ROC++: Robust Optimization in C++

Phebe Vayanos,† Qing Jin,† and George Elissaios
†University of Southern California, CAIS Center for Artificial Intelligence in Society
{phebe.vayanos,qingjin}@usc.edu

Over the last two decades, robust optimization techniques have emerged as a very popular means to address
decision-making problems affected by uncertainty. Their success has been fueled by their attractive robustness
and scalability properties, by ease of modeling, and by the limited assumptions they need about the uncertain
parameters to yield meaningful solutions. Robust optimization techniques are available that can address
both single- and multi-stage decision-making problems involving real-valued and/or binary decisions, and
affected by both exogenous (decision-independent) and endogenous (decision-dependent) uncertain parameters. Robust optimization techniques rely on duality theory (potentially augmented with approximations)
to transform a semi-infinite optimization problem to a finite program (the “robust counterpart”). While
writing down the model for a robust optimization problem is usually a simple task, obtaining the robust
counterpart requires expertise in robust optimization. To date, very few solutions are available that can
facilitate the modeling and solution of such problems. This has been a major impediment to their being put
to practical use. In this paper, we propose ROC++, an open source C++ based platform for automatic robust
optimization, applicable to a wide array of single- and multi-stage robust problems with both exogenous and
endogenous uncertain parameters, that is easy to both use and extend. It also applies to certain classes of
stochastic programs involving continuously distributed uncertain parameters and decision-dependent informa-
tion discovery. Our platform naturally extends existing off-the-shelf deterministic optimization platforms
and offers ROPy, a Python interface in the form of a callable library. Our paper also proposes the ROB
file format that generalizes the LP file format to robust optimization. We showcase the modeling power
of ROC++ on several decision-making problems of practical interest. Our platform can help streamline the
modeling and solution of stochastic and robust optimization problems for both researchers and practitioners.
It comes with detailed documentation to facilitate its use and expansion. ROC++ can be downloaded from
https://sites.google.com/usc.edu/robust-opt-cpp/.

Key words: robust optimization, sequential decision-making under uncertainty, exogenous uncertainty,
endogenous uncertainty, decision-dependent uncertainty, decision-dependent information discovery.

1. Introduction

1.1. Background & Motivation

Decision-making problems involving uncertain parameters are faced routinely by individuals, firms, policy-makers, and governments. Uncertain parameters may correspond to prediction errors, measurement errors, or implementation errors, see e.g., Ben-Tal et al.
Prediction errors arise when some of the data elements have not yet materialized at the time of decision-making and must thus be predicted/estimated (e.g., future prices of stocks, future demand, or future weather). Measurement errors arise when some of the data elements (e.g., characteristics of raw materials) cannot be precisely measured (due to e.g., limitations of the technological devices available). Implementation errors arise when some of the decisions may not be implemented exactly as planned/recommended by the optimization (due to e.g., physical constraints).

If all decisions must be made before the uncertain parameters are revealed, the decision-making problem is referred to as static or single-stage. In contrast, if the uncertain parameters are revealed sequentially over time and decisions are allowed to adapt to the history of observations, the decision-making problem is referred to as adaptive or multi-stage.

In decision-making under uncertainty, the realizations of the uncertain parameters and their time of revelation may either be exogenous, being independent of the decision-maker’s actions, or they may be endogenous, being possible for the decision-maker to influence or control. To the best of our knowledge, this terminology was originally coined by Jonsbråten (1998). A decision-making problem involving uncertain parameters whose time of revelation is endogenous are said to involve decision-dependent information discovery.

Examples of decision-making problems involving exogenous uncertain parameters are: financial portfolio optimization (see e.g., Markowitz (1952)), inventory and supply-chain management (see e.g., Scarf (1958)), vehicle routing (Bertsimas and van Ryzin (1991)), unit commitment (see e.g., Takriti et al. (1996)), and option pricing (see e.g., Haarbrücker and Kuhn (2009)). Examples of decision-making problems involving endogenous uncertain parameters are: R&D project portfolio optimization (see e.g., Solak et al. (2010)), clinical trial planning (see e.g., Colvin and Maravelias (2008)), offshore oilfield exploration (see e.g., Goel and Grossman (2004)), best box and Pandora’s box problems (see e.g., Weitzman (1979)), and preference elicitation (see e.g., Vayanos et al. (2020)).

1.2. Stochastic Programming & Robust Optimization
Whether the decision-making problem is affected by exogenous and/or endogenous uncertain parameters, ignoring uncertainty altogether when deciding on the actions to take usually results in suboptimal or even infeasible actions. To this end, researchers in stochastic programming and robust optimization have devised optimization-based models and solution approaches that explicitly capture the uncertain nature of these parameters. These
frameworks model decisions as functions (decision rules) of the history of observations, capturing the adaptive and non-anticipative nature of the decision-making process.

Stochastic programming assumes that the distribution of the uncertain parameters is perfectly known, see e.g., Kall and Wallace (1994), Prékopa (1995), Birge and Louveaux (2000), and Shapiro et al. (2009). This assumption is well justified in many situations. For example, this is the case if the distribution is stationary and can be well estimated from historical data. If the distribution of the uncertain parameters is discrete, the stochastic program admits a deterministic equivalent that can be solved with off-the-shelf solvers potentially augmented with dedicated procedures such as Bender’s decomposition, see e.g., Benders (1962), the progressive hedging algorithm, see e.g., Rockafellar and Wets (1991), the L-shaped method, see e.g., Louveaux and Birge (2001), or stochastic dual dynamic programming, see e.g., Pereira and Pinto (1991), Shapiro (2011). If the distribution of the uncertain parameters is continuous, the reformulation of the uncertain optimization problem may or not be computationally tractable since even evaluating the objective function usually requires computing a high-dimensional integral. If this problem is not computationally tractable, discretization approaches (such as the sample average approximation) may be employed, see e.g., Shapiro et al. (2009). While discretization is practicable for smaller problems, it may be impracticable when applied to large and medium sized problems. Conversely, using only very few discretization points may result in solutions that are suboptimal or not possible to implement in practice. Over the last two decades, stochastic programming techniques have been extended to address problems involving endogenous uncertain parameters, see e.g., Goel and Grossman (2004, 2005, 2006), Goel et al. (2006), Gupta and Grossmann (2011), Tarhan et al. (2013), Colvin and Maravelias (2008, 2009, 2010). We refer the reader to Kall and Wallace (1994), Prékopa (1995), Birge and Louveaux (2000), and Shapiro et al. (2009) for in-depth reviews of the field of stochastic programming.

Robust optimization does not necessitate knowledge of the distribution of the uncertain parameters. Rather than modeling uncertainty by means of distributions, it assumes that the uncertain parameters belong in a so-called uncertainty set. The decision-maker then seeks to be immunized against all possible realizations of the uncertain parameters in this set. The robust optimization paradigm gained significant traction starting in the late 1990s and early 2000s following the works of Ben-Tal and Nemirovski (1999, 1998, 2000), Ben-Tal
et al. (2004), and Bertsimas and Sim (2003, 2004, 2006), among others. Over the last two decades, research on robust optimization has burgeoned, fueled by the limited assumptions it needs about the uncertain parameters to yield meaningful solutions, by its attractive robustness and scalability properties, and by ease of modelling, see e.g., Bertsimas et al. (2010), Gorissen et al. (2015).

Robust optimization techniques are available that can address both single- and multi-stage decision-making problems involving real-valued and/or binary decisions, and affected by exogenous and/or endogenous uncertain parameters. In the single-stage setting, robust optimization techniques rely on duality theory to transform a semi-infinite optimization problem to an equivalent finite program (the “robust counterpart”) that is solvable with off-the-shelf solvers, see e.g., Ben-Tal et al. (2009). In robust optimization, endogeneity of the realizations of the uncertain parameters is modelled by letting the uncertainty set depend on the decisions. In this case, the robust counterpart is a bilinear program. If the uncertainty set depends on binary decisions only, the bilinear terms involve only products of binary and real-valued decisions and can thus be linearized using standard methods, see e.g., Nohadani and Sharma (2018), Lappas and Gounaris (2018). In the multi-stage setting, the dualization step is usually preceded by an approximation step that transforms the multi-stage problem to a single-stage robust program. The idea is to restrict the space of the decisions based either on a decision rule approximation or a finite adaptability approximation. The decision rule approximation consists in restricting the adjustable decisions to those presenting e.g., linear, piecewise linear, or polynomial dependence on the uncertain parameters, see e.g., Ben-Tal et al. (2004), Kuhn et al. (2009), Bertsimas et al. (2011), Vayanos et al. (2012), Georghiou et al. (2015). The finite adaptability approximation consists in selecting a finite number of candidate strategies today and choosing the best of those strategies in an adaptive fashion once the uncertain parameters are revealed, see e.g., Bertsimas and Caramanis (2010), Hanasusanto et al. (2015). These ideas have also been extended to settings where the realizations of the uncertain parameters are endogenous, see Bertsimas and Vayanos (2017), and to settings with decision-dependent information discovery, see Vayanos et al. (2011, 2019). While writing down the model for a robust optimization problem is usually a simple task (akin to formulating a deterministic optimization problem), obtaining the robust counterpart is typically tedious and requires expertise in robust optimization, see Ben-Tal et al. (2009).
Robust optimization techniques have been extended to address certain classes of multi-stage stochastic programming problems involving continuously distributed uncertain parameters and affected by both exogenous uncertainty, see Kuhn et al. (2009), Bodur and Luedtke (2018), and endogenous uncertainty, see Vayanos et al. (2011). Compared to discretization based methods, robust optimization techniques applied to stochastic programs have the salient advantage that they do not require a discretization of the distribution of the uncertain parameters thus being guaranteed to return feasible solutions while presenting attractive scalability properties. In recent years, the field of distributionally robust optimization has burgeoned, which seeks to immunize decision-makers against ambiguity in the distribution of the uncertain parameters, see e.g., Wiesemann et al. (2014), Rahimian and Mehrotra (2019). Similarly to classical robust optimization, deterministic equivalent reformulations of distributionally robust optimization problems can be obtain based on duality theory.

