Robust Optimization Made Easy with ROME

Joel Goh* Melvyn Sim †

July 2009

Abstract

We introduce an algebraic modeling language, named ROME, for a class of robust optimization problems. ROME serves as an intermediate layer between the modeler and optimization solver engines, allowing modelers to express robust optimization problems in a mathematically meaningful way. In this paper, we highlight key features of ROME which expediates the modeling and subsequent numerical analysis of such problems. We conclude with two comprehensive examples on how to model (1) a service-constrained robust inventory management problem, and (2) a robust portfolio selection problem using ROME. ROME is freely distributed for academic use from `www.robustopt.com`.

*NUS Business School, University of Singapore. Email: joelgoh@nus.edu.sg

†NUS Business School and NUS Risk Management Institute, National University of Singapore. Email: dscsimm@nus.edu.sg. The research of the author is supported by Singapore-MIT Alliance and NUS academic research grant R-314-000-068-122.

1 Introduction

Robust optimization was originally used to protect optimization problems from infeasibility caused by uncertainties in the model parameters (e.g. the models studied by Soyster [33], Ben-Tal and Nemirovski [3], Bertsimas and Sim [4]). Typically, the original uncertain optimization problem is converted into an equivalent deterministic form (called the robust counterpart) using duality arguments and then solved using standard convex optimization algorithms. Ben-Tal et al. [2] substantially generalized the original robust optimization framework, and introduced the Adjustable Robust Counterpart (ARC) to allow for delayed decision-making that is dependent on the realization of all or part of the model uncertainties, allowing the robust optimization method to be used for modeling recourse decisions in two-stage and multi-stage problems. However, while the ARC allowed for richer modeling, it was also shown to be generally computationally intractable, and the authors subsequently focused on a subset of the ARC, where the primitive uncertainties were restricted to affine functions of the uncertainties, which they termed the Affinely-Adjustable Robust Counterpart (AARC), also termed Linear Decision Rules (LDRs).

Robust optimization models can be seen as a special case of minimax stochastic programs, the study of which was pioneered by Záčková [36] and subsequently furthered in other works such as [6, 15, 14, 30, 31]. In this setting, uncertainties are modeled as having a distribution which is not fully characterized, known only to lie in a family of distributions. Optimal decisions are then sought for the *worst-case* distribution within the family. Solutions are therefore *distributionally robust* toward ambiguity in the uncertainty distribution. Distributional families are typically defined by classical properties such as moments or support, or more recently introduced distributional properties such as directional deviations (Chen, Sim, and Sun [11]). Frequent choices of families of distributions used by researchers are listed in [16].

Minimax stochastic programs with recourse decisions are generally difficult to solve. Even in the case when the family of distributions consists of a single probability measure, a two-stage problem was formally shown to be generally #P-hard to solve (Dyer and Stougie [17]). While there is no formal complexity established for the standard multi-stage stochastic programming problem[1], most authors believe that it is similarly computationally intractable (see e.g. the arguments by Shapiro and Nemirovski [32]).

Instead of attempting to solve these intractable problems directly, a recent body of research in robust optimization focuses on adapting the methodology of Ben-Tal et al. [2] to obtain sub-optimal, but ultimately tractable, approximations of such problems by restricting the structure of recourse decisions to simple ones, such as LDRs. A common theme in this body of work is a search for techniques to relax the stringent affine requirement on the recourse decisions to allow for more flexible, but tractable decision rules. Such approaches include the deflected and segregated LDRs of Chen et al. [12], the extended AARC of Chen and Zhang [13], the truncated LDR of See and Sim [28], and the bi-deflected and (generalized) segregated LDRs in our previous work (Goh and Sim [18]).

[1] Dyer and Stougie [17] showed that a variant is PSPACE-hard.

Despite these recent advances in robust optimization theory and techniques, there has been a conspicuous lack of accompanying technology to aid the transition from theory to practice. Furthermore, this problem is compounded by the fact that the deterministic forms of many robust optimization models are exceedingly complex and tedious to model explicitly. We believe that the development of such technology would firstly enable robust optimization models to be applied practically and secondly allow for more extensive computational tests on theoretical robust optimization models. At present, the MATLAB-based YALMIP modeling toolbox by Löfberg [23, 24] represents the first and only publicly-available software tool for modeling of robust optimization problems. Other dominant MATLAB-based modeling toolboxes such as CVX by Grant and Boyd [19, 20] primarily solve deterministic models.

We aim to contribute toward the development of such a technology by introducing an algebraic modeling toolbox[2] for modeling robust opimization problems, named Robust Optimization Made Easy (ROME), which runs in the MATLAB environment. Using ROME, we can readily model and solve robust optimization problems within the framework presented in our earlier work (see Section 2 for a brief review of the framework, and [18] for a detailed discussion). This paper covers the public release version of ROME, version 1.0 (beta) and its sub-versions.

In terms of the classes of problems that ROME can model and solve, ROME is considerably less versatile than other MATLAB-based modeling packages such as YALMIP or CVX, which can model much more general classes of problems. The problems that can be modeled in ROME are restricted to the linear structure as described in [18]. We feel that ROME's key contribution is in its comparatively richer modeling of uncertainties, not just through their support, but also through other distributional properties such as moments and directional deviations. ROME also allows for non-anticipative requirements on decision-making, as well as piecewise-linear recourse decisions based on the bi-deflected linear decision rule in [18].

This paper is structured as follows: In Section 2, we briefly review the distributionally robust optimization framework, which ROME is designed to solve. Section 3 covers the basic structure of a ROME program, while we highlight key features of ROME in Section 4. We proceed to provide two comprehensive modeling examples in Section 5. Finally, Section 6 concludes.

Notations We denote a random variable by the tilde sign, i.e., $\tilde{x}$. Bold lower case letters such as $\boldsymbol{x}$ represent vectors and the upper case letters such as $\boldsymbol{A}$ denote matrices. In addition, $x^+ = \max\{x, 0\}$ and $x^- = \max\{-x, 0\}$. The same notation can be used on vectors, such as $\boldsymbol{y}^+$ and $\boldsymbol{z}^-$ which denotes that the corresponding operations are performed component-wise. For any set S, we will denote by $\mathbb{1}_S$ the indicator function on the set. Also, we will denote by $[N]$ the set of positive running indices to N, i.e. $[N] = \{1, 2, \ldots, N\}$, for some positive integer N. For completeness, we assume $[0] = \emptyset$. We also denote with a superscripted letter "c" the complement of a set, e.g. I^c. We denote by $\boldsymbol{e}$ the vector of all ones, and by $\boldsymbol{e}^i$ the i^{th} standard basis vector. We will use superscripted indices on vectors to index members of a collection of vectors, while subscripted indexes on a vector denotes its components, i.e. $x_i = \boldsymbol{e}^{i\prime}\boldsymbol{x}$.

[2]Freely-available for academic use from `http://www.robustopt.com`

2 Optimization Framework

We designed ROME to solve robust optimization problems cast in the framework proposed in our earlier work (Goh and Sim [18]). We briefly review this framework for completeness, and interested readers can refer to [18] for a more detailed description and some theoretical properties. We denote by $\tilde{\boldsymbol{z}}$ an N-dimensional vector of uncertainties, defined on the probability space $(\Omega, \mathcal{F}, \mathbb{P})$. We do not assume that the uncertainty distribution $\mathbb{P}$ is precisely known, but instead, we may have knowledge of certain distributional properties of $\tilde{\boldsymbol{z}}$, namely, its support, mean support, covariance matrix, and upper bounds on its direction deviations (introduced by Chen, Sim, and Sun [11]). The presence of these properties serve to characterize a family of distributions, which we generally denote by $\mathbb{F}$, which contains the actual distribution $\mathbb{P}$.

The decision variables in our framework comprise $\boldsymbol{x}$, an n-dimensional vector of decisions to be made before the realization of any of the uncertainties, as well as a set of K vector-valued recourse decisions $\boldsymbol{y}^k(\cdot)$, with image in $\Re^{m_k}$ for each $k \in [K]$. Our framework explicitly handles non-anticipative requirements, which are captured by the index sets $\{I_k\}_{k=1}^K$, where $I_k \subseteq [N]\ \forall k \in [K]$. The information index sets enforce the dependencies of the recourse decisions on the uncertainties. In the most general setting, these recourse decisions are chosen from the space of measurable functions, adhering to the non-anticipativity requirements. We denote this by a parameterized set $\mathcal{Y}(m, N, I)$, where the positive integers m and N respectively denote the dimensions of the vector spaces of the range and domain of the functions, and the index set $I \subset [N]$ captures the non-anticipative requirements. Explicitly,

$$\mathcal{Y}(m, N, I) \triangleq \left\{ \boldsymbol{f} : \Re^N \to \Re^m \ : \ \boldsymbol{f}\left(\boldsymbol{z} + \sum_{i \notin I} \lambda_i \boldsymbol{e}^i\right) = \boldsymbol{f}(\boldsymbol{z}), \forall \boldsymbol{\lambda} \in \Re^N \right\}. \tag{2.1}$$

The general model which we consider, is then:

$$\begin{array}{rlll} Z_{GEN}{}^* = & \min\limits_{\boldsymbol{x}, \left\{\boldsymbol{y}^k(\cdot)\right\}_{k=1}^K} & \boldsymbol{c}^{0\prime}\boldsymbol{x} + \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\sum\limits_{k=1}^K \boldsymbol{d}^{0,k\prime}\boldsymbol{y}^k(\tilde{\boldsymbol{z}})\right) & \\ & \text{s.t.} & \boldsymbol{c}^{l\prime}\boldsymbol{x} + \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\sum\limits_{k=1}^K \boldsymbol{d}^{l,k\prime}\boldsymbol{y}^k(\tilde{\boldsymbol{z}})\right) \leq b_l & \forall l \in [M] \\ & & \boldsymbol{T}(\tilde{\boldsymbol{z}})\boldsymbol{x} + \sum\limits_{k=1}^K \boldsymbol{U}^k\boldsymbol{y}^k(\tilde{\boldsymbol{z}}) = \boldsymbol{v}(\tilde{\boldsymbol{z}}) & \\ & & \underline{\boldsymbol{y}}^k \leq \boldsymbol{y}^k(\tilde{\boldsymbol{z}}) \leq \overline{\boldsymbol{y}}^k & \forall k \in [K] \\ & & \boldsymbol{x} \geq \boldsymbol{0} & \\ & & \boldsymbol{y}^k \in \mathcal{Y}(m_k, N, I_k) & \forall k \in [K]\,. \end{array} \tag{2.2}$$

We note the following points about the model above. Firstly, for each $l \in \{0\} \cup [M]$ and $k \in [K]$, the quantities $b_l, \boldsymbol{c}^l, \boldsymbol{d}^{l,k}, \boldsymbol{U}^k$, and I_k are model parameters which are precisely known. Similarly, $\boldsymbol{T}(\tilde{\boldsymbol{z}})$ and $\boldsymbol{v}(\tilde{\boldsymbol{z}})$ are model parameters that are assumed to be affinely-dependent on the underlying uncertainties. The upper and lower bounds on the recourse variable $\boldsymbol{y}^k(\cdot)$, respectively denoted by $\overline{\boldsymbol{y}}^k$ and $\underline{\boldsymbol{y}}^k$ are also model parameters which are possibly infinite component-wise. Secondly, the model above

contains equalities and inequalities which involve functions of the uncertainty terms $\tilde{\boldsymbol{z}}$, and we adopt the convention that these (in)equalities hold almost surely, for every distribution $\mathbb{P}$ in the family $\mathbb{F}$, i.e.

$$\boldsymbol{y}(\tilde{\boldsymbol{z}}) \geq \boldsymbol{0} \Leftrightarrow \mathbb{P}(\boldsymbol{y}(\tilde{\boldsymbol{z}}) \geq \boldsymbol{0}) = 1, \forall \mathbb{P} \in \mathbb{F}, \tag{2.3}$$

and similarly for equalities.

Many important problems can be cast into the form of problem (2.2). For example, we provide a modeling example of a multi-period inventory problem with service constraints later in this paper. Our model also encompasses multistage stochastic programs, when the family of distributions is a singleton, e.g. $\mathbb{F} = \{\mathbb{P}\}$. In addition, our description of the non-anticipativities is rich enough to allow us to model more general non-anticipative situations other than a sequential decision process, which, for example, may occur in network problems.

Unfortunately, while problem (2.2) is quite general, it is also computationally intractable in most cases (see Ben Tal et al. [2] or Shapiro and Nemirovski [32] for a more detailed discussion). To address the issue of computational tractability, we consider a linear approximation, where we restrict the recourse decisions to affine functions of the uncertainties, called linear decision rules (LDRs). Formally, we define:

$$\mathcal{L}(m, N, I) \triangleq \left\{ \boldsymbol{f} : \Re^N \to \Re^m : \exists \boldsymbol{y}^0 \in \Re^m, \boldsymbol{Y} \in \Re^{m \times N} : \boldsymbol{f}(\boldsymbol{z}) = \boldsymbol{y}^0 + \boldsymbol{Y}\boldsymbol{z}, \boldsymbol{Y}\boldsymbol{e}^i = \boldsymbol{0}, \forall i \in I^c \right\}. \tag{2.4}$$

Using this notation, we have

$$\boldsymbol{y}^k \in \mathcal{L}(m_k, N, I_k) \quad \forall k \in [K]\,. \tag{2.5}$$

In this model, we seek to find optimal decisions $\boldsymbol{x}, \left\{\boldsymbol{y}^k(\cdot)\right\}_{k=1}^K$ for the following linear problem:

$$\begin{array}{rll}
Z_{LDR} = \min\limits_{\boldsymbol{x}, \left\{\boldsymbol{y}^k(\cdot)\right\}_{k=1}^K} & \boldsymbol{c}^{0\prime}\boldsymbol{x} + \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}} \left(\sum\limits_{k=1}^K \boldsymbol{d}^{0,k\prime} \boldsymbol{y}^k(\tilde{\boldsymbol{z}}) \right) & \\
\text{s.t.} & \boldsymbol{c}^{l\prime}\boldsymbol{x} + \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}} \left(\sum\limits_{k=1}^K \boldsymbol{d}^{l,k\prime} \boldsymbol{y}^k(\tilde{\boldsymbol{z}}) \right) \leq b_l & \forall l \in [M] \\
& \boldsymbol{T}(\tilde{\boldsymbol{z}})\boldsymbol{x} + \sum\limits_{k=1}^K \boldsymbol{U}^k \boldsymbol{y}^k(\tilde{\boldsymbol{z}}) = \boldsymbol{v}(\tilde{\boldsymbol{z}}) & \\
& \underline{\boldsymbol{y}}^k \leq \boldsymbol{y}^k(\tilde{\boldsymbol{z}}) \leq \overline{\boldsymbol{y}}^k & \forall k \in [K] \\
& \boldsymbol{x} \geq \boldsymbol{0} & \\
& \boldsymbol{y}^k \in \mathcal{L}(m_k, N, I_k) & \forall k \in [K]\,.
\end{array} \tag{2.6}$$

Restricting recourse decisions to LDRs is admittedly rather severe, and we will later show how we can relax this restriction to allow for more complex decision rules, namely the bi-deflected LDR (BDLDR) and segregated LDR (SLDR) introduced by Goh and Sim [18]. For now, the basic LDR suffices as a skeletal structure on which we can build our modeling toolbox.

