, provided "stochastic=true" and a valid license file name is given.
- **Env::Env(string license, int processID, int runID):** Creates the Seeker$^{(TM)}$ environment which will coordinate with other processes where the environment was created using the exact same unique "runID" integer value. The processID numbers range from 0 to however many parallel runs your license allows. Use 0 for your first parallel process, 1 for your second, 2 for your third, and so on. **Note: It is important to start at 0 and count up the processIDs for this process to work properly. Do not run two different optimization processes for different models using the same runID!**
- **Env::Env(string license, int processID, int runID, bool stochastic):** Creates a stochastic Seeker$^{(TM)}$ environment which will coordinate with other processes where the environment was created using the exact same unique "runID" integer value. The processID numbers range from 0 to however many parallel runs your license allows. Use 0 for your first parallel process, 1 for your second, 2 for your third, and so on. **Note: It is important to start at 0 and count up the processIDs for this process to work properly. Do not run two different optimization processes for different models using the same runID!**
- **void end(void)**: Must be called before terminating the environment to avoid memory leaks.
- ∼**Env(void)**: Destructor. Terminates the Seeker$^{(TM)}$ environment.

2.2 Basic Decision Variables

- **Term continuous(double l, double h)**: Creates a variable that can take any continuous floating point value within the interval $[l, h]$. At the beginning of the optimization, the variable is initialized with an unspecified value in the same interval.
- **Term continuous(double l, double h, double v)**: Creates a variable that can take any continuous floating point value within the interval $[l, h]$. At the beginning of the optimization, the variable is initialized with value v.
- **Term ordinal(double l, double h)**: Creates a variable that can take any integer value within the interval $[l, h]$. At the beginning of the optimization, the variable is initialized with an unspecified integer value in the same interval.
- **Term ordinal(double l, double h, double v)**: Creates a variable that can take any integer value within the interval $[l, h]$. At the beginning of the optimization, the variable is initialized with value round(v), where the function round returns the nearest integer value within the allowed interval.
- **Term categorical(double l, double h)**: Creates a variable that can take any integer value within the interval $[l, h]$. At the beginning of the optimization, the variable is initialized with an unspecified value in the same interval. The difference to the same "Ordinal" variable is how the variable is handled within the optimization.
- **Term categorical(double l, double h, double v)**: Creates a variable that can take any integer value within the interval $[l, h]$. At the beginning of the optimization, the variable is initialized with value round(v), where the function round returns the nearest integer value within the allowed interval. The difference to the same "Ordinal" variable is how the variable is handled within the optimization.
- **Term categorical(double l, double h, vector<int> allowed)**: Creates a variable that can take any integer value within the interval $[l, h]$ that is listed in "allowed". At the beginning of the optimization, the variable is initialized with an unspecified value in "allowed".
- **Term categorical(double l, double h, vector<int> allowed, double v)**: Creates a variable that can take any integer value within the interval $[l, h]$ that is listed in "allowed". At the beginning of the optimization, the variable is initialized with value round(v), where the function round returns the nearest integer value within the allowed interval. This value is required to be listed in "allowed."

2.3 Meta Decision Variables

- **Partition partition(int numberOfPartitions, int numberOfItems)**: Creates a partitioning that distributes items numbered from 0 to *numberOfItems*-1 into *numberOfPartitions* partitions.
- **Partition packing(int numberOfSets, int numberOfItems)**:Creates a packing that distributes items numbered from 0 to *numberOfItems*-1 over *numberOfSets* sets, whereby some items may not be assigned to any set.
- **Term boolean(Partition part, int item, int index)**: Returns a term that is true if and only if item *item* is assigned to the set/partition with index *index*.
- **vector<Term> convex_combination(int n):** Returns a vector of continuous 0 to 1 decision variables that will always sum to 1.
- **Permutation permutation(int n):** Creates a Permutation object from which two different sets of ordinal decision variables, which each form a permutation.

2.3.1 The Permutation Class

- **vector<Term> Permutation::get_permutation(void):** Returns the values of n integer variables in $[0, \ldots, n-1]$ which are all different, thereby forming a permutation.
- **vector<Term> Permutation::get_permutation_inverse(void):** Returns the values of n integer variables in $[0, \ldots, n-1]$ which are all different, thereby forming a permutation, which is the inverse of the permutation provided by "get_permutation".

2.4 Unary Operators

- **Term abs(Term a)**: Returns the value of a if $a \geq 0$ and $-a$ otherwise.
- **Term sqr(Term a)**: Returns a^2.
- **Term sqrt(Term a)**: Returns $\sqrt{a}$ if $a \geq 0$. Returns "undefined" otherwise.
- **Term exp(Term a)**: Returns e^a, where e is the Euler number.
- **Term log(Term a)**: Returns the natural logarithm, $\log(a)$ if $a > 0$. Returns "undefined" otherwise.
- **Term min_0(Term a)**: Returns the value of a if $a < 0$, and 0 otherwise.
- **Term max_0(Term a)**: Returns the value of a if $a > 0$, and 0 otherwise.
- **Term sin(Term a)**: Returns the sine of a in radians: $\sin(a)$.
- **Term cos(Term a)**: Returns the cosine of a in radiance: $\cos(a)$.
- **Term ceil(Term a)**: Returns the smallest possible integer value which is greater than or equal to the value of a.
- **Term floor(Term a)**: Returns the largest possible integer value which is less than or equal to the value of a.
- **Term round(Term a)**: Returns the integral value that is nearest to the value of a, with halfway cases rounded away from zero.
- **Term trunc(Term a)**: Rounds the value of a towards zero and returns the nearest integral value that is not larger in magnitude than a.

2.5 Binary Operators

- **Term abs(Term a, Term b)**: Returns $|a - b|$.
- **Term eucl(Term a, Term b)**: Returns $(a - b)^2$.
- **Term power_round_exp(Term a, Term b)**: Returns $a^{\bar{b}}$ if $a \neq 0$, where $\bar{b} = \text{round}(b)$. Returns 1 if $a = \bar{b} = 0$, and 0 if $a = 0$ and $\bar{b} \neq 0$.
- **Term power(Term a, Term b)**: Returns a^b if $a > 0$, and "undefined" if $a < 0$. Returns 1 if $a = b = 0$, and 0 if $a = 0$ and $b \neq 0$.
- **Term round_div(Term a, Term b)**: If $\bar{b} \neq 0$, the operator returns the nearest integer that is not larger in magnitude than $\bar{a}/\bar{b}$ (rounding towards zero), whereby $\bar{b} = \text{round}(b)$. Returns "undefined" if $\bar{b} = 0$.
- **Term div(Term a, Term b)**: If $b \neq 0$, the operator returns the nearest integer that is not larger in magnitude than a/b (rounding towards zero). Returns "undefined" if $b = 0$.
- **Term round_mod(Term a, Term b)**: If $\bar{b} \neq 0$, the operator returns $\bar{a} - (\bar{a} \text{ div } \bar{b}) * \bar{b}$, whereby $\bar{x} = \text{round}(x)$. If $\bar{b} = 0$, the operator returns "undefined."
- **Term mod(Term a, Term b)**: Undefined, use "Term Term::operator%(Term a, Term b)."

2.6 Indexing

- **template <class T>**
  Term index(Term ind, const vector<T>& terms): For any integer value that "ind" takes in the set $\{0,\ldots,\text{terms.size()}-1\}$, the operator returns the value of "terms[ind]." The return value is "undefined" if the value of "ind" does not fall into this set. The template type "T" must match either Term, bool, int, long, float, or double.

