Seeker$^{(TM)}$ 1.0.0 User's Manual

InsideOpt$^{(TM)}$

October 2023

Contents

1 Introduction **4**

2 First Examples **4**
2.1 Designing a Material-Efficient Can 4
2.1.1 Can Design in C++ 4
2.1.2 Can Design in Python 5
2.2 Constrained Facility Location 6
2.2.1 Facility Location in C++ 7
2.2.2 Facility Location in Python 9

3 Basic Functions and Operators **11**
3.1 Environment Functions and Operators 11
3.1.1 Environment Creation and Termination 11
3.1.2 Creating Decision Variables 11
3.1.3 Optimization Functions 12
3.1.4 Constraint Definition 13
3.1.5 Unary Operators 13
3.1.6 Binary Operators 14
3.1.7 Index Operators 15
3.1.8 Choice Operator 15
3.1.9 Aggregators 15
3.2 Term Operators 17
3.2.1 Unary Operators 17
3.2.2 Binary Operators 17
3.2.3 Status Check 18

4 Models for Classical Optimization Problems **19**
4.1 Bin Packing 19
4.1.1 C++ 19
4.1.2 Python 20
4.2 Continuous Optimization 20
4.2.1 C++ 21
4.2.2 Python 21
4.3 Graph Coloring 22
4.3.1 C++ 22
4.3.2 Python 23

5 Advanced Modeling Concepts **24**
5.1 Partitionings and Packings 24
5.1.1 Bin Packing - Continued 25
5.1.2 Discrete Capacitated Facility Location - Continued 26
5.2 Efficient Boolean Indicators 28
5.2.1 Switch Conditions 28
5.2.2 Interval Conditions 32

5.3 Permutations and Convex Combinations . . . 33
5.3.1 Permutations . . . 33
5.3.2 Convex Combinations . . . 34
5.4 Sortings and Quantiles . . . 34
5.4.1 Sortings . . . 34
5.4.2 Quantiles . . . 36
5.5 Logic Combinations of Constraints . . . 36
5.5.1 Constraint Generation . . . 36
5.5.2 2D Rectangle Packing in Python . . . 37
5.6 User-Defined Terms . . . 38
5.6.1 Creating New Terms and Functions . . . 38
5.6.2 Adding User Terms to the Model . . . 39
5.6.3 A Continuous Optimization Example . . . 40

6 Multi-Objective Optimization 41

7 Nested Linear Optimization 42
7.1 Linear Programming Terms . . . 42
7.2 Linear Constraints Doubling in Seeker(TM) . . . 43
7.3 Non-Linear Optimization With Nested LP . . . 43

8 Stochastic Optimization 45
8.1 Environment Creation and Stochastic Parameters . . . 45
8.2 Creating Stochastic Data Terms . . . 45
8.3 User-Defined Distributions . . . 46
8.4 Solution-Dependent Distributions . . . 47
8.5 Aggregation . . . 48

9 Stochastic Optimization Examples 50
9.1 Monty Hall Problem . . . 50
9.2 Betting . . . 50
9.3 Integrated Pricing and Production Planning . . . 51

10 Parameters and Tuning 53
10.1 Automatic Tuning . . . 53
10.2 Manual Tinkering . . . 53

11 Parallel Optimization 55
11.1 Prerequisites . . . 55
11.2 Preparing Your Program . . . 55
11.3 Adding More Parallel Processes and Crash Recovery . . . 55
11.4 Parallel Coordination Parameters . . . 56

12 Progress Reports and Search Statistics 57

1 Introduction

The Seeker$^{(TM)}$ Library consists of two main classes, Env and Term, plus some few helper classes. This manual first introduces the main classes that are used to formulate and solve optimization problems. Later, we will cover the additional classes that allow us to formulate specific variables, such as set partitions, packings, permutations, quantiles, and sortings.

2 First Examples

2.1 Designing a Material-Efficient Can

Problem: Assume we are to design a cylindrical soup can with a given volume V that minimizes the material costs, which are determined by the surface of the can. The two variables we need to determine are the diameter D of the cylinder bottom, and the height H of the cylinder. Since the volume is given, as soon as the diameter is chosen, we already know what the height must be to obtain the desired volume:

$$V = \frac{\pi H D^2}{4} \qquad \text{or} \qquad H = \frac{4V}{\pi D^2}.$$

The surface S of the can is given by

$$S = \frac{\pi D^2}{2} + \pi D H$$

Substituting H, we get

$$S = \frac{\pi D^2}{2} + \frac{4V}{D}$$

Our problem can therefore be stated as

$$\text{Minimize } S = \tfrac{\pi D^2}{2} + \tfrac{4V}{D} \text{ such that } D \geq 0.$$

2.1.1 Can Design in C++

To model this problem in Seeker$^{(TM)}$, we must first create our environment. All objects, variables and expressions that we will create later will be registered to our environment. The environment will also provide us with functions to create complex terms, constraints, and our optimization objective. Every Seeker$^{(TM)}$ application therefore starts with the creation of an environment object env of type Seeker::Env: "Env env("license.sio");" (Line 12). Note that you must specify the path and name to a valid license file to use the Seeker$^{(TM)}$.

Next, we need to declare a decision variable, the diameter D of our can. We use the environment function "Seeker::Term Seeker::Env::continuous(double lowerBound, double upperBound)" to create the corresponding term: "Term diameter=env.continuous(0,1e6);" (Line 15).

Based on the math we did above, we next compute the surface of the can based on the diameter: "Term surface=pi*env.sqr(diameter)/2+4*volume/diameter;" (Line 16). Note that we use the environment function "Term sqr(Term x)" to square the value of *diameter*.

Our last step is to call the environment function "void Seeker::Env::minimize(-Term obj, double timeLimit, double loweBound)," handing over the "surface" Term as our minimization target: "env.minimize(surface,tl,bound);" (Line 20).

To read out the values of any "Seeker::Term" that was created along the way, we use the function "double Seeker::Term::get_value(void)" (Line 22).

Finally, we clean up our environment by calling the function "void Seeker::-Env::end(void)" (Line 28) which frees the memory of the objects associated with the environment.

```
1  #include <math.h>
2  #include <iostream>
3  #include <env.hpp>
4  #include <term.hpp>
5
6  using namespace std;
7  using namespace Seeker;
8
9  int main(int argc, char** argv) {
10   double volume=atof(argv[1]);
11   // Create environment
12   Env env("license.sio");
13   double pi=M_PI;
14   // Define Diameter variable and objective term
15   Term diameter=env.continuous(0,1e6);
16   Term surface=pi*env.sqr(diameter)/2+4*volume/diameter;
17   // timelimit 10 seconds, surface lower bound is 0
18   double tl=0.01, bound=0;
19   // minimize surface
20   env.minimize(surface,tl,bound);
21   // read final values
22   double optS=surface.get_value(), optD=diameter.get_value();
23   double optH=4*volume/(pi*optD*optD);
24   cout << "Optimal Diameter = " << optD << endl
25        << "Optimal Height = " << optH << endl
26        << "Optimal Surface = " << optS << endl;
27   // Terminate environment
28   env.end();
29 }
```

2.1.2 Can Design in Python

In Python 3, we also first create our environment, to which all objects, variables and expressions will be registered to. After importing the library in Line 1, we declare "env = skr.Env("license.sio")" (Line 4). Note that you have to provide the path and name to a valid license file to use the Seeker$^{(TM)}$ library.

Next, we declare a decision variable, the diameter D of our can. We use the environment function "Seeker::Term Seeker::Env::continuous(double lower-

Bound, double upperBound)" to create the corresponding term: "diameter=-env.continuous(0,1e6);" (Line 7).

Based on the math we did above, we next compute the surface of the can based on the diameter: "surface=pi*env.sqr(diameter)/2+4*volume/diameter;" (Line 8). Note that we use the environment function "Term sqr(Term x)" to square the value of *diameter*.

Our last step is to call the environment function "void Seeker::Env::minimize(-Term obj, double timeLimit, double loweBound)," handing over the "surface" Term as our minimization target: "env.minimize(surface,tl,bound);" (Line 14).

To read out the values of any "Seeker::Term" that was created along the way, we use the function "double Seeker::Term::get_value(void)" (Lines 17/18).

Finally, we clean up our environment by calling the function "void Seeker::-Env::end(void)" (Line 25) which frees the memory of the objects associated with the environment.

```
1  import seeker as skr
2  import math
3  def main(volume):
4      env = skr.Env("license.sio")
5      pi = math.pi
6      #Define Diameter variable and objective term
7      diameter = env.continuous(0, 1e6)
8      surface = (pi/2)*env.sqr(diameter)+(4*volume)/diameter
9      #timelimit 10 seconds, surface lower bound is 0
10     tl = 0.01
11     bound = 0
12     #minimize surface
13     print("start minimization")
14     env.minimize(surface, tl, bound)
15     print("done with minimization")
16     #read final values
17     optS = surface.get_value()
18     optD = diameter.get_value()
19     optH = 4*volume/(pi*optD*optD)
20     print("Optimal Diameter =", optD)
21     print("Optimal Height =", optH)
22     print("Optimal Surface =", optS)
23     print("Volume is =", pi*optD*optD*optH/4)
24     #Terminate environment
25     env.end();
```

2.2 Constrained Facility Location

Optimizing the can is particularly easy to model since we managed to formulate the problem without any side constraints. That is to say, any value in the range we specified when declaring the diameter variable D is legitimate.

