RSOME in Python: An Open-Source Package for Robust Stochastic Optimization Made Easy

Zhi Chen[†], Peng Xiong[‡]

[†]Department of Management Sciences, College of Business, City University of Hong Kong

[‡]Department of Analytics & Operations, NUS Business School, National University of Singapore

We introduce a Python package called RSOME for modeling a wide spectrum of robust and distributionally robust optimization problems. RSOME serves as an open-source framework for modeling various optimization problems subject to distributional ambiguity in a highly readable and mathematically intuitive manner. It is versatile and fits well in the open-source software community, in the sense that (*i*) it is consistent with NumPy arrays in indexing and slicing, as well as array operations; (*ii*) together with the rich Python libraries for machine learning, data analysis and visualization, it is easy to implement data-driven models; and (*iii*) it provides convenient interfaces for users to switch and tune parameters among different solvers.

Keywords: (Distributionally) Robust Optimization, Algebraic Modeling Package, Adaptive Decision-Making, Data-Driven Analytics.

1 Introduction

Robust and distributionally robust optimization has developed into one of the most popular paradigms for addressing decision-making in the presence of uncertainty. Typically, uncertainty is modeled as a random variable governed by an ambiguous probability distribution about which we only have partial knowledge. In robust optimization (Ben-Tal and Nemirovski 1998, Bertsimas and Sim 2004, El Ghaoui et al. 1998, Soyster 1973), such partial knowledge is an *uncertainty set* to which the support of uncertainty is confined; while in distributionally robust optimization (Ben-Tal et al. 2013, Chen et al. 2019, Delage and Ye 2010, Goh and Sim 2010, Mohajerin Esfahani and Kuhn 2018, Wiesemann et al. 2014), it is an *ambiguity set* characterized by various types of statistical information—such as support, moments and distance to some reference distributions. We refer to Rahimian and Mehrotra (2019) and references therein for a comprehensive review and recent advances in robust and distributionally robust optimization. Apart from rich characterizations on the partial knowledge about the ambiguous distribution, another notable advantage of robust and distributionally robust optimization models is that for a variety of practical problems and a rich family of uncertainty/ambiguity sets, these models admit reformulations that are "equivalent" to their deterministic counterparts in terms of computational complexity.

Along with the theoretical studies, progress has also been made in developing accompanying technology to facilitate the implementations of new robust and distributionally robust models. Nowadays, commercial optimization solvers (*e.g.*, CPLEX, Gurobi, and MOSEK) and open-source optimization solvers (*e.g.*, CBC from COIN-OR (Martin 2010) and OR-Tools) provide APIs for commonly used programming languages, including C/C++, Java, Python and R. These APIs are typically restricted to formulating deterministic problems, thus users have to manually convert a (distributionally) robust model into its deterministic counterpart before it can be solved. Such a conversion, however, could be rather tedious and error-prone.

Modeling packages, which are different from APIs and rely on external optimization solvers to solve optimization models, provide friendly algebraic modeling environments for specifying models and automatically transforming them into standard forms that solvers recognize. Such packages include, for example, the commercial AIMMS and open-source ones such as CVX (Grant and Boyd 2014) and YALMIP (Lofberg 2004) in MATLAB, as well as JuMP (Dunning et al. 2017) in Julia. It is worth noting that as robust optimization has gained increasing popularity in recent years, AIMMS, YALMIP, and JuMP are all upgraded with new add-ons for building robust optimization models directly; see, for example, the JuMPeR extension that builds on JuMP. However, modeling capabilities of these packages (*e.g.*, JuMPeR) are still largely unexplored, especially in dealing with adaptive decisions or distributionally robust optimization models.

Early attempts in developing modeling packages for more general adaptive (distributionally) robust optimization problems can be traced back to ROME (Goh and Sim 2011) in MATLAB. This work then inspires follow-up theoretical and software developments: for example, Bertsimas et al. (2019) propose a unified framework for adaptive distributionally robust optimization, based on which modeling packages ROC[1] in C++ and XProg[2] in MATLAB are developed. The framework of Bertsimas et al. (2019) has been substantially extended in a more recent work of Chen et al. (2020) to incorporate event-wise distributional information and to devise more flexible event-wise recourse adaptations. The concept of Chen et al. (2020) has been applied in RSOME[3]—a generic algebraic modeling package in MATLAB that encompasses a wide spectrum of (distributionally) robust optimization models with possibly data-driven approaches. More recently, Vayanos et al. (2020) develop an ROC++ package for robust optimization models involving possibly endogenous

[1]ROC website: https://github.com/g0900971/RobustOptimization/.
[2]XProg website: http://xprog.weebly.com/.
[3]RSOME website: https://www.rsomerso.com/.

and binary-valued adaptive decisions,[4] while Isenberg et al. 2020 and Wiebe and Misener (2021) develop (respectively) PyROS and ROmodel to extend the modeling capabilities of the algebraic modeling language Pyomo in Python to robust optimization problems. However, none of these three packages addresses distributionally robust optimization models. The Python package PICOS[5] (Stahlberg 2020) supports a number of robust and distributionally robust optimization models that may impose additional restrictions on the ambiguity set: for example, the current version of PICOS only accommodates the whole real space as support and the L_2-norm as cost function of transport when considering Wasserstein ambiguity.

In this paper, we introduce the Python version of RSOME as a new open-source modeling framework for robust and distributionally robust optimization. To the best of our knowledge, it is the first modeling package in Python that is designed for addressing such a wide variety of (distributionally) robust optimization models. Instead of merely migrating from MATLAB to Python, the new RSOME package in Python is upgraded with the following new features.

1. The package consists of four layers of modules, each of which targets specifically a class of optimization problems. Such a multi-layer structure is not only more efficient in reusing the code and maintaining the project, but also serves as an open-source framework, which could be smoothly upgraded in the future in ways that new modeling frameworks as well as applications are built upon existing modules rather than starting from scratch.

2. RSOME in Python generates robust counterparts in a standard format of a Pandas `DataFrame`,[6] which can be easily processed, analyzed, and exported using analytical tools in the Pandas library. When passing robust counterparts to solvers, RSOME provides interfaces to both the state-of-the-art commercial solvers (such as CPLEX, Gurobi, and MOSEK) and free open-source ones. This benefits users who do not have access to commercial solvers.

3. The RSOME package is more flexible in specifying optimization models. Specifically, compared with the MATLAB version that is restricted to two-dimensional matrices and defines only a single uncertainty/ambiguity set for a given model, the Python package enables users (i) to define variables, functions, and constraints in an arbitrary dimensional array and (ii) to specify different uncertainty/ambiguity sets for the objective function and each constraint.

[4] ROC++ website: https://sites.google.com/usc.edu/robust-opt-cpp/.

[5] Documentation of PICOS: https://picos-api.gitlab.io/picos/index.

[6] We write code segments, objects, and Python packages or modules in monospaced letters (*e.g.*, `code`).

4. The package is freely distributed on GitHub[7] to facilitate version control and collaborative developments. It can be conveniently installed via the third-party software repository PyPi, and it comes accompanied with detailed documentation and application examples.[8]

Main contributions of RSOME in Python lie in being an open-source framework for modeling robust and distributionally robust optimization problems. The package is consistent with existing Python packages in syntax rules and arithmetic operations, thus is friendly to users who have engaged with Python programming. Unlike most of other modeling packages that define variables as scalars, vectors or matrices, RSOME inherits the widely used N-dimensional arrays and their operations (such as indexing, slicing, broadcasting, transpose, as well as element-wise and algebraic computations) in the NumPy libarary. Hence, models in RSOME can be defined with concise and highly readable array expressions. Our package also benefits from the large and active ecosystem of Python programming. It is easy to implement data-driven models using RSOME along with other machine learning and analytical libraries for processing data, tuning hyper-parameters, testing model performance, as well as visualizing results.

