OPERATIONS RESEARCH
Vol. 59, No. 4, July–August 2011, pp. 973–985
ISSN 0030-364X | EISSN 1526-5463 | 11 | 5904 | 0973

http://dx.doi.org/10.1287/opre.1110.0944

Robust Optimization Made Easy with ROME

Joel Goh
Stanford Graduate School of Business; NUS Business School, National University of Singapore, Singapore 119245, Republic of Singapore, joelgoh@stanford.edu

Melvyn Sim
NUS Business School and NUS Risk Management Institute, National University of Singapore, Singapore 119245, Republic of Singapore, dscsimm@nus.edu.sg

We introduce ROME, an algebraic modeling toolbox 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 discuss how ROME can be used to model (1) a service-constrained robust inventory management problem, (2) a project-crashing problem, and (3) a robust portfolio optimization problem. Through these modeling examples, we highlight the key features of ROME that allow it to expedite the modeling and subsequent numerical analysis of robust optimization problems. ROME is freely distributed for academic use at http://www.robustopt.com.

Subject classifications: robust optimization; algebraic modeling toolbox; MATLAB; stochastic programming; decision rules; inventory control; PERT; project management; portfolio optimization.
Area of review: Computing and Information Technologies.
History: Received July 2009; revision received April 2010; accepted August 2010.

1. Introduction

Robust optimization is an approach for modeling optimization problems under uncertainty, where the modeler aims to find decisions that are optimal for the worst-case realization of the uncertainties within a given set. Typically, the original uncertain optimization problem is converted into an equivalent deterministic form (called the robust counterpart) using strong duality arguments and then solved using standard optimization algorithms. Soyster (1973), Ben-Tal and Nemirovski (1998), and Bertsimas and Sim (2004) describe how to explicitly construct these robust counterparts for uncertainty sets of various structures. A significant extension on the scope of robust optimization was made by Ben-Tal et al. (2004), who considered the problem of robust decision making in a dynamic environment where uncertainties are progressively revealed. They described how adjustable robust counterparts (ARCs) can be used to model recourse decisions, which are decisions made after some or all of the uncertainties are realized. In the latter part of the same paper, they also specialized the ARC to affinely adjustable robust counterparts (also termed linear decision rules, LDRs), where decision variables are affine functions of the uncertainties, and showed that solving for such decision rules under the worst-case realization of uncertainties is typically computationally tractable.

Classical robust optimization problems are technically a special case of minimax stochastic programs, the study of which was pioneered by Žáčková (1966) and subsequently furthered in other works such as Breton and El Hachem (1995), Delage and Ye (2010), Dupačová (1987), Shapiro and Ahmed (2004), and Shapiro and Kleywegt (2002). In this setting, uncertainties are modeled as having a distribution that 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 et al. 2007). Frequent choices of families of distributions used by researchers are listed in Dupačová (2001). Throughout this paper, the term "robust optimization problem" will be used to refer to a problem in this generalized distributionally robust setting.

A recent body of research in robust optimization focuses on adapting the methodology of Ben-Tal et al. (2004) to obtain suboptimal, but ultimately tractable, approximations of solutions to 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. (2008), the extended AARC of Chen and Zhang (2009), the truncated LDR of See and Sim (2009), and the bideflected and (generalized) segregated LDRs of Goh and Sim (2010).

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, as we will illustrate in §2, 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 will firstly enable robust optimization models to be applied practically, and secondly, allow for more extensive computational tests on theoretical robust optimization models.

We aim to contribute toward the development of such a technology by introducing an algebraic modeling toolbox[1] for modeling robust optimization problems, named Robust Optimization Made Easy (ROME), which runs in the MATLAB environment. This paper covers the public release version of ROME, version 1.0 (beta) and its subversions. Using ROME, we can readily model and solve a variety of robust optimization problems. In particular, ROME allows users to use and manipulate decision rules such as LDRs, as well as LDR-based decision rules such as bideflected or segregated LDRs within their robust optimization models with relative ease. Section 5 describes the general class of problems that ROME is designed to solve, and readers are referred to Goh and Sim (2010) for a deeper discussion of the general problem and its theoretical properties.

A related problem class to the generic robust optimization problem handled in ROME is the class of multistage stochastic recourse programs (MSRPs). Both problem classes involve a progressive revelation of information, and decision making that depends on the revealed information. A common technique used to model an MSRP is the use of scenario trees to exhaustively represent its probabilistic outcomes, and recent work (e.g., by Fourer and Lopes 2009, Kaut et al. 2008, Valente et al. 2009) has focused on developing modeling tools to enhance existing algebraic modeling languages with the capability of processing and manipulating such scenario trees. In contrast, ROME does not employ scenario generation, but instead uses robust optimization theory to convert uncertain optimization models (input by a user) into their robust counterparts, which are then passed to numerical solver packages. In this regard, ROME parallels the functionality of the deterministic equivalent generator in the management system for stochastic decompositions described by Fourer and Lopes (2006).

ROME's core functionality involves translating modeling code, input by the user, into an internal structure in ROME, which is then marshaled into a solver-specific input format for solving. At present, users have a choice of the following solvers in ROME: ILOG CPLEX (IBM 2011), MOSEK (MOSEK ApS 2011), and SDPT3 (Toh et al. 1999). ROME calls both MOSEK and SDPT3 solvers through their prepackaged MATLAB interfaces, and CPLEX through the CPLEXINT interface (Baotic and Kvasnica 2006). By design, ROME's core functionality is essentially independent from the choice of solver used, allowing users to use whichever solver they are most familiar with, using the same modeling code in ROME. Due to the internal conversions performed by ROME, for a solver to be compatible with ROME, it has to at least be able to solve second-order conic programs (SOCPs).

Because ROME is built in the MATLAB environment, modelers are able to take advantage of the strength of MATLAB's numerical computational framework to prepare data, analyze optimization results, and integrate ROME more easily into their applications. Moreover, ROME is designed using similar syntax and constructs as those of MATLAB, so that users already familiar with MATLAB have a shallow learning curve in using ROME. The trade-off is that ROME's restriction to the MATLAB environment limits its flexibility in model building and expression, and lacks the versatility of specialized algebraic modeling languages such as AMPL (Fourer et al. 1990, 2002) or GAMS (Brooke et al. 1997).

ROME is similar to other MATLAB-based algebraic modeling toolboxes for optimization, such as YALMIP (Löfberg 2004, 2008), CVX (Grant and Boyd 2008, 2011), or TOMLAB (Holmström 1999), in that it aims to serve as an intermediate layer between the modeler and underlying numerical solvers. Our design goal with ROME is also similar: we aim to make the models in ROME as natural and intuitive as their algebraic formulations. A fundamental distinction between ROME and these other toolboxes is that robust optimization models typically cannot be directly modeled in these other toolboxes, with the notable exception of YALMIP, which allows users to model robust counterparts through uncertainty sets. In this respect, ROME's key distinction with YALMIP lies in ROME's comparatively richer modeling of uncertainties, not just through their support, but also through other distributional properties such as moments and directional deviations. In addition, decision rules are modeled in ROME very naturally, and ROME even incorporates more complex piecewise-linear decision rules based on the bideflected linear decision rule (Goh and Sim 2010). The trade-off is that, compared to these other toolboxes, ROME is narrower in scope in terms of the different types of deterministic optimization problems that it can model.

In this paper, we discuss several detailed modeling examples of robust optimization problems and describe how these problems have a natural representation in ROME. Through these examples, we demonstrate key features that make ROME amenable to modeling such problems. The *User's Guide to ROME* (Goh and Sim 2009) contains a comprehensive description of ROME's functionality and usage, as well as installation instructions. An electronic companion to this paper is available as part of the online version that can be found at http://or.journal.informs.org/.

