

Robust Optimization Made Easy with ROME

Joel Goh*

Melvyn Sim †

Submitted: July 2009, Revised: Mar 2010

Abstract

We introduce ROME, an algebraic modeling toolbox for modeling 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 which allows it to expedite the modeling and subsequent numerical analysis of robust optimization problems. ROME is freely distributed for academic use from www.robustopt.com.

*Stanford Graduate School of Business and NUS Business School, National University of Singapore. Email: joelgoh@stanford.edu. The research of the author is supported by the Kaneko/Lainovic International Fellowship.

†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 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 [49], Ben-Tal and Nemirovski [6], Bertsimas and Sim [8] describe how to explicitly construct these robust counterparts for uncertainty sets of various structures. A significant extension on the scope of in robust optimization was made by Ben-Tal et al. [5], 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 Záčková [56] and subsequently furthered in other works such as [10, 18, 19, 46, 47]. 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 [14]). Frequent choices of families of distributions used by researchers are listed in [20]. 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. [5] to obtain sub-optimal, 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. [15], the extended AARC of Chen and Zhang [16], the truncated LDR of See and Sim [44], and the bi-deflected and (generalized) segregated LDRs of Goh and Sim [27].

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 Section 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 would 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 toolbox¹ 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 sub-versions. Using ROME, we can readily model and solve a variety of robust optimization problems. Section 5 describes the general class of problems that ROME is designed to solve, and readers are referred to [27] 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 a MSRP is the use of scenario trees to exhaustively represent its probabilistic outcomes, and recent work (e.g. by Fourer and Lopes [24], Kaut et al. [34], Valente et al. [53]) 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 [23].

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: IBM ILOG CPLEX [32], MOSEK [41], and SDPT3 [51]. ROME calls both MOSEK and SDPT3 solvers through their pre-packaged MATLAB interfaces, and CPLEX through the CPLEXINT [3] interface. 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 the ROME, for a solver to be compatible with ROME, it has to at least be able to solve second-order conic programs (SOCPs).

As 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 MATLAB, so that users already familiar with MATLAB have a shallow learning curve in using ROME. The tradeoff 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 [21, 22] or GAMS [11].

ROME is similar to other MATLAB-based algebraic modeling toolboxes for optimization, such as YALMIP [37, 38], CVX [28, 29], or TOMLAB [31], 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:

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

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 bi-deflected linear decision rule (Goh and Sim [27]). The tradeoff 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 which make ROME amenable to modeling such problems. The ROME User’s Guide [26] contains a comprehensive description of ROME’s functionality and usage, as well as installation instructions.

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. $[N] = \{1, 2, \dots, N\}$, 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 \mathbf{e} the vector of all ones, and by \mathbf{e}^i the i^{th} 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 $\mathbb{E}_{\mathbb{P}}(\tilde{\mathbf{z}})$ and its covariance matrix $\text{Cov}_{\mathbb{P}}(\tilde{\mathbf{z}})$. 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} , $\tilde{\mathbf{z}} \geq \mathbf{c} \iff \mathbb{P}(\tilde{\mathbf{z}} \geq \mathbf{c}) = 1 \forall \mathbb{P} \in \mathbb{F}$.

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}

$$\begin{aligned} \max_{x,y} \quad & x + 2y \\ \text{s.t.} \quad & \tilde{z}x + y \leq 1 \\ & x, y \geq 0, \end{aligned} \tag{2.1}$$

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,

$$\begin{aligned}
\max_{r,s,x,y} \quad & x + 2y \\
\text{s.t.} \quad & r + s + y \leq 1 \\
& r - s - x = 0 \\
& r, s, x, y \geq 0,
\end{aligned} \tag{2.2}$$

and solved by standard numerical solvers.

For most readers, the equivalence between (2.1) and (2.2) should not be immediately evident from the algebraic formulations. Indeed, converting from (2.1) to (2.2) requires reformulating the uncertain constraint of the original problem into an embedded LP, and invoking a strong duality argument (Ben-Tal and Nemirovski [6], Bertsimas and Sim [8]). Furthermore, (2.2) has the added auxiliary variables r and s , which obscures the simplicity and interpretation of the original model (2.1).

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 which 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 (2.1) can be modeled directly, without the modeler having to manually perform the tedious and error-prone conversion into its deterministic equivalent (2.2). 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 non-anticipativity. 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 system-wide 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

$$\mathcal{Y}(m, N, I) \equiv \left\{ \mathbf{f} : \mathbb{R}^N \rightarrow \mathbb{R}^m : \mathbf{f} \left(\mathbf{z} + \sum_{i \notin I} \lambda_i \mathbf{e}^i \right) = \mathbf{f}(\mathbf{z}), \forall \boldsymbol{\lambda} \in \mathbb{R}^N \right\}, \tag{3.1}$$

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 [4, 5, 16, 27] 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

$$\mathcal{L}(m, N, I) \equiv \{ \mathbf{f} : \mathbb{R}^N \rightarrow \mathbb{R}^m : \exists \mathbf{y}^0 \in \mathbb{R}^m, \mathbf{Y} \in \mathbb{R}^{m \times N} : \mathbf{f}(\mathbf{z}) = \mathbf{y}^0 + \mathbf{Y}\mathbf{z}, \mathbf{Y}\mathbf{e}^i = \mathbf{0}, \forall i \in I^c \}. \quad (3.2)$$

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 multi-stage robust optimization problems, Bertsimas et al. [7] show that LDRs are, in fact, sufficient for optimality. However, in general, LDRs can be further improved upon. For example, Chen et al. [15] proposed different piecewise-linear decision rules to overcome the restrictiveness imposed by LDRs. This approach was later generalized by Goh and Sim [27] to bi-deflected 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.