The rest of this paper proceeds as follows. In Section 2, we introduce the four-layer structure of RSOME and focus on the top-layer framework for modeling distributionally robust optimization problems. In Section 3, we discuss the logic and basic syntax of building distributionally robust models in RSOME. Features and modeling capabilities of RSOME is demonstrated through a vehicle pre-allocation problem in Section 4. Section 5 concludes our work.

2 Frameworks of RSOME

Unlike the MATLAB version that provides a generic robust stochastic optimization framework for modeling all applicable optimization problems, RSOME in Python is built upon four layers of modules: see an illustration in Figure 1. The lower two layers of RSOME modules provide modeling tools for deterministic linear and second-order cone (SOC) programs, which are the cornerstones for building the upper-level robust and distributionally robust optimization modules. In the Python version of RSOME, all robust counterparts of upper-level robust and distributionally robust models are formulated into the lower-level deterministic problems before being sent to solvers.

[7] RSOME in Python package: https://github.com/XiongPengNUS/rsome/.
[8] RSOME in Python website: https://xiongpengnus.github.io/rsome/.

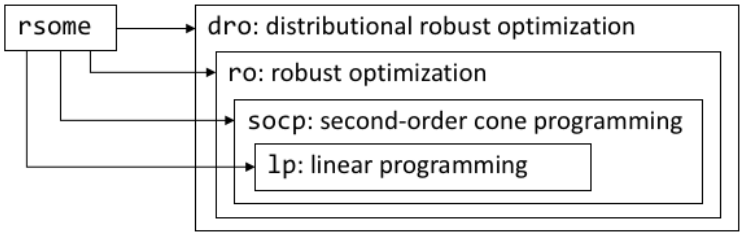

Figure 1. The structure of RSOME modeling frameworks

We note that a higher layer of RSOME module could address a more general class of problems compared with lower-layer ones. The top layer `dro` module for distributionally robust optimization, associated with the event-wise ambiguity sets proposed in Chen et al. (2020), is the most general framework among all. For example, robust optimization problems can be treated as special cases of a distributionally robust optimization problem where the ambiguity set, specifying only the support information, reduces to an uncertainty set; while deterministic problems are special cases of a robust optimization problem whose uncertainty set reduces to a known singleton. The `ro` module, although less general, provides a tailored modeling framework specifically for robust optimization problems, thus it models uncertainty sets and formulates the worst-case objective function as well as robust constraints in a more concise manner. Besides targeting to and tailoring for different types of optimization problems, the multi-layer structure is more effective in reusing existing codes. This benefits the maintenance of the open-source software and also enables easy upgrades in the future since new modeling frameworks could be developed upon the existing layers. Nevertheless, each layer of RSOME modules follows the consistent syntax in defining variables, objective functions, and constraints. In the remainder, we will focus on the most general `dro` framework and we leave detailed discussions on the `ro` framework to an online supplement of this paper.

Models supported by the `dro` module can be cast into the following standard format:

$$
\begin{array}{lll}
\underset{\boldsymbol{x},\,\boldsymbol{y}}{\text{minimize}} & \sup\limits_{\mathbb{P}\in\mathcal{F}_0} \mathbb{E}_{\mathbb{P}}[\boldsymbol{a}_0^\top(\tilde{s},\tilde{\boldsymbol{z}})\boldsymbol{x}(\tilde{s}) + \boldsymbol{b}_0^\top(\tilde{s})\boldsymbol{y}(\tilde{s},\tilde{\boldsymbol{z}}) + c_0(\tilde{s},\tilde{\boldsymbol{z}})] & \\
\text{subject to} & \max\limits_{s\in[S],\,\boldsymbol{z}\in\mathcal{Z}_{ms}} \left\{\boldsymbol{a}_m^\top(s,\boldsymbol{z})\boldsymbol{x}(s) + \boldsymbol{b}_m^\top(s)\boldsymbol{y}(s,\boldsymbol{z}) + c_m(s,\boldsymbol{z})\right\} \le 0 & \forall m\in\mathcal{M}_1 \\
& \sup\limits_{\mathbb{P}\in\mathcal{F}_m} \mathbb{E}_{\mathbb{P}}[\boldsymbol{a}_m^\top(\tilde{s},\tilde{\boldsymbol{z}})\boldsymbol{x}(\tilde{s}) + \boldsymbol{b}_m^\top(\tilde{s})\boldsymbol{y}(\tilde{s},\tilde{\boldsymbol{z}}) + c_m(\tilde{s},\tilde{\boldsymbol{z}})] \le 0 & \forall m\in\mathcal{M}_2 \\
& x_i \in \mathcal{A}(\mathcal{C}_i) & \forall i\in[I_x] \\
& y_i \in \bar{\mathcal{A}}(\bar{\mathcal{C}}_i,\mathcal{J}_i) & \forall i\in[I_y] \\
& \boldsymbol{x}(s)\in\mathcal{X}_s & \forall s\in[S],
\end{array}
$$

where $\tilde{\boldsymbol{z}}$ is a J-dimensional vector of random variables, $\tilde{s}$ is a random scalar indicating the outcome of a random scenario, and vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are decision variables that follow different adaptation schemes denoted by $\mathcal{A}(\mathcal{C})$ and $\bar{\mathcal{A}}(\mathcal{C}_i, \mathcal{J}_i)$, respectively. Here, in a fixed scenario s, $\mathcal{X}_s$ is an SOC representable feasible set of decision variables $\boldsymbol{x}(s)$, and for each $m \in \mathcal{M}_1 \cup \mathcal{M}_2 \cup \{0\}$, $\boldsymbol{b}_m(s) = \boldsymbol{b}_{ms} \in \mathbb{R}^{I_y}$ is a vector of fixed parameters of $\boldsymbol{y}$ while uncertain parameters $\boldsymbol{a}_m(s, \boldsymbol{z})$ as well as $c_m(s, \boldsymbol{z})$ are affine mappings of uncertainty realization $\boldsymbol{z}$:

$$\boldsymbol{a}_m(s, \boldsymbol{z}) = \boldsymbol{a}^0_{ms} + \sum_{j\in[J]} \boldsymbol{a}^j_{ms} z_j \quad \text{and} \quad c_m(\boldsymbol{z}) = c^0_{ms} + \sum_{j\in[J]} c^j_{ms} z_j,$$

where $\boldsymbol{a}^j_{ms} \in \mathbb{R}^{I_x}$ and $c^j_{ms} \in \mathbb{R}$, indexed by $j \in [J] \cup \{0\}$, are proper coefficients.