Notations. We denote a random variable by the tilde sign, i.e., $\tilde{x}$. Bold lower-case letters such as $\mathbf{x}$ represent vectors, and the upper-case letters such as $\mathbf{A}$ denote matrices. In addition, $x^+ \equiv \max\{x, 0\}$ and $x^- \equiv \max\{-x, 0\}$. The same notation can be used on vectors, such as $\mathbf{y}^+$ and $\mathbf{z}^-$, indicating that the corresponding operations are performed

componentwise. 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 $[0] = \varnothing$. We also denote with a superscripted letter "c" the complement of a set, e.g., I^c. We denote by $\mathbf{e}$ the vector of all ones, and by $\mathbf{e}^i$ the ith standard basis vector. Matrix/vector transposes are denoted by the prime ($'$) symbol, e.g., $\mathbf{x}'\mathbf{y}$ denotes the inner product between two column vectors $\mathbf{x}$ and $\mathbf{y}$. The expectation of a random variable $\tilde{\mathbf{z}}$ with distribution $\mathbb{P}$ is denoted $\mathrm{E}_{\mathbb{P}}(\tilde{\mathbf{z}})$ and its covariance matrix $\mathrm{Cov}_{\mathbb{P}}(\tilde{\mathbf{z}})$. Finally, if $\tilde{\mathbf{z}}$ has a distribution $\mathbb{P}$, known only to reside in some family $\mathbb{F}$, we adopt the convention that (in)equalities involving $\tilde{\mathbf{z}}$ hold almost surely for all $\mathbb{P} \in \mathbb{F}$, i.e., for some constant vector $\mathbf{c}$, $\tilde{\mathbf{z}} \geqslant \mathbf{c} \Leftrightarrow \mathbb{P}(\tilde{\mathbf{z}} \geqslant \mathbf{c}) = 1 \ \forall \mathbb{P} \in \mathbb{F}$.

2. Motivation

The motivation for our work can be illustrated by a simple example. Consider the following simple robust optimization problem, an uncertain linear program (LP), with decision variables x and y, and a scalar uncertainty $\tilde{z}$:

$$\begin{array}{ll} \max\limits_{x,\,y} & x + 2y \\ \text{s.t.} & \tilde{z}x + y \leqslant 1 \\ & x, y \geqslant 0, \end{array} \tag{1}$$

where the only distributional information we have about the scalar uncertainty $\tilde{z}$ is that $\tilde{z} \in [-1, 1]$ almost surely. This may be interpreted as an LP with some uncertainty in a coefficient of its constraint matrix. To solve this problem numerically, we have to convert it into its robust counterpart,

$$\begin{array}{ll} \max\limits_{r,\,s,\,x,\,y} & x + 2y \\ \text{s.t.} & r + s + y \leqslant 1 \\ & r - s - x = 0 \\ & r, s, x, y \geqslant 0, \end{array} \tag{2}$$

and solve it by standard numerical solvers.

For most readers, the equivalence between (1) and (2) should not be immediately evident from the algebraic formulations. Indeed, converting from (1) to (2) requires reformulating the uncertain constraint of the original problem into an embedded LP, and invoking a strong duality argument (Ben-Tal and Nemirovski 1998, Bertsimas and Sim 2004). Furthermore, (2) has the added auxiliary variables r and s, which obscures the simplicity and interpretation of the original model (1).

This simple example exemplifies the problem of using existing deterministic algebraic modeling languages for robust optimization. To solve a robust optimization problem, the modeler has to convert the original uncertain optimization problem into its deterministic equivalent, which may be structurally very different from the original problem and have many unnecessary variables. Furthermore, for more complex problems, e.g., models that involve recourse, the conversion involves significantly more tedium, and tends to impede, rather than promote, intuition about the model.

For us, a key design goal in ROME was to build a modeling toolbox where robust optimization problems such as (1) can be modeled directly, without the modeler having to manually perform the tedious and error-prone conversion into its deterministic equivalent (2). This and other related mechanical conversions are performed internally within the ROME system to allow the modeler to focus on the core task of modeling a given problem.

3. Nonanticipative Decision Rules

A distinctive feature in ROME is the use of decision rules to model recourse decisions. In this section, we describe general properties of decision rules and the concept of nonanticipativity. We also introduce linear decision rules, which are fundamental building blocks in ROME.

We consider decision making in a dynamic system evolving in discrete time, where uncertainties are progressively revealed, and decisions, which may affect both system dynamics and a systemwide objective, are also made at fixed time epochs. At a given decision epoch, a *decision rule* is a prescription of an action, given the full history of the system's evolution until the current time. Any meaningful decision rule in practice should therefore be functionally dependent only the uncertainties that have been revealed by the current time. Such decision rules are said to be adapted or *non-anticipative*.

Assuming throughout this section that we have N model uncertainties, we may formally let $I \subseteq [N]$ represent the index set of the uncertainties that are revealed by the current time, which we will term an *information index set*. A generic $\mathbb{R}^m$-valued nonanticipative decision rule belongs to the set

$$\mathcal{Y}(m, N, I) \equiv \left\{ \mathbf{f}\colon \mathbb{R}^N \to \mathbb{R}^m\colon \mathbf{f}\left(\mathbf{z} + \sum_{i \notin I} \lambda_i \mathbf{e}^i\right) = \mathbf{f}(\mathbf{z}),\ \forall \lambda \in \mathbb{R}^N \right\}, \tag{3}$$

where the first two parameters of $\mathcal{Y}(\cdot)$ are dimensional parameters, and the last parameter I captures the functional dependence on the revealed uncertainties.

However, searching for a decision rule in $\mathcal{Y}(\cdot)$ requires searching over a space of functions, which, in general, is computationally difficult. The approach taken by recent works (Ben-Tal et al. 2006, 2004; Chen and Zhang 2009; Goh and Sim 2010) has been to restrict the search space to *linear decision rules* (LDRs), functions that are affine in the uncertainties. The LDRs are also required to satisfy the same nonanticipative requirements. Formally, a $\mathbb{R}^m$-valued LDR belongs to the set

$$\mathcal{L}(m, N, I) \equiv \left\{ \mathbf{f}\colon \mathbb{R}^N \to \mathbb{R}^m\colon \exists \mathbf{y}^0 \in \mathbb{R}^m,\ \mathbf{Y} \in \mathbb{R}^{m \times N}\colon \mathbf{f}(\mathbf{z}) = \mathbf{y}^0 + \mathbf{Y}\mathbf{z},\ \mathbf{Y}\mathbf{e}^i = \mathbf{0}, \forall i \in I^c \right\}. \tag{4}$$

Observe that by construction, $\mathcal{L}(m, N, I) \subset \mathcal{Y}(m, N, I)$. A generic $\mathbb{R}^m$-valued LDR $\mathbf{x}(\tilde{\mathbf{z}})$ can be represented in closed form as $\mathbf{x}(\tilde{\mathbf{z}}) = \mathbf{x}^0 + \mathbf{X}\tilde{\mathbf{z}}$, where $\mathbf{x}^0 \in \mathbb{R}^m$, $\mathbf{X} \in \mathbb{R}^{m\times N}$. If, in addition, $\mathbf{x}(\tilde{\mathbf{z}})$ has information index set I, then the columns of $\mathbf{X}$ with indices in the set I^c must be constrained to be all zeros. The LDR coefficients $\mathbf{x}^0$, $\mathbf{X}$ correspond to decision variables in standard mathematical programming, which are the actual quantities that are optimized in a given model.

Intuitively, one might expect that by restricting decision rules to LDRs, the modeler may suffer a penalty on the optimization objective. For a class of multistage robust optimization problems, Bertsimas et al. (2010) show that LDRs are, in fact, sufficient for optimality. However, in general, LDRs can be further improved upon. For example, Chen et al. (2008) proposed different piecewise-linear decision rules to overcome the restrictiveness imposed by LDRs. This approach was later generalized by Goh and Sim (2010) to bideflected linear decision rules (BDLDRs). These works showed that by searching over a larger space (of piecewise-linear functions), and paying a (typically) minor computational cost, the modeler can potentially improve the quality of the decision rule.

Although a detailed discussion of the theory of BDLDRs is beyond the scope of this paper (an in-depth analysis is provided in Goh and Sim 2010), we will highlight the key principles involved in their construction and usage, because BDLDRs are central to ROME's design and functionality. A generic $\mathbb{R}^m$-valued BDLDR $\hat{\mathbf{x}}(\tilde{\mathbf{z}})$ may be expressed as $\hat{\mathbf{x}}(\tilde{\mathbf{z}}) = \mathbf{w}^0 + \mathbf{W}\tilde{\mathbf{z}} + \mathbf{P}\left(\mathbf{y}^0 + \mathbf{Y}\tilde{\mathbf{z}}\right)^-$, and comprises a linear part, $\mathbf{w}^0 + \mathbf{W}\tilde{\mathbf{z}}$, and deflected part, $\mathbf{P}\left(\mathbf{y}^0 + \mathbf{Y}\tilde{\mathbf{z}}\right)^-$. Similar to the LDR, $\mathbf{w}^0 \in \mathbb{R}^m$ and $\mathbf{W} \in \mathbf{R}^{m\times N}$ are decision variables in the model. However, $\mathbf{P} \in \mathbb{R}^{m\times M}$ is a constant coefficient matrix, that is constructed on the fly, based on the structure of the model constraints and objective. In turn, its inner dimension, M, is also dependent on the problem structure. The construction of $\mathbf{P}$ involves solving a series of secondary linear optimization problems based on the problem structure (details are given in Goh and Sim 2010). Finally, $\mathbf{y}^0 \in \mathbb{R}^M$ and $\mathbf{Y} \in \mathbb{R}^{M\times N}$ are also decision variables that are solved for in the model. Notice that if I represents the information index set of $\hat{\mathbf{x}}(\tilde{\mathbf{z}})$, the resulting decision variables $(\mathbf{w}^0, \mathbf{W}, \mathbf{y}^0, \mathbf{Y})$ must also obey similar structural constraints as the LDR. However, in the BDLDR case, we have an additional layer of complexity, where the algorithm that constructs $\mathbf{P}$ is also dependent on the structure of I.