3 ROME Basics

We describe the basic structure of a model in ROME, to provide readers with an overall view of how ROME works. In this section, we will focus on describing general structures, and omit detailed

descriptions. In Section 4 which follows, we describe in greater depth various features of ROME which we believe makes it a suitable language for modeling uncertain optimization problems. Code Segment 1 depicts the various portions of a program in ROME.

Users enter the ROME environment by issuing a call to `rome_begin`, which allocates the necessary internal structures for modeling in ROME. Each optimization problem in ROME is encapsulated in a `rome_model` object, which serves as a repository for collating all data (variables, constaints, objective) for a particular problem. A `rome_model` object is created by the epynonymous `rome_model` function, which accepts an optional string argument representing a descriptive name for the model. This returns a handle to the model object, which in turn is used to access model data. Multiple `rome_model` objects can exist in the same ROME environment. Entering the ROME environment, and allocating a model object forms the preamble for all ROME programs.

For a given `rome_model` we can create variables, which are represented as objects from the `rome_var` class. Variables are classified in three basic subtypes: deterministic variables, uncertainties, and LDRs. The modeler can set various distributional properties (support, mean, covariance matrix, directional deviations) on uncertainties, which represents some partial characterization of the uncertainties. We furnish details on uncertainties and their properties in Section 4.1. Uncertainties and their properties should always be declared at the start of each model.

The main body of the modeling code has similar components as other mathematical programming languages, where other variables (deterministic and LDRs) are declared. LDRs are not typical in traditional deterministic programming, and we describe the declaration for LDRs (as well as non-anticipativity) in more detail in Section 4.2. We can set various linear and some nonlinear constraints, and we will have to specify an objective for the model. Standard arithmetic operations (addition, array and matrix mutiplication, indexing, etc) have been overloaded to work with variables in ROME, and we refer interested readers to the User's Guide for a full list. In addition, we also overload the `mean` operator for uncertain and LDR variables, which corresponds to taking worst-case expectations, e.g. in the objective and first M constraints of problem (2.6).

Next, the model in ROME is concluded by a call to the `solve` member function of the `rome_model` object. In Section 4.4 we discuss an alternate call to `solve_deflect` to invoke bi-deflected LDRs [18] in the model. After completing the call to `solve`, and if the underlying solver engine indicates that an optimal solution has been found, the optimal solution is encoded within the model object, and the modeler now is free to extract the optimal solution from the model for further analysis. Section 4.6 details how the solutions can be extracted, interpreted, and manipulated.

Finally, a call to `rome_end` concludes the ROME session and clears all memory allocated within ROME structures. We note that this call is optional, and provided as a convenience for users to remove only the memory allocated by ROME. Calling `rome_end` will erase all data within all model objects.

4 ROME Features

In this section, we describe several features of ROME. We do not attempt to describe all of ROME's functionality in detail (we refer interested readers to the detailed User's Guide). Instead we hope to

```
% ROME_STRUCTURE.m
% File to illustrate ROME's structure. No real model to be solved.

% PREAMBLE
% Begins ROME Environment and creates model object
rome_begin;                          % Begins the ROME Environment
h = rome_model('Dummy File');        % Creates a model object

% MAIN CODE BODY
% Declare variables, objective and constraints
% Declare uncertain variables first, and set their distributional properties,
% mean, covariance, directional deviations, support. Ends with call to solve.
% Declare variables:
newvar z(5, 1) uncertain;            % Declare z as 5 x 1 uncertainty vector
z.set_mean(0);                       % Set mean of z
z.Covar = 1;                         % Set covariance of z

newvar x(3, 1);                      % Declare 3 x 1 variable x
newvar y(3, 4, z) linearrule;        % Declare 3 x 4 LDR variable y

% Declare constraints:
rome_constraint(x(2:3) <= 2*x(1:2)); %
rome_constraint(sum(y, 1) == 1);     % Various constraints can be applied.
rome_constraint(y(:, 1) + x >= 2);   % ROME variables can be manipulated by
rome_constraint(norm2(x) <= 1);      % standard arithmetic operations.
rome_constraint(mean(y(2, :)) <= 5); %

% Declare objective:
rome_minimize(sum(x) + sum(y(:, 2))); % objective

% Instruct ROME to solve
h.solve;                             % Terminate the current model with solve

% GET RESULTS
% Get optimization results
y_sol = h.eval(y);                   % Get y
u_sol = h.eval(y(2, 2) + x(1));      % Get function of variables

% CLEANUP
% Clear up ROME memory
rome_end;                            % Complete modeling and deallocate ROME memory
```

Code Segment 1: Dummy code illustrating ROME structure

describe the features of ROME which makes it particularly useful for modeling robust optimization problems. Namely, we focus on several important features, (1) uncertainty description, (2) primitive handling of LDRs, (3) non-anticipative requirements, (4) support for BDLDRs, (5) in-built worst-case bounds on the expected positive part of LDRs, and (6) analysis of optimization results.

4.1 Uncertainty Description

Uncertainties are primitive objects in ROME, and can be declared using the `newvar` command with the `uncertain` property, and can be constructed in a variety of sizes. For example, the following code snippet can be used to declare various sizes of model uncertainties:

```
newvar z1       uncertain;  % create scalar uncertainty
newvar z2(5)    uncertain;  % create 5 x 1 uncertainty column vector
newvar z3(4, 3) uncertain;  % create 4 x 3 uncertainty matrix
```

Code Segment 2: Declare uncertain variables

The variables `z1`, `z2`, and `z3` are now recognized in the ROME environment as model uncertainties. We can now set distributional properties of the uncertainties, such as their support, mean, covariance, and directional deviations. The following code snippet illustrates how a vector uncertainty can be declared and how its distributional properties can be set.

```
newvar z(10, 1)   uncertain;  % create 10 x 1 uncertainty vector

rome_constraint(z.mean == 1); % Sets the mean to be exactly 1

rome_constraint(z >= 0);      % Sets the support of z to be
rome_constraint(z <= 2);      % between 0 and 2 component-wise

z.Covar = eye(10);            % Sets an identity covariance matrix

z.FDev = 1.8*ones(10, 1);     % Sets the forward deviation
z.BDev = 1.8*ones(10, 1);     % Sets the backward deviation
```

Code Segment 3: Setting distributional properties of uncertainties

The equivalent mathematical description of the family of distributions, $\mathbb{F}$, would be

$$\mathbb{F} = \left\{\mathbb{P} : \mathrm{E}_{\mathbb{P}}\left(\tilde{\boldsymbol{z}}\right) = \boldsymbol{e}, \mathrm{E}_{\mathbb{P}}\left(\tilde{\boldsymbol{z}}\tilde{\boldsymbol{z}}'\right) = \boldsymbol{I} + \boldsymbol{e}\boldsymbol{e}', \mathbb{P}(\boldsymbol{0} \leq \tilde{\boldsymbol{z}} \leq 2\boldsymbol{e}) = 1, \boldsymbol{\sigma}_{\boldsymbol{f}_{\mathbb{P}}}(\tilde{\boldsymbol{z}}) \leq 1.8\boldsymbol{e}, \boldsymbol{\sigma}_{\boldsymbol{b}\mathbb{P}}(\tilde{\boldsymbol{z}}) \leq 1.8\boldsymbol{e}\right\}.$$

When distributional properties of the uncertainties are set, they are stored in the current model, and used internally when required. In the code segment above, we have used `rome_constraint`, a key function in ROME to set the support and mean of $\tilde{\boldsymbol{z}}$. If the mean of $\tilde{\boldsymbol{z}}$ were also uncertain, say, only known component-wise to take values between 0.9 and 1.1, we could use the code below to specify this uncertain mean.

```
rome_constraint(z.mean >= 0.9); % Sets the lower bound on the mean to be 0.9
rome_constraint(z.mean <= 1.1); % Sets the upper bound on the mean to be 1.1
```

Code Segment 4: Setting support for uncertain means

ROME accepts general polytopic regions for both support and mean support, which can be expressed as a finite set of linear inequalities on $\tilde{\boldsymbol{z}}$. For some pre-defined parameters `A` and `b`, a polytopic region for the support (and analagously for the mean support) can be defined as follows:

```
w = A * z;                  % Assign to a placeholder w
rome_constraint(w <= b);    % Apply the constraint A*x <= b
```

Code Segment 5: Defining a polytopic support

In addition, ROME 1.0.4 and later allows for specification of ellipsoidal support sets (Ben-Tal and Nemirovski [3]) using the function `rome_constraint`, as follows:

```
rome_constraint(norm2(A*z + b) <= 1);   % For appropriately sized A and b
```

Code Segment 6: Defining an ellipsoidal support

4.2 Primitive Handling of LDRs

While ROME provides an interface for the input of distributional properties of uncertainties into a model, utimately, the true utility of this feature is so that the underlying machinery of ROME can handle the uncertainties, while the modeler is free to focus on the structuring the problem in a meaningful way. ROME achieves this by defining LDRs as atomic object, which allows modelers to declare and manipulate LDRs directly. This allows modelers to write code which is physically meaningful, and frees them from the bookkeeping of how LDRs are internally represented. The following code snippets creates LDRs of different dimensions, affinely dependent on the uncertainty `z`:

```
newvar z(5)         uncertain ;  % make uncertainty vector
newvar y1(z)        linearrule;  % create scalar LDR
newvar y2(3, z)     linearrule;  % create 3 x 1 LDR
newvar y3(4, 3, z) linearrule;  % create 4 x 3 LDR
```

Code Segment 7: Declaring Linear Decision Rules (LDRs)

We notice that the declaration of the variables `y1`, `y2`, and `y3` are invariant to the dimensions of `z`. After the variable declarations, the LDR variables can be manipulated by common operations such as addition, subtraction, subscripted indexing. In addition, we can apply linear equality and inequality constraints to the variables, again using the `rome_constraint` function. This level of abstraction makes modeling robust optimization problems in ROME much simpler than explicitly modeling the robust counterpart.

For example, suppose that we have known parameters $\boldsymbol{U}, \boldsymbol{b}$, and a fully specified affine function $\boldsymbol{v}(\boldsymbol{z}) = \boldsymbol{v}^0 + \boldsymbol{V}\boldsymbol{z}$, i.e. with known parameters $\boldsymbol{v}^0$ and $\boldsymbol{V}$. We wish to model the following constraints

$$\begin{aligned} \boldsymbol{U}\boldsymbol{y}(\tilde{\boldsymbol{z}}) &= \boldsymbol{v}(\tilde{\boldsymbol{z}}) \\ \boldsymbol{y}(\tilde{\boldsymbol{z}}) &\geq \boldsymbol{0}, \end{aligned}$$

where the uncertainty $\tilde{\boldsymbol{z}}$ has a full-dimensional polyhedral support set $\mathcal{W}$. From standard robust optimization techniques (Ben-Tal and Nemirovski [3], Bertsimas and Sim [4]), we have to convert the

constraints above into its robust counterpart before it can be solved. The equality constraint should be expanded into $N+1$ equality constraints, equating the coefficients of component of $\tilde{\boldsymbol{z}}$, while the inequality constraint should be expanded into a series of inequalities involving the set $\mathcal{W}$.

While it is certainly possible to perform these conversion steps manually, they are also rather tedious (and thereby quite prone to human error). More importantly, the final deterministic form of the robust counterpart is structurally more complex, involving several dual variables, constraints on the dual polytope, and consequently does not readily lend itself to interpretation. To address these issues, ROME handles this conversion internally, and the constraints above can be expressed in ROME using the few lines of code,

```
newvar y(10, z) linearrule;     % create a 10 x 1 vector LDR
v = v0 + V * z;                 % make an affine function of z
rome_constraint(U * y == v);    % apply equality constraint
rome_constraint(y >= 0);        % apply inequality constraint
```

Code Segment 8: Constraints in ROME

where we assume that the variables `v0`, `V`, and `U` have been previously defined and have consistent dimensions, and that the support of the uncertainty vector `z` has been previously specified, as in the preceding section, Section 4.1.

