Robust Stochastic Optimization Made Easy

Users Guide for RSOME

Zhi Chen, Melvyn Sim, Peng Xiong

Version 1.2

December 2019

Contents

1. Introduction

RSOME (Robust Stochastic Optimization Made Easy) is a MATLAB algebraic toolbox designed for generic optimization modeling under uncertainty.

RSOME is based on a novel robust stochastic optimization (RSO) framework proposed by [CSX19]. The framework unifies a wide variety of approaches for optimization under uncertainty, including the traditional scenario-tree based stochastic linear optimization, classical robust optimization, as well as the emerging distributionally robust optimization that considers state-of-the-art data-driven ambiguity sets.

RSOME is consistent with standard MATLAB syntax and data structure, and it is very friendly to users who are familiar with MATLAB. In particular, RSOME provides a powerful interface to specify optimization problems for both static and dynamic decision-making under uncertainty. RSOME automatically transforms these optimization problems into their equivalent deterministic counterparts in forms of linear, second-order cone, or mixed-integer programs, so they can be solved by commercials solvers such as CPLEX, Gurobi, and MOSEK.

In this user guide, we will provide detailed instructions and examples for modeling with the RSOME toolbox. For notations, we will use bold letters to denote matrices and vectors. Entries of matrices or vectors are expressed as regular letters with subscripts indicating the indices. All functions and classes are written as "`functions`", while class instances and input/output arguments are given as "`arguments`".

2. Modeling with RSOME

2.1 System Requirement and Preparations

RSOME is a MATLAB modeling toolbox compatible with the 64-bit Windows, macOS, and Linux operating systems. It can be installed by adding the path of the software into the working directory via calling **addpath**(**cd**) in the RSOME root folder. The current version of RSOME uses CPLEX as the default solver. The solver can be configured by following "Setting up CPLEX for MATLAB". For Gurobi and Mosek users, you may follow the steps in "Setting up the Gurobi MATLAB interface" and "MOSEK Optimization Toolbox for MATLAB", respectively, to configure the selected solvers.

2.2 Basic Syntax of RSOME

2.2.1 RSOME Model

In the RSOME toolbox, key components like variables, functions, and constraints are created through a user-specified RSOME class. An RSOME model and its associate components can be declared by the following functions and methods.

1. `rsome`

The first step of running RSOME is to create a new RSOME model instance by using the constructor **rsome**.

model = **rsome** creates an RSOME **model** with a default name "untitled".

model = **rsome**(**modelName**) creates an RSOME **model** with a user defined name **modelName**.

2. `decision`

As a method of RSOME **model**, **decision** defines new decision variables.

x = **model**.**decision** defines a continuous decision variable for **model**.

x = **model**.**decision**(**N**) defines a column vector containing **N** continuous decision variables for **model**.

x = **model**.**decision**(**N**, **M**) defines a matrix containing **N**×**M** continuous decision variables for **model**.

x = **model**.**decision**(**N**, **M**, **type**) defines a matrix containing **N**×**M** decision variables for **model**. These variables are continuous if **type**='C', binary if **type**='B', and general integer if **type**='I'. Other values of **type** would generate an error message.

x = **model**.**decision**(**N**, **M**, **type**, **name**) defines a matrix of decision variables, where the matrix dimension is **N**×**M** and the variable type is specified by **type**. The last input argument **name** specifies the name of these decision variables.

3. min/max

Methods **min** and **max** of an RSOME **model** are used to define the objective function. Please note that the objective function must be a scalar.

model.**min**(**function**) defines the objective of **model** to be minimizing **function**.

model.**max**(**function**) defines the objective of **model** to be maximizing **function**.

4. set

As a method of RSOME **model**, **set** creates a feasible set of variables.

model.**set** creates a feasible set of the RSOME instance **model**.

model.**set**(**constraint1**, **constraint2**, ...) creates a feasible set of RSOME, which is defined by the given constraints.

5. append

As a method of RSOME **model**, **append** adds a new set of constraints.

model.**append**(**constraint**) appends **constraint** to **model**.

model.**append**(**set**) appends constraints that define the feasible set **set** to **model**.

6. solve

The method **solve** is used to solve the RSOME **model** by the selected solvers.

model.**solve** solves the RSOME **model** with the default settings.

model.**solve**(**mipGap**) solves the RSOME instance **model**, with the MIP gap specified by **mipGap**. The default value of the MIP gap is 10^{-4}.

7. get

The method **get** retrieves the optimal objective value as well as the optimal solution of an RSOME model after it is solved.

model.**get** retrieves the optimal objective value of **model**.

x.**get** retrieves the optimal solution of the decision **x**.

2.2.2 Variables and Linear Constraints

The RSOME toolbox arranges variables, functions, and constraints as matrices. The Syntac is consistent with the standard MATLAB syntax in matrix indexing and arithmetic operations, such as addition, subtraction, sum, matrix and element-wise multiplication, etc. Some illustrative examples are provided as follows.

```
model = rsome;                          % Create an RSOME model;

x = model.decision(5, 6);               % Define a 5*6 decision variable matrix
y = model.decision(3);                  % Define a 3*1 decision variable matrix
z = model.decision(5);                  % Define a 5*1 decision variable matrix

model.append(x >= 0);                   % Each element of x is nonnegative
model.append(x >= zeros(5,6));          % Same as above
model.append(3*x(1, 5) <= 10);          % Element of x in row 1, column 5
model.append(x(1, :) <= x(5, :));       % The 1st row is not larger than the 5th row
model.append(x(8) <= 2);                % Element of x in row 3 and column 2

A = ones(5, 5);                         % A is a 5*5 matrix
b = ones(5, 1);                         % b is a 5*1 vector
c = rand(3, 1);                         % c is a 3*1 vector
model.append(A*z + b >= 0);             % Matrix multiplication A*x
model.append(sum(z) - y'*c == 1);       % Transpose of y, sum of each element of x
model.append(c.*y <= 2)                 % Element-wise multiplication c.*y
```

2.2.3 Convex Functions

The RSOME toolbox provides several commonly used convex functions for modeling nonlinear constraints. An error message would appear if a constraint is non-convex.

1. `abs`

Function **`abs`** returns the absolute value of a matrix of affine functions.

`abs`(**`affine`**) returns the absolute value of **`affine`**, which is a matrix of affine expressions.

2. `norm`

Function **`norm`** returns the vector norm of affine functions. It cannot be applied to matrices.

`norm`(**`affine`**) returns the Euclidean norm of **`affine`**, which is a vector of affine functions.

`norm`(**`affine`**, **`p`**) returns the **`p`**-norm of **`affine`** as a vector of affine expressions, where **`p`** can only be 1, 2, or **`inf`**.

3. Element-wise square **`.^2`**

The operator **`.^2`** conducts element-wise square of the input matrix.

`affine.^2` conducts element-wise square of the given matrix **`affine`**.

4. `sumsqr`

Function **`sumsqr`** returns the sum of squares of affine functions that reside in a vector. It cannot be applied to matrices.

`sumsqr`(**`affine`**) returns the sum of squares of affine expressions in a vector **`affine`**.

5. maxfun/minfun

Functions **maxfun**/**minfun** create piecewise expressions by taking the maximum/minimum of a number of affine functions or constants.

maxfun(**exprCell**) creates a piecewise expression by taking the maximum of affine functions or constants stored in the given cell array **exprCell**. All affine functions and constants must be scalars.

minfun(**exprCell**) creates a piecewise expression by taking the minimum of affine functions or constants stored in the given cell array **exprCell**. All affine functions and constants must be scalars.

