XProg 1.0 Users' Guide

MATLAB Toolbox for Optimization under Uncertainty

Peng Xiong

xiongpengnus@gmail.com

October 17, 2016

Contents

1 INTRODUCTION

XProg is a toolbox designed for constructing optimization models under uncertainty in MATLAB environment (version 2012a or above). It is capable of solving deterministic, stochastic, robust, and distributionally robust optimization problems [1, 2, 3]. Inspired by decision rule techniques [1, 2, 4, 5], this toolbox also enables modeling adjustable recourse decisions in multistage problems as tractable decision rule approximations.

Being consistent with the MATLAB matrix computing syntax, XProg is extremely easy to implement, and is compatible with other MATLAB numerical and graphic functions. Aimed at solving large-scale optimization problems, this tool box is developed based on highly efficient matrix processing functions, and it is capable of exploiting the special structure of uncertainty sets to reduce computational costs. The IBM ILOG CPLEX mixed-integer optimizer is utilized in the current version to solve models in the forms of linear, quadratic, or second-order cone programs.

Throughout this Users' Guide, 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 in the toolbox are written as "`functions`", while the input and output arguments are written as "`arguments`".

The subsequent chapters are organized as follows. The next chapter presents the configuration, data structure, and the basic functions of this toolbox. A few deterministic and stochastic programming examples are utilized to demonstrate the implementation of XProg. Chapter 3 presents the general procedure of solving robust optimization problems, including creating random variables and uncertainty sets, as well as defining decision rule functions for adjustable decisions. The final chapter discusses how to model and solve distributionally robust optimization problems.

2 START WITH XPROG

2.1 Preparation

XProg works under both 32-bit and 64-bit MATLAB, version 2012a or above. It requires to install IBM ILOG CPLEX as the mixed-integer optimizer. This software is a popular commercial solver and is free for academic users. The current version of XProg is compatible with CPLEX 12.6.3/12.6.2/12.6.1. It should be noted that a 32-bit CPLEX solver is only compatible with the 32-bit MATLAB, and the 64-bit CPLEX solver only works under 64-bit MATLAB. If a different version of CPLEX solver is used, then a compatible C/C++ compiler, such as the Visual Stuido, is required to compile the solver. It is strongly recommended the user install CPLEX Optimization Studio 12.6.3 in the default folder, so that XProg may be used without any extra configuration. Before the first use of XProg, the users need to go through the following steps to configure the toolbox.

2.2 Configuration

XProg uses IBM ILOG CPLEX Optimization Studio 12.6.3/12.6.2/12.6.1 as the solver. If these solvers have been installed under the default directories, they can be easily linked with our toolbox. The users only need to run the interactive input program **`configure`** in the command window. As illustrated by Figure, 2.1, you may then select the version of CPLEX optimizer you are using. Then program is going to ask if the selected solver is installed in the default directory. Once the users type "y", the program completes the configure process.

If other versions of CPLEX optimizer are used, or if these solvers are installed in other directories, the program **`configure`** helps the users to compile the solver function once the version and the directory of the solver are specified. Please note that in this case, you need to install a compatible C/C++ compiler in you computer, and they compiler should be activated beforehand by calling "mex-setup" at the MATLAB command prompt. Detailed procedure is given in Figure 2.2.

The final step of configuring XProg is to copy the "dll" file "cplexXXXX.dll" from the directory "CPLEX_root_directory\CPLEXXXX\cplex\bin" to the "functions" folder of the XProg toolbox, where

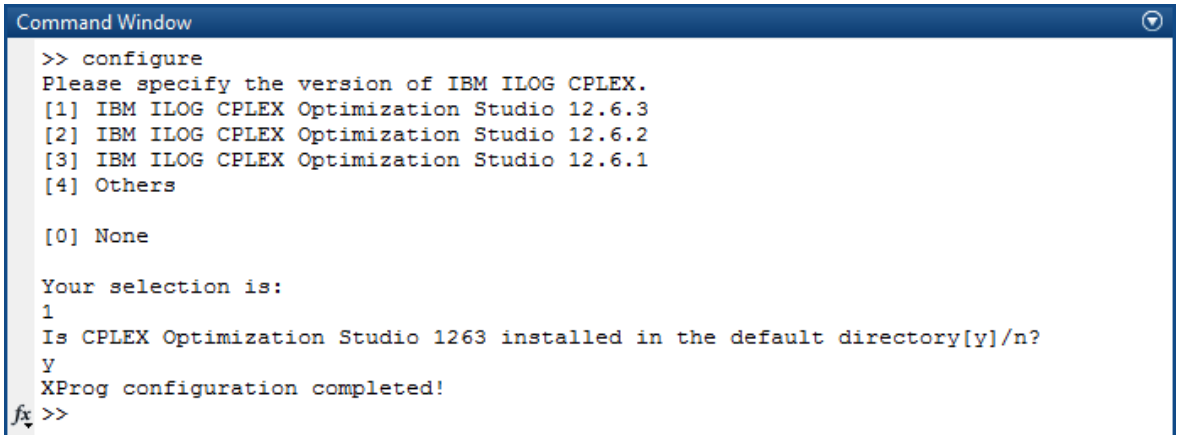

Figure 2.1: Configure XProg for CPLEX 12.6.3 installed in the default directory

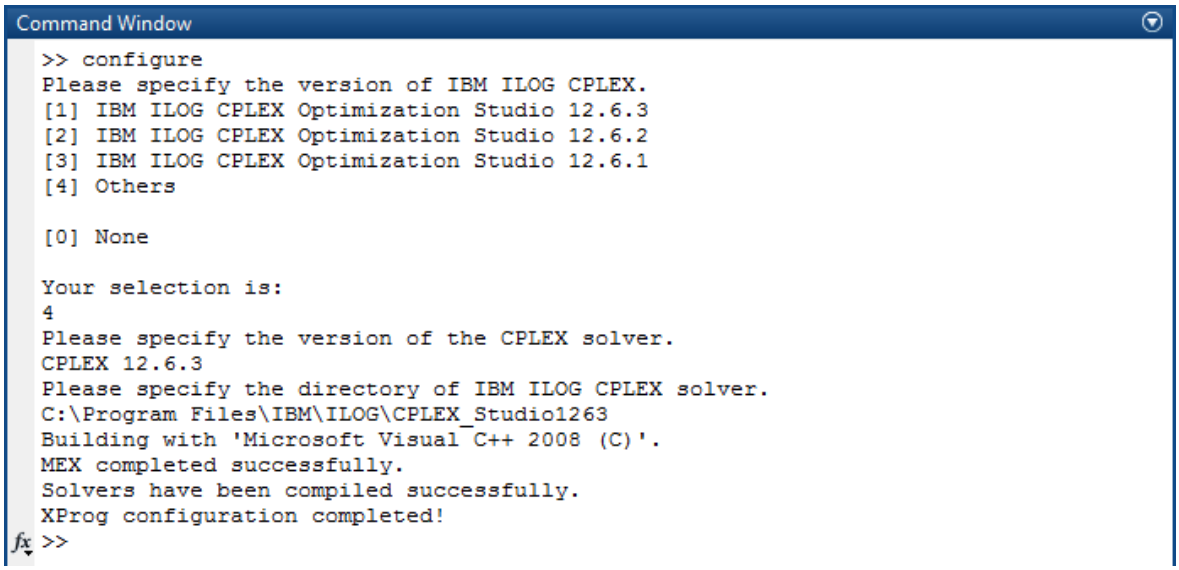

Figure 2.2: Configure XProg for user specified solver

"XXXX" represents the version number of the CPLEX optimizer. Notice that the configuration is only required before the first use of XProg. There is no need to repeat these steps for future implementation, unless a different version of CPLEX optimizer is used.

2.3 Basics about XProg

2.3.1 XProg model, decisions, and constraints

Following objects and functions are defined in XProg toolbox to construct and to solve optimization models.

1) start_xprog and end_xprog

Function **start_xprog** is used to includes the XProg toolbox into the working directory, and it is required before running any code based on XProg. Once the toolbox is included into the working directory, users can excute XProg programs in any folders.

Function **end_xprog** removes XProg from the current working directory. This function is used if there is a contradiction of function or object names with other toolboxes. Names of all XProg objects are listed in the table below.

Table 2.1. Objects of the XProg Toolbox

Names	Definitions
absexp	absolute value of a linear affine expression
bound	bounds of decision variables or random variables
constraint	constraints
drcconstraint	robust counterpart of uncertain constraints with expectation terms
exp	exponential function of a linear affine expression
expconstraint	constraints involving expected terms
explinfun	expectation of an uncertain linear function
explinfun	expectation of an uncertain linear function
expoconstraint	exponential constraints
linfun	linear functions
linoptim	deterministic optimization program
normconstraint	second-order cone constraints
normexp	second-order cone expressions
qexpconstraint	conic exponential quadratic constraints
qexpfun	conic exponential quadratic functions
rcconstraint	robust counterpart of uncertain constraints
refrule	decision rule functions with indices
rule	decision rule functions
sqconstraint	a constraint with squared term
sqconstraints	constraints with squared term
sqexp	sqaured expressions
sqmat	sqaured expressions in matrix form
subsets	confidence sets
unconstraints	uncertain constraints
unconstraints	uncertain linear functions
variables	decision variables or random variables
xprog	xprog model

2) xprog

Function **xprog** creates a new XProg optimization model object.

model = **xprog** creates a optimization model object **model**.

model = **xprog**(**name**) creates a optimization object **model** with a user defined title **name**.

3) decision

Function **decision** defines new decision variables for the model object.

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

x = **model.decision**(**N**) defines continuous decision variables as an $N \times 1$ vector **x** for **model**.

x = **model.decision**(**N**, **M**) defines continuous decision variables as an $N \times M$ matrix **x** for **model**.

x = **model.decision**(**N**, **M**, **type**) defines decision variables as an $N \times M$ matrix **x** for **model**. These variable are continuous if **type**=0, binaries if **type**=1, and general integers if **type**=2.

x = **model.decision**(**N**, **M**, **type**, **name**) defines decision variables as a matrix **x** for **model**. The dimension of the matrix is specified by **N** and **M**. The type of variables is assigned by **type**. The name of this set of variables is given as **name**.

4) add

Function **add** is used to include a set of constraints into the model object.

model.add(**constraint**) includes **constraint** into optimization model object **model**. The argument **constraint** can be any deterministic quadratic representable constraints, or linear constraints involving random variables or expectation terms.

model.add(**constraint**, **name**) includes **constraint** into optimization model object **model**. The name of these constraints is assigned to be **name**.

5) min and max

Functions **min** and **max** are used to define the objective function. The objective function must be a 1×1 expression. Note that it is not allowed to define objective functions more than once for the same model. An error message will be given if the objective function is redefined.

model.min(**Fun**) minimizes the objective function **Fun**.

model.max(**Fun**) maximizes the objective function **Fun**.