4.3 Non-anticipative Requirements

Non-anticipative requirements can also be easily be applied to LDR variables. Suppose we have a $N \times 1$ uncertainty vector, `z`, and we wish to create a $N \times 1$ LDR, `y`, such that the i^{th} component of the LDR depends on the components of `z` from 1 to i. This is a common non-anticipative requirement in temporal multi-stage modeling, where LDR `y` can only depend on realized uncertainties in the past. This can be achieved in ROME using a loop, as in Code Segment 9. In each iteration of the loop, we create an LDR which depends on a subset of the full uncertainties, with an increasing running index on `z`.

```
newvar z(N) uncertain;                  % create an uncertainty vector
y = [];                                 % allocate an empty matrix
for ii = 1:N
    newvar tmp(1, z(1:ii)) linearrule;  % create an LDR in each iteration
    y = [y; tmp];                       % append to the output matrix
end
```

Code Segment 9: Declaring non-anticipative variables with loops

MATLAB is optimized for vectorized code instead of loops, and Code Segment 9, though simple to understand, is quite inefficient. A more efficient way of achieving the same end is to create the `y` with a pre-specified binary 'pattern', which enforces the dependencies on the linear rules. For this example, the pattern takes the form of a lower triangular matrix of ones. The following Code Segment 10 efficiently creates `y` with the same non-anticipative requirement as before:

```
pY = [true(N, 1), tril(true(N, N))];        % augmented lower triangular pattern
newvar z(N) uncertain;                      % create an uncertainty vector
newvar y(N, z, 'Pattern', pY) linearrule;   % construct the LDR with the pattern
```

Code Segment 10: Declaring non-anticipative variables directly

After declaring `y` in either of these ways, we can use `y` as a variable in ROME, which has the specified dependency structure on the uncertainties `z`.

4.4 Support for BDLDRs

The final key feature which we highlight here is ROME's in-built support for more complex decision rules, namely the bi-defected linear decision rule (BDLDR). The BDLDR is a piecewise linear decision rule introduced by Goh and Sim [18] that is based upon (and improves) the LDR, and was shown to generalize the DLDR of Chen et al. [12]. In addition, the BDLDR can also handle non-anticipative requirements. However, the downside of the BDLDR is its structural complexity, especially if the modeler requires non-anticipativity. Indeed, its explicit construction involves a series of steps, which we describe briefly here, to give the reader an idea of the complexity involved to use the BDLDR.

To explicitly construct the BDLDR, the modeler has to extract structural patterns within the original optimization problem and solve a series of sub-problems to compute the coefficients on the deflected (non-linear) components. Finally, using the results of the sub-problems, the modeler has to modify the original optimization problem, and apply some distributionally robust bounds on the expectation of the positive part of a LDR, to convert the problem into a deterministic form, and finally solved using standard solver packages.

While the steps involved might be complex, they are sufficiently mechanical to be automated in ROME. To use the BDLDR in ROME, we would model the problem identically as if we were using LDRs alone, with the sole exception that the modeler will issue a call to the member function `solve_deflected` to complete the modeling, instead of the usual call to `solve`. We built this functionality into ROME with the design consideration that regardless of the complexity of the BDLDR (or any other decision rule for that matter), the model in code should aim to closely resemble the algebraic mathematical model.

The call to `solve_deflected` instructs ROME to internally perform the BDLDR transformation steps described above, and automatically modifies the model into the final form. The distributional properties that are specified for `z` and are stored in the model, are internally used in the construction of the required distributionally robust bounds. This complexity, however, is transparent to the modeler, and the modeler will only need to extract the piecewise-linear decision rule, which is the result of the final solve. We refer interested readers to the examples in Section 5 for an illustration of how the BDLDR can be used in ROME.

4.5 Worst-case Bounds on the Expected Positive Part of an LDR

In our previous work [18], we showed that together with the mean, each of the distributional properties: support, covariance matrix, and directional deviations, led to different bounds on the expected part of an LDR, or explicitly, $\sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\left(y^0 + \boldsymbol{y}'\tilde{\boldsymbol{z}}\right)^+\right)$. As alluded to in the previous subsection, these bounds are used internally when converting the BDLDR solution into its deterministic form prior to solving.

These bounds can also be invoked directly by the modeler through calls the respective functions `rome_supp_bound`, `rome_covar_bound`, and `rome_dirdev_bound`, which are used internally by the BDLDR conversion function. These functions use the corresponding (previously declared) distributional properties of the uncertainties to form the individual bounds. For uncertainties that are described by multiple distributional properties, we can use the technique of infimal convolution (see Goh and Sim [18] or Chen and Sim [10]) to combine these bounds into a unified, and better bound, on $\sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\left(y^0 + \boldsymbol{y}'\tilde{\boldsymbol{z}}\right)^+\right)$. This feature has been implemented in ROME through the function `rome_create_bound`, which accepts a comma-separated list of function handles, convolving them to construct the unified bound.

4.6 Analysis of Optimization Results

In deterministic mathematical programming, an optimization routine typically outputs in numerical values being assigned to decision variables (for feasible problems). In the case of optimization with uncertainties having recourse decisions, we typically seek solutions that are parameterized *functions* of the model uncertainties. Consequently, the handling and analysis of the optimization is more complex than in deterministic programming, and requires some special mention.

After the modeler issues a call to the `solve` member function, assuming that the underlying solver engine reports that an optimal solution has been found, the modeler can extract solutions for various variables in ROME, using the `eval` member function. `eval` accepts a single argument, the variable of interest to be evaluated, and outputs an object from the class `rome_sol` which encapsulates numerical values and methods required to analyze the values of the requested variable.

Suppose, for full generality, that the variable of interest, which we denote here by `x`, is one which has non-linear (deflected) components. Then the output of `eval`, which we denote by `x_sol`, will internally store the linear and deflected components of the optimized solution. We notice that a deflected variable $\hat{\boldsymbol{x}}(\tilde{\boldsymbol{z}})$, can be expressed as a piecewise-linear function of the model uncertainties $\tilde{\boldsymbol{z}}$ as follows:

$$\hat{\boldsymbol{x}}(\tilde{\boldsymbol{z}}) = \left(\boldsymbol{x}^0 + \boldsymbol{X}\tilde{\boldsymbol{z}}\right) + \boldsymbol{P}\left(\boldsymbol{y}^0 + \boldsymbol{Y}\tilde{\boldsymbol{z}}\right)^- \tag{4.1}$$

where $\boldsymbol{x}^0 \in \Re^n, \boldsymbol{X} \in \Re^{n\times N}, \boldsymbol{y}^0 \in \Re^m, \boldsymbol{Y} \in \Re^{m\times N}, \boldsymbol{P} \in \Re^{n\times m}$ are matrices (vectors) of numerical values which are obtained after optimization, and fully characterize the optimal deflected solution of the variable $\hat{\boldsymbol{x}}(\tilde{\boldsymbol{z}})$. We proceed to describe how several member functions can be used to extract these parameters, and how they can be interpreted.

Applying the `linearpart` function to `x_sol` will return the parameters $(\boldsymbol{x}_0, \boldsymbol{X})$ as a single matrix, while applying the `deflectedpart` function to `x_sol` will return two arguments. The first argument

is the matrix $\boldsymbol{P}$ of coefficients, while the second is the parameters $(\boldsymbol{y}_0, \boldsymbol{Y})$ as a single matrix. These parameters are sufficient to reconstruct the optimal deflected variable $\hat{\boldsymbol{x}}(\tilde{\boldsymbol{z}})$ using equation (4.1).

ROME also provides a "prettyprint" functionality, for ease of analysis. While the `linearpart` and `deflectedpart` functions above are useful for numerical computation, the prettyprint is easier for human analysis and code prototyping for small programs. The prettyprint is the default display for `rome_sol` objects, and will be displayed whenever an assignment to a `rome_sol` object is not terminated by a semi-colon.

We also can use `rome_sol` objects for Monte-Carlo simulation analysis of the optimization results. It is typically useful to compare the result of the robust optimization solution with benchmarks from alternate algorithms or techniques. We can do this easily in ROME. Since each `rome_sol` object represents a function of the model uncertainties, we can instantiate the `rome_sol` object (using the provided `insert` function) with realizations of uncertainties to obtain concrete numeric decisions, which can be compared *vis-a-vis* decisions obtained from other methods. Code Segment 11 illustrates how to use a loop to instantiate a solution object with a series of standard normal variables. Vectorized instantiation is also permitted by using a flag in the instantiation, as in Code Segment 12.

```
1    x_sol = h.eval(x);          % Get the solution object
2    z_vals = randn(N, 100);     % Draw r.v.s from standard normal distribution
3    x_vals = zeros(M, 100);     % Allocate output matrix
4    for ii = 1:100
5        % instantiate solution with r.v.
6        x_vals(:, ii) = x_sol.insert(z_vals(:, ii));
7    end
```

Code Segment 11: Instantiating a solution with uncertainties

```
1    x_sol = h.eval(x);          % Get the solution object
2    z_vals = randn(N, 100);     % Draw r.v.s from standard normal distribution
3    vectorize_flag = true;      % Set the vectorize flag to true
4    x_vals = x_sol.insert(z_vals, vectorize_flag); % instantiate
```

Code Segment 12: Instantiating a solution with uncertainties (Vectorized)

Instantiating solutions with uncertainties is not only useful for Monte-Carlo simulation analysis and benchmarking, but we can use this even as a prescriptive tool for non-anticipative modeling. Suppose we have some solution `y_sol` after an optimization in ROME, which we have explicitly constrained to only depend on the first component of an N-dimensional uncertainty vector, i.e. $\boldsymbol{y}^{sol}(\tilde{\boldsymbol{z}}) = \boldsymbol{y}^{sol}(\tilde{z}_1)$. Now suppose we are at the point in our timeline when we are ready to make the decision corresponding to `y_sol`, i.e. the first component of uncertainty has been revealed, with, say, a numerical value of 10. We can instantiate `y_sol` with an uncertainty vector with 10 as the first component and zeros (or random numbers) on the other components to find the numerical value of the decision corresponding

to `y_sol`, since by the explicit non-anticipative constraints, `y_sol` is invariant to all other components of the uncertainty vector. Code Segment 13 illustrates this in ROME:

```
1    z_val = [10; zeros(N-1, 1)]; % Make instantiation vector
2    y_val = y_sol.insert(z_val)  % Instantiate
```

Code Segment 13: Using `insert` to prescribe non-anticipative decisions

4.7 Simple Example

We present a simple example to demonstrate the above features in ROME (except the non-anticipative LDR feature, which we illustrate in the longer examples in Section 5). We consider a very simple robust optimization problem, which fits into the general form of model (2.6):

$$\begin{aligned} Z = \min_{y(\cdot),u(\cdot),v(\cdot)} \quad & \sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(u(\tilde{z}) + v(\tilde{z})\right) \\ \text{s.t.} \quad & u(\tilde{z}) - v(\tilde{z}) = y(\tilde{z}) - \tilde{z} \\ & 0 \le y(\tilde{z}) \le 1 \\ & u(\tilde{z}), v(\tilde{z}) \ge 0 \\ & y, u, v \in \mathcal{L}\left(1, 1, \{1\}\right), \end{aligned} \tag{4.2}$$

where the model uncertainty consists of a scalar variable $\tilde{z}$, which is known to have a distribution $\mathbb{P}$, which lies in the family of distributions $\mathbb{F}$ with infinite support, zero mean, and unit variance. Explicitly,

$$\mathbb{F} = \left\{\mathbb{P} : \mathrm{E}_{\mathbb{P}}\left(\tilde{z}\right) = 0, \mathrm{E}_{\mathbb{P}}\left(\tilde{z}^2\right) = 1\right\}. \tag{4.3}$$

The corresponding code in ROME which models this problem is shown in Code Segment 14. We note that the code has a natural correspondence with the mathematical formulation. For example, lines 12 – 13 corresponds with the characterization of the family of uncertainties in equation (4.3). Also, the objective and first two constraints in problem (4.2) corresponds with the lines 19 – 24 of the code. In fact, we could even explicitly write out the $u(\tilde{z}), v(\tilde{z}) \ge 0$ constraints of problem (4.2) in code, but we chose to express these constraints implicitly in the declaration of the `u, v` variables on line 16 using the `nonneg` keyword.

If we were to model problem (4.2) exactly, we would have to uncomment line 28 and comment out line 27 (to call `solve` instead of `solve_deflected`). However, it turns out that using the basic LDR to solve problem (4.2) will result in an infeasible solution. We therefore call `solve_deflected` to invoke the more complex BDLDR for a better solution. As an aside, we note the seamless manner in which ROME hides the complexity and internal machinery of the BDLDR; indeed, the corresponding code using the BDLDR is almost identical to the code using the LDR. Using the BDLDR, the corresponding output of the program is:

```
1  y_sol =
2
3  -0.000 + 1.000*z1 + 1.00(-0.000 + 1.000*z1)^- - 1.00(1.000 - 1.000*z1)^-
```

Code Segment 15: Output from simple_example.m

```
% SIMPLE_EXAMPLE.m
% Script to demonstrate several key features and functions in ROME,
% and illustrate overall structure of a ROME program

% ROME MODELING CODE
% Preamble
rome_begin;                              % Begins the ROME Environment
h = rome_model('Simple Example');        % Creates a model object

% Feature 1: Handling Uncertainties
newvar z uncertain;                      % Declare z as a scalar uncertainty
z.set_mean(0);                           % Set distributional properties of z
z.Covar = 1;                             % zero mean and unit variance.

% Feature 2: Declaring LDRs as primitive objects
newvar u(z) v(z) linearrule nonneg;      % nonnegative LDRs u and v
newvar y(z) linearrule;                  % LDR y

% Objective
rome_minimize(mean(u + v));

% Constraints
rome_constraint(u - v == y - z);         % equality constraint
rome_box(y, 0, 1);                       % 0 <= y <= 1 constraint

% Feature 4: Support for BDLDRs
h.solve_deflected;               % Uses BDLDR as decision rules
% h.solve;                       % Use this instead to use basic LDRs as decision rules

% Feature 5: Extract solution for analysis
y_sol = h.eval(y);               % Call 'eval' after solve and before rome_end

% Complete modeling and deallocate ROME memory
rome_end;

% ANALYZE RESULTS
y_sol                            % Notice no semi-colon, prettyprints y

% Monte Carlo simulation
z_vals = randn(1, 100);          % Get r.v.s from some distribution within family
vec_flag = true;                 % Set vectorize flag to true

% instantiate solution
y_vals = y_sol.insert(z_vals, vec_flag);
```

Code Segment 14: ROME Code for Simple Example

which can be interpreted as

$$\begin{aligned} \hat{y}(\tilde{z}) &= (\tilde{z})^+ - (\tilde{z}-1)^+ \\ &= \min\left\{\max\left\{\tilde{z}, 0\right\}, 1\right\} \end{aligned} \tag{4.4}$$

as expected.