While a detailed discussion of the theory of BDLDRs are beyond the scope of this paper (a detailed analysis is provided in [27]), we will highlight the key principles involved in their construction and usage. A generic \mathbb{R}^m -valued BDLDR $\hat{\mathbf{x}}(\tilde{\mathbf{z}})$ may be expressed as $\hat{\mathbf{x}}(\tilde{\mathbf{z}}) = \mathbf{w}^0 + \mathbf{W}\tilde{\mathbf{z}} + \mathbf{P}(\mathbf{y}^0 + \mathbf{Y}\tilde{\mathbf{z}})^-$, and comprises a linear part, $\mathbf{w}^0 + \mathbf{W}\tilde{\mathbf{z}}$, and deflected part, $\mathbf{P}(\mathbf{y}^0 + \mathbf{Y}\tilde{\mathbf{z}})^-$. Similar to the LDR, $\mathbf{w}^0 \in \mathbb{R}^m$ and $\mathbf{W} \in \mathbb{R}^{m \times N}$ are decision variables in the model. However, $\mathbf{P} \in \mathbb{R}^{m \times M}$ is a constant coefficient matrix, which 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 [27]). Finally, $\mathbf{y}^0 \in \mathbb{R}^M$ and $\mathbf{Y} \in \mathbb{R}^{M \times N}$ are also decision variables which 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 which 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 two-fold. 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, while 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 catered 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 Section 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. As the emphasis in this section is on ROME’s utility 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, multi-period inventory management problem. Our model differs from the classical treatment of inventory management in the literature (Arrow et al. [1]) 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 back ordering within the model constraints. Specifically, we impose a constraint on the inventory *fill rate*, which is defined as (Cachon and Terwiesch [12]) 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 [9], Shang and Song [45]) 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 [44] also study a robust single-product, single-echelon, multi-period inventory management problem, but they also use a penalty cost instead of a service constraint².

Therefore, in our model, the inventory manager’s problem is to find ordering quantities in each period, which minimizes the worst-case expected total ordering and holding cost over the finite horizon,

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

and satisfies 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, $\tilde{z} \in \mathbb{R}^T$. The exact distribution of \tilde{z} is unknown, but the inventory manager has knowledge of some distributional information, which 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 $[0, z^{MAX}]^T$, 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

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

which represents an autoregressive model for demand, similar to Johnson and Thompson [33] and Veinott [55], to capture inter-temporal 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 4.1 lists several programming variables and the model parameters which they represent. We assume for the rest of this discussion that these variables have been declared and assigned appropriate numerical values, as we will use them within various code segments later.

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

Table 4.1: List of programming variables and their associated model parameters for the inventory management problem

4.1.3 Model

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

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

In ROME, we model this by first declaring a programming variable \mathbf{z} which represents the uncertainty, and assign various distributional properties to it.

```

24 newvar z(T) uncertain;           % declare an uncertain demand
25 rome_constraint(z >= 0);
26 rome_constraint(z <= zMax);    % set distributional support
27 z.set_mean(mu);               % set mean
28 z.Covar = S;                  % set covariance

```

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

Notice that the covariance matrix \mathbf{S} is a numerical quantity that can be constructed from \mathbf{alpha} and \mathbf{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, $[0] \equiv \emptyset$. Further, we let $y_t(\tilde{\mathbf{z}})$ be the decision rule which 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.

```

31 newvar x(T) empty; % allocate an empty variable array
32 for t = 1:T % iterate over each period
33     newvar xt(1, z(1:(t-1))) linearrule; % construct the period t decision rule
34     x(t) = xt; % assign it to the tth entry of x
35 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 \mathbf{a} and \mathbf{b} are MATLAB variables with assigned integer values with \mathbf{b} not smaller than \mathbf{a} , the expression $\mathbf{a}:\mathbf{b}$ returns an array with unit increments, starting from \mathbf{a} and ending at \mathbf{b} . If \mathbf{b} is (strictly) smaller than \mathbf{a} , the expression $\mathbf{a}:\mathbf{b}$ returns an empty array.

The code segment begins declaring \mathbf{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 \mathbf{xt} , which represents the period t decision rule $x_t(\tilde{\mathbf{z}})$. When t exceeds 1, the expression $\mathbf{z}(1:(t-1))$ extracts the first $t-1$ components of \mathbf{z} , which specifies the information index set of \mathbf{xt} . When t is exactly equal to 1, the expression $\mathbf{z}(1:(t-1))$ returns an empty matrix, which indicates to ROME an empty information index set. The next line stores the constructed decision rule into the array \mathbf{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

$$\begin{aligned}
 y_1(\tilde{\mathbf{z}}) &= x_1(\tilde{\mathbf{z}}) - \tilde{z}_1 \\
 y_t(\tilde{\mathbf{z}}) &= y_{t-1}(\tilde{\mathbf{z}}) + x_t(\tilde{\mathbf{z}}) - \tilde{z}_t \quad \forall t \in \{2, \dots, T\}
 \end{aligned}$$

Assuming that the programming variables \mathbf{x} and \mathbf{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.

```

45 rome_constraint(y(1) == x(1) - z(1)); % period 1 inventory balance
46 for t = 2:T % iterate over each period
47     rome_constraint(y(t) == y(t-1) + x(t) - z(t)); % period t inventory balance
48 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

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

We may express this set of constraints in ROME as

```

51 rome_constraint(x >= 0); % order quantity lower limit
52 rome_constraint(x <= xMax); % order quantity 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 [12]) as 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

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

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

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

We notice that the RHS of the simplified inequality can be computed as a function of the model parameters. On the LHS, we have a term which is the expected positive part of an LDR. In a previous work (Goh and Sim [27]), 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 [27] 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

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

Code Segment 5: Modeling the fill rate constraint.

Notice that the MATLAB operator `.*` represents an element-wise (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

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

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

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

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
```

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. Though 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 with 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 [26] contains a detailed description of how to use the `Pattern` option, and Appendix A 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 [35, 36] for a more complete introduction to the problem. An Activity-on-Arc (AOA) project network is a representation of a project in an 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 [30] 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 [52]. PERT makes several strong modeling assumptions, and has consequently come under strong criticism (e.g. Roman [43], van Slyke [54]). An alternative to PERT, which avoids the distributional assumption on the activity times, is to formulate the problem as a robust optimization problem (Cohen et al. [17], Goh et al. [25]) 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, which have uncertain completion times represented by an uncertainty vector $\tilde{z} \in \mathbb{R}^N$. Again, we do not presume exact knowledge of the actual distribution of \tilde{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, z_L and the most pessimistic activity times, z_H . These parameters form lower and upper bounds of \tilde{z} respectively.