The hard constraints (with indices $m \in \mathcal{M}_1$) must be satisfied almost surely. The soft constraints (with indices $m \in \mathcal{M}_2$), on the other hand, only needs to be satisfied in expectation over all possible distributions within an *event-wise ambiguity set* introduced by Chen et al. (2020):

$$\mathcal{F}_m = \left\{ \mathbb{P} \in \mathcal{P}_0(\mathbb{R}^J \times [S]) \;\middle|\; \begin{array}{ll} (\tilde{\boldsymbol{z}}, \tilde{s}) \sim \mathbb{P} & \\ \mathbb{E}_{\mathbb{P}}[\tilde{\boldsymbol{z}} \mid \tilde{s} \in \mathcal{E}_{mk}] \in \mathcal{Q}_{mk} & \forall k \in [K] \\ \mathbb{P}[\tilde{\boldsymbol{z}} \in \mathcal{Z}_{ms} \mid \tilde{s} = s] = 1 & \forall s \in [S] \\ \mathbb{P}[\tilde{s} = s] = p_s & \forall s \in [S] \\ \text{for some } \boldsymbol{p} \in \mathcal{P}_m & \end{array} \right\}.$$

Here, for the objective and constraints indexed by $m \in \{0\} \cup \mathcal{M}_2$, conditional expectations of $\tilde{\boldsymbol{z}}$ over events (defined as subsets of scenarios and denoted by, *e.g.*, $\mathcal{E}_{mk} \subseteq [S]$) are known to reside in SOC representable sets $\mathcal{Q}_{m1}, \ldots, \mathcal{Q}_{mK}$, respectively; the support of $\tilde{\boldsymbol{z}}$ in each scenario $s \in [S]$ is another SOC representable set $\mathcal{Z}_{ms}$; and probabilities of scenarios, collectively denoted by a vector $\boldsymbol{p}$, are confined to an SOC representable subset $\mathcal{P}_m \subseteq \{\boldsymbol{p} \in \mathbb{R}^S_{++} \mid \boldsymbol{e}^\top \boldsymbol{p} = 1\}$ in the probability simplex. We would like to note that unlike the standard form proposed by Chen et al. (2020), which accommodates only one ambiguity set, the `dro` framework enables users to define different ambiguity sets for the objective function and constraints.

In the standard form, decisions $\boldsymbol{x}$ and $\boldsymbol{y}$, which are potentially adaptive and can be an arbitrary functional of uncertainty realization $\boldsymbol{z}$ given scenario s, are infinite-dimensional and thus are hard to to optimize. A common technique for tractability, called linear decision rule (or affine decision rule), is to restrict adaptive decisions to simpler and easy-to-optimize affine functions of revealed uncertainty given a scenario s. In RSOME, such dynamics of decision-making are captured by the

event-wise recourse adaptation of two types—the *event-wise static adaptation* $\mathcal{A}(\mathcal{C})$ for decision $\boldsymbol{x}$, and the *event-wise affine adaptation* $\bar{\mathcal{A}}(\mathcal{C},\mathcal{J})$ for decision $\boldsymbol{y}$. In particular, given a fixed number of S scenarios and a partition $\mathcal{C}$ of these scenarios (*i.e.*, a collection of mutually exclusive and collectively exhaustive events), the event-wise recourse adaptation is formally defined as follows:

$$\mathcal{A}(\mathcal{C}) = \left\{ x : [S] \mapsto \mathbb{R} \;\middle|\; \begin{array}{l} x(s) = x^{\mathcal{E}},\ \mathcal{E} = \mathcal{H}_{\mathcal{C}}(s) \\ \text{for some } x^{\mathcal{E}} \in \mathbb{R} \end{array} \right\},$$

$$\bar{\mathcal{A}}(\mathcal{C},\mathcal{J}) = \left\{ y : [S] \times \mathbb{R}^J \mapsto \mathbb{R} \;\middle|\; \begin{array}{l} y(s,z) = y^0(s) + \sum\limits_{j\in\mathcal{J}} y^j(s) z_j \\ \text{for some } y^0(s), y^j(s) \in \mathcal{A}(\mathcal{C}), j \in \mathcal{J} \end{array} \right\}.$$

Here, $\mathcal{H}_{\mathcal{C}} : [S] \mapsto \mathcal{C}$ is a function such that $\mathcal{H}_{\mathcal{C}}(s) = \mathcal{E}$ maps the scenario s to the only event $\mathcal{E}$ in $\mathcal{C}$ that contains s, and $\mathcal{J} \subseteq [J]$ is an index subset of random components $\tilde{z}_1, \ldots, \tilde{z}_J$ that the affine adaptation depends on. Note that in the `dro` framework, we do not differentiate here-and-now decisions (made before any uncertainty realization) and wait-and-see decisions (made after partial uncertainty realization): by default, a here-and-now decision follows a special event-wise static adaptation $\mathcal{A}(\mathcal{C})$, where $\mathcal{C} = \{\{1, \ldots, S\}\}$ is a singleton set of the event $\{1, \ldots, S\}$ that consists of all scenarios. Thus, both here-and-now and wait-and-see decisions are defined by the same function in the `dro` library and their adaptation schemes are then specified respectively.

3 Basics of Modeling in RSOME

The RSOME package is developed based on the large and active open-source ecosystem of Python, wherein it can be easily upgraded and integrated with other Python libraries for various applications. The steps of modeling in RSOME are summarized (with sample code) in Figure 2.

The first step is to create a `dro` model, whereafter decisions and random variables can be defined as components of the model in the second step. Users could also define an ambiguity set of random variables, as well as the event-wise recourse adaptation for decision variables in Step 2. In Step 3, these decisions and random variables are used to specify the objective function and constraints. RSOME enables users to define variables, functions, and constraints as arrays, which are consistent with the widely used NumPy arrays in operations such as indexing, slicing, broadcasting, transpose, and vectorization. Hence, RSOME is friendly to users who have engaged with Python programming

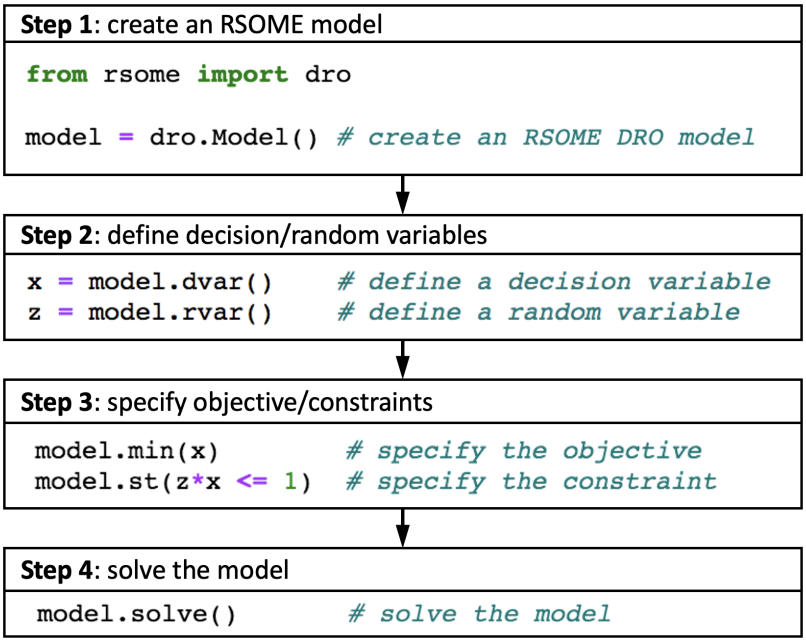

Figure 2. Steps of modeling in RSOME.