6) solve

Function **solve** solves the optimization problem constructed by the XProg toolbox.

model.solve is used to solve the optimization problem modeled as object **model**. The relative mixed-

integer programming (MIP) gap is set to be 10^{-4} if the problem contains integer variables.

model.solve(MIPGAP) solves the optimization problem **model**. The solution program terminates if the MIP gap is smaller than the tolerance **MIPGAP**. Note that MIP gap tolerance is ineffective if all decision variables of **model** are continuous.

7) get

Function **get** is used to retrieve the objective value of the model, or the optimal solution of selected decision variables.

model.get gets the objective value of **model**.

x.get gets the solution of decision variable matrix **x**.

2.3.2 Matrix indexing and arithmetic operations

XProg applies the same indexing system as MATLAB to access decision variables in matrices. Examples below are provided to demonstrate how to formulate constraints for indexed decision variables.

```
model=xprog;                        % create a model;

x=model.decision(5,6);              % define a 5*6 decision variable matrix

model.add(x>=0);                    % each element of x is nonnegative
model.add(x>=zeros(5,6));           % same as above
model.add(3*x(1,5)<=10);            % 3*x_15<=10
model.add(x(1,:)<=x(5,:));          % the 1st row of x is no larger than the 5th
model.add(x(:,end-1:end)<=1);       % the last two columns are no larger than 1
model.add(x([1 3 5],[2 4])<=1);     % x_12, x_32, x_52, x_14, x_34, x_54 <=1
model.add(x(8)<=x(14));             % x_32<=x_43
```

Besides, XProg is consistent with MATLAB in most array and matrix operations, e.g. addition, substraction, sum, matrix and element-wise multiplication, etc. The current version does not support any forms of division or power, except the element-wise square. Some instances are given below.

```
model=xprog;                        % create a model;

x=model.decision(5);                % define a 5*1 decision variable matrix
y=model.decision(3);                % define a 3*1 decision variable matrix

A=ones(3,5);                        % A is a 3*5 matrix
b=ones(5,1);                        % b is a 5*1 vector
c=rand(3,1);                        % c is a 3*1 vector

model.add(A*x+b>=0);                % matrix multiplication A*x
model.add(sum(x)-y'*c==1);          % transpose of y, sum of each row of x
```

```
model.add(c.*y<=2)                    % element-wise multiplication c.*y
model.add(sum(b,2)==sum(c,2));        % sum of each column of b and c
```

Finally, matrices and vectors of decision variables, random variables, and their linear affine expressions can also be concatenated with one another, and the dimension of matrices and vectors can be modified by using function **reshape**. Specific examples are provided as follows.

```
model=xprog;                          % create a model;

x=model.decision(8,2);                % define a 2*8 decision variable matrix
y=model.decision(2,8);                % define a 2*8 decision variable matrix

model.add([x;y(:,2:3)]>=0);           % vertical concatenation...
                                      % x and y(:,2:3) have the same number of rows
model.add([x y']==1);                 % horizontal concatenation...
                                      % x and y' have the same number of columns
model.add(reshape(y,4,4)>=eye(4));    % reshape y into a 4*4 matrix...
                                      % the number of elements must not change
```

2.3.3 Convex functions

Some commonly used convex functions are incorporated into the XProg toolbox. Note that in order to maintain convexity, convex functions are only allowed to be less or equal to a linear expression. Exceptions lead to an error message indicating that the constraints are nonconvex.

1) abs

Function **abs** returns the absolute value of a linear affine expression.

abs(**fun**) returns the absolute value of a linear affine function expressed as matrix **fun**.

2) norm

Function **norm** returns the Frobenius norm of a linear affine expression.

norm(**fun**) returns the Frobenius norm of a linear affine expression **fun** as a matrix form. Suppose **fun** is a matrix in the form of $\boldsymbol{C} = \boldsymbol{AX} + \boldsymbol{B}$, where $\boldsymbol{C} \in \mathbb{R}^{M\times N}$, then the function **norm** returns $\sqrt{\sum_{i=1}^{M}\sum_{j=1}^{N}|C_{ij}|^2} = \sqrt{\mathbf{Tr}\,(\boldsymbol{CC}^T)}$.

3) square

Function **aquare** returns the square of a linear affine expression.

square(**fun**) returns the square of a linear affine expression **fun** as a vector. Suppose **fun** is a vector in the form of $Ax + b$, then this function returns $(Ax + b)^T(Ax + b)$.

4) element-wise square: .^2

Operator "**.^2**" returns the element-wise square of a matrix.

(fun).^2 returns the element-wise square of a linear affine expression **fun**. The input argument **fun** is generally in a matrix form $\boldsymbol{C} = \boldsymbol{AX} + \boldsymbol{B}$.

5) exp

Function "**exp**" approximates the exponential function of a linear affine expression by second-order conic expressions.

exp(Fun) approximates the exponential function of the 1linear affine expression **Fun**. The accuracy of the approximation depends on the parameter **model.Param.ExpL**.

6) quadexpcone

Function "**quadexpcone**" approximates the quadratic exponential cone function proposed in reference [6] by second-order conic expressions.

quadexpcone(a,b,c) approximates the expression $\inf_{c>0} \mu \exp\left(\frac{a}{c} + \frac{b^2}{c^2}\right)$, where the input arguments **a**, **b**, and **c** are all 1×1 variables or linear affine functions.

Examples of using these convex functions to construct constraints are provided as follows.

```
model=xprog;                          % create a model

x=model.decision(8,2);                % define a 2*8 decision variable matrix
y=model.decision(2,8);                % define a 2*8 decision variable matrix

A=ones(1,8);                          % A is a 3*8 matrix
B=ones(3,2);                          % B is a 3*2 matrix

model.add(abs(x)<=y');                % absolute value of x
model.add(y(1)>=norm(x));             % norm of x
model.add(-square(A*x)+4>=0);         % square of A*x, which is equivalent to (A*x)'*(A*x)
model.add(x(1:3,:).^2-B<=0);          % element-wise square of x(1:3,:)
```

2.3.4 Mathematical formulation of XProg models

XProg allows users to access important data of the optimization formulation, such as problem dimensions, constraint matrices, types and bounds of decision variables, etc. Such data can be returned in a MATLAB structure data format by function **problem**. We also developed a function **showsparse** to reveal the sparsity of linear and second-order cone constraint matrices. Details of these two functions are provided

as follows.

1) problem

Function **problem** returns the data of the optimization problem.

Problem=**model.problem** returns a MATLAB structure of problem data for **model**.

Details of this structure are provided below.

- **Problem.A** is the left-hand-side matrix of all constraints.
- **Problem.b** is the right-hand-side vector of all constraints.
- **Problem.sense** is the vector indicating the sense of each constraint. "0" represents "$\leq$" and "1" stands for "=".
- **Problem.Fun** is the vector of objective function coefficients.
- **Problem.M** is the vector indicating variable type. "0" represents continuous, "1" for binary, and "2" for general integers.
- **Problem.LB** is the vector of variable lower bounds.
- **Problem.UB** is the vector of variable upper bounds.
- **Problem.Qmat** is a sparse matrix containing the data of all second-order cone constraints.

2) showsparse

Function **showsparse** displays the sparsity of a sparse matrix or an optimization model.

model.showsparse displays the sparsity of the constraint matrix of **model**. Linear constraints are represented by blue circles, and second order cone constraints are represented by red circles.

showsparse(A) displays the sparsity of matrix **A**.

2.3.5 Parameters

For each XProg object, there is a field named "Param" that are used to assign certain parameters of optimization models. These parameters are listed below.

model.Param.DecRand sets the option for reducing the size of robust optimization problem. If **DecRand** is set to be "1", XProg generates a much smaller robust counterpart formulation by exploiting the special structure of uncertainty sets. The default value is "0", and the size reduction is not applied. Further details are given in Chapter 3 section 3.1.3.

model.Param.ExpName is the name of the exported "lp" file. The default value is an empty set. The name of the exported file is the same as the name of the model, if this field is not specified.

model.Param.IsExport is the export option. An "lp" file associated with the model is generated if

IsExport is set to be "1", and no file is exported if **IsExport** is "0". The default value is "0".

model.Param.IsPrint is the display option. Solution time and status are displayed if **IsPrint** is "1", and no display if **IsPrint** is "0". The default value is "1".

model.Param.MIPGap sets the relative MIP gap for problems containing integer variables. The default value is 10^{-4}. This filed can also be set by the input argument of function **solve**.

model.Param.IntTol sets the integrality tolerance for integer variables. The default value is 10^{-5}. This parameter is quite useful in dealing with constraints with big-M terms.

model.Param.ExpL sets the constant L for approximating the exponential and the quadratic exponential conic constraints. L must be a positive integer, and the default value is 4.

2.4 Illustrative examples

2.4.1 Deterministic optimization examples

Two examples are provided to illustrate the general procedure of using XProg to solve optimization problems. A simple deterministic linear program is firstly addressed, then a practical wind allocation problem is presented as a demonstration of defining quadratic functions and integer decision variables.

Example 2.4.1. Consider a simple linear programming problem (2.1).

$$\begin{aligned} \max \quad & 3x + 4y \\ \text{s.t.} \quad & 2.5x + y <= 20 \\ & 3x + 3y <= 30 \\ & x + 2y <= 16 \\ & x, y >= 0 \end{aligned} \tag{2.1}$$

This linear program can be easily represented by the following code, which gives the optimal value to be 36, and optimal solution to be $x = 4$ and $y = 6$.

```
model=xprog('Simple LP');          % create a model, named "Simple LP"

x=model.decision;                  % create a decision x for model
y=model.decision;                  % create a decision y for model

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

model.add(2.5*x+y<=20);            % add the 1st constraint to model
```

```
model.add(3*x+3*y<=30);              % add the 2nd constraint to model
model.add(x+2*y<=16);                % add the 3rd constraint to model
model.add(x>=0);                     % bound of x
model.add(y>=0);                     % bound of y

model.solve;                         % solve the problem

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

This example shows that the first step of formulating an optimization problem is to create an XProg object. Then the user may define the decision variables, the objective function, and constraints. This optimization model can be solved by calling the function "**solve**". The objective value and the optimal solutions can be accessed by function "**get**".

Example 2.4.2. The second example considers the optimal allocation of 40 wind turbines across seven sites [7], for minimizing the total wind power variance. Suppose that the expected wind energy of all sites is represented by a vector $\mu \in \mathbb{R}^7$, the standard deviation of wind power is denoted by a vector $\sigma \in \mathbb{R}^7$, and the correlation of wind power is indicated by a correlation matrix $\mathcal{P} \in \mathbb{R}^{7\times 7}$. The values of entries in vector $\boldsymbol{\sigma}$ and in matrix $\boldsymbol{P}$ can be found in reference [7]. Let $\boldsymbol{n} \in \mathbb{Z}^7$ be the vector of numbers of turbines in all sites, and the expected total wind power output should be 50MW, this wind diversification problem can be expressed as the following quadratic programming problem.

$$\begin{aligned} \min \quad & [\mathbf{diag}(\boldsymbol{\sigma})\boldsymbol{n}]^T \boldsymbol{P} [\mathbf{diag}(\boldsymbol{\sigma})\boldsymbol{n}] \\ \text{s.t.} \quad & \boldsymbol{\mu}^T \boldsymbol{n} = 50 \\ & \mathbf{1}^T \boldsymbol{n} = 40 \\ & \boldsymbol{n} \in \mathbb{Z}_+^7 \end{aligned} \tag{2.2}$$

The corresponding code based on XProg is listed below.