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

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	z_L	Optimistic activity completion time
zH	N x 1	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	\boldsymbol{c}	Crashing cost per unit time
u	N x 1	\boldsymbol{u}	Crashing limit
B	1 x 1	B	Project budget

Table 4.2: List of programming variables and their associated model parameters for the project crashing problem

4.2.3 Model

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

$$\mathbb{F} = \{ \mathbb{P} : \mathbb{P}(\boldsymbol{z}_L \leq \tilde{\boldsymbol{z}} \leq \boldsymbol{z}_H) = 1, \mathbb{E}_{\mathbb{P}}(\tilde{\boldsymbol{z}}) = \boldsymbol{\mu}, \text{Cov}_{\mathbb{P}}(\tilde{\boldsymbol{z}}) = \boldsymbol{\Sigma} \},$$

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 $y_k(\tilde{\boldsymbol{z}})$ 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

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

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

$$I_y^1 = \emptyset \quad \text{and} \quad I_y^k = \mathcal{P}(l) \cup \bigcup_{k' \in \mathcal{P}(l)} I_y^{k'}, \quad (4.1)$$

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.

```

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

```

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

Here, `prioractivities` is a function which implements recursion (4.1), and returns the array `ind`, which contain the indices of the dependent activities. Appendix B details the code for its implementation. The next line then uses `ind` to index into the uncertainty vector `z` to construct the dependencies of the k^{th} 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

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

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

The time evolution of the project is represented by the inequality

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

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

```

61 for k = 1:N % iterate over each activity
62     ii = find(A(:, k) == -1); % activity k leaves this node
63     jj = find(A(:, k) == 1); % activity k enters this node
64     rome_constraint(x(jj) - x(ii) >= pos(z(k) - y(k))); % make constraint
65 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 $\mathbf{0} \leq \mathbf{y}(\tilde{\mathbf{z}}) \leq \mathbf{u}$, secondly the constraint that all times should be nonnegative, which is represented by $\mathbf{x}(\tilde{\mathbf{z}}) \geq \mathbf{0}$, and finally the requirement that the total crashing cost must be within the budget, which is represented by $\mathbf{c}'\mathbf{y}(\tilde{\mathbf{z}}) \leq B$. These inequalities are easily expressed in ROME as

```

68 rome_box(y, 0, u);           % crash limit constraints
69 rome_constraint(x >= 0);     % nonnegative time constraints
70 rome_constraint(c' * y <= B); % project 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, $\sup_{\mathbb{P} \in \mathbb{F}} \mathbb{E}_{\mathbb{P}}(\mathbf{x}_M(\tilde{\mathbf{z}}))$.

This can be expressed in ROME using the code segment

```

73 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 which 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 \mathbf{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

```

82 Nsims = 10000; % number of simulation runs
83 expenditure = zeros(Nsims, 1); % allocate array for expenditure
84 for ii = 1:Nsims
85     z_vals = zL + (zH - zL) .* rand(N, 1); % simulate activity times
86     expenditure(ii) = c' * y_sol.insert(z_vals); % instantiate crash costs
87 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, which 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 $(9.6072, 9.6125)$, while 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 of expected project crashing cost, subject to a constraint 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. [17].

4.3 Robust Portfolio Selection

4.3.1 Description

In this section, we consider the problem of robust portfolio selection. Markowitz [39, 40] 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 [50] and Chamberlain [13]).

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 [42]. CVaR satisfies many desirable properties of a risk measure, qualifying it as a *coherent* measure of risk (Artzner et al. [2]). 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 pre-specified 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 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 which 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 which has a mean return above τ , and minimizes the worst β -CVaR over all uncertainty distributions within \mathbb{F} . Table 4.3 lists several programming variables and the model parameters which they represent. We assume for the rest of this discussion that these variables have been declared and assigned appropriate numerical values, as we will use them within various code segments later.

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

Table 4.3: List of programming variables and their associated model parameters for the portfolio optimization problem

4.3.3 Model

In this model, the portfolio manager’s decision is a vector $\mathbf{x} \in \mathbb{R}^N$, with the i^{th} component representing the fraction of the net wealth to be invested in asset i . Since 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

$$\alpha_\beta(\mathbf{x}) = \min \left\{ v \in \mathbb{R} : \mathbb{P} \left(\tilde{\ell}(\mathbf{x}) \leq v \right) \geq \beta \right\}.$$

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

$$\text{CVaR}_\beta(\mathbf{x}) = \mathbb{E}_{\mathbb{P}} \left(\tilde{\ell}(\mathbf{x}) \mid \tilde{\ell}(\mathbf{x}) \geq \alpha_\beta(\mathbf{x}) \right).$$

Rockafellar and Uryasev [42, Theorem 1] show that the expression CVaR can be written in a more amenable form for optimization,

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

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

$$\text{CVaR}_\beta(\mathbf{x}) = \min_{v \in \mathbb{R}} \left\{ v + \frac{1}{1 - \beta} \sup_{\mathbb{P} \in \mathbb{F}} \mathbb{E}_{\mathbb{P}} \left(\left(\tilde{\ell}(\mathbf{x}) - v \right)^+ \right) \right\}. \quad (4.2)$$

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

$$\begin{aligned} \min_{\mathbf{x}} \quad & \text{CVaR}_\beta(-\tilde{\mathbf{r}}'\mathbf{x}) \\ \text{s.t.} \quad & \boldsymbol{\mu}'\mathbf{x} \geq \tau \\ & \mathbf{e}'\mathbf{x} = 1 \\ & \mathbf{x} \geq \mathbf{0}. \end{aligned} \quad (4.3)$$

The first constraint represents the requirement that the optimized portfolio must have a mean return above τ . The second constraint is simply a normalization. Since \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

```

9 function x_sol = optimizeportfolio(N, mu, Sigma, beta, tau)
10     h = rome_begin('Portfolio Optimization'); % begin the ROME environment
11
12     newvar r(N) uncertain; % declare r as an uncertainty
13     r.set_mean(mu); % set mean

```

```

14 r.Covar = Sigma;           % set covariance
15
16 newvar x(N) nonneg;        % declare a nonnegative variable x
17 rome_minimize(CVaR(-r' * x, beta)); % objective: minimize CVaR
18 rome_constraint(mu' * x >= tau);   % mean return must exceed tau
19 rome_constraint(sum(x) == 1);      % x must sum to 1
20
21 h.solve_deflected;        % solve the model
22 if(isinf(h.objective)) % check for infeasibility / unboundedness
23     x_sol = [];           % assign an empty matrix
24 else
25     x_sol = h.eval(x); % get the optimal solution
26 end
27 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

$$CV(\tau) \equiv \frac{\sqrt{\mathbf{x}^*(\tau)' \boldsymbol{\Sigma} \mathbf{x}^*(\tau)}}{\boldsymbol{\mu}' \mathbf{x}^*(\tau)}. \quad (4.4)$$

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.