In most applications, we have more than one decision variable, and not all combinations of values in their respective range can be realized in practice. Consider the example of discrete capacitated facility location (DCFL): Assume we are given l locations, where new facilities can be opened to serve c customers. In this discrete variant of the problem, each customer must be assigned to

exactly one (open) facility, where it creates a facility-independent service load ($\text{custDemand}_i \in \mathbb{R}^+$ for $0 \leq i < c$), whereby each facility offers only a limited capacity for serving customers ($\text{locCapacity}_j \in \mathbb{R}^+$ for $0 \leq j < l$).

Opening a facility at a given location incurs a location-specific cost ($\text{locCost}_j \in \mathbb{R}^+$ for $0 \leq j < l$). Moreover, the cost of serving a customer incurs a cost that depends on both the customer and the facility location ($\text{servCost}_{ji} \in \mathbb{R}^+$ for $0 \leq j < l$ and $0 \leq i < c$).

To formulate this problem, we will need Seeker$^{(\text{TM})}$ to enforce the capacity constraints. In the C++ and Python programs below, we show one way how this problem can be formulated.[1]

2.2.1 Facility Location in C++

First, we define decision variables $\text{custAllocation}_i \in \{0, \ldots, l-1\}$ for $0 \leq i < c$ which determine from which location each customer will be served (Line 33). We use categorical variables as there is no meaningful ordering of the locations.

To infer the total load that this allocation generates at each location, we next define a matrix $\text{locCustEffort}_{ji}$ for $0 \leq j < l$ and $0 \leq i < c$, where $\text{locCustEffort}_{ji} = \text{custDemand}_i$ if, and only if, $\text{custAllocation}_i == j$, and 0 otherwise (Lines 40/41). For this purpose, we use the environment function "Term Seeker::Env::if_ (Term condTerm, Term thenTerm, Term elseTerm)" which returns the value of the "thenTerm" if the condition evaluates to true (whereby, here and in general, any value evaluates to true if and only if it is not equal zero), and the value of the "elseTerm" otherwise.

Based on this matrix, we now need to compute the total load at each location. To this end, we simply need to sum up the values in the rows of the matrix. Now, technically we could loop over the Terms to sum and use the "Term Term::operator+(const Term&, const Term&)." This is, however, an inefficient way to formulate the model, and should be avoided.

The sum of terms, much like other operators such as minimum, maximum, and, or, product, or statistical measures like variance, standard deviation, or various norms, are *aggregates* over some, usually more than two, operands. For these, the vector of operands is handed over in one call, for example to "Term Seeker::Env::sum(vector<Term> operands)" (Line 46).

Equipped with the total service demands at each location, we can now formulate the capacity constraints. For this purpose, we use the environment function "void Seeeker::Env::enforce_leq(Term left, Term right)" (Line 48). This function tells the environment, that any feasible solution is expected to obey the constraint "left $\leq$ right."

The rest of the model is then straight forward with the concepts we just introduced.

```
1 #include <math.h>
2 #include <iostream>
3 #include <string>
```

[1] Please note that the models we present are not optimized for performance but used to illustrate the features of the solver.

```
#include <vector>
#include <env.hpp>
#include <term.hpp>

using namespace std;
using namespace Seeker;

extern void readFile(string fileName, int& l, int& c,
                     vector<double>& locCost,
                     vector<double>& locCapacity,
                     vector<vector<double> >& servCost,
                     vector<double>& custDemand);

int main(int argc, char** argv) {
  // gather instance data
  string inputFile(argv[1]);
  // l=numberOfLocations, c=numberOfCustomers
  int l,c;
  vector<double> locCost;
  vector<vector<double> > servCost;
  vector<double> custDemand;
  vector<double> locCapacity;
  readFile(inputFile,l,c,locCost,locCapacity,servCost,custDemand);

  // Create environment
  Env env("license.sio");
  // create customer allocation variables
  vector<Term> custAllocation;
  for (int i=0; i<c; i++)
    custAllocation.push_back(env.categorical(0,l-1));

  // for each location/customer pair, compute the
  // corresponding effort, based on allocation
  vector<vector<Term> > locCustEffort(l);
  for (int j=0; j<l; j++)
    for (int i=0; i<c; i++)
        locCustEffort[j].push_back(env.if_(custAllocation[i]==j,
                                           custDemand[i],0));

  // compute the load per location
  vector<Term> locLoad;
  for (int j=0; j<l; j++) {
    locLoad.push_back(env.sum(locCustEffort[j]));
    // enforce the load per location does not exceed it capacity
    env.enforce_leq(locLoad[j],locCapacity[j]);
  }

  // compute costs for opening the locations that are being used
  vector<Term> locUtilizationCost;
  for (int j=0; j<l; j++)
      locUtilizationCost.push_back(env.if_(locLoad[j]>0,
                                           locCost[j],0));
  Term locationCosts=env.sum(locUtilizationCost);

  // compute the assignment-specific costs
  vector<vector<Term> > locCustServCosts(l);
  for (int j=0; j<l; j++)
```

```
61      for (int i=0; i<c; i++)
62          locCustServCosts[j].push_back(env.if_(custAllocation[i]==j,
63                                                servCost[j][i],0));
64    vector<Term> locServCosts;
65    for (int j=0; j<l; j++)
66      locServCosts.push_back(env.sum(locCustServCosts[j]));
67    Term assignmentCosts=env.sum(locServCosts);
68
69    // timelimit is 10 seconds, lower bound on costs is 0
70    double tl=10, bound=0;
71
72    // define objective term
73    Term totalCosts=locationCosts+assignmentCosts;
74    // minimize
75    env.minimize(totalCosts,tl,bound);
76
77    //read final cost value
78    cout << "Optimal Costs = " << totalCosts.get_value() << endl;
79
80    //Terminate environment
81    env.end();
82  }
```

2.2.2 Facility Location in Python

First, we define decision variables $\text{custAllocation}_i \in \{0, \ldots, l-1\}$ for $0 \leq i < c$ which determine from which location each customer will be served (Line 15). We use categorical variables as there is no meaningful ordering of the locations.

To infer the total load that this allocation generates at each location, we next define a matrix $\text{locCustEffort}_{ji}$ for $0 \leq j < l$ and $0 \leq i < c$, where $\text{locCustEffort}_{ji} = \text{custDemand}_i$ if, and only if, $\text{custAllocation}_i == j$, and 0 otherwise (Line 23). For this purpose, we use the environment function "Term Seeker::Env::if_ (Term condTerm, Term thenTerm, Term elseTerm)" which returns the value of the "thenTerm" if the condition evaluates to true (whereby, here and in general, any value evaluates to true if and only of it is not equal zero), and the value of the "elseTerm" otherwise.

Based on this matrix, we now need to compute the total load at each location. To this end, we simply need to sum up the values in the rows of the matrix. The sum of terms, much like other operators such as minimum, maximum, and, or, product, or statistical measures like variance, standard deviation, or various norms, are aggregates over some, usually more than two, operands. For these, the vector of operands is handed over in one call, for example to "Term Seeker::Env::sum(vector<Term> operands)" (Line 29).

Equipped with the total service demands at each location, we can now formulate the capacity constraints. For this purpose, we use the environment function "void Seeeker::Env::enforce_leq(Term left, Term right)" (Line 31). This function tells the environment, that any feasible solution is expected to obey the constraint "left $\leq$ right."

The remaining model follows.

```
import seeker as skr

def read_instance(fileName,env):
    #read instance from file and return numberOfLocations,
    numberOfClients, locationCosts, serviceCosts, customerDemand,
    and locationCapacity, whereby arrays of Terms are returned (
    using env.convert(...))
    return #...

def main(fileName):
    # Create environment
    env = skr.Env("license.sio")
    l, c, locCost, servCost, custDemand, locCapacity =
    read_instance(fileName, env)

    # create customer allocation variables
    custAllocation = [env.categorical(0, l - 1) for _ in range(c)]

    # for each location/customer pair, compute the
    # corresponding effort, based on allocation
    locCustEffort = [[env.if_(custAllocation[i] == j,
                              custDemand[i], 0)
                      for i in range(c)] for j in range(l)]

    # compute the load per location
    locLoad = [env.sum(locCustEffort[j]) for j in range(l)]
    for j in range(l):
        env.enforce_leq(locLoad[j], locCapacity[j])

    # compute costs for opening the locations that are being used
    locUtilizationCost = [env.if_(locLoad[j] > 0, locCost[j], 0)
                          for j in range(l)]
    locationCosts = env.sum(locUtilizationCost)

    # compute the assignment-specific costs
    locCustServCosts = [[env.if_(custAllocation[i] == j,
                                 servCost[j][i], 0)
                         for i in range(c)] for j in range(l)]
    locServCosts = [env.sum(locCustServCosts[j]) for j in range(l)]
    assignmentCosts = env.sum(locServCosts)

    # timelimit is 10 seconds, lower bound on costs is 0
    tl = 10
    bound = 0

    # define objective term and minimize
    totalCosts = locationCosts + assignmentCosts
    env.minimize(totalCosts, tl, bound)

    # read final cost value
    print("Optimal Costs =", totalCosts.get_value())

    # Terminate environment
    env.end()
```

3 Basic Functions and Operators

3.1 Environment Functions and Operators

Having introduced two examples, we now provide a more comprehensive overview of environment functions that are available for modeling optimization problems. In C++, all of the following are part of the Seeker::Env namespace.

3.1.1 Environment Creation and Termination

- **Env(string)**: Constructor. Creates the Seeker$^{(TM)}$ environment, provided a valid license file name is given.
- **void end(void)**: Must be called before terminating the environment to avoid memory leaks.
- ∼**Env(void)**: Destructor. Terminates the Seeker$^{(TM)}$ environment.