Robust optimization techniques have been used successfully to address single-stage problems in inventory management (Ardestani-Jaafari and Delage (2016)), network optimization (Bertsimas and Sim (2003)), product pricing (Adida and Perakis (2006), Thiele (2009)), portfolio optimization (Goldfarb and Iyengar (2003)), and healthcare (Gupta et al. (2020), Bandi et al. (2018), Chan et al. (2018)). They have also been used to successfully tackle sequential problems in energy (Zhao et al. (2013), Jiang et al. (2014)), inventory and supply-chain management (Ben-Tal et al. (2005), Mamani et al. (2017)), network optimization (Atamtürk and Zhang (2007)), preference elicitation (Vayanos et al. (2020)), vehicle routing (Gounaris et al. (2013)), process scheduling (Lappas and Gounaris (2016)), and R&D project portfolio optimization (Vayanos et al. (2019)).

In spite of its success at addressing a diverse pool of problems in the literature, to date, very few platforms are available that can facilitate the modeling and solution of robust optimization problems, and those available can only tackle limited classes of robust problems. At the same time, and as mentioned above, reformulating such problems in a way that they can be solved by off-the-shelf solvers requires expertise. This is particularly true in the case of multi-stage problems and of problems affected by endogenous uncertainty.

In this paper, we fill this gap by proposing ROC++, a C++ based platform for modeling, approximating, automatically reformulating, and solving general classes of robust optimization problems. Our platform provides several modeling objects (decision variables,
uncertain parameters, constraints, optimization problems) and a suite of reformulation/approximation strategies (e.g., linear decision rule, finite adaptability, reformulation of robust problem as finite program) that can easily be extended to provide even more capabilities. It also leverages overloaded operators in C++ to allow for ease of modeling using a syntax similar to that of state-of-the-art deterministic optimization solvers like CPLEX\(^1\) or Gurobi.\(^2\) ROC++ leverages the power of C++ and in particular of polymorphism to code dynamic behavior, allowing the user to select their reformulation strategy at runtime. Finally, ROC++ comes with a Python library that provides most of the platform’s functionality. While our platform is not exhaustive, it provides a framework that is easy to update and expand and lays the foundation for more development to help facilitate research in, and real-life applications of, robust optimization.

1.3. Related Literature

Tools for Modelling and Solving Deterministic Optimization Problems. There exist many commercial and open-source tools for modeling and solving deterministic optimization problems. On the commercial front, the most popular solvers for conic (integer) optimization are Gurobi,\(^3\) IBM CPLEX Optimizer,\(^4\) Mosek,\(^5\) and FICO Xpress.\(^6\) On the open source side, they are GLPK,\(^7\) Cbc\(^8\) and Clp\(^9\) from COIN-OR,\(^10\) and SCIP.\(^11\) These solvers provide dedicated interfaces for C/C++, Python, or other commonly used programming languages. For example, Gurobi’s callable library can be accessed through C++, Python, and R among others while SCIP’s callable library can be accessed through C++. To streamline the modeling process and to facilitate switching between solvers, several organizations have designed dedicated commercial algebraic modeling languages, see e.g., AMPL\(^,12\) GAMS\(^,13\) and AIMMS.\(^,14\) These can connect to a large number of commercial and open-source solvers. Commercial and open-source solvers can also be accessed from several open-source algebraic modeling languages for mathematical optimization that are embedded in popular high-level languages. These include PuLP\(^,15\) and Pyomo,\(^,16\) see Hart et al. (2011), Bynum et al. (2021), and CVXPY, see Diamond and Boyd (2016) and Agrawal et al. (2018), which are embedded in Python. They also include JuMP, see Dunning et al. (2017), and Convex.jl, see Udell et al. (2014), which are embedded in Julia, Yalmip and CVX that are embedded in MATLAB, see Löfberg (2004) and Grant and Boyd (2014, 2008), respectively, and FlopC++,\(^,17\) Rehearse,\(^,18\) and Gravity\(^,19\) that are embedded in C++. FlopC++ and Rehearse, which are both part of COIN-OR, offer support for linear programming (LP)
and leverage the Osi Open Solver Interface\textsuperscript{20} to interface with a wide array of solvers (e.g., Cbc, Clp, CPLEX, and GLPK). Gravity can handle more general nonlinear programs with links to CPLEX, Gurobi, and Mosek, among others. These interfaces exploit operator overloading in C++ to describe objectives and constraints in a human readable format that is similar to languages such as GAMS and AMPL. Finally, several commercial vendors provide modeling capabilities combined with built-in solvers, see e.g., Lindo Systems Inc.,\textsuperscript{21} FrontlineSolvers,\textsuperscript{22} and Maximal.\textsuperscript{23}

\textit{Tools for Modelling and Solving Stochastic Optimization Problems.} Tools for modelling and solving stochastic optimization problems are available in Python, Julia, and C++. In Python, the most popular open source packages are PySP,\textsuperscript{24} which is an extension to Pyomo, see Watson et al. (2012), StochOptim,\textsuperscript{25} and MSPPy, see Ding et al. (2019). These packages all provide support for directly solving the “extensive form” formulation of two-stage and multi-stage scenario-based stochastic programs. In addition, PySP offers an implementation of the progressive hedging algorithm. StochOptim provides tools for building scenario-trees from given probability distributions or directly from historical data based on the works of Keutchayan et al. (2018, 2020). MSPPy provides implementations of stochastic dual dynamic (integer) programming. In Julia, the most popular packages are StochJuMP,\textsuperscript{26} StochasticPrograms,\textsuperscript{27} see Biel and Johansson (2019), and SDDP,\textsuperscript{28} see Dowson and Kapelevich (2021). StochJuMP and StochasticPrograms can solve the extensive form problem. In addition, StochasticPrograms offers decomposition capabilities, including the L-shaped method and the progressive hedging algorithm. It also has parallelization capabilities that leverage the standard Julia library for distributed computing. SDDP implements the papers of Dowson (2018) and Dowson et al. (2020a,b), enabling the solution of large multi-stage convex stochastic programs using stochastic dual dynamic programming. Recently, the open source Julia package JuDGE\textsuperscript{29} was released, see Downward et al. (2020). It allows the solution of capacity expansion problems using Dantzig-Wolfe decomposition, see Dantzig and Wolfe (1960). In C++, we are only aware of one stochastic programming package, the SMI: Stochastic Modeling Interface for optimization under uncertainty.\textsuperscript{30} This interface is based on FlopC++ and provides methods for solving scenario-based problems, for generating scenarios, and for interfacing with solvers. Finally, several of the commercial vendors also provide modeling capabilities for stochastic programming, see e.g., Lindo Systems Inc.,\textsuperscript{31} FrontlineSolvers,\textsuperscript{32} Maximal,\textsuperscript{33} GAMS,\textsuperscript{34} AMPL.
We emphasize, that all the aforementioned tools assume that the distribution of the uncertain parameters in the stochastic problem is discrete or provide mechanisms for generating samples from a continuous distribution to feed in a scenario approximation to the true problem.