5 Modeling Examples

5.1 Service-constrained Inventory Management

5.1.1 Problem Description

In this section, we model a robust multi-period inventory management problem in ROME and study some computational studies. Classically, stockouts are penalized using a linear backorder cost rate (underage cost) within the objective function. While using a linear underage cost rate is mathematically convenient, this cost rate may be difficult to quantify in practice. An alternate model of penalizing stockouts is to use explicit service constraints to guarantee a lower limit on the event or quantity of non-stockouts. Because of the mathematical difficulty of handling service constraints, a technique commonly used in the inventory management literature is to approximate the service constraint using backorder costs (e.g. Boyaci and Gallego [5], Shang and Song [29]), and derive the optimal policy based on a known demand process. In our model, since we do not presume precise knowledge of the demand distribution, we cannot apply similar approximations. Instead, we handle the service constraints explicitly by considering a distributionally robust service guarantee on the *fill rate*, which is a common metric used by industry practitioners. The robust inventory model that we consider here is similar to that considered by See and Sim [28], although they also use a penalty cost rate instead of a service constraint[3].

Notation: We consider an inventory management problem with a planning horizon of T periods, and a primitive uncertainty vector, $\tilde{\boldsymbol{z}} \in \Re^T$, representing the exogenous uncertain demand in each period. The exact distribution of $\tilde{\boldsymbol{z}}$ is unknown, but the inventory manager has knowledge of some distributional information, which characterizes a family $\mathbb{F}$ of uncertainty distributions. For simplicity, we shall assume that we know the mean of $\tilde{\boldsymbol{z}}$ exactly, i.e. $\mathrm{E}_{\mathbb{P}}\left(\tilde{\boldsymbol{z}}\right) = \boldsymbol{\mu},\ \forall \mathbb{P} \in \mathbb{F}$, where $\boldsymbol{\mu} > \mathbf{0}$. A similar formulation can be done (albeit with more notation) for random means, provided that the mean support is fully contained within the positive orthant. Explicitly, we require $\hat{\mathcal{W}} \subseteq \left\{\boldsymbol{x} \in \Re^T : \boldsymbol{x} > \mathbf{0}\right\}$.

In each period $t \in [T]$, we denote by c_t the ordering cost per unit and the maximum order quantity by x_t^{MAX}. Undelivered inventory at the end of each period can be held over to the next period, with a per unit holding (overage) cost of h_t. We assume no fixed cost components to all costs involved. We denote the minimum required fill rate in each period by β_t. We shall assume that the goods have no salvage value at the end of the T periods, and that we possess no starting inventory. We also assume that the inventory manager is ambiguity-averse and aims to minimize the worst-case expected loss over all distributions in $\mathbb{F}$.

[3]ROME code for both their model and ours can be freely obtained from `http://www.robustopt.com/examples.html`

5.1.2 Problem Model

We let $x_t(\tilde{\boldsymbol{z}}) = x_t^0 + \boldsymbol{x}_t'\tilde{\boldsymbol{z}}$, $\forall t \in [T]$ be a set of linear decision rules representing the order quantity in the t^{th} period. We let the auxilliary linear rule $y_t(\tilde{\boldsymbol{z}})$, $\forall t \in [T]$ represent the inventory level at the end of each period, and $y_0(\tilde{\boldsymbol{z}})$ represent the starting inventory. Vector quantities (e.g. $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{x}^{MAX}$) will be used to represent their respective quantities (respectively order quantity, inventory level, maximum order quantity) over all periods. For example, to represent the order quantity over all periods, we have $\boldsymbol{x}(\tilde{\boldsymbol{z}}) = \boldsymbol{x}^0 + \boldsymbol{X}\tilde{\boldsymbol{z}}$.

Since we have no starting inventory, the inventory balance equation can be written in vector form for $\boldsymbol{y}(\tilde{\boldsymbol{z}}) = [y_1(\tilde{\boldsymbol{z}}) \ldots, y_T(\tilde{\boldsymbol{z}})]$, as

$$\boldsymbol{D}\boldsymbol{y}(\tilde{\boldsymbol{z}}) - \boldsymbol{x}(\tilde{\boldsymbol{z}}) \;=\; -\tilde{\boldsymbol{z}},$$

where the constant matrix $\boldsymbol{D} \in \Re^{T\times T}$ is a first-order differencing matrix:

$$\boldsymbol{D} = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ -1 & 1 & 0 & \ldots & 0 \\ 0 & -1 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{bmatrix}.$$

The non-anticipative requirement in this case is straightforward. To make an order decision in a particular period, t, we are permitted to use realized demands in all preceding periods. We express this non-anticipative requirement on the LDRs $x_t(\cdot)$ and $y_t(\cdot)$, $\forall t \in [T]$, by

$$\begin{aligned} x_t &\in \mathcal{L}(1, T, [t-1]), \\ y_t &\in \mathcal{L}(1, T, [t]). \end{aligned}$$

Using the definition of fill rate provided in Cachon and Terwiesch [7], the robust fill rate constraint can be expressed in vector form as:

$$\sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\boldsymbol{y}(\tilde{\boldsymbol{z}})^-\right) \leq (\boldsymbol{I} - \mathbf{diag}(\boldsymbol{\beta}))\boldsymbol{\mu},$$

where $\mathbf{diag}(\cdot) : \Re^T \to \Re^{T\times T}$ represents the diagonalization operator. We note that the expectation operation is applied component-wise. Thus, the ambiguity-averse cost-minimizer can be obtained as the solution to the following problem

$$\begin{aligned} Z_{LDR} = \min_{\boldsymbol{x}(\cdot),\boldsymbol{y}(\cdot)} \quad & \sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\boldsymbol{c}'\boldsymbol{x}(\tilde{\boldsymbol{z}}) + \boldsymbol{h}'\left(\boldsymbol{y}(\tilde{\boldsymbol{z}})\right)^+\right) \\ \text{s.t.} \quad & \sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\boldsymbol{y}(\tilde{\boldsymbol{z}})^-\right) \leq (\boldsymbol{I} - \mathbf{diag}(\boldsymbol{\beta}))\boldsymbol{\mu} \\ & \boldsymbol{D}\boldsymbol{y}(\tilde{\boldsymbol{z}}) - \boldsymbol{x}(\tilde{\boldsymbol{z}}) = -\tilde{\boldsymbol{z}} \\ & \mathbf{0} \leq \boldsymbol{x}(\tilde{\boldsymbol{z}}) \leq \boldsymbol{x}^{MAX} \\ & x_t \in \mathcal{L}(1, T, [t-1]) \qquad \forall t \in [T] \\ & y_t \in \mathcal{L}(1, T, [t]) \qquad \forall t \in [T]. \end{aligned} \tag{5.1}$$

To cast the problem into our linearized framework, we introduce auxilliary LDRs $\boldsymbol{s}, \boldsymbol{r}, \boldsymbol{u}, \boldsymbol{w}$, to obtain

$$
\begin{array}{rlll}
{Z_{LDR}}^{(2)} = & \min\limits_{\boldsymbol{x}(\cdot),\boldsymbol{y}(\cdot)} & \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\boldsymbol{c}'\boldsymbol{x}(\tilde{\boldsymbol{z}}) + \boldsymbol{h}'\boldsymbol{r}(\tilde{\boldsymbol{z}})\right) & \\
& \text{s.t.} & \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\boldsymbol{s}(\tilde{\boldsymbol{z}})\right) \leq (\boldsymbol{I} - \mathbf{diag}\,(\boldsymbol{\beta}))\boldsymbol{\mu} & \\
& & \boldsymbol{D}\boldsymbol{y}\left(\tilde{\boldsymbol{z}}\right) - \boldsymbol{x}\left(\tilde{\boldsymbol{z}}\right) = -\tilde{\boldsymbol{z}} & \\
& & \boldsymbol{r}\left(\tilde{\boldsymbol{z}}\right) - \boldsymbol{y}\left(\tilde{\boldsymbol{z}}\right) = \boldsymbol{u}\left(\tilde{\boldsymbol{z}}\right) & \\
& & \boldsymbol{s}\left(\tilde{\boldsymbol{z}}\right) + \boldsymbol{y}\left(\tilde{\boldsymbol{z}}\right) = \boldsymbol{w}\left(\tilde{\boldsymbol{z}}\right) & \\
& & \mathbf{0} \leq \boldsymbol{x}\left(\tilde{\boldsymbol{z}}\right) \leq \boldsymbol{x}^{MAX} & \\
& & \boldsymbol{r}\left(\tilde{\boldsymbol{z}}\right), \boldsymbol{s}\left(\tilde{\boldsymbol{z}}\right), \boldsymbol{u}\left(\tilde{\boldsymbol{z}}\right), \boldsymbol{w}\left(\tilde{\boldsymbol{z}}\right) \geq \mathbf{0} & \\
& & x_t \in \mathcal{L}(1, T, [t-1]) & \forall t \in [T] \\
& & r_t, s_t, u_t, w_t, y_t \in \mathcal{L}(1, T, [t]) & \forall t \in [T]\,.
\end{array}
\tag{5.2}
$$

5.1.3 Numerical Study

We can either model the more verbose problem (5.2) or the more compact problem (5.1) in ROME. Since problem (5.2) is a linearized approximation of problem (5.1), we expect $Z_{LDR} \leq {Z_{LDR}}^{(2)}$. However, since problem (5.2) adheres to the distributional robust optimization framework of Goh and Sim [18], we can use the BDLDR to obtain a better solution, with an improved objective $Z_{BDLDR} \leq Z_{LDR} \leq {Z_{LDR}}^{(2)}$. Appendix A contains the ROME code used to model problem (5.1) using the LDR, and problem (5.2) using both the LDR and BDLDR as decision rules.

In the code in Appendix A, we have defined the numerical values of the various model parameters as a group at the top of the program (lines 13 – 23). Users with a working ROME installation can easily change these values and run the code to observe how the objectives of each problem varies as the model parameters change. For completeness, we will nonetheless tabulate values of the objectives for some values of the model parameters to illustrate the sensitivities to certain model parameters.

We use the following parameters for the numerical examples:

- Number of periods, $T = 10$.
- Order cost rate in each period, $c_t = 1,\ \forall t \in [T]$.
- We use the same holding cost rate in all periods, i.e. $h_t = h,\ \forall t \in [T]$. We vary h within each numerical experiment.
- We use the same mean demand in all periods, denoted by $\mu_t = \mu,\ \forall t \in [T]$ is known. We vary μ within each numerical experiment.
- We use an autoregressive model for demand, similar to Johnson and Thompson [22] and Veinott [35], to capture inter-temporal demand correlation. Specifically, we consider a simplified one-parameter demand model, where all the autoregressive parameters are equal to a parameter $\alpha \in [0, 1]$. This yields a demand covariance matrix with the following structure:

$$\boldsymbol{\Sigma} = \sigma \boldsymbol{L}(\alpha)\boldsymbol{L}(\alpha)',$$

where the lower triangular matrix $\boldsymbol{L}(\alpha) \in \Re^{T \times T}$ has the lower triangular structure:

$$\boldsymbol{L}(\alpha) = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \alpha & 1 & 0 & \dots & 0 \\ \alpha & \alpha & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha & \alpha & \alpha & \dots & 1 \end{bmatrix}.$$

We vary α within our numerical experiments, while σ is a scaling factor, which we fix for all experiments at a value of $\sigma = 20$.

- We assume that the a demand has support on the hypercube $[0, z^{MAX}]^T$. We vary z^{MAX} between experiments.
- We use the same fill rate parameter in each period, i.e. $\beta_t = \beta, \ \forall t \in [T]$. We vary β between experiments.

We present the numerical results in Appendix B, Tables B.1 through B.4. We notice that solving problem (5.2) using basic LDRs (i.e. Z_{LDR}) is very much poorer than using the robust bounds on $\sup_{\mathbb{P} \in \mathbb{F}} \mathrm{E}_{\mathbb{P}} \left((\cdot)^+ \right)$ to directly solve problem (5.1), with objective ${Z_{LDR}}^{(2)}$, to the extent that when $z^{MAX} = 200$, problem (5.2) becomes infeasible, even with the relatively weak fill rate constraint $\beta = 0.5$. However, if we instead use the more complex BDLDR to solve problem (5.2), we get Z_{BDLDR} which is uniformly less than ${Z_{LDR}}^{(2)}$. The extent of the improvement increases when z^{MAX} increases.

5.2 Robust Portfolio Selection

5.2.1 Problem Description

In this section, we consider the problem of robust portfolio selection. Markowitz [25, 26] pioneered the use of optimization to handle the tradeoff between risk and return in portfolio selection, and to solve this problem, introduced the (now classical) technique of minimizing portfolio variance, subject to the mean portfolio return attaining the target. However, various authors have indicated several problems with using variance as a measure of risk, namely that this approach is only appropriate if the returns distribution is elliptically symmetric (Tobin [34] and Chamberlain [8]).

A risk metric which overcomes many of the shortcomings of variance is the Conditional Value-at-Risk (CVaR) risk metric, popularized by Rockafellar and Uryasev [27]. CVaR satisfies many desirable properties of a risk measure, qualifying it as a *coherent* measure of risk (Artzner et al. [1]). CVaR is a measure of tail risk, measuring the expected loss within a specified quantile. We denote this quantile (also termed the CVaR-level) by β, and take β as an exogenously specified parameter in our model.

In this example, we will use CVaR as our objective and optimize our portfolio subject to some standard portfolio constraints. In our model, we make the standard assumption that the assets are traded in a frictionless market (i.e. with no transaction costs and no illiquidity).