3.1.2 Creating 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."

3.1.3 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.

Note: To avoid numerical problems, the user is advised to make sure that the objective to be optimized runs 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. Similarly, if your objective operates on a very small scale, consider multiplying it with 1e3 or more.

After the optimization, the user can check the status of the optimization run using the following environment function:

- **StatusType Env::get_status(void)**

This function returns the status of the last run, which can be either

- **'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.

- **'infeasibleProblem':** The problem has no feasible solution and Seeker$^{(TM)}$ found a proof that the problem is infeasible.
- **'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.
- **'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.
- **'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.
- **'optimal':** Seeker$^{(TM)}$ found a feasible solution that is provably optimal for the problem.

3.1.4 Constraint Definition

- **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 $r \neq l$.

3.1.5 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.

3.1.6 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)" (see Section 3.2.2).

3.1.7 Index Operators

Here, and in general, the template type T is defined for T being either bool, int, long, float, or double. No other types are permitted.

- **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, 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, 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(TM) will exit if this is not the case and the result of this term is material for the objective or the constraint status.

3.1.8 Choice Operator

- **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.

3.1.9 Aggregators

As mentioned in our description above, we refer to operators that take an arbitrary number of terms as input as "aggregators." While, for reasons of convenience, for many of these aggregators we also provide the corresponding binary operators (see Section 3.2.2), for reasons of efficiency when optimizing the models we highly recommend users to formulate their models using the functions listed in this section.

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

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

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

- **Term min(vector<Term> terms)**: Returns the minimum of the values in "terms."
- **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.
- **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.
- **Term and_ (vector<Term> terms):** Returns 0 if any of the values in "terms" evaluates to 0, and 1 otherwise.
- **Term or_ (vector<Term> terms):** Returns 1 if any of the values in "terms" does not evaluate to 0, and 0 otherwise.
- **Term mean(vector<Term> terms):** Returns the sum of the values in "terms" divided by their number.
- **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 standard_deviation(vector<Term> terms)**: Returns the square root of the variance of the values in "terms."
- **Term norm1(vector<Term> terms)**: Returns the sum of absolute values in "terms."
- **Term norm2(vector<Term> terms)**: Returns the square root of the sum of square values in "terms."
- **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 rmsv(vector<Term> terms)**: Returns the square root of the mean of the squared values in "terms."
- **Term average_absolute_value(vector<Term> terms)**: Returns the mean of the absolute values in "terms."

At times, it can be convenient to use constant data as part of aggregators and within other operators and functions. To enable this, Seeker$^{(\text{TM})}$ provides functions that turn individual data points, vectors, and matrices into the corresponding structures of terms.

- **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.

3.2 Term Operators

Apart from the above environment functions, we can also build new terms using term operators.

3.2.1 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.

3.2.2 Binary Operators

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 $b < 0$.

Logic Operators: Please note that the following binary logic operators return a new Boolean term that takes values 1 (for true) or 0 (for false). To construct a constraint, please use environment functions, such as "Env::leq" for a lower or equal constraint, and "Env::enforce" to add the constraint to the model.

- **Term operator==(const Term& a, const Term& b)**: Returns 1 if $a = b$ and 0 otherwise.
- **Term operator!=(const Term& a, const Term& b)**: Returns 0 if $a = b$ and 1 otherwise.
- **Term operator<=(const Term& a, const Term& b)**: Returns 1 if $a \leq b$, and 0 otherwise. *Do not confuse this with the corresponding constraint from Section 3.1.4.*
- **Term operator>=(const Term& a, const Term& b)**: Returns 1 if $a \geq b$, and 0 otherwise. *Do not confuse this with the corresponding constraint from Section 3.1.4.*
- **Term operator<(const Term& a, const Term& b)**: Returns 1 if $a < b$, and 0 otherwise. *Do not confuse this with the corresponding constraint from Section 3.1.4.*
- **Term operator>(const Term& a, const Term& b)**: Returns 1 if $a > b$, and 0 otherwise. *Do not confuse this with the corresponding constraint from Section 3.1.4.*

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.

3.2.3 Status Check

A term may take undefined values, for example after division by zero, indexing a vector outside its dimension, etc. This is not uncommon and poses no issue as long as the undefined value does not need to be used to evaluate the objective function value or the status of a constraint. When assessing the value of a term, it is therefore good practice to check if its status is okay.

- **bool Term::status_ok(void)**

4 Models for Classical Optimization Problems

Using the above functions, we now demonstrate the usage of Seeker$^{(TM)}$ for some classical optimization problems. Note that the models presented here are usually not the most efficient, but serve the purpose of demonstrating how models can be formulated and solved using the Seeker$^{(TM)}$ library.

4.1 Bin Packing

Given a set of $n \in \mathbb{N}$ items with weights $w_0, \ldots, w_{n-1} \in \mathbb{R}^+$, a number $m \in \mathbb{N}$ of available bins, and a bin capacity $C \in \mathbb{R}^+$, we are to assign each item to a bin such that 1. the total weight of items assigned to a bin does not exceed the capacity, and 2. the number of bins which are assigned at least one item is minimized.

Using Seeker$^{(TM)}$, we can formulate this problem as follows.

4.1.1 C++

```
#include "env.hpp"
#include "term.hpp"
#include <iostream>
#include <vector>

using namespace Seeker;
using namespace std;

extern void read_instance(int& m, vector<double>& w, double& C);

int main(void) {
  int m; vector<double> w; double C;
  readInstance(m,w,C);
  int n=w.size();

  Env env("license.sio");
  // create Decision Variables
  vector<Term> assignment;
  vector<Term> weight;
  for (int i=0; i<n; i++) {
    assignment.push_back(env.categorical(0,m-1));
    weight.push_back(Term(env,w[i]));
  }
  // Compute Bin Weights and Bin Usage
  vector<vector<Term> > weightMatrix(m);
  vector<Term> binWeight, binUsed;
  for (int b=0; b<m; b++) {
    for (int i=0; i<n; i++)
      weightMatrix[b].push_back(env.if_(assignment[i]==b,
                                        weight[i],0));
                binWeight.push_back(env.sum(weightMatrix[b]));
      // Add Capacity Constraint
      env.enforce\_leq(binWeight[b],C);
      binUsed.push_back(env.if_(binWeight[b]>0,1,0));
  }
```

```
  // Minimize number of bins used for 100 seconds
  // lower bound is 1
  Term numberUsed=env.sum(binUsed);
  env.minimize(numberUsed,100,1);
  // Report the solution
  for (int b=0; b<m; b++)
    cout << "Bin " << b << ": " << binWeight[b].get_value()
         << endl;
  cout << "Bins used = " << numberUsed.get_value() << endl;
  env.end();
  return 0;
}
```

4.1.2 Python

```
import seeker as skr

def main(fileName):
    maxBins, weights, capacity = read_instance(fileName)
    env = skr.Env("license.sio")
    # create Decision Variables
    assignment = [env.categorical(0, maxBins - 1)
                  for _ in range(len(weights))]
    # Compute Bin Weights and Bin Usage
    weightMatrix = [[env.if_(assignment[i] == b, weights[i], 0)
                     for i in range(len(weights))]
                    for b in range(maxBins)]
    binWeight = [env.sum(weightMatrix[b]) for b in range(maxBins)]
    binUsed = [binWeight[b] > 0 for b in range(maxBins)]
    for b in range(maxBins):
        # Add Capacity Constraint
        env.enforce_leq(binWeight[b], capacity)

    # Minimize number of bins used for 100 seconds
    # lower bound is 1
    numberUsed = env.sum(binUsed)
    env.minimize(numberUsed, 100, 1)
    # Report the solution
    for b in range(maxBins):
        print("Bin", b, ":", binWeight[b].get_value())
    print("Bins used =", numberUsed.get_value())
    env.end()
```

4.2 Continuous Optimization

Given a number n, we are to minimize the Schwefel function in n dimensions, which is given by the formula:

$$418.9828872724339n - \sum_{i=0}^{n-1} x_i \sin(\sqrt{|x_i|}),$$

whereby $-500 \leq x_i \leq 500 \; \forall \; 0 \leq i < n$.

4.2.1 C++

```cpp
#include "env.hpp"
#include "term.hpp"
#include <iostream>
#include <vector>

using namespace Seeker;
using namespace std;

int main(int argc, char** argv) {
  int n=atoi(argv[1]);
  Env env("license.sio");
  // Create the model
  Term A=env.convert(n*418.9828872724339);
  vector<Term> summands(n);
  vector<Term> variables(n);
  for (int i=0; i<n; i++) {
    variables[i]=env.continuous(-500,500);
    summands[i]=variables[i]*env.sin(env.sqrt(
                                env.abs(variables[i])));
  }
  Term obj=A-env.sum(summands);
  // Minimize the objective for 50 seconds with lower bound 0
  env.minimize(obj,50,0);
  // Report the solution
  cout << "Optimum is " << obj.get_value() << endl;
  for (int i=0; i<minimum(n,10); i++)
    cout << variables[i].get_value() << " ";
  cout << endl;
  env.end();
}
```

4.2.2 Python

```python
import seeker as skr
import math
import random as rd

def main(n, timeLimit):
    env = skr.Env("my_license.sio")
    A = env.convert(n * 418.9828872724339)
    variables = [env.continuous(-500
                                , 500
                                , rd.randint(-50000, 50000) * 0.01)
                 for i in range(n)]
    summands = [variables[i]
                * env.sin(env.sqrt(env.abs(variables[i])))
                for i in range(n)]
    obj = A - env.sum(summands)
    env.minimize(obj, timeLimit, 0)
    print("Optimum is", obj.get_value())
    print("Number of function evaluatios",
          env.get_number_evaluations())
    env.end()
```

4.3 Graph Coloring

Graph coloring is another classical optimization problem. We are given n items ("nodes"), each associated with a list of items that are incompatible with the given item ("arcs"). The objective is to minimize the number of sets such that each item is assigned to a set, and all items in the same set are compatible.