- **template <class T>**
  Term index(Term ind1, Term ind2, const vector<vector<T>>& terms): For any integer value that "ind1" takes in the set $\{0,\ldots,\text{terms.size()}-1\}$, and for any integer value that "ind2" takes in the set $\{0,\ldots,\text{terms[ind1].size()}-1\}$, the operator returns the value of "terms[ind1][ind2]." The return value is "undefined" if any of the values of "ind1" and "ind2" do not fall into the respective sets. The template type "T" must match either Term, bool, int, long, float, or double.

Note: Please check carefully that the index term or terms only take feasible values that fit the dimensions of the vector or matrix provided. Seeker$^{\text{(TM)}}$ will exit if this is not the case and the result of this term is material for the objective or the constraint status.

2.7 Conditioning

- **Term if_ (Term condTerm, Term thenTerm, Term elseTerm)**: Returns the value of ”thenTerm” if ”condTerm” evaluates to non-Zero, and the value of ”elseTerm” otherwise.
- **template <class T> Term if_ (Term condTerm, Term thenTerm, T elseTerm)**: Returns the value of ”thenTerm” if ”condTerm” evaluates to non-Zero, and ”elseTerm” of type ”T” otherwise, whereby the type ”T” is either bool, int, long, float, or double.
- **template <class T> Term if_ (Term condTerm, T thenTerm, Term elseTerm)**: Returns the value of ”thenTerm” of type ”T” if ”condTerm” evaluates to non-Zero, and the value of ”elseTerm” otherwise, whereby the type ”T” is either bool, int, long, float, or double.
- **template <class T, class S> Term if_ (Term condTerm, T thenTerm, S elseTerm)**: Returns the value of ”thenTerm” of type ”T” if ”condTerm” evaluates to non-Zero, and ”elseTerm” of type ”S” otherwise, whereby the types ”T” and ”S” are either bool, int, long, float, or double.

2.8 Aggregation

2.8.1 Sum

- **Term sum(vector<Term> terms)**: Returns the sum of the values in "terms."
- **Term sum_if(vector<Term> terms, Partition part, int i)**: Returns the sum over all terms in *terms* whose index in the vector corresponds to the items in partition i in the partitioning *part*.
- **Term sum_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. The term returned will equal the sum of the terms 'terms' for which their corresponding switch condition matches the corresponding case.
- **Term sum_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** Returns the sum of those terms in 'terms' for which the corresponding SwitchCondition object matches its caseIndex'th case.
- **Term sum_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. The term returned will equal the sum of the terms 'terms' for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]].
- **Term sum_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** Returns the sum of those terms in 'terms' for which the corresponding interval condition term falls into [low, high].

2.8.2 Product

- **Term prod(vector<Term> terms)**: Returns the product of the values in "terms."

- **Term prod_if(vector<Term> terms, Partition part, int i)**: Returns the product over all terms in *terms* whose index in the vector corresponds to the items in partition *i* in the partitioning *part*.

- **Term prod_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. The term returned will equal the product of the terms 'terms' for which their corresponding switch condition matches the corresponding case.

- **Term prod_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** Returns the product of those terms in 'terms' for which the corresponding SwitchCondition object matches its caseIndex'th case.

- **Term prod_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. The term returned will equal the product of the terms 'terms' for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]].

- **Term prod_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** Returns the product of those terms in 'terms' for which the corresponding interval condition term falls into [low, high].

2.8.3 Maximum

- **Term max(vector<Term> terms)**: Returns the maximum of the values in "terms."

- **Term max_if(vector<Term> terms, Partition part, int i)**: Returns the maximum over all terms in *terms* whose index in the vector corresponds to the items in partition *i* in the partitioning *part*.

- **Term max_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. The term returned will equal the maximum of the terms 'terms' for which their corresponding switch condition matches the corresponding case. The value is -1e20 in case the list of terms is empty.

- **Term max_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** Returns the maximum of those terms in 'terms' for which the corresponding SwitchCondition object matches its caseIndex'th case. The value is -1e20 in case the list of terms is empty.

- **Term max_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. The term returned will equal the maximum of the terms 'terms' for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]]. The value is -1e20 in case the list of terms is empty.

- **Term max_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** Returns the maximum of those terms in 'terms' for which the corresponding interval condition term falls into [low, high]. The value is -1e20 in case the list of terms is empty.

2.8.4 Minimum

- **Term min(vector<Term> terms)**: Returns the minimum of the values in "terms."

- **Term min_if(vector<Term> terms, Partition part, int i)**: Returns the minimum over all terms in *terms* whose index in the vector corresponds to the items in partition i in the partitioning *part*.

- **Term min_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. The term returned will equal the minimum of the terms 'terms' for which their corresponding switch condition matches the corresponding case. The value is +1e20 in case the list of terms is empty.

- **Term min_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** Returns the minimum of those terms in 'terms' for which the corresponding SwitchCondition object matches its caseIndex'th case. The value is +1e20 in case the list of terms is empty.

- **Term min_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. The term returned will equal the minimum of the terms 'terms' for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]]. The value is +1e20 in case the list of terms is empty.

- **Term min_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** Returns the minimum of those terms in 'terms' for which the corresponding interval condition term falls into [low, high]. The value is +1e20 in case the list of terms is empty.

2.8.5 ArgMax

- **Term argmax(vector<Term> terms):** Returns the position $\{0, 1, \ldots, n-1\}$ of a term in the given vector of n terms which takes a value lower or equal to all other terms. If more than one term determines the maximum the position returned may correspond to any one of them. The value is "undefined" in case the list of terms is empty.

- **Term argmax_if(vector<Term> terms, Partition part, int i)**: Returns the position $\{0, 1, \ldots, n-1\}$ of a term in the given vector of n terms which takes a value lower or equal to all other terms in *terms* whose index in the vector corresponds to the items in partition i in the partitioning *part*. The value is -1 in case the list of terms is empty.

- **Term argmax_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. The term returned will equal the index of the maximum of the terms 'terms' for which their corresponding switch condition matches the corresponding case. The value is -1 in case the list of terms is empty.

- **Term argmax_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** Returns the index of the maximum of those terms in 'terms' for which the corresponding SwitchCondition object matches its caseIndex'th case. The value is -1 in case the list of terms is empty.

- **Term argmax_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. The term returned will equal the index of the maximum of the terms 'terms' for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]]. The value is -1 in case the list of terms is empty.

- **Term argmax_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** Returns the index of the maximum of those terms in 'terms' for which the corresponding interval condition term falls into [low, high]. The value is -1 in case the list of terms is empty.

2.8.6 ArgMin

- **Term argmin(vector<Term> terms):** Returns the position $\{0, 1, \ldots, n-1\}$ of a term in the given vector of n terms which takes a value lower or equal to all other terms. If more than one term determines the minimum the position returned may correspond to any one of them. The value is "undefined" in case the list of terms is empty.

- **Term argmin_if(vector<Term> terms, Partition part, int i)**: Returns the position $\{0, 1, \ldots, n-1\}$ of a term in the given vector of n terms which takes a value greater or equal to all other terms in *terms* whose index in the vector corresponds to the items in partition i in the partitioning *part*. The value is -1 in case the list of terms is empty.

- **Term argmin_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. The term returned will equal the index of the minimum of the terms 'terms' for which their corresponding switch condition matches the corresponding case. The value is -1 in case the list of terms is empty.