```

14 Npts = 200;                % number of points in to plot
15 cv = zeros(Npts, 1);      % allocate result array
16 tau_arr = linspace(0, tH, Npts); % array of target means to test
17
18 for ii = 1:Npts
19     % Find the CVaR-optimal portfolio
20     x_sol = optimizeportfolio(N, mu, Sigma, beta, tau_arr(ii));
21
22     % Store the coefficients of variation
23     if isempty(x_sol) cv(ii) = Inf;
24     else cv(ii) = sqrt(x_sol' * Sigma * x_sol) / (mu' * x_sol);
25     end
26 end
27
28 plot(tau_arr, cv);        % plot CV against tau

```

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 which 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, Sim, and Sun [14]). 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 \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 $\mathbf{y}^k \in \mathcal{Y}(m_k, N, I_k), \forall k \in [K]$, where \mathcal{Y} is the set of nonanticipative decision rules, as defined in (3.1).

The general model which we consider, is then:

$$\begin{aligned}
 Z_{GEN}^* = & \min_{\mathbf{x}, \{\mathbf{y}^k(\cdot)\}_{k=1}^K} \mathbf{c}^{0'} \mathbf{x} + \sup_{\mathbb{P} \in \mathbb{F}} \mathbb{E}_{\mathbb{P}} \left(\sum_{k=1}^K \mathbf{d}^{0,k'} \mathbf{y}^k(\tilde{\mathbf{z}}) \right) \\
 \text{s.t.} & \mathbf{c}^l \mathbf{x} + \sup_{\mathbb{P} \in \mathbb{F}} \mathbb{E}_{\mathbb{P}} \left(\sum_{k=1}^K \mathbf{d}^{l,k'} \mathbf{y}^k(\tilde{\mathbf{z}}) \right) \leq b_l \quad \forall l \in [M] \\
 & \mathbf{T}(\tilde{\mathbf{z}}) \mathbf{x} + \sum_{k=1}^K \mathbf{U}^k \mathbf{y}^k(\tilde{\mathbf{z}}) = \mathbf{v}(\tilde{\mathbf{z}}) \\
 & \underline{\mathbf{y}}^k \leq \mathbf{y}^k(\tilde{\mathbf{z}}) \leq \overline{\mathbf{y}}^k \quad \forall k \in [K] \\
 & \mathbf{x} \geq \mathbf{0} \\
 & \mathbf{y}^k \in \mathcal{Y}(m_k, N, I_k) \quad \forall k \in [K].
 \end{aligned} \tag{5.1}$$

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 $\overline{\mathbf{y}}^k$ and $\underline{\mathbf{y}}^k$ are also model parameters which are possibly infinite component-wise.

Notice that in ROME, we do not solve (5.1) exactly, but instead solve for optimized decision rules residing in structured subsets of \mathcal{Y} , corresponding to a restriction of (5.1). This is because while problem (5.1) is quite general, it is also unfortunately computationally intractable in most cases (see Ben Tal et al. [5] or Shapiro and Nemirovski [48] for a more detailed discussion). The set of LDRs, \mathcal{L} , defined in (3.2), 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.

Acknowledgements

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

References

- [1] K. J. Arrow, T. Harris, and J. Marschak. Optimal inventory policy. *Econometrica*, 19(3):250–272, 1951.
- [2] P. Artzner, F. Delbaen, J.-M. Eber, and D. Heath. Coherent measures of risk. *Mathematical Finance*, 9(3):203–228, 1999.
- [3] M. Baotic and M. Kvasnica. CPLEXINT - MATLAB interface for the CPLEX solver. <http://control.ee.ethz.ch/~hybrid/cplexint.php>, June 2006.
- [4] A. Ben-Tal, S. Boyd, and A. Nemirovski. Extending scope of robust optimization: Comprehensive robust counterparts of uncertain problems. *Mathematical Programming, Series B*, 107:63–89, February 2006.
- [5] A. Ben-Tal, A. Goryashko, E. Guslitzer, and A. Nemirovski. Adjustable robust solutions of uncertain linear programs. *Mathematical Programming*, 99:351–376, 2004.
- [6] A. Ben-Tal and A. Nemirovski. Robust convex optimization. *Mathematics of Operations Research*, 23(4):769–805, 1998.
- [7] D. Bertsimas, D. A. Iancu, and P. A. Parrilo. Optimality of affine policies in multi-stage robust optimization. *Working Paper*, 2010.
- [8] D. Bertsimas and M. Sim. The price of robustness. *Operations Research*, 52(1):35–53, 2004.

- [9] T. Boyaci and G. Gallego. Serial production/distribution systems under service constraints. *Manufacturing & Service Operations Management*, 3(1):43–50, 2001.
- [10] M. Breton and S. El Hachem. Algorithms for the solution of stochastic dynamic minimax problems. *Computational Optimization and Applications*, 4:317–345, 1995.
- [11] A. Brooke, D. Kendrick, A. Meeraus, and R. Raman. *GAMS Language Guide*. GAMS Development Corporation, 1997. Release 2.25.
- [12] G. Cachon and C. Terwiesch. *Matching Supply with Demand: An Introduction to Operations Management*. McGraw Hill, 2009.
- [13] G. Chamberlain. A characterization of the distributions that imply mean-variance utility functions. *Journal of Economic Theory*, 29(1):185–201, Feb 1983.
- [14] X. Chen, M. Sim, and P. Sun. A robust optimization perspective on stochastic programming. *Operations Research*, 55(6):1058–1071, 2007.
- [15] 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.
- [16] X. Chen and Y. Zhang. Uncertain linear programs: Extended affinely adjustable robust counterparts. *Operations Research*, 2009.
- [17] I. Cohen, B. Golany, and A. Shtub. The stochastic time-cost tradeoff problem: A robust optimization approach. *Networks*, 49(2):175–188, 2007.
- [18] E. Delage and Y. Ye. Distributionally robust optimization under moment uncertainty with application to data-driven problems. *Operations Research*, forthcoming.
- [19] J. Dupačová. The minimax approach to stochastic programming and an illustrative application. *Stochastics*, 20(1):73–88, January 1987.
- [20] 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.
- [21] R. Fourer, D. M. Gay, and B. W. Kernighan. A modeling language for mathematical programming. *Management Science*, 36(5):519–554, May 1990.