```
% input data is omitted. Please refer to example_2_4_2_winddiv.m
% mu(7*1) is the expected wind power from each site
% sig(7*1) is wind power standard deviations of each site
% P(7*7) is the correlation matrix

model=xprog('Wind Div');             % create a model, named "Wind Div"
model.Param.IsExport=1;              % export model in lp format
model.Param.ExpName='WindDiv';       % name of the export file is "WindDiv"

n=model.decision(7,1,2,'#Turbine');% integer decisions as the no. of turbines

```

```
12 Fun=square(P^(1/2)*(diag(sig)*n)); % define a quadratic expression by function "square"
13 model.min(Fun);                    % define objective function
14
15 MeanWind=50;                       % expected total wind output
16 model.add(mu'*n==MeanWind);        % add 1st constraint to model
17 model.add(sum(n)==40);             % add 2nd constraint to model
18 model.add(n>=0);                   % bound of n
19
20 model.solve(1e-5)                  % solve the problem, MIP gap set to be 1e-5
21
22 Obj=model.get;                     % get optimal objective
23 N  =n.get;                         % get optimal solution of n
```

The objective value is 1265.67MW^2, and the optimal wind turbine allocation is $\boldsymbol{n} = (12, 20, 1, 0, 0, 2, 5)$. As an integer programming problem, we can change the duality gap tolerance to $1e-5$ by using function **solve** in line 20. This parameter, together with the integrality tolerance variables, can also be changed by assigning values to the fields of **model.Param**. Besides, by setting the parameter "**IsExport**" to be "1", an "lp" file is exported to the current directory, with a user-specified name "WindDiv". If the name of the exported "lp" file is not specified, then the file name is the same as the model name.

By calling function **model.showsparse**, the sparsity of the constraint matrix can be displayed as Figure 2.3. Note XProg generalizes all problems into second-order cone programs, so a few intermediate variables are introduced into the formulation to transform quadratic constraints into equivalent second-order cone constraints.

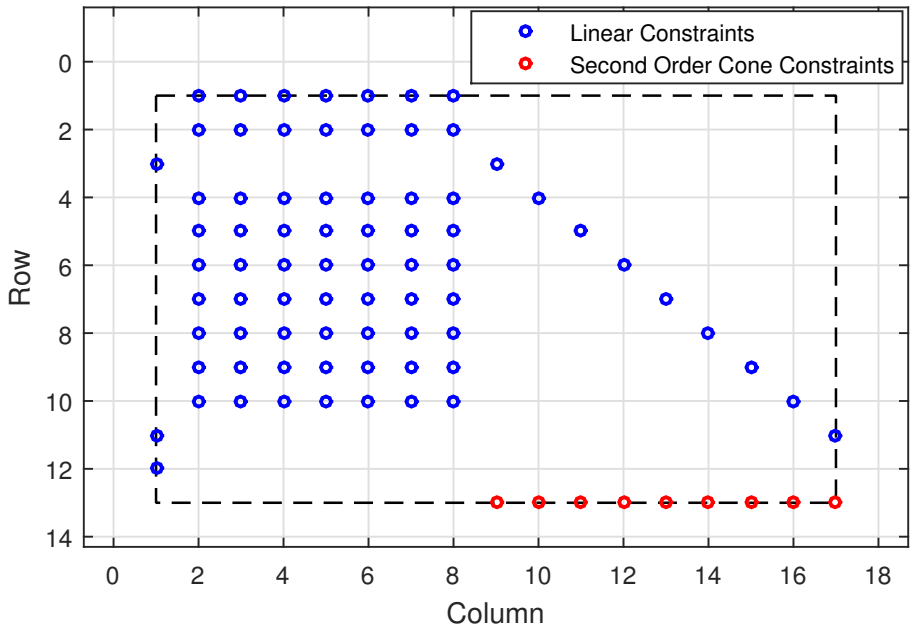

Figure 2.3: Sparsity of the wind diversification model

2.4.2 Stochastic programming examples

XProg may also be used to solve stochastic programming problems. Example 2.4.3 shows how to create decision variables and formulate constraints for different scenarios by exploiting the cell array of MATLAB.

Example 2.4.3. Consider a furniture manufacture problem discussed in reference [8]. A furniture company manufactures desks, tables, and chairs, which sell for \$60, \$40, and \$10, respectively. The manufacture of each product requires lumber and two types skilled labor: finishing and carpentry. The cost of each resource and the amount of needed to make each type of furniture is given in Table 2.2.

Table 2.2. Product Demands in Each Scenario

	Scenario s		
	Low	Most Likely	High
Desk (D_{1s})	50	150	250
Table (D_{2s})	20	110	250
Chair (D_{3s})	200	225	500
Probability P_s	0.3	0.4	0.3

In this example, the demand of each product is subject to uncertainty. The uncertain demand is expressed by three scenarios: "low", "most likely", and "high". The outcomes of demand scenarios and the associated probabilities are provided in Table 2.3.

Table 2.3. Cost of Resources for Furniture

		Resource Requirement		
Resources	Cost (\$)	Desk	Table	Chair
Carpentry	5.2	2	1.5	0.5
Finishing	4	4	2	1.5
Lumber	2	8	6	1

The notation D_{is} is the demand of furniture i in scenario s, and P_s is the probability of scenario s. Let $\boldsymbol{x}$ be the first-stage decisions indicating the amount of resources purchased, and the recourse decisions $\boldsymbol{y}_s$ represent the number of desks, tables, and chairs produced in the sth scenario. The stochastic programming problem that attempts to maximize the expected profit is expressed as follows.

$$
\begin{aligned}
\max \quad & -2x_1 - 4x_2 - 5.2x_3 + \sum_{s=1}^{3} P_s \left(60y_{1s} + 40y_{2s} + 10y_{3s}\right) \\
\text{s.t.} \quad & 8y_{1s} + 6y_{2s} + y_{3s} - x_1 \leq 0, && \forall s = 1,2,3 \\
& 4y_{1s} + 2y_{2s} + 1.5y_{3s} - x_2 \leq 0, && \forall s = 1,2,3 \\
& 2y_{1s} + 1.5y_{2s} + 0.5y_{3s} - x_3 \leq 0, && \forall s = 1,2,3 \\
& x \geq 0, 0 \leq y_{is} \leq D_{is}, \quad \forall i = 1,2,3, && \forall s = 1,2,3
\end{aligned} \tag{2.3}
$$

The deterministic equivalent formulation of the stochastic program is expressed by the following code.

```
model=xprog('SP Example');          % create a model, named "SP Example"

x=model.decision(3);                % 1st-stage decisions
y=cell(1,3);                        % create a cell for three scenarios
for s=1:3
    y{s}=model.decision(3);         % 2nd-stage decisions for scenario s
end

P= [0.3 0.4 0.3];                   % probability
c=-[2 4 5.2];                       % 1st-stage objective vector
q= [60 40 10];                      % recourse objective vector
Exp=0;
for s=1:3
    Exp=Exp+P(s)*q*y{s};            % expected recourse objective
end
model.max(c*x+Exp);                 % include objective function

W=[8 6   1;                         % define constraint matrix
   4 2   1.5;
   2 1.5 0.5];
D=[50  150 250;                     % define random demand
   20  110 250;
   200 225 500];
for s=1:3
    model.add(W*y{s}-x<=0);         % add 1st~3rd constraints
    model.add(y{s}>=0);             % bound of y(s)
    model.add(y{s}<=D(:,s));        % demand constriants
end
model.add(x>=0)                     % bound of x

model.solve;                        % solve the program

Obj=model.get;                      % get optimal objective
X  =x.get;                          % get optimal solution of x
```

The expectation of the maximum profit is \$1730, and the optimal decision suggests that the manufacturer should purchase 1300 units of lumber, 540 units of finishing labor, and 325 units of carpentry labor. It can be seen that the recourse decision for each scenario s is straightforwardly defined by the sth entry of the cell array. Besides decision variables, cell array structure can also be applied to random variables, decision rules, or even XProg model objects themselves. By using cells, users are able to group variables into different sets, so that they can easily express more complicated models.

3 ROBUST OPTIMIZATION

3.1 XProg for robust optimization

XProg is a specialized toolbox for robust optimization. Unlike stochastic programs that models the random parameters by scenario representations, robust optimization seeks an optimal solution that is feasible under all uncertainty realizations over a given uncertainty set. This chapter discusses the procedure of formulating and solving robust optimization problem by XProg.

3.1.1 Random variables and uncertainty sets

The following functions are utilized to define random variables and uncertainty sets for robust optimization problems.

1) random

Function **random** defines new random variables for XProg models.

z = **model.random** defines a random variable **z** for model object **model**.

z = **model.random(N)** defines random variables as an $N \times 1$ vector **z** for **model**.

x = **model.random(N, M)** defines random variables as an $N \times M$ matrix **z** for **model**.

x = **model.random(N, M, name)** defines random variables **z** using specific **N** and **M**. The name of these random variables is assigned to be **name**.

2) uncertain

Function **uncertain** includes constraints of random variables into the uncertainty set (or ambiguity set in the case of distributionally robust optimization).

model.uncertain(constraint) includes **constraint** into uncertainty (ambiguity) set **model**.

model.uncertain(constraint, name) includes **constraint** into uncertainty (ambiguity) set **model**. The name of these constraints is assigned to be **name**.

It should be noted that in XProg, random variables are addressed almost the same way as decision variables. It means that the indexing and arithmetic operations applied to decision variables are also

applicable to random variables. The only difference is that random variables should always be continuous, instead of binaries or general integers.

3.1.2 Recourse decisions

In optimization models, some decision may be are made or adjusted after the realization of uncertain parameters. These wait-and-see decisions, or recourse decisions, are typically intractable to solve due to the "curse of dimensionality". This is why various decision rule approximation models [1, 2, 4, 5] are proposed to overcome this difficulty. In XProg toolbox, decision rules can be designed in a straightforward manner, by using functions **recourse** and **depend**.

1) recourse

Function **recourse** defines decision rule that approximates the actual recourse decisions.

y = **model.recouse** defines a decision rule **y** for model object **model**.

y = **model.recourse(N)** defines decision rule as a $N \times 1$ vector **z** for **model**.

y = **model.recourse(N, M)** defines decision rule as a $N \times M$ matrix **z** for **model**.

y = **model.recourse(N, M, name)** defines decision rule **z** using specific **N** and **M**. The name of such a decision rule is assigned to be **name**.

2) depend

Function **recourse** defines the dependency of decision rule upon specific random variables.

y.depend(z) defines dependency of decision rule **y** on each element of random matrix **z**.

y.depend(z,n) defines dependency of decision rule **y** on elements of **z**, indexed by **n**.

y.depend(z,n,m) defines dependency of decision rule **y** on elements of **z**, indexed by row **n** and column **m**.

y(i).depend(z,n,m) defines dependency of decision rule **y**, indexed by **i**, on elements **z**, indexed by row **n** and column **m**.

y(i,j).depend(z,n,m) defines dependency of decision rule **y**, indexed by row **i** and column **j**, depends on elements **z**, indexed by row **n** and column **m**.

3) get

Function **get** returns the optimal decision rule after the problem is solved.

y.get returns the constant term of decision rule **y**.

y.get(z) returns the coefficients of decision rule **y** associated with random variables **z**.

y(i).get(z) returns the coefficient of the decision rule **y**, indexed by **i**, that are associated with random

variables `z`.

`y(i,j).get(z)` returns the coefficient of the decision rule `y`, indexed by row `i` and column `j`, that are associated with random variables `z`.

4) test

Function **`test`** calculates the recourse decisions under a specific outcome of random variables based on the decision rule policies. It can be used to access how well decision rule approximates the actual optimal recourse decisions.

`Y=y.test(randVal)` evaluates the adjustable decision `y` under a specific outcome of uncertainty as a matrix `randVal`, based on the decision rule policies. Note that the number of elements in matrix `randVal` should be the same as the total number of random variables defined for this model (The shape may be different). This function can only be called after the model is solved and the decision rule is obtained.

Note that XProg does not allow redefining of the same decision rule dependency. In the example below, an error message is given because `y(4)` depends on `z(1)` has already been defined in the first line.