- **Term argmin_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** Returns the index of the minimum of those terms in 'terms' for which the corresponding SwitchCondition object matches its caseIndex'th case. The value is -1 in case the list of terms is empty.

- **Term argmin_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. The term returned will equal the index of the minimum of the terms 'terms' for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]]. The value is -1 in case the list of terms is empty.

- **Term argmin_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** Returns the index of the minimum of those terms in 'terms' for which the corresponding interval condition term falls into [low, high]. The value is -1 in case the list of terms is empty.

2.8.7 And

- **Term and_ (vector<Term> terms):** Returns 0 if any of the values in "terms" evaluates to 0, and 1 otherwise.

- **Term and_if(vector<Term> terms, Partition part, int i)**: Returns 0 if any of the values in "terms" evaluates to 0 whose index in the vector corresponds to the items in partition i in the partitioning *part*. The value is "1" in case the list of terms is empty.

- **Term and_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. Returns 0 if any of the values in "terms" evaluates to 0 for which their corresponding switch condition matches the corresponding case, and 1 otherwise. The value is 1 in case the list of terms is empty.

- **Term and_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** All vectors **must** have the exact same length. Returns 0 if any of the values in "terms" evaluates to 0 for which the corresponding SwitchCondition object matches its caseIndex'th case, and 1 otherwise. The value is 1 in case the list of terms is empty.

- **Term and_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. Returns 0 if any of the values in "terms" evaluates to 0 for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]], and 1 otherwise. The value is 1 in case the list of terms is empty.

- **Term and_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** All vectors **must** have the exact same length. Returns 0 if any of the values in "terms" evaluates to 0 for which their corresponding interval condition term falls into the corresponding interval [low, high], and 1 otherwise. The value is 1 in case the list of terms is empty.

2.8.8 Or

- **Term or_ (vector<Term> terms):** Returns 1 if any of the values in "terms" does not evaluate to 0, and 0 otherwise.

- **Term or_if(vector<Term> terms, Partition part, int i)**: Returns 1 if any of the values in "terms" does not evaluate to 0 whose index in the vector corresponds to the items in partition *i* in the partitioning *part*, and 0 otherwise. The value is "0" in case the list of terms is empty.

- **Term or_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. Returns 1 if any of the values in "terms" does not evaluate to 0 for which their corresponding switch condition matches the corresponding case, and 0 otherwise. The value is 0 in case the list of terms is empty.

- **Term or_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** All vectors **must** have the exact same length. Returns 1 if any of the values in "terms" does not evaluate to 0 for which the corresponding SwitchCondition object matches its caseIndex'th case, and 0 otherwise. The value is 0 in case the list of terms is empty.

- **Term or_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. Returns 1 if any of the values in "terms" does not evaluate to 0 for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]], and 0 otherwise. The value is 0 in case the list of terms is empty.

- **Term or_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** All vectors **must** have the exact same length. Returns 1 if any of the values in "terms" does not evaluate to 0 for which their corresponding interval condition term falls into the corresponding interval [low, high], and 0 otherwise. The value is 0 in case the list of terms is empty.

2.8.9 Arithmetic Mean

- **Term mean(vector<Term> terms):** Returns the sum of the values in "terms" divided by their number.
- **Term mean_if(vector<Term> terms, Partition part, int i)**: Returns the arithmetic mean of the values in "terms" whose index in the vector corresponds to the items in partition i in the partitioning *part*, and 0 otherwise. The value is "0" in case the list of terms is empty.
- **Term mean_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. Returns the arithmetic mean of the values in "terms" for which their corresponding switch condition matches the corresponding case, and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term mean_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** All vectors **must** have the exact same length. Returns the arithmetic mean of the values in "terms" for which the corresponding SwitchCondition object matches its caseIndex'th case, and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term mean_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. Returns the arithmetic mean of the values in "terms" for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]], and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term mean_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** All vectors **must** have the exact same length. Returns the arithmetic mean of the values in "terms" for which their corresponding interval condition term falls into the corresponding interval [low, high], and 0 otherwise. The value is 0 in case the list of terms is empty.

2.8.10 Variance

- **Term variance(vector<Term> terms):** Returns the variance of the values in "terms." Note that this operator returns the (biased) value of the canonical definition of the variance, i.e., $\frac{\sum_i (\text{terms}[i]-\mu)^2}{n}$, whereby μ is the mean of the values in "terms."

- **Term variance_if(vector<Term> terms, Partition part, int i)**: Returns the biased variance estimate of the values in "terms" whose index in the vector corresponds to the items in partition i in the partitioning *part*, and 0 otherwise. The value is "0" in case the list of terms is empty.

- **Term variance_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. Returns the biased variance estimate of the values in "terms" for which their corresponding switch condition matches the corresponding case, and 0 otherwise. The value is 0 in case the list of terms is empty.

- **Term variance_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** All vectors **must** have the exact same length. Returns the biased variance estimate of the values in "terms" for which the corresponding SwitchCondition object matches its caseIndex'th case, and 0 otherwise. The value is 0 in case the list of terms is empty.

- **Term variance_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. Returns the biased variance estimate of the values in "terms" for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]], and 0 otherwise. The value is 0 in case the list of terms is empty.

- **Term variance_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** All vectors **must** have the exact same length. Returns the biased variance estimate of the values in "terms" for which their corresponding interval condition term falls into the corresponding interval [low, high], and 0 otherwise. The value is 0 in case the list of terms is empty.

2.8.11 Unbiased Variance

- **Term sample_variance(vector<Term> terms):** Returns the unbiased variance estimator for the values in "terms." Note that this operator returns the unbiased value of of the variance, i.e., $\frac{\sum_i (\text{terms}[i]-\mu)^2}{(n-1)}$, whereby μ is the mean of the values in "terms." The list of terms must contain at least two terms.

- **Term sample_variance_if(vector<Term> terms, Partition part, int i)**: Returns the unbiased variance estimator for the values in "terms" whose index in the vector corresponds to the items in partition i in the partitioning *part*, and 0 otherwise. The value is "0" in case the list of active terms contains less than two entries.

- **Term sample_variance_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. Returns the unbiased variance estimator for the values in "terms" for which their corresponding switch condition matches the corresponding case, and 0 otherwise. The value is "0" in case the list of active terms contains less than two entries.

- **Term sample_variance_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** All vectors **must** have the exact same length. Returns the unbiased variance estimator for the values in "terms" for which the corresponding SwitchCondition object matches its caseIndex'th case, and 0 otherwise. The value is "0" in case the list of active terms contains less than two entries.

- **Term sample_variance_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. Returns the unbiased variance estimator for the values in "terms" for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]], and 0 otherwise. The value is "0" in case the list of active terms contains less than two entries.

- **Term sample_variance_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** All vectors **must** have the exact same length. Returns the unbiased variance estimator for the values in "terms" for which their corresponding interval condition term falls into the corresponding interval [low, high], and 0 otherwise. The value is "0" in case the list of active terms contains less than two entries.