- [22] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, Brooks/Cole Publishing, Pacific Grove, CA, 2002.
- [23] R. Fourer and L. Lopes. A management system for decompositions in stochastic programming. *Annals of Operations Research*, 142:99–118, 2006.
- [24] R. Fourer and L. Lopes. StAMPL: A filtration-oriented modeling tool for multistage stochastic recourse problems. *INFORMS Journal on Computing*, 21(2):242–256, Spring 2009.

- [25] J. Goh, N. G. Hall, and M. Sim. Robust optimization strategies for total cost control in project management. *Working Paper, NUS Business School.*, 2010.
- [26] J. Goh and M. Sim. *User's Guide to ROME*. http://www.robustopt.com/references/ROME_Guide_1.0.pdf, Sept 2009.
- [27] J. Goh and M. Sim. Distributionally robust optimization and its tractable approximations. *Operations Research*, forthcoming.
- [28] 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.
- [29] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming (web page and software). <http://stanford.edu/~boyd/cvx>, June 2009.
- [30] W. Herroelen and R. Leus. Project scheduling under uncertainty: Survey and research potentials. *European Journal of Operations Research*, 165:289–306, 2005.
- [31] K. Holmström. The TOMLAB optimization environment in MATLAB. *Advanced Modeling and Optimization*, 1:47–69, 1999.
- [32] IBM. IBM ILOG CPLEX. <http://www-01.ibm.com/software/integration/optimization/cplex/>.
- [33] G. D. Johnson and H. E. Thompson. Optimality of myopic inventory policies for certain dependent demand processes. *Management Science*, 21(11):1303–1307, 1975.
- [34] M. Kaut, A. King, and T. H. Hultberg. A C++ modelling environment for stochastic programming. Technical report, IBM, 2008.
- [35] H. Kerzner. *Project Management: A Systems Approach to Planning, Scheduling, and Controlling*. Wiley, Hoboken, NJ, 10th edition, 2009.
- [36] T. D. Klastorin. *Project Management: Tools and Trade-Offs*. Wiley, Hoboken, NJ, 1st edition, 2004.
- [37] J. Löfberg. YALMIP : a toolbox for modeling and optimization in MATLAB. In *IEEE International Symposium on Computer Aided Control Systems Design*, 2004.
- [38] 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.
- [39] H. M. Markowitz. Portfolio selection. *Journal of Finance*, 7:77–91, 1952.
- [40] H. M. Markowitz. *Portfolio selection: Efficient diversification of investments*. John Wiley & Sons, New York, 1959.

- [41] MOSEK ApS. The MOSEK optimization software. <http://www.mosek.com>.
- [42] R. T. Rockafellar and S. Uryasev. Optimization of conditional value-at-risk. *Journal of Risk*, 2:493–517, 2000.
- [43] D. D. Roman. The pert system: An appraisal of program evaluation review technique. *The Journal of the Academy of Management*, 5(1):57–65, April 1962.
- [44] C.-T. See and M. Sim. Robust approximation to multi-period inventory management. *Operations Research*, forthcoming, 2009.
- [45] 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.
- [46] A. Shapiro and S. Ahmed. On a class of minimax stochastic programs. *SIAM Journal on Optimization*, 14(4):1237–1249, 2004.
- [47] A. Shapiro and A. Kleywegt. Minimax analysis of stochastic programs. *Optimization Methods and Software*, 17(3):523–542, 2002.
- [48] 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.
- [49] A. L. Soyster. Convex programming with set-inclusive constraints and applications to inexact linear programming. *Operations Research*, 21(5):1154–1157, 1973.
- [50] J. Tobin. Liquidity preference as behavior toward risk. *Review of Economic Studies*, 25:65–85, 1958.
- [51] K. Toh, M. Todd, and R. Tütüncü. Sdpt3 — a matlab software package for semidefinite programming. *Optimization Methods and Software*, 11:545–581, 1999.
- [52] U.S. Navy. Pert summary report, phase I. Technical report, Special Projects Office, Bureau of Naval Weapons, July 1958.
- [53] C. Valente, G. Mitra, M. Sadki, and R. Fourer. Extending algebraic modelling languages for stochastic programming. *INFORMS Journal on Computing*, 21(1):107–122, Winter 2009.
- [54] R. M. van Slyke. Monte carlo methods and the PERT problem. *Operations Research*, 11(5):839–860, 1963.
- [55] A. Veinott. Optimal policy for a multi-product, dynamic, nonstationary inventory problem. *Management Science*, 12(3):206–222, 1965.
- [56] J. Žáčková. On minimax solution of stochastic linear programming problems. *Časopis pro Pěstování Matematiky*, 91:423–430, 1966.

Appendix A Details for Inventory Management Example

For reference, in this section we present the full algebraic and ROME models for the inventory management example.

$$\begin{aligned}
 \min_{\mathbf{x}(\cdot), \mathbf{y}(\cdot)} \quad & \sup_{\mathbb{P} \in \mathbb{F}} \mathbb{E}_{\mathbb{P}} (\mathbf{c}' \mathbf{x}(\tilde{\mathbf{z}}) + \mathbf{h}' (\mathbf{y}(\tilde{\mathbf{z}}))^+) \\
 \text{s.t.} \quad & \sup_{\mathbb{P} \in \mathbb{F}} \mathbb{E}_{\mathbb{P}} (y_t(\tilde{\mathbf{z}})^-) \leq (1 - \beta_t) \mu_t \quad \forall t \in [T] \\
 & y_1(\tilde{\mathbf{z}}) = x_1(\tilde{\mathbf{z}}) - \tilde{z}_1 \\
 & y_t(\tilde{\mathbf{z}}) = y_{t-1}(\tilde{\mathbf{z}}) - x_t(\tilde{\mathbf{z}}) - \tilde{z}_t \quad \forall t \in \{2, \dots, T\} \\
 & \mathbf{0} \leq \mathbf{x}(\tilde{\mathbf{z}}) \leq \mathbf{x}^{MAX} \\
 & x_t \in \mathcal{L}(1, T, [t-1]) \quad \forall t \in [T] \\
 & y_t \in \mathcal{L}(1, T, [t]) \quad \forall t \in [T].
 \end{aligned} \tag{A.1}$$

The ROME code is