Some examples of using these convex functions in RSOME models are provided below.

```
model = rsome;                          % Create an RSOME model
x = model.decision(8, 1);               % Define a 8*1 decision variable matrix
y = model.decision(1, 3);               % Define a 1*3 decision variable matrix

B = ones(3, 1);                         % B is a 3*1 matrix
model.append(abs(x(1:3)) <= y');        % Absolute value of x(1:3)
model.append(y(1) >= norm(x));          % Euclidean norm of x
model.append(y(2) >= norm(x, 1));       % 1-norm of x
model.append(-sumsqr(x) + 4 >= 0);      % Sum of square of vector x
model.append(x(1:3, :).^2 - B<=0);      % Element-wise square of x(1:3,:)
```

2.2.4 Parameters

The current version of RSOME allows users to specify three parameters while solving the optimization problem. These parameters are fields of the structure **model.Param** and they can be specified as demonstrated in the following code segment.

```
model = rsome;                          % Create an RSOME model

model.Param.solver = 'gurobi';          % Change the solver to be Gurobi
model.Param.solver = 'cplex';           % Change the solver to be CPLEX

model.Param.display = 0;                % Disable the display of solution status
model.Param.display = 1;                % Enable the display of solution status

model.Param.mipgap = 1e-3;              % Set the MIP gap to 1e-3
```

2.3 Computational Examples

■ **Example 2.3.1** Consider a linear program (2.1) with two decision variables x and y,

$$\begin{aligned} \max \quad & 3x + 4y \\ \text{s.t.} \quad & 2.5x + y \leq 20 \\ & x + 2y \leq 16 \\ & |y| \leq 4. \end{aligned} \tag{2.1}$$

Such a linear program can be easily implemented by the following code, which gives the optimal objective value to be 35.2, and optimal solution to be $x = 6.4$ and $y = 4$.

```
model = rsome('LP Exmaple');          % Create an RSOME model named "LP Example"

x = model.decision;                   % Create a decision variable x
y = model.decision;                   % Create a decision variable y

model.max(3*x + 4*y);                 % Define the objective function

model.append(2.5*x + y <= 20);        % Add the 1st constraint
model.append(x + 2*y <= 16);          % Add the 2nd constraint
model.append(abs(y) <= 4);            % Add the 3rd constraint

model.solve;                          % Solve the problem

Obj = model.get;                      % Get the objective value
X = x.get;                            % Get the optimal solution of x
Y = y.get;                            % Get the optimal solution of y
```

This example shows that the first step of formulating an optimization problem is to create an RSOME instance **model**. The user then needs to define decision variables, the objective function, and constraints of the **model**. Lastly, the model is solved by calling the method **solve**. The objective value and the optimal solutions can be accessed via the method **get**. ■

■ **Example 2.3.2** The second example is a portfolio problem presented in [BS04] and [BN99]. Consider a set of $n = 150$ stocks and each stock has a random return $\tilde{p}_i$. Let x_i be the fraction of the total wealth to be invested in stock i, then the classical mean-variance approach for portfolio decision-making can be written as the following quadratic programming model:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} p_i x_i - \phi \sum_{i=1}^{n} \sigma^2 x_i^2 \\ \text{s.t.} \quad & \sum_{i=1}^{n} x_i = 1 \\ & x_i \geq 0, \quad \forall i = 1, 2, .., n, \end{aligned} \tag{2.2}$$

where p_i and σ_i are respectively the mean value and standard deviation of the random return $\tilde{p}_i$, and $\phi = 5$ is a constant that controls the trade-off between the expected return and the risk. Following [BS04] and [BN99], the parameters are set to be:

$$p_i = 1.15 + i\frac{0.05}{150}, \ \sigma_i = \frac{0.05}{450}\sqrt{2in(n+1)}, \ \forall i \in [n]. \tag{2.3}$$

The RSOME code of this quadratic programming problem is given below.

```
n  = 150;                                  % Number of stocks
p  = 1.15+ 0.05/150*(1:n)';                % Mean returns
sigma = 0.05/450*sqrt(2*(1:n)'*n*(n+1));   % Standard deviations
```

```
phi = 5;                                    % Trade-off constant

model = rsome('mean-var portfolio');        % Create a model

x = model.decision(n);                      % Decisions as fractions of investment

model.max(p'*x - phi*sumsqr(sigma.*x));     % Define the objective function
model.append(sum(x)==1);                    % Constraint of x
model.append(x>=0);                         % Bound of x

model.solve;                                % Solve the problem
```

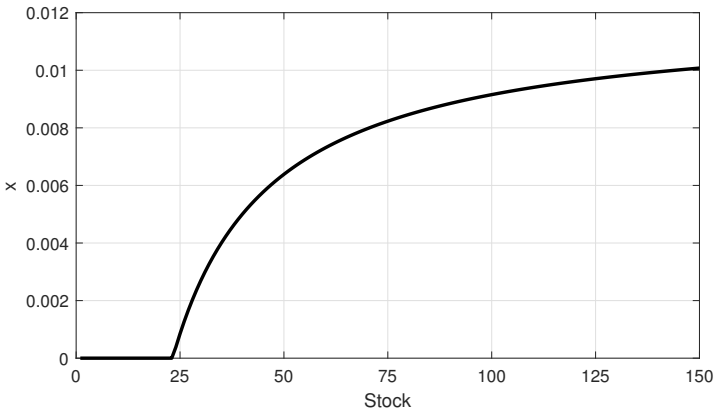

Figure 2.1: The solution to the mean deviation portfolio model

Note that the quadratic term in the objective function is expressed by the function **sumsqr** for sum of squares. The RSOME toolbox first standardizes the above problem into a second-order cone program and then solves it by calling the CPLEX solver. The objective value is 1.170 and the optimal solution, in terms of the investment in each stock, is shown in Figure 2.1. ■

2.4 Summary

This chapter discusses the basic syntax of the RSOME toolbox in addressing ordinary linear or quadratic programming problems. The usage of random variables and the declaration of the proposed event-wise ambiguity set are presented in the next chapter. Some robust optimization and distributionally robust optimization examples will also be provided to demonstrated the implementation of RSOME in decision-making under uncertainty.

3. Event-Wise Ambiguity Set

3.1 Mathematical Format

The RSOME toolbox replies on a generic event-wise ambiguity set, proposed by [CSX19], to capture the randomness in model parameters. Similar to the scenario-representation of uncertainty of stochastic programming models, the ambiguity set is defined based on S distinctive scenarios. Given these scenarios, we also consider K events $\mathcal{E}_k,\ k \in [K]$, each of which is a collection of scenarios. Let $\tilde{\boldsymbol{z}} \in \mathbb{R}^{I_z}$ be a random vector of all random variables and $\tilde{s}$ be a random scalar indicating the realization of uncertain scenarios. The event-wise ambiguity set, denoted by $\mathcal{F}$, can be expressed in the following format:

$$\mathcal{F} = \left\{ \mathbb{P} \in \mathcal{P}_0\left(\mathbb{R}^{I_z} \times [S]\right) \;\middle|\; \begin{array}{ll} (\tilde{\boldsymbol{z}}, \tilde{s}) \sim \mathbb{P} & \\ \mathbb{E}_{\mathbb{P}}\left[\tilde{\boldsymbol{z}} \,|\tilde{s} \in \mathcal{E}_k\right] \in \mathcal{Q}_k & \forall k \in [K] \\ \mathbb{P}\left[\tilde{\boldsymbol{z}} \in \mathcal{Z}_s \,|\tilde{s} = s\right] = 1 & \forall s \in [S] \\ \mathbb{P}\left[\tilde{s} = s\right] = p_s & \forall s \in [S] \\ \text{for some } \boldsymbol{p} \in \mathcal{P} & \end{array} \right\}, \tag{3.1}$$

where $\mathcal{Z}_s, s \in [S]$, $\mathcal{Q}_k, k \in [K]$, and $\mathcal{P} \subseteq \left\{ \boldsymbol{p} \in \mathbb{R}^S_{++} \,\middle|\, \sum_{s\in[S]} p_s = 1 \right\}$ are closed and convex sets. For each scenario s, we define a specific support $\mathcal{Z}_s$ for random variables $\tilde{\boldsymbol{z}}$, and for each event $\mathcal{E}_k$, the conditional expectation of $\tilde{\boldsymbol{z}}$ is defined by the set $\mathcal{Q}_k$. In the last two lines of the ambiguity set, $\boldsymbol{p}$ is the vector of probabilities of all scenarios and its uncertainty set is captured by $\mathcal{P}$.

[CSX19] show the worst-case expectation over the event-wise ambiguity set can be effectively determined by solving a classical robust optimization model. However, due to the complicated scenario/event structures, directly formulating such a robust optimization problem could be time consuming. To address this issue, the RSOME toolbox provides a friendly user interface to specify the mathematical format (3.1) into highly readable MATLAB code. Details of the RSOME syntax are provided in the next section.

3.2 Defining Ambiguity Sets in RSOME

3.2.1 Random Variables

Similar to decision variables introduced in the previous chapter, random variables in RSOME are also arranged as vectors and matrices, and they can be created by the method `random` with the given dimensions and names.

1. `random`

As a method of the RSOME `model`, **`random`** defines new random variables.

`z` = `model`.**`random`** defines a random variable as a scalar for `model`.

`z` = `model`.**`random`**(`N`) defines a column vector of `N` random variables for `model`.

`z` = `model`.**`random`**(`N`, `M`) defines a matrix of `N`×`M` random variables for `model`.

`z` = `model`.**`random`**(`N`, `M`, `name`) defines a matrix of `N`×`M` random variables for `model`, named as `name`.

3.2.2 Event-Wise Ambiguity Set

The RSOME toolbox can be implemented to incorporate the event-wise ambiguity set via the **`ambiguity`** data structure and a series of associated methods, as discussed below.

1. `ambiguity`

As a method of `model`, **`ambiguity`** creates an ambiguity set.

`P` = `model`.**`ambiguity`** constructs an ambiguity set `P` of the RSOME `model`. By default, the event-wise ambiguity set `P` has only one scenario.