and are familiar with the NumPy analytical tools. For example, the constraint system

$$\sum_{i\in[I]} b_i x_i = 1 \tag{1a}$$

$$\sum_{i\in[I]} A_{ji} x_i \le c_j \qquad \forall j \in [J] \tag{1b}$$

$$\sum_{j\in[J]} \sum_{i\in[I]} y_{ji} \ge 1 \tag{1c}$$

$$\sum_{i\in[I]} y_{ji} \ge 0 \qquad \forall j \in [J] \tag{1d}$$

$$y_{ji} \le c_j \qquad \forall j \in [J], i \in [I] \tag{1e}$$

$$A_{ji} x_i \ge 1 \qquad \forall j \in [J],\ i \in [I] \tag{1f}$$

$$A_{ji} y_{ji} + x_i \ge 0 \qquad \forall j \in [J],\ i \in [I] \tag{1g}$$

with decision variables $\boldsymbol{x} \in \mathbb{R}^I$ and $\boldsymbol{y} \in \mathbb{R}^{J\times I}$, as well as parameters $\boldsymbol{A} \in \mathbb{R}^{J\times I}$, $\boldsymbol{b} \in \mathbb{R}^I$ and $\boldsymbol{c} \in \mathbb{R}^J$, can be conveniently specified through highly readable array operations as follows.[9]

[9] Although the example only shows operations on one and two-dimensional arrays, the same NumPy-style syntax applies to arrays of an arbitrary shape or dimension. We refer interested readers to our online example "Integer Programming for Sudoku", where three-dimensional arrays are used.

```
# define variables as arrays
x = model.dvar(I)                    # define x as an I-dimensional array
y = model.dvar((J, I))               # define y as a JxI array

# constraints are specified below
b @ x == 1                           # constraints (1a)
A @ x <= c                           # constraints (1b)
y.sum() >= 1                         # constraints (1c)
y.sum(axis=1) >= 0                   # constraints (1d)
y.T <= c                             # constraints (1e)
A * x >= 1                           # constraints (1f)
A*y + x >= 0                         # constraints (1g)
```

RSOME will automatically transform well-specified distributionally robust models into their deterministic reformulations, which are then passed to the selected external solver. The current version of RSOME provides interfaces to open-source and commercial solvers listed in Table 1.

Solver	License type	RSOME interface	Integer variables	SOC constraints
`scipy.optimize`	Open-source	`lpg_solver`	No	No
`CyLP`	Open-source	`clp_solver`	Yes	No
`OR-Tools`	Open-source	`ort_solver`	Yes	No
`ECOS`	Open-source	`eco_solver`	Yes	Yes
`CPLEX`	Commercial	`cpx_solver`	Yes	Yes
`Gurobi`	Commercial	`grb_solver`	Yes	Yes
`MOSEK`	Commercial	`msk_solver`	Yes	Yes

Table 1. Solver interfaces in RSOME.

Once installed, these solvers can be imported and used to solve an RSOME model. The sample code below demonstrates how Gurobi is selected via the interface `grb` to solve an RSOME model. Other commercial or open-source solvers can be chosen in a similar manner, so one can switch from one solver to another with negligible effort.

```
from rsome import grb_solver as grb     # import interface for Gurobi

model.solve(grb)                        # solve the model by Gurobi
```

4 A Demo Application

In this section, we solve a vehicle pre-allocation problem (which originates from Hao et al. 2020) to demonstrate the syntax and basic features of RSOME. Consider an urban area with I supply nodes and J demand nodes. The operator, before the random demand $\tilde{\boldsymbol{d}} = (\tilde{d}_j)_{j\in[J]}$ realizes, allocates x_{ij} vehicles from supply node $i \in [I]$ (which has a number q_i of idle vehicles) to demand node $j \in [J]$ at a unit cost c_{ij}. After demand realization, the operator collects $\sum_{j\in[J]} r_j \min\{\sum_{i\in[I]} x_{ij}, d_j\}$ revenue. The pre-allocation decision is made by solving a distributionally robust optimization problem

$$
\begin{array}{lll}
\underset{\boldsymbol{x},\,\boldsymbol{y}}{\text{minimize}} & \displaystyle\sum_{i\in[I]}\sum_{j\in[J]}(c_{ij}-r_j)x_{ij} + \sup_{\mathbb{P}\in\mathcal{F}} \mathbb{E}_{\mathbb{P}}\left[\sum_{j\in[J]} r_j y_j(s,\tilde{\boldsymbol{d}},\tilde{\boldsymbol{u}})\right] & \\
\text{subject to} & y_j(s,\boldsymbol{d},\boldsymbol{u}) \geq \displaystyle\sum_{i\in[I]} x_{ij} - d_j & \forall (\boldsymbol{d},\boldsymbol{u}) \in \mathcal{Z}_s,\ s\in[S],\ j\in[J] \\
& y_j(s,\boldsymbol{d},\boldsymbol{u}) \geq 0 & \forall (\boldsymbol{d},\boldsymbol{u}) \in \mathcal{Z}_s,\ s\in[S],\ j\in[J] \\
& y_j \in \bar{\mathcal{A}}(\mathcal{C},\mathcal{J}) & \forall j\in[J] \\
& \displaystyle\sum_{j\in[J]} x_{ij} \leq q_i & \forall i\in[I] \\
& x_{ij} \geq 0 & \forall i\in[I],\ j\in[J],
\end{array}
\tag{2}
$$

where $\tilde{\boldsymbol{u}}$ denotes auxiliary random variables and $\boldsymbol{y}$ is a vector of wait-and-see decisions accounting for bookkeeping revenues. The random demand $\tilde{\boldsymbol{d}}$ is captured by the ambiguity set below:

$$
\mathcal{F} = \left\{ \mathbb{P}\in\mathcal{P}_0(\mathbb{R}^J\times\mathbb{R}^J\times[S]) \;\middle|\; \begin{array}{ll}
(\tilde{\boldsymbol{d}},\tilde{\boldsymbol{u}},\tilde{s}) \sim \mathbb{P} & \\
\mathbb{E}_{\mathbb{P}}[\tilde{\boldsymbol{d}} \mid \tilde{s}=s] = \boldsymbol{\mu}_s & \forall s\in[S] \\
\mathbb{E}_{\mathbb{P}}[\tilde{\boldsymbol{u}} \mid \tilde{s}=s] \leq \boldsymbol{\phi}_s & \forall s\in[S] \\
\mathbb{P}[(\tilde{\boldsymbol{d}},\tilde{\boldsymbol{u}})\in\mathcal{Z}_s \mid \tilde{s}=s] = 1 & \forall s\in[S] \\
\mathbb{P}[\tilde{s}=s] = w_s & \forall s\in[S]
\end{array} \right\},
\tag{3}
$$

which involves a number of S random scenarios. For each scenario $s \in [S]$, the conditional distribution of $\tilde{\boldsymbol{d}}$ is characterized by the conditional mean $\boldsymbol{\mu}_s$ and variance $\boldsymbol{\phi}_s$, as well as the lifted support $\mathcal{Z}_s = \{(\boldsymbol{d},\boldsymbol{u}) \in \mathbb{R}^J\times\mathbb{R}^J : \boldsymbol{d}\in[\underline{\boldsymbol{d}}_s, \bar{\boldsymbol{d}}_s],\ (d_j-\mu_j)^2 \leq u_j\ \forall j\in[J]\}$. The vector $\boldsymbol{w}$ is used to denote scenario weights, which amount to the fractions of data points residing in each scenario. Scenarios of the ambiguity set are generated from the historical data[10] of the regional taxi demands (with noise added for data desensitization) and side information in terms of the rainfall records using

[10] Available at the "taxi_rain.csv" file in our online example "Robust Vehicle Pre-Allocation".