2.8.12 Standard Deviation

- **Term standard_deviation(vector<Term> terms)**: Returns the square root of the biased variance estimate of the values in "terms."
- **Term standard_deviation_if(vector<Term> terms, Partition part, int i)**: Returns the root of the biased variance estimate of the values in "terms" whose index in the vector corresponds to the items in partition i in the partitioning *part*, and 0 otherwise. The value is "0" in case the list of terms is empty.
- **Term standard_deviation_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. Returns the root of the biased variance estimate of the values in "terms" for which their corresponding switch condition matches the corresponding case, and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term standard_deviation_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** All vectors **must** have the exact same length. Returns the root of the biased variance estimate of the values in "terms" for which the corresponding SwitchCondition object matches its caseIndex'th case, and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term standard_deviation_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. Returns the root of the biased variance estimate of the values in "terms" for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]], and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term standard_deviation_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** All vectors **must** have the exact same length. Returns the root of the biased variance estimate of the values in "terms" for which their corresponding interval condition term falls into the corresponding interval [low, high], and 0 otherwise. The value is 0 in case the list of terms is empty.

2.8.13 Norm 1

- **Term norm1(vector<Term> terms)**: Returns the sum of absolute values in "terms."
- **Term norm1_if(vector<Term> terms, Partition part, int i)**: Returns the sum of absolute values in "terms" whose index in the vector corresponds to the items in partition i in the partitioning *part*, and 0 otherwise. The value is "0" in case the list of terms is empty.
- **Term norm1_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. Returns the sum of absolute values in "terms" for which their corresponding switch condition matches the corresponding case, and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term norm1_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** All vectors **must** have the exact same length. Returns the sum of absolute values in "terms" for which the corresponding SwitchCondition object matches its caseIndex'th case, and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term norm1_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. Returns the sum of absolute values in "terms" for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]], and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term norm1_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** All vectors **must** have the exact same length. Returns the sum of absolute values in "terms" for which their corresponding interval condition term falls into the corresponding interval [low, high], and 0 otherwise. The value is 0 in case the list of terms is empty.

2.8.14 Norm 2

- **Term norm2(vector<Term> terms)**: Returns the square root of the sum of square values in ”terms.”
- **Term norm2_if(vector<Term> terms, Partition part, int i)**: Returns the square root of the sum of square values in ”terms” whose index in the vector corresponds to the items in partition i in the partitioning *part*, and 0 otherwise. The value is ”0” in case the list of terms is empty.
- **Term norm2_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in ’cases’ must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. Returns the square root of the sum of square values in ”terms” for which their corresponding switch condition matches the corresponding case, and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term norm2_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** All vectors **must** have the exact same length. Returns the square root of the sum of square values in ”terms” for which the corresponding SwitchCondition object matches its caseIndex’th case, and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term norm2_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. Returns the square root of the sum of square values in ”terms” for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]], and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term norm2_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** All vectors **must** have the exact same length. Returns the square root of the sum of square values in ”terms” for which their corresponding interval condition term falls into the corresponding interval [low, high], and 0 otherwise. The value is 0 in case the list of terms is empty.

2.8.15 Geometric Mean

- **Term geometric_mean(vector<Term> terms)**: Returns the nth root of the product of the n terms in "terms." It is the user's responsibility to make sure the values of the terms aggregated are all non-negative.

- **Term geometric_mean_if(vector<Term> terms, Partition part, int i)**: Returns the geometric mean of the values in "terms" whose index in the vector corresponds to the items in partition i in the partitioning *part*, and 0 otherwise. The value is "1" in case the list of terms is empty.

- **Term geometric_mean_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. Returns the geometric mean of the values in "terms" for which their corresponding switch condition matches the corresponding case, and 0 otherwise. The value is 1 in case the list of terms is empty.

- **Term geometric_mean_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** All vectors **must** have the exact same length. Returns the geometric mean of the values in "terms" for which the corresponding SwitchCondition object matches its caseIndex'th case, and 0 otherwise. The value is 1 in case the list of terms is empty.

- **Term geometric_mean_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. Returns the geometric mean of the values in "terms" for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]], and 0 otherwise. The value is 1 in case the list of terms is empty.

- **Term geometric_mean_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** All vectors **must** have the exact same length. Returns the geometric mean of the values in "terms" for which their corresponding interval condition term falls into the corresponding interval [low, high], and 0 otherwise. The value is 1 in case the list of terms is empty.

2.8.16 Root Mean Squared Value

- **Term rmsv(vector<Term> terms)**: Returns the square root of the mean of the squared values in "terms."
- **Term rmsv_if(vector<Term> terms, Partition part, int i)**: Returns the square root of the mean of the squared values in "terms" whose index in the vector corresponds to the items in partition *i* in the partitioning *part*, and 0 otherwise. The value is "0" in case the list of terms is empty.
- **Term rmsv_if(vector<Term> terms, vector<SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. Returns the square root of the mean of the squared values in "terms" for which their corresponding switch condition matches the corresponding case, and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term rmsv_if(vector<Term> terms, vector<SwitchCondition> conditions, int caseIndex):** All vectors **must** have the exact same length. Returns the square root of the mean of the squared values in "terms" for which the corresponding SwitchCondition object matches its caseIndex'th case, and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term rmsv_if(vector<Term> terms, vector<IntervalCondition> conditions, vector<double> lows, vector<double> highs):** All vectors **must** have the exact same length. Returns the square root of the mean of the squared values in "terms" for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]], and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term rmsv_if(vector<Term> terms, vector<IntervalCondition> conditions, double low, double high):** All vectors **must** have the exact same length. Returns the square root of the mean of the squared values in "terms" for which their corresponding interval condition term falls into the corresponding interval [low, high], and 0 otherwise. The value is 0 in case the list of terms is empty.

2.8.17 Average Absolute Value

- **Term average_absolute_value(vector<Term> terms)**: Returns the mean of the absolute values in "terms."
- **Term average_absolute_value_if(vector<Term> terms, Partition part, int i)**: Returns the mean of the absolute values in "terms" whose index in the vector corresponds to the items in partition *i* in the partitioning *part*, and 0 otherwise. The value is "0" in case the list of terms is empty.
- **Term average_absolute_value_if(vector<Term> terms, vector< SwitchCondition> conditions, vector<int> cases):** All vectors **must** have the exact same length. The values provided in 'cases' must be non-negative integers no greater than the number of matches their respective SwitchCondition objects have been constructed with. Returns the mean of the absolute values in "terms" for which their corresponding switch condition matches the corresponding case, and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term average_absolute_value_if(vector<Term> terms, vector< SwitchCondition> conditions, int caseIndex):** All vectors **must** have the exact same length. Returns the mean of the absolute values in "terms" for which the corresponding SwitchCondition object matches its caseIndex'th case, and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term average_absolute_value_if(vector<Term> terms, vector< IntervalCondition> conditions, vector<double> lows, vector< double> highs):** All vectors **must** have the exact same length. Returns the mean of the absolute values in "terms" for which their corresponding interval condition term falls into the corresponding interval [lows[i], highs[i]], and 0 otherwise. The value is 0 in case the list of terms is empty.
- **Term average_absolute_value_if(vector<Term> terms, vector< IntervalCondition> conditions, double low, double high):** All vectors **must** have the exact same length. Returns the mean of the absolute values in "terms" for which their corresponding interval condition term falls into the corresponding interval [low, high], and 0 otherwise. The value is 0 in case the list of terms is empty.

2.9 Quantiles and Sorting

- **vector<Term> ith_smallest_value(Term i, vector<Term> terms):** Returns a vector with two terms. The first holds the *value* of the i'th smallest value in the list. The second gives the *position* of the i'th smallest value in the list. Note that the term 'i' can be variable but must always evaluate to an integer number in $\{0, 1, \ldots, n-1\}$ when n is the length of the vector 'terms.'