Notation: We let N denote the total number of assets which available for investment. Our decision is a vector $\boldsymbol{x} \in \Re^N$, with the i^{th} component representing the fraction of our net wealth to be invested in asset i. The drivers of uncertainty in this model are the asset returns, which we denote by the uncertainty vector $\tilde{\boldsymbol{r}} \in \Re^N$. We do not assume the precise distribution of $\tilde{\boldsymbol{r}} \in \Re^N$, but instead, we assume that we have accurate estimates of the asset return means $\tilde{\boldsymbol{\mu}}$ and covariance matrix $\boldsymbol{\Sigma}$, which characterizes a family of distributions $\mathbb{F}$. We aim to find a portfolio allocation which minimizes its CVaR, subject to attaining an exogenously specified mean target return τ.

5.2.2 Problem Model

Adapting the definition of CVaR in [27] to our robust scenario, we use CVaR_β as our objective:

$$\text{CVaR}_\beta(\boldsymbol{x}) \triangleq \min_{v \in \Re} \left\{ v + \frac{1}{1-\beta} \sup_{\mathbb{P} \in \mathbb{F}} \text{E}_{\mathbb{P}} \left(\left(-\tilde{\boldsymbol{r}}' \boldsymbol{x} - v \right)^+ \right) \right\}. \tag{5.3}$$

Putting this together with the standard portfolio constraints, we have the model:

$$\begin{array}{rll} Z_{PORT} = & \min\limits_{\boldsymbol{x}, v} & v + \dfrac{1}{1-\beta} \sup\limits_{\mathbb{P} \in \mathbb{F}} \text{E}_{\mathbb{P}} \left(\left(-\tilde{\boldsymbol{r}}' \boldsymbol{x} - v \right)^+ \right) \\ & \text{s.t.} & \boldsymbol{\mu}' \boldsymbol{x} \geq \tau \\ & & \boldsymbol{e}' \boldsymbol{x} = 1 \\ & & \boldsymbol{x} \geq \boldsymbol{0}. \end{array} \tag{5.4}$$

We compare three methods of solving problem the portfolio optimization problem (5.4). In all of the methods which we study, we presume to have asset returns data structured into two groups: an in-sample period, and an out-of-sample period. We take the in-sample data as returns data that has already been revealed to the modeler, and hence can be used in the modeling and portfolio construction. Conversely, we take the out-of-sample data as model uncertainties, unknown to the modeler at the point of constructing the portfolio.

We begin with a sampling approach as a benchmark, where we use historical samples to approximate the expectation term in the objective of problem (5.4). Denoting the number of trading days in the in-sample period as T, and the realized in-sample returns as $\left\{ \boldsymbol{r}^t \right\}_{t=1}^T$, the explicit model can be written as

$$\begin{array}{rll} {Z_{PORT}}^{(1)} = & \min\limits_{\boldsymbol{x}, v} & v + \dfrac{1}{(1-\beta)T} \sum\limits_{t=1}^{T} \left(-\boldsymbol{r}^{t\prime} \boldsymbol{x} - v \right)^+ \\ & \text{s.t.} & \boldsymbol{\mu}' \boldsymbol{x} \geq \tau \\ & & \boldsymbol{e}' \boldsymbol{x} = 1 \\ & & \boldsymbol{x} \geq \boldsymbol{0}. \end{array} \tag{5.5}$$

Our second approach is to linearize problem (5.4) into the form of the model (2.6). To this end, we introduce a scalar-valued LDR auxilliary variable $y(\tilde{\boldsymbol{r}})$ to linearize the objective and fit this into our

framework. We obtain the transformed model:

$$\begin{aligned} {Z_{PORT}}^{(2)} = \quad \min_{\boldsymbol{x},v} \quad & v + \frac{1}{1-\beta} \sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(y(\tilde{\boldsymbol{r}})\right) \\ \text{s.t.} \quad & \boldsymbol{\mu}'\boldsymbol{x} \geq \tau \\ & \boldsymbol{e}'\boldsymbol{x} = 1 \\ & \boldsymbol{x} \geq \boldsymbol{0} \\ & y(\tilde{\boldsymbol{r}}) \geq -\tilde{\boldsymbol{r}}'\boldsymbol{x} - v \\ & y(\tilde{\boldsymbol{r}}) \geq 0 \\ & y \in \mathcal{L}(1, N, [N]). \end{aligned} \tag{5.6}$$

The family of distributions $\mathbb{F}$ is defined by the mean and covariance of the uncertainties, estimated from the sample mean and covariance during the in-sample period, i.e.

$$\mathbb{F} = \left\{ \mathbb{P} : \mathrm{E}_{\mathbb{P}}\left(\tilde{\boldsymbol{r}}\right) = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{r}^t, \mathrm{E}_{\mathbb{P}}\left(\tilde{\boldsymbol{r}}\tilde{\boldsymbol{r}}'\right) = \hat{\boldsymbol{\Gamma}} : \hat{\Gamma}_{ij} = \frac{1}{T-1}\sum_{t=1}^{T}\left(r_i^t r_j^t\right) \right\}. \tag{5.7}$$

Our final method is in all respects identical to the second method, with the sole exception that we partition the uncertainties into positive and negative half-spaces, and use the segregated uncertainties and moments instead. Denoting the segregated uncertainties as $\tilde{\boldsymbol{s}}$, this can be explicitly written as:

$$\tilde{\boldsymbol{s}} = \begin{bmatrix} \tilde{\boldsymbol{r}}^+ \\ \tilde{\boldsymbol{r}}^- \end{bmatrix}. \tag{5.8}$$

Consequently, we can reconstitute $\tilde{\boldsymbol{r}}$ from $\tilde{\boldsymbol{s}}$ as $\tilde{\boldsymbol{r}} = \begin{bmatrix} \boldsymbol{I} & -\boldsymbol{I} \end{bmatrix} \tilde{\boldsymbol{s}}$. Using segregated uncertainties, Problem (5.4) can be explicitly written as:

$$\begin{aligned} {Z_{PORT}}^{(3)} = \quad \min_{\boldsymbol{x},v} \quad & v + \frac{1}{1-\beta} \sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(y(\tilde{\boldsymbol{s}})\right) \\ \text{s.t.} \quad & \boldsymbol{\mu}'\boldsymbol{x} \geq \tau \\ & \boldsymbol{e}'\boldsymbol{x} = 1 \\ & \boldsymbol{x} \geq \boldsymbol{0} \\ & y(\tilde{\boldsymbol{s}}) \geq -\left(\begin{bmatrix} \boldsymbol{I} & -\boldsymbol{I} \end{bmatrix} \tilde{\boldsymbol{s}}\right)' \boldsymbol{x} - v \\ & y(\tilde{\boldsymbol{s}}) \geq 0 \\ & y \in \mathcal{L}(1, 2N, [2N]), \end{aligned} \tag{5.9}$$

with the corresponding family of uncertainties:

$$\mathbb{F} = \left\{ \mathbb{P} : \mathbb{P}(\tilde{\boldsymbol{s}} \geq \boldsymbol{0}) = 1, \mathrm{E}_{\mathbb{P}}\left(\tilde{\boldsymbol{s}}\right) = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{s}^t, \mathrm{E}_{\mathbb{P}}\left(\tilde{\boldsymbol{s}}\tilde{\boldsymbol{s}}'\right) = \hat{\boldsymbol{\Gamma}} : \hat{\Gamma}_{ij} = \frac{1}{T-1}\sum_{t=1}^{T}\left(s_i^t s_j^t\right) \right\}, \tag{5.10}$$

where the derived in-sample data $\left\{\boldsymbol{s}^t\right\}_{t=1}^{T}$ is given by

$$\boldsymbol{s}^t = \begin{bmatrix} \left(\boldsymbol{r}^t\right)^+ \\ \left(\boldsymbol{r}^t\right)^- \end{bmatrix} \quad \forall t \in [T]. \tag{5.11}$$

The last two methods fit exactly into our framework, while the first (sampling) method is effectively a deterministic optimization problem. All three methods can be modeled using ROME, and the modeling code is presented in Appendix C.

5.2.3 Numerical Study

In our first numerical study, we use historical daily returns data of stocks traded on the NYSE and the AMEX from 1962 to 2006. We carry out 40 independent experiments on this data. In each experiment, we use a sampling period of 5 years, and use a year immediately following the sampling period as our testing period. Using the standard random sample selection procedure as in Chan et al. [9] and Jagannathan and Ma [21], we randomly select 100 stocks in each experiment (i.e. $N = 100$), with the criteria that the stocks had prices greater than 5 dollars and market capitalization more than the 80th percentile of the size distribution of NYSE firms. Furthermore, a stock would only be selected if it had 6 years of subsequent daily returns from the year which it was selected, so that each stock has full sample and testing information. The out-of sample empirical CVaR values are respectively denoted $\text{CVaR}^{(k)}\ \forall k \in \{1, 2, 3\}$ for each method, and these values are tabulated for CVaR levels $\beta = 95\%$, 98%, and 99%. The tables are presented in Tables D.1 to D.3 of Appendix D.

We observe that on average, using moments and partitioned moments (methods 2 and 3) achieve lower out-of-sample empirical CVaR values than the sampling method (method 1). We perform a single-tailed Student's t-test to analyze the results, with the following null and alternate hypotheses:

$$\begin{aligned} H_0 :&\quad \text{CVaR}^{(k)} = \text{CVaR}^{(1)}, \\ H_1 :&\quad \text{CVaR}^{(k)} < \text{CVaR}^{(1)}, \end{aligned} \tag{5.12}$$

for $k \in \{2, 3\}$, with corresponding p-values $\{p_k\}_{k=2}^{3}$, which are tabulated in Table 5.1.

β	95%		98%		99%	
p_2	1.7992×10^{-4}	**	2.0947×10^{-6}	**	3.2304×10^{-6}	**
p_3	0.3589		0.0114	*	4.4746×10^{-4}	**

Table 5.1: Comparison of portfolios constructed using different methods for the 100 stock dataset. A single asterisk (*) indicates significance at the 5% level, while a double asterisk (**) indicates significance at the 1% level.

The results indicate that in this dataset, the method of using moments or segregated moments to construct the portfolo tends to produce portfolios with better out-of-sample CVaR characteristics. An intuitive explanation is that any non-stationarity in the returns distribution is exacerbated by the sampling method, which tends to overfit the data with historical returns. However, we notice that while the segregated moments method does appear to improve upon the sampling method, the improvement is not as significant as the improvement by using ordinary moments. In particular, for $\beta = 95\%$, there is insufficient empirical evidence at the 5% level to reject the null hypothesis. One possible reason is that segregated moments are adept at capturing asymmetry in the assets. For a portfolio constructed from a large number of assets, the skew might be less significant, and other factors such as errors in moment-estimation gets pronounced, consequently degrading the result.

To investigate this phenomenon, we study our second dataset which is contains fewer assets. This dataset is taken from the data library in Kenneth French's website[4], and consists of daily returns of 10

[4] `http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html`

industry portfolios formed from securities on the NYSE, AMEX, and NASDAQ. We perform the same analysis as before, with an in-sample period of 5 years, and a testing period of 1 year. We used data from 1966 to 2006 for our test. The out-of-sample CVaR values are tabulated in Appendix D, Tables D.4 to D.6. Again, we perform the same t-tests, and summarize the analysis in Table 5.2

β	95%		98%		99%	
p_2	0.0050	**	0.0189	*	0.0067	**
p_3	0.0084	**	0.0073	**	0.0012	**

Table 5.2: Comparison of portfolios constructed using different methods for the 10 industry portfolio dataset. A single asterisk (*) indicates significance at the 5% level, while a double asterisk (**) indicates significance at the 1% level.

From the second experiment, we observe that both moment and segregated moment methods, on average, exhibit significant improvement over the sampling method.

6 Conclusion

In this paper, we have presented ROME and its utility in modeling robust optimization problems. We have briefly introduced the syntax and structure of a ROME program, and have discussed several key features of ROME which renders it suitable for modeling and solving robust optimization problems, especially problems cast in the DRO framework presented in our earlier work [18]. Finally, we have used ROME to model robust versions two typical problems in operations management and finance - an inventory management problem and a portfolio selection problem. Through the examples, we have demonstrated how ROME can model otherwise complex problems with relative ease, and in a mathematically intuitive manner. As such, we believe that ROME can be a helpful and valuable tool for further academic and industrial research in the field of robust optimization.

References

[1] P. Artzner, F. Delbaen, J.-M. Eber, and D. Heath. Coherent measures of risk. *Mathematical Finance*, 9(3):203–228, 1999.

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

[3] A. Ben-Tal and A. Nemirovski. Robust convex optimization. *Mathematics of Operations Research*, 23(4):769–805, 1998.

[4] D. Bertsimas and M. Sim. The price of robustness. *Operations Research*, 52(1):35–53, 2004.

[5] T. Boyaci and G. Gallego. Serial production/distribution systems under service constraints. *Manufacturing & Service Operations Management*, 3(1):43–50, 2001.

[6] M. Breton and S. El Hachem. Algorithms for the solution of stochastic dynamic minimax problems. *Computational Optimization and Applications*, 4:317–345, 1995.

[7] G. Cachon and C. Terwiesch. *Matching Supply with Demand: An Introduction to Operations Management*. McGraw Hill, 2009.

[8] G. Chamberlain. A characterization of the distributions that imply mean-variance utility functions. *Journal of Economic Theory*, 29(1):185–201, Feb 1983.

[9] L. K. C. Chan, J. Karceski, and J. Lakonishok. On portfolio optimization: Forecasting covariances and choosing the risk model. *Review of Financial Studies*, 12:937–974, 1999.

[10] W. Chen and M. Sim. Goal-driven optimization. *Operations Research*, 57(2)(2):342–357, 2009.

[11] X. Chen, M. Sim, and P. Sun. A robust optimization perspective on stochastic programming. *Operations Research*, 55(6):1058–1071, 2007.

[12] X. Chen, M. Sim, P. Sun, and J. Zhang. A linear decision-based approximation approach to stochastic programming. *Operations Research*, 56(2):344–357, 2008.