*Tools for Modelling and Solving Robust Optimization Problems.* Our robust optimization platform ROC++ most closely relates to several open-source tools released in recent years. All of these tools present a similar structure: they provide a modeling platform combined with an approximation/reformulation toolkit that can automatically obtain the robust counterpart, which is then solved using existing open-source and/or commercial solvers. The platform that most closely relates to ROC++ is called ROC and is based on the paper of Bertsimas et al. (2019). It can be used to solve multi-stage distributionally robust optimization problems with real-valued adaptive variables to minimize worst-case expected cost over an ambiguity set of probability distributions. It tackles this class of problems by approximating the adaptive decisions by linear decision rules or enhanced linear decision rules and solves the resulting problem using CPLEX. Contrary to our platform, it cannot solve problems with endogenous uncertainty, nor with binary adaptive variables. It appears to be harder to extend since the problems that it can model are a lot more limited (e.g., no decisions in the uncertainty set, no decision-dependent information discovery, no binary adaptive variables) and since it does not provide a general framework for building new approximations/reformulations. Moreover, it does not provide a Python interface. The majority of the remaining platforms is based on the MATLAB modeling language. One tool builds upon YALMIP, see Löfberg (2012), and provides support for single-stage problems with exogenous uncertainty. A notable advantage of YALMIP is that the robust counterpart output by the platform can be solved using any one of a variety of open-source or commercial solvers. Other platforms, like ROME and RSOME, are entirely motivated by the (stochastic) robust optimization modeling paradigm, see Goh and Sim (2011) and Chen et al. (2020), and provide support for both single- and multi-stage (distributionally) robust optimization problems affected by exogenous uncertain parameters and involving only real-valued adaptive variables. The robust counterparts output by ROME and RSOME can be solved with CPLEX, Mosek, and SDPT3. Recently, JuMPeR has been proposed as an add-on to JuMP. It can be used to model and solve single-stage
problems with exogenous uncertain parameters. JuMPeR can be connected to a large variety of open-source and commercial solvers. On the commercial front, AIMMS is currently equipped with an add-on that can be used to model and automatically reformulate robust optimization problems. It can tackle both single- and multi-stage problems with exogenous uncertainty. A CPLEX license is needed to operate this add-on. To the best of our knowledge, none of the available platforms can address (neither model nor solve) problems involving endogenous uncertain parameters (decision-dependent uncertainty sets or decision-dependent information discovery). None of them can tackle (neither model nor solve) problems presenting binary adaptive variables.

File Formats for Specifying Optimization Problems. To facilitate the sharing and storing of optimization problems, dedicated file formats have been proposed. The two most popular file formats for deterministic mathematical programming problems are the MPS and LP formats. MPS is an older format established on mainframe systems. It is not very intuitive to use. In contrast, the LP format is a lot more interpretable: it captures problems in a way similar to how it is modelled on paper. The SMPS file format is the most popular format for storing stochastic programs and mirrors the role MPS plays in the deterministic setting, see Birge et al. (1987), Gassmann and Schweitzer (2001). To the best of our knowledge, no format exists in the literature for storing and sharing robust optimization problems.

1.4. Contributions

We now summarize our main contributions and the key advantages of our platform:

(a) We propose ROC++, a C++ based platform for modelling, automatically reformulating, and solving robust optimization problems. Our platform is the first capable of addressing both single- and multi-stage problems involving exogenous and/or endogenous uncertain parameters and real- and/or binary-valued adaptive variables. It can also be used to address certain classes of single- or multi-stage stochastic programs whose distribution is continuous and supported on a compact set. Our platform comes with a suite of reformulation/approximation strategies that can be applied either individually or in sequence to convert the robust problem input by the user to a deterministic program that can be solved by off-the-shelf solvers. These include, constant, linear, piecewise constant and piecewise linear decision rules, finite adaptability approximation, duality based reformulation of robust programs, and many more. Our reformulations are (mixed-integer) linear or second-order cone optimization problems
and thus any solver that can tackle such problems can be used to solve the robust counterparts output by our platform. Currently, ROC++ offers an interface to the commercial solver Gurobi and to the open source solver SCIP. We provide a Python library, called ROPy, that features all the main functionality of ROC++.

(b) Thanks to operator overloading in C++, our platform is very easy to use, providing a modeling language similar to the one provided for the deterministic case by solvers such as CPLEX or Gurobi. We illustrate the flexibility and ease of use of our platform on several stylized problems.

(c) We propose the ROB file format, the first file format for storing and sharing general robust optimization problems. Our format builds upon the LP file format and is thus interpretable and easy to use.

(d) ROC++ can easily be extended to support more types of optimization problems, more reformulation/approximation strategies, and other solvers. Any added functionality is also easy to pass to ROPy. We discuss the design rationale of ROC++ and the hooks available for expanding it.

(e) Our platform comes with detailed documentation (created with Doxygen) to facilitate its use and expansion. Our framework is open-source. The source code, installation instructions, and dependencies of ROC++ are available at https://sites.google.com/usc.edu/robust-opt-cpp/.

1.5. Organization of the Paper & Notation

The remainder of this paper is organized as follows. Section 2 describes the broad class of problems to which ROC++ applies. Section 3 describes the approximation schemes that are provided by ROC++. Section 4 discusses the software design and the design rationale of ROC++. A sample model created and solved using ROC++ is provided in Section 5 to illustrate functionality. Section 6 introduces the ROB file format. Section 7 presents extensions to the core model that can also be tackled by ROC++ and briefly highlights the ROPy interface. Finally, Section 8 concludes the paper.

Notation. Throughout this paper, vectors (matrices) are denoted by boldface lowercase (uppercase) letters. The kth element of a vector $\mathbf{x} \in \mathbb{R}^n \ (k \leq n)$ is denoted by $x_k$. Scalars are denoted by lowercase or upper case letters, e.g., $\alpha$ or $N$. We let $L^n_k$ denote the space of all functions from $\mathbb{R}^k$ to $\mathbb{R}^n$. Accordingly, we denote by $B^n_k$ the space of all functions
from $\mathbb{R}^k$ to $\{0,1\}^n$. Given two vectors of equal length, $x, y \in \mathbb{R}^n$, we let $x \circ y$ denote the Hadamard product of the vectors, i.e., their element-wise product.

Throughout the paper, we denote the uncertain parameters by $\xi \in \mathbb{R}^k$. We consider two settings: a robust setting and a stochastic setting. In the robust setting, we assume that the decision-maker wishes to be immunized against realizations of $\xi$ in the uncertainty set $\Xi$. In the stochastic setting, we assume that the distribution $\mathbb{P}$ of the uncertain parameters is fully known. In this case, we let $\Xi$ denote its support and we let $\mathbb{E}(\cdot)$ denote the expectation operator with respect to $\mathbb{P}$.

2. Modeling Robust Optimization Problems

In this section, we present the main classes of robust optimization problems and uncertainty sets to which ROC++ applies.

2.1. Problem Classes

2.1.1. Single-Stage Robust Optimization A single-stage robust optimization problem with exogenous uncertainty set is representable as

$$\begin{align*}
\text{minimize} & \quad \max_{\xi \in \Xi} \quad c(\xi) \top y + d \top z \\
\text{subject to} & \quad y \in Y, \quad z \in Z \\
& \quad A(\xi)y + B(\xi)z \leq h(\xi) \quad \forall \xi \in \Xi,
\end{align*}$$

(1)

where $y \in Y \subseteq \mathbb{R}^n$ and $z \in Z \subseteq \{0,1\}^\ell$ stand for the vectors of real- and binary-valued (static) decisions that must be made before the uncertain parameters $\xi \in \mathbb{R}^k$ are observed. Here, $c(\xi) \in \mathbb{R}^n$ and $d(\xi) \in \mathbb{R}^\ell$ can be interpreted as cost vectors, while $h(\xi) \in \mathbb{R}^m$, and $A(\xi) \in \mathbb{R}^{m \times n}$ and $B(\xi) \in \mathbb{R}^{m \times \ell}$ represent the right-hand-side vector and constraint coefficient matrices, respectively. Without much loss of generality, we assume that $c(\xi), d(\xi), A(\xi), B(\xi)$, and $h(\xi)$ are all linear in $\xi$. The goal of the decision-maker is to select, among all decisions that are robustly feasible (i.e., that satisfy the problem constraints for all realizations of $\xi \in \Xi$), one that achieves the smallest value of the cost, in the worst-case.