```

1 % inventory_fillrate_example.m
2 % Script to model robust fillrate-constrained inventory management.
3 %
4 % 1. Solves a linearized version of the problem using LDRs
5
6 % model parameters
7 T = 5; % planning horizon
8 c = 1 * ones(T, 1); % order cost rate
9 h = 2 * ones(T, 1); % holding cost rate
10 beta = 0.95 * ones(T, 1); % minimum fillrate in each period
11 xMax = 60 * ones(T, 1); % maximum order quantity in each period
12 alpha = 0.5; % temporal autocorrelation factor
13 L = alpha * tril(ones(T), -1) + eye(T); % autocorrelation matrix
14
15 % numerical uncertainty parameters
16 zMax = 60 * ones(T, 1); % maximum demand in each period
17 mu = 30 * ones(T, 1); % mean demand in each period
18 S = 20 * (L * L'); % temporal demand covariance
19
20 % begin model
21 hmodel = rome_begin('Inventory Management');
22
23 % declare uncertainties
24 newvar z(T) uncertain; % declare an uncertain demand
25 rome_constraint(z >= 0);
26 rome_constraint(z <= zMax); % set distributional support
27 z.set_mean(mu); % set mean
28 z.Covar = S; % set covariance
29
30 % define LDRs
31 newvar x(T) empty; % allocate an empty variable array
32 for t = 1:T % iterate over each period
33     newvar xt(1, z(1:(t-1))) linearrule; % construct the period t decision rule

```

```

34     x(t) = xt;                                % assign it to the tth entry of x
35 end
36
37 % define LDRs
38 newvar y(T) empty;                            % allocate an empty variable array
39 for t = 1:T                                    % iterate over each period
40     newvar yt(1, z(1:t)) linearrule;          % construct the period t decision rule
41     y(t) = yt;                                % assign it to the tth entry of y
42 end
43
44 % inventory balance constraints
45 rome_constraint(y(1) == x(1) - z(1));          % period 1 inventory balance
46 for t = 2:T                                    % iterate over each period
47     rome_constraint(y(t) == y(t-1) + x(t) - z(t)); % period t inventory balance
48 end
49
50 % order quantity constraints
51 rome_constraint(x >= 0); % order quantity lower limit
52 rome_constraint(x <= xMax); % order quantity upper limit
53
54 % fill rate constraint
55 rome_constraint(mean(neg(y)) <= mu - mu .* beta); % fill rate constraint
56
57 % objective
58 rome_minimize(c'*mean(x) + h'*mean(pos(y))); % model objective
59
60 % solve and display optimal objective
61 hmodel.solve; % solve using LDR
62 hmodel.solve_deflected; % solve using BDLDR
63
64 hmodel.objective % display optimal objective
65 x_sol = hmodel.eval(x) % display optimal ordering decision rule

```

Code Segment 15: ROME code for the fill rate constrained robust inventory management problem

An equivalent vectorized code in ROME that is more concise but less intuitive, is

```

1 % begin model
2 hmodel = rome_begin('Vectorized Inventory Management');
3
4 % declare uncertainties
5 newvar z(T) uncertain; % declare an uncertain demand
6 rome_box(z, 0, zMax) % set distributional support
7 z.set_mean(mu); % set mean
8 z.Covar = S; % set covariance
9
10 % define dependency patterns for LDRs
11 pX = [tril(true(T)), false(T, 1)];
12 pY = [true(T, 1), tril(true(T))];
13

```

```

14 % define LDRs
15 newvar x(T, z, 'Pattern', pX) linearrule;
16 newvar y(T, z, 'Pattern', pY) linearrule;
17
18 % inventory balance constraints
19 D = eye(T) - diag(ones(T-1, 1), -1); % make a differencing matrix
20 rome_constraint(D*y == x - z); % inventory balance constraint
21
22 % order quantity constraints
23 rome_box(x, 0, xMax);
24
25 % fill rate constraint
26 rome_constraint(mean(neg(y)) <= mu - mu .* beta); % fill rate constraint
27
28 % objective
29 rome_minimize(c'*mean(x) + h'*mean(pos(y))); % model objective
30
31 % solve and display optimal objective
32 hmodel.solve; % solve using LDR
33 % hmodel.solve_deflected; % solve using BDLDR
34
35 hmodel.objective % display optimal objective
36 x_sol = hmodel.eval(x) % display optimal ordering decision rule

```

Code Segment 16: Vectorized ROME code for the fill rate constrained robust inventory management problem

This code uses the `rome_box` contraction also used in the other examples to combine the upper and lower constraints into a single statement. It also makes comparatively heavier use of MATLAB array construction and manipulation functions such as `eye` (constructs an identity matrix), `ones` (constructs an array of all ones), `diag` (constructs a diagonal matrix), `tril` (extracts the lower triangular part of a matrix), `true` and `false` (constructs logical 0-1 matrices).

Appendix B Details for Project Crashing Example

The full algebraic model for the project crashing problem is

$$\begin{aligned}
 \min_{\mathbf{x}(\cdot), \mathbf{y}(\cdot)} \quad & \sup_{\mathbb{P} \in \mathbb{F}} \mathbb{E}_{\mathbb{P}}(x_M(\tilde{\mathbf{z}})) \\
 \text{s.t.} \quad & x_j(\tilde{\mathbf{z}}) - x_i(\tilde{\mathbf{z}}) \geq (\tilde{z}_k - y_k(\tilde{\mathbf{z}}))^+ \quad \forall k \in [N], A_{ik} = -1, A_{jk} = 1 \\
 & \mathbf{c}'\mathbf{y}(\tilde{\mathbf{z}}) \leq B \\
 & \mathbf{0} \leq \mathbf{y}(\tilde{\mathbf{z}}) \leq \mathbf{u} \\
 & \mathbf{x} \geq \mathbf{0} \\
 & x_i \in \mathcal{L}(1, N, I_x^i) \quad \forall i \in [M] \\
 & y_k \in \mathcal{L}(1, N, I_y^k) \quad \forall k \in [N].
 \end{aligned} \tag{B.1}$$

We consider a numerical instance of a project crashing problem with an AOA project network depicted in Figure B.1. The corresponding modeling code in ROME is

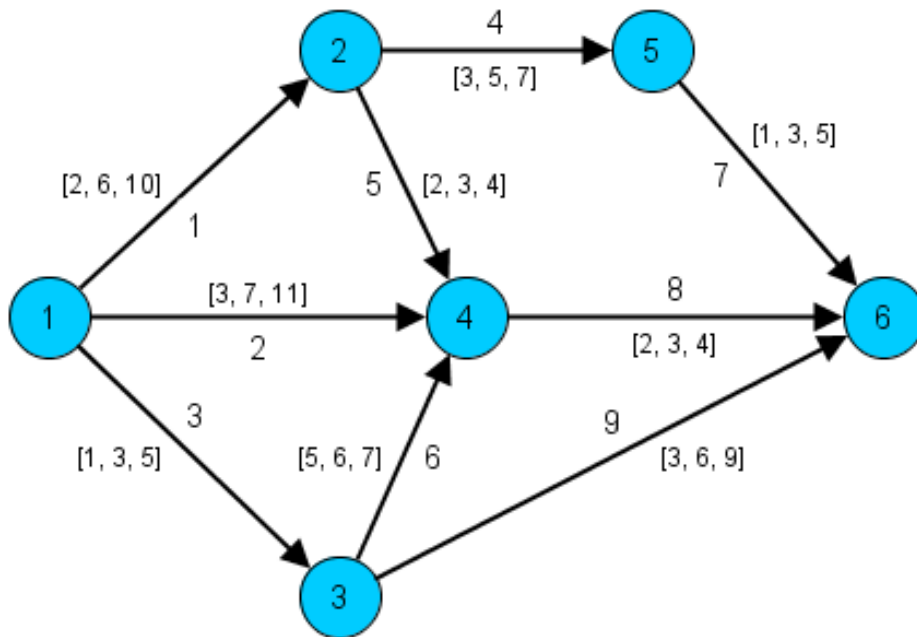


Figure B.1: Example AOA Project Network. Arcs are labeled with activity indices and a bracketed triplet: [optimistic activity time, mean activity time, pessimistic activity time].