`P` = `model`.**`ambiguity`**(`S`) constructs an event-wise ambiguity set `P` of the RSOME `model`. This event-wise ambiguity set `P` has `S` scenarios.

2. `suppset`

As a method of ambiguity sets, **`suppset`** is used to define scenario-wise support for random variables.

`P`.**`suppset`**(`set`) assigns the support for all scenarios of the ambiguity set `P` to be the given feasible set `set`.

`P`(`s`).**`suppset`**(`set`) assigns the support for scenarios indexed by `s` of the ambiguity set `P` to be the given feasible set `set`.

`P`.**`suppset`**(`constraint1`, `constraint2`, ...) assigns the support for all scenarios of the ambiguity set `P` to be defined by a number of given constraints.

`P`(`s`).**`suppset`**(`constraint1`, `constraint2`, ...) assigns the support for scenarios indexed by `s` of the ambiguity set `P` to be defined by a number of given constraints.

3. `expect`

Function **`expect`** returns the expectation of random variables. It is used in defining the uncertainty set of the (conditional) event-wise expectations.

`ez` = **`expect`**(`z`) returns `ez` as the expectation of random variables `z`.

4. `exptset`

As a method of the ambiguity set, **`exptset`** defines the uncertainty set of the (conditional) event-wise expectations of random variables. Note that the function **`expect`** must be used in **`exptset`** to express the constraints of expectations.

`P`.**`exptset`**(`set`) assigns the uncertainty set of random variable expectations to be `set` .

`P`(`s`).**`exptset`**(`set`) assigns the uncertainty set of random variable conditional expectations, given the event as a collection of scenarios `s`, to be `set`.

`P`.**`exptset`**(`constraint1`, `constraint2`, ...) assigns the uncertainty set of random variable expectations to be a series of given constraints.

`P`(`s`).**`exptset`**(`constraint1`, `constraint2`, ...) assigns the uncertainty set of random variable conditional expectations, given the event as a collection of scenarios `s`, to be a series of given constraints.

5. `prob`

As an attribute of the ambiguity set, **`prob`** is a column vector containing the probabilities of all scenarios. It is used in defining the uncertainty set of the likelihoods of scenarios.

`P`.**`prob`** returns a column vector of probabilities of all scenarios in the ambiguity set `P`.

6. `probset`

As a method of ambiguity sets, **`probset`** is used to define the uncertainty set of scenario probabilities `P`.**`prob`**.

`P`.**`probset`**(`set`) assigns the uncertainty set of scenario probabilities associated with the ambiguity set `P` to be the given feasible set `set`.

`P`.**`probset`**(`constraint1`, `constraint2`, ...) assigns the uncertainty set of scenario probabilities in the ambiguity set `P` to be defined by a number of given constraints.

7. `with`

The method **`with`** selects the ambiguity set for the RSOME model.

`model`.**`with`**(`P`) selects `P` as the ambiguity set of the RSOME model `model`.

3.3 Computational Examples

■ **Example 3.3.1** In the first example, we consider the same portfolio problem discussed in the previous chapter, but solve it by a classical robust optimization approach as in [BS04]. The formulation of the robust portfolio model is given as below:

$$\begin{aligned} \min \quad & \max_{\boldsymbol{z}\in\mathcal{Z}} \sum_{i=1}^{n} (p_i + \sigma_i z_i)\, x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} x_i = 1 \\ & x_i \geq 0, \quad \forall i = 1, 2, ..., n, \end{aligned} \tag{3.2}$$

where the variable z_i represents the uncertain deviation of each stock return from its mean value p_i, and the uncertainty set $\mathcal{Z}$ is expressed as equation (3.3):

$$\mathcal{Z} = \{\boldsymbol{z} \in \mathbb{R}^n \,|\, \|\boldsymbol{z}\|_\infty \leq 1, \|\boldsymbol{z}\|_1 \leq \Gamma\}\,. \tag{3.3}$$

Given the budget of uncertainty $\Gamma = 3$, the mean return p_i and its standard deviation σ_i as in equations (2.3), the RSOME code of the robust model is given below.

```
1  %% Parameters
2  n  = 150;                                % Number of stocks
3  p  = 1.15 + 0.05/150*(1:n)';             % Mean return
4  sigma = 0.05/450*sqrt(2*n*(n+1)*(1:n)'); % Maximum deviation
5  Gamma = 3;                               % Gamma as the budget of uncertainty
6
7  %% RSOME model
8  model = rsome('portfolio');              % Create a model, named "portfolio"
9
10 %% Random variables and the ambiguity set
11 z = model.random(n);                      % Random deviation z
12 P = model.ambiguity;                      % Create P as the ambiguity set
13 P.suppset(norm(z, Inf) <= 1, ...
14           norm(z, 1) <= Gamma);          % Uncertainty set of z
15 model.with(P);                           % Set P as the ambiguity set of model
16
17 %% Decision variables
18 x = model.decision(n);                    % Decision x as fractions of investment
19
20 %% Objective function and constraints
21 model.max((p+sigma.*z)' * x)              % Define the objective function
22 model.append(sum(x) == 1);                % Constraint of x
23 model.append(x >= 0);                     % x are non-negative
24
25 %% Solution
26 model.solve;                              % Solve the problem
```

In the above code segment, the uncertainty set $\mathcal{Z}$ defined in line 13-14 gives the support of the single-scenario ambiguity set $\mathcal{P}$. The robust model hence maximizes the objective

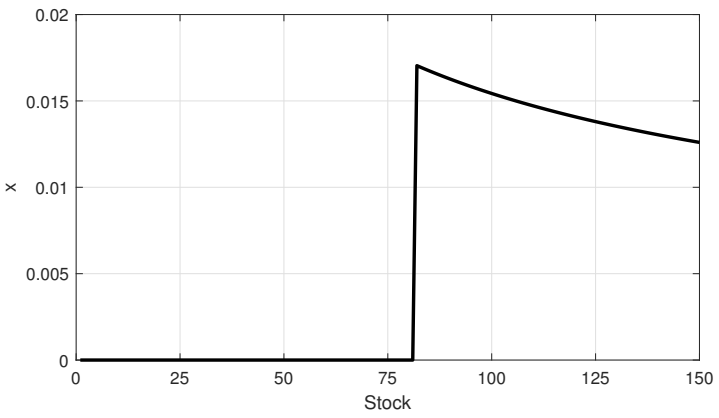

Figure 3.1: The solution to the robust mean deviation portfolio model

function (see line 21) under the worst-case realization over the uncertainty set $\mathcal{Z}$. RSOME transforms this robust optimization model into a tractable deterministic counterpart and then solve it by calling the solver. The optimal objective value is 1.1771 and the optimal solution in terms of the allocation of investments is shown in Figure 3.1. ■

■ **Example 3.3.2** In this example, we formulate a one-product newsvendor problem as a distributionally robust optimization model considering a Wasserstein ambiguity set studied by [MK18]. According to reference [CSX19], such a Wasserstein ambiguity set can be mapped into the format of an event-wise ambiguity set. Let $\tilde{u} \in [0, \bar{U}]$ be the random demand of the product and $\hat{u}_s$ be its empirical distribution, then the Wasserstein ambiguity set can be written as:

$$\bar{\mathcal{F}} = \left\{ \mathbb{P} \in \mathcal{P}_0(\mathbb{R} \times [S]) \;\middle|\; \begin{array}{ll} (\tilde{u}, \tilde{s}) \sim \mathbb{P} & \\ \mathbb{E}_{\mathbb{P}}\left[\rho(\tilde{u}, \hat{u}_{\tilde{s}}) \,|\tilde{s} \in [S]\right] \leq \theta & \\ \mathbb{P}\left[\tilde{u} \in [0, \bar{U}] \,|\tilde{s} = s\right] = 1, & \forall s \in [S] \\ \mathbb{P}\left[\tilde{s} = s\right] = 1/S, & \forall s \in [S] \end{array} \right\}, \tag{3.4}$$

where S is the sample size and θ is the radius of the Wasserstein ball. The ambiguity set $\bar{\mathcal{F}}$ above can be rewritten as the following lifted form by introducing in an auxiliary random variable $\tilde{v}$ to represent the epigraph of $\rho(\tilde{u}, \hat{u}_{\tilde{s}})$ in each scenario s:

$$\mathcal{F} = \left\{ \mathbb{P} \in \mathcal{P}_0(\mathbb{R}^2 \times [S]) \;\middle|\; \begin{array}{ll} (\tilde{z}, \tilde{s}) = ((\tilde{u}, \tilde{v}), \tilde{s}) \sim \mathbb{P} & \\ \mathbb{E}_{\mathbb{P}}\left[\tilde{v} \,|\tilde{s} \in [S]\right] \leq \theta & \\ \mathbb{P}\left[\begin{array}{l} \tilde{u} \in [0, \bar{U}], \\ \rho(\tilde{u}, \hat{u}_{\tilde{s}}) \leq \tilde{v} \end{array} \middle| \tilde{s} = s\right] = 1, & \forall s \in [S] \\ \mathbb{P}\left[\tilde{s} = s\right] = 1/S, & \forall s \in [S] \end{array} \right\}. \tag{3.5}$$