The benefit of using BDLDRs, (as compared to LDRs), to model decision rules is twofold. First, the feasible region is enlarged by searching over a larger space of piecewise-linear decision rules. Second, when worst-case expectations are taken over the BDLDR, distributional information on $\tilde{\mathbf{z}}$ leads to good bounds on the nonlinear deflected component, which improves the optimization objective. The cost of using BDLDRs is an added computational cost of solving the secondary linear programs, which is typically quite minor, and a possible increase in complexity of the final deterministic equivalent (from an LP to a SOCP) when constructing bounds on the nonlinear terms.

For a given LDR, although the decision variables from a numerical solver's perspective are $\mathbf{x}^0$ and $\mathbf{X}$, the object that is meaningful from a modeling perspective is the affine function $\mathbf{x}(\tilde{\mathbf{z}})$. In a modeling system catering to robust optimization applications, users should be able to work directly with $\mathbf{x}(\tilde{\mathbf{z}})$ and essentially ignore its internal representation. For BDLDRs, the case is even stronger. The onerous task of analyzing the problem structure, solving the secondary linear programs, and constructing the appropriate bounds, should be handled internally by the modeling system, and not by the modeler. Such considerations were fundamental to ROME's design. Through the examples presented in §4, we illustrate the utility of this design choice for modeling robust optimization problems.

4. Modeling Examples

In this section we discuss several robust optimization problems and how they can be modeled algebraically, and also how their algebraic formulations can be naturally expressed within ROME. Because the emphasis in this section is on ROME's utilty as a modeling tool, we will only present the relevant sections of algebraic formulations and the ROME models. For reference, we have included full algebraic and ROME models in the appendices. The line numbers for the code excerpts in this section correspond to those in the appendices for ease of reference.

4.1. Service-Constrained Inventory Management

4.1.1. Description. We consider a distributionally robust version of a single-product, single-echelon, multiperiod inventory management problem. Our model differs from the classical treatment of inventory management in the literature (Arrow et al. 1951) in our modeling of shortages. We assume that back orders are allowed, but instead of penalizing stockouts by imposing a linear penalty cost within the problem objective, we impose constraints on the extent of backordering within the model constraints. Specifically, we impose a constraint on the inventory *fill rate*, which is defined as (Cachon and Terwiesch 2009) the ratio of filled orders to the mean demand. Our model can be interpreted as a form of service guarantee to customers.

Other service-constrained inventory models in the literature (e.g., Boyaci and Gallego 2001, Shang and Song 2006) typically use the structural properties of the demand process to approximate the service constraint by an appropriate penalty cost within the objective. Such techniques are not suitable for our model as the actual demand distribution is unknown. See and Sim (2009) also study a robust single-product, single-echelon, multiperiod inventory management problem, but they also use a penalty cost instead of a service constraint.[2]

Therefore, in our model, the inventory manager's problem is to find ordering quantities in each period that

minimize the worst-case expected total ordering and holding cost over the finite horizon, and satisfy the fill rate constraint in each period, as well as the other standard inventory constraints. Through this example, we aim to introduce various modeling constructs in ROME and show how the ROME code is a natural expression of the model's algebraic formulation.

4.1.2. Parameters. We assume a finite planning horizon of T periods, with an exogenous uncertain demand in each period, modeled by a primitive uncertainty vector, $\tilde{\mathbf{z}} \in \mathbb{R}^T$. The exact distribution of $\tilde{\mathbf{z}}$ is unknown, but the inventory manager has knowledge of some distributional information that characterizes a family $\mathbb{F}$ of uncertainty distributions. In this example, the family $\mathbb{F}$ contains all distributions $\mathbb{P}$ that have support on the hypercube $[0, z^{\text{MAX}}]^T$, have known mean $\boldsymbol{\mu}$, and known covariance matrix $\boldsymbol{\Sigma}$, represented as $\boldsymbol{\Sigma} = \sigma \mathbf{L}(\alpha)\mathbf{L}(\alpha)'$ for known model parameters σ and α. The lower triangular matrix $\mathbf{L}(\alpha) \in \mathbb{R}^{T \times T}$ has structure

$$\mathbf{L}(\alpha) = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ \alpha & 1 & 0 & \cdots & 0 \\ \alpha & \alpha & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha & \alpha & \alpha & \ldots & 1 \end{bmatrix},$$

which represents an autoregressive model for demand, similar to Johnson and Thompson (1975) and Veinott (1965), to capture intertemporal demand correlation.

In each period $t \in [T]$, we denote by c_t the unit ordering cost and the maximum order quantity by x_t^{MAX}, with no lead time for ordering. Leftover 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. In each period, we denote the minimum required fill rate by β_t. Finally, we assume that the goods have no salvage value at the end of the T periods, there is no starting inventory, and that the inventory manager is ambiguity averse.

Table 1 lists several programming variables and the model parameters that they represent. We assume for the rest of this discussion that these variables have been declared and assigned appropriate numerical values, because we will use them within various code segments later.

4.1.3. Model. We begin by describing how uncertainties are modeled. Formally, the family of distributions $\mathbb{F}$ that contains the true distribution of the demand $\tilde{\mathbf{z}}$ can be characterized as

$$\mathbb{F} = \left\{ \mathbb{P} \colon \mathbb{P}\left(\tilde{\mathbf{z}} \in [0, z^{\text{MAX}}]^T\right) = 1, \mathrm{E}_{\mathbb{P}}(\tilde{\mathbf{z}}) = \boldsymbol{\mu}, \mathrm{Cov}_{\mathbb{P}}(\tilde{\mathbf{z}}) = \boldsymbol{\Sigma} \right\}.$$

In ROME, we model this by first declaring a programming variable z that represents the uncertainty, and assigning various distributional properties to it.

```
27 newvar z(T) uncertain;  % declare an uncertain demand
28 rome_constraint(z >= 0);
29 rome_constraint(z <= zMax); % set the support
30 z.set_mean(mu);             % set the mean
31 z.Covar = S;                % set the covariance
```

Code Segment 1: Specifying distributional properties on the uncertain demand $\tilde{\mathbf{z}}$.

Notice that the covariance matrix S is a numerical quantity that can be constructed from *alpha* and *sigma* by appropriate matrix operations.

We let $x_t(\tilde{\mathbf{z}})$ be the decision rule that represents the order quantity in period t. At the point of this decision, only the demands in the first $t-1$ periods have been realized. Hence, $x_t(\tilde{\mathbf{z}})$ should only be functionally dependent on the first $t-1$ values of $\tilde{\mathbf{z}}$, and has corresponding information index set $[t-1]$. Recall that by our convention, $[0] \equiv \varnothing$. Further, we let $y_t(\tilde{\mathbf{z}})$ be the decision rule that represents the inventory level at the *end* of period t. The corresponding information index set for $y_t(\tilde{\mathbf{z}})$ is $[t]$. When the decision rules are restricted to LDRs, for each $t \in [T]$, we require $x_t \in \mathcal{L}(1, N, [t-1])$ and $y_t \in \mathcal{L}(1, N, [t])$.

In Code Segment 2, we show how to use ROME to model the order quantity decision rule, $\mathbf{x}(\tilde{\mathbf{z}})$. The inventory level can be modeled in an identical way.

```
34 % allocate an empty variable array
35 newvar x(T) empty;
36 % iterate over each period
37 for t = 1:T
38     % construct the period t decision rule
39     newvar xt(1, z(1:(t-1))) linearrule;
40     x(t) = xt;      % assign it to the tth entry of x
41 end
```

Code Segment 2: Constructing the order quantity decision rule in ROME.