[13] X. Chen and Y. Zhang. Uncertain linear programs: Extended affinely adjustable robust counterparts. *Operations Research*, 2009.

[14] E. Delage and Y. Ye. Distributionally robust optimization under moment uncertainty with application to data-driven problems. *Operations Research*, forthcoming.

[15] J. Dupačová. The minimax approach to stochastic programming and an illustrative application. *Stochastics*, 20(1):73–88, January 1987.

[16] J. Dupačová. Stochastic programming: Minimax approach. In C. Floudas and P. Pardalos, editors, *Encyclopedia of Optimization (Vol. 5)*, pages 327–330. Kluwer Academic Publishers, 2001.

[17] M. Dyer and L. Stougie. Computational complexity of stochastic programming problems. *Math. Programming Ser. A*, 106:423–432, 2006.

[18] J. Goh and M. Sim. Distributionally robust optimization and its tractable approximations. *Working Paper*, 2009.

[19] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control (a tribute to M. Vidyasagar)*, pages 95–110. Springer, 2008.

[20] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming (web page and software). `http://stanford.edu/~boyd/cvx`, June 2009.

[21] R. Jagannathan and T. Ma. Risk reduction in large portfolios: why imposing the wrong constraints helps. *Journal of Finance*, 58(4):1651–1683, 2003.

[22] G. D. Johnson and H. E. Thompson. Optimality of myopic inventory policies for certain dependent demand processes. *Management Science*, 21(11):1303–1307, 1975.

[23] J. Löfberg. YALMIP : a toolbox for modeling and optimization in MATLAB. In *IEEE International Symposium on Computer Aided Control Systems Design*, 2004.

[24] J. Löfberg. Modeling and solving uncertain optimization problems in YALMIP. In *Proceedings of the 17th World Congress: The International Federation of Automatic Control*, Seoul, Korea, 2008.

[25] H. M. Markowitz. Portfolio selection. *Journal of Finance*, 7:77–91, 1952.

[26] H. M. Markowitz. *Portfolio selection: Efficient diversification of investments*. John Wiley & Sons, New York, 1959.

[27] R. T. Rockafellar and S. Uryasev. Optimization of conditional value-at-risk. *Journal of Risk*, 2:493–517, 2000.

[28] C.-T. See and M. Sim. Robust approximation to multi-period inventory management. *Operations Research, forthcoming*, 2009.

[29] K. H. Shang and J.-S. Song. A closed-form approximation for serial inventory systems and its application to system design. *Manufacturing & Service Operations Management*, 8(4):394 – 406, September 2006.

[30] A. Shapiro and S. Ahmed. On a class of minimax stochastic programs. *SIAM Journal on Optimization*, 14(4):1237–1249, 2004.

[31] A. Shapiro and A. Kleywegt. Minimax analysis of stochastic programs. *Optimization Methods and Software*, 17(3):523–542, 2002.

[32] A. Shapiro and A. Nemirovski. On complexity of stochastic programming problems. In V. Jeyakumar and A. Rubinov, editors, *Continuous Optimization*, pages 111–146. Springer USA, 2005.

[33] A. L. Soyster. Convex programming with set-inclusive constraints and applications to inexact linear programming. *Operations Research*, 21(5):1154–1157, 1973.

[34] J. Tobin. Liquidity preference as behavior toward risk. *Review of Economic Studies*, 25:65–85, 1958.

[35] A. Veinott. Optimal policy for a multi-product, dynamic, nonstationary inventory problem. *Management Science*, 12(3):206–222, 1965.

[36] J. Žáčková. On minimax solution of stochastic linear programming problems. *Časopis pro Pěstováni Matematiky*, 91:423–430, 1966.

Appendix A ROME Code for Robust Inventory Management

```
% inventory_fillrate_example.m
% Script to model robust fillrate-constrained inventory management.
%
% Solves three problems.
% 1. First models and solves the problem by directly applying robust bounds
%    to the nominal problem, decision rules are still LDRs
%
% 2. Solves a linearized version of the problem using LDRs
%
% 3. Solves a linearized version of the problem using BDLDRs
%

% model parameters
T = 10;                   % planning horizon
c = 1 *ones(T, 1);        % order cost rate
hcost = 2*ones(T, 1);     % holding cost rate
beta  = 0.50*ones(T, 1); % minimum fillrate in each period
xMax  = 100*ones(T, 1);  % maximum order quantity in each period
alpha = 0.5;              % temporal autocorrelation factor
L = alpha * tril(ones(T), -1) + eye(T); % autocorrelation matrix

% numerical uncertainty parameters
zMax  = 105*ones(T, 1);   % maximum demand in each period
zMean = 30*ones(T, 1);    % mean demand in each period
zCovar = 20*(L * L');     % temporal demand covariance

% differencing matrix
D = eye(T) - diag(ones(T-1, 1), -1);

% dependency structure
pX = logical([tril(ones(T)), zeros(T, 1)]);

% Step 1: Direct Method
% ---------------------
h = rome_begin('Robust Inventory (Direct)');
tic;
% declare vector of uncertainties
newvar z(T) uncertain nonneg;

% define uncertainty parameters
rome_constraint(z <= zMax);  % support
z.set_mean(zMean);           % mean
z.Covar = zCovar;            % covariances

% define LDR variables
newvar x(T, z, 'Pattern', pX) linearrule;  % order quantity
```

```
newvar y(T, z) linearrule;                % inventory level

% fillrate constraint
rome_constraint(rome_create_bound(-y) <= diag(ones(T, 1) - beta) * zMean);

% inventory balance constraint
rome_constraint(D*y == x - z);

% order quantity constraints
rome_box(x, 0, xMax);

% objective
rome_minimize(c'*mean(x) + hcost' * rome_create_bound(y));

% solve and display optimal objective
h.solve;
disp(sprintf('Direct Obj = %0.2f, time = %0.2f secs', h.ObjVal, toc));
x_sol_direct = h.eval(x)


% Step 2: LDR Method
% --------------------
h = rome_begin('Robust Inventory (LDR)');
tic;

% declare uncertainties
newvar z(T) uncertain nonneg;

% define uncertainty parameters
rome_constraint(z <= zMax);  % support
z.set_mean(zMean);           % mean
z.Covar = zCovar;            % covariance

% define LDR variables
newvar x(T, z, 'Pattern', pX) linearrule;  % order quantity
newvar y(T, z) linearrule;                 % inventory level

% define auxilliary variables
newvar r(T, z) s(T, z) linearrule nonneg;

% auxilliary constraints
rome_constraint(r >=  y); % since r >= y^+
rome_constraint(s >= -y); % since s >= y^-

% fillrate constraint
rome_constraint(mean(s) <= diag(ones(T, 1) - beta) * zMean);

% inventory balance constraint
```

```
rome_constraint(D*y == x - z);

% order quantity constraints
rome_box(x, 0, xMax);

% objective
rome_minimize(c'*mean(x) + hcost'*mean(r));

% solve and display optimal objective
h.solve;
disp(sprintf('LDR Obj = %0.2f, time = %0.2f secs', h.ObjVal, toc));
x_sol_ldr = h.eval(x)

% Step 3: BDLDR Method
% --------------------
h = rome_begin('Robust Inventory (BDLDR)');
tic;

% declare uncertainties
newvar z(T) uncertain nonneg;

% define uncertainty parameters
rome_constraint(z <= zMax);  % support
z.set_mean(zMean);           % mean
z.Covar = zCovar;            % covariance

% define LDR variables
newvar x(T, z, 'Pattern', pX) linearrule;  % order quantity
newvar y(T, z) linearrule;                 % inventory level

% define auxilliary variables
newvar r(T, z) s(T, z) linearrule nonneg;

% auxilliary constraints
rome_constraint(r >=  y); % since r >= y^+
rome_constraint(s >= -y); % since s >= y^-

% fillrate constraint
rome_constraint(mean(s) <= diag(ones(T, 1) - beta) * zMean);

% inventory balance constraint
rome_constraint(D*y == x - z);

% order quantity constraints
rome_box(x, 0, xMax);

% objective
rome_minimize(c'*mean(x) + hcost'*mean(r));
```

```
143
144 % solve and display optimal objective
145 h.solve_deflected;
146 disp(sprintf('BDLDR Obj = %0.2f, time = %0.2f secs', h.ObjVal, toc));
147 x_sol_bdldr = h.eval(x)
148
149 rome_end;
```

Code Segment 16: ROME Code for fill rate-constrained Robust Inventory Management

Appendix B Numerical Results for Robust Inventory Management

h	μ	α	$Z_{LDR}{}^{(2)}$	Z_{LDR}	Z_{BDLDR}	h	μ	α	$Z_{LDR}{}^{(2)}$	Z_{LDR}	Z_{BDLDR}
1	10	0.00	590.0	106.0	106.0	3	10	0.00	1490.0	126.0	126.0
1	10	0.20	590.0	108.2	108.1	3	10	0.20	1490.0	131.8	131.5
1	10	0.50	590.0	119.9	117.2	3	10	0.50	1490.0	163.0	155.9
1	10	0.90	590.0	153.3	135.8	3	10	0.90	1490.0	251.8	205.4
1	20	0.00	630.0	195.5	195.5	3	20	0.00	1430.0	205.5	205.5
1	20	0.20	630.0	196.6	196.6	3	20	0.20	1430.0	208.4	208.3
1	20	0.50	630.0	202.4	201.9	3	20	0.50	1430.0	223.8	222.6
1	20	0.90	630.0	218.3	214.8	3	20	0.90	1430.0	266.1	256.8
1	30	0.00	670.0	288.7	288.7	3	30	0.00	1370.0	295.3	295.3
1	30	0.20	670.0	289.4	289.4	3	30	0.20	1370.0	297.3	297.2
1	30	0.50	670.0	293.2	293.1	3	30	0.50	1370.0	307.5	307.1
1	30	0.90	670.0	303.6	302.5	3	30	0.90	1370.0	335.3	332.0
1	50	0.00	750.0	477.2	477.2	3	50	0.00	1250.0	481.2	481.2
1	50	0.20	750.0	477.6	477.6	3	50	0.20	1250.0	482.4	482.4
1	50	0.50	750.0	479.9	479.9	3	50	0.50	1250.0	488.5	488.3
1	50	0.90	750.0	486.1	485.8	3	50	0.90	1250.0	504.9	504.1
1	70	0.00	830.0	666.6	666.6	3	70	0.00	1130.0	669.4	669.4
1	70	0.20	830.0	666.9	666.9	3	70	0.20	1130.0	670.3	670.3
1	70	0.50	830.0	668.5	668.5	3	70	0.50	1130.0	674.6	674.6
1	70	0.90	830.0	672.9	672.9	3	70	0.90	1130.0	686.3	686.0
2	10	0.00	1040.0	116.0	116.0	4	10	0.00	1940.0	136.0	136.0
2	10	0.20	1040.0	120.0	119.8	4	10	0.20	1940.0	143.6	143.2
2	10	0.50	1040.0	141.4	136.5	4	10	0.50	1940.0	184.5	175.2
2	10	0.90	1040.0	202.6	170.6	4	10	0.90	1940.0	301.1	240.2
2	20	0.00	1030.0	200.5	200.5	4	20	0.00	1830.0	210.5	210.5
2	20	0.20	1030.0	202.5	202.5	4	20	0.20	1830.0	214.3	214.2
2	20	0.50	1030.0	213.1	212.2	4	20	0.50	1830.0	234.5	232.9
2	20	0.90	1030.0	242.2	235.8	4	20	0.90	1830.0	290.1	277.9
2	30	0.00	1020.0	292.0	292.0	4	30	0.00	1720.0	298.7	298.7
2	30	0.20	1020.0	293.3	293.3	4	30	0.20	1720.0	301.2	301.2
2	30	0.50	1020.0	300.3	300.1	4	30	0.50	1720.0	314.6	314.0
2	30	0.90	1020.0	319.5	317.2	4	30	0.90	1720.0	351.1	346.7
2	50	0.00	1000.0	479.2	479.2	4	50	0.00	1500.0	483.2	483.2
2	50	0.20	1000.0	480.0	480.0	4	50	0.20	1500.0	484.7	484.7
2	50	0.50	1000.0	484.2	484.1	4	50	0.50	1500.0	492.7	492.6
2	50	0.90	1000.0	495.5	495.0	4	50	0.90	1500.0	514.3	513.2
2	70	0.00	980.0	668.0	668.0	4	70	0.00	1280.0	670.9	670.9
2	70	0.20	980.0	668.6	668.6	4	70	0.20	1280.0	671.9	671.9
2	70	0.50	980.0	671.6	671.5	4	70	0.50	1280.0	677.6	677.6
2	70	0.90	980.0	679.6	679.5	4	70	0.90	1280.0	693.0	692.6