```
1 y(2:4).depend(z,1:5);          % defined dependency
2 y(4:6).depend(z,1);            % y(4) depends on z(1) has been defined...
3                                % error message: Redefinition of dependency.
```

Finally, XProg forbids defining new decisions, new random variables, new decision rules, or redefining the uncertainty set, after including any robust constraints.

3.1.3 Decomposition of uncertainty sets

Suppose that an uncertainty set $\mathcal{Z}$ for random variables $\tilde{\boldsymbol{z}} \in \mathbb{R}^N$ has a special structure and can be decomposed into T distinct sets, expressed as equation (3.1).

$$\mathcal{Z} = \mathcal{Z}_1 \times \mathcal{Z}_2 \times \mathcal{Z}_3 \times ... \times \mathcal{Z}_T \tag{3.1}$$

where each $\mathcal{Z}_t$ is an uncertainty set of random variables $\tilde{\boldsymbol{z}}_t \in \mathbb{R}^{N_t}$, for all $t = 1, 2, ..., T$. Random variables satisfy $\tilde{\boldsymbol{z}} = (\boldsymbol{z}_1, \boldsymbol{z}_2, \boldsymbol{z}_3, ...\boldsymbol{z}_T)$, and $N = N_1 + N_2 + N_3 + ... + N_T$.

Now consider the following uncertain linear constraint.

$$\boldsymbol{z}_t^T \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{x} + \boldsymbol{z}_t^T\boldsymbol{c} + d \leq f, \qquad \forall \boldsymbol{z} = (\boldsymbol{z}_1, \boldsymbol{z}_2, \boldsymbol{z}_3, ...\boldsymbol{z}_t ...\boldsymbol{z}_T) \in \mathcal{Z} \tag{3.2}$$

Apparently, the uncertain constraint (3.2) is equivalent to the constraint (3.3).

$$\boldsymbol{z}_t^T \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{x} + \boldsymbol{z}_t^T\boldsymbol{c} + d \leq f, \qquad \forall \boldsymbol{z}_t \in \mathcal{Z}_t \tag{3.3}$$

The robust counterpart of constraint (3.3) is smaller than (3.2), because there are fewer random variables and constraints involved in set $\mathcal{Z}_t$ compared with overall uncertainty set $\mathcal{Z}$.

The decomposition of uncertainty sets is suitable for addressing multi-period problems with linear decision rule policy applied to approximate the optimal recourse decisions, where uncertainties and constraints are usually defined separately for each time period. XProg is capable of detecting this special structure of uncertainty sets, and it is also able to smartly decompose uncertainty sets for each uncertain constraint, leading to smaller problem dimensions and lower computational cost. The decomposition of uncertainty sets is enabled as the model parameter **`model.Param.DecRand`** is assigned to be "1". The default value of **`model.Param.DecRand`** is "0". It should be noted that if the uncertainty set does not have such a decomposable uncertainty set, it may take much longer time for XProg to check the structure of uncertainty set constraints. It is hence recommended that this decomposition be applied with caution. Especially when the uncertainty set is not decomposable, the decomposition option should be turned off.

3.2 Illustrative example

Following examples are provided to illustrate how to define random variables, uncertainty sets, and decision rule approximation for robust optimization models.

3.2.1 Simple portfolio example

Example 3.2.1. This a portfolio selection problem adopted from reference [9]. Suppose that there are $n = 150$ stocks to be selected, and the uncertain return of each stock i, denoted by $\tilde{p}_i$, is within a symmetric interval $[p_i - \sigma_i, p_i + \sigma_i]$, where p_i is the expected return, and σ_i represents the deviation of the random return of stock i. The uncertainty set $\mathcal{Z}$ is formulated as equation (3.4).

$$\mathcal{Z} = \{\boldsymbol{z} : \ \|\boldsymbol{z}\|_\infty \leq 1, \|\boldsymbol{z}\|_1 \leq \Gamma\} \tag{3.4}$$

The robust optimization model for this portfolio problem can be expressed as the problem below.

$$\begin{aligned} \min \quad & \max_{\boldsymbol{z}\in\mathcal{Z}} \sum_{i=1}^{n} (p_i + \sigma_i z_i)\, x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} x_i = 1 \\ & x \geq 0 \end{aligned} \tag{3.5}$$

In this example, it is assumed that the budget of uncertainty $\Gamma = 5$, and parameters p_i and σ_i are

expressed as equations (3.6), and

$$
\begin{aligned}
p_i &= 1.15 + i\frac{0.05}{150}, \qquad \forall i = 1, 2, 3, ..., 150 \\
\sigma_i &= \frac{0.05}{450}\sqrt{2n(n+1)i}, \quad \forall i = 1, 2, 3, ..., 150
\end{aligned} \tag{3.6}
$$

These two equations imply that the stock with higher expected returns also have higher levels of uncertainty. This robust optimization model can be simply represented by the code listed below.

```
% input data is omitted. Please refer to example 3_2_1_portfolio.m
% p(1*n) is the expected return of each stock
% sigma(1*n) is the uncertain range of random stock returns
% Gamma=5 is the budget of uncertainty

model=xprog('Portfolio');           % create a model, named "Portfolio"

x=model.decision(n);                % decisions as fraction of investment

z=model.random(n);                  % random varaibles z
u=model.random(n);                  % absolute value of z
model.uncertain(abs(z)<=1);         % norm_inf(z)<=1
model.uncertain(abs(z)<=u);         % expressing the absolute value of z
model.uncertain(sum(u)<=Gamma);     % norm_1(z)=sum(abs(z))<=sum(u)<=Gamma

model.max((p+sigma.*z)'*x);         % define the objective function
model.add(sum(x)==1);               % constraint of x
model.add(x>=0);                    % bound of x

model.solve;                        % solve the problem
```

The objective value is 1.1709, and the optimal solution in terms of the stock is shown by Figure 3.1.

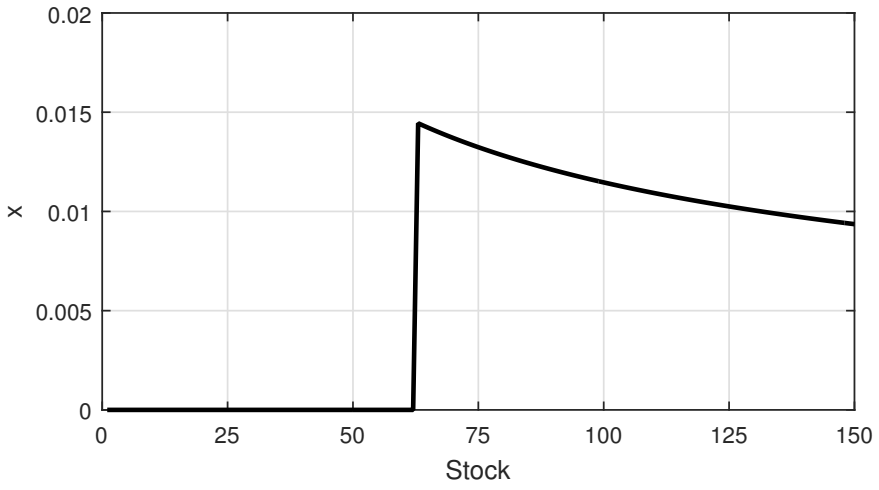

Figure 3.1: The solution to the simple portfolio problem

This case shows the steps of solving robust optimization problems by using XProg. The constraints of random variables can be included into the uncertainty set by function "uncertain". Note that users must define a specific uncertainty set before solving the model, otherwise the program will be terminated with an error message.

3.2.2 Inventory example

Example 3.2.2. This example considers a multistage inventory management problem presented in [4]. The inventory system is comprised of a warehouse and I factories in a horizon with T time steps. The uncertainty appears in the demand for the product at each time step t, denoted by d_t. An uncertainty set $\mathcal{D}$ expressed as the following equation is used to define the upper and lower bounds of random demand.

$$\mathcal{D} = \{\boldsymbol{d} :\ d_t \in [d_t^* - \delta d_t^*, d_t^* + \delta d_t^*]\,,\ \forall t = 1, ..., T\} \tag{3.7}$$

where d_t^* is the expected demand at time t, and δ is a constant implying the deviation of random demand. In this example, the constant δ is set to be 0.2, and d_t^* follows the seasonal pattern below.

$$d_t^* = 1000\left(1 + \frac{1}{2}\sin\left(\frac{\pi(t-1)}{12}\right)\right),\quad \forall t = 1, ..., T \tag{3.8}$$

Suppose that the number of factories is $I = 3$, and the number of time steps is $T = 24$. The production capacity $P_{it} = 567$, and the total production cannot exceed $Q_i = 13600$. The inventory at the warehouse should be between $V_{min} = 500$ and $V_{max} = 2000$, and the initial inventory $v_1 = 1000$. The production cost c_{it} for each factory i at time t is given as the equation (3.9).

$$c_{it} = \alpha_i\left(1 + \frac{1}{2}\sin\left(\frac{\pi(t-1)}{12}\right)\right),\ \forall i = 1, ..., I,\ \forall t = 1, ..., T \tag{3.9}$$

where coefficients $\alpha_1 = 1$, $\alpha_2 = 1.5$, and $\alpha_3 = 2$. The multistage robust inventory management problem can be formulated as follows.

$$\begin{aligned}
\min\quad & \max_{\boldsymbol{d}\in\mathcal{D}} \sum_{t=1}^{T}\sum_{i=1}^{I} c_{it}p_{it}(\boldsymbol{d}) \\
\text{s.t.}\quad & 0 \le p_{it}(\boldsymbol{d}) \le P, && \forall \boldsymbol{d} \in \mathcal{D}, \forall i = 1, ..., I, \forall t = 1, ..., T \\
& \sum_{t=1}^{T} p_{it}(\boldsymbol{d}) \le Q, && \forall \boldsymbol{d} \in \mathcal{D}, \forall i = 1, ..., I \\
& v_{t+1}(\boldsymbol{d}) = v_t(\boldsymbol{d}) + \sum_{i}^{I} p_{it}(\boldsymbol{d}) - d_t, && \forall \boldsymbol{d} \in \mathcal{D}, \forall t = 1, ..., T \\
& V_{min} \le v_t(\boldsymbol{d}) \le V_{max}, && \forall \boldsymbol{d} \in \mathcal{D}, \forall t = 2, ..., T+1
\end{aligned} \tag{3.10}$$

Note that the production decisions $p_{it}(\boldsymbol{d})$ are recourse decisions, which is approximated by the linear affine function (3.11).

$$p_{it}(\boldsymbol{d}) = \pi_{it}^0 + \sum_{\tau \in I_t} \pi_{it}^\tau d_t, \qquad \forall i = 1, ..., I, \ \forall t = 1, ..., T \tag{3.11}$$

We adopt the *standard information basis* assumption in [4] such that $I_t = \{1, ..., t-1\}$. The code for this problem is provided below.