decision tree regressor. The code for generating scenarios and calculating parameters $\boldsymbol{\mu}$, $\boldsymbol{\phi}$, $\bar{\boldsymbol{d}}$, $\underline{\boldsymbol{d}}_s$ and $\boldsymbol{w}$ (*i.e.*, `mu`, `phi`, `d_ub`, `d_lb`, and `w` in the code segment, respectively) is given as follows.

```
from sklearn.tree import DecisionTreeRegressor
import pandas as pd

data = pd.read_csv('https://xiongpengnus.github.io/rsome/taxi_rain.csv')

D, V = data.iloc[:, :10], data.iloc[:, 10:]         # D: demand & V: side information

regr = DecisionTreeRegressor(max_leaf_nodes=4,      # max leaf nodes
                             min_samples_leaf=3)    # min sample size of each leaf
regr.fit(V, D)
mu, index, counts = np.unique(regr.predict(V), axis=0,
                              return_inverse=True,
                              return_counts=True)   # conditional mean
w = counts/V.shape[0]                               # scenario weights
phi = np.array([D.values[index==i].var(axis=0)
                for i in range(len(counts))])       # conditional variance
d_ub = np.array([D.values[index==i].max(axis=0)
                for i in range(len(counts))])       # upper bound of each scenario
d_lb = np.array([D.values[index==i].min(axis=0)
                for i in range(len(counts))])       # lower bound of each scenario
```

Parameters for the event-wise affine adaptation $\bar{\mathcal{A}}(\mathcal{C}, \mathcal{J})$ are (i) $\mathcal{J} = [2J]$ and (ii) $\mathcal{C} = \{\{1\}, \ldots, \{S\}\}$, suggesting that each recourse decision y_j affinely adapts to random variables $(\tilde{\boldsymbol{d}}, \tilde{\boldsymbol{u}})$ and the affine adaptation could be different in each scenario. The distributionally robust model is implemented by the code below.

```
from rsome import dro                           # import the dro module
from rsome import square                        # import the element-wise square function
from rsome import E                             # import the notion of expectation
from rsome import grb_solver as grb             # import the Gurobi interface

model = dro.Model(S)                            # create a DRO model with S scenarios

d = model.rvar(J)                               # random demand as the variable d
u = model.rvar(J)                               # auxiliary random variable u
fset = model.ambiguity()                        # create an ambiguity set
for s in range(S):                              # for each scenario:
    fset[s].exptset(E(d) == mu[s],              # specify the expectation set of d and u
                    E(u) <= phi[s])
    fset[s].suppset(d >= d_lb[s],               # specify the support of d and u
                    d <= d_ub[s],
                    square(d - mu[s]) <= u)
pr = model.p                                    # an array of scenario probabilities
fset.probset(pr == w)                           # w as scenario weights

x = model.dvar((I, J))                          # here-and-now decision x
```

```
21 y = model.dvar(J)                                  # wait-and-see decision y
22 y.adapt(d)                                         # y affinely adapts to d
23 y.adapt(u)                                         # y affinely adapts to u
24 for s in range(S):                                 # for each scenario:
25     y.adapt(s)                                     # affine adaptation of y is different
26
27 model.minsup(((c-r)*x).sum() + E(r@y), fset)       # minimize the worst-case objective
28 model.st(y >= x.sum(axis=0) - d, y >= 0)           # robust constraints
29 model.st(x >= 0, x.sum(axis=0) <= q)              # deterministic constraints
30
31 model.solve(grb)                                   # solve the model by Gurobi
32 objval = model.get()                               # get the optimal objective value
33 xsol = x.get()                                     # get the optimal solution
34 status = model.solution.status                     # return the solution status
35 stime = model.solution.time                        # return the solution time
```

Here, random variables `d` and `u` are defined by the `rvar()` method in lines 8 and 9, where `J` specifies the shape of the variable array. In line 10, the `ambiguity()` method is called to create an ambiguity set, whose support and expectation information are specified by the `suppset()` and `exptset()`, respectively. The scenario weights are specified in line 18, where `pr` is pre-defined to be an array of scenario probabilities in line 17. Decision variables and their adaptations are defined from line 20 to line 25. Note that decision variables are recognized as here-and-now by default, while decision `y` becomes adaptive after the `adapt()` method is called to specify its event-wise affine adaptation. The subsequent lines define the objective function and constraints. Finally, the model is solved by Gurobi via the `grb` solver interface. Once the solution procedure completes, the optimal objective value and the optimal solution of a given decision variable can be retrieved using the `get()` method; see lines 32 and 33. Users can also obtain the solution status and computation time as attributes of the `model.solution` object, as shown in lines 34 and 35.

Different robust and distributionally robust models, with ambiguity sets of their own emphases and the corresponding event-wise affine adaptations, could be seeminglessly recast as an instance of problem (2) with an ambiguity set (3). For more detail, we refer to our online materials[11].

5 Conclusion

We introduce an algebraic modeling package RSOME in Python for modeling robust and distributionally robust optimization problems in a highly readable and mathematically intuitive manner. We believe that RSOME in Python, served as an open-source modeling framework, remains open

[11] Available at "RSOME Paper Numerical Cases".

to further developments that depend on the needs of users. For example, robust satisficing models that arise from the recent frameworks in Long et al. (2022) can be deployed in RSOME if properly tailored. As another example, customized parameter settings (such as those of mixed-integer models) in a solver are also possible.

Acknowledgements. We are grateful to Erick Delage and Melvyn Sim for encouragement and inspiring discussions that motivated this project. We thank Zhaowei Hao, Long He, Zhenyu Hu and Jun Jiang for helpful discussions on the vehicle pre-allocation problem and for sharing the taxi demand and rainfall data. Valuable feedback from RSOME users is also gratefully acknowledged. The research of Zhi Chen is funded by the Strategic Research Grant (project number: 7005792) from the City University of Hong Kong. The research of Peng Xiong is supported by the Ministry of Education, Singapore, under its 2019 Academic Research Fund Tier 3 grant call (Grant MOE-2019-T3-1-010).

References

Ben-Tal, Aharon, Dick den Hertog, Anja De Waegenaere, Bertrand Melenberg, Gijs Rennen. 2013. Robust solutions of optimization problems affected by uncertain probabilities. *Management Science* **59**(2) 341–357.

Ben-Tal, Aharon, Arkadi Nemirovski. 1998. Robust convex optimization. *Mathematics of Operations Research* **23**(4) 769–805.

Bertsimas, Dimitris, Melvyn Sim. 2004. The price of robustness. *Operations Research* **52**(1) 35–53.

Bertsimas, Dimitris, Melvyn Sim, Meilin Zhang. 2019. Adaptive distributionally robust optimization. *Management Science* **65**(2) 604–618.

Chen, Zhi, Melvyn Sim, Peng Xiong. 2020. Robust stochastic optimization made easy with RSOME. *Management Science* **66**(8) 3329–3339.

Chen, Zhi, Melvyn Sim, Huan Xu. 2019. Distributionally robust optimization with infinitely constrained ambiguity sets. *Operations Research* **67**(5) 1328–1344.

Delage, Erick, Yinyu Ye. 2010. Distributionally robust optimization under moment uncertainty with application to data-driven problems. *Operations Research* **58**(3) 595–612.

Dunning, Iain, Joey Huchette, Miles Lubin. 2017. JuMP: a modeling language for mathematical optimization. *SIAM Review* **59**(2) 295–320.

El Ghaoui, Laurent, Francois Oustry, Hervé Lebret. 1998. Robust solutions to uncertain semidefinite programs. *SIAM Journal on Optimization* **9**(1) 33–52.