Table B.1: Numerical Results for fill rate-constrained Inventory Management with $(z^{MAX}, \beta) = (100, 0.5)$

h	μ	α	${Z_{LDR}}^{(2)}$	Z_{LDR}	Z_{BDLDR}	h	μ	α	${Z_{LDR}}^{(2)}$	Z_{LDR}	Z_{BDLDR}
1	10	0.00	$+\infty$	121.0	107.0	3	10	0.00	$+\infty$	166.1	128.0
1	10	0.20	$+\infty$	167.9	109.3	3	10	0.20	$+\infty$	285.9	133.8
1	10	0.50	$+\infty$	313.6	119.4	3	10	0.50	$+\infty$	654.1	159.9
1	10	0.90	$+\infty$	722.3	141.6	3	10	0.90	$+\infty$	1668.7	216.7
1	20	0.00	$+\infty$	203.9	196.1	3	20	0.00	$+\infty$	227.8	206.5
1	20	0.20	$+\infty$	229.0	197.4	3	20	0.20	$+\infty$	292.0	209.5
1	20	0.50	$+\infty$	304.0	203.4	3	20	0.50	$+\infty$	482.0	224.7
1	20	0.90	$+\infty$	490.5	218.6	3	20	0.90	$+\infty$	949.1	262.7
1	30	0.00	$+\infty$	294.9	289.3	3	30	0.00	$+\infty$	312.0	296.1
1	30	0.20	$+\infty$	313.0	290.2	3	30	0.20	$+\infty$	358.3	298.2
1	30	0.50	$+\infty$	366.6	294.6	3	30	0.50	$+\infty$	493.7	308.9
1	30	0.90	$+\infty$	495.2	306.0	3	30	0.90	$+\infty$	816.5	336.7
1	50	0.00	$+\infty$	482.0	477.9	3	50	0.00	$+\infty$	493.9	482.1
1	50	0.20	$+\infty$	495.0	478.7	3	50	0.20	$+\infty$	527.2	483.5
1	50	0.50	$+\infty$	533.1	481.8	3	50	0.50	$+\infty$	623.4	490.4
1	50	0.90	$+\infty$	622.5	490.1	3	50	0.90	$+\infty$	847.6	509.0
1	70	0.00	$+\infty$	671.0	667.5	3	70	0.00	$+\infty$	681.3	670.6
1	70	0.20	$+\infty$	682.6	668.5	3	70	0.20	$+\infty$	710.7	672.1
1	70	0.50	$+\infty$	716.1	671.6	3	70	0.50	$+\infty$	795.3	677.8
1	70	0.90	$+\infty$	794.2	679.7	3	70	0.90	$+\infty$	990.7	693.5
2	10	0.00	$+\infty$	143.6	117.5	4	10	0.00	$+\infty$	188.6	138.5
2	10	0.20	$+\infty$	226.9	121.5	4	10	0.20	$+\infty$	344.9	146.0
2	10	0.50	$+\infty$	483.8	139.7	4	10	0.50	$+\infty$	824.3	180.2
2	10	0.90	$+\infty$	1195.5	179.2	4	10	0.90	$+\infty$	2141.9	254.2
2	20	0.00	$+\infty$	215.8	201.3	4	20	0.00	$+\infty$	239.8	211.6
2	20	0.20	$+\infty$	260.5	203.5	4	20	0.20	$+\infty$	323.5	215.5
2	20	0.50	$+\infty$	393.0	214.1	4	20	0.50	$+\infty$	571.0	235.3
2	20	0.90	$+\infty$	719.8	240.6	4	20	0.90	$+\infty$	1178.3	284.7
2	30	0.00	$+\infty$	303.4	292.7	4	30	0.00	$+\infty$	320.5	299.6
2	30	0.20	$+\infty$	335.7	294.2	4	30	0.20	$+\infty$	381.0	302.2
2	30	0.50	$+\infty$	430.1	301.7	4	30	0.50	$+\infty$	557.3	316.0
2	30	0.90	$+\infty$	655.8	321.3	4	30	0.90	$+\infty$	977.1	352.0
2	50	0.00	$+\infty$	487.9	480.0	4	50	0.00	$+\infty$	499.9	484.1
2	50	0.20	$+\infty$	511.1	481.1	4	50	0.20	$+\infty$	543.3	485.9
2	50	0.50	$+\infty$	578.3	486.1	4	50	0.50	$+\infty$	668.6	494.7
2	50	0.90	$+\infty$	735.1	499.6	4	50	0.90	$+\infty$	960.2	518.4
2	70	0.00	$+\infty$	676.1	669.1	4	70	0.00	$+\infty$	686.4	672.2
2	70	0.20	$+\infty$	696.6	670.3	4	70	0.20	$+\infty$	724.7	673.8
2	70	0.50	$+\infty$	755.7	674.7	4	70	0.50	$+\infty$	834.9	680.9
2	70	0.90	$+\infty$	892.4	686.6	4	70	0.90	$+\infty$	1088.9	700.3

Table B.2: Numerical Results for fill rate-constrained Inventory Management with $(z^{MAX}, \beta) = (200, 0.5)$

h	μ	α	$Z_{LDR}{}^{(2)}$	Z_{LDR}	Z_{BDLDR}	h	μ	α	$Z_{LDR}{}^{(2)}$	Z_{LDR}	Z_{BDLDR}
1	10	0.00	890.0	125.5	125.5	3	10	0.00	2330.0	175.5	175.5
1	10	0.20	890.0	131.0	130.3	3	10	0.20	2330.0	190.2	188.3
1	10	0.50	890.0	161.8	149.0	3	10	0.50	2330.0	272.1	238.0
1	10	0.90	890.0	264.6	193.2	3	10	0.90	2330.0	542.5	352.9
1	20	0.00	900.0	209.8	209.8	3	20	0.00	2180.0	234.8	234.8
1	20	0.20	900.0	212.5	212.3	3	20	0.20	2180.0	242.0	241.5
1	20	0.50	900.0	227.3	223.3	3	20	0.50	2180.0	281.5	270.6
1	20	0.90	900.0	271.1	248.6	3	20	0.90	2180.0	397.7	336.6
1	30	0.00	910.0	303.2	303.2	3	30	0.00	2030.0	319.8	319.8
1	30	0.20	910.0	305.0	304.9	3	30	0.20	2030.0	324.7	324.4
1	30	0.50	910.0	314.8	312.6	3	30	0.50	2030.0	350.7	344.5
1	30	0.90	910.0	342.8	330.8	3	30	0.90	2030.0	425.2	391.3
1	50	0.00	930.0	495.5	495.5	3	50	0.00	1730.0	505.5	505.5
1	50	0.20	930.0	496.6	496.6	3	50	0.20	1730.0	508.4	508.3
1	50	0.50	930.0	502.4	501.5	3	50	0.50	1730.0	523.9	520.8
1	50	0.90	930.0	518.7	513.5	3	50	0.90	1730.0	567.4	550.7
1	70	0.00	950.0	689.9	689.9	3	70	0.00	1430.0	697.1	697.1
1	70	0.20	950.0	690.7	690.7	3	70	0.20	1430.0	699.1	699.1
1	70	0.50	950.0	694.9	694.5	3	70	0.50	1430.0	710.2	708.6
1	70	0.90	950.0	706.4	704.4	3	70	0.90	1430.0	740.9	731.9
2	10	0.00	1610.0	150.5	150.5	4	10	0.00	3050.0	200.5	200.5
2	10	0.20	1610.0	160.6	159.3	4	10	0.20	3050.0	219.7	217.2
2	10	0.50	1610.0	217.0	193.5	4	10	0.50	3050.0	327.3	282.5
2	10	0.90	1610.0	403.6	273.1	4	10	0.90	3050.0	681.4	432.8
2	20	0.00	1540.0	222.3	222.3	4	20	0.00	2820.0	247.3	247.3
2	20	0.20	1540.0	227.2	226.9	4	20	0.20	2820.0	256.8	256.1
2	20	0.50	1540.0	254.4	247.0	4	20	0.50	2820.0	308.6	294.2
2	20	0.90	1540.0	334.4	292.7	4	20	0.90	2820.0	461.0	380.6
2	30	0.00	1470.0	311.5	311.5	4	30	0.00	2590.0	328.2	328.2
2	30	0.20	1470.0	314.8	314.6	4	30	0.20	2590.0	334.5	334.1
2	30	0.50	1470.0	332.7	328.6	4	30	0.50	2590.0	368.7	360.5
2	30	0.90	1470.0	384.0	361.1	4	30	0.90	2590.0	466.4	421.6
2	50	0.00	1330.0	500.5	500.5	4	50	0.00	2130.0	510.5	510.5
2	50	0.20	1330.0	502.5	502.4	4	50	0.20	2130.0	514.3	514.1
2	50	0.50	1330.0	513.2	511.2	4	50	0.50	2130.0	534.7	530.5
2	50	0.90	1330.0	543.0	532.2	4	50	0.90	2130.0	591.7	569.2
2	70	0.00	1190.0	693.5	693.5	4	70	0.00	1670.0	700.6	700.6
2	70	0.20	1190.0	694.9	694.9	4	70	0.20	1670.0	703.4	703.3
2	70	0.50	1190.0	702.5	701.6	4	70	0.50	1670.0	717.9	715.5
2	70	0.90	1190.0	723.7	718.2	4	70	0.90	1670.0	758.2	745.4

Table B.3: Numerical Results for Fillrate-constrained Inventory Management with $(z^{MAX}, \beta) = (100, 0.8)$

h	μ	α	$Z_{LDR}{}^{(2)}$	Z_{LDR}	Z_{BDLDR}	h	μ	α	$Z_{LDR}{}^{(2)}$	Z_{LDR}	Z_{BDLDR}
1	10	0.00	$+\infty$	163.4	129.5	3	10	0.00	$+\infty$	276.6	185.8
1	10	0.20	$+\infty$	285.3	136.0	3	10	0.20	$+\infty$	588.2	202.4
1	10	0.50	$+\infty$	722.1	161.5	3	10	0.50	$+\infty$	1681.6	267.8
1	10	0.90	$+\infty$	3402.0	2563.5	3	10	0.90	$+\infty$	9192.7	7019.9
1	20	0.00	$+\infty$	230.7	211.3	3	20	0.00	$+\infty$	290.7	238.0
1	20	0.20	$+\infty$	294.9	214.3	3	20	0.20	$+\infty$	454.8	245.7
1	20	0.50	$+\infty$	500.3	227.0	3	20	0.50	$+\infty$	972.2	278.3
1	20	0.90	$+\infty$	1171.0	265.8	3	20	0.90	$+\infty$	2617.0	374.0
1	30	0.00	$+\infty$	318.8	304.2	3	30	0.00	$+\infty$	361.6	321.8
1	30	0.20	$+\infty$	365.1	306.3	3	30	0.20	$+\infty$	479.9	326.9
1	30	0.50	$+\infty$	508.2	315.0	3	30	0.50	$+\infty$	840.8	348.9
1	30	0.90	$+\infty$	911.0	337.4	3	30	0.90	$+\infty$	1838.2	404.0
1	50	0.00	$+\infty$	507.6	496.5	3	50	0.00	$+\infty$	537.6	507.0
1	50	0.20	$+\infty$	541.0	497.9	3	50	0.20	$+\infty$	623.2	510.2
1	50	0.50	$+\infty$	641.8	503.8	3	50	0.50	$+\infty$	877.1	524.0
1	50	0.90	$+\infty$	904.6	518.7	3	50	0.90	$+\infty$	1530.7	558.4
1	70	0.00	$+\infty$	701.4	691.2	3	70	0.00	$+\infty$	727.6	698.9
1	70	0.20	$+\infty$	731.3	692.7	3	70	0.20	$+\infty$	803.8	701.6
1	70	0.50	$+\infty$	820.4	698.0	3	70	0.50	$+\infty$	1028.1	712.8
1	70	0.90	$+\infty$	1048.9	712.3	3	70	0.90	$+\infty$	1596.0	742.2
2	10	0.00	$+\infty$	220.0	157.7	4	10	0.00	$+\infty$	333.2	213.9
2	10	0.20	$+\infty$	436.7	169.2	4	10	0.20	$+\infty$	739.6	235.6
2	10	0.50	$+\infty$	1201.9	214.7	4	10	0.50	$+\infty$	2161.2	320.9
2	10	0.90	$+\infty$	6300.0	4792.3	4	10	0.90	$+\infty$	12083.3	9209.7
2	20	0.00	$+\infty$	260.7	224.7	4	20	0.00	$+\infty$	320.7	251.4
2	20	0.20	$+\infty$	374.9	230.0	4	20	0.20	$+\infty$	534.8	261.5
2	20	0.50	$+\infty$	736.3	252.6	4	20	0.50	$+\infty$	1208.2	303.9
2	20	0.90	$+\infty$	1894.1	319.9	4	20	0.90	$+\infty$	3339.9	428.0
2	30	0.00	$+\infty$	340.2	313.0	4	30	0.00	$+\infty$	383.0	330.6
2	30	0.20	$+\infty$	422.5	316.6	4	30	0.20	$+\infty$	537.3	337.2
2	30	0.50	$+\infty$	674.5	332.0	4	30	0.50	$+\infty$	1007.1	365.8
2	30	0.90	$+\infty$	1374.6	370.7	4	30	0.90	$+\infty$	2301.9	437.3
2	50	0.00	$+\infty$	522.6	501.8	4	50	0.00	$+\infty$	552.7	512.2
2	50	0.20	$+\infty$	582.1	504.1	4	50	0.20	$+\infty$	664.2	516.3
2	50	0.50	$+\infty$	759.5	513.9	4	50	0.50	$+\infty$	994.8	534.1
2	50	0.90	$+\infty$	1217.7	538.6	4	50	0.90	$+\infty$	1843.8	578.3
2	70	0.00	$+\infty$	714.5	695.1	4	70	0.00	$+\infty$	740.7	702.7
2	70	0.20	$+\infty$	767.5	697.2	4	70	0.20	$+\infty$	840.0	706.0
2	70	0.50	$+\infty$	924.3	705.4	4	70	0.50	$+\infty$	1132.0	720.1
2	70	0.90	$+\infty$	1322.5	727.3	4	70	0.90	$+\infty$	1869.5	757.0

Table B.4: Numerical Results for fill rate-constrained Inventory Management with $(z^{MAX}, \beta) = (200, 0.8)$