```
1  % input data is omitted. Please refer to example 3_2_2_inventory.m
2  % T=24 is the number of time step
3  % I=3 is the number of factories
4  % d0(1*T) is the vector of expected demand
5  % c(I*T) is the matrix of production cost coefficients
6  % P is the production capacity for each time step
7  % Q is total production capacity for the entire time horizon
8  % Vmin and Vmax are the minimum and maximum bounds of inventory
9  % V0 is the initial inventory
10 % delta is the uncertainty level
11
12 model=xprog('inventory');          % create a model, named 'inventory'
13 %model.Param.DecRand=1;            % enable decomposition of uncertainty set
14
15 d=model.random(1,T);               % define random demand
16 model.uncertain(d<=(1+delta)*d0);  % upper bound of random demand
17 model.uncertain(d>=(1-delta)*d0);  % lower bound of random demand
18
19 p=model.recourse(I,T);             % define recourse decision as decision rule
20
21 for t=2:T                          % define dependency in a for loop
22     I_t=1:t-1;                     % standard info. basis assumption
23     p(:,t).depend(d,I_t);          % dependency based on standard info. basis
24 end
25
26 model.min(sum(sum(c.*p)));         % define objective
27
28 model.add(p>=0);                   % lower bound of p
29 model.add(p<=P);                   % upper bound of p
30 model.add(sum(p,2)<=Q)             % total production capacity Q
31 v=V0;                             % initial inventory
32 for t=1:T
33     v=v+(sum(p(:,t))-d(t));        % update inventory v_{t+1}
34     model.add(v<=Vmax);            % upper limit of inventory
35     model.add(v>=Vmin);            % lower limit of inventory
36 end
37
38 model.solve;                       % solve the problem
```

The optimal objective value is 4206.9. It can be seen from line 21 to 24 that by assigning index of

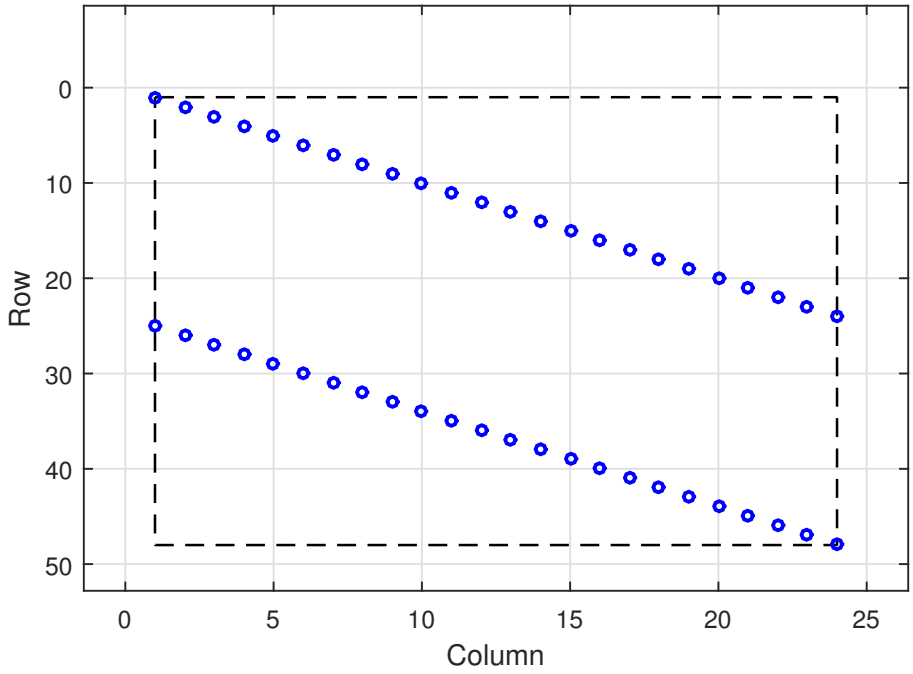

Figure 3.2: The uncertainty set of the inventory example can be decomposed

recourse decision `p` and random variable `d`, we can conveniently define flexible decision rule dependency to address multistage decision-making problems.

It is observed from Figure 3.2 that the uncertainty set has the special decomposable structure discussed in section 3.1.3 and can be reduced into (3.3). The decomposition is enabled as line 13 is uncommented. By applying uncertainty set decomposition, the same objective value and optimal solution are achieved, yet the problem dimensions are reduced from 4900 constraints and 10309 variables to 3240 constraints and 6989 variables.

We can also conduct simulations as in reference [4] to access the performance of decision rules by using function **`test`**. Function **`test`** computes the recourse decisions corresponding to a specific outcome of system uncertainty according to the decision rule function. An example of the simulation is given below.

```
1 S=100;                              % number of samples
2
3 Cost=zeros(1,S);
4 for s=1:S
5     R=rand(1,T);                    % uniformly distributed random numbers
6     randD=d0.*(1-delta+2*delta*R); % sample of random demand
7     P_dr=p.test(randD);             % recourse based on decision rule policy
8     Cost(s)=sum(sum(c.*P_dr));      % total cost for the sth sample
9 end
```

4 DISTRIBUTIONALLY ROBUST OPTIMIZATION

4.1 XProg for distributionally robust optimization

XProg is designed to construct and efficiently solve distributionally robust optimization models proposed by references [1, 2, 3]. Technical details and examples are provided in the remaining part of this chapter.

4.1.1 Ambiguity sets

The distributionally robust optimization addresses uncertainties by an ambiguity set that characterizes a family of distributions. The standard ambiguity set of random variables $\tilde{\boldsymbol{z}} \in \mathbb{R}^I$ is expressed as (4.1).

$$\mathbb{F} = \left\{ \mathbb{P} \in \mathcal{P}_0\left(\mathbb{R}^I\right) : \begin{array}{ll} \tilde{\boldsymbol{z}} \in \mathbb{R}^I & \\ \mathbb{E}_{\mathbb{P}}\left(\boldsymbol{G}\tilde{\boldsymbol{z}}\right) = \boldsymbol{\mu} & \\ \mathbb{E}_{\mathbb{P}}\left(g_j(\tilde{\boldsymbol{z}})\right) \leq \sigma_j, & \forall j \in \mathcal{J} \\ \mathbb{P}\left(\tilde{\boldsymbol{z}} \in \mathcal{W}_0\right) = 1 & \\ \mathbb{P}\left(\tilde{\boldsymbol{z}} \in \mathcal{W}_s\right) \in \left[\underline{P}_s\ \overline{P}_s\right], & \forall s \in \mathcal{S} \end{array} \right\} \tag{4.1}$$

with matrix $\boldsymbol{G} \in \mathbb{R}^{L \times I}$ and vector $\boldsymbol{\mu} \in \mathbb{R}^L$. Each function $g_j(\boldsymbol{z})$ for all $j \in \mathcal{J}$ is a convex conic quadratic representable function. The support set $\mathcal{W}_0$ and each confidence set $\mathcal{W}_s$ for all $s \in \mathcal{S}$ are also conic quadratic. Note that all of these subsets must satisfy the nesting condition $\mathcal{W}_0 \supset \mathcal{W}_1 \supset \mathcal{W}_2 \supset ...\mathcal{W}_{|\mathcal{S}|}$. The last line of the ambiguity set $\mathbb{F}$ suggests that the probability of having random variables $\tilde{\boldsymbol{z}}$ within the confidence set $\mathcal{W}_s$ is higher than $\underline{P}_s$ and lower than $\overline{P}_s$, where $0 \leq \underline{P}_s \leq \overline{P}_s \leq 1$.

According to references [1] [2], this ambiguity set is then reformulated into an extended form involving auxiliary variables. The extended ambiguity set can be conveniently modeled by XProg, and the enhanced decision rule [1, 2] can be easily defined as linear affine functions of random variables and auxiliary variables that are introduced into the extended ambiguity set. Details of implementing XProg to solve distributionally robust optimization problems are provided in the next subsection.

4.1.2 Formulating extended ambiguity set

This distributionally robust optimization framework requires introducing some auxiliary random variables into the ambiguity sets, in order to express the expectation of function $g_j(\tilde{\boldsymbol{z}})$, and to improve the linear

rule approximation. These auxiliary variables are defined exactly the same as random variables. Since the extended support sets $\hat{\mathcal{W}}_0$ and all confidence sets $\hat{\mathcal{W}}_s$ are conic quadratic representable, these sets can be defined in the same manner as ordinary uncertainty sets for robust optimization. XProg also uses function **expect** to express the expectation of linear functions of random variables and to formulate constraints with distribution information. These constraints can also be incorporated into the ambiguity set by function **uncertain**.

1) expect

Function **expect** expresses the expectation of linear functions of random variables. The expectation of linear functions are supposed to be incorporated into the ambiguity set.

expect(fun) returns the expected value of **fun**. The argument **fun** must be a linear affine function of random variables.

2) subset

Function **subset** defines a new confidence set as a proper subset of the support set $\mathcal{W}_0$ or preceding confidence sets.

Subset=model.subset(P) defines a confidence set for **model** as a proper subset of the support set $\mathcal{W}_0$ with probability interval **P**. Note that if **P** is a vector of two elements, the first element is the probability lower bound $\underline{P}_s$, and the second is the probability upper bound $\overline{P}_s$. If the argument **P** is a scaler, then both the upper and lower bounds of the probability are equal to **P**.

Subset=model.subset(P, Set) defines a confidence set for **model** as a proper subset of the confidence set **Set** with probability intervals defined by **P**.

Examples of using **subset** to define confidence set of **model** are given below.