4.3.1 C++

```
#include "env.hpp"
#include "term.hpp"
#include <vector>
#include <math.h>
#include <iostream>

using namespace std;
using namespace Seeker;

extern void readInstance(char* fileName,
                         vector<vector<int> >& adjacency);

int main(int argc, char** argv)
{
  vector<vector<int> > adjacency;
  readInstance(argv[1],adjacency);
  // Model the problem
  Env env("license.sio");
  int n=adjacency.size();
  vector<Term> color;
  for (int i=0; i<n; i++)
    color.push_back(env.categorical(0,n-1));
  for (int i=0; i<n; i++)
    for (int h=0; h<(int)adjacency[i].size(); h++) {
      int j=adjacency[i][h];
      // Enforce compatibility constraints
      env.enforce_neq(color[i],color[j]);
    }
  Term obj=env.max(color);
  // Minimize objective for 100 seconds with lower bound 1
  env.minimize(obj,100,1);
  cout << "Found a solution with " << obj.get_value()
       << " colors\n";
  env.end();
}
```

4.3.2 Python

```python
import seeker as skr
import random as rd

def main(n, timeLimit):
    adjacency = readInstance(n)
    #Model the problem
    env = skr.Env("license.sio")
    color = [env.categorical(0, n-1) for _ in range(n)]
    for i in range(n):
        for j in adjacency[i]:
            #Enforce compatibility constraints
            env.enforce_neq(color[i], color[j])
    obj = env.max(color)
    #Minimize objective for timeLimit seconds with lower bound 1
    env.minimize(obj, timeLimit, 1)
    print("Found a solution with", int(obj.get_value()), "colors")
    env.end()
```

5 Advanced Modeling Concepts

We now cover a number of concepts that help model more complex problems as well as improve the efficiency with which models can be solved.

5.1 Partitionings and Packings

Many real world problems require the decision how to distribute a set of n items into m sets. A *partitioning* requires the solver to assign each item to exactly one set. A *packing* enforces that items are assigned to no more than one set, but items may also remain left out and not assigned to any set.

To create the respective structures, Seeker$^{(\mathrm{TM})}$ provides two functions:

- **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.

Partitionings and Packings can be used in two different ways. First, a Boolean term can be created for each item and index of the respective collection of partitions or sets, whereby these are enumerated from 0 to *numberOfSets*-1.

- **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*.

An even more convenient way of using partitionings and packings is within the aggregators listed in Section 3.1.9. For each aggregator *agg*, there exists a corresponding *agg_if* which, given a vector of terms, a packing/partition, and an index, only aggregates those terms whose index in the vector corresponds to the item number in the respective partition/set with the given index. For example:

- **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*.

Note that, for more evolved aggregators such as the mean, variance, standard deviation, etc., the resulting term automatically computes the corresponding statistic over the ground set of terms that only correspond to items in the respective partition/set. In case of the mean, e.g., the resulting term divides the sum of all terms that fall into the partition/set by the number of items in that partition/set.

5.1.1 Bin Packing - Continued

The bin packing example from Section 4.1 can be modeled, both more conveniently and more efficiently, using a partitioning.

Bin Packing with Partitions in C++:

```
#include "env.hpp"
#include <vector>
#include <math.h>
#include <iostream>

using namespace std;
using namespace Seeker;

void binPack(int numberOfItems, int numberOfBins,
             int binCapacity, int* items, double timelimit) {
  //build instance
  double totalWeight=0;
  for (int i=0; i<numberOfItems; i++)
    totalWeight+=items[i];
  int lowerBound=ceil(totalWeight/binCapacity);
  //create model
  Env env("my_license.sio");
  vector<Term> itemsData=env.convert(items);
  Partition part=env.partition(numberOfBins,numberOfItems);
  vector<Term> binSize(numberOfBins);
  vector<Term> binUsed;
  for (int s=0; s<numberOfBins; s++) {
    binSize[s]=env.sum_if(itemsData,part,s);
    env.enforce_leq(binSize[s],binCapacity);
    binUsed.push_back(env.if_(binIdent[s],1,0));
  }
  Term obj=env.sum(binUsed);
  cout << "Calling Minimization!\n";
  //minimize for tl seconds
  env.minimize(obj,timelimit,lowerBound);
  //read out solution
  bool feasible=true;
  for (int s=0; s<numberOfBins; s++)
    if (binSize[s].get_value()>binCapacity) feasible=false;
  if (feasible) cout << "Number of Bins used = "
                     << obj.get_value() << endl;
  else cout << "No feasible solution found\n";
  env.end();
}
```

Bin Packing with Partitions in Python:

```
import seeker as skr
import math
import numpy as np

def bin_pack(numberOfItems, numberOfBins,
             binCapacity, items, timelimit):
  #build instance
  totalWeight = np.sum(items)
  lowerBound = math.ceil(totalWeight / binCapacity)
```

```
  #create model
  env = skr.Env("license.sio")
  itemsData = env.convert(items)
  part = env.partition(numberOfBins, numberOfItems)
  binSize = [env.sum_if(itemsData,part, s)
                for s in range(numberOfBins)]
  binUsed = [binSize[s]>0 for s in range(numberOfBins)]
  for s in range(numberOfBins):
    env.enforce_leq(binSize[s], binCapacity)
  obj = env.sum(binUsed)
  print("Calling Minimization!")
  #minimize for tl seconds with lower bound
  env.minimize(obj, timelimit, lowerBound)
  #read out solution
  feasible = True
  for s in range(numberOfBins):
      if (binSize[s].get_value() > binCapacity): feasible = False
  if (feasible):
      print("Number of Bins used =", obj.get_value())
  else:
      print("No feasible solution found")
  env.end()
```

5.1.2 Discrete Capacitated Facility Location - Continued

Using partitionings, we can reformulate the discrete capacitated facility location problem from Section 2.2. The reformulation is both more compact and can be solved more efficiently.

Discrete Capacitated Facility Location with Partitions in C++:

```
#include <math.h>
#include <iostream>
#include <string>
#include <vector>
#include <env.hpp>
#include <term.hpp>

using namespace std;
using namespace Seeker;

extern void readFile(string fileName, int& l, int& c,
                     vector<double>& locCost,
                     vector<double>& locCapacity,
                     vector<vector<double> >& servCost,
                     vector<double>& custDemand);

int main(int argc, char** argv) {
  // gather instance data
  string inputFile(argv[1]);
  // l=numberOfLocations, c=numberOfCustomers
  int l,c;
  vector<double> locCost;
  vector<vector<double> > servCost;
  vector<double> custDemand;
  vector<double> locCapacity;
```

```
  readFile(inputFile,l,c,locCost,locCapacity,servCost,custDemand);

  // Create environment
  Env env("my_license.sio");
  // create customer partition
  Partition part=env.partition(l,c);
  vector<Term> custDemandTerms;
  for (int i=0; i<c; i++)
    custDemandTerms.push_back(Term(env,custDemand[i]));
  vector<vector<Term> > servCostTerms(l);
  for (int j=0; j<l; j++)
    for (int i=0; i<c; i++)
      servCostTerms[j].push_back(Term(env,servCost[j][i]));
  // compute the resulting load per location
  vector<Term> locLoad;
  for (int j=0; j<l; j++) {
    locLoad.push_back(env.sum_if(custDemandTerms,part,j));
    // enforce the load per location doesnt exceed its capacity
    env.enforce_leq(locLoad[j],locCapacity[j]);
  }
  // compute costs of opening the locations that are being used
  vector<Term> locUtilizationCost;
  for (int j=0; j<l; j++)
    locUtilizationCost.push_back(env.if_(locLoad[j]>0,
                                         locCost[j],0));
  Term locationCosts=env.sum(locUtilizationCost);
  //compute the cost of servicing the customers
  //at respective locations
  vector<Term> locServCosts;
  for (int j=0; j<l; j++)
    locServCosts.push_back(env.sum_if(servCostTerms[j],part,j));
  Term assignmentCosts=env.sum(locServCosts);
  //assign lower bound on costs to 0
  double tl=atoi(argv[2]), bound=0;
  // define objective term
  Term totalCosts=locationCosts+assignmentCosts;
  // minimize
  env.minimize(totalCosts,tl,bound);
  //read final cost value
  cout << "Optimized Costs = " << totalCosts.get_value() << endl;
  env.end();
  return 0;
}
```

Discrete Capacitated Facility Location with Partitions in Python:

```
def cfl(instance):
    # gather instanc data
    l, c, locCost, locCapacity, servCost, custDemand =\
        readInstance(instance)

    # Create environment
    env = skr.Env("license.sio")
    # create customer partition
    part = env.partition(l, c)
    custDemandTerms = env.convert(custDemand)
    servCostTerms = env.convert(servCost)
    # compute the resulting load per location
```

```
    locLoad = [env.sum_if(custDemandTerms, part, j)
               for j in range(l)]
    for j in range(l):
        # enforce the load per location doesnt exceed its capacity
        env.enforce_leq(locLoad[j], locCapacity[j])
    # compute costs of opening the locations that are being used
    locUtilizationCost = [(locLoad[j] > 0) * locCost[j]
                          for j in range(l)]
    locationCosts = env.sum(locUtilizationCost)
    # compute the cost of servicing the customers
    # at respective locations
    locServCosts = [env.sum_if(servCostTerms[j], part, j)
                    for j in range(l)]
    assignmentCosts = env.sum(locServCosts)
    # define objective term
    totalCosts = locationCosts + assignmentCosts
    # minimize with lower bound 0
    env.minimize(totalCosts, timeLimit, 0)
    # read final cost value
    print("Optimized Costs =", totalCosts.get_value())
    env.end()
```

5.2 Efficient Boolean Indicators

A crucial modeling tool are efficient Boolean indicators. It is often convenient to make a term dependent of the value of another. We can achieve this, semantically correctly, by using if-conditions. Using the Boolean indicators presented in this section, we can greatly improve computational efficiency, though. The trick here consists in exploiting that a term having one value obviously cannot also be equal to another value. The Seeker$^{(TM)}$ Solver does this automatically when the model uses the concepts Seeker::SwitchCondition and Seeker::IntervalCondition.