In the case of endogenous uncertainty set, we replace the uncertainty set $\Xi$ in the formulation above by $\Xi(z)$, see Section 2.2.
2.1.2. Multi-Stage Robust Optimization with Exogenous Information Discovery

A multi-stage robust optimization problem with exogenous uncertainty over the finite planning horizon \( t \in \mathcal{T} := \{1, \ldots, T\} \) is representable as

\[
\begin{align*}
\text{minimize} & \quad \max_{\xi \in \Xi} \left[ \sum_{t \in \mathcal{T}} c^\top_t y_t(\xi) + d_t(\xi) z_t(\xi) \right] \\
\text{subject to} & \quad y_t \in \mathcal{L}_k^{n_t}, \quad z_t \in \mathcal{B}_k^{\ell_t} \quad \forall t \in \mathcal{T} \\
& \quad \sum_{\tau=1}^{T} A_{t\tau} y_{\tau}(\xi) + B_{t\tau}(\xi) z_{\tau}(\xi) \leq h_t(\xi) \quad \forall \xi \in \Xi, t \in \mathcal{T} \\
& \quad y_t(\xi) = y_t(\xi') \\
& \quad z_t(\xi) = z_t(\xi') \quad \forall t \in \mathcal{T}, \forall \xi, \xi' \in \Xi : w_{t-1} \circ \xi = w_{t-1} \circ \xi'
\end{align*}
\]

where \( y_t(\xi) \in \mathbb{R}^{n_t} \) and \( z_t(\xi) \in \{0,1\}^{\ell_t} \) represent the vectors of real- and binary-valued decisions for time \( t \), respectively. The adaptive nature of the decisions is modeled mathematically by allowing them to depend on the observed realization of the random vector \( \xi \in \mathbb{R}^k \). The vectors \( c_t \in \mathbb{R}^{n_t} \) and \( d_t(\xi) \in \mathbb{R}^{\ell_t} \) can be interpreted as cost vectors, \( h_t(\xi) \in \mathbb{R}^{m_t} \) are the right-hand-side vectors, and \( A_{t\tau} \in \mathbb{R}^{m_t \times n_t} \) and \( B_{t\tau}(\xi) \in \mathbb{R}^{m_t \times \ell_t} \) are the constraint coefficient matrices. Without much loss, we assume that \( d_t(\xi), h_t(\xi), \) and \( B_{t\tau}(\xi) \) are all linear in \( \xi \). The binary vector \( w_t \in \{0,1\}^k \) represents the information base for time \( t + 1 \), i.e., it encodes the information revealed up to time \( t \). Specifically, we have \( w_{t,i} = 1 \) if and only if \( \xi_i \) is observed at some time \( \tau \in \{0, \ldots, t\} \). As information is never forgotten, \( w_t \geq w_{t-1} \) for all \( t \in \mathcal{T} \). The last set of constraints in (2) enforces non-anticipativity by stipulating that \( y_t \) and \( z_t \) must be constant in those uncertain parameters that have not been observed by time \( t \).

As before, in the case of endogenous uncertainty set, we replace the uncertainty set \( \Xi \) in the formulation above by \( \Xi(z) \), see Section 2.2.

2.1.3. Multi-Stage Robust Optimization with Endogenous Information Discovery

Multi-stage robust optimization problems with endogenous information discovery constitute a variant to problem (2) where the information base for each time \( t \in \mathcal{T} \) is kept flexible and under the control of the decision-maker. Thus, the information base must now be modeled as an adaptive decision variable that is itself allowed to depend on \( \xi \) and we denote it by \( w_t(\xi) \in \mathcal{W}_t \subseteq \{0,1\}^k \). The set \( \mathcal{W}_t \) may include constraints that require some uncertain
parameters to be observed at some point in the planning horizon, etc. Therefore, multi-stage robust optimization problems with endogenous information discovery are expressible as

$$\begin{align*}
\text{minimize} & \quad \max_{\xi \in \Xi} \left[ \sum_{t \in T} c_t^\top y_t(\xi) + d_t(\xi) z_t(\xi) + f_t(\xi) w_t(\xi) \right] \\
\text{subject to} & \quad y_t \in L^m_k, \quad z_t \in B^k_t, \quad w_t \in B^k_t \quad \forall t \in T \\
& \quad \sum_{\tau=1}^{\tau=T} A_{t\tau} y_{\tau}(\xi) + B_{t\tau}(\xi) z_{\tau}(\xi) + C_{t\tau}(\xi) w_{\tau}(\xi) \leq h_t(\xi) \\
& \quad w_t(\xi) \in W_t \\
& \quad w_t(\xi) \geq w_{t-1}(\xi) \\
& \quad y_t(\xi) = y_t(\xi') \\
& \quad z_t(\xi) = z_t(\xi') \\
& \quad w_t(\xi) = w_t(\xi') \\
& \quad \forall \xi \in \Xi, t \in T
\end{align*}$$

where $f_{t,i}(\xi) \in \mathbb{R}$ can be interpreted as the cost of including the uncertain parameter $\xi_i$ in the information base at time $t$ and $C_{t\tau}(\xi)$ collects the coefficients of $w_{\tau}$ in the time $t$ constraint. The third constraint ensures that information observed in the past cannot be forgotten while the last set of constraints are decision-dependent non-anticipativity constraints that model the requirement that decisions can only depend on information that the decision-maker chose to observe in the past. Without much loss of generality, we assume that the cost vectors $d_t(\xi)$ and $f_t(\xi)$ and the matrices $B_{t\tau}(\xi) \in \mathbb{R}^{m_t \times \ell_t}$ and $C_{t\tau}(\xi) \in \mathbb{R}^{m_t \times k}$ are all linear in $\xi$.

We note that extensions are available in ROC++ that can cater for more classes of problems than described here, see Section 7.

2.2. Modeling Uncertainty

We now discuss our model for the uncertainty sets $\Xi$ and $\Xi(x)$ for the exogenous and endogenous cases, respectively.

2.2.1. Exogenous Uncertainty Set For the exogenous setting, we assume that $\Xi$ is compact and admits a conic representation, i.e., it is expressible as

$$\Xi := \{ \xi \in \mathbb{R}^k : \exists \zeta^s \in \mathbb{R}^{k_s}, \ s = 1, \ldots, S : P^s \xi + Q^s \zeta^s + q^s \in \mathcal{K}^s, \ s = 1, \ldots, S \}$$

for some matrices $P^s \in \mathbb{R}^{r_s \times k}$ and $Q^s \in \mathbb{R}^{r_s \times k_s}$, and vector $q^s \in \mathbb{R}^{r_s}, \ s = 1, \ldots, S$, where $\mathcal{K}^s, \ s = 1, \ldots, S$, are closed convex pointed cones in $\mathbb{R}^{r_s}$. Finally, we assume that the representation above is strictly feasible (unless the cones involved in the representation are...
polyhedral, in which case this assumption can be relaxed). In our platform, we focus on
the cases where the cones $K^s$ are either polyhedral, i.e., $K^s = \mathbb{R}^r_+$, or Lorentz cones, i.e.,
$K^s = \{ u \in \mathbb{R}^r : \sqrt{u_1^2 + \cdots + u_{r-1}^2} \leq u_r \}$.

Uncertainty sets of the form (4) arise naturally from statistics or from knowledge of
the distribution of the uncertain parameters. The uncertainty set $\Xi$ can be constructed as
the support of the distribution of the uncertain parameters, see e.g., Kuhn et al. (2009).
More often, it is constructed in a data-driven fashion to guarantee that constraints are
satisfied with high probability, see e.g., Bertsimas et al. (2018). More generally, disciplined
methods for constructing uncertainty sets from “random” uncertainty exist, see e.g., Bandi
and Bertsimas (2012), Ben-Tal et al. (2009). We now discuss several uncertainty sets from
the literature that can be modelled in the form (4).

**Example 1 (Budget Uncertainty Sets).** Uncertainty sets of the form (4) can be
used to model 1-norm and $\infty$-norm uncertainty sets with budget of uncertainty $\Gamma$, given
by $\{ \xi \in \mathbb{R}^k : \| \xi \|_1 \leq \Gamma \}$ and $\{ \xi \in \mathbb{R}^k : \| \xi \|_\infty \leq \Gamma \}$, respectively. More generally, they can be
used to impose budget constraints at various levels of a given hierarchy. For example, they
can be used to model uncertainty sets of the form

$$\left\{ \xi \in \mathbb{R}^k : \sum_{i \in H_h} |\xi_i| \leq \Gamma_h \xi_h \quad \forall h = 1, \ldots, H \right\},$$

where the sets $H_h \subseteq \{1, \ldots, k\}$ collect the indices of all uncertain parameters in the $h$th
level of the hierarchy and $\Gamma_h \in \mathbb{R}_+$ is the budget of uncertainty for hierarchy $h$, see e.g.,
Simchi-Levi et al. (2019).

**Example 2 (Ellipsoidal Uncertainty Sets).** Uncertainty sets of the form (4) cap-
ture as special cases ellipsoidal uncertainty sets, which arise for example as confidence
regions from Gaussian distributions. These are expressible as

$$\{ \xi \in \mathbb{R}^k : (\xi - \bar{\xi})^T P^{-1} (\xi - \bar{\xi}) \leq 1 \},$$

for some matrix $P \in \mathbb{S}^k_+$ and vector $\bar{\xi} \in \mathbb{R}^k$, see e.g., Ben-Tal et al. (2009).

**Example 3 (Central Limit Theorem Uncertainty Sets).** Sets of the form (4)
can be used to model Central Limit Theorem based uncertainty sets. These sets arise for
example as confidence regions for large numbers of i.i.d. uncertain parameters and are
expressible as

$$\left\{ \xi \in \mathbb{R}^k : \left| \sum_{i=1}^k \xi_k - \mu_k \right| \leq \Gamma \sigma \sqrt{k} \right\},$$
where \( \mu \) and \( \sigma \) are the mean and standard deviation of the i.i.d. parameters \( \xi_i, i = 1, \ldots, k \), see Bandi and Bertsimas (2012), Bandi et al. (2018).