In MATLAB syntax, the colon ":" operator is used to construct numerical arrays with fixed increments between its elements. In particular, when a and b are MATLAB variables with assigned integer values with b not smaller than a, the expression a : b returns an array with unit increments,

Table 1. List of programming variables and their associated model parameters for the inventory management problem.

Variable name	Size	Model parameter	Description
T	1 × 1	T	Number of periods
zMax	T × 1	$z^{\text{MAX}}\mathbf{e}$	Support parameter for $\tilde{\mathbf{z}}$
mu	T×1	$\boldsymbol{\mu}$	Mean of $\tilde{\mathbf{z}}$
sigma	1 × 1	σ	Covariance parameter for $\tilde{\mathbf{z}}$
alpha	1 × 1	α	Covariance parameter for $\tilde{\mathbf{z}}$
S	T × T	$\boldsymbol{\Sigma}$	Covariance matrix of $\tilde{\mathbf{z}}$
xMax	T × 1	$\mathbf{x}^{\text{MAX}}$	Order quantity upper limit
c	T × 1	$\mathbf{c}$	Unit ordering cost
h	T × 1	$\mathbf{h}$	Unit holding cost
beta	T × 1	$\boldsymbol{\beta}$	Target (minimum) fill rate

starting from `a` and ending at `b`. If `b` is (strictly) smaller than `a`, the expression `a:b` returns an empty array.

The code segment begins declaring `x` as a size `T` empty array, which we will later use to store the constructed decision rules. In each iteration of the `for` loop, we construct a scalar-valued variable `xt`, which represents the period t decision rule $x_t(\tilde{\mathbf{z}})$. When `t` exceeds 1, the expression `z(1:(t-1))` extracts the first `t-1` components of `z`, which specifies the information index set of `xt`. When `t` is exactly equal to 1, the expression `z(1:(t-1))` returns an empty matrix, which indicates to ROME an empty information index set. The next line stores the constructed decision rule into the array `x` for later use.

As in standard inventory models, we have the inventory balance constraints that describe the system dynamics. The inventory level in period $t+1$ is the difference between the period t inventory level (after ordering) and the period t demand. The constraints may be written as

$$y_1(\tilde{\mathbf{z}}) = x_1(\tilde{\mathbf{z}}) - \tilde{z}_1$$
$$y_t(\tilde{\mathbf{z}}) = y_{t-1}(\tilde{\mathbf{z}}) + x_t(\tilde{\mathbf{z}}) - \tilde{z}_t \quad \forall t \in \{2, \ldots T\}.$$

Assuming that the programming variables `x` and `y` have been previously declared and represent the order quantity and inventory-level decision rules, respectively, Code Segment 3 shows how the balance constraints may be modeled in ROME.

```
% Period 1 inventory balance
rome_constraint(y(1) == x(1) - z(1));
% iterate over each period
for t = 2:T
    % period t inventory balance
    rome_constraint(y(t) == y(t-1) + x(t) - z(t));
end
```

Code Segment 3: Modeling the inventory balance constraints.

Similarly, the upper and lower limits on the order quantities can be modeled by the constraints

$$0 \leqslant x_t(\tilde{\mathbf{z}}) \leqslant x_t^{\mathrm{MAX}} \quad \forall t \in [T].$$

We may express this set of constraints in ROME as

```
rome_constraint(x >= 0);    % order qty. lower limit
rome_constraint(x <= xMax); % order qty. upper limit
```

Code Segment 4: Modeling the limits on the order quantity.

Observe that the programming variables are vector valued, and that the code segment imposes the inequality constraints component-wise.

The fill rate in a period t is defined (Cachon and Terwiesch 2009) as the ratio of expected filled orders to the mean demand, where the sales in period t can be computed as the minimum between the inventory level after ordering and the demand in the same period. In our model, we desire to meet the target (minimum) fill rate β_t for all distributions $\mathbb{P} \in \mathbb{F}$. Hence, the distributionally robust fill rate constraint reads

$$\frac{\inf_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}(\min\{y_{t-1}(\tilde{\mathbf{z}}) + x_t(\tilde{\mathbf{z}}), \tilde{z}_t\})}{\mu_t} \geqslant \beta_t$$

for each $t \in [T]$. Together with the inventory balance constraints, the fill rate constraints can be simplified to

$$\sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}((y_t(\tilde{\mathbf{z}}))^-) \leqslant \mu_t(1-\beta_t), \quad \forall t \in [T].$$

We notice that the right-hand side of the simplified inequality can be computed as a function of the model parameters. On the left-hand side, we have a term that is the expected positive part of an LDR. In a previous work (Goh and Sim 2010), we showed how such functions could be computed, or at least bounded from above, given the distributional properties of $\tilde{\mathbf{z}}$. The algebraic formulations for these bounds are typically quite messy, and dependent on the types of distributional information of $\tilde{\mathbf{z}}$ available. We refer interested readers to Goh and Sim (2010) for the exact formulations. In ROME, however, the complexity the bounds are hidden from the user, and we can model this constraint in ROME as

```
% fill rate constraint
rome_constraint(mean(neg(y)) <= mu - mu .* beta);
```

Code Segment 5: Modeling the fill rate constraint.

Notice that the MATLAB operator `.*` represents an elementwise (Hadamard) product of two vectors.

Finally, the inventory manager's objective in this problem is to minimize the worst-case expected total cost over all the distributions in the family. In our model, the total cost comprises the inventory ordering cost and the holding cost. This can be expressed as

$$\min \sup_{\mathbb{P}\in\mathbb{F}-} \mathrm{E}_{\mathbb{P}}\left(\sum_{t=1}^{T} c_t x_t(\tilde{\mathbf{z}}) + \sum_{t=1}^{T} h_t y_t^+(\tilde{\mathbf{z}})\right),$$

or more compactly, as $\min \sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}(\mathbf{c}'\mathbf{x}(\tilde{\mathbf{z}}) + \mathbf{h}'\mathbf{y}^+(\tilde{\mathbf{z}}))$. In ROME, the objective function is modeled in a similarly intuitive way. We model this in ROME by the statement

```
% model objective
rome_minimize(c'*mean(x) + h'*mean(pos(y)));
```

Code Segment 6: Modeling the inventory manager's objective.

Notice that in MATLAB syntax, the " ' " symbol represents the transpose of a vector.

4.1.4. Solutions. After modeling the problem in ROME, we can issue a call to either `solve` or `solve_deflected` to instruct ROME to, respectively, solve the problem using standard LDRs or BDLDRs. The computed solutions can be extracted for further analysis using the `eval` function. A sample numerical output (corresponding to the parameter values used in the ROME code in Appendix A) for the LDR describing the first two periods of the order quantity is

```
57.000
0.000 + 1.000*z1
```

An electronic companion containing the appendices to this paper is available as part of the online version at http://or.journal.informs.org. Solving via BDLDRs, the sample output for the first two periods of order quantities is

```
31.833
1.882 + 0.969*z1
      + 1.00(1.882 + 0.969*z1)^-
      - 1.00(58.118 - 0.969*z1)^-
```

The above displays show sample "prettyprints" of how the decision rules are output to the user and are useful for human analysis and basic solution understanding. However, this functionality clearly diminishes in utility for larger and more complex problems. ROME provide the functions (`linearpart` and `deflectedpart`) for users to extract the relevant coefficients of the decision rules for numerical analysis, if desired. However, from a modeling perspective, the numerical quantities of interest are often not the actual coefficients, but rather the instantiated values of the decision rules for the realized uncertainties. This is best exemplified in the subsequent project-crashing modeling example, and we will defer this discussion until later.

Finally, the optimized objective, which corresponds to the worst-case total cost over the family of distributions, can also be read from the ROME by calling the `objective` function.

4.1.5. Remarks. Readers who are familiar with the vectorized programming paradigm of MATLAB may find our ROME model for the inventory management problem somewhat awkward because of the use of many loops within the model. Although the model presented in this section is less computationally efficient, we feel that it bears stronger similarity to the algebraic formulation and has value in its relative ease of understanding and implementation from a modeling standpoint.

If desired, the ROME modeling code can also be vectorized for computational efficiency. In particular, the inventory balance constraint can be reformulated into a single constraint involving a matrix-vector product. The decision rules with their associated nonanticipative requirements can be constructed without using loops as well, using the `Pattern` option within their declaration, which specifies the dependency pattern on the uncertainty vector using a logical matrix. The *User's Guide to ROME* (Goh and Sim 2009) contains a detailed description of how to use the `Pattern` option, and Appendix B contains an example of a vectorized ROME model for this problem.