5.2.1 Switch Conditions

The SwitchCondition class allows to define aggregators or derive Boolean values from a conditioned Term. Seeker::Env provides two functions for creating SwitchCondition objects:

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

The SwitchCondition object created can be used in two ways. The first is to create an (efficient) Boolean term:

- **Term Env::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.

Consider this excerpt of a larger Seeker$^{(TM)}$ model:

```
// ...
Term a = env.ordinal(0,3);
Term b = env.ordinal(0,3);
Term c = env.categorical(0,2);
vector<Term> matches(5);
matches[0] = a+b;
matches[1] = 2*a+b;
matches[2] = a+2*b;
matches[3] = a-b;
matches[4] = env.abs(matches[3]);
SwitchCondition switch = env.switch_condition(c,matches);
vector<Term> bools(5);
for (int i=0; i<5; i++) {
    bools[i] = env.boolean(switch, i);
}
// ...
```

We notice a number of things.

1. The conditional variable 'c' can only take 3 different values, yet when creating the SwitchCondition object, we provide 5 potential matches. There can be fewer, equal, or more potentially matching terms when creating the switch condition than the underlying conditional term can take values.

2. We notice that the matching terms can take values outside of the range of the conditional term 'c.' For example, when a=b=3, matches[0]=6 which is outside of the range of 'c.'

3. Several matching terms may have the same value, depending on the choice of decision variable assignment. For example, when a=b=0, all five matching terms have value 0.

All of the above is allowed. However, the user **must** ensure that, when creating the Boolean derivatives, the case index is between 0 and 4 (0 and the number of matches minus 1 in general).

Let us consider some settings for the variables a, b, and c and see the resulting values in the bools vector. The easiest way to think about the 'bools' vector is this:

$$bools = [c == a + b, c == 2 * a + b, c == a + 2 * b, c == a - b, c == |a - b|]$$

$$a = b = c = 0 \rightarrow bools = [1, 1, 1, 1, 1]$$

$$a = b = 0, c = 2 \rightarrow bools = [0, 0, 0, 0, 0]$$
$$a = 0, b = 1, c = 1 \rightarrow bools = [1, 1, 0, 0, 1]$$
$$a = 2, b = 1, c = 1 \rightarrow bools = [0, 0, 0, 1, 1]$$
$$a = 1, b = 1, c = 3 \rightarrow bools = [0, 1, 1, 0, 0]$$

Using the second method for obtaining a switch condition is simply a shorthand for specifying matching terms as below:

```
// ...
vector<int> values(n);
for (int i=0; i<n; i++) values[i]=i;
vector<Term> matches=env.convert(values);
// ...
```

The second way to use switch conditions is in combination with aggregators. Every Seeker$^{(TM)}$ aggregator 'agg' comes with a method 'agg_if' which takes, on top of the terms to be aggregated, a vector of switch conditions and a vector of case indices.

For example:

- **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.

For the case when all switch conditions are to match the same case (note: this may **still** be a different matching value for each condition!), there is a more convenient aggregation method:

- **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.

Bin Packing with Switch Conditions in C++:

```
#include "env.hpp"
#include <vector>
#include <math.h>
#include <iostream>

using namespace std;
using namespace Seeker;

void binPack(int numberOfItems, int numberOfBins,
             int binCapacity, int* items, double timelimit) {
  //build instance
  double totalWeight=0;
```

```
  for (int i=0; i<numberOfItems; i++)
    totalWeight+=items[i];
  int lowerBound=ceil(totalWeight/binCapacity);
  //create model
  Env env("my_license.sio");
  vector<Term> itemsData=env.convert(items);
  vector<Term> assignment(numberOfItems);
  vector<SwitchCondition> conditions(numberOfItems);
  for (int i=0; i<numberOfItems; i++) {
    assignment[i] = env.categorical(0,numberOfBins-1);
    conditions[i] = env.switch_condition(assignment[i],
                                         numberOfBins);
  }
  vector<Term> binSize(numberOfBins);
  vector<Term> binUsed;
  for (int s=0; s<numberOfBins; s++) {
    binSize[s]=env.sum_if(itemsData,conditions,s);
    env.enforce_leq(binSize[s],binCapacity);
    binUsed.push_back(env.if_(binIdent[s],1,0));
  }
  Term obj=env.sum(binUsed);
  cout << "Calling Minimization!\n";
  //minimize for tl seconds
  env.minimize(obj,timelimit,lowerBound);
  //read out solution
  bool feasible=true;
  for (int s=0; s<numberOfBins; s++)
    if (binSize[s].get_value()>binCapacity) feasible=false;
  if (feasible) cout << "Number of Bins used = "
                     << obj.get_value() << endl;
  else cout << "No feasible solution found\n";
  env.end();
}
```

Bin Packing with Switch Conditions in Python:

```
import seeker as skr
import math
import numpy as np

def bin_pack(numberOfItems, numberOfBins,
             binCapacity, items, timelimit):
  #build instance
  totalWeight = np.sum(items)
  lowerBound = math.ceil(totalWeight / binCapacity)
  #create model
  env = skr.Env("my_license.sio")
  itemsData = env.convert(items)
  assignment = [env.categorical(0,numberOfBins-1)
                for _ in range(numberOfItems)]
  conditions = [env.switch_condition(assignment[i], numberOfBins)
                for i in range(numberOfItems)]
  binSize = []
  binUsed = []
  for s in range(numberOfBins):
    binSize.append(env.sum_if(itemsData, conditions, s))
    env.enforce_leq(binSize[s], binCapacity)
    binUsed.append(env.if_(binSize[s]>0, 1, 0))
```

```
  obj = env.sum(binUsed)
  print("Calling Minimization!")
  #minimize for tl seconds with lower bound
  env.minimize(obj, timelimit, lowerBound)
  #read out solution
  feasible = True
  for s in range(numberOfBins):
      if (binSize[s].get_value() > binCapacity): feasible = False
  if (feasible):
      print("Number of Bins used =", obj.get_value())
  else:
      print("No feasible solution found")
  env.end();
```

5.2.2 Interval Conditions

For many applications, it can be useful to check if a given term falls into one or more given intervals. Based on our discussion of switch conditions above, interval conditions are simply the analogous concept for "falls into the interval" instead of "is equal to the term."

- **IntervalCondition Env::interval_condition(Term cond):** Returns an IntervalCondition object for the Term 'cond.'

The IntervalCondition object created can be used in two ways. The first is to create an (efficient) Boolean term:

- **Term Env::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].

Let us consider a short excerpt of a model using interval conditions.

```
// ...
vector<double> lows =  {-0.1, 0.2, -0.4};
vector<double> highs = {0.4, 0.3, 0.6};
Term x = env.continuous(0, math.pi);
Term switch = env.interval_condition(env.cos(x));
vector<Term> bools(3);
for (int i=0; i<3; i++) {
    bools[i] = env.boolean(switch, lows[i], highs[i]);
}
// ...
```

This code snippet defines a vector of Booleans terms:

$$bools = [cos(x) \in [-0.1, 0.4], cos(x) \in [0.2, 0.3], cos(x) \in [-0.4, 0.6]]$$

Once more, we gain efficiency provided that the condition term (in our case, $cos(x)$) has to be checked against membership in several fixed-size intervals.

The second way of using interval conditions is by using them in 'agg_if' definitions of aggregation terms. For example:

- **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]].

For the case when all interval conditions are to fall into the same interval (note: we will only gain any efficiency in this case provided that there are other aggregators using the same interval conditions but different intervals), there is a more convenient aggregation method:

- **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].

Consider the excerpt of a machine scheduling model where we need to enforce that no more than a given load is executed on a machine at the same time.

```
// ...
vector<Term> loads = env.convert(taskLoads);
vector<Term> taskStart(numTasks);
vector<IntervalCondition> taskConds(numTasks);
for (int i=0; i<numTasks; i++) {
    assert(maxTime>=duration[i]);
    taskStart[i] = env.ordinal(0, maxTime - duration[i] + 1);
    taskConds[i] = env.interval_condition(taskStart[i]);
}
for (int t=0; t<=maxTime; t++) {
    vector<double> lows(numTasks);
    vector(double> highs(numTasks);
    for (int i=0; i<numTasks; i++) {
        lows[i] = t - duration[i] + 1;
        highs[i] = t;
    }
    Term load = env.sum_if(loads, taskConds, lows, highs);
    env.enforce_leq(load, maxLoad);
}
//...
```

5.3 Permutations and Convex Combinations

Apart from categorical, ordinal, and continuous variables, Seeker$^{(TM)}$ also provides permutations and convex combinations.