Let w be the decision variable determining the ordering quantity, the distributionally robust newsvendor model over the lifted ambiguity set $\mathcal{F}$ can be expressed as:

$$\begin{array}{ll} \max & (p-c)w - \sup\limits_{\mathbb{P} \in \mathcal{F}} \mathbb{E}_{\mathbb{P}} \max\{p \cdot (w - \tilde{u}), 0\} \\ \text{s.t.} & w \geq 0. \end{array} \tag{3.6}$$

In this example, we set the unit selling price and unit ordering cost to $p = 1.5$ and $c = 1.0$, respectively. The upper bound of random demand is $\bar{U} = 100$ and $S = 500$ independent empirical demand realizations are randomly generated from a uniform distribution between 0 and $\bar{U}$. Finally, the Wasserstein ball radius is set to be $\theta = 0.01\bar{U}$. The one-product newsvendor model can be implemented by the MATLAB program below.

```
1  %% Parameters
2  Ubar = 100;                          % Upper bounds of random demands
3  S = 500;                             % Number of samples
4  Uhat = Ubar * rand(1, S);            % Empirical demand realizations
5
```

```
6  p = 1.5;                                  % Unit selling price
7  c = 1.0;                                  % Unit ordering cost
8  theta = Ubar*0.01;                        % Radius of the Wasserstein ball
9
10 %% Create a RSOME model
11 model = rsome('newsvendor');               % Create a model, named "newsvendor"
12
13 %% Random variables and the Wasserstein ambiguity set
14 u = model.random;                          % Random demand
15 v = model.random;                          % Auxiliary random variable
16 P = model.ambiguity(S);                    % Create an ambiguity set with S scenarios
17 for n = 1:S
18     P(n).suppset(0 <= u, u <= Ubar, ...
19                  norm(u-Uhat(n)) <= v );% Define the support set for each scenario
20 end
21 P.exptset(expect(v) <= theta);
22 prob = P.prob;                           % pr for all scenario probabilities
23 P.probset(prob == 1/S);                  % The probability set of the ambiguity set
24 model.with(P);                           % The ambiguity set of the model is P
25
26 %% Decision
27 w = model.decision;                      % Define a decision variable w
28
29 %% Objective function
30 loss = maxfun({p*(w-u), 0});             % Profit loss due to unsold ordering
31 model.max((p-c)*w - expect(loss));       % Objective as the worst-case expectation
32
33 %% Constraints
34 model.append(w >= 0);                    % The variable w is non-negative
35
36 %% Solution
37 model.solve;                             % Solve the model
```

The ambiguity set is defined in line 16-24. According to the ambiguity set in equation (3.5), each empirical demand realization is associated with a scenario s. For each scenario, the lifted support is specified by a **for** loop in lines 17-20, and the equal probability of each scenario is $1/S$ as specified in line 23. The expectation of the auxiliary variable $\tilde{v}$ over all scenarios is bounded by the radius θ as in line 21.

The objective function in (3.6) involves the worst-case expectation of a piecewise term $\max\{p \cdot (w - \tilde{u}), 0\}$, which is refered to as the profit loss due to unsold ordering. The piecewise term can be expressed by the **maxfun** function in line 30, with the two pieces $p \cdot (w - \tilde{u})$ and 0 being stored in a cell array. Further details of this function are given in Section 2.2.3. The function **expect** in line 31 indicates the worst-case expectation over the ambiguity set **P**. Note that RSOME could tell if the worst-case expectation takes the supremum or infimum value by checking whether it is a minimization or maximization problem. Hence, there is no need for the users to specify the sup operator as in the mathematical formulation (3.6). ∎

3.4 Summary

This chapter presents the basic syntax and the procedure of defining the event-wise ambiguity set in the RSOME toolbox. It can be seen that codes in RSOME match the mathematical expressions of the ambiguity set very well, so they are easy to read and convenient to implement. So far we have only covered non-adaptive models where all decisions are taken before any uncertainty realization. In the next chapter, we extend the framework to address adaptive decision-making using dynamic models.

4. Adaptive Models

4.1 Event-Wise Recourse Adaptations

The scenario structure of the event-wise ambiguity set inspires an event-wise recourse adaptation scheme for non-anticipative decision-making in various adaptive optimization models. As in the event-wise ambiguity set, an event $\mathcal{E}$ is defined as a subset of all scenarios, *i.e.*, $\mathcal{E} \subseteq [S]$, where S is the total number of scenarios defined in the event-wise ambiguity set. A partition of scenarios then induces a collection $\mathcal{C}$ of mutually exclusive and collectively exhaustive (MECE) events. Then a mapping $\mathcal{H}_{\mathcal{C}} : [S] \mapsto \mathcal{C}$ is defined such that $\mathcal{H}_{\mathcal{C}}(s) = \mathcal{E}$, for which $\mathcal{E}$ is the only event in $\mathcal{C}$ that contains the scenario s. Based on the collection $\mathcal{C}$ of MECE events and a set $\mathcal{I} \in [I_z]$ of random variable indices, we define the *event-wise static adaptation* by

$$\mathcal{A}(\mathcal{C}) = \left\{ x : [S] \mapsto \mathbb{R} \;\middle|\; \begin{array}{l} x(s) = x^{\mathcal{E}}, \mathcal{E} = \mathcal{H}_{\mathcal{C}}(s) \\ \text{for some } x^{\mathcal{E}} \in \mathbb{R} \end{array} \right\}, \tag{4.1}$$

and the *event-wise affine adaptation* by

$$\bar{\mathcal{A}}(\mathcal{C}, \mathcal{I}) = \left\{ y : [S] \times \mathbb{R}^I \mapsto \mathbb{R} \;\middle|\; \begin{array}{l} y(s,z) = y^0(s) + \sum\limits_{i \in \mathcal{I}} y^i(s) z_i \\ \text{for some } y^0, y^i \in \mathcal{A}(\mathcal{C}),\ i \in \mathcal{I} \end{array} \right\}. \tag{4.2}$$

Intuitively speaking, the event-wise static adaptation $\mathcal{A}(\mathcal{C})$ is similar to the scenario-representation of uncertainty in stochastic optimization models, where recourse decisions take different values in each event of the collection $\mathcal{C}$. The event-wise affine adaptation $\bar{\mathcal{A}}(\mathcal{C}, \mathcal{I})$ further follows from the spirit of adaptive robust optimization models, and it is an affine mapping of components (indexed by $\mathcal{I} \subseteq \mathcal{I}_z$) of random variables $\tilde{\boldsymbol{z}}$, while the coefficients of the affine function take different values in each event.

In this chapter, we would use $\boldsymbol{w}$ to denote non-adaptive decisions. Event-wise static and affine recourse decisions are denoted by $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. Details of defining

these recourse decisions in RSOME are provided subsequently.

4.2 Defining Recourse Adaptations in RSOME

As mentioned in previous chapters, decision variables in RSOME models are declared by the method **decision**. By default, all decision variables are created to be non-adaptive. The event-wise adaptation of recourse decisions and their dependencies on random variables can be further specified by the following RSOME data structures.

4.2.1 Event Set and Affine Mapping

1. evtadapt

Method **evtadapt** defines the event-wise static adaptation according to the given indices of event $\mathcal{E}$. An error message would be reported if the events are not MECE.

x.**evtadapt**(**s**) adds scenarios indexed by **s** as a new event that the decision **x** adapts to.

2. affadapt

Method **affadapt** defines the affine mapping between random variables and adaptive decisions.

x.**affadapt**(**z**) defines the affine relation between decision **x** and random variables **z**.

x(**i**, **j**).**affadapt**(**z**(**m**, **n**)) defines affine relations between decision **x**(**i**, **j**) and random variables **z**(**m**, **n**).

The following code is provided to demonstrated how various types of recourse decisions are specified in RSOME.

```
model = rsome;                          % Create an RSOME model

%% Random variables and an ambiguity set with NS=10 scenarios
z = model.random(3, 5);                 % Random variables z
S = 10;                                 % Number of scenarios
P = model.ambiguity(S);                 % Create an ambiguity set
for s = 1:S
    P(s).suppset(z == rand(3, 5));      % Support of each scenario
end
model.with(P);                          % The ambiguity set of model is P

%% Non-adaptive decisions x
w = model.decision(2, 3);               % Non-adaptive decision w

%% Event-wise static adaptive decisions x
x = model.decision(2, 3);               % Decision x
for s = 1:S
    x.evtadapt(s);                      % x adapts to each scenario
end

%% Event-wise affinely adaptive decisions y
y = model.decision(2, 3);               % Decision y
y.evtadapt(1:3);                        % y adapt to an event with scenarios 1:3
y.evtadapt(4:5);                        % y adapt to an event with scenarios 4:5
```

```
y.evtadapt(6:10);                    % y adapt to an event with scenarios 6:10
y.affadapt(z(1:3));                  % y also affinely depends on z(1:3)
```

Variables **w** are here-and-now decisions by the default setting. Variables **x** and **y** are event-wise adaptive recourse decisions that adapt to each event, and variable **y** further affinely depend on a part of random variables **z**. Please note that these methods for specifying adaptations can only be called after the ambiguity set is defined.