**Example 4 (Uncertainty Sets based on Factor Models).** Sets of the form (4) capture as special cases uncertainty sets based on factor models that are popular in finance and economics. These are expressible in the form

\[
\{ \xi \in \mathbb{R}^k : \exists \zeta \in \mathbb{R}^\kappa : \xi = \Phi \zeta + \phi, \|\zeta\|_2 \leq 1 \},
\]

for some vector \( \phi \in \mathbb{R}^k \) and matrix \( \Phi \in \mathbb{R}^{k \times \kappa} \).

**2.2.2. Endogenous Uncertainty Set** For the endogenous setting, we assume that the uncertainty set is expressible as

\[
\Xi(z) := \{ \xi \in \mathbb{R}^k : \exists \zeta^s \in \mathbb{R}^{k_s}, s = 1, \ldots, S : P^s(z)\xi + Q^s(z)\zeta^s + q^s(z) \in K^s, s = 1, \ldots, S \},
\]

where \( z \) are binary variables and \( P^s(z) \in \mathbb{R}^{r_s \times k}, Q^s(z) \in \mathbb{R}^{r_s \times k_s}, \) and \( q^s(z) \in \mathbb{R}^{r_s} \), are all linear in \( z \), and \( K^s, s = 1, \ldots, S \), are either polyhedral or Lorentz cones in \( \mathbb{R}^{r_s} \).

**3. Decision Rule & Finite Adaptability Approximations**

In formulation (3), the recourse decisions are very hard to interpret since decisions are modelled as (potentially very complicated) functions of the history of observations. The functional form of the decisions combined with the infinite number of constraints involved in problem (3) also imply that this problem cannot be solved directly. This has motivated researchers in robust optimization to propose several approximation schemes capable of bringing problem (3) to a form amenable to solution by off-the-shelf solvers, see Section 1. Broadly speaking, these approximation schemes fall in two categories: interpretable decision rule approximations, which restrict the functional form of the recourse decisions; and finite adaptability approximation schemes, which yield a moderate number of contingency plans that are candidates to be implemented at each stage. These approximations have the benefit of improving the tractability properties of the problem and of rendering decisions more interpretable, a highly desirable property of any automated decision support system.

We now describe the approximation schemes supported by our platform. Our choice of approximations is such that they apply to problems with exogenous and/or endogenous uncertainty (decision-dependent uncertainty set or decision-dependent information discovery). A decision tree describing the papers that ROC++ relies on in each case is described in Figure 1.
3.1. Interpretable Decision Rules

**Constant Decision Rule and Linear Decision Rule.** The most crude (and perhaps most interpretable) decision rules that are available in ROC++ are the constant decision rule (CDR) and the linear decision rule (LDR), see Ben-Tal et al. (2009). These apply to binary and real-valued decision variables, respectively. Under the constant decision rule, the binary decisions \( z_t(\xi) \) and \( w_t(\xi) \) are no longer allowed to adapt to the history of observations – it is assumed that the decision-maker will take the same action, independently of the realization of the uncertain parameters. Mathematically, we have

\[
z_t(\xi) = z_t \quad \text{and} \quad w_t(\xi) = w_t \quad \forall t \in T, \forall \xi \in \Xi,
\]

for some vectors \( z_t \in \{0,1\}^{\ell_t} \) and \( w_t \in \{0,1\}^{k_t} \), \( t \in T \). Under the linear decision rule, the real-valued decisions are modelled as affine functions of the history of observations, i.e.,

\[
y_t(\xi) = Y_t \xi + y_t \quad \forall t \in T,
\]

for some matrix \( Y_t \in \mathbb{R}^{n_t \times k_t} \) and vector \( y_t \in \mathbb{R}^{n_t} \). The LDR model leads to very interpretable decisions – we can think of this decision rule as a scoring rule that assigns different values
(coefficients) to each uncertain parameter. These coefficients can be interpreted as the sensitivity of the decision variables to changes in the uncertain parameters. Under the CDR and LDR approximations the adaptive variables in the problem are eliminated and the quantities $z_t$, $w_t$, $Y_t$, and $y_t$ become the new decision variables of the problem.

**Piecewise Constant and Piecewise Linear Decision Rules.** In piecewise constant (PWC) and piecewise linear (PWL) decision rules, the binary (resp. real-valued) adjustable decisions are approximated by functions that are piecewise constant (resp. piecewise linear) on a preselected partition of the uncertainty set. Specifically, the uncertainty set $\Xi$ is split into hyper-rectangles of the form $\Xi_s := \{ \xi \in \Xi : c^i_{s_{i-1}} \leq \xi_i < c^i_{s_i}, \ i = 1, \ldots, k \}$, where $s \in \mathcal{S} := \times_{i=1}^k \{1, \ldots, r_i\} \subseteq \mathbb{Z}^k$ and $c^1_i < c^2_i < \cdots < c^r_i$ represent $r_i - 1$ breakpoints along the $\xi_i$ axis. Mathematically, the binary and real-valued decisions are expressible as

$$z_t(\xi) = \sum_{s \in \mathcal{S}} \mathbb{I}(\xi \in \Xi_s) z^s_t, \quad w_t(\xi) = \sum_{s \in \mathcal{S}} \mathbb{I}(\xi \in \Xi_s) w^s_t,$$

and

$$y_t(\xi) = \sum_{s \in \mathcal{S}} \mathbb{I}(\xi \in \Xi_s) (Y^s_t \xi + y^s_t),$$

for some vectors $z^s_t \in \mathbb{R}^{\ell_t}$, $w^s_t \in \mathbb{R}^k$, $y^s_t \in \mathbb{R}^{n_t}$ and matrices $Y^s_t \in \mathbb{R}^{n_t \times k}$, $t \in \mathcal{T}$, $s \in \mathcal{S}$. We can think of this decision rule as a scoring rule that assigns different values (coefficients) to each uncertain parameter; the score assigned to each parameter depends on the subset of the partition in which the uncertain parameter lies. Although less interpretable than CDR/LDR, the PWC/PWL approximation enjoys better optimality properties: it will usually outperform CDR/LDR, since the decisions that can be modelled are more flexible.

### 3.2. Contingency Planning via Finite Adaptability

Another solution approach available in ROC++ is the so-called finite adaptability approximation that applies to robust problems with binary decisions, see Hanasusanto et al. (2015), Vayanos et al. (2019). Under the finite adaptability approximation, the adaptive decisions in problems (2) and (3) are approximated as follows: in the first decision-stage (here-and-now), a moderate number $K_t$ of candidate strategies are chosen for each decision-stage $t$; at the beginning of each period, the best of these candidate strategies is selected, in an adaptive fashion.

Mathematically, the finite adaptability approximation of a problem is a multi-stage robust optimization problem wherein in the first period, a collection of contingency plans...
$z_t^{k_1,\ldots,k_t} \in \{0,1\}^{k_t}$ and $w_t^{k_1,\ldots,k_t} \in \{0,1\}^k$, $k_t \in \{1,\ldots,K_t\}$, $t \in T$ for the variables $z_t(\xi)$ and $w_t(\xi)$ are chosen. Then, at the begin of each period $t \in T$, one of the contingency plans for that period is selected to be implemented, in an adaptive fashion.

Relative to the piecewise constant decision rule, the finite adaptability approach usually results in better performance in the following sense: the number of contingency plans needed in the finite adaptability approach to achieve a given objective value is never greater than the number of subsets needed in the piecewise constant decision rule to achieve that same value. However, the finite adaptability approximation does not apply directly to problems with real-valued decision variables and is thus less attractive in that sense since more approximations are needed before it can be used on problems of that type.

4. ROC++ Software Design and Design Rationale
4.1. Classes Involved in the Modeling of Optimization Problems

The main building blocks to model optimization problems in the ROC++ platform are the optimization model interface class, ROCPPOptModelIF, the constraint interface class, ROCPPConstraintIF, the decision variable interface class, ROCPPVarIF, the objective function interface class, ROCPPObjectiveIF, their derived classes, and the uncertain parameter class, ROCPPUnc. These classes mainly act as containers to which several reformulations, approximations, and solvers can be applied as appropriate, see Sections 4.3, 4.4, and 4.5. Inheritance in the aforementioned classes implements the “is a” relationship. The main advantages of inheritance are code reusability and readability, since derived classes inherit the properties and functionality of the parent class. We now give a more detailed description of some of these classes and how they relate to one another.

![Inheritance diagram for the ROCPPVarIF class.](image)
The ROCPPVarIF class is an abstract base class that provides a common interface to all decision variable types. Its class diagram is provided in Figure 2. Its children are the abstract classes, ROCPPStaticVarIF and ROCPPAdaptVarIF, that model static and adaptive variables, respectively. Each of these present three children each of which model static (resp. adaptive) real-valued, binary, or integer variables, see Figure 2.