```
model=xprog;                          % create a model

W1=model.subset(0.8);                 % W1 is a proper subset of W0...
                                      % probability equals to 0.8
W2=model.subset([0.3 0.6],W1);        % W2 is a proper subset of W1...
                                      % probability is between 0.3 and 0.6
```

3) uncertain

Function **uncertain** adds constraints of random variables into the ambiguity set.

model.uncertain(constraint) includes **constraint** into the ambiguity set. The input argument **constraint** can be conic quadratic constraints of random variables (these constraints are added into the support set $\mathcal{W}_0$), or constraints involving the expectation of linear expressions of random variables.

model.uncertain(constraint, Set) includes **constraint** into the confidence set **Set**. If **Set** is empty, constraints are included into the support set $\mathcal{W}_0$.

model.uncertain(constraint, Set, name) includes constraints **constraint** into the support set or the confidence set **Set**. The name of these constraints is assigned to be **name**.

4.1.3 Expectation over the ambiguity set

The function **expect** is overloaded to express the worst-case expectation of an uncertain function over all distributions characterized by the aforementioned ambiguity set.

1) expect

Function **expect** expresses the worst-case expectation of an uncertain function.

expect(unfun) returns the worst-case expectation of **unfun**, which is an uncertain linear affine function, over all distributions characterized by the ambiguity set. For example, consider the constraint (4.2)

$$\mathbb{E}_{\mathbb{P}}\left\{\tilde{z}^T A x + b^T x + c^T \tilde{z}\right\} \leq d^T x, \quad \forall \mathbb{P} \in \mathbb{F} \tag{4.2}$$

The corresponding code is given below.

```
% z is a vector of random variables
% x is a vector of decision variables
% A is a constant matrix
% b, c, and d are constant vectors

% the constraint holds under all distributions in the ambiguity set
model.add(expect(z'*A*x+b'*x+c'*z)<=d'*x);
```

Note that if there is no distribution information incorporated into the ambiguity set, the code above gives an error message. In addition, similar to previous cases, XProg forbids defining new decision variables, new random variables, new decision rules, or redefining the ambiguity set after adding any robust or worst-case expectation constraints.

4.2 Illustrative examples

4.2.1 Simple distributionally robust optimization problem

Example 4.2.1. A simple distributionally robust optimization example is firstly presented to demonstrate how to model ambiguity set and define enhanced decision rule by XProg. Suppose that there is one random

variable $\tilde{z}$, and the adjustable decision made after the observation of uncertainty realization z is denoted by $y(z)$. This distributionally robust optimization model is expressed as the following problem.

$$\begin{aligned} \min \quad & \sup_{\mathbb{P}\in\mathbb{F}} \mathbb{E}_{\mathbb{P}}\left\{y(\tilde{z})\right\} \\ \text{s.t } & y(z) \geq z, \qquad \forall z \in \mathbb{R} \\ & y(z) \geq -z, \quad \forall z \in \mathbb{R} \end{aligned} \tag{4.3}$$

The ambiguity set $\mathbb{F}$ is expressed in equation (4.4). It is then transformed into the extended form (4.5).

$$\mathbb{F} = \left\{ \mathbb{P} \in \mathcal{P}_0\left(\mathbb{R}\right) : \begin{array}{l} \tilde{z} \in \mathbb{R} \\ \mathbb{E}_{\mathbb{P}}(\tilde{z}) = 0 \\ \mathbb{E}_{\mathbb{P}}(\tilde{z}^2) \leq 1 \\ \mathbb{P}\left\{-2 \leq \tilde{z} \leq 2\right\} = 1 \end{array} \right\} \tag{4.4}$$

$$\mathbb{G} = \left\{ \mathbb{Q} \in \mathcal{P}_0\left(\mathbb{R}^2\right) : \begin{array}{l} \tilde{z} \in \mathbb{R},\ \tilde{u} \in \mathbb{R} \\ \mathbb{E}_{\mathbb{Q}}(\tilde{z}) = 0 \\ \mathbb{E}_{\mathbb{Q}}(\tilde{u}) \leq 1 \\ \mathbb{Q}\left\{ \begin{array}{l} -2 \leq \tilde{z} \leq 2 \\ \tilde{z}^2 \leq u \leq 4 \end{array} \right\} = 1 \end{array} \right\} \tag{4.5}$$

The actual recourse decision $y(z)$ is approximated by enhanced linear decision rule in equation (4.6), as a linear affine function of random variable z and auxiliary variable u.

$$y(z,u) = y^0 + y^z z + y^u u \tag{4.6}$$

The code of this model is presented below.

```
model=xprog('simple_dro');                    % create a model named 'simple_dro'

y=model.recourse(1,1);                        % define recourse decision y

z=model.random(1);                            % define random variable z
u=model.random(1);                            % define auxiliary variable u

y.depend(z);                                  % define dependency of y on z
y.depend(u);                                  % define dependency of y on u

model.uncertain(expect(z)==0);                % 2nd line of set G
model.uncertain(expect(u)<=1);                % 3rd line of set G
model.uncertain(z<= 2);                       % 4th line of set G
model.uncertain(z>=-2);                       % 4th line of set G
model.uncertain(z.^2<=u);                     % 4th line of set G
```

```
model.uncertain(u<=2^2);                    % 4th line of set G

model.min(expect(y));                       % define objective function

model.add(y>=z);                            % 1st constraint
model.add(y>=-z);                           % 2nd constraint

model.solve;                                % solve the problem
```

The objective value of this problem is 1. Apparently the optimal recourse decision is $y(z) = |z|$. The optimal solution suggests that the decision rule is $y(z,u) = \frac{1+u}{2} = \frac{1+z^2}{2}$. The optimal recourse decision and decision rule under various uncertainty realizations, as well as the worst-case distribution, are depicted by Figure 4.1.

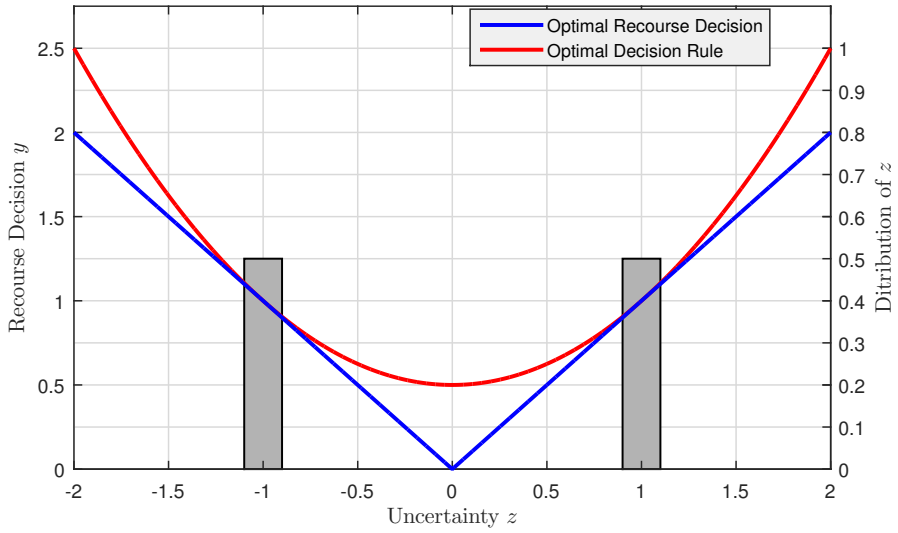

Figure 4.1: Optimal decision and decision rule, together with the worst-case distribution

Now consider incorporating two confidence sets into the ambiguity set. These two confidence sets are proper subsets of the support set. The new ambiguity set therefore is in the form of (4.7), the corresponding extended ambiguity set is expressed as (4.8).

$$\mathbb{F} = \left\{ \mathbb{P} \in \mathcal{P}_0\left(\mathbb{R}\right) : \begin{array}{l} \tilde{z} \in \mathbb{R} \\ \mathbb{E}_{\mathbb{P}}(\tilde{z}) = 0 \\ \mathbb{E}_{\mathbb{P}}(\tilde{z}^2) \leq 1 \\ \mathbb{P}\left\{-2 \leq \tilde{z} \leq 2\right\} = 1 \\ \mathbb{P}\left\{-1 \leq \tilde{z} \leq 1\right\} = 0.9 \\ \mathbb{P}\left\{-0.5 \leq \tilde{z} \leq 0.5\right\} \in [0.6\ 0.7] \end{array} \right\} \tag{4.7}$$

$$\mathbb{G} = \left\{ \mathbb{Q} \in \mathcal{P}_0\left(\mathbb{R}^2\right) : \begin{array}{l} \tilde{z} \in \mathbb{R},\ \tilde{u} \in \mathbb{R} \\ \mathbb{E}_{\mathbb{Q}}(\tilde{z}) = 0 \\ \mathbb{E}_{\mathbb{Q}}(\tilde{z}^2) \le 1 \\ \mathbb{Q}\left\{ \begin{array}{l} -2 \le \tilde{z} \le 2 \\ \tilde{z}^2 \le \tilde{u} \le 4 \end{array} \right\} = 1 \\ \mathbb{Q}\left\{ \begin{array}{l} -1 \le \tilde{z} \le 1 \\ \tilde{z}^2 \le \tilde{u} \le 1 \end{array} \right\} = 0.9 \\ \mathbb{Q}\left\{ \begin{array}{l} -0.5 \le \tilde{z} \le 0.5 \\ \tilde{z}^2 \le \tilde{u} \le 0.25 \end{array} \right\} \in [0.6\ 0.7] \end{array} \right\} \tag{4.8}$$

The code for this distributionally robust optimization model is provided as follows.