4.2.2 Decision Tree for Multi-Stage Models

In the previous subsection, we introduced the method **evtadapt** for defining the event-wise adaptation of recourse decisions. In cases of multi-stage models, however, using **evtadapt** to define a lengthy scenario tree adaptation could be tedious and inefficient. Alternatively, RSOME provides a **tree** data structure to conveniently design scenario trees for multi-stage optimization models. Details of this data structure are presented as follows.

1. tree

As a method of ambiguity sets, **tree** creates a scenario tree data structure.
P.**tree**(**n**) creates a scenario tree with **n** stages which adapts to all scenarios defined in the ambiguity set **P** as the default setting.

2. mergechild

As a method of scenario trees, the method **mergechild** modifies the adaptation path by merging events at the next stage.
tree(**t**).**mergechild**(**indicesCell**) merges events at the stage **t+1** to form a new event partition of the decision tree structure **tree** at the stage **t** according to the event indices, which are collectively given by a cell array **indicesCell**.

3. plot

The method **plot** visualizes the decision tree structure.
tree.**plot** draws a diagram to display the structure of **tree**.

Note that at the final stage of the scenario tree, each scenario is an individual event. Event sets at precedent stages can be formed backwardly by recursively calling the method **mergechild**. An arbitrary tree can be thus constructed, as demonstrated by the following sample code.

```
model = rsome;                       % Create an RSOME model

z = model.random(4);                 % Random variables z

S = 8;                               % Number of scenarios
P = model.ambiguity(S);              % Create an ambiguity set with 8 scenarios
model.with(P);                       % The ambiguity set of the model is P

```

```
tree = P.tree(4);                % Create a scenario tree with 4 stages
tree(3).mergechild({1:2, 3:4, ...
                    5:7, 8});    % Merge stage 4 events to form stage 3 event set
tree(2).mergechild({1:2, 3:4}); % Merge stage 3 events to form stage 2 event set
tree(1).mergechild({1:2});       % Merge stage 2 events to form stage 1 event set

tree.plot;                       % Visualize the scenario tree
```

The scenario tree generated by the code above is shown as Figure 4.1. It can be seen that the third stage considers four events obtained from merging events according to the cell array **{1:2, 3:4, 5:7, 8}**. Likewise, event sets at the other two stages are constructed based on the cell arrays **{1:2, 3:4}** and **{1:2}**, respectively.

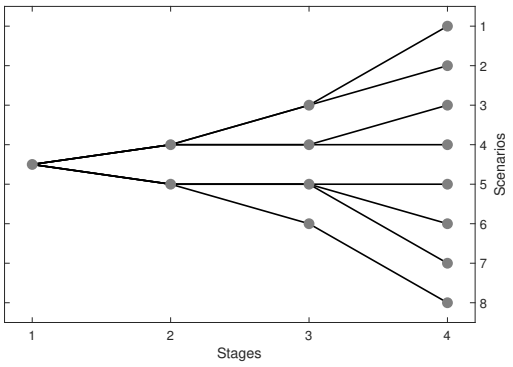

Figure 4.1: An example of a scenario tree

Based on the scenario tree, we can use the following methods to define adaptive recourse decisions that are associated with specific stages of the tree.

4. `decision`

As a method of the tree structure, **decision** creates new decision variables following the event-wise adaptation defined at each stage of the scenario tree

y = **tree**(**t**).**decision** creates a continuous decision variable **y** for each stage indexed by **t** of the scenario tree structure **tree**.

y = **tree**(**t**).**decision**(**N**) creates a column vector of **N** continuous decision variables **y** for each stage indexed by **t** of the scenario tree structure **tree**.

y = **tree**(**t**).**decision**(**N**, **M**) creates a matrix of **N**×**M** continuous decision variables **y** for each stage indexed by **t** of the scenario tree structure **tree**.

y = **tree**(**t**).**decision**(**N**, **M**, **type**) creates a matrix of **N**×**M** variables **y** for each stage indexed by **t** of the scenario tree structure **tree**, with **type** specifying the variable type.

y = **tree**(**t**).**decision**(**N**, **M**, **type**, **name**) creates a matrix of **N**×**M** variables **y** for each stage indexed by **t** of the scenario tree structure **tree**, with **type** specifying the variable type, and **name** specifying the name of the decision variables.

y = **tree**.**decision**(...) creates adaptive decisions **y** for each stage of the entire **tree**.

The method **decision** is especially convenient to define decisions that are repeatedly made across a number of stages. However, such a data structure as cross-stage decisions cannot be directly used in formulating models. RSOME expressions are only compatible with variables at a single stage. Selecting decision variables at one specific stage can be done by the method **stage**, and the syntax of applying **stage** is presented below.

5. stage

As a method of adaptive tree decisions, **stage** is used for isolating decision variables at a single stage so that they can be used in formulating RSOME models. When adaptive recourse decisions are defined for only one stage, the **stage** method can be omitted.

y.**stage**(**t**) returns variables of cross-stage decisions **y** at one stage indexed by **t**.

y(**i**).**stage**(**t**) returns a subset of variables via the index **i** of the cross-stage decision variables **y** at one stage indexed by **t**.

Based on the scenario tree created in the previous sample code, we can define recourse decisions associated with specific stages, as shown by the code segment below.

```
w = tree(1).decision(2, 2, 'B');                % Binary decisions w for stage 1
x = tree(2:3).decision(2, 1);                   % Decision x for stage 2 and 3
y = tree(3:4).decision(2, 3);                   % Decision y for stage 3 and 4

model.append(w(:, 1) + x.stage(2) == 0);        % w at stage 1 and x at stage 2
model.append(x.stage(3) <= y(:, 1).stage(3));   % x at stage 3 and y at stage 3
```

In this example, decisions **w** is only defined for the first stage, so we can directly apply it in constrains without calling the **stage** method. In contrast, both decisions **x** and **y** are defined for two stages. Hence, when they are applied in formulating constraints, we need to use the **stage** method to indicate the selected stage.

4.3 Computational Examples

■ **Example 4.3.1** In the first example, we address the multi-stage financial planning problem presented in [BL97] to illustrate the implementation of the scenario tree approach in RSOME. As a multi-stage model problem, the here-and-now decision allocates the total wealth d between two investment types: stocks (S) and bonds (B). Each investment type may have a high return and a low return as two possible outcomes. It is assumed that the high returns for stocks and bonds are 1.25 and 1.14, and the low returns are 1.06 and 1.12, respectively. These outcomes are independent and with equal likelihood, so throughout the subsequent stages, we would have eight scenarios with equal probabilities. The corresponding scenario tree is shown by Figure 4.2, where the constants $\boldsymbol{a}_i(s)$ are the returns of stocks and bonds at the stage $i+1$ in scenario s.

The uncertainty model with eight scenarios can be written as the following event-wise

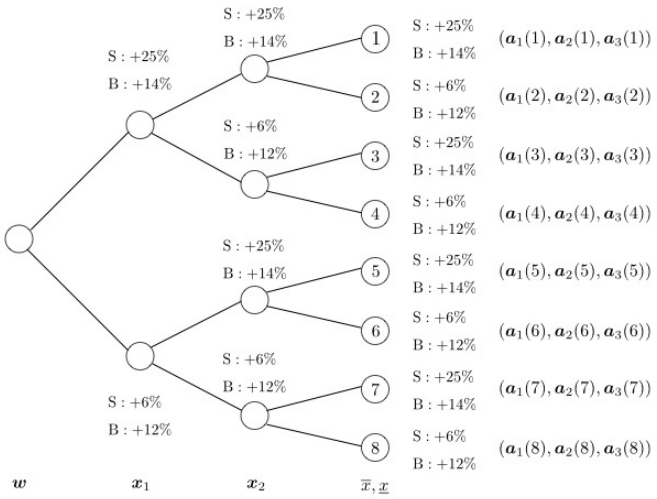

Figure 4.2: The scenario tree of the stochastic financial planning model

ambiguity set that only contains a singleton discrete distribution.

$$\mathcal{F} = \left\{ \mathbb{P} \in \mathcal{P}_0(\mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \times [S]) \;\middle|\; \begin{array}{ll} (\tilde{\boldsymbol{z}}_1, \tilde{\boldsymbol{z}}_2, \tilde{\boldsymbol{z}}_3, \tilde{s}) \sim \mathbb{P} & \\ \mathbb{P}\left[(\tilde{\boldsymbol{z}}_1, \tilde{\boldsymbol{z}}_2, \tilde{\boldsymbol{z}}_3) \in \mathcal{Z}_s | \tilde{s} = s\right] = 1 & \forall s \in [S] \\ \mathbb{P}[\tilde{s} = s] = \frac{1}{S} & \forall s \in [S] \end{array} \right\}, \quad (4.3)$$

where the scenario number $S = 8$ and the support sets $\mathcal{Z}_s = \{(\boldsymbol{a}_1(s), \boldsymbol{a}_2(s), \boldsymbol{a}_3(s))\}$ enforce the returns of stocks and bonds to be fixed values in each scenario $s \in [S]$.