5.3.1 Permutations

The environment method

- **Permutation Env::permutation(int n)**

creates a Permutation object from which two different sets of ordinal decision variables, which each form a permutation, can be derived using the functions

- **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 the first function.

The vector returned by the second function is often very helpful when the new position of an element under the first permutation needs to be found quickly. A classical example for using permutations is in the context of the travelling salesperson problem.

```
// ...
vector<vector<Term> > distance = env.convert(distanceMatrix);
Permutation tourObject = env.permutation(n);
vector<Term> tour = tourObject.get_permutation();
vector<Term> legDistance(n);
for (int i=0; i<n; i++)
    legDistance[i] = env.index(tour[i], tour[(i+1) % n], distance);
Term tourLength = env.sum(legDistances);
env.minimize(tourLength, 60, 0);
// ...
```

5.3.2 Convex Combinations

Using the environment method

- **vector<Term> Env::convex_combination(int n)**

we can easily create a set of n non-negative continuous decision variables which always sum to 1 (which implies that each variable lives in the interval $[0, 1]$). This construct is more computationally efficient than creating n non-negative continuous variables and adding a constraint that their sum must equal 1.

5.4 Sortings and Quantiles

Another powerful concept that Seeker$^{(\mathrm{TM})}$ provides are sortings and quantiles which can be used to formulate highly complex problems with great simplicity.

5.4.1 Sortings

Seeker$^{(\mathrm{TM})}$ provides three different types of sortings, for terms whose values are rounded to the nearest integers first, for floating point valued terms, and partial sortings in case only the top k or bottom k values need to be computed in order. The environment functions for creating the corresponding Sorting objects are:

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

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

- **Sorting Env::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.

The Sorting object returned by the function can be used to obtain three different vectors of terms:

- **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

The following code excerpt illustrates use and semantic invariants.

```
// ...
int n = inputNumbers.size();
vector<Term> numbers = env.convert(inputNumbers);
Sorting sorter = env.float_sorting(numbers);
vector<Term> sort = sorter.get();
vector<Term> originalPosition = sorter.get_permutation();
vector<Term> rank = sorter.get_permutation_inverse();

cout << "Min: " << sort[0].get_value() << endl;
cout << "Max: " << sort[n-1].get_value() << endl;
cout << "ArgMin: " << originalPosition[0].get_value() << endl;
cout << "ArgMax: " << originalPosition[n-1].get_value() << endl;
cout << "The third number in the original list is the "
     << rank[2].get_value() << "th smallest number overall."
// ...
```

Finally, please note that the vectors returned by a Sorting object that was created using the 'partial_sorting' environment function all have the same length as the original input vector 'terms.' The original values will be sorted up to the 'k'th smallest (or largest, if 'maximize'=true), all other values, positions, and ranks will be arbitrary and may not reflect the original ordering.

5.4.2 Quantiles

Quantiles are the natural counterpart to sortings, but can be more efficient provided that we only need to keep track of one or very few values, such as, for example, the median, or the 25% quantile of a list of terms. When you need to track a lot of these values, using sortings may be more computationally efficient.

Seeker$^{(TM)}$ provides four different functions for tracking quantiles:

- **vector<Term> Env::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' may vary but must always evaluate to an integer number in $\{0, 1, \ldots, n-1\}$ when n is the length of the vector 'terms.' If the value for 'i' is fixed, you may also use

- **vector<Term> Env::ith_smallest_value(int i, vector<Term> terms):** As above, but with a constant index 'i'.

The use of the functions 'ith_largest_value' follows analogously.

5.5 Logic Combinations of Constraints

In Section 3.1.4, we previously introduced ways to create and directly enforce constraints. Alternatively, you can also first create constraints, create logical combinations of constraints, and then enforce these.

Note: The creation of a constraint has no effect on the model. To be considered by Seeker$^{(TM)}$, the constraint must be posted to the model fist by using "Env::enforce"!

5.5.1 Constraint Generation

Seeker$^{(TM)}$ provides the following methods for creating constraints:

- **Constraint Env::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 Env::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 Env::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 Env::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 Env::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 Env::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 $l \neq r$.

Seeker$^{(TM)}$ provides the following functions for the creation of logic combinations of constraints:

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

Finally, to post a constraint to the environment, use

- **void Env::enforce(Constraint constraint):** 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.

5.5.2 2D Rectangle Packing in Python

We demonstrate the use of logic combinations of constraints using the example of 2-dimensional packing of rectangles. Note how the constraints that no two rectangles overlap is realized by creating disjunctions of lower-or-equal constraints.

```
import seeker as skr
import numpy as np

numberSquares = 20
np.random.seed(13)
widths = [np.random.rand() / 2 for _ in range(numberSquares)]
heights = [np.random.rand() / 2 for _ in range(numberSquares)]

env = skr.Env("license.sio")
x = [env.continuous(0, numberSquares)
       for _ in range(numberSquares)]
y = [env.continuous(0, numberSquares)
       for _ in range(numberSquares)]
rotate = [env.categorical(0, 1) for _ in range(numberSquares)]
w = env.convert(widths)
h = env.convert(heights)
trueW = [env.if_(rotate[i], h[i], w[i])
           for i in range(numberSquares)]
trueH = [env.if_(rotate[i], w[i], h[i])
           for i in range(numberSquares)]
rightx = [x[i] + trueW[i] for i in range(numberSquares)]
top = [y[i] + trueH[i] for i in range(numberSquares)]
minWidth = env.min(x)
maxWidth = env.max(rightx)
minHeight = env.min(y)
maxHeight = env.max(top)
areaWidth = maxWidth - minWidth
areaHeight = maxHeight - minHeight
totalArea = areaWidth * areaHeight
```

```
for i in range(numberSquares - 1):
    for j in range(i + 1, numberSquares):
        above = env.geq(y[i], top[j])
        below = env.leq(top[i], y[j])
        left = env.leq(rightx[i], x[j])
        right = env.geq(x[i], rightx[j])
        no_overlap = env.or_([above, left, right, below])
        env.enforce(no_overlap)

env.minimize(totalArea, 60)
print("Total Area", totalArea.get_value())
env.end()
```

5.6 User-Defined Terms

Sometimes, rather than modelling all effects that the decisions to be optimized would have using Seeker(TM)-provided terms, it can sometimes be more convenient to define your own terms.

5.6.1 Creating New Terms and Functions

To create a new term or a new function, we first need to create a class that will be derived from a Seeker(TM) base class and which will overload certain functions that Seeker(TM) needs so that it can perform its optimization.

To create new term, i.e., a function that takes a number of values as input and outputs a single new value, we will derive from the base class "UserEvalTerm."

```
class UserEvalTerm
{
public:
  UserEvalTerm(void) {}
  virtual double evaluate(std::vector<double> values) = 0;
  virtual double inc_evaluate(std::vector<int> valueIndices,
                              std::vector<double> oldValues,
                              std::vector<double> values);
  virtual bool incremental(void) const = 0;
};
```

As can be seen from this declaration, when deriving a new class from "UserEvalTerm," the user must overload the functions "evaluate" and "incremental." In "evaluate," the user implements the desired function. As input, the inputs to the function are given in a vector. The function then returns the desired output. The implementation of "incremental" consists simply in returning 'true' or 'false.' This functions informs Seeker(TM) on whether the function "inc_evaluate" was also overloaded by the user. If "incremental" simply returns "false," then nothing else is required.

If "incremental" returns true, then "inc_evaluate" should also implement the function, *but* it receives the inputs slightly differently. Namely, Seeker(TM) tells "inc_evaluate" the indices of all inputs that have changed *when compared to the last call to this function from the same user-defined term.* For each of these

changed inputs, Seeker$^{(TM)}$ then provides the old and the new values. This allows the user to implement a faster, incremental version of their function.

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!

Sometimes, we want our self-defined functions to return an entire vector of values instead of just one. For this case, Seeker$^{(TM)}$ provides the base class "UserEvalVector."

```
typedef std::pair<std::vector<int>,std::vector<double> > IncPair;
class UserEvalVector
{
public:
  UserEvalVector(void) {}
  virtual std::vector<double> recompute(std::vector<double> values)
    = 0;
  virtual IncPair inc_recompute(std::vector<int> valueIndices,
                                std::vector<double> oldValues,
                                std::vector<double> values);
  virtual bool incremental(void) const = 0;
};
```

The use of this base class works exactly analogously to "UserEvalTerm," with the only difference that "inc_recompute" returns a pair of two vectors, the first providing the indices (numbered starting at 0) of outputs that have changed, and the second vector reflecting the corresponding new output values.

5.6.2 Adding User Terms to the Model

After deriving a new class from "UserEvalTerm" or "UserEvalVector," we can now define personalized terms and functions for our Seeker$^{(TM)}$ model. Seeker$^{(TM)}$ provides the following two functions to define your own terms or functions:

- **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.