4.2. Project Crashing

4.2.1. Description. In this example, we consider a distributionally robust version of the project-crashing problem. We refer readers to Kerzner (2009) and Klastorin (2004) for a more complete introduction to the problem. An activity-on-arc (AOA) project network is a representation of a project in a directed acyclic graph, with arcs representing individual activities for the project. The topology of the graph represents precedence constraints of the various activities. We consider a model in which completion times of the individual activities are uncertain, but can be expedited or *crashed* by deterministic amounts by committing additional resources to the activities. Herroelen and Leus (2005) provide a comprehensive survey of the various techniques for project scheduling for uncertain activity times. In our model, we assume that the project manager's objective is to minimize the expected completion time of the project, subject to a project budget constraint, which effectively limits the amount of crashing.

A widely used technique for project analysis in practice is PERT U.S. Navy (1958). PERT makes several strong modeling assumptions and has consequently come under strong criticism (e.g., Roman 1962, van Slyke 1963). An alternative to PERT that avoids the distributional assumption on the activity times is to formulate the problem as a robust optimization problem (Cohen et al. 2007, Goh et al. 2010) to find the optimal crash amounts and activity start times. We adopt the robust optimization formulation here and show how ROME can be used to model and solve the project-crashing problem.

This example demonstrates how ROME can be used to model nonanticipative constraints over a network, and how the optimization results of ROME can be extracted for numerical analysis. In part, the latter consideration motivated our decision to design ROME within the MATLAB environment so that users can take advantage of the capabilities of the host MATLAB environment for numerical analysis and manipulation.

4.2.2. Parameters. Consider a project with N activities that have uncertain completion times represented by an uncertainty vector $\tilde{\mathbf{z}} \in \mathbb{R}^N$. Again, we do not presume exact knowledge of the actual distribution of $\tilde{\mathbf{z}}$, but instead assume that the true distribution belongs to a family of distributions $\mathbb{F}$, characterized by certain distributional properties. In particular, we assume that we know the mean activity time vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ of the activity times. Moreover, we have accurate estimates of the most optimistic activity times $\mathbf{z}_L$ and the most pessimistic activity times $\mathbf{z}_H$. These parameters form lower and upper bounds of $\tilde{\mathbf{z}}$, respectively.

Table 2. List of programming variables and their associated model parameters for the project-crashing problem.

Variable name	Size	Model parameter	Description
N	1 x 1	N	Number of activities, number of arcs in network
M	1 x 1	M	Number of nodes in network
zL	N x 1	$\mathbf{z}_L$	Optimistic activity completion time
zH	N x 1	$\mathbf{z}_H$	Pessimistic activity completion time
mu	N x 1	$\boldsymbol{\mu}$	Mean activity time
Sigma	N x T	$\boldsymbol{\Sigma}$	Covariance matrix of activity time
c	N x 1	$\mathbf{c}$	Crashing cost per unit time
u	N x 1	$\mathbf{u}$	Crashing limit
B	1 x 1	$\mathbf{B}$	Project budget

The precedence relations in the network are described by an AOA project network, with M nodes and N arcs. We represent the graph topology by an *incidence matrix* $\mathbf{A} \in \mathbb{R}^{M\times N}$, which has components

$$A_{ik} = \begin{cases} -1 & \text{if arc } k \text{ leaves node } i, \\ 1 & \text{if arc } k \text{ enters node } i, \\ 0 & \text{otherwise.} \end{cases}$$

This representation is more natural for our model instead of the (V, E) representation or an adjacency matrix representation, because it naturally preserves the integer indices on the activities. In addition, we assume that the nodes are indexed such that the starting node has index 1 and the terminal node has index M.

For each activity $k \in [N]$, we denote the maximum amount that it can be crashed by the parameter u_k, and the crashing cost per unit time by c_k. The project budget is denoted by B. Table 2 lists several programming variables and the model parameters that they represent. We assume for the rest of this discussion that these variables have been declared and assigned appropriate numerical values, because we will use them within various code segments later.

4.2.3. Model. We begin by modeling the uncertainty $\tilde{\mathbf{z}}$. This follows an identical structure to the inventory management example. The family of uncertainty distributions is formally described by

$$\mathbb{F} = \{\mathbb{P}\colon \mathbb{P}(\mathbf{z}_L \leqslant \tilde{\mathbf{z}} \leqslant \mathbf{z}_H) = 1,\ \mathrm{E}_{\mathbb{P}}(\tilde{\mathbf{z}}) = \boldsymbol{\mu},\ \mathrm{Cov}_{\mathbb{P}}(\tilde{\mathbf{z}}) = \boldsymbol{\Sigma}\},$$

The corresponding code in ROME is also identical in structure to the inventory management example, and we omit displaying it here. For reference, we have included the full algebraic model and ROME code in Appendix B.

Next, we let $y_k(\tilde{\mathbf{z}})$ be the decision rule that represents the crash amount for activity k. At the point of this decision, only the activities that precede activity k have been completed. We can recursively construct the corresponding information index set I_y^k. For any activity $l \in [N]$, define the set

$$\mathcal{P}(l) = \{l' \in [N]\colon \exists i \in [M]\colon A_{il} = -1,\ A_{il'} = 1\},$$

which is the set of activity indices that immediately precede l. Then, we may recursively evaluate I_y^k as

$$I_y^1 = \varnothing \quad \text{and} \quad I_y^k = \mathcal{P}(l) \cup \bigcup_{k' \in P(l)} I_y^{k'}, \tag{5}$$

which simply recursively includes all predecessors of k into I_y^k. The ROME model for constructing the crash amount LDR is almost identical to how the order quantity LDR was constructed in the previous example, and is shown below.

```
% y: crash amounts
newvar y(N) empty; % allocate an empty variable array
% iterate over each activity
for k = 1:N
    % get indices of dependent activities
    ind = prioractivities(k, A);
    % construct the decision rule
    newvar yk(1, z(ind)) linearrule;
    % assign it to the kth entry of y
    y(k) = yk;
end
```

Code Segment 7: Constructing the crash amount decision rule in ROME.

Here, `prioractivities` is a function that implements recursion (5) and returns the array `ind`, which contain the indices of the dependent activities. The electronic companion details the code for its implementation. The next line then uses `ind` to index into the uncertainty vector $\mathbf{z}$ to construct the dependencies of the kth activity, represented by `yk`.

The remaining decision rule is $x_i(\tilde{\mathbf{z}})$, for each node i of the network, which represents the time of node i, or, equivalently, the common start time of the activities that exit node i. Its corresponding information index set, I_x^i, can be constructed as

$$I_x^i = \begin{cases} I_y^k \text{ for any } k \text{ such that } A_{ik} = -1 & \text{if } i \neq M, \\ [N] & \text{if } i = M. \end{cases}$$

Modeling this in ROME follows directly from the construction of y above.

The time evolution of the project is represented by the inequality

$$x_j(\tilde{\mathbf{z}}) - x_i(\tilde{\mathbf{z}}) \geqslant (\tilde{z}_k - y_k(\tilde{\mathbf{z}}))^+ \quad \forall k \in [N],\ A_{ik} = -1,\ A_{jk} = 1,$$

which states that the difference in event times between two nodes that are joined by an activity must exceed the activity time after accounting for the crash. The $(\cdot)^+$ operation is a statement that regardless of how much an activity is crashed, it cannot have negative completion time. In ROME,

assuming that x and y have been properly constructed, we can express this as

```
% iterate over each activity
for k = 1:N
  ii = find(A(:, k) == -1); % activity k leaves node
  jj = find(A(:, k) == 1);  % activity k enters node
  % make constraint
  rome_constraint(x(jj) - x(ii) >= pos(z(k) - y(k)));
end
```

Code Segment 8: Expressing the project dynamics in ROME.

The remaining constraints are, firstly, the limits on the crash amounts, which are represented by the inequality $\mathbf{0} \leqslant \mathbf{y}(\tilde{\mathbf{z}}) \leqslant \mathbf{u}$; secondly, the constraint that all times should be nonnegative, which is represented by $\mathbf{x}(\tilde{\mathbf{z}}) \geqslant \mathbf{0}$; and finally, the requirement that the total crashing cost must be within the budget, which is represented by $\mathbf{c}'\mathbf{y}(\tilde{\mathbf{z}}) \leqslant B$. These inequalities are easily expressed in ROME as

```
rome_box(y, 0, u);             % crash limit
rome_constraint(x >= 0);       % nonnegative time
rome_constraint(c' * y <= B); % budget constraint
```

Code Segment 9: Project constraints in ROME.