Goh, Joel, Melvyn Sim. 2010. Distributionally robust optimization and its tractable approximations. *Operations Research* **58**(4) 902–917.

Goh, Joel, Melvyn Sim. 2011. Robust optimization made easy with ROME. *Operations Research* **59**(4) 973–985.

Grant, Michael, Stephen Boyd. 2014. CVX: MATLAB software for disciplined convex programming, version 2.1. http://cvxr.com/cvx.

Hao, Zhaowei, Long He, Zhenyu Hu, Jun Jiang. 2020. Robust vehicle pre-allocation with uncertain covariates. *Production and Operations Management* **29**(4) 955–972.

Isenberg, Natalie, John Siirola, Chrysanthos Gounaris. 2020. PyROS: A Pyomo robust optimization solver for robust process design. *2020 Virtual AIChE Annual Meeting*. AIChE.

Lofberg, Johan. 2004. YALMIP: a toolbox for modeling and optimization in MATLAB. *2004 IEEE international conference on robotics and automation*. IEEE, 284–289.

Long, Daniel Zhuoyu, Melvyn Sim, Minglong Zhou. 2022. Robust satisficing. Forthcoming in *Operations Research*.

Martin, Kipp. 2010. Tutorial: COIN-OR: software for the OR community. *Interfaces* **40**(6) 465–476.

Mohajerin Esfahani, Peyman, Daniel Kuhn. 2018. Data-driven distributionally robust optimization using the Wasserstein metric: performance guarantees and tractable reformulations. *Mathematical Programming* **171**(1-2) 1–52.

Rahimian, Hamed, Sanjay Mehrotra. 2019. Distributionally robust optimization: a review. *arXiv preprint arXiv:1908.05659*.

Soyster, Allen. 1973. Convex programming with set-inclusive constraints and applications to inexact linear programming. *Operations Research* **21**(5) 1154–1157.

Stahlberg, Maximilian. 2020. Robust conic optimization in Python. *Master thesis*, Technische Universität Berlin, Germany.

Vayanos, Phebe, Qing Jin, George Elissaios. 2020. ROC++: robust optimization in C++. *arXiv preprint arXiv:2006.08741*.

Wiebe, Johannes, Ruth Misener. 2021. ROmodel: modeling robust optimization problems in Pyomo. *Optimization and Engineering* 1–22.

Wiesemann, Wolfram, Daniel Kuhn, Melvyn Sim. 2014. Distributionally robust convex optimization. *Operations Research* **62**(6) 1358–1376.

Online Supplementary of "RSOME in Python: An Open-Source Package for Robust Stochastic Optimization Made Easy"

1 The Robust Optimization Framework

The `ro` module in RSOME is designed for robust optimization problems, where tailored modeling tools are developed for specifying random variables, uncertainty sets, the objective function or constraints under the worst-case scenarios that may arise from the uncertainty set, as well as decision rules for recourse decisions. The general framework is given by

$$\begin{array}{ll} \underset{\boldsymbol{x},\,\boldsymbol{y}}{\text{minimize}} & \max\limits_{\boldsymbol{z}\in\mathcal{Z}_0}\left\{\boldsymbol{a}_0^\top(\boldsymbol{z})\boldsymbol{x}+\boldsymbol{b}_0^\top\boldsymbol{y}(\boldsymbol{z})+c_0(\boldsymbol{z})\right\} \\ \text{subject to} & \max\limits_{\boldsymbol{z}\in\mathcal{Z}_m}\left\{\boldsymbol{a}_m^\top(\boldsymbol{z})\boldsymbol{x}+\boldsymbol{b}_m^\top\boldsymbol{y}(\boldsymbol{z})+c_m(\boldsymbol{z})\right\}\leq 0 \quad \forall m\in\mathcal{M} \\ & y_i\in\mathcal{L}(\mathcal{J}_i) \qquad\qquad\qquad\qquad\qquad\qquad \forall i\in[I_y] \\ & \boldsymbol{x}\in\mathcal{X}. \end{array}$$

Here, parameters of proper dimensions,

$$\boldsymbol{a}_m(\boldsymbol{z}):=\boldsymbol{a}_m^0+\sum_{j\in[J]}\boldsymbol{a}_m^j z_j \quad\text{and}\quad c_m(\boldsymbol{z}):=c_m^0+\sum_{j\in[J]}c_m^j z_j,$$

are defined similarly as in the distributionally robust optimization framework and $\mathcal{X}$ is an SOC representable feasible set of the here-and-now decision $\boldsymbol{x}$. The wait-and-see decision $\boldsymbol{y}$ is restricted to a simpler and easy-to-optimize affine function in the following form:

$$\mathcal{L}(\mathcal{J}):=\left\{y:\mathbb{R}^{[J]}\mapsto\mathbb{R}\;\middle|\;y(\boldsymbol{z})=y^0+\sum_{j\in\mathcal{J}}y^j z_j\right\},$$

where the prescribed subset $\mathcal{J}\subseteq[J]$ includes indices of those random components $\tilde{z}_1,\ldots,\tilde{z}_J$ of $\tilde{\boldsymbol{z}}$ to which a particular non-anticipative decision can adapt.

2 Vehicle Pre-Allocation Problem Revisit

In this section, we revisit the vehicle pre-allocation problem and use the `ro` framework to build a robust model and a sample robust model (proposed by Bertsimas et al. 2022b) for this problem.

We also look at the sample robust model from the distributionally robust optimization perspective and relate it to other data-driven distributionally robust models.

2.1 The Robust Model

The robust model is given by

$$\begin{array}{lll} \underset{\boldsymbol{x},\,\boldsymbol{y}}{\text{minimize}} & \max\limits_{\boldsymbol{d}\in\mathcal{Z}} \left\{ \sum\limits_{i\in[I]}\sum\limits_{j\in[J]} (c_{ij}-r_j)x_{ij} + \sum\limits_{j\in[J]} r_j y_j(\boldsymbol{d}) \right\} & \\ \text{subject to} & y_j(\boldsymbol{d}) \geq \sum\limits_{i\in[I]} x_{ij} - d_j & \forall \boldsymbol{d}\in\mathcal{Z},\ j\in[J] \\ & y_j(\boldsymbol{d}) \geq 0 & \forall \boldsymbol{d}\in\mathcal{Z},\ j\in[J] \\ & y_j \in \mathcal{L}([J]) & \forall j\in[J] \\ & \sum\limits_{j\in[J]} x_{ij} \leq q_i & \forall i\in[I] \\ & x_{ij} \geq 0 & \forall i\in[I],\ j\in[J]. \end{array} \tag{1}$$

Here, the wait-and-see decision $\boldsymbol{y}$ is approximated by a linear decision rule $\mathcal{L}([J])$, implying that each y_j affinely depends on the demand realization $\boldsymbol{d}$. The uncertainty set $\mathcal{Z}$ is a box with upper and lower bounds identified as follows.

```
import pandas as pd

data = pd.read_csv('taxi_rain.csv')          # read data from the csv file

demand = data.loc[:, 'Region1':'Region10']   # taxi demand data

d_ub = demand.max().values                   # upper bound of demand
d_lb = demand.min().values                   # lower bound of demand
```