```
model=xprog('simple_dro');                  % create a model named 'simple_dro'

y=model.recourse(1,1);                      % define recourse decision y

z=model.random(1);                          % define random variable z
u=model.random(1);                          % define auxiliary variable u

y.depend(z);                                % define dependency of y on z
y.depend(u);                                % define dependency of y on u

model.uncertain(expect(z)==0);              % 2nd line of set G
model.uncertain(expect(u)<=1);              % 3rd line of set G

model.uncertain(z<= 2);                     % 4th line of set G
model.uncertain(z>=-2);                     % 4th line of set G
model.uncertain(z.^2<=u);                   % 4th line of set G
model.uncertain(u<=2^2);                    % 4th line of set G

Z1=model.subset(0.9);                       % Z1 is a subset of support set...
                                            % P{Z1} is 0.9
model.uncertain(z<= 1,Z1);                  % 5th line of set G
model.uncertain(z>=-1,Z1);                  % 5th line of set G
model.uncertain(z.^2<=u,Z1);                % 5th line of set G
model.uncertain(u<=1^2,Z1);                 % 5th line of set G

Z2=model.subset([0.6 0.7],Z1);              % Z2 is a subset of Z1...
                                            % P{Z2} is between 0.6 and 0.7
model.uncertain(z<= 0.5,Z2);                % 6th line of set G
model.uncertain(z>=-0.5,Z2);                % 6th iine of set G
model.uncertain(z.^2<=u,Z2);                % 6th line of set G
model.uncertain(u<=0.5^2,Z2);               % 6th line of set G

model.min(expect(y));                       % define objective function
```

```
34
35 model.add(y>=z);                                    % 1st constraint
36 model.add(y>=-z);                                   % 2nd constraint
37
38 model.solve;                                       % solve the problem
```

The objective value is 0.9220. The optimal recourse decision and decision rule, as well as the worst-case distribution, are displayed in Figure 4.2. Note that the distribution information in terms of confidence sets changes the shape of the worst-case distribution, and the decision rule policy is also changed correspondingly.

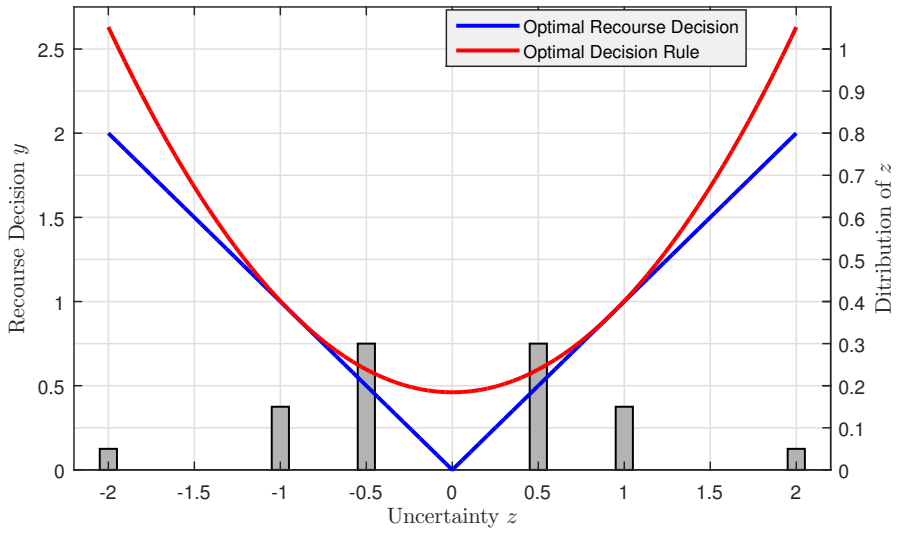

Figure 4.2: Optimal decision and decision rule, together with the worst-case distribution

4.2.2 Newsvendor problem

Example 4.2.2. This example repeats the experiment on the newsvendor problem in reference [2]. Let $\tilde{\boldsymbol{z}} \in \mathbb{R}^N$ denote random variables that indicate the uncertain demands of N products. The probability distributions of these random variables are characterized by the ambiguity set $\mathbb{F}$ in equation (4.9).

$$\mathbb{F} = \left\{ \mathbb{P} \in \mathcal{P}_0\left(\mathbb{R}^N\right) : \begin{array}{l} \mathbb{E}_{\mathbb{P}}\left\{\tilde{\boldsymbol{z}}\right\} = \boldsymbol{\mu} \\ \mathbb{E}_{\mathbb{P}}\left\{\tilde{z}_i^2\right\} \leq \mu_i^2 + \sigma_i^2, \quad \forall i = 1, ..., N \\ \mathbb{E}_{\mathbb{P}}\left\{\sum\limits_{i=1}^{N} p_i \cdot \max\left(\mu_i - \tilde{z}_i, 0\right)\right\} \leq \psi \\ \mathbb{P}\left\{\mathbf{0} \leq \tilde{\boldsymbol{z}} \leq \bar{\boldsymbol{z}}\right\} = 1 \end{array} \right\} \tag{4.9}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ represent the expected values and standard deviations of random variables, respectively. The vector $\boldsymbol{p}$ indicates the selling price of each product.

The extended ambiguity set can be therefore derived as (4.10).

$$\mathbb{G} = \left\{ \mathbb{Q} \in \mathcal{P}_0\left(\mathbb{R}^N \times \mathbb{R}^{N+1} \times \mathbb{R}^N\right) : \begin{array}{l} \mathbb{E}_{\mathbb{Q}}\left\{\tilde{\boldsymbol{z}}\right\} = \boldsymbol{\mu} \\ \mathbb{E}_{\mathbb{Q}}\left\{\tilde{u}_i\right\} \leq \mu_i^2 + \sigma_i^2, \quad \forall i = 1, ..., N \\ \mathbb{E}_{\mathbb{Q}}\left\{\tilde{u}_{N+1}\right\} \leq \psi \\ \mathbb{Q}\left\{(\tilde{\boldsymbol{z}}, \tilde{\boldsymbol{u}}, \tilde{\boldsymbol{v}}) \in \hat{\mathcal{W}}\right\} = 1 \end{array} \right\} \tag{4.10}$$

where $\hat{\mathcal{W}}$ is the extended support set expressed as below.

$$\hat{\mathcal{W}} = \left\{ (\boldsymbol{z}, \boldsymbol{u}, \boldsymbol{v}) \in \mathbb{R}^N \times \mathbb{R}^{N+1} \times \mathbb{R}^N : \begin{array}{l} \boldsymbol{0} \leq \tilde{\boldsymbol{z}} \leq \bar{\boldsymbol{z}} \\ z_i^2 \leq u_i, \quad \forall i = 1, ..., N \\ u_{N+1} \geq \boldsymbol{p}^T \boldsymbol{v} \\ \boldsymbol{v} \geq \boldsymbol{\mu} - \boldsymbol{z} \\ \boldsymbol{v} \geq \boldsymbol{0} \end{array} \right\} \tag{4.11}$$

The formulation of this newsvendor problem is given as (4.12)

$$\begin{aligned} \min \quad & -\boldsymbol{p}^T \boldsymbol{x} + \max_{\mathbb{Q} \in \mathbb{G}} \mathbb{E}_{\mathbb{Q}}\left\{\boldsymbol{p}^T \boldsymbol{y}\left(\tilde{\boldsymbol{z}}, \tilde{\boldsymbol{u}}, \tilde{\boldsymbol{v}}\right)\right\} \\ \text{s.t.} \quad & \boldsymbol{c}^T \boldsymbol{x} \leq \Gamma \\ & \boldsymbol{x} \geq \boldsymbol{0} \\ & \boldsymbol{y}(\boldsymbol{z}, \boldsymbol{u}, \boldsymbol{v}) \geq \boldsymbol{x} - \boldsymbol{z} \\ & \boldsymbol{y}(\boldsymbol{z}, \boldsymbol{u}, \boldsymbol{v}) \geq \boldsymbol{0} \end{aligned} \tag{4.12}$$

where $\boldsymbol{x}$ are the here-and-now decisions, and $\boldsymbol{y}(\boldsymbol{z}, \boldsymbol{u}, \boldsymbol{v})$ is the decision rule approximation of the recourse under uncertainty realization $\boldsymbol{z}$, expressed as (4.13)

$$\boldsymbol{y}(\boldsymbol{z}, \boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{y}^0 + \sum_{i=1}^{N} \boldsymbol{y}_i^z z_i + \sum_{i=1}^{N+1} \boldsymbol{y}_i^u u_i + \sum_{i=1}^{N} \boldsymbol{y}_i^v v_i \tag{4.13}$$

In this example, ψ is set to be 100, and Γ is assigned to be 500. The corresponding code is given as below.

```
% input data omitted. Please refer to example 4_2_2_newsvendor.m
% N=10 is the number of products
% Price(10*1) is the vector of product selling prices
% Cost(10*1) is the vector of product ordering costs
% mu(10*1) is the vector of mean values of random variables z
% barZ(10*1) is the vector of upper bound of random variables z
% sig(10*1) is the vector of standard deviations of random variables z
% Psi(1*1) is expected value of the positive uncertainty of random varaibles z

model=xprog('newsvendor');                    % create a model named 'newsvendor'
```