Let $\tau = 80$ be a prescribed target and $r(s)$ be the final return, then the objective of the stochastic model is written as the expectation of a utility function

$$U(r(s), \tau) = \begin{cases} r(s) - \tau, & \text{if } r(s) \geq \tau \\ 4(r(s) - \tau), & \text{otherwise.} \end{cases} \quad (4.4)$$

The utility function can be written as the linear expression $\overline{x}(s) - 4\underline{x}(s)$ where $\overline{x}(s)$ and $\underline{x}(s)$ are auxiliary recourse decisions respectively indicating the excess over and shortfall below the target τ.

As shown by Figure 4.2, let $\boldsymbol{w}$ be the initial investment decisions, $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ be the rebalanced investment decisions respectively at the second and the third stage, then the

stochastic program can be formulated as the following RSO model:

$$
\begin{aligned}
\max\ & \inf_{\mathbb{P}\in\mathcal{F}} [\overline{x}(\tilde{s}) - 4\underline{x}(\tilde{s})] \\
\text{s.t. } & w_1, w_2 \geq 0,\ w_1 + w_2 = d \\
& x_{11}(s) + x_{12}(s) - \boldsymbol{z}_1^\top \boldsymbol{w} = 0 && \forall \boldsymbol{z} \in \mathcal{Z}_s, s \in [S] \\
& x_{21}(s) + x_{22}(s) - \boldsymbol{z}_2^\top \boldsymbol{x}_1(s) = 0 && \forall \boldsymbol{z} \in \mathcal{Z}_s, s \in [S] \\
& \boldsymbol{z}_3^\top \boldsymbol{x}_2(s) - \overline{x}(s) + \underline{x}(s) = \tau && \forall \boldsymbol{z} \in \mathcal{Z}_s, s \in [S] \\
& \boldsymbol{x}_1(s), \boldsymbol{x}_2(s), \overline{x}(s), \underline{x}(s) \geq 0 && \forall s \in [S] \\
& x_{11}, x_{12} \in \mathcal{A}(\{\{1,2,3,4\},\{5,6,7,8\}\}) \\
& x_{21}, x_{22} \in \mathcal{A}(\{\{1,2\},\{3,4\},\{5,6\},\{7,8\}\}) \\
& \overline{x}, \underline{x} \in \mathcal{A}(\{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{8\}\}).
\end{aligned} \tag{4.5}
$$

Notice that the last three lines in the formulation (4.5) state the event-wise adaptation of recourse decisions $\boldsymbol{x}_1(s)$, $\boldsymbol{x}_2(s)$, $\overline{x}(s)$ and $\underline{x}(s)$. The adaptation rules $\mathcal{A}$ with the corresponding MECE event sets can be defined by RSOME method **`evtadapt`**. Alternatively, it is more convenient to specify the multi-stage adaptation by the **`tree`** data structure, as demonstrated in the following code.

```
%% Parameters
S = 8;                                % Number of scenarios
A = 1 + [0.25, 0.14; ...
         0.06, 0.12];                 % Outcomes of investment returns
a = cell(1, S);                       % A cell array for uncertain returns
for s = 1:S
    s1 = ceil(s/4);
    s2 = mod(ceil(s/2)-1, 2) + 1;
    s3 = mod(s-1, 2) + 1;
    a{s} = [A(s1, :)', ...
            A(s2, :)', ...
            A(s3, :)'];               % Outcome of returns in scenario s
end
tau = 80;
d = 55;

%% Create an RSOME model
model = rsome('sp model');            % Create a model named "sp model"

%% Random variables and an ambiguity set with eight scenarios
z = model.random(2, 3);               % Random return of investments as z
P = model.ambiguity(S);               % Create a ambiguity set P
for s = 1:S
    P(s).suppset(z == a{s});          % Support of u as a fixed value
end
pr = P.prob;                          % pr as the vector of scenario probabilities
P.probset(pr == 1/S);                 % Equal probabilities of all scenarios
model.with(P);                        % Ambiguity set of model is P

%% Decisions and decision tree
tree = P.tree(4);                     % A decision tree with 4 stages
```

```
32 tree(3).mergechild({1:2, 3:4, ...
33                     5:6, 7:8});     % Define event set for stage 3
34 tree(2).mergechild({1:2, 3:4});     % Define event set for stage 2
35 tree(1).mergechild({1:2});          % Define event set for stage 1
36
37 w = tree(1).decision(2);            % Nonadaptive decisions as the root
38 x = tree(2:3).decision(2);          % Adaptive decisions x for stages 2 and 3
39 xo = tree(4).decision;              % Excess for stage 4
40 xu = tree(4).decision;              % Shortfall for stage 4
41
42 %% Objective function
43 model.max(expect(xo - 4*xu));       % Objective function as the expected utility
44
45 %% Constraints
46 model.append(w >= 0);
47 model.append(w(1) + w(2) == d);
48 model.append(x(1).stage(2) + x(2).stage(2) - z(:, 1)'*w == 0);
49 model.append(x(1).stage(3) + x(2).stage(3) - z(:, 2)'*x.stage(2) == 0);
50 model.append(z(:, 3)'*x.stage(3) - xo + xu == tau);
51 model.append(x.stage(2) >= 0);
52 model.append(x.stage(3) >= 0);
53 model.append(xu >= 0);
54 model.append(xo >= 0);
55
56 %% Solution
57 model.solve;                        % Solve the model
```

Notice that the scenario tree is defined in lines 31-35 and decisions with respect to the corresponding stages are specified in lines 37-40. The optimal objective value of this stochastic financial planning problem is -1.5141. ■

■ **Example 4.3.2** The second example attempts to solve the same newsvendor problem discussed in Section 3.3.2 that considers the Wasserstein ambiguity set. Unlike the model in Section 3.3.2 that considers the exact evaluation of the term $\max\{p \cdot (w-u)\}$, the following model introduce an auxiliary recourse decision for this term and applies the event-wise recourse adaptation to approximate the recourse decision:

$$\begin{aligned}
\max \quad & (p-c)w - \sup_{\mathbb{P}\in\mathcal{F}} \mathbb{E}_{\mathbb{P}}[y(\tilde{s},(\tilde{u},\tilde{v}))] \\
\text{s.t.} \quad & w \in \mathbb{R}_+ \\
& y(s,(u,v)) \geq p(w-u), && \forall z \in \mathcal{Z}, v \geq \rho(u,\hat{u}_s), s \in [S] \\
& y(s,(u,v)) \geq 0, && \forall z \in \mathcal{Z}, v \geq \rho(u,\hat{u}_s), s \in [S] \\
& y \in \bar{\mathcal{A}}(\mathcal{C},[2]), && (4.6)
\end{aligned}$$

where the collection of singleton MECE events $\mathcal{C} = \{\{s\} \mid s \in [S]\}$ and $\bar{\mathcal{A}}(\mathcal{C},[2])$ suggests that y adapts to each scenario and also affinely depends on random variables u and v. The newsvendor model can be then implemented by the code below.

```
1 %% Parameters
2 Ubar = 100;                              % Upper bounds of random demands
```

```
S = 500;                                    % Number of samples
Uhat = Ubar * rand(1, S);                   % Empirical demand realizations

p = 1.5;                                    % Unit selling price
c = 1.0;                                    % Unit ordering price
theta = Ubar*0.01;                          % Radius of the Wasserstein ball

%% Create an RSOME model
model = rsome('newsvendor');                % Create a model named "newsvendor"

%% Random variables and a Warsserstein ambiguity set
u = model.random;                           % Random demand
v = model.random;                           % Auxiliary random variable
P = model.ambiguity(S);                     % Create an ambiguity set P
for n = 1:S
    P(n).suppset(0 <= u, u <= Ubar, ...
                 norm(u - Uhat(n)) <= v); % Support set for each scenario
end
P.exptset(expect(v) == theta);
pr=P.prob;                                  % pr for all scenario probabilities
P.probset(1/S == pr);                       % The probability set
model.with(P);                              % Ambiguity set of the model is P

%% Decisions and event-wise recourse adaptation
w = model.decision;                         % Non-adaptive decision variable w
y = model.decision;                         % Recourse decision y
for n = 1:S
    y.evtadapt(n);                          % Decision y adapts to each scenario
end
y.affadapt(u);
y.affadapt(v);                              % Decision y affinely depends on (u, v)

model.max((p-c)*w - expect(y));             % Worst-case expectation
model.append(y >= 0);
model.append(y >= p * (w-u));
model.append(w >= 0);                       % Variable w is non-negative

%% Solution
model.solve;                                % Solve the model
```