```

1 % project_crashing_example
2 % Solves a project crashing example
3
4 % parameters
5 M = 6; % number of nodes
6 N = 9; % number of arcs
7 A = spalloc(M, N, 2*N); % allocate space for matrix
8

```

```

9 % build the incidence matrix
10 A([1, 2], 1) = [-1; 1];
11 A([1, 4], 2) = [-1; 1];
12 A([1, 3], 3) = [-1; 1];
13 A([2, 5], 4) = [-1; 1];
14 A([2, 4], 5) = [-1; 1];
15 A([3, 4], 6) = [-1; 1];
16 A([5, 6], 7) = [-1; 1];
17 A([4, 6], 8) = [-1; 1];
18 A([3, 6], 9) = [-1; 1];
19
20 zL = [ 2, 3, 1, 3, 2, 5, 1, 2, 3]'; % optimistic activity times
21 mu = [ 6, 7, 3, 5, 3, 6, 3, 3, 6]'; % mean activity times
22 zH = [10, 11, 5, 7, 4, 7, 5, 4, 9]'; % pessimistic activity times
23
24 Sigma = diag((zH - zL).^2/12); % covariance matrix
25 c = ones(N, 1); % unit crashing cost
26 B = 10; % project budget
27 u = zL; % crash limit
28
29 % ROME model
30 hmodel = rome_begin('Project Crashing Example');
31
32 % uncertainties
33 newvar z(N) uncertain; % Declare uncertainties;
34 rome_box(z, zL, zH); % Set support;
35 z.set_mean(mu); % Set mean;
36 z.Covar = Sigma; % Set covariance;
37
38 % Decision Rules
39 % y: crash amounts
40 newvar y(N) empty; % allocate an empty variable array
41 for k = 1:N % iterate over each activity
42     ind = prioractivities(k, A); % get indices of dependent activities
43     newvar yk(1, z(ind)) linearrule; % construct the decision rule
44     y(k) = yk; % assign it to the kth entry of y
45 end
46
47 % x: node times
48 newvar x(M) empty; % allocate an empty variable array
49 for ii = 1:M % iterate over each node
50     if(ii < M)
51         k = find(A(ii, :) < 0, 1); % find any activity that exits this node
52         ind = prioractivities(k, A); % get indices of dependent activities
53         newvar xi(1, z(ind)) linearrule; % construct the decision rule
54     else
55         newvar xi(1, z) linearrule; % construct the decision rule
56     end

```

```

57 x(ii) = xi;
58 end
59
60 % time evolution constraint
61 for k = 1:N % iterate over each activity
62 ii = find(A(:, k) == -1); % activity k leaves this node
63 jj = find(A(:, k) == 1); % activity k enters this node
64 rome_constraint(x(jj) - x(ii) >= pos(z(k) - y(k))); % make constraint
65 end
66
67 % other constraints
68 rome_box(y, 0, u); % crash limit constraints
69 rome_constraint(x >= 0); % nonnegative time constraints
70 rome_constraint(c' * y <= B); % project budget constraint
71
72 % objective: minimize worst-case mean completion time
73 rome_minimize(mean(x(M)));
74
75 % hmodel.solve; % solve using LDRs
76 hmodel.solve_deflected; % solve using BDLDRs
77 y_sol = hmodel.eval(y); % extract crash amount decision rule
78 hmodel.objective % display optimization objective
79
80 % simulate
81 rand('state', 1); % set seed of random number generator
82 Nsims = 10000; % number of simulation runs
83 expenditure = zeros(Nsims, 1); % allocate array for expenditure
84 for ii = 1:Nsims
85 z_vals = zL + (zH - zL) .* rand(N, 1); % simulate activity times
86 expenditure(ii) = c' * y_sol.insert(z_vals); % instantiate crash costs
87 end

```

Code Segment 17: ROME code for the robust project crashing problem

In the construction of the decision rules, we invoke the helper function `prioractivities`, which is a recursive numerical routine, implemented in the following code

```

1 % PRIORACTIVITIES
2 % Given an AOA incidence matrix A, and an arc index k, returns the indices
3 % of all activities that are predecessors of k
4
5 function ind = prioractivities(k, A)
6     ind = []; % make an empty array
7     ind = rec_prioractivities(k, A, ind); % call a recursive helper
8
9
10 % REC_PRIORACTIVITIES
11 % Helper function that recursively gets the indices of the prior activities
12
13 function ind = rec_prioractivities(k, A, ind)

```

```

14 % finds the unique exit node for this arc
15 exitnode = find(A(:, k) < 0);
16
17 % terminal condition
18 if(exitnode == 1)
19     return;
20 else
21     % add the immediate predecessor activities
22     immediate_prior_activites = find(A(exitnode, :) > 0);
23     ind = [ind, immediate_prior_activites];
24
25     % recurse over all immediate prior activites
26     for l = immediate_prior_activites
27         ind = rec_prioractivities(l, A, ind);
28     end
29 end

```

Code Segment 18: Implementation details for `prioractivities`. This returns the index of all activities prior to the current (k^{th}) activity.

Appendix C Details for Portfolio Optimization Example

For completeness, we repeat the algebraic formulation of the portfolio CVaR optimization problem for a fixed parameter τ , which is

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

The portfolio optimization code used in this example comprises a script, and two functions, which we list here. The `optimizeportfolio` function is the key driver which contains the ROME model.