Notice that we have used the contraction rome_box as a convenient shorthand to express a constraint on a variable having upper and lower limits.

The project completion time is simply the time of the terminal node, $\mathbf{x}_M(\tilde{\mathbf{z}})$. The project manager aims to minimize the worst-case expected completion time over the family of distributions, $\sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}(\mathbf{x}_M(\tilde{\mathbf{z}}))$. This can be expressed in ROME using the code segment

```
% objective: minimize worst-case mean completion time
rome_minimize(mean(x(M)));
```

Code Segment 10: Minimizing the expected project completion time.

4.2.4. Solutions. Similar to the inventory management example, we can use eval to return the optimized LDR or BDLDR after solving. For example, using the numerical instance of the project-crashing problem described in Appendix B, the LDR solution that represents the crash amounts, $\mathbf{y}(\tilde{\mathbf{z}})$, has the displayed output

```
y_sol =
 2.000
 1.000
 1.000
 0.381 + 0.262*z1
 0.195 - 0.020*z1
 1.021 - 0.004*z3
 0.289 + 0.033*z1 + 0.054*z4
 2.500 - 0.250*z1 + 0.000*z2
       - 0.000*z3 + 0.000*z5 - 0.000*z6
 0.529 + 0.094*z3
```

which also can be used to verify that y does indeed satisfy the nonanticipative constraints, which is more complicated than the multistage nonanticipativity seen in the inventory management example.

Recall that in our model we solve for the worst-case distribution over a family of distributions. A question that is of both theoretical and practical interest is how the decision rules perform for specific distributions. From a theoretical perspective, if we can find a distribution that attains the worst case, then we know that the bound is tight and cannot be further improved. From a practical standpoint, if we do indeed have an accurate estimate of the activity time distribution, we may want to measure how well the worst-case optimized decision rules perform against this distribution.

In the numerical instance in Appendix B, we have used mean and covariance parameters such that an independent uniform distribution on the support of $\tilde{\mathbf{z}}$ is a possible distribution in $\mathbb{F}$. We can then generate these independent uniforms and instantiate the decision rules and the objective to evaluate their performance on this specific distribution. For example, if we want to study the statistics of the project expenditure, we would use the following code segment

```
Nsims = 10000;          % number of simulation runs
expenditure = zeros(Nsims, 1);  % allocate array
for ii = 1:Nsims
    % simulate activity times
    z_vals = zL + (zH - zL) .* rand(N, 1);
    % instantiate crash costs
    expenditure(ii) = c' * y_sol.insert(z_vals);
end
```

Code Segment 11: Example of instantiating decision rules with uncertainty realizations.

Notice that MATLAB's in-built zeros function creates an array of all zeros, which we use to store the computed expenditures. The rand function generates an array of N independent uniform random $[0,1]$ numbers that we shift and scale to get the simulated activity times.

The key ROME functionality used here is the insert function, which instantiates the LDR solution with the uncertainties. The project expenditure in simulation run is the inner product of the unit cost vector, c, with the result of the y_sol.insert(z_vals), which is a numerical vector. The computed expenditure is then stored in the expenditure array and can be used for various numerical analysis. As an example, in this particular instance we can find that using LDRs, the 95% confidence interval of the mean expenditure is $(9.6072, 9.6125)$, whereas if BDLDRs are used, the 95% confidence interval of the mean expenditure is $(9.8781, 9.8863)$.

4.2.5. Remarks. ROME can also model several variants of the project-crashing model presented here. For example, we could minimize the expected project-crashing cost, subject to a constraint on the project completion deadline. Alternatively, if appropriate linear penalty cost parameters can be associated with late completion, we could instead minimize the expected total (crashing and

penalty) cost in the objective. Finally, instead of minimizing expected costs, we could also minimize worst-case costs, as in the model of Cohen et al. (2007).

4.3. Robust Portfolio Selection

4.3.1. Description. In this section, we consider the problem of robust portfolio selection. Markowitz (1952, 1959) pioneered the use of optimization to handle the trade-off 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 1958 and Chamberlain 1983).

A risk metric that overcomes many of the shortcomings of variance is the Conditional Value-at-Risk (CVaR) risk metric popularized by Rockafellar and Uryasev (2000). CVaR satisfies many desirable properties of a risk measure, qualifying it as a *coherent* measure of risk (Artzner et al. 1999). CVaR is a measure of tail risk, measuring the expected loss within a specified quantile. In this example, we will use CVaR as our objective for the portfolio optimization. In addition, we will require that the optimized portfolio has a mean return above a prespecified target, and study how the characteristics of the optimal portfolio changes as the target return level varies. In particular, we will investigate how the coefficient of variation (CV) of the optimized portfolio return varies with the target return level. We will assume that no short-selling is permitted, and we also make the standard assumption that the assets are traded in a frictionless market (i.e., a fully liquid market with no transaction costs).

In this example, we aim to show how ROME can be used to study robust optimization problems that do not ostensibly require decision rules, how ROME models can be developed in a modular manner by writing custom functions, and finally, how a ROME model can be easily embedded as components of a larger program.

4.3.2. Parameters. We let N denote the total number of assets available for investment. The drivers of uncertainty in this model are the asset returns, which we denote by the uncertainty vector $\tilde{\mathbf{r}} \in \mathbb{R}^N$. We do not presume to know the precise distribution of $\tilde{\mathbf{r}} \in \mathbb{R}^N$, but instead, we assume that we have accurate estimates of the asset return means $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$, which characterizes a family of distributions $\mathbb{F}$.

A key exogenous parameter is the CVaR-level, β, which specifies the quantile over which the conditional expected loss is computed. Typical values of β are 95%, 98%, or 99%. In addition, we let $[\tau_L, \tau_H]$ represent the range of target returns that we search over. For a fixed target return $\tau \in [\tau_L, \tau_H]$, the portfolio manager's problem is to find a portfolio allocation that has a mean return above τ, and minimizes the worst β-CVaR over all uncertainty distributions within $\mathbb{F}$. Table 3 lists several programming variables and the model parameters that they represent. We assume for the rest of this discussion that these variables have been declared and assigned appropriate numerical values, because we will use them within various code segments later.

Table 3. List of programming variables and their associated model parameters for the inventory management problem.

Variable name	Size	Model parameter	Description
`N`	1 x 1	N	Number of assets
`mu`	N x 1	$\boldsymbol{\mu}$	Mean return
`Sigma`	N x T	$\boldsymbol{\Sigma}$	Covariance matrix returns
`beta`	1 x 1	β	CVaR-level
`tL`	1 x 1	τ_L	Lower limit of target return
`tH`	1 x 1	τ_H	Upper limit of target return

4.3.3. Model. In this model, the portfolio manager's decision is a vector $\mathbf{x} \in \mathbb{R}^N$, with the ith component representing the fraction of the net wealth to be invested in asset i. Because we do not consider any dynamics in this problem, the primary decision rule $\mathbf{x}$ is a standard vector-valued decision variable in this problem.

For a fixed β, and known distribution $\mathbb{P}$, letting the portfolio loss as a function of the decision be represented by $\tilde{\ell}(\mathbf{x}) = -\tilde{\mathbf{r}}'\mathbf{x}$, the β-quantile of the loss distribution can be found via

$$\alpha_\beta(\mathbf{x}) = \min\{v \in \mathbb{R}\colon \mathbb{P}(\tilde{\ell}(\mathbf{x}) \leqslant v) \geqslant \beta\}.$$

Therefore, $1-\beta$ represents the probability that the loss exceeds $\alpha_\beta(\mathbf{x})$. The β-CVaR is defined as the conditional loss, given that the loss exceeds $\alpha_\beta(\mathbf{x})$. This is in turn found via

$$\mathrm{CVaR}_\beta(\mathbf{x}) = \mathbb{E}_{\mathbb{P}}(\tilde{\ell}(\mathbf{x}) \mid \tilde{\ell}(\mathbf{x}) \geqslant \alpha_\beta(\mathbf{x})).$$

Rockafellar and Uryasev (2000, Theorem 1) show that the expression CVaR can be written in a more amenable form for optimization,

$$\mathrm{CVaR}_\beta(\mathbf{x}) = \min_{v\in\mathbb{R}} \left\{ v + \frac{1}{1-\beta}\mathbb{E}_{\mathbb{P}}\big(\tilde{\ell}(\mathbf{x}) - v\big)^+ \right\}.$$

In our distributionally robust setting, we consider the worst-case CVaR over all distributions $\mathbb{P}$ within the family $\mathbb{F}$, and we get

$$\mathrm{CVaR}_\beta(\mathbf{x}) = \min_{v\in\mathbb{R}} \left\{ v + \frac{1}{1-\beta}\sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\big(\big(\tilde{\ell}(\mathbf{x}) - v\big)^+\big) \right\}. \quad (6)$$

We can encapsulate this within a function:

```
6 function cvar = CVaR(loss, beta)
7    newvar v;   % declare an auxilliary variable v
8    cvar = v + (1 / (1 - beta)) * mean(pos(loss - v));
```

Code Segment 12: Implementing a custom CVaR function in ROME.