The model above would recover the same optimal solution as in the exact approach when there is only one product. For multi-product cases, the exact approach may not be practically favorable because the problem size grows exponentially as the number of products increases. In such cases, the event-wise recourse adaptation could be used to achieve a slightly conservative but much more scalable solution. ■

■ **Example 4.3.3** This example considers a medical scheduling problem based on [BSZ18] where a clinic is assigning appointment time for N patients. Let $\boldsymbol{w} \in \mathbb{R}^N_+$ be the here-and-now decision as the appointment time and $\tilde{\boldsymbol{u}} \in \mathbb{R}^N_+$ be the random service time of all patients. The scheduling problem can be written as a distributionally robust model with an objective of min $\sup_{\mathbb{P}\in\bar{\mathcal{F}}}[Q(\boldsymbol{w}, \tilde{\boldsymbol{u}})]$, where the function $Q(\boldsymbol{w}, \boldsymbol{u})$ represents the total waiting cost given the schedule $\boldsymbol{w}$ and the outcome $\boldsymbol{u}$ of the random service time. The

function $Q(\boldsymbol{w}, \boldsymbol{u})$ is expressed as the minimization problem below:

$$\begin{aligned} Q(\boldsymbol{w}, \boldsymbol{u}) = \min \ & \sum_{i\in[N]} y_i + c y_{N+1} \\ \text{s.t. } & y_{i+1} \geq y_i + u_i - w_i, && \forall i \in [N] \\ & y_i \geq 0, && \forall i \in [N] \\ & x_i \geq 0, && \forall i \in [N] \\ & \sum_{i\in[N]} x_i \leq T, \end{aligned} \tag{4.7}$$

To demonstrate the modeling power of RSOME, three ambiguity sets and their corresponding event-wise affine adaptation policies are adopted in the following experiments. In these experiments, we consider $N = 8$ patients and the cost parameter $c = 2$. For each patient, the random service time has a mean value μ_i generated uniformly over $[30, 60]$ and a standard deviation σ_i generated uniformly over $[0, 0.3\mu_i]$. It is then assumed that the random service time follows a normal distribution $\mathbb{N}(\mu_i, \sigma_i)$ and the total working hours of the physician are set to be $T = \sum_{i=1}^{N} \mu_i + 0.5\|\boldsymbol{\sigma}\|_2$.

1. Partial cross moments ambiguity set: the paper [BSZ18] proposed an ambiguity set considering partial cross moment information:

$$\bar{\mathcal{F}} = \left\{ \mathbb{P} \in \mathcal{P}_0(\mathbb{R}^N) \;\middle|\; \begin{array}{ll} \tilde{\boldsymbol{u}} \sim \mathbb{P} & \\ \mathbb{E}_{\mathbb{P}}\left[\tilde{\boldsymbol{u}}\right] = \boldsymbol{\mu} & \\ \mathbb{E}_{\mathbb{P}}\left[\tilde{u}_i^2\right] \leq \sigma_i^2 & \forall i \in [N] \\ \mathbb{E}_{\mathbb{P}}\left[\left(\sum_{i\in[N]} \tilde{u}_i\right)^2\right] \leq \sum_{i\in[N]} \sigma_i^2 & \\ \mathbb{P}\left[\tilde{\boldsymbol{u}} \in \mathbb{R}_+^N\right] = 1 & \end{array} \right\}. \tag{4.8}$$

Introducing an auxiliary variables $\tilde{\boldsymbol{v}}$ to express the epigraph of partial moments, we may rewrite the ambiguity set $\bar{\mathcal{F}}$ into its lifted form $\mathcal{F}$ as an event-wise ambiguity set with only one scenario:

$$\mathcal{F} = \left\{ \mathbb{P} \in \mathcal{P}_0(\mathbb{R}^N \times \mathbb{R}^{N+1} \times [1]) \;\middle|\; \begin{array}{ll} ((\tilde{\boldsymbol{u}}, \tilde{\boldsymbol{v}}), \tilde{s}) \sim \mathbb{P} & \\ \mathbb{E}_{\mathbb{P}}\left[\tilde{\boldsymbol{u}} \middle| \tilde{s} = 1\right] = \boldsymbol{\mu} & \\ \mathbb{E}_{\mathbb{P}}\left[v_i \middle| \tilde{s} = 1\right] \leq \sigma_i^2 & \forall i \in [N] \\ \mathbb{E}_{\mathbb{P}}\left[v_{N+1} \middle| \tilde{s} = 1\right] \leq \sum_{i\in[N]} \sigma_i^2 & \\ \mathbb{P}\left[(\tilde{\boldsymbol{u}}, \tilde{\boldsymbol{v}}) \in \mathcal{Z} \middle| \tilde{s} = 1\right] = 1 & \\ \mathbb{P}\left[\tilde{s} = 1\right] = 1 & \end{array} \right\}, \tag{4.9}$$

where $\mathcal{Z}$ is the lifted support of $(\tilde{\boldsymbol{u}}, \tilde{\boldsymbol{v}})$, given by

$$\mathcal{Z} = \left\{ (\boldsymbol{u}, \boldsymbol{v}) \in \mathbb{R}^N \times \mathbb{R}^{N+1} \;\middle|\; \begin{array}{ll} \boldsymbol{u} \in \mathbb{R}_+^N & \\ u_i^2 \leq v_i, & \forall i \in [N] \\ \sum_{i\in[N]} u_i^2 \leq v_{N+1} & \end{array} \right\}. \tag{4.10}$$

The event-wise recourse adaptation is thus defined as $y_i \in \bar{\mathcal{A}}([1],[2N+1])$, for all $i \in [N]$, implying that y affinely depends on $\boldsymbol{u}$ and $\boldsymbol{v}$. The RSOME code for the partial cross moment ambiguity set and the corresponding adaptation rule is provided below.

```
%% Parameters
N = 8;                                          % The number of patients
c = 2;                                          % Cost parameter

mu = 30 + 30*rand(N, 1);                        % Mean values of service time
sigma = 0.3*mu .* rand(N, 1);                   % Std. deviations of service time
T = sum(mu) + 0.5*norm(sigma);                  % Total working hours

%% Create an RSOME model
model = rsome;                                  % Create a model

%% Random variables and a partial cross moment ambiguity set
u = model.random(N);                            % Random service time
v = model.random(N+1);                          % Auxiliary random variables
P = model.ambiguity;                            % Create an ambiguity set
P.suppset(u >= 0, ...
          (u-mu).^2 <= v(1:end-1),...
          sum((u-mu)).^2 <= v(end));            % Lifted support for u
P.exptset(expect(u) <= mu, ...
          expect(v(1:end-1)) <= sigma.^2, ...
          expect(v(end)) <= sum(sigma.^2));     % Partial cross moments
model.with(P);

%% Decisions and event-wise recourse adaptation
w = model.decision(N);                          % Appointment time
y = model.decision(N+1);                        % Recourse decision y as waiting time
y.affadapt(u);                                  % y affinely depends on u
y.affadapt(v);                                  % y affinely depends on v

%% Objective function
model.min(expect(sum(y(1:end-1)) + c*y(end)));

%% Constraints
model.append(y(2:end) >= y(1:end-1) + u - w);
model.append(y >= 0);
model.append(sum(w) <= T);
model.append(w >= 0);

%% Solution
model.solve;                                    % Solve the model
```

2. Type-1 Wasserstein ambiguity set: given demand realizations $\hat{\boldsymbol{u}}_s$ of the random demand $\tilde{\boldsymbol{u}}$, we are using the lifted representation provided in [CSX19] to construct the following type-1 Wasserstein ambiguity set:

$$\bar{\mathcal{F}} = \left\{ \mathbb{P} \in \mathcal{P}_0(\mathbb{R}^N \times [S]) \;\middle|\; \begin{array}{ll} (\tilde{\boldsymbol{u}}, \tilde{s}) \sim \mathbb{P} & \\ \mathbb{E}_{\mathbb{P}}\left[\rho(\tilde{\boldsymbol{u}}, \hat{\boldsymbol{u}}_{\tilde{s}}) \middle| \, \tilde{s} \in [S]\right] \leq \theta & \forall s \in [S] \\ \mathbb{P}\left[\tilde{\boldsymbol{u}} \in \mathcal{U} \middle| \, \tilde{s} = s\right] = 1 & \forall s \in [S] \\ \mathbb{P}\left[\tilde{s} = s\right] = \frac{1}{S} & \end{array} \right\}, \tag{4.11}$$

where θ is the radius of the Wasserstein ball. The lifted form is derived as follows.