```
11
12 x=model.decision(N);                          % here-and-now decisions x
13
14 y=model.recourse(N);                          % define decision rule
15
16 z=model.random(N);                            % define random variables z
17 u=model.random(N+1);                          % define auxiliary varialbes u
18 v=model.random(N);                            % define auxiliary variables z
19
20 y.depend(z);                                  % define dependency of y on z
21 y.depend(u);                                  % define dependency of y on u
22 y.depend(v);                                  % define dependency of y on v
23
24 % extended ambiguity set is defined below
25 model.uncertain(expect(z)==mu);               % 1st line of set G
26 mu2 =mu.*mu;                                  % mu^2
27 sig2=sig.*sig;                                % sigma^2
28 model.uncertain(expect(u(1:N)')<=mu2+sig2);   % 2nd line of set G
29 model.uncertain(expect(u(N+1))<=Psi);         % 3rd line of set G
30
31 % extended support set is defined below
32 model.uncertain(z>=0);                         % lower bound of z
33 model.uncertain(z<=barZ);                      % upper bound of z
34 model.uncertain(z.^2<=u(1:N)');                % 2nd line of set \hat{W}
35 model.uncertain(u(N+1)>=Price'*v);             % 3rd line of set \hat{W}
36 model.uncertain(v>=mu-z);                      % 4th line of set \hat{W}
37 model.uncertain(v>=0);                         % 5th line of set \hat{W}
38
39 model.max(Price'*x-expect(Price'*y));          % define objective function
40
41 model.add(Cost'*x<=Gamma);                     % 1st constraint: budget
42 model.add(x>=0);                               % 2nd constraint: nonnegtivity
43 model.add(y>=x-z);                             % 3rd constraint
44 model.add(y>=0);                               % 4th constraint
45
46 model.solve;                                   % solve the problem
```

The objective value and the optimal solution $\boldsymbol{x}$ are provided in the table below.

Table 4.1. Optimal Solution

Optimal	Optimal Solution $\boldsymbol{x}$									
Objective	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
1850.10	30.00	35.00	40.00	45.00	23.40	0.00	0.00	0.00	0.00	0.00

This example demonstrates that XProg provides a convenient way to formulate extended ambiguity sets, and to construct enhanced decision rules to address recourse decisions.

4.2.3 Medical Appointment Scheduling

Example 4.2.3. This example solves the medical appointment scheduling problem discussed in [1]. Suppose that there are N patients arriving in sequence, and the consultation time for the nth patient is represented by random variable $\tilde{z}_n$. The distribution of $\boldsymbol{z} \in \mathbb{R}^N$ can be characterized by an ambiguity set, expressed by $\mathbb{F}$ in equation (4.14).

$$\mathbb{F} = \left\{ \mathbb{P} \in \mathcal{P}_0\left(\mathbb{R}^N\right) : \begin{array}{l} \mathbb{E}_{\mathbb{P}}\left(\tilde{\boldsymbol{z}}\right) = \boldsymbol{\mu} \\ \mathbb{E}_{\mathbb{P}}\left((\tilde{z}_n - \mu_n)^2\right) \leq \sigma_n^2 \\ \mathbb{E}_{\mathbb{P}}\left(\left(\mathbf{1}^T(\tilde{\boldsymbol{z}} - \boldsymbol{\mu})\right)^2\right) \leq \phi^2 \\ \mathbb{P}\left(\tilde{\boldsymbol{z}} \in \mathbb{R}_+^N\right) = 1 \end{array} \right\} \tag{4.14}$$

where $\boldsymbol{\mu} \in \mathbb{R}^N$ indicates the mean values of $\tilde{\boldsymbol{z}}$, and $\boldsymbol{\sigma} \in \mathbb{R}^N$ represents the upper limits of standard deviations of $\tilde{\boldsymbol{z}}$. The constant ϕ^2 in the third line of the ambiguity set $\mathbb{F}$ is the upper bound on the variance of $\mathbf{1}^T(\tilde{\boldsymbol{z}} - \boldsymbol{\mu})$. This inequality captures the partial cross moments information [1] of the distribution. The last line of $\mathbb{F}$ suggests that all random variables must be non-negative. The extended ambiguity set is then formulated as follows.

$$\mathbb{G} = \left\{ \mathbb{Q} \in \mathcal{P}_0\left(\mathbb{R}^{2\times N+1}\right) : \begin{array}{l} \mathbb{E}_{\mathbb{Q}}\left(\tilde{\boldsymbol{z}}\right) = \boldsymbol{\mu} \\ \mathbb{E}_{\mathbb{Q}}\left(\tilde{u}_n\right) \leq \sigma_n^2, \ \forall n = 1, 2, ..., N \\ \mathbb{E}_{\mathbb{Q}}\left(\tilde{u}_{N+1}\right) \leq \phi^2 \\ \mathbb{Q}\left((\tilde{\boldsymbol{z}}, \tilde{\boldsymbol{u}}) \in \hat{\mathcal{W}}\right) = 1 \end{array} \right\} \tag{4.15}$$

where $\tilde{\boldsymbol{u}} \in \mathbb{R}^{N+1}$ is the vector of auxiliary variables, and $\hat{\mathcal{W}}$ is the extended support set, which is expressed as the equation below.

$$\hat{\mathcal{W}} = \left\{ (\boldsymbol{z}, \boldsymbol{u}) \in \mathbb{R}^N \times \mathbb{R}^{N+1} : \begin{array}{l} \boldsymbol{z} \geq \mathbf{0} \\ (z_n - \mu_n)^2 \leq u_n, \ n = 1, 2, ..., N \\ \left(\mathbf{1}^T\left(\boldsymbol{z} - \boldsymbol{u}\right)\right)^2 \leq u_{N+1} \end{array} \right\} \tag{4.16}$$

Let $\boldsymbol{x}$ be the here-and-now decisions, where x_n for all $n = 1, 2, ..., N-1$ is the inter-arrival time between the nth and the $(n+1)$th patient, and x_N represents the time between the arrival of the last patient and the scheduled completion time for the physician before overtime commences. The waiting time for the nth patient is denoted by the wait-and-see decision y_n, and the overtime of the physician is denoted by y_{N+1}. The overtime y_{N+1} causes a cost of γ per unit delay. In this example, the wait-and-see decisions $\boldsymbol{y} \in \mathbb{R}^{N+1}$ are approximated by the decision rule function (4.17).

$$\boldsymbol{y}(\boldsymbol{z}, \boldsymbol{u}) = \boldsymbol{y}^0 + \sum_{n=1}^{N} \boldsymbol{y}_n^z z_n + \sum_{n=1}^{N+1} \boldsymbol{y}_n^u u_n \tag{4.17}$$

with vector $\boldsymbol{y}^0 \in \mathbb{R}^{N+1}$, $\boldsymbol{y}_n^z \in \mathbb{R}^{(N+1)\times N}$, and $\boldsymbol{y}_n^u \in \mathbb{R}^{(N+1)\times N+1}$ indicating the coefficients of the decision rule. The medical appointment scheduling problem is therefore formulated as the distributionally robust model (4.18)-(4.22).

$$\min \ \sup_{\mathbb{Q}\in\mathbb{G}} \mathbb{E}_{\mathbb{Q}} \left(\sum_{n=1}^{N} y_n(\tilde{\boldsymbol{z}}, \tilde{\boldsymbol{u}}) + \gamma y_{N+1}(\tilde{\boldsymbol{z}}, \tilde{\boldsymbol{u}}) \right) \tag{4.18}$$

$$\text{s.t. } y_{i+1}(\boldsymbol{z}, \boldsymbol{u}) - y_i(\boldsymbol{z}, \boldsymbol{u}) + x_n \geq z_n, \ \forall (\boldsymbol{z}, \boldsymbol{u}) \in \hat{\mathcal{W}}, n = 1, 2, ..., N \tag{4.19}$$

$$\boldsymbol{y}(\boldsymbol{z}, \boldsymbol{u}) \geq \boldsymbol{0}, \qquad \forall (\boldsymbol{z}, \boldsymbol{u}) \in \hat{\mathcal{W}}, n = 1, 2, ..., N \tag{4.20}$$

$$\sum_{n}^{N} x_n \leq T \tag{4.21}$$

$$\boldsymbol{x} \geq \boldsymbol{0} \tag{4.22}$$

In this example, the number of patients is $N = 8$, and the overtime cost is $\gamma = 2$. The mean value of each random variable $\tilde{z}_n$ is $\mu_n = 45$, and the standard deviation is $\sigma_n = 0.3\mu$. The constant ϕ^2 is given as the equation below.

$$\phi^2 = \sum_{n=1}^{N} \sigma_n^2 \tag{4.23}$$

The evaluation period T is expressed as equation (4.24).

$$T = \sum_{n=1}^{N} \mu_n + 0.5 \sqrt{\sum_{n=1}^{N} \sigma_n^2} \tag{4.24}$$

Such a medical appointment scheduling model can be solved by the following code.

```
% input data omitted. Please refer to example 4_2_3_med_app.m
% N=8 is the number of patients
% gamma is the overtime cost of per unit delay
% mu(N*1) represents the mean value of uncertain consultation time z
% sig(N*1) represents the standard deviation of uncertain sonsultation time z
% phi2 is the constant of phi^2

model=xprog('MA Scheduling');                    % create a model, named 'MA Scheduling'

x=model.decision(N);                             % here-and-now decision

y=model.recourse(N+1);                           % define decision rule

z=model.random(N);                               % define random varaibles
u=model.random(N+1);                             % define auxiliary variables

y.depend(z);                                     % define dependency of y on z
y.depend(u);                                     % define dependency of y on u
```

```
19
20 % extended ambiguity set is defined below
21 model.uncertain(expect(z)==mu);                          % first line of G
22 model.uncertain(expect(u(1:N)')<=sig.^2);                % second line of G
23 model.uncertain(expect(u(N+1))<=phi2);                   % third line of G
24
25 % extended support set is defined below
26 model.uncertain(z>=0);                                   % first line of \hat{W}
27 model.uncertain((z-mu).^2<=u(1:N)');                     % second line of \hat{W}
28 model.uncertain(square(sum(z)-sum(mu))<=u(N+1));         % third line of \hat{W}
29
30 model.min(expect(sum(y(1:N))+gamma*y(N+1)));             % define objective function
31
32 for n=1:N
33     model.add(y(n+1)-y(n)+x(n)>=z(n));                   % 1st constraint
34 end
35
36 model.add(y>=0);                                         % 2nd constraint
37 model.add(sum(x)<=T);                                    % 3rd constraint
38 model.add(x>=0);                                         % 3rd constraint
39
40 model.solve;                                             % solve the problem
```

The optimal solution is presented in the following table.

Table 4.2. Optimal Solution

Optimal	Optimal Solution $\boldsymbol{x}$							
Objective	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
222.32	54.25	52.94	51.38	49.61	47.37	50.65	43.34	29.54

References

[1] D. Bertsimas, M. Sim, and M. Zhang, "Distributionally adaptive optimization," March 2016. [Online]. Available: http://www.optimization-online.org/DB_HTML/2016/03/5353.html

[2] ——, "A practicable framework for distributionally robust linear optimization," July 2013. [Online]. Available: http://www.optimization-online.org/DB_HTML/2013/07/3954.html

[3] W. Wiesemann, D. Kuhn, and M. Sim, "Distributionally robust convex optimization," *Oper. Res.*, vol. 62, no. 6, pp. 1358–1376, Dec. 2014.

[4] A. Ben-Tal, A. Goryashko, E. Guslitzer, and A. Nemirovski, "Adjustable robust solutions of uncertain linear programs," *Mathematical Programming*, vol. 99, pp. 351–376, 2004.

[5] S. Garstka and R. Wets, "On decision rules in stochastic programming," *Mathematical Programming*, vol. 7, no. 1, pp. 117–143, 1974.

[6] W. Chen and M. Sim, "Goal-driven optimization," *Operations Research*, vol. 57, no. 2, pp. 342–357, 2009.

[7] Y. Degeilh and C. Singh, "A quantitative approach to wind farm diversification and reliability," *International Journal of Electrical Power & Energy Systems*, vol. 33, no. 2, pp. 303–314, 2011.

[8] J. L. Higle, "Stochastic programming: optimization when uncertainty matters," *TutORials in operations research. INFORMS, Baltimore*, pp. 30–53, 2005.

[9] D. Bertsimas and M. Sim, "The price of robustness," *Operations Research*, vol. 52, pp. 35–53, 2004.