- **vector<Term> ith_smallest_value(int i, vector<Term> terms):** Returns a vector with two terms. The first holds the *value* of the i'th smallest value in the list. The second gives the *position* of the i'th smallest value in the list. Note that the term 'i' can be variable but must always evaluate to an integer number in $\{0, 1, \ldots, n-1\}$ when n is the length of the vector 'terms.'

- **vector<Term> ith_largest_value(Term i, vector<Term> terms):** Returns a vector with two terms. The first holds the *value* of the i'th largest value in the list. The second gives the *position* of the i'th largest value in the list. Note that the term 'i' can be variable but must always evaluate to an integer number in $\{0, 1, \ldots, n-1\}$ when n is the length of the vector 'terms.'

- **vector<Term> ith_largest_value(int i, vector<Term> terms):** Returns a vector with two terms. The first holds the *value* of the i'th largest value in the list. The second gives the *position* of the i'th largest value in the list. Note that the term 'i' can be variable but must always evaluate to an integer number in $\{0, 1, \ldots, n-1\}$ when n is the length of the vector 'terms.'

- **Sorting int_sorting(vector<Terms> terms):** Returns a Sorting object which will first round all values of the terms in the vector 'terms'.

- **Sorting float_sorting(vector<Terms> terms):** Returns a Sorting object which will sort the terms from smallest to largest.

- **Sorting partial_sorting(vector<Term> terms, int k, bool maximize):** Returns a Sorting object which will sort the top (if 'maximize' is true, otherwise: bottom) k values of the terms in the vector 'terms.' The order returned is from largest to smallest when 'maximize' is true, and from smallest to largest otherwise.

2.9.1 The Sorting Class

- **vector<Term> Sorting::get(void):** returns the vector of values sorted from smallest to largest, unless if the Sorting instance was created using the 'partial_sorting' environment method with the parameter 'maximize=true.'
- **vector<Term> Sorting::get_permutation(void):** returns the vector of positions in $\{0, 1, \ldots, n-1\}$ in the original vector 'terms' that was used to create the Sorting object.
- **vector<Term> Sorting::get_permutation_inverse(void):** returns the vector of positions in $\{0, 1, \ldots, n-1\}$ in the sorted list with respect to the original terms in the vector 'term' that was used to create the Sorting object

2.10 Conversion

- **template <class T>**
  Term convert(T data): Turns a Boolean, integer, long integer, float, or double into a Term.

- **template <class T>**
  vector<Term> convert(vector<T> data): Turns a vector of Booleans, integers, long integers, floats, or doubles into a vector of Terms.

- **template <class T>**
  vector<vector<Term> > convert(vector <vector<T> > data): Turns a matrix of Booleans, integers, long integers, floats, or doubles into a matrix of Terms.

2.11 Efficient Boolean Indicators

2.11.1 Switch Conditions

- **SwitchCondition switch_condition(Term cond, vector<Term> matches):** Returns a SwitchCondition object for the Term 'cond' for matching one or more terms provided in the vector 'matches.'

- **SwitchCondition switch_condition(Term cond, int n):** Returns a SwitchCondition object for the Term 'cond' for matching the values 0 to n-1.

- **Term boolean(SwitchCondition c, int caseIndex):** Returns a term that is true if, and only if, the conditional term underlying the SwitchCondition 'c' matches the value of the caseIndex'th matching term, whereby 'caseIndex' must take a value between 0 and matches.size()-1 or 0 and n-1, depending on which function was used to create SwitchCondition 'c'. In case one or more of the terms involved take floating point values, their values are considered identical if, and only if, their difference in value is at most 1e-7.

2.11.2 Interval Conditions

- **IntervalCondition interval_condition(Term cond):** Returns an IntervalCondition object for the Term 'cond.'
- **Term boolean(IntervalCondition c, double low, double high):** Returns a term that is true if, and only if, the conditional term underlying the IntervalCondition 'c' is in [low,high].

2.12 User-Defined Terms

- **Term Env::user_defined_term(std::vector<Term> terms, UserEvalTerm& userEval):** Creates a user-defined term. The values of the terms in ”terms” will be provided to the overloaded function ”double UserEvalTerm::evaluate(vector<double>)” which implements the user-defined function.

- **vector<Term> Env::user_defined_term(std::vector<Term> terms, UserEvalVector& userEval, int numberOfTargets):** Creates a user-defined vector function. The values of the terms in ”terms” will be provided to the overloaded function ”vector<double> UserEvalVector::evaluate(vector<double>)” which returns a vector of size ”numberOfTargets” and implements the user-defined function.

Note: When creating the objects of the overloaded classes UserEvalTerm and UserEvalVector, care must be taken that these objects persist until the Seeker$^{(TM)}$ environment is ended. Otherwise, Seeker$^{(TM)}$ will make calls to these objects which may result in runtime failures.

2.12.1 The UserEvalTerm Class

- **UserEvalTerm(void) :** Base class constructor.
- **virtual double evaluate(vector<double> values) = 0:** Must be overloaded to create a user defined term.
- **virtual double inc_evaluate(vector<int> valueIndices, vector< double> oldValues, vector<double> values):** Can be overloaded when creating a user defined term to allow for fast recomputation. Inputs are the indices of values that have changed, as they are indexed as input for "evaluate"; the old values of the input terms that have changed; the new values of the input terms that have changed their value since the last ca;; to "evaluate" or "inc_evaluate."
- **virtual bool incremental(void) const = 0:** Must be overloaded to create a user defined term. Must return a static "false" if "inc_evaluate" was not overloaded. Otherwise return a static "true" to enable fast re-computation.

Note: When having "incremental" return 'true' and implementing "inc_evaluate," each user-defined term that is created should be provided with its own class object!

2.12.2 The UserEvalVector Class

- **UserEvalVector(void) :** Base class constructor.
- **virtual vector<double> recompute(vector<double> values) = 0:** Must be overloaded to create a user defined vector.
- **virtual IncPair inc_recompute(vector<int> valueIndices, vector<double> oldValues, vector<double> values):** Can be overloaded when creating a user defined vector to allow for fast recomputation. Inputs are the indices of values that have changed, as they are indexed as input for "recompute"; the old values of the input terms that have changed; the new values of the input terms that have changed their value since the last call to "recompute" or "inc_recompute." The function must return a pair of two vectors. The first contains the list of indices of output values that have changed, indexed in the same order as returned by "recompute" while starting the numbering at 0. The second vector contains the list, of same length, of the new values for the respective outputs.
- **virtual bool incremental(void) const = 0:** Must be overloaded to create a user defined vector. Must return a static "false" if "inc_recompute" was not overloaded. Otherwise return a static "true" to enable fast recomputation.

Note: When having "incremental" return 'true' and implementing "inc_recompute," each user-defined vector that is created should be provided with its own class object!

2.12.3 The IncPair Type

- **typedef pair<vector<int>, vector<double>** > **IncPair:** An IncPair is a pair of vectors, the first of output indices that have changed, and the second the new values of the respective outputs.

2.13 Nested Linear Optimization

- **LP lp(vector<Term> objTerms, vector<Term> varBounds, vector<Term> rowBounds, vector<vector< Term>> matrix, bool maximize):** Given a vector with n terms that specify the objective function, a vector with $2n$ variable bounds, whereby two consecutive values specify first the lower bound and then the upper bound on each variable, a vector with $2m$ row bounds, where again the even entries give the lower, and the odd entries specify the upper bounds for the linear constraints, a matrix that consists of m times n entries (i.e., in row-wise representation), and a Boolean flag indicating whether the linear solver is to maximize or minimize the objective, Seeker$^{(\mathrm{TM})}$ returns an LP object from which various terms can be derived.