Therefore, for a fixed $\tau \in [\tau_L, \tau_H]$, the β-CVaR minimizing portfolio, denoted by $\mathbf{x}^*(\tau)$, solves

$$\begin{aligned} \min_{\mathbf{x}} \ & \mathrm{CVaR}_\beta(-\tilde{\mathbf{r}}'\mathbf{x}) \\ \text{s.t.} \ & \boldsymbol{\mu}'\mathbf{x} \geqslant \tau \\ & \mathbf{e}'\mathbf{x} = 1 \\ & \mathbf{x} \geqslant \mathbf{0}. \end{aligned} \tag{7}$$

The first constraint represents the requirement that the optimized portfolio must have a mean return above τ. The second constraint is simply a normalization. Because $\mathbf{x}$ is a vector whose components represents fractions of net wealth, its components must sum to 1. Finally, the last constraint is simply a statement of the modeling assumption that short sales are not permitted.

This can be modeled in ROME as

```
function x_sol = optimizeportfolio (N, mu, Sigma, ...
    beta, tau)

% begin the ROME environment
h = rome_begin('Portfolio Optimization');

newvar r(N) uncertain; % declare r as uncertainty
r.set_mean(mu);        % set mean
r.Covar = Sigma;       % set covariance

% declare a nonnegative variable x
newvar x(N) nonneg;

% objective: minimize CVaR
rome_minimize(CVaR(-r' * x, beta));
% mean return must exceed tau
rome_constraint(mu' * x >= tau);
% x must sum to 1
rome_constraint(sum(x) == 1);

% solve the model
h.solve_deflected;

% check for infeasibility / unboundedness
if(isinf(h.objective))
    x_sol = [];        % assign an empty matrix
else
    x_sol = h.eval(x); % get the optimal solution
end
rome_end;              % end the ROME environment
```

Code Segment 13: Portfolio optimization example in ROME.

The CV is a dimensionless measure of variability of the optimized portfolio return and is defined as the ratio of its standard deviation to its mean. As a function of τ, it is formally defined as

$$\mathrm{CV}(\tau) \equiv \frac{\sqrt{\mathbf{x}^*(\tau)'\boldsymbol{\Sigma}\mathbf{x}^*(\tau)}}{\boldsymbol{\mu}'\mathbf{x}^*(\tau)}. \tag{8}$$

To investigate how the CV changes for $\tau \in [\tau_L, \tau_H]$, we can simply compute the CV for a uniform sample of τ within this range, and plot the ensuing CV against τ. This can be done by the following code segment.

```
Npts = 200;              % number of points in to plot
cv = zeros(Npts, 1);     % allocate result array

% array of target means to test
tau_arr = linspace(0, tH, Npts);

for ii = 1:Npts
  % Find the CVaR-optimal portfolio
  x_sol = optimizeportfolio(N, mu, Sigma, ...
     beta, tau_arr(ii));

   % Store the coeffients of variation
  if(isempty(x_sol))
    cv(ii) = Inf;
  else
    cv(ii) = sqrt(x_sol'*Sigma*x_sol) / (mu'*x_sol);
  end
end

plot(tau_arr, cv);             % plot CV against tau
xlabel('\tau'); ylabel('CV'); % label axes
title('Plot of CV vs \tau');  % label graph
```

Code Segment 14: Plotting the CV of β-CVaR optimized portfolio against τ.

In the previous code segment, `linspace(0, tH, Npts)` is a MATLAB in-built function that returns an array with `Npts` elements with equal consecutive differences, starting with the element `0` and ending with `tH`.

5. ROME's Scope

In the previous section, we discussed how ROME can be used in several example applications in different areas of operations research. In this section we formally establish ROME's scope and the class of problems that it can model and solve.

We denote by $\tilde{\mathbf{z}}$ 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 $\tilde{\mathbf{z}}$, namely, its support, mean support, covariance matrix, and upper bounds on its directional deviations (introduced by Chen et al. 2007). The presence of these properties serve to characterize a family of distributions, which we generally denote by $\mathbb{F}$, that contains the actual distribution $\mathbb{P}$.

The decision variables in our framework comprise $\mathbf{x}$, 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 decisions rules $\mathbf{y}^k(\cdot)$, with image in $\mathbb{R}^{m_k}$ for each $k \in [K]$. We assume that we are given as parameters the information index sets $\{I_k\}_{k=1}^K$, where $I_k \subseteq [N]$ $\forall k \in [K]$. To model the nonanticipative requirements, we require that the decision rules belong to the sets $y^k \in \mathcal{Y}(m_k, N, I_k)$, $\forall k \in [K]$, where $\mathcal{Y}$ is the set of nonanticipative decision rules as defined in (3).

The general model that we consider is then

$$Z^*_{\mathrm{GEN}} = \min_{\mathbf{x},\, \{\mathbf{y}^k(\cdot)\}_{k=1}^K} \mathbf{c}^{0\prime}\mathbf{x} + \sup_{\mathbb{P}\in\mathbb{F}-} \mathrm{E}_{\mathbb{P}}\left(\sum_{k=1}^K \mathbf{d}^{0,k\prime}\mathbf{y}^k(\tilde{\mathbf{z}})\right)$$

$$\text{s.t.}\, \mathbf{c}^{l\prime}\mathbf{x} + \sup_{\mathbb{P}\in\mathbb{F}-} \mathrm{E}_{\mathbb{P}}\left(\sum_{k=1}^K \mathbf{d}^{l,k\prime}\mathbf{y}^k(\tilde{\mathbf{z}})\right) \leqslant b_l \quad \forall\, l \in [M] \qquad (9)$$

$$\mathbf{T}(\tilde{\mathbf{z}})\mathbf{x} + \sum_{k=1}^K \mathbf{U}^k\mathbf{y}^k(\tilde{\mathbf{z}}) = \mathbf{v}(\tilde{\mathbf{z}})$$

$$\underline{\mathbf{y}}^k \leqslant \mathbf{y}^k(\tilde{\mathbf{z}}) \leqslant \bar{\mathbf{y}}^k \quad \forall\, k \in [K]$$

$$\mathbf{x} \geqslant \mathbf{0}$$

$$\mathbf{y}^k \in \mathcal{Y}(m_k, N, I_k) \quad \forall\, k \in [K]. \qquad (10)$$

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

Notice that in ROME we do not solve (9) exactly, but instead solve for optimized decision rules residing in structured subsets of $\mathcal{Y}$, corresponding to a restriction of (9). This is because although problem (9) is quite general, it is also unfortunately computationally intractable in most cases (see Ben-Tal et al. 2004 or Shapiro and Nemirovski 2005 for a more detailed discussion). The set of LDRs, $\mathcal{L}$, defined in (4), is the default subset of $\mathcal{Y}$ used in ROME. If desired, at the expense of a typically minor computational cost, users can also choose to solve for BDLDRs, which reside in a larger subset of $\mathcal{Y}$.

6. Conclusion

In this paper, we have introduced ROME, a MATLAB-based robust optimization modeling toolbox, and demonstrated its utility in modeling robust optimization problems. Through the three modeling examples discussed in this paper, we have introduced ROME's key features, and we have demonstrated how ROME allows users to model otherwise complex problems with relative ease, in a mathematically intuitive manner. In addition, through the final portfolio optimization example, we have also demonstrated how ROME might be integrated into a sample application. We believe that ROME can be a helpful and valuable tool for further academic and industrial research in the field of robust optimization.

7. Electronic Companion

An electronic companion to this paper is available as part of the online version that can be found at http://or.journal.informs.org/.