The robust optimization model can be implemented with the following code segment.

```
from rsome import ro                         # import the ro module
from rsome import grb_solver as grb          # import the Gurobi interface

model = ro.Model()                           # create an RO model

d = model.rvar(J)                            # create an array of random variables
zset = (d <= d_ub, d >= d_lb)                # define a box uncertainty set

x = model.dvar((I, J))                       # define here-and-now decisions as array x
```

```
y = model.ldr(J)                                # define linear decision rules as array y
y.adapt(d)                                      # y affinely adapts to d

model.minmax(((c-r)*x).sum() + r@y, zset)       # minimize the worst-case objective
model.st(y >= x.sum(axis=0) - d, y >= 0)        # robust constraints
model.st(x.sum(axis=1) <= q, x >= 0)            # deterministic constraints

model.solve(grb)                                # solve the model with Gurobi
```

2.2 The Sample Robust Model Using the `ro` Framework

Suppose there are a collection $\{\hat{\boldsymbol{d}}_1, \ldots, \hat{\boldsymbol{d}}_S\}$ of historical demand samples available. For a general two-stage problem, Bertsimas et al. (2022b) recently propose a sample robust model with a specific linear decision rule called multi-policy approximation. In particular, for the vehicle pre-allocation problem, the corresponding sample robust model can be cast as follows:

$$
\begin{aligned}
\underset{\boldsymbol{x},\,\boldsymbol{y}}{\text{minimize}} \quad & \sum_{i\in[I]}\sum_{j\in[J]}(c_{ij}-r_j)x_{ij} + \frac{1}{S}\sum_{s\in[S]} a_s \\
\text{subject to} \quad & a_s \geq \sum_{j\in[J]} r_j y_{sj}(\boldsymbol{d}) && \forall \boldsymbol{d}\in\mathcal{Z}_s, s\in[S] \\
& y_{sj}(\boldsymbol{d}) \geq \sum_{i\in[I]} x_{ij} - d_j && \forall \boldsymbol{d}\in\mathcal{Z}_s,\ j\in[J],\ s\in[S] \\
& y_{sj}(\boldsymbol{d}) \geq 0 && \forall \boldsymbol{d}\in\mathcal{Z}_s,\ j\in[J],\ s\in[S] \\
& y_{sj} \in \mathcal{L}([J]) && \forall j\in[J],\ s\in[S] \\
& \sum_{j\in[J]} x_{ij} \leq q_i && \forall i\in[I] \\
& x_{ij} \geq 0 && \forall i\in[I],\ j\in[J].
\end{aligned}
\tag{2}
$$

Here, $\boldsymbol{a}\in\mathbb{R}^S$ is a vector of intermediate variables for the worst-case costs in each scenario and an uncertainty set $\mathcal{Z}_s = \{\boldsymbol{d}\in[\underline{\boldsymbol{d}},\bar{\boldsymbol{d}}] \mid \|\boldsymbol{d}-\hat{\boldsymbol{d}}_s\|\leq\varepsilon\}$—an ε-neighbourhood defined by a general norm $\|\cdot\|$—is constructed around each demand sample $\hat{\boldsymbol{d}}_s$. The multiple-policy approximation then allows different affine dependencies around different samples, leading to the two-dimensional decision rules $(y_{sj}(\boldsymbol{d}))_{s\in[S],j\in[J]}$. Such a sample robust model (assuming the parameter $\varepsilon = 0.25$) is implemented as shown in the following code segment.

```
from rsome import ro                                    # import the ro module
from rsome import norm                                  # import the norm function
from rsome import grb_solver as grb                     # import the Gurobi interface

dhat = demand.values                                    # sample demand as an array
S = dhat.shape[0]                                       # sample size of the dataset
epsilon = 0.25                                          # parameter of robustness

model = ro.Model()                                      # create an RO model

d = model.rvar(J)                                       # random variable d
a = model.dvar(S)                                       # variable as the recourse cost
x = model.dvar((I, J))                                  # here-and-now decision x
y = model.ldr((S, J))                                   # linear decision rule y
y.adapt(d)                                              # y affinely adapts to d

model.min(((c-r)*x).sum() + (1/S)*a.sum())              # minimize the objective
for s in range(S):
    zset = (d <= d_ub, d >= d_lb,
            norm(d - dhat[s]) <= epsilon)               # sample-wise uncertainty set
    model.st((a[s] >= r@y[s]).forall(zset))             # constraints for the sth sample
    model.st((y[s] >= x.sum(axis=0) - d).forall(zset))  # constraints for the sth sample
    model.st((y[s] >= 0).forall(zset))                  # constraints for the sth sample
model.st(x.sum(axis=1) <= q, x >= 0)                    # constraints

model.solve(grb)                                        # solve the model by Gruobi
```

We would like to highlight that the `ro` module enables users to specify different uncertainty sets for the objective function and each of the constraints: in the above sample robust model, different uncertainty sets are defined around samples and these sets for constraints can be easily specified by calling the `forall()` method. Such a feature makes the robust optimization framework more flexible than that in the MATLAB version and can be used to address a rich range of robust models, including the distributional interpretation of robust formulation (Xu et al. 2012), as well as the notion of Pareto robustly optimal solution (de Ruiter et al. 2016).

2.3 The Sample Robust Model Using the `dro` Framework

As pointed out by Bertsimas et al. (2022b), the sample robust model can also be cast as the following distributionally robust optimization problem:

$$
\begin{array}{lll}
\underset{\boldsymbol{x},\,\boldsymbol{y}}{\text{minimize}} & \displaystyle\sum_{i\in[I]}\sum_{j\in[J]}(c_{ij}-r_j)x_{ij}+\sup_{\mathbb{P}\in\mathcal{F}}\mathbb{E}_{\mathbb{F}}\left[\sum_{j\in[J]} r_j y_j(\tilde{s},\tilde{\boldsymbol{d}})\right] & \\
\text{subject to} & y_j(\tilde{s},\boldsymbol{d})\geq \displaystyle\sum_{i\in[I]} x_{ij}-d_j & \forall \boldsymbol{d}\in\mathcal{Z}_s,\ s\in[S],\ j\in[J] \\
& y_j(\tilde{s},\boldsymbol{d})\geq 0 & \forall \boldsymbol{d}\in\mathcal{Z}_s,\ s\in[S],\ j\in[J] \\
& y_j\in\bar{\mathcal{A}}(\mathcal{C},\mathcal{J}) & \forall j\in[J] \\
& \displaystyle\sum_{j\in[J]} x_{ij}\leq q_i & \forall i\in[I] \\
& x_{ij}\geq 0 & \forall i\in[I],\ j\in[J],
\end{array}
\tag{3}
$$

where $\tilde{s}$ is a random scalar corresponds to demand samples. The ambiguity set is given by

$$
\mathcal{F}=\left\{\mathbb{P}\in\mathcal{P}_0(\mathbb{R}^J\times[S])\;\middle|\;\begin{array}{ll}(\tilde{\boldsymbol{d}},s)\sim\mathbb{P} & \\ \mathbb{P}[\tilde{\boldsymbol{d}}\in\mathcal{Z}_s\mid\tilde{s}=s]=1 & \forall s\in[S]\\ \mathbb{P}[\tilde{s}=s]=w_s & \forall s\in[S]\end{array}\right\},
\tag{4}
$$

where for each scenario $s\in[S]$, the weight is $w_s=1/S$ and the corresponding support set $\mathcal{Z}_s=\{\boldsymbol{d}\in[\underline{\boldsymbol{d}},\bar{\boldsymbol{d}})\mid\|\boldsymbol{d}-\hat{\boldsymbol{d}}_s\|\leq\varepsilon\}$ is an ε-neighbourhood around the demand sample $\hat{\boldsymbol{d}}_s$.[1] The multiple-policy approximation is equivalent to the event-wise recourse adaptation that states $y_j\in\bar{\mathcal{A}}(\{\{1\},\cdots,\{S\}\},[J])$, suggesting that y_j affinely depends on the demand realization and around each sample, the affine adaptation is different. The distributionally robust model can be implemented by the sample code below.

```
from rsome import dro                    # import the dro module
from rsome import norm                   # import the norm function
from rsome import E                      # import the expectation notion
```