2.13.1 The LP Class

- **Term LP::get_solution_status(void):** This function should always be called. It returns a term that reflects the value of the optimization. The term is 1 in case the LP could be solved to optimality. The value is 0 if the LP has no feasible solution. The value is 2 if the LP is unbounded. The value is -1 in case the solver had a problem.
- **Term LP::get_objective(void):** In case the status returned above is 1, the term returned by this function reflects the optimal objective function value.
- **vector<Term> LP::get_solution(void):** In case the status returned above is 1, the vector of terms returned by this function give the values of an optimal solution.
- **vector<Term> LP::get_row_sums(void):** In case the status returned above is 1, the vector of terms returned by this function equal the product Ax, where A is the current matrix of the LP, and x is the optimal solution.
- **Term LP::get_dual_status(void):** This function should always be used before any of the functions below are utilized. It returns a term that reflects the status of the dual solution. Particularly, the term returned is true if, and only if, the LP solver was able to find a valid dual solution.
- **vector<Term> LP::get_row_duals(void):** In case the dual status returned above is true, the vector of terms returned by this function give the row duals.
- **vector<Term> LP::get_column_duals(void):** In case the dual status returned above is true, the vector of terms returned by this function give the column duals.

2.14 Canonical Distribution Terms

- **Term Env::continuous_uniform(double low, double high):** Returns a data point that takes a uniformly random value in the interval [low, high].
- **Term Env::discrete_uniform(double low, double high):** Returns a data point that takes a uniformly random integer value in the interval [low, high].
- **Term Env::continuous_exponential(double lambda, double low, double high):** Returns a data point that takes a non-negative value sampled according to the exponential probability distribution with density $f(x) = \lambda e^{-\lambda x}$ for $x \geq 0$ and $f(x) = 0$, otherwise. The parameter "lambda" must be positive. If the value sampled by the distribution falls out of the specified range [low, high], then the value returned will equal the closest number within that interval.
- **Term Env::discrete_exponential(double lambda, double low, double high):** Returns a data point that takes a non-negative integer value sampled according to the exponential probability distribution with density $f(x) = \lambda e^{-\lambda x}$ for $x \geq 0$ and $f(x) = 0$, otherwise. The parameter "lambda" must be positive. The value sampled by the distribution is first rounded down and then, should it fall out of the specified range [low, high], then the value returned will equal the closest integer number within that interval. Note: Since Seeker$^{(TM)}$ always rounds down the sampled continuous values, this function can also be understood as creating a random Term that follows the geometric distribution.
- **Term Env::bernoulli(double prob):** Creates a stochastic Term with that takes random value 1 with probability "prob" and 0, otherwise.
- **Term Env::categorical_distribution(vector<double> weights, vector<double> values):** Creates a Term that takes a random value in "values" according to the normalized distribution of the non-negative weights in the vector "weights."
- **Term Env::binomial(double p, long n):** Creates a random Term that takes random integer values in the interval $[0, n]$ according to the binomial distribution with density $f(k) = \binom{n}{k} p^k (1-p)^{n-k}$.
- **Term Env::poisson(double lamba, double high):** Creates a Term that takes random non-negative integer values according to the Poisson distribution with density $f(k) = \frac{\lambda^k e^{-\lambda}}{k!}$. If the value sampled by the distribution is greater than "high", then the value returned will equal the largest integer lower or equal "high."

- **Term Env::normal(double mu, double sigma, double low, double high):** Creates a Term that takes random values according to the Normal distribution with density $f(k) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$. If the value sampled by the distribution falls out of the specified range [low, high], then the value returned will equal the closest number within that interval.
- **Term Env::gamma(double shape, double scale, double high):** Creates a Term that takes random values according to the Gamma distribution with density $f(k) = \frac{1}{\Gamma(k)\Theta^k} x^{k-1} e^{-\frac{x}{\Theta}}$, whereby $k > 0$ is the shape parameter and $\Theta > 0$ is the scale parameter. If the value sampled by the distribution is greater than "high", then the value returned will equal the largest integer lower or equal "high."

2.15 User-Defined Stochastic Distributions

- **Term user_distribution(UserDistribution& ud):** Returns a stochastic Term that takes random values according to the user-defined distribution "ud." The user needs to create a class that derives from UserDistribution and overload a single function vector<double> sample_n(int n) which will return n samples from the distribution. The maximum number n that Seeker$^{(TM)}$ will ask for is given by the parameter "resolution."

- **vector<Term> Env::user_vector_distribution(UserVectorDistribution& ud):** Returns a vector of stochastic Terms that take random values according to the joint user-defined distribution "ud." The user needs to create a class that derives from UserVectorDistribution and overload a single function vector<vector<double>> sample_n(int n) which will return n samples from the joint distribution. The maximum number n that Seeker$^{(TM)}$ will ask for is given by the parameter "resolution".

2.15.1 The UserDistribution Class

- **UserDistribution(void) :** Base class constructor.
- **virtual vector<double> sample_n(int n) = 0:** Must be overloaded to create a user defined distribution. The maximum number n that Seeker$^{(TM)}$ will ask for is given by the parameter "resolution".

2.15.2 The UserVectorDistribution Class

- **UserVectorDistribution(void) :** Base class constructor.
- **virtual vector<vector<double> > sample_n(int n) = 0:** Must be overloaded to create a user defined vector distribution. The maximum number n that Seeker$^{(\mathrm{TM})}$ will ask for is given by the parameter "resolution".

2.16 Decision-Dependent Distributions

- **Term Env::user_defined_stochastic_term(vector<Term> features, UserStochTerm& ust):** Returns a stochastic Term that takes random values according to the user-defined distribution ”ust” which depends on the current values of the *deterministic* terms in ”features.” **Note: The terms in ”features” cannot themselves be stochastic terms.**

- **vector<Term> Env::user_defined_stochastic_vector(vector< Term> features, UserStochVector& usv, int numberTargets):** Returns a vector of ”numberTargets” stochastic Terms that take random values according to the joint user-defined vector distribution ”usv” which depends on the current values of the *deterministic* terms in ”features.” **Note: The terms in ”features” cannot themselves be stochastic terms.**

2.16.1 The UserStochTerm Class

- **UserStochTerm(void) :** Base class constructor.
- **vector<double> sample_n(vector<double> features, int n):** Must be overloaded to create a user defined feature-dependent stochastic distribution. The maximum number n that Seeker$^{(TM)}$ will ask for is given by the parameter ”resolution”.

2.16.2 The UserStochVector Class

- **UserStochVector(void) :** Base class constructor.
- **vector<vector<double>> sample_n(vector<double> features, int n):** Must be overloaded to create a user defined feature-dependent stochastic vector distribution. The maximum number n that Seeker$^{(\mathrm{TM})}$ will ask for is given by the parameter "resolution".

2.17 Stochastic Aggregation

2.17.1 Parametric Statistics

- **Term Env::aggregate_mean(Term source):** Returns the estimated arithmetic mean value of the stochastic term "source." "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.
- **Term Env::aggregate_geometric_mean(Term source):** Returns the estimated geometric mean of the stochastic term "source." "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.
- **Term Env::aggregate_variance(Term source):** Returns the estimated variance of the stochastic term "source." "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.
- **Term Env::aggregate_aav(Term source):** Returns the estimated mean of the absolute values of the stochastic term "source." "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.
- **Term Env::aggregate_rmsv(Term source):** Returns the root of the estimated mean of the square values of the stochastic term "source." "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.
- **Term Env::aggregate_stdev(Term source):** Returns the estimated standard deviation of the stochastic term "source." "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.