```

1 % OPTIMIZEPORTFOLIO(N, mu, Sigma, beta, tau)
2 % Computes the beta-CVaR optimal portfolio
3 %   N      : Number of assets
4 %   mu     : Mean of asset returns
5 %   Sigma  : Covariance matrix of asset returns
6 %   beta   : CVaR level
7 %   tau    : Target return
8
9 function x_sol = optimizeportfolio(N, mu, Sigma, beta, tau)
10     h = rome_begin('Portfolio Optimization'); % begin the ROME environment
11
12     newvar r(N) uncertain; % declare r as an uncertainty
13     r.set_mean(mu);        % set mean
14     r.Covar = Sigma;      % set covariance
15
16     newvar x(N) nonneg;    % declare a nonnegative variable x
17     rome_minimize(CVaR(-r' * x, beta)); % objective: minimize CVaR
18     rome_constraint(mu' * x >= tau);   % mean return must exceed tau
19     rome_constraint(sum(x) == 1);      % x must sum to 1
20
21     h.solve_deflected; % solve the model
22     if(isinf(h.objective)) % check for infeasibility / unboundedness
23         x_sol = []; % assign an empty matrix
24     else
25         x_sol = h.eval(x); % get the optimal solution
26     end
27     rome_end; % end the ROME environment

```

Code Segment 19: Function to compute the optimal portfolio for fixed τ

It calls a custom function, `CVaR`, which separately models the β -CVaR, for increased code modularity.

```

1 % CVaR(loss, beta)
2 % Computes the beta-CVaR
3 %   loss   : ROME variable representing the portfolio loss
4 %   beta   : CVaR-level
5

```

```

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 20: Function to compute the β -CVaR

Finally, the script, `plotcvportfolio`, instantiates various parameters and iterates through a range of target returns. It concludes by plotting the coefficient of variation (CV) against τ .

```

1 % PLOTVCVPORTFOLIO
2 % Script which plots the coefficients of variation of the beta-CVaR
3 % optimized portfolio for different target mean returns, tau.
4
5 rand('state', 1); % fix seed of random number generator
6 tL = 0.05; % lower limit of target mean
7 tH = 0.15; % upper limit of target mean
8 N = 20; % number of assets
9 mu = unifrnd(tL, tH, N, 1); % randomly generate mu
10 A = unifrnd(tL, tH, N, N);
11 Sigma = A'*A; % randomly generate Sigma
12 beta = 0.95; % CVaR-level
13
14 Npts = 200; % number of points in to plot
15 cv = zeros(Npts, 1); % allocate result array
16 tau_arr = linspace(0, tH, Npts); % array of target means to test
17
18 for ii = 1:Npts
19     % Find the CVaR-optimal portfolio
20     x_sol = optimizeportfolio(N, mu, Sigma, beta, tau_arr(ii));
21
22     % Store the coefficients of variation
23     if isempty(x_sol) cv(ii) = Inf;
24     else cv(ii) = sqrt(x_sol' * Sigma * x_sol) / (mu' * x_sol);
25     end
26 end
27
28 plot(tau_arr, cv); % plot CV against tau
29 xlabel('\tau'); ylabel('CV'); % label axes
30 title('Plot of CV vs \tau'); % label graph

```

Code Segment 21: Script to compute coefficients of variation for a uniform sample of $\tau \in (\tau_L, \tau_H)$.

The output plot for this numerical example is shown in Figure C.1.

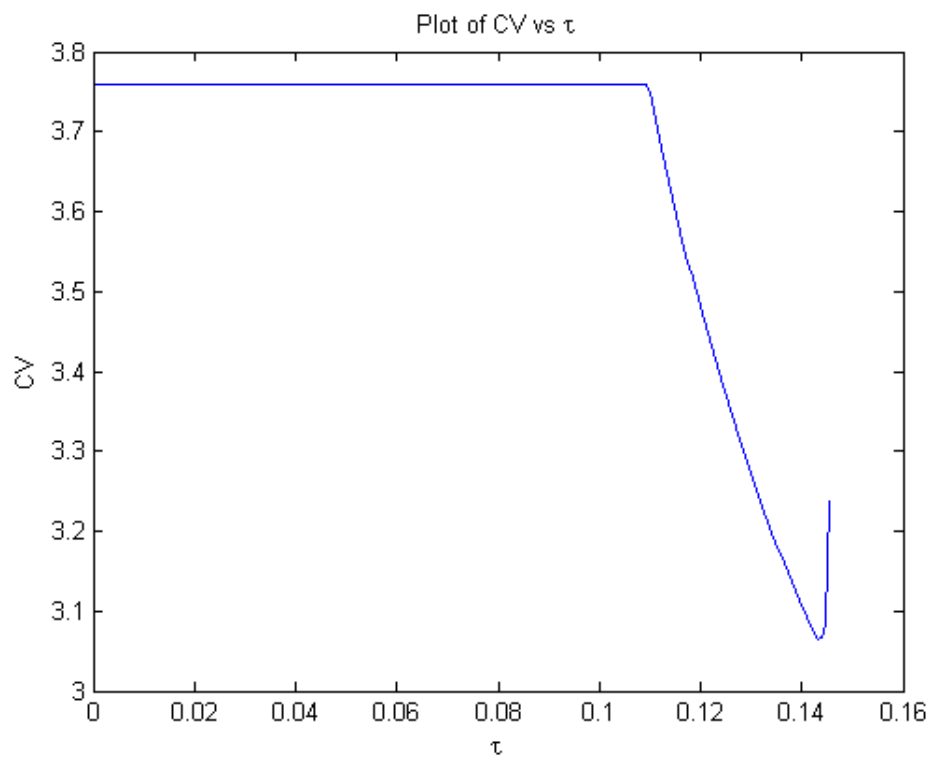


Figure C.1: Plot of the coefficient of variation of the CVaR-optimized portfolios against τ .