[1] If the uncertainty parameter $\varepsilon=0$, the sample robust model is equivalent to the SAA approach. Conceptually, the sample robust model is based on a specific type of Wasserstein metric. We refer to Chen et al. (2020, section 5.5) for models based on other types of Wasserstein metric and their approximations of the wait-and-see decisions.

```
4  from rsome import grb_solver as grb                  # import the Gurobi interface
5
6  dhat = demand.values                                 # sample demand as an array
7  S = dhat.shape[0]                                    # sample size of the dataset
8  epsilon = 0.25                                       # parameter of robustness
9  w = 1/S                                              # weights of scenarios
10
11 model = dro.Model(S)                                 # a DRO model with S scenarios
12
13 d = model.rvar(J)                                    # random variable d
14 fset = model.ambiguity()                             # create an ambiguity set
15 for s in range(S):                                   # for each scenario
16     fset[s].suppset(d <= d_ub, d >= d_lb,
17                     norm(d - dhat[s]) <= epsilon)    # define the support set
18 pr = model.p                                         # an array of scenario weights
19 fset.probset(pr == w)                                # specify scenario weights
20
21 x = model.dvar((I, J))                               # here-and-now decision x
22 y = model.dvar(J)                                    # wait-and-see decision y
23 y.adapt(d)                                           # y affinely adapts to d
24 for s in range(S):
25     y.adapt(s)                                       # y adapts to each scenario s
26
27 model.minsup(((c-r)*x).sum() + E(r@y), fset)         # the worst-case expectation
28 model.st(y >= x.sum(axis=0) - d, y >= 0)             # robust constraints
29 model.st(x.sum(axis=1) <= q, x >= 0)                # deterministic constraints
30
31 model.solve(grb)                                     # solve the model by Gruobi
```

Recall the `ro` framework in Section 2.2, the decision rules $(y_{sj}(\boldsymbol{d}))_{s\in[S],j\in[J]}$ therein is defined as a two-dimensional array. Here, in the the `dro` framework, $\boldsymbol{y}$ is one-dimensional and the multiple-policy adaptation is defined by a loop in lines 24 and 25, where event-wise affine adaptation is automatically created by calling the `adapt()` method, with `s` being the sample index.

2.4 Comparison of Models

The robust model (1) with support information, sample robust model (2) with sample information, and distributionally robust model with side information—problem (2) in the main paper—all admit a tractable deterministic reformulation as a linear or second-order cone (SOC) program that is well recognized by the external solver. The reformulations, however, differ largely in terms of the problem size, auxiliary variables, constraints, as well as coefficients; see a summary in Table 1. This reminds us the lack of intuition in the tedious and error-prone transition from (distributionally) robust models to their deterministic mathematical equivalents, and more importantly, shows the

need of an accompanying technology to free modelers/practitioners from the transition procedure.

Model	Support information	Sample information	Side information
Number of variables	373	44132	4889
Continuous/binaries/integers	373/0/0	44132/0/0	4889/0/0
Number of linear constraints	232	17789	1930
Inequalities/equalities	211/21	16172/1617	502/1428
Number of coefficients	823	92526	10759
Number of SOC constrains	0	1617	840

Table 1. Summary of deterministic reformulations.

Basic information of the deterministic reformulation (in a standard form) of a (distributionally) robust model can be retrieved by calling the `do_math()` method the `Model` object; see below.

```
dc = model.do_math()                # deterministic counterpart of the model
formula = dc.show()                 # formula of the deterministic counterpart
```

Here, the `show()` method further exports detailed information of the deterministic reformulation as a Pandas `DataFrame`, shown in Figure 1. The `DataFrame` contains coefficients of the objective function (`Obj`) and linear constraints (`LC`), as well as upper/lower bounds (`UB`/`LB`) and types (`Type`) of decision variables. Such data can be conveniently processed by functions and operations provided by the Pandas library, and could be useful for debugging or exploring the problem's structure.

2.5 Alternative Data-Driven Approaches

To incorporate with side information, Bertsimas et al. (2022a) propose to adjust the weights $\boldsymbol{w}$ of samples in the original sample robust model of Bertsimas et al. (2022b), where the robustness parameter ε is used to control the distance to (or equivalently, admissible deviation from) the sample point. Indeed, as pointed out by Bertsimas et al. (2022a), the ambiguity set (4) corresponds to a variety of data-driven approaches, provided that the robustness parameter `epsilon` and weights

```
         x1   x2   x3   x4  ... x372 x373 sense constant
Obj       1    0    0    0  ...    0    0     -        -
LC1       0    1    0    0  ...    0    0    <=       -0
LC2       0    0    1    0  ...    0    0    <=       -0
...     ...  ...  ...  ...  ...  ...  ...   ...      ...
UB      inf  inf  inf  inf  ...    0    0     -        -
LB     -inf    0    0    0  ... -inf -inf     -        -
Type      C    C    C    C  ...    C    C     -        -

[236 rows x 375 columns]
```

Figure 1. Information of the deterministic reformulation of the robust model (1).

`w` are properly specified; see a summary in Table 2. With the help of third-party packages (*e.g.*, PyTorch, Scikit-Learn and TensorFlow) in Python ecosystem, it is easy to implement various machine learning methods, such as K-nearest neighbors, kernel regression, classification and regression tree, and random forest, for determining the weight factors $\boldsymbol{w}$. For more detail, we refer interested readers to Bertsimas et al. (2022a) and Bertsimas and Kallus (2020). Equipped with the distributionally robust optimization framework in RSOME, *different* data-driven approaches in Table 2 can be implemented by using the *same* sample code as that in Section 2.3.

	`w` $= 1/S$	`w` from machine learning
`epsilon` $= 0$	SAA	Bertsimas and Kallus (2020)
`epsilon` > 0	Bertsimas et al. (2022b)	Bertsimas et al. (2022a)

Table 2. Data-driven approaches.

References

Bertsimas, Dimitris, Nathan Kallus. 2020. From predictive to prescriptive analytics. *Management Science* **66**(3) 1025–1044.

Bertsimas, Dimitris, Christopher McCord, Bradley Sturt. 2022a. Dynamic optimization with side information. *Forthcoming in European Journal of Operational Research.*

Bertsimas, Dimitris, Shimrit Shtern, Bradley Sturt. 2022b. Two-stage sample robust optimization. *Operations Research* **70**(1) 624–640.

Chen, Zhi, Melvyn Sim, Peng Xiong. 2020. Robust stochastic optimization made easy with RSOME. *Management Science* **66**(8) 3329–3339.

de Ruiter, Frans, Ruud Brekelmans, Dick den Hertog. 2016. The impact of the existence of multiple adjustable robust solutions. *Mathematical Programming* **160**(1) 531–545.

Xu, Huan, Constantine Caramanis, Shie Mannor. 2012. A distributional interpretation of robust optimization. *Mathematics of Operations Research* **37**(1) 95–110.