2.17.2 Order Statistics and Probability Masses

- **Term Env::aggregate_quantile(Term source, double ratio, bool maximize):** Returns the estimated "ratio"-quantile of the stochastic term "source." For example, if "ratio" = 0.75, then the term returned by this function would provide an estimate of the 75% quantile of the distribution of "source." If "maximize" is false, we would expect 75% of values of "source" to be smaller than this quantile. If "maximize" is true, we would expect 75% of values of "source" to be larger than this quantile. "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.

- **Term Env::aggregate_min(Term source):** Returns the estimated minimum value of the stochastic term "source." "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.

- **Term Env::aggregate_max(Term source):** Returns the estimated maximum value of the stochastic term "source." "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.

- **Term Env::aggregate_relative_frequency_geq(Term source, Term threshold):** Returns the estimated probability mass of the stochastic "source" term taking a value that is greater or equal than the deterministic term "threshold." "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.

- **Term Env::aggregate_relative_frequency_leq(Term source, Term threshold):** Returns the estimated probability mass of the stochastic "source" term taking a value that is lower or equal than the deterministic term "threshold." "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.

- **Term Env::aggregate_relative_frequency_eq(Term source, Term threshold):** Returns the estimated probability mass of the stochastic "source" term taking a value that equals that of the deterministic term "threshold." "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.

2.17.3 Descriptive Statistics

- **Term Env::aggregate_or(Term source):** Returns a term that is true if and only if the stochastic term "source" is estimated to assume non-zero values. "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.
- **Term Env::aggregate_and(Term source):** Returns a term that is true if and only if the stochastic term "source" is estimated to assume only non-zero values. "Source" must be a stochastic term. The function returns a deterministic term that can be used in constraints and the objective.

2.18 Constraints

2.18.1 Constraint Generation

- **Constraint leq(Term l, Term r)**: Creates a constraint that is true if, and only if, for the values of the terms l and r it holds that $l \leq r$.
- **Constraint lt(Term l, Term r)**: Creates a constraint that is true if, and only if, for the values of the terms l and r it holds that $l < r$.
- **Constraint geq(Term l, Term r)**: Creates a constraint that is true if, and only if, for the values of the terms l and r it holds that $l \geq r$.
- **Constraint gt(Term l, Term r)**: Creates a constraint that is true if, and only if, for the values of the terms l and r it holds that $l > r$.
- **Constraint eq(Term l, Term r)**: Creates a constraint that is true if, and only if, for the values of the terms l and r it holds that $l = r$.
- **Constraint neq(Term l, Term r)**: Creates a constraint that is true if, and only if, for the values of the terms l and r it holds that $l \neq r$.

Note: Creating a constraint has no effect on Seeker$^{(\mathrm{TM})}$ unless the constraint is posted to the model first! Use "enforce" to add constraints to your model.

2.18.2 Constraint Composition

- **Constraint and_(vector<Constraint> constraints)**: Creates a constraint that is true if, and only if, all constraints in "constraints" are true.
- **Constraint or_(vector<Constraint> constraints)**: Creates a constraint that is true if, and only if, at least one constraint in "constraints" is true.

2.18.3 Constraint Posting and Joint Generation and Posting

- **void enforce(Constraint constraint):** Only after calling this function, Seeker$^{(TM)}$ will consider assignments of values to decision variables feasible only of the constraint "constraint" evaluates to "true" under the assignment.
- **void enforce_leq(Term l, Term r)**: Enforces that the Seeker$^{(TM)}$ solver will only consider assignments to the decision variables for which for the values of the terms l and r it holds that $l \leq r$.
- **void enforce_lt(Term l, Term r)**: Enforces that the Seeker$^{(TM)}$ solver will only consider assignments to the decision variables for which for the values of the terms l and r it holds that $l < r$.
- **void enforce_geq(Term l, Term r)**: Enforces that the Seeker$^{(TM)}$ solver will only consider assignments to the decision variables for which for the values of the terms l and r it holds that $l \geq r$.
- **void enforce_gt(Term l, Term r)**: Enforces that the Seeker$^{(TM)}$ solver will only consider assignments to the decision variables for which for the values of the terms l and r it holds that $l > r$.
- **void enforce_eq(Term l, Term r)**: Enforces that the Seeker$^{(TM)}$ solver will only consider assignments to the decision variables for which for the values of the terms l and r it holds that $r = l$.
- **void enforce_neq(Term l, Term r)**: Enforces that the Seeker$^{(TM)}$ solver will only consider assignments to the decision variables for which for the values of the terms l and r it holds that $r \neq l$.

2.19 Optimization Functions

- **double minimize(Term obj, double time, double bound=-**$1e20$**)**: Minimizes the target term "obj" for "time" seconds. The minimization terminates early if the value of "obj" drops to or below "bound." If no bound is specified the optimization runs until the time limit.
- **double maximize(Term obj, double time, double bound=**$1e20$**)**: Maximizes the target term "obj" for "time" seconds. The maximization terminates early if the value of "obj" increases to or above "bound." If no bound is specified the optimization runs until the time limit.
- **double multi_objective(vector<Term> objectives, vector<double> fair, vector<double> excellent, vector<bool> directionMax, double time):** Optimizes the target terms "objectives" for "time" seconds. The direction of the optimization of the respective objective is maximization if, and only if, the corresponding entry in "directionMax" is true. For each objective, the respective values in "fair" and "excellent" must not be equal and be in the correct order. For an objective to be maximized, that value in "excellent" is expected to be strictly greater than that in "fair." For minimization, Seeker$^{(TM)}$ analogously expects the value in "excellent" to be strictly lower than that the "fair" value.

Note: To avoid numerical problems, the user is advised to make sure that the objective(s) to be optimized run(s) somewhere in [-1e08, 1e08]. If your optimization target can take larger absolute values, please consider dividing the term to be optimized by 1e03 or more, if needed. Analogously, if your objective function operates on a very small scale, consider multiplying by 1e3 or more.

2.20 Status, Progress Reports, and Search Statistics

- **void Env::set_report(double interval):** Reports the status of the search ever "interval" seconds. The reporting time may deviate for problems where evaluating the model takes a comparably long time.
- **void Env::set_report(double interval, vector<Term> reports, vector<string>):** The same as above, but in this case Seeker$^{(TM)}$ also reports the current value of all terms listed in "reports." To make the output more readable or parseable, the user is asked to provide a vector of equal length with corresponding identifiers for the reports.
- **long Env::get_number_evaluations(void):** Returns the number of objective function evaluations that Seeker$^{(TM)}$ conducted from creation to the call of this routine. This can help understand how complex the evaluation of the model is.
- **StatusType Env::get_status(void)** Returns the status of the last run (see below).