The ROCPPConstraintIF class is an abstract base class with three children: the abstract class ROCPPClassicConstraint whose two children, ROCPPEqConstraint and ROCPPIneqConstraint, model equality and inequality constraints, respectively; the class ROCPPSOSConstraint that is used to model SOS constraints; and the class ROCPPIfThenConstraint that is used to model logical forcing constraints. Constraints can either be main problem constraints or define the uncertainty set and may involve decision variables and/or uncertain parameters. The ROCPPObjectiveFunctionIF abstract base class presents two children, ROCPPSimpleObjective and ROCPPMaxObjective that model linear and piecewise linear convex objective functions, respectively. The key building block for the ROCPPConstraintIF and ROCPPObjectiveFunctionIF classes are the ROCPPExpr class that models an expression, which is a sum of terms of abstract base type ROCPPCstrTermIF. The ROCPPCstrTermIF class has two children: ROCPPProdTerm, which are used to model monomials and ROCPPNorm, which are used to model the two-norm of an expression.

---

**Figure 3** Inheritance diagram for the ROCPPOptModelIF class.

The ROCPPOptModelIF is an abstract base class that provides a common and standardized interface to all optimization problem types. It consists of decision variables, constraints, an objective function, and potentially uncertain parameters. Its class diagram is shown in Figure 3. ROCPPOptModelIF presents two derived classes, ROCPPDetOptModel and

---

Vayanos, Jin, and Elissaios: *ROC++: Robust Optimization in C++*
ROCPPUncOptModel, which are used to model deterministic optimization models and optimization models involving uncertain parameters, respectively. While RCPPDetOptModel can involve arbitrary deterministic constraints, its derived classes, RCPPMISOCP and RCPPBilinMISOCP, can only model mixed-integer second order cone problems (MISOCP) and MISOCPs that also involve bilinear terms. The RCPPUncOptModel class presents two derived classes, RCPPUncSSOptModel and RCPPUncMSOptModel, that are used to model single- and multi-stage problems respectively. Finally, RCPPUncMSOptModel has two derived classes, RCPPOptModelExoID and RCPPOptModelDDID, that can model multi-stage optimization problems where the time of information discovery is exogenous and endogenous, respectively. The type of constraints and uncertain parameters that can be added to each problem depend on the problem type. The main role of inheritance here is to ensure that the problems constructed are of types to which the available tools (reformulators, approximators, or solvers) can apply. Naturally, if the platform is augmented with more such tools that enable the solution of different/more general optimization problems, the existing inheritance structure can be leveraged to easily extend the code.

### 4.2. Interpretable Problem Input through Operator Overload

ROC++ leverages operator overloading in C++ to enable the creation of problem expressions and constraints in a highly interpretable human-readable format. RCPPExpr, RCPPCstrTermIF, RCPPVarIF, and RCPPUnc objects can be added or multiplied together to form new RCPPExpr objects that can be used as left-hand-sides of constraints. The double equality ("==") or inequality ("<=") signs can be used to create constraints. This framework effectively generalizes the modeling setup of modern solvers like Gurobi or CPLEX to the uncertain setting, see Section 5 for examples.

### 4.3. Dynamic Behavior via Strategy Pattern

*Reformulation Strategies.* The central objective of our platform is to convert (potentially through approximations) the original uncertain problem input by the user to a form that it can be fed into and solved by an off-the-shelf solver. This is achieved in our code through the use of reformulation strategies applied sequentially to the input problem. Currently, our platform provides a suite of such strategies, all of which are derived from the abstract base class ReformulationStrategyIF. The main approximation strategies are: the linear and constant decision rule approximations, provided by the classes
ROCPPLinearDR and ROCPPConstantDR, respectively; the piecewise decision rule approximation, provided by the ROCPPPWDR class; and the $K$-adaptability approximation, provided by the ROCPPKAdapt class. The main equivalent reformulation strategies are: the ROCPPRobustifyEngine, which can convert a single-stage robust problem to its deterministic counterpart; and the ROCPPMItoMB class, which can linearize bilinear terms involving products of binary and real-valued (or other) decisions.

Reformulation Orchestrator. In ROC++, the user can select at runtime which strategies to apply to their input problem and the sequence in which these strategies should be used. This is achieved by using the idea of a strategy pattern, which allows an object to change its behavior based on some unpredictable factor, see e.g., Perez (2018). To implement the strategy pattern, we provide, in addition to the reformulation strategies discussed above, the class ROCPPOrchestrator that will act as the client, being aware of the existence of strategies but not needing to know what each strategy does. At runtime, an optimization problem, the context, is provided to the ROCPPOrchestrator together with a strategy or set of strategies to apply to the context and the orchestrator applies the strategies in sequence, after checking that they can apply to the input problem.

Using and Extending the Code. Thanks to the idea of the strategy pattern, the code is very easy to use (the user simply needs to provide the input problem and the sequence of reformulation strategies). It is also very easy to extend; a researcher can create more reformulation strategies and leverage the existing client code to apply these strategies at runtime to the input problem.

4.4. Solver Interface

The ROC++ platform provides an abstract base class, ROCPPSolverInterface, which is used to convert deterministic MISOCPs in ROC++ format to a format that is recognized and solved by a commercial or open source solver. Currently, there is support for two solvers: Gurobi, through the ROCPPGurobi class, and SCIP, through the ROCPPSCIP class. Both these classes are children of ROCPPSolverInterface and allow for changing the solver parameters, solving the problem, retrieving an optimal solution, etc. New solvers can conveniently be added by creating children classes to ROCPPSolverInterface and implementing its pure virtual member functions.
4.5. Tools to Facilitate Extension

The ROC++ platform comes with several classes that can be leveraged to construct new reformulation strategies, such as polynomial decision rules, see e.g., Bampou and Kuhn (2011) and Vayanos et al. (2012), or constraint sampling approximations, see Campi and Garatti (2008). The key classes that can help construct new approximators and reformulators are the abstract base class ROCPPVariableConverterIF and its abstract derived classes ROCPPOneToOneVarConverterIF and ROCPPOneToExprVarConverterIF, which can map variables in the problem to other variables, and variables to expressions, respectively. For example, one of the derived classes of ROCPPOneToExprVarConverterIF is ROCPPPredefO2EVarConverter, which takes a map from variable to expression as input and maps all variables in the problem to their corresponding expressions in the map. We have used it to implement the linear decision rule by passing a map from adaptive variables to affine functions of uncertain parameters. New decision rule approximations, such as polynomial decision rules, can be added in a similar way.

5. Modelling and Solving Decision-Making Problems in ROC++

In this section, we showcase the ease of use of our platform through a concrete example. Additional examples are provided in Electronic Companion EC.1.

5.1. Robust Pandora’s Box: Problem Description

We consider a robust variant of the celebrated stochastic Pandora Box (PB) problem due to Weitzman (1979). This problem models selection from a set of unknown, alternative options, when evaluation is costly. There are \( I \) boxes indexed in \( \mathcal{I} := \{1, \ldots, I\} \) that we can choose or not to open over the planning horizon \( \mathcal{T} := \{1, \ldots, T\} \). Opening box \( i \in \mathcal{I} \) incurs a cost \( c_i \in \mathbb{R}_+ \). Each box has an unknown value \( \xi_i \in \mathbb{R} \), \( i \in \mathcal{I} \), which will only be revealed if the box is opened. At the beginning of each time \( t \in \mathcal{T} \), we can either select a box to open or keep one of the opened boxes, earn its value (discounted by \( \theta^{t-1} \)), and stop the search.

We assume that the box values are restricted to lie in the set

\[
\Xi := \{ \xi \in \mathbb{R}^I : \exists \zeta \in [-1,1]^M, \xi_i = (1 + \Phi_i^T \zeta /2) \bar{\xi}_i, \quad \forall i \in \mathcal{I} \},
\]

where \( \zeta \in \mathbb{R}^M \) represent \( M \) risk factors, \( \Phi_i \in \mathbb{R}^M \) represent the factor loadings, and \( \bar{\xi} \in \mathbb{R}^I \) collects the nominal box values.
In this problem, the box values are endogenous uncertain parameters whose time of revelation can be controlled by the box open decisions. Thus, the information base, encoded by the vector \( w_t(\xi) \in \{0,1\}^T, t \in \mathcal{T} \), is a decision variable. In particular, \( w_{t,i}(\xi) = 1 \) if and only if box \( i \in \mathcal{I} \) has been opened on or before time \( t \in \mathcal{T} \) in scenario \( \xi \). We assume that \( w_0(\xi) = 0 \) so that no box is opened before the beginning of the planning horizon. We denote by \( z_{t,i}(\xi) \in \{0,1\} \) the decision to keep box \( i \in \mathcal{I} \) and stop the search at time \( t \in \mathcal{T} \).