5.6.3 A Continuous Optimization Example

Given $n \in \mathbb{N}$, we will maximize the following, challenging function over $n+1$ continuous variables $x_0, \ldots, x_n \in [0,1]$:

$$f(x) = \sum_{i=0}^{n-1} x_i \cos((x_i - x_{i+1})^2 - \frac{i+1}{n+1}) - x_{i+1} \arccos(|x_i - x_{i+1}| - \frac{i+1}{n+1}).$$

```
import seeker as skr
from math import cos, acos, fabs
from numpy import random

class MyTerm(skr.UserEvalTerm):
    def __init__(self, alpha):
        skr.UserEvalTerm.__init__(self)
        self.alpha = alpha
    def incremental(self):
        return False
    def evaluate(self, value):
        x1 = value[0]
        x2 = value[1]
        diff = fabs(x1 - x2)
        return x1 * cos(diff * diff - self.alpha)
               - x2 * acos(diff - self.alpha)

n = 30
alpha = [(i + 1) / (n + 1) for i in range(n)]
env = skr.Env("license.sio")
x = [env.continuous(0, 1) for _ in range(n + 1)]
summands = [env.user_defined_term([x[i], x[i + 1]],
                                  MyTerm(alpha[i]))
            for i in range(n)]
obj = env.sum(summands)
env.set_report(5)
env.set_exploration_size(0.2, 0.7)
env.set_restart_likelihood(0.01)
env.maximize(obj, 60)
print("X", [x[i].get_value() for i in range(n)])
print("Obj", obj.get_value(), flush=True)
env.end()
```

6 Multi-Objective Optimization

Seeker$^{(TM)}$ offers an optimization mode for true, non-hierarchical multi-objective optimization. In the literature, this often results in providing a Pareto frontier of (near) non-dominated solutions that the user can then mine further. However, this is typically not what is needed in deployed applications. Instead, a single solution from the frontier needs to be provided which achieves a reasonable trade-off between all key performance metrics which all have equal priority.

To provide such an important practical functionality, there is obviously a need for the user to provide additional information which allows Seeker$^{(TM)}$ to judge when trading a loss in one objective is worth accepting in return for a gain in another. Seeker$^{(TM)}$ therefore requires the user to provide two thresholds for each objective.

The first threshold tells Seeker$^{(TM)}$ when the respective objective function has reached a "fair" performance. The solver will first strive to get all objectives to achieve this level of performance.

The second threshold tells Seeker$^{(TM)}$ when the respective objective has reached "excellent" performance. From here on out, an increase in performance is still desirable, but a further increase results in diminished additional benefits for the use case.

The multi-objective optimization function then basically works in the same manner as the 'minimize' and 'maximize' functions from Section 3.1.3.

- **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.

```
import seeker as skr
env = skr.Env("license.sio")
x = env.continuous(0, 2, 2)
y = env.continuous(0, 3, 1)
env.enforce_leq(2*x+y, 5)
env.enforce_leq(y-x, 1)
objs = [env.sqrt(env.sqr(x)+env.sqr(y)), -3*x-y]
fair = [1.5, -5]
excellent = [3, -8]
directionMax = [True, False]
env.multi_objective(objs, fair, excellent, directionMax, 1)
print("objs", [o.get_value() for o in objs])
print("X", x.get_value(), " Y", y.get_value())
env.end()
```

7 Nested Linear Optimization

Frequently, optimization problems become computationally tractable after some decisions have been taken. Think, e.g., of a network design problem where we first decide on what nodes to open, and the we need to route some minimum-cost flow. In that case, it can be more efficient to use a nested linear optimization term to set the remaining variables.

7.1 Linear Programming Terms

Seeker$^{(TM)}$ provides an integrated linear programming solver that can be used to solve nested linear programming problems. The corresponding function is:

- **LP Env::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$^{(TM)}$ returns an LP object from which various terms can be derived.

Note that the function above takes vectors of Terms as input, which allows the user to make the objective, the bounds, and even the constraint matrix dependent on decisions to be optimized. The LP class provides the following functions:

- **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.

7.2 Linear Constraints Doubling in Seeker(TM)

Unless the linear program is always soluble with an optimal solution, it will be important to inform Seeker(TM) when this is not the case. Typically, this means to forbid that the LP status if anything but 1. However, just posting this constraint to Seeker(TM) does not inform the solver when it is getting closer to setting the other variables in the model in such a way that the LP becomes more feasible. To give the solver this kind of gradient-like information, it is strongly suggested to post constraints $L_r \leq Ax \leq U_r$ as well as $L_v \leq x \leq U_v$ to Seeker(TM) directly, where L_r, U_r represent the row bounds, and L_v, U_v represent the variable bounds.

7.3 Non-Linear Optimization With Nested LP

Consider the following problem: Maximize α + y such that

$$\frac{\alpha}{2} + \frac{\alpha}{\beta} + \beta \leq 5$$

$$-\alpha x + y \leq \frac{1}{\alpha}$$

$$2 - \frac{\beta}{3}x + y \leq 4$$

with $\alpha, \beta \in [1, 5]$ and $x, y \in [0, 5]$.

We note that, once α and β are set, the remaining problem is a simple linear program. We can therefore formulate this problem as follows:

```
import seeker as skr
env = skr.Env("license.sio")
alpha = env.continuous(1, 5)
beta = env.continuous(1, 5)
lpObj = env.convert([0, 1])
lpVarBounds = env.convert([0, 5, 0, 5])
noBound = env.convert(-1e5)
lpRowBounds = [noBound, 1 / alpha, noBound, env.convert(4)]
lpMatrix = [[-alpha, env.convert(1)],
            [2 - beta / 3, env.convert(1)]]
```

```
lp = env.lp(lpObj, lpVarBounds, lpRowBounds, lpMatrix, True)
env.enforce_eq(lp.get_solution_status(), 1)
obj = lp.get_objective() + alpha
env.enforce_leq(alpha / 2 + alpha / beta + beta, 5)
env.maximize(obj, 1)
print("Alpha", alpha.get_value(), " Beta", beta.get_value())
lpSolution = lp.get_solution()
print("LP Solution", [v.get_value() for v in lpSolution])
print("Profit", obj.get_value())
print(env.get_number_evaluations())
env.end()

In this example, and in general, please note that the linear
    program can and will be called with all kinds of settings to $\
    alphe$ and $\beta$, {\bf including infeasible settings} that
    violate the \Seeker\ constraints on these variables.
```

8 Stochastic Optimization

One of the hallmarks of Seeker$^{(TM)}$ is its ability to optimize decisions given uncertain data.

8.1 Environment Creation and Stochastic Parameters

To use the stochastic version of Seeker$^{(TM)}$, simply set the Boolean flag when creating the environment:

- **Env::Env(string license, bool stochastic)**: Constructor. Creates the Seeker$^{(TM)}$ environment, provided a valid license file name is given. When "stochastic" is "true," an environment is created that handles stochastic data in models as well.

There are two parameters that govern the resolution and evaluation speed of stochastic models. To set them, use the following function:

- **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).

8.2 Creating Stochastic Data Terms

To add stochastic data to your model, Seeker$^{(TM)}$ provides the following functions:

- **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$^{(\mathrm{TM})}$ 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."

8.3 User-Defined Distributions

The above functions generate some of the most commonly used random Terms. However, there are times when our model depends on data that we observe in the world which appears to be generated from a different distribution. In this

case, we may wish to use the historic data directly, or derive a statistical model from this data, which allows us to sample from the inferred distribution.

Moreover, we often face a situation where multiple stochastic data points needed for our optimization model are not independent but somewhat correlated. In this case, we want Seeker$^{(TM)}$ to consider "scenarios" where the values for these respective data points are drawn from a joint distribution, rather than independently from one another.

As these needs arise, Seeker$^{(TM)}$ provides the following two functions.

- **Term Env::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" (see Section 8.1).

- **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" (see Section 8.1).

8.4 Solution-Dependent Distributions

For some real-world problems, the stochasticity of some data Terms may depend on the very decisions that we are aiming to optimize. Consider the example where the demand for a product stochastically depends on the price we set for this product, which is one of the decision variables in the model. Rather than using a deterministic function for deriving the demand from the price, Seeker$^{(TM)}$ allows the user to express that the demand is, in fact, an estimate that comes with uncertainty. Seeker$^{(TM)}$ provides the following two functions:

- **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.

  The user needs to create a class that derives from UserStochTerm and overload a single function vector<double> sample_n(vector<double> features, int n) which will return n samples from the posterior distribution given the feature values. The maximum number n that Seeker$^{(TM)}$ will ask for is given by the parameter "resolution" (see Section 8.1).

- **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 distribution "usv" which depends on the current values of the *deterministic* terms in "features." Note: The terms in "features" cannot themselves be stochastic terms.

  The user needs to create a class that derives from UserStochVector and overload a single function vector<vector<double>> sample_n(vector<double> features, int n) which will return n vector samples from the joint posterior distribution given the feature values. The maximum number n that Seeker$^{(TM)}$ will ask for is given by the parameter "resolution" (see Section 8.1).

8.5 Aggregation

Terms generated by any of the functions above are stochastic. This means that, within Seeker$^{(TM)}$, they do not have one deterministic value. You can use these terms to derive new terms, for example by using term operators or term aggregators. It does not matter whether the terms that a stochastic term is combined with are themselves stochastic or not. However, the result of the combination will always be a non-deterministic, stochastic term.

Now, terms that appear as part of constraints and/or the objective function are required to be deterministic. This means that we will require methods to turn a stochastic term into a deterministic term. This is achieved by aggregation. The most common way of aggregating is by considering the expected value of a stochastic term, but there are many more. In the following, we list the aggregators that Seeker$^{(TM)}$ provides:

- **Term Env::aggregate_mean(Term source):** Returns the estimated arithmetic mean value of the stochastic term "source."
- **Term Env::aggregate_geometric_mean(Term source):** Returns the estimated geometric mean of the stochastic term "source."
- **Term Env::aggregate_variance(Term source):** Returns the estimated variance of the stochastic term "source."
- **Term Env::aggregate_aav(Term source):** Returns the estimated mean of the absolute values of the stochastic term "source."
- **Term Env::aggregate_rmsv(Term source):** Returns the root of the estimated mean of the square values of the stochastic term "source."
- **Term Env::aggregate_stdev(Term source):** Returns the estimated standard deviation of the stochastic term "source."
- **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.