2.20.1 The StatusType Enumeration

- **'StatusType::unoptimized':** The status request was conducted before Seeker$^{(TM)}$ was called to minimize or maximize the objective.
- **'malformed':** Seeker$^{(TM)}$ encountered a problem in that the term to be optimized or a constraint of the problem could not be evaluated. This happens, for example, when dividing by 0.
- **'StatusType::infeasibleProblem':** The problem has no feasible solution and Seeker$^{(TM)}$ found a proof that the problem is infeasible.
- **'StatusType::infeasibleSolution':** Seeker$^{(TM)}$ was not able to find a feasible solution in the time allowed, but also was not able to prove that the problem has no feasible solutions.
- **'StatusType::timeout':** Seeker$^{(TM)}$ ran out of time, but found a feasible solution that does not meet the optimization bound. Please note that the latter may exist, but Seeker$^{(TM)}$ was simply not able to find it.
- **'StatusType::bounded':** Seeker$^{(TM)}$ found a feasible solution that meets or exceeds the optimization bound and consequent;y stopped the optimization before the allotted time ran out.
- **'StatusType::optimal':** Seeker$^{(TM)}$ found a feasible solution that is provably optimal for the problem.

2.21 Parameters

- **void Env::set_parameters(vector<double> pa):** The function sets the internal parameters of Seeker$^{(TM)}$. As input, the function expects a vector of 191 doubles, whereby the first 180 are expected to take integer values in $[-1000, 1000]$ and the final 11 parameters to take integer values in $[1, 100]$. **Note: This function should be used in combination with an automatic tuner after the model development process is completed and before the model is deployed.**

- **void Env::set_local_improvement_size(double s):** Sets the relative size of the neighborhood considered when trying to improve a solution locally. The input parameter s is expected to be in $[0, 1]$. The neighborhood is comparably larger, and local improvements are comparably more costly, when s is closer to 1.

- **void Env::set_global_improvement_size(double s):** Sets the relative size of the neighborhood considered when trying to improve a solution by means of recombination. The input parameter s is expected to be in $[0, 1]$. The neighborhood is comparably larger, and recombination improvements are comparably more costly, when s is closer to 1.

- **void Env::set_exploration_size(double s, double t):** Sets the relative size of the neighborhood considered when trying to explore the larger search space. The input parameters $s < t$ are expected to be in $[0, 1]$. The neighborhood can be comparably larger, and exploration excursions can be comparatively more elaborate, when t is closer to 1. Analogously, the neighborhood may be comparably smaller, and exploration excursions will then focus more closely in the neighborhood of the current solution, when s is closer to 0.

- **void Env::set_restart_likelihood(double prob):** Sets the probability "prob" in $[0, 1]$ which affects the frequency with which the search starts from a new exploration point. This value can have dramatic effects on search performance and should be changed with care. A value of 0.01 would be considered high for this parameter.

- **Env::set_stochastic_parameters(int resolution, double speed)**: Even though this is not exactly how Seeker$^{(TM)}$ works, you may think of the "resolution" parameter as the number of stochastic "scenarios," whereby all "scenarios" are considered equally likely. This parameter thereby determines the smallest event probability that Seeker$^{(TM)}$ will be able to consider, hence the name "resolution." The second parameter "speed" has to be set in the interval $[0, 1]$ and affects the speed and accuracy with which Seeker$^{(TM)}$ attempts to assess the model. Generally speaking, Seeker$^{(TM)}$ will need more time per evaluation if the speed is set closer to 0 and less time when set closer to 1 (at the cost of larger approximation errors).

- **void Env::set_parallel_coordination_parameters(double interval, double initialWait):** Sets the time interval "interval" seconds in which a process will coordinate its work with the other processes, as well as the time "initialWait (also in seconds) that the process should wait initially or after an internal restart before coordinating.

3 The Term Class

3.1 Construction

A term cannot be constructed by the user directly. To construct a term, please use the respective environment functions. Note that decision variables are terms, data can be converted into terms using "env.convert," and terms can be derived from terms using other environment functions.

The operators outlined in this Section also allow deriving terms from other terms using the implicit language notation. This enables the user to derive terms in a natural way, for example by writing $x = y + 1$ instead of $x = \mathrm{env.sum}([y, \mathrm{env.convert}(1)])$ which has the exact same meaning.

3.2 Unary Operators

- **Term Term::operator-(void)**: Returns the negative of the value of the given term.
- **Term Term::operator!(void)**: Returns 1 if the original term evaluates to 0, and 0 otherwise. In Python, you need to use **Term Term::not_(void)**. For example: a = env.categorical(0,1). b = a.not_(). b = not a is not allowed.

3.3 Binary Operators

3.3.1 Arithmetic Operators

- **Term operator+(const Term& a, const Term& b)**: Returns $a + b$.
- **Term operator-(const Term& a, const Term& b)**: Returns $a - b$.
- **Term operator/(const Term& a, const Term& b)**: Returns a/b if b evaluates to a value now equal 0. Otherwise, the return value is "undefined."
- **Term operator*(const Term& a, const Term& b)**: Returns $a * b$.
- **Term operator%(const Term& a, const Term& b)**: If $b \neq 0$, the operator returns $a-(a \text{ div } b)*b$. If $b = 0$, the operator returns "undefined."
- **Term operator^(const Term& a, const Term& b)**: Returns a^b if $a > 0$, and "undefined" if $a < 0$. If $a = 0$, the operator returns 1 if $b = 0$, 0 if $b > 0$, and "undefined" if $b < 0$.

3.3.2 Logic Operators

- **Term operator==(const Term& a, const Term& b)**: Returns a term that is 1 if $a = b$ and 0 otherwise. *Do not confuse this with the corresponding constraint.*
- **Term operator!=(const Term& a, const Term& b)**: Returns a term that is 0 if $a = b$ and 1 otherwise. *Do not confuse this with the corresponding constraint.*
- **Term operator<=(const Term& a, const Term& b)**: Returns a term that is 1 if $a \leq b$, and 0 otherwise. *Do not confuse this with the corresponding constraint.*
- **Term operator>=(const Term& a, const Term& b)**: Returns a term that is 1 if $a \geq b$, and 0 otherwise. *Do not confuse this with the corresponding constraint.*
- **Term operator<(const Term& a, const Term& b)**: Returns a term that is 1 if $a < b$, and 0 otherwise. *Do not confuse this with the corresponding constraint.*
- **Term operator>(const Term& a, const Term& b)**: Returns a term that is 1 if $a > b$, and 0 otherwise. *Do not confuse this with the corresponding constraint.*

Note: To avoid numerical issues arising from limited machine precision, when one or both of the terms in a comparison evaluate to a float or double, the comparison is conducted with slack. E.g., $a <= b$ **implicitly becomes** $a <= b + 1\mathrm{e} - 8$**, and** $a < b$ **implicitly becomes** $a < b - 1\mathrm{e} - 8$**. This can lead to unexpexted behavior. For example, env.if_($a < 0$, a, env.sqrt(a)) may lead to an undefined term evaluation as "a" may not be greater or equal 0 on the "else" side of the condition. In this case, please adjust the value of "a" accordingly. In the example, we should use env.if_($a < 0$, a, env.sqrt(env.max_0(a))).**

Note: The following operators are for C++ only. In Python, please use the explicit functions 'Env::or_' and 'Env::and_'. Please do not use the corresponding Python operators "__or__" and "__and__" as their behavior is undefined.

- **Term operator||(const Term& a, const Term& b)**: Returns 0 if $a = b = 0$, and 1 otherwise.
- **Term operator&&(const Term& a, const Term& b)**: Returns 0 if $a = 0$ of $b = 0$, and 1 otherwise.

3.4 Term Status Check

- **bool Term::status_ok(void):** Returns ”true” if, and only if, the respective term could be evaluated correctly.