The requirement that at most one box be opened at each time \( t \in \mathcal{T} \) and that no box be opened if we have stopped the search can be captured by the constraint

\[
\sum_{i \in \mathcal{I}} (w_{t,i}(\xi) - w_{t-1,i}(\xi)) \leq 1 - \sum_{\tau=1}^{t} \sum_{i \in \mathcal{I}} z_{\tau,i}(\xi) \quad \forall t \in \mathcal{T}.
\]

The requirement that only one of the opened boxes can be kept is expressible as

\[
z_{t,i}(\xi) \leq w_{t-1,i}(\xi) \quad \forall t \in \mathcal{T}, \forall i \in \mathcal{I}.
\]

The objective of the PB problem is to select the sequence of boxes to open and the box to keep so as to maximize worst-case net profit. Since the decision to open box \( i \) at time \( t \) can be expressed as the difference \( (w_{t,i} - w_{t-1,i}) \), the objective of the PB problem is

\[
\max \min_{\xi \in \Xi} \sum_{t \in \mathcal{T}} \sum_{i \in \mathcal{I}} \theta^{t-1}_i \xi_i z_{t,i}(\xi) - c_i(w_{t,i}(\xi) - w_{t-1,i}(\xi)).
\]

The mathematical model for this problem can be found in Electronic Companion EC.3.1.

### 5.2. Robust Pandora’s Box: Model in ROC++

We present the ROC++ model for the PB problem. We assume that the data of the problem have been defined in C++ as summarized in Table 1. We discuss how to construct the key elements of the problem here. The full code can be found in Electronic Companion EC.3.2.

The PB problem is a multi-stage robust optimization problem involving uncertain parameters whose time of revelation is decision-dependent. Such models can be stored in the ROCPPOptModelDDID class which is derived from ROCPPOptModelIF. We note that in ROC++ all optimization problems are minimization problems. All models are pointers to the interface class ROCPPOptModelIF. Thus, the robust PB problem can be initialized as:

```cpp
// Create an empty robust model with T periods for the PB problem
ROCPPOptModelIF_Ptr PBModel(new ROCPPOptModelDDID(T, robust));
```
Next, we create the ROC++ variables associated with uncertain parameters and decision variables in the problem. The correspondence between variables is summarized in Table 2.

The uncertain parameters of the PB problem are $\xi \in \mathbb{R}^I$ and $\zeta \in \mathbb{R}^M$. We store the ROC++ variables associated with these in the Value and Factor maps, respectively. Each uncertain parameter is a pointer to an object of type ROCPPUnc. The constructor of the ROCPPUnc class takes two input parameters: the name of the uncertain parameter and the period when that parameter is revealed (first time stage when it is observable). As $\xi$ has a time of revelation that is decision-dependent, we can omit the second parameter when we construct the associated ROC++ variables. The ROCPPUnc constructor also admits a third (optional) parameter with default value true that indicates if the uncertain parameter is observable. As $\zeta$ is an auxiliary uncertain parameter, we set its time period as being, e.g., 1 and indicate through the third parameter in the constructor of ROCPPUnc that this parameter is not observable.

```cpp
3 // Create empty maps to store the uncertain parameters
4 map<uint, ROCPPUnc_Ptr> Value, Factor;
5 for (uint i = 1; i <= I; i++)
6   // Create the uncertainty associated with box i and add it to Value
7   Value[i] = ROCPPUnc_Ptr(new ROCPPUnc("Value_"+to_string(i)));
8 for (uint m = 1; m <= M; m++)
9   // The risk factors are not observable
10  Factor[m] = ROCPPUnc_Ptr(new ROCPPUnc("Factor_"+to_string(m),1,false));
```
The decision variables of the problem are the measurement variables $w$ and the variables $z$ which decide on the box to keep. We store these in the maps $\text{MeasVar}$ and $\text{Keep}$, respectively. In ROC++, the measurement variables are created automatically for all time periods in the problem by calling the $\text{add\_ddu()}$ function, which is a public member of $\text{ROCPPOptModelIF}$. This function admits four input parameters: an uncertain parameter, the first and last time period when the decision-maker can choose to observe that parameter, and the cost for observing the parameter. In this problem, cost for observing $\xi_i$ is equal to $c_i$. The measurement variables constructed in this way can be recovered using the $\text{getMeasVar()}$ function, which admits as inputs the name of an uncertain parameter and the time period for which we want to recover the measurement variable associated with that uncertain parameter.

```cpp
map< uint , map< uint , ROCPPVarIF_Ptr > > MeasVar;
for ( uint i = 1; i <= I; i++) {
    // Create the measurement variables associated with the value of box i
    PBModel->add\_ddu(Value[i], 1, T, obsCost[i]);
    // Get the measurement variables and store them in MeasVar
    for ( uint t = 1; t <= T; t++)
        MeasVar[t][i] = PBModel->getMeasVar(Value[i]->getName(), t);
}
```

The boolean $\text{Keep}$ variables can be built in ROC++ using the constructors of the $\text{ROCPPStaticVarBool}$ and $\text{ROCPPAdaptVarBool}$ classes for the static and adaptive variables, respectively. The constructor of $\text{ROCPPStaticVarBool}$ admits one input parameter: the name of the variable. The constructor of $\text{ROCPPAdaptVarBool}$ admits two input parameters: the name of the variable and the time period when the decision is made.

```cpp
map< uint , map< uint , ROCPPVarIF_Ptr > > Keep;
for ( uint t = 1; t <= T; t++) {
    for ( uint i = 1; i <= I; i++) {
        if (t == 1) // In the first period, the Keep variables are static
            Keep[t][i] = ROCPPVarIF_Ptr(new ROCPPStaticVarBool("Keep_"+to\_string(t )+"_"+to\_string(i)));
        else // In the other periods, the Keep variables are adaptive
            Keep[t][i] = ROCPPVarIF_Ptr(new ROCPPAdaptVarBool("Keep_"+to\_string(t ) +"_"+to\_string(i), t));
    }
}
```

Having created the decision variables and uncertain parameters, we turn to adding the constraints to the model. To this end, we use the $\text{StoppedSearch}$ expression, which tracks the running sum of the $\text{Keep}$ variables, to indicate if at any given point in time, we have
already decided to keep one box and stop the search. We also use the `NumOpened` expression that, at each period, stores the expression for the total number of boxes that we choose to open in that period. Using these expressions, the constraints can be added to the problem using the following code.

```cpp
// Create the constraints and add them to the problem
ROCPPExpr_Ptr StoppedSearch(new ROCPPExpr());
for (uint t = 1; t <= T; t++) {
    // Create the constraint that at most one box be opened at t (none if the search has stopped)
    ROCPPExpr_Ptr NumOpened(new ROCPPExpr());
    // Update the expressions and the constraint to the problem
    for (uint i = 1; i <= I; i++) {
        StoppedSearch = StoppedSearch + Keep[t][i];
        if (t > 1)
            NumOpened = NumOpened + MeasVar[t][i] - MeasVar[t-1][i];
        else
            NumOpened = NumOpened + MeasVar[t][i];
    }
    PBModel->add_constraint(NumOpened <= 1. - StoppedSearch);
}

// Constraint that only one of the open boxes can be kept
for (uint i = 1; i <= I; i++)
    PBModel->add_constraint((t > 1) ? (Keep[t][i] <= MeasVar[t-1][i]) : (Keep[t][i] <= 0.));
```

Next, we create the uncertainty set and the objective function.

```cpp
// Create the uncertainty set constraints and add them to the problem
// Add the upper and lower bounds on the risk factors
for (uint m = 1; m <= M; m++) {
    PBModel->add_constraint_uncset(Factor[m] >= -1.0);
    PBModel->add_constraint_uncset(Factor[m] <= 1.0);
}

// Add the expressions for the box values in terms of the risk factors
for (uint i = 1; i <= I; i++) {
    ROCPPExpr_Ptr ValueExpr(new ROCPPExpr());
    for (uint m = 1; m <= M; m++)
        ValueExpr = ValueExpr + RiskCoeff[i][m]*Factor[m];
    PBModel->add_constraint_uncset(Value[i] == (1. + 0.5*ValueExpr)*NomVal[i]);
}

// Create the objective function expression
ROCPPExpr_Ptr PBObj(new ROCPPExpr());
for (uint t = 1; t <= T; t++)
    for (uint i = 1; i <= I; i++)
        PBObj = PBObj + pow(theta, t-1)*Value[i]*Keep[t][i];

// Set objective (multiply by -1 for maximization)
PBModel->set_objective(-1.0*PBObj);
```

We emphasize that the observation costs were automatically added to the objective function when we called the `add_ddu()` function.
5.3. Robust Pandora’s Box: Solution in ROC++