Appendix C ROME Code for Portfolio Selection

```
% PORTFOLIO_FUNCTION.m
%
% Solves a portfolio problem with CVaR objective using
%
% 1. Sampling Method
% 2. Moments Method
% 3. Segregated Moments Method
%

function [x_sol, objval] = portfolio_function(cur_data, beta)

% extract moments and parameters from historical data
mean_ret  = mean(cur_data)';
covar_ret = cov(cur_data);
tau   = mean(mean_ret);
N = length(mean_ret);
T = size(cur_data, 1);

% partitioned statisitcs
part_data = [pos(cur_data), neg(cur_data)];
part_mean = mean(part_data)';
part_covar = cov(part_data);

% output arrays
x_sol = zeros(N, 3);
objval = zeros(1, 3);

%%%%%%%%%%%%%%%%%%%%%%
% 1. Sampling Method %
%%%%%%%%%%%%%%%%%%%%%%
h = rome_begin('Portfolio Example (Sampling)'); % make model

% Decisions
newvar v;           % cvar aux variable
newvar x(N) nonneg; % asset weights
newvar y(T) nonneg; % aux variable

% Objective
rome_minimize(v + (1./((1-beta)*T)) * sum(y));

% Constraints
rome_constraint(mean_ret' * x >= tau); % Target Mean Return
rome_constraint(sum(x) == 1);          % Sum of Weights
rome_constraint(y >= -cur_data*x - v); % Auxilliary constraint

% Completing the Model
```

```
h.solve;                        % solve
x_sol(:, 1) = h.eval_var(x); % get portfolio weights
objval(1)   = h.ObjVal;      % get objective value
rome_end;                       % clear memory

%%%%%%%%%%%%%%%%%%%%
% 2. Moments Method %
%%%%%%%%%%%%%%%%%%%%
h = rome_begin('Portfolio Example (Moments)');  % make model

% Uncertainties
newvar r(N) uncertain;  % represents uncertain returns
r.set_mean(mean_ret);   % specify mean
r.Covar = covar_ret;    % specify covariance

% Decisions
newvar v;               % cvar aux variable
newvar x(N) nonneg;     % asset weights
newvar y(1, r) linearrule nonneg;  % auxilliary variable

% Objective
rome_minimize(v + (1./(1-beta)) * mean(y));

% Constraints
rome_constraint(mean_ret' * x >= tau); % Target Mean Return
rome_constraint(sum(x) == 1);          % Sum of Weights
rome_constraint(y >= -r'*x - v); % Auxilliary constraint

% Complete
h.solve_deflected;           % solve
x_sol(:, 2) = h.eval_var(x); % get portfolio weights
objval(2) = h.ObjVal;        % get objective value
rome_end;                    % clear memory

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 3. Partitioned Moments Method %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
h = rome_begin('Portfolio Example (Partitioned)'); % make model

% Uncertainties
newvar s(2*N) uncertain nonneg;  % Uncertainties: partitioned returns
s.set_mean(part_mean);           % Specify partitioned mean
s.Covar = part_covar;            % Specify partitioned covariance

% Actual returns: function of uncertainties
r = [eye(N), -eye(N)] * s;

% Decisions
```

```
newvar v;                % cvar aux variable
newvar x(N) nonneg;      % asset weights
newvar y(1, s) linearrule nonneg;  % auxilliary variable

% Objective
rome_minimize(v + (1./(1-beta)) * mean(y));

% Constraints
rome_constraint(mean_ret' * x >= tau); % Target Mean Return
rome_constraint(sum(x) == 1);          % Sum of Weights
rome_constraint(y >= -r'*x - v); % Auxilliary constraint

% Complete
h.solve_deflected;           % solve
x_sol(:, 3) = h.eval_var(x); % get portfolio weights
objval(3) = h.ObjVal;        % get objective value
rome_end;                    % clear memory
```

Code Segment 17: ROME Code for Robust Portfolio Management

Appendix D Results for Portfolio Selection

No	T	$\text{CVaR}^{(1)}$	$\text{CVaR}^{(2)}$	$\text{CVaR}^{(3)}$
1	1259	0.85	0.77	0.81
2	1259	1.31	1.14	1.17
3	1232	1.42	1.39	1.40
4	1230	2.00	1.97	2.00
5	1232	1.15	1.14	1.12
6	1235	0.94	0.84	0.87
7	1234	1.45	1.51	1.50
8	1262	2.26	2.11	2.25
9	1263	0.90	0.82	0.92
10	1263	0.72	0.67	0.65
11	1262	0.82	0.79	0.78
12	1263	1.62	1.41	1.52
13	1264	2.01	2.05	2.09
14	1265	1.48	1.51	1.46
15	1262	1.54	1.33	1.44
16	1262	1.33	1.16	1.15
17	1265	0.91	0.88	0.88
18	1265	0.88	0.71	0.81
19	1264	1.01	0.95	1.14
20	1263	1.52	1.48	1.49
21	1264	2.86	2.65	2.54
22	1264	0.93	0.78	0.85
23	1263	1.05	0.96	1.06
24	1264	0.98	0.99	0.99
25	1264	0.90	0.96	1.03
26	1265	0.70	0.77	0.85
27	1264	0.80	0.73	0.79
28	1266	1.03	1.00	1.04
29	1265	0.81	0.79	0.76
30	1266	1.07	1.02	1.06
31	1263	0.94	1.01	1.00
32	1263	1.46	1.43	1.53
33	1261	0.78	0.79	0.84
34	1263	1.13	1.03	1.09
35	1262	1.06	1.10	1.13
36	1256	1.26	1.20	1.31
37	1256	0.78	0.77	0.81
38	1257	1.65	1.73	1.96
39	1254	0.73	0.70	0.74
40	1256	0.94	0.90	0.92

Table D.1: Numerical Results for Portfolio Selection for 100 stock dataset with $\beta = 95\%$. CVaR Values are given in basis points

No	T	$CVaR^{(1)}$	$CVaR^{(2)}$	$CVaR^{(3)}$
1	1259	1.22	0.83	0.93
2	1259	1.95	1.42	1.57
3	1232	1.82	1.64	1.67
4	1230	3.02	2.67	2.61
5	1232	1.56	1.40	1.31
6	1235	1.11	1.03	1.01
7	1234	2.06	1.91	1.84
8	1262	3.05	2.68	2.80
9	1263	0.96	1.02	1.19
10	1263	1.08	0.80	0.80
11	1262	1.00	0.98	0.97
12	1263	2.05	1.87	2.10
13	1264	2.64	2.72	2.74
14	1265	1.98	2.02	1.88
15	1262	1.77	1.64	1.80
16	1262	1.69	1.66	1.63
17	1265	1.22	1.11	1.14
18	1265	1.16	0.82	0.90
19	1264	1.33	1.14	1.43
20	1263	1.97	1.94	2.04
21	1264	4.41	4.37	4.49
22	1264	1.23	0.94	1.02
23	1263	1.37	1.24	1.48
24	1264	1.28	1.27	1.24
25	1264	1.29	1.21	1.30
26	1265	0.93	0.91	0.98
27	1264	0.93	0.89	0.98
28	1266	1.41	1.31	1.31
29	1265	1.17	1.08	1.07
30	1266	1.39	1.25	1.33
31	1263	1.56	1.47	1.54
32	1263	1.99	1.88	1.95
33	1261	1.12	0.95	0.98
34	1263	1.46	1.26	1.30
35	1262	1.61	1.75	1.82
36	1256	1.81	1.52	1.73
37	1256	0.98	0.96	1.01
38	1257	2.06	2.23	2.39
39	1254	0.98	0.85	0.92
40	1256	1.02	1.01	1.06

Table D.2: Numerical Results for Portfolio Selection for 100 stock dataset with $\beta = 98\%$. CVaR Values are given in basis points

No	T	$\text{CVaR}^{(1)}$	$\text{CVaR}^{(2)}$	$\text{CVaR}^{(3)}$
1	1259	1.50	0.91	1.09
2	1259	2.16	1.64	1.72
3	1232	2.28	2.16	2.17
4	1230	3.91	3.49	3.32
5	1232	2.28	1.57	1.48
6	1235	1.48	1.23	1.24
7	1234	2.46	2.08	2.02
8	1262	5.97	3.81	3.96
9	1263	1.11	1.20	1.49
10	1263	1.16	1.01	0.93
11	1262	1.13	1.16	1.15
12	1263	2.50	2.58	2.93
13	1264	3.52	3.13	3.24
14	1265	2.24	2.19	2.02
15	1262	1.97	1.86	2.06
16	1262	2.85	2.24	2.22
17	1265	1.65	1.36	1.30
18	1265	1.55	0.92	0.95
19	1264	1.40	1.29	1.60
20	1263	2.31	2.45	2.67
21	1264	6.80	6.32	7.00
22	1264	2.21	1.19	1.25
23	1263	2.30	1.54	1.91
24	1264	1.94	1.55	1.36
25	1264	1.61	1.49	1.53
26	1265	1.39	1.09	1.13
27	1264	1.23	1.08	1.15
28	1266	1.91	1.70	1.62
29	1265	1.75	1.44	1.51
30	1266	1.59	1.44	1.61
31	1263	2.11	1.99	2.09
32	1263	2.35	2.35	2.40
33	1261	1.33	1.09	1.07
34	1263	1.86	1.62	1.52
35	1262	2.09	2.05	2.18
36	1256	2.79	1.98	2.23
37	1256	1.29	1.14	1.22
38	1257	2.46	2.44	2.58
39	1254	1.39	0.96	1.11
40	1256	1.10	1.04	1.15

Table D.3: Numerical Results for Portfolio Selection for 100 stock dataset with $\beta = 99\%$. CVaR Values are given in basis points

No	T	$\text{CVaR}^{(1)}$	$\text{CVaR}^{(2)}$	$\text{CVaR}^{(3)}$
1	1233	1.03	1.03	1.03
2	1234	0.72	0.72	0.71
3	1234	1.64	1.67	1.64
4	1260	2.01	2.04	2.02
5	1263	1.27	1.26	1.26
6	1262	0.83	0.83	0.83
7	1262	0.78	0.78	0.78
8	1263	0.92	0.92	0.92
9	1263	1.15	1.13	1.15
10	1263	1.74	1.74	1.74
11	1263	1.49	1.51	1.50
12	1263	1.52	1.33	1.42
13	1264	1.18	1.11	1.16
14	1265	1.04	0.92	0.98
15	1265	1.14	1.11	1.13
16	1264	1.95	1.86	1.93
17	1264	3.41	3.20	3.33
18	1264	1.23	1.22	1.22
19	1264	0.96	0.96	0.96
20	1263	1.55	1.54	1.55
21	1264	1.05	1.05	1.05
22	1264	1.00	0.98	0.97
23	1265	1.18	1.19	1.19
24	1265	1.57	1.56	1.58
25	1265	1.23	1.04	1.04
26	1264	1.60	1.59	1.58
27	1265	1.96	1.92	1.92
28	1264	2.35	2.34	2.32
29	1263	1.82	1.82	1.80
30	1263	2.54	2.37	2.55
31	1263	2.55	2.60	2.61
32	1257	3.30	3.31	3.25
33	1256	1.64	1.62	1.63
34	1256	1.26	1.24	1.26
35	1256	1.35	1.35	1.32

Table D.4: Numerical Results for Portfolio Selection for 10 industry dataset with $\beta = 95\%$

No	T	$\text{CVaR}^{(1)}$	$\text{CVaR}^{(2)}$	$\text{CVaR}^{(3)}$
1	1233	1.27	1.20	1.18
2	1234	0.85	0.84	0.83
3	1234	1.94	1.99	1.95
4	1260	2.39	2.40	2.40
5	1263	1.67	1.65	1.64
6	1262	0.97	0.97	0.98
7	1262	0.99	0.97	0.97
8	1263	1.11	1.11	1.11
9	1263	1.67	1.65	1.68
10	1263	2.07	2.08	2.09
11	1263	1.80	1.89	1.85
12	1263	1.86	1.62	1.71
13	1264	1.47	1.24	1.34
14	1265	1.61	1.09	1.16
15	1265	1.54	1.55	1.52
16	1264	2.54	2.34	2.41
17	1264	6.34	5.11	5.54
18	1264	1.71	1.71	1.71
19	1264	1.39	1.39	1.39
20	1263	1.92	1.91	1.92
21	1264	1.27	1.27	1.27
22	1264	1.18	1.14	1.12
23	1265	1.69	1.67	1.67
24	1265	1.97	1.95	1.97
25	1265	1.63	1.30	1.32
26	1264	2.09	2.07	2.09
27	1265	2.63	2.58	2.61
28	1264	2.92	2.99	2.94
29	1263	2.06	2.06	2.04
30	1263	3.19	2.99	3.15
31	1263	3.52	3.40	3.36
32	1257	3.92	4.11	3.92
33	1256	2.03	2.00	2.01
34	1256	1.45	1.44	1.45
35	1256	1.56	1.54	1.52

Table D.5: Numerical Results for Portfolio Selection for 10 industry dataset with $\beta = 98\%$

No	T	$\text{CVaR}^{(1)}$	$\text{CVaR}^{(2)}$	$\text{CVaR}^{(3)}$
1	1233	1.38	1.34	1.32
2	1234	0.94	0.90	0.92
3	1234	2.20	2.13	2.12
4	1260	2.77	2.79	2.75
5	1263	1.99	1.91	1.95
6	1262	1.10	1.10	1.11
7	1262	1.12	1.13	1.14
8	1263	1.16	1.18	1.18
9	1263	2.28	2.22	2.25
10	1263	2.41	2.38	2.38
11	1263	1.98	2.18	2.13
12	1263	2.40	2.09	2.24
13	1264	1.70	1.44	1.58
14	1265	1.76	1.23	1.29
15	1265	2.12	2.12	2.09
16	1264	3.72	3.18	3.23
17	1264	10.05	8.49	9.46
18	1264	2.50	2.51	2.51
19	1264	1.99	1.99	1.99
20	1263	2.50	2.50	2.50
21	1264	1.39	1.39	1.39
22	1264	1.39	1.36	1.35
23	1265	2.18	2.17	2.17
24	1265	2.30	2.28	2.30
25	1265	1.88	1.61	1.64
26	1264	2.57	2.55	2.53
27	1265	3.80	3.75	3.80
28	1264	3.59	3.71	3.65
29	1263	2.22	2.23	2.20
30	1263	3.95	3.61	3.75
31	1263	4.16	4.13	4.14
32	1257	4.78	4.50	4.30
33	1256	2.32	2.31	2.33
34	1256	1.68	1.55	1.55
35	1256	1.91	1.62	1.60

Table D.6: Numerical Results for Portfolio Selection for 10 industry dataset with $\beta = 99\%$