- **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."

- **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."

- **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."

- **Term Env::aggregate_min(Term source):** Returns the estimated minimum value of the stochastic term "source."

- **Term Env::aggregate_max(Term source):** Returns the estimated maximum value of the stochastic term "source."

- **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.

- **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.

9 Stochastic Optimization Examples

We demonstrate the use of the stochastic modelling capabilities of Seeker$^{(TM)}$ by means of a couple of examples.

9.1 Monty Hall Problem

In the famous Monty Hall problem, a candidate in a game show is asked to pick one of three possible doors. Behind one of them is a car, behind the other two a goat. After picking a door, one door with a goat is opened, and the candidate is offered to switch the door. Should they switch or not?

```
import seeker as skr
env = skr.Env(license="license.sio", stochastic=True)
car = env.discrete_uniform(1, 3)
guess = env.discrete_uniform(1, 3)
switch = env.categorical(0, 1)
correctGuess = car == guess
win = correctGuess * switch.not_() + correctGuess.not_() * switch
winRate = env.aggregate_mean(win)
env.maximize(winRate, 0.1)
print("Win Rate", winRate.get_value())
print("Switch Decision", switch.get_value())
env.end()
```

9.2 Betting

Working in a store, clients arrive according to a Poisson process with $\lambda = 5$. You bet with your co-worker that you can call out when the last client walks into the store before the store closes. If you win, you get $10 from them, if you lose, you give the same to them.

Your strategy is to pick a length of time before the store closes. You will call out the first client who enters the store after that time as the last client. How long should the time interval be, and what is your chance to win?

```
import seeker as skr
env = skr.Env(license="license.sio", stochastic=True)
env.set_stochastic_parameters(100, 0)
storeClosure = 100
length = env.ordinal(0, 100)
lambd = 5
clients = 50
occurences = [env.poisson(lambd, storeClosure+1)
              for _ in range(clients)]
arrivals = [env.sum(occurences[:i]) for i in range(1, clients+1)]
intervalArrival = [(arrivals[i]<=storeClosure)*
                   (arrivals[i]>=storeClosure-length)
                   for i in range(clients)]
numberIntervalArrivals = env.sum(intervalArrival)
win = numberIntervalArrivals==1
winRate = env.aggregate_mean(win)
env.maximize(winRate, 1)
print("Win Rate", winRate.get_value())
```

```
print("Length", length.get_value())
env.end()
```

9.3 Integrated Pricing and Production Planning

We present the example of a small bakery that sells buns, pastries, and cake slices. We need to set the prices for each of the three products, as well as the quantities we produce of each. The only production constraint we are facing in this simple example is that we can only invest a certain amount of money to produce our goods.

The price-sensitivity is learnt from data. A Gaussian Process Regressor is trained to provide a posterior distribution of demands for a given set of prices. Note that this regressor takes in all prices and then predicts all demands. This allows learning from the data how sales of different products affect each other, for example because some products are typically only bought in conjunction with certain staple products, or because some products cannibalize their respective demands.

The program below illustrates how to set up Seeker$^{(TM)}$ for such a scenario.

```

import seeker as skr
from seeker import Env
from seeker import UserStochVector

def create_gpr(filename):
    [...]

class Predict(UserStochVector):
    def __init__(self, filename):
        skr.UserStochVector.__init__(self)
        self.gpr, self.mu, self.sigma = create_gpr(filename)
    def sample_n(self, features, n):
        pre = self.gpr.sample_y([(features - self.mu)
                                 / self.sigma], n_samples=n)
        return np.transpose(pre[0]).astype(float)

def main():
    # instance data
    number_of_products = 3
    product_costs = [0.11, 0.33, 0.24]
    max_price = [120, 200, 200]
    max_qua = [5000, 2000, 600]
    production_budget = 600

    # prep data for modeling
    print("Learning price-demand model...")
    filename = 'myTrainingData.csv'
    model = Predict(filename)

    # Seeker model
    env = Env('license.sio', stochastic=True)
    env.set_stochastic_parameters(int(1e4), 0.8)

```

```
    prices = [env.ordinal(0, max_price[i]) * 0.01
                for i in range(number_of_products)]
    quants = [env.ordinal(0, max_qua[i])
                for i in range(number_of_products) ]

    # constraints
    production_costs = [quants[i] * product_costs[i]
                            for i in range(number_of_products)]
    total_production_costs = env.sum(production_costs)
    env.enforce_leq(total_production_costs, production_budget)

    # objective
    predicted_demands = env.user_defined_stochastic_vector(
                            prices,
                            model,
                            number_of_products)
    product_demands = [env.round(env.max_0(pd))
                          for pd in predicted_demands]
    sales = [env.min([quants[i], product_demands[i]])
                for i in range(number_of_products)]
    profit = [sales[i] * prices[i] - production_costs[i]
                for i in range(number_of_products)]
    total_profit = env.sum(profit)
    exp_total_profit = env.aggregate_mean(total_profit)

    # Seeker optimization
    exp_sales = [env.aggregate_mean(s) for s in sales]
    exp_profit = [env.aggregate_mean(p) for p in profit]
    env.set_report(5, exp_profit + exp_sales,
                    ["Buns Profit", "Pstrs Profit"
                        ,"Cakes Profit", "Buns Sales"
                        , "Pstrs Sales", "Cakes Sales"])
    env.set_exploration_size(0.2, 0.7)
    env.set_restart_likelihood(0.007)
    timeout = 120
    print("Optimizing with InsideOpt Seeker for"
            , timeout, "seconds...")
    env.maximize(exp_total_profit, timeout)
    print("Prices", [p.get_value() for p in prices])
    print("Quants", [q.get_value() for q in quants])
    print("Production Costs", total_production_costs.get_value())
    print("Expected Profit", exp_total_profit.get_value())
    print(env.get_number_evaluations())
    env.end()
```

10 Parameters and Tuning

10.1 Automatic Tuning

Seeker$^{(TM)}$ is not a one-size-fits-all solver. In fact, Seeker$^{(TM)}$ implements a massive suite of different search and optimization strategies. By using automatic tuning, this allows you to tailor Seeker$^{(TM)}$ specifically for *your* application.

The actual tuning process is beyond the scope of this user's manual, but we list the function with which the automatic tuner can set the 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 $[1, 100]$.

The above function should be used in combination with an automatic tuner after the model development process is completed and before the model is deployed.

10.2 Manual Tinkering

During the modeling process, it can be helpful to tinker with some interpretable parameters manually. This does not replace the automatic tuning, though, and care must be taken that the calls to manual parameter setting functions do not interfere with the parameters that were found by a tuner. Please make sure to comment out or delete calls to the functions below if you previously called the function above.

That being said, the following functions allow you to manipulate the Seeker$^{(TM)}$ search process.

- **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.

11 Parallel Optimization

In many cases, it will be convenient to run the solver in a distributed setting, where multiple programs churn on the same problem instance. Seeker$^{(TM)}$ makes it extremely easy for you to start as many of your programs using the library at the same time and have them work together. The only requirement is that all programs have access to the same file system.[2]

11.1 Prerequisites

First, you need to prepare your scrap folder. Say your path to your scrap folder is /tmp/seeker/ and your Seeker$^{(TM)}$ library is installed under /usr/lib/seeker.

- In your Seeker install directory (e.g., /usr/lib/seeker), execute this command: "/user/seeker/scripts/installscrap.sh /tmp/seeker". This creates some subfolders under /tmp/seeker (you can change this path too any other directory as well) that are needed to run seeker in parallel. It also sets the environment variable $SeekerPath to /tmp/seeker (or the folder you chose).

11.2 Preparing Your Program

- Change the constructor call to your environment by adding two integer numbers: **Env::Env(string license, int processID, int runID)** or **Env::Env(string license, int processID, int runID, bool stochastic)**.

- 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.**

- The runID is a unique non-negative integer value that identifies the particular problem instance your program is tackling. Use the same runID for all program that you want to churn on the same problem instance. If you solve different instances using the same runID, your runs will exit without solving the problems.

11.3 Adding More Parallel Processes and Crash Recovery

You can add more compute power at any time during the optimization process, and even after, for example, should your hardware have crashed and you want to resume the optimization where it left off.

[2] In case you run a slow system that backs up your data, you may want to use a path to a fast, non-backed up folder as the parallel programs will read and write frequently.

- When adding more compute power while the other processes are running, simply keep increasing the processID numbers with which your program calls the environment.

- After a hardware crash, restart your processes with processIDs starting at 1, making sure not to create a new process that uses the same processID as another process that is still running. Your optimization will then continue where it left off before the hardware failure.

  Note: Be sure not to use processID = 0 when continuing a prior run. As soon as one of your processes calls the environment constructor with processID 0, the slate will be cleaned and your entire parallel optimization will restart from scratch! Moreover, the process with processID 1 handles the corrdination of all processes, so please ensure that exactly ony process with processID = 1 is always running.

11.4 Parallel Coordination Parameters

When starting a process that is part of a parallel/distributed run, Seeker$^{(TM)}$ allows specifying how tightly coupled this process will run together with the other processes. The following function enables setting two parameters that govern the coordination:

- **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.

12 Progress Reports and Search Statistics

We conclude this overview of Seeker$^{(TM)}$ with a list of functions that can help analysing Seeker$^{(TM)}$'s search behavior based on the model provided.

- **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.