Endnotes

1. Freely available for academic use from http://www.robustopt.com

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

Acknowledgments

The authors thank the associate editor and the anonymous referees for their comments and critique on the first version of this paper. They are also especially grateful for the detailed comments by Area Editor Robert Fourer for his detailed feedback on the first version of this paper. Their feedback has allowed the authors to significantly improve the structure and quality of this paper.

References

Arrow, K. J., T. Harris, J. Marschak. 1951. Optimal inventory policy. *Econometrica* **19**(3) 250–272.

Artzner, P., F. Delbaen, J. M. Eber, D. Heath. 1999. Coherent measures of risk. *Math. Finance* **9**(3) 203–228.

Baotic, M., M. Kvasnica. 2006. CPLEXINT—MATLAB interface for the CPLEX solver. Accessed August 9, 2011, http://control.ee.ethz.ch/~hybrid/cplexint.php.

Ben-Tal, A., A. Nemirovski. 1998. Robust convex optimization. *Math. Oper. Res.* **23**(4) 769–805.

Ben-Tal, A., S. Boyd, A. Nemirovski. 2006. Extending scope of robust optimization: Comprehensive robust counterparts of uncertain problems. *Math. Programming Ser. B* **107**(1–2) 63–89.

Ben-Tal, A., A. Goryashko, E. Guslitzer, A. Nemirovski. 2004. Adjustable robust solutions of uncertain linear programs. *Math. Programming* **99**(2) 351–376.

Bertsimas, D., M. Sim. 2004. The price of robustness. *Oper. Res.* **52**(1) 35–53.

Bertsimas, D., D. A. Iancu, P. A. Parrilo. 2010. Optimality of affine policies in multistage robust optimization. *Math. Oper. Res.* **35**(2) 363–394.

Boyaci, T., G. Gallego. 2001. Serial production/distribution systems under service constraints. *Manufacturing Service Oper. Management* **3**(1) 43–50.

Breton, M., S. El Hachem. 1995. Algorithms for the solution of stochastic dynamic minimax problems. *Comput. Optim. Appl.* **4** 317–345.

Brooke, A., D. Kendrick, A. Meeraus, R. Raman. 1997. *GAMS Language Guide*. Release 2.25. GAMS Development Corporation, Washington, DC.

Cachon, G., C. Terwiesch. 2009. *Matching Supply with Demand: An Introduction to Operations Management*. McGraw-Hill, Boston.

Chamberlain, G. 1983. A characterization of the distributions that imply mean-variance utility functions. *J. Econom. Theory* **29**(1) 185–201.

Chen, X., Y. Zhang. 2009. Uncertain linear programs: Extended affinely adjustable robust counterparts. *Oper. Res.* **57**(6) 1469–1482.

Chen, X., M. Sim, P. Sun. 2007. A robust optimization perspective on stochastic programming. *Oper. Res.* **55**(6) 1058–1071.

Chen, X., M. Sim, P. Sun, J. Zhang. 2008. A linear decision-based approximation approach to stochastic programming. *Oper. Res.* **56**(2) 344–357.

Cohen, I., B. Golany, A. Shtub. 2007. The stochastic time-cost tradeoff problem: A robust optimization approach. *Networks* **49**(2) 175–188.

Delage, E., Y. Ye. 2010. Distributionally robust optimization under moment uncertainty with application to data-driven problems. *Oper. Res.* **58**(3) 595–612.

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

Dupačová, J. 2001. Stochastic programming: Minimax approach. C. Floudas, P. Pardalos, eds. *Encyclopedia of Optimization*, Vol. 5. Kluwer Academic Publishers, Norwell, MA, 327–330.

Fourer, R., L. Lopes. 2006. A management system for decompositions in stochastic programming. *Ann. Oper. Res.* **142** 99–118.

Fourer, R., L. Lopes. 2009. StAMPL: A filtration-oriented modeling tool for multistage stochastic recourse problems. *INFORMS J. Comput.* **21**(2) 242–256.

Fourer, R., D. M. Gay, B. W. Kernighan. 1990. A modeling language for mathematical programming. *Management Sci.* **36**(5) 519–554.

Fourer, R., D. M. Gay, B. W. Kernighan. 2002. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, Brooks/Cole Publishing, Pacific Grove, CA.

Goh, J., M. Sim. 2009. *User's Guide to ROME*. Accessed August 9, 2011, http://www.robustopt.com/references/ROME_Guide_1.0.pdf.

Goh, J., M. Sim. 2010. Distributionally robust optimization and its tractable approximations. *Oper. Res.* **58**(4) 902–917.

Goh, J., N. G. Hall, M. Sim. 2010. Robust optimization strategies for total cost control in project management. Working paper, NUS Business School, Singapore.

Grant, M., S. Boyd. 2008. Graph implementations for nonsmooth convex programs. V. Blondel, S. Boyd, H. Kimura, eds. *Recent Advances in Learning and Control (a Tribute to M. Vidyasagar)*. Springer, New York, 95–110.

Grant, M., S. Boyd. 2011. CVX: MATLAB software for disciplined convex programming (Web page, software). Retrieved August 9, 2011, http://cvxr.com/cvx.

Herroelen, W., R. Leus. 2005. Project scheduling under uncertainty: Survey and research potentials. *Eur. J. Oper. Res.* **165** 289–306.

Holmström, K. 1999. The TOMLAB optimization environment in MATLAB. *Adv. Model. Optim.* **1**(1) 47–69.

IBM. 2011. IBM ILOG CPLEX. Accessed August 9, 2011, http://www-01.ibm.com/software/integration/optimization/cplex-optimizer.

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

Kaut, M., A. King, T. H. Hultberg. 2008. A C++ modelling environment for stochastic programming. Technical report, IBM, New York.

Kerzner, H. 2009. *Project Management: A Systems Approach to Planning, Scheduling, and Controlling*, 10th ed. Wiley, Hoboken, NJ.

Klastorin, T. D. 2004. *Project Management: Tools and Trade-Offs*, 1st ed. Wiley, Hoboken, NJ.

Löfberg, J. 2004. YALMIP: A toolbox for modeling and optimization in MATLAB. *IEEE Internat. Sympos. Comput. Aided Control Systems Design*, Taiwan.

Löfberg, J. 2008. Modeling and solving uncertain optimization problems in YALMIP. *Proc. 17th World Congress: The Internat. Federation of Automatic Control*, Seoul, Korea.

Markowitz, H. M. 1952. Portfolio selection. *J. Finance* **7** 77–91.

Markowitz, H. M. 1959. *Portfolio Selection: Efficient Diversification of Investments*. John Wiley & Sons, New York.

MOSEK ApS. The MOSEK optimization software. Accessed August 9, 2011, http://www.mosek.com.

Rockafellar, R. T., S. Uryasev. 2000. Optimization of conditional value-at-risk. *J. Risk* **2** 493–517.

Roman, D. D. 1962. The PERT system: An appraisal of program evaluation review technique. *J. Acad. Management* **5**(1) 57–65.

See, C.-T., M. Sim. 2009. Robust approximation to multiperiod inventory management. *Oper. Res.* **58**(3) 583–594.

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

Shapiro, A., S. Ahmed. 2004. On a class of minimax stochastic programs. *SIAM J. Optim.* **14**(4) 1237–1249.

Shapiro, A., A. Kleywegt. 2002. Minimax analysis of stochastic programs. *Optim. Methods Software* **17**(3) 523–542.

Shapiro, A., A. Nemirovski. 2005. On complexity of stochastic programming problems. V. Jeyakumar, A. Rubinov, eds. *Continuous Optimization*. Springer, New York, 111–146.

Soyster, A. L. 1973. Convex programming with set-inclusive constraints and applications to inexact linear programming. *Oper. Res.* **21**(5) 1154–1157.

Tobin, J. 1958. Liquidity preference as behavior toward risk. *Rev. Econom. Stud.* **25** 65–85.

Toh, K., M. Todd, R. Tütüncü. 1999. SDPT3—A MATLAB software package for semidefinite programming. *Optim. Methods Software* **11** 545–581.

U.S. Navy. 1958. PERT summary report, phase I. Technical report, Special Projects Office, Bureau of Naval Weapons, Washington, DC.

Valente, C., G. Mitra, M. Sadki, R. Fourer. 2009. Extending algebraic modelling languages for stochastic programming. *INFORMS J. Comput.* **21**(1) 107–122.

Van Slyke, R. M. 1963. Monte Carlo methods and the PERT problem. *Oper. Res.* **11**(5) 839–860.

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

Žáčková, J. 1966. On minimax solution of stochastic linear programming problems. *Časopis pro Pěstování Matematiky* **91**(4) 423–430.