$$\mathcal{F} = \left\{ \mathbb{P} \in \mathcal{P}_0(\mathbb{R}^N \times \mathbb{R} \times [S]) \;\middle|\; \begin{array}{ll} ((\tilde{\boldsymbol{u}}, \tilde{v}), \tilde{s}) \sim \mathbb{P} & \\ \mathbb{E}_{\mathbb{P}}\left[\tilde{v} | \, \tilde{s} \in [S]\right] \leq \theta & \forall s \in [S] \\ \mathbb{P}\left[(\tilde{\boldsymbol{u}}, \tilde{v}) \in \mathcal{Z}_s | \, \tilde{s} = s\right] = 1 & \forall s \in [S] \\ \mathbb{P}\left[\tilde{s} = s\right] = \frac{1}{S} & \end{array} \right\}, \tag{4.12}$$

where $\tilde{v}$ is an auxiliary variable and the expression of the lifted support is given by

$$\mathcal{Z}_s = \left\{ (\boldsymbol{u}, v) \in \mathbb{R}^N \times \mathbb{R} \;\middle|\; \begin{array}{l} \boldsymbol{u} \in \mathbb{R}^N_+ \\ \rho(\tilde{\boldsymbol{u}}, \hat{\boldsymbol{u}}_s) \leq v \end{array} \right\} \quad \forall s \in [S]. \tag{4.13}$$

The event-wise affine adaptation is then designed as $y_i \in \bar{\mathcal{A}}([S], [N+1])$, suggesting that the recourse decisions adapt to each scenario $s \in [S]$ and are affinely dependent on $\tilde{\boldsymbol{u}}$ and $\tilde{v}$. The corresponding code is provided as follows.

```
%% Parameters
N = 8;                                          % The number of patients
c = 2;                                          % Cost parameter
S = 256;                                        % Sample size N
theta = S^(-1/8);                               % Robustness parameter

mu = 30 + 30*rand(N, 1);                        % Mean values of service time
sigma = 0.3*mu .* rand(N, 1);                   % Std. deviations of service time
T = sum(mu) + 0.5*norm(sigma);                  % Total working hours

Uhat = mu*ones(1, S) + ...
       diag(sigma)*randn(N, S);                 % Randomly generated sample

%% Create an RSOME model
model = rsome;                                  % Create a model

%% Random variables and the type-one Wasserstein ambiguity set
u = model.random(N);                            % Random service time
v = model.random;                               % Auxiliary random variables
P = model.ambiguity(S);                         % Create an S-scenario ambiguity set
for s = 1:S
    P(s).suppset(u >= 0, ...
                 norm(u-Uhat(:, s)) <= v);      % Conditional lifted support for u
end
P.exptset(expect(v) <= theta);                  % Expectation of u
pr = P.prob;                                    % Vector of scenario probabilities
P.probset(pr == 1/S);                           % Set of scenario probabilities
model.with(P);

%% Decisions and event-wise recourse adaptation
w = model.decision(N);                          % Appointment time
y = model.decision(N+1);                        % Recourse decision y as waiting time
for s = 1:S
    y.evtadapt(s);                              % Adaptation to each scenario
end
y.affadapt(u);                                  % y affinely depends on u
y.affadapt(v);                                  % y affinely depends on v

```

```
%% Objective function
model.min(expect(sum(y(1:end-1)) + c*y(end)));

%% Constraints
model.append(y(2:end) >= y(1:end-1) + u - w);
model.append(y >= 0);
model.append(sum(w) <= T);
model.append(w >= 0);

%% Solution
model.solve;                                % Solve the model
```

3. Type-∞ Wasserstein ambiguity set: in a recent paper, [BSS19] proposed a sample robust optimization framework considering a type-∞ Wasserstein ambiguity set together with a multi-policy approximation to address two-stage decision-making problems. [BSS19] have shown that such an ambiguity set can also be mapped into the format of the event-wise ambiguity set as follows:

$$\bar{\mathcal{F}} = \left\{ \mathbb{P} \in \mathcal{P}_0(\mathbb{R}^N \times [S]) \;\middle|\; \begin{array}{l} (\tilde{\boldsymbol{u}}, \tilde{s}) \sim \mathbb{P} \\ \mathbb{P}\left[\tilde{\boldsymbol{u}} \in \mathbb{R}^N_+, \rho(\tilde{\boldsymbol{u}}, \hat{\boldsymbol{u}}_{\tilde{s}}) \leq \theta \middle| \, \tilde{s} = s\right] = 1 \quad \forall s \in [S] \\ \mathbb{P}\left[\tilde{s} = s\right] = \frac{1}{S} \end{array} \right\}. \quad (4.14)$$

Quite interestingly, the multi-policy approximation coincides with the event-wise recourse adaptation $y_i \in \bar{\mathcal{A}}([S], [N])$, meaning that the recourse decision adapts to each scenario $s \in [S]$ and is affinely dependent on random variables $\tilde{\boldsymbol{u}}$. The sample robust optimization model can be conveniently implemented through the RSOME code below.

```
%% Parameters
N = 8;                                      % The number of patients
c = 2;                                      % Cost parameter
S = 256;                                    % Sample size S
theta = S^(-1/8);                           % Robustness parameter

mu = 30 + 30*rand(N, 1);                    % Mean values of service time
sigma = 0.3*mu .* rand(N, 1);               % Std. deviations of service time
T = sum(mu) + 0.5*norm(sigma);              % Total working hours

Uhat = mu*ones(1, S) + ...
       diag(sigma)*randn(N, S);             % Randomly generated sample

%% Create an RSOME model
model = rsome;

%% Random variables and the type-infinity Wasserstein ambiguity set
u = model.random(N);
P = model.ambiguity(S);                     % Create an S-scenario ...
    ambiguity set
for s = 1:S
    P(s).suppset(u >= 0, ...
                 norm(u-Uhat(:, s)) <= theta); % Conditional support for u
end
pr = P.prob;                                % Vector of scenario probabilities
P.probset(pr == 1/S);                       % Set of scenario probabilities
```

```
model.with(P);

%% Decisions and event-wise recourse adaptation
w = model.decision(N);                          % Appointment time
y = model.decision(N+1);                        % Recourse decision y as ...
    waiting time
for s = 1:S
    y.evtadapt(s);                              % Adaptation to each scenario
end
y.affadapt(u);                                  % y affinely depends on u

%% Objective function
model.min(expect(sum(y(1:end-1)) + c*y(end)));

%% Constraints
model.append(y(2:end) >= y(1:end-1) + u - w);
model.append(y >= 0);
model.append(sum(w) <= T);
model.append(w >= 0);

%% Solution
model.solve;                                    % Solve the model
```

■

4.4 Summary

In this chapter, we discuss the implementation of adaptive models with the RSOME toolbox. Our toolbox enables users to define sophisticated recourse adaptations in terms of MECE event sets and affine dependencies. Although the event-wise recourse adaptation is a conservative approximation, it is considered more practical in addressing various adaptive problems due to its good scalability.

Bibliography

[BN99] Aharon Ben-Tal and Arkadi Nemirovski. "Robust solutions of uncertain linear programs". In: *Operations Research Letters* 25.1 (1999), pages 1–13 (cited on page 10).

[BS04] Dimitri Bertsimas and Melvyn Sim. "The price of robustness". In: *Operations Research* 52 (2004), pages 35–53 (cited on pages 10, 14).

[BSS19] Dimitris Bertsimas, Shimrit Shtern, and Bradley Sturt. *Two-Stage Sample Robust Optimization*. 2019. URL: https://arxiv.org/abs/1907.07142 (cited on page 31).

[BSZ18] Dimitris Bertsimas, Melvyn Sim, and Meilin Zhang. "Adaptive distributionally robust optimization". In: *Management Science* 65.2 (2018), pages 604–618 (cited on pages 27, 28).

[BL97] J.R. Birge and F. Louveaux. *Introduction to stochastic programming*. Springer Verlag, 1997 (cited on page 23).

[CSX19] Zhi Chen, Melvyn Sim, and Peng Xiong. *Robust stochastic optimization*. 2019. URL: http://www.optimization-online.org/DB_HTML/2017/06/6055.html (cited on pages 5, 12, 16, 29).

[MK18] Peyman Mohajerin Esfahani and Daniel Kuhn. "Data-driven distributionally robust optimization using the Wasserstein metric: performance guarantees and tractable reformulations". In: *Mathematical Programming* 171.1-2 (2018), pages 115–166 (cited on page 16).