The PB problem is a multi-stage robust problem with decision-dependent information discovery, see Vayanos et al. (2011, 2019). ROC++ offers two options for solving this class of problems: finite adaptability and piecewise constant decision rules, see Section 3. Here, we illustrate how to solve PB using the finite adaptability approach, see Section 3.2. We let Kmap store the number of contingency plans $K_t$ per period—the index in the map indicates the time period $t$ and the value it maps to corresponds to the choice of $K_t$. The process of computing the optimal contingency plans is streamlined in ROC++.

```cpp
// Construct the reformulation orchestrator
ROCPPOrchestrator_Ptr pOrch(new ROCPPOrchestrator());
// Construct the finite adaptability reformulation strategy with 2 candidate policies in the each time stage
ROCPPStrategy_Ptr pKadaptStrategy(new ROCPPKAdapt(Kmap));
// Construct the robustify engine reformulation strategy
ROCPPStrategy_Ptr pRE(new ROCPPRobustifyEngine());
// Construct the linearization strategy based on big M constraints
ROCPPStrategy_Ptr pBTR(new ROCPPBTR_bigM());
// Approximate the adaptive decisions using the linear/constant decision rule approximator and robustify
vector<ROCPPStrategy_Ptr> strategyVec {pKadaptStrategy, pRE, pBTR};
ROCPPOptModelIF_Ptr PBModelKAdapt = pOrch->Reformulate(PBModel, strategyVec);
// Construct the solver and solve the problem
ROCPPSolver_Ptr pSolver(new ROCPPGurobi(SolverParams()));
pSolver->solve(PBModelKAdapt);
```

We consider the instance of PB detailed in Electronic Companion EC.3.3 for which $T=4$, $M=4$, and $I=5$. For $K_t=1$ (resp. $K_t=2$ and $K_t=3$) for all $t \in T$, the problem takes under half a second (resp. under half a second and 6 seconds) to approximate and robustify. Its objective value is 2.12 (resp. 9.67 and 9.67). Note that with $T=4$ and $K_t=2$ (resp. $K_t=3$), the total number of contingency plans is 8 (resp. 27).

Next, we showcase how optimal contingency plans can be retrieved in ROC++.

```cpp
// Retrieve the optimal solution from the solver
map<string, double> optimalSln(pSolver->getSolution());
// Print the optimal decision (from the original model)
// Print decision rules for variable Keep_4_2 from the original problem automatically
ROCPPKAdapt_Ptr pKadapt = static_pointer_cast<ROCPPKAdapt>(pKadaptStrategy);
pKadapt->printOut(PBModel, optimalSln, Keep[4][2]);
```

When executing this code, the values of all variables $z_{t,k_1...k_t}$ for all contingency plans $(k_1, ..., k_t) \in \times_{t=1}^T \{1, ..., K^T\}$ are printed. We show here the subset of the output associated with contingency plans where $z_{2,4}(\xi)$ equals 1 (for the case $K=2$).
Thus, at time 4, we will keep the second box if and only if the contingency plan we choose is \((k_1, k_2, k_3, k_4) = (1, 2, 2, 1)\). We can display the first time that an uncertain parameter is observed using the following ROC++ code.

```cpp
// Prints the observation decision for uncertain parameter Value_2
pKadapt->printOut(PBModel, optimalSln, Value[2]);
```

When executing this code, the time when \(\xi_2\) is observed under each contingency plan \((k_1, \ldots, k_T) \in \times_{T \subseteq \mathcal{T}} \mathcal{K}^T\) is printed. In this case, part of the output we get is as follows.

| Parameter Value_2 under contingency plan (1-1-1-1) is never observed |
| Parameter Value_2 under contingency plan (1-2-2-1) is observed at time 2 |

Thus, in an optimal solution, \(\xi_2\) is opened at time 2 under contingency plan \((k_1, k_2, k_3, k_4) = (1, 2, 1, 1)\). On the other hand it is never opened under contingency plan \((1, 1, 1, 1)\).

### 6. ROB File Format

Given a robust/stochastic optimization problem expressed in ROC++, our platform can generate a file displaying the problem in human readable format. We use the Pandora Box problem from Section 5 to illustrate our proposed format, with extension ".rob".

The file starts with the **Objective** part that presents the objective function of the problem: to minimize either expected or worst-case costs, as indicated by \(E\) or \(\max\), respectively. For example, since the PB problem optimizes worst-case profit, we obtain the following.

```
Objective:
min max -1 Keep_1_1 Value_1 -1 Keep_1_2 Value_2 -1 Keep_1_3 Value_3 ...
```

Then come the **Constraints** and **Uncertainty Set** parts, which list the constraints using interpretable “\(<=\)”,”\(>=\),” and “\(==\)” operators. We list one constraint for each part here.

```
Constraints:
c0: -1 mValue_2_1 +1 mValue_1_1 <= +0 ...
Uncertainty Set:
c0: -1 Factor_1 <= +1 ...
```

The next part, **Decision Variables**, lists the decision variables of the problem. For each variable, we list its name, type, whether it is static or adaptive, its time stage, and whether it is a measurement variable or not. If it is a measurement variable, we also display the uncertain parameter whose time of revelation it controls.
Decision Variables:
Keep_1_1: Boolean, Static, 1, Non-Measurement
mValue_2_2: Boolean, Adaptive, 2, Measurement, Value_2

The Bounds part then displays the upper and lower bounds for the decision variables.

Bounds:
0 <= Keep_1_1 <= 1

Finally, the Uncertainties part lists, for each uncertain parameter, its name, whether the parameter is observable or not, its time stage, if the parameter has a time of revelation that is decision-dependent, and the first and last stages when the parameter can be observed.

Uncertainties:
Factor_4: Not Observable, 1, Non-DDU
Value_1: Observable, 1, DDU, 1, 4

7. Extensions
7.1. Integer Decision Variables
ROC++ can solve problems involving integer decision variables. In the case of the CDR/PWC approximations, integer adaptive variables are directly approximated by constant/piecewise constant decisions that are integer on each subset of the partition. In the case of the finite adaptability approximation, bounded integer variables must first be expressed as finite sums of binary variables before the approximation is applied. This can be achieved through the reformulation strategy ROCPPBinaryMItoMB.

7.2. Stochastic Programming Capability
ROC++ currently provides limited support for solving stochastic programs with exogenous and/or decision-dependent information discovery based on the paper Vayanos et al. (2011). In particular, the approach from Vayanos et al. (2011) is available for the case where the uncertain parameters are uniformly distributed in a box. We showcase this functionality via an example on a stochastic best box problem in Section EC.1.2.

7.3. Limited Memory Decision Rules
For problems involving long time-horizons (> 100), the LDR/CDR and PWL/PWC decision rules can become computationally expensive. Limited memory decision rules approximate adaptive decisions by linear functions of the recent history of observations. The memory parameter of the ROCPPConstantDR, ROCPLinearDR, and ROCPPWDR can be used in ROC++ to trade-off optimality with computational complexity.
7.4. The ROPy Python Interface

We use pybind11, a lightweight header-only library, to create Python bindings of the C++ code. With the Python interface we provide, users can generate a Python library called ROPy, which contains all the functions needed for creating decision variables, constraints, and models supported by ROC++. ROPy also implements the dynamic behavior via strategy pattern. It includes all reformulation strategies of ROC++ and uses the reformulation orchestrator to apply the strategy sequentially. The concise grammar of Python makes ROPy easy to use. Code extendability is guaranteed by pybind11. Developers may directly extend the library ROPy (by e.g., deriving new classes) in Python without looking into the C++ code or by rebuilding the library after making changes in C++. ROPy code to all the examples in our paper can be found in our GitHub repository.

8. Conclusion

We proposed ROC++, an open-source platform for automatic robust optimization in C++ that can be used to solve single- and multi-stage robust optimization problems with binary and/or real-valued variables, with exogenous and/or endogenous uncertainty set, and with exogenous and/or endogenous information discovery. ROC++ is very easy to use thanks to operator overloading in C++ that allows users to enter constraints to a ROC++ model in the same way that they look on paper and thanks to the strategy pattern that allows users to select the reformulation strategy to employ at runtime. ROC++ is also very easy to extend thanks to extensive use of inheritance throughout and thanks to the numerous hooks that are available (e.g., new reformulation strategies, new solvers). We also provide a Python library to ROC++, named ROPy. ROPy is easy to extend either directly in Python or in C++. We believe that ROC++ can facilitate the use of robust optimization among both researchers and practitioners.

Some desirable extensions to ROC++ that we plan to include in future releases are test capability, support for distributionally robust optimization, polynomial decision rules, and constraint sampling. We also hope to generalize the classes of stochastic programming problems that can be addressed by ROC++ by adding support for problems where the mean and covariance of the uncertain parameters are known.
Notes

1https://www.ibm.com/analytics/cplex-optimizer
2https://www.gurobi.com
3See https://www.gurobi.com.
4See https://www.ibm.com/analytics/cplex-optimizer.
7See https://www.gnu.org/software/glpk/.
8See https://github.com/coin-or/Cbc
9See https://github.com/coin-or/Clp
10See https://www.coin-or.org
11See https://www.scipopt.org.
12See https://ampl.com
15See https://pypi.org/project/PuLP/.
16See http://www.pyomo.org.
17See https://projects.coin-or.org/FlopC++.
18See https://projects.coin-or.org/Rehearse
19https://projects.coin-or.org/Gravity
20See https://github.com/coin-or/Osi
22See https://www.solver.com/.
23See http://www.maximalsoftware.com/.
25See https://pypi.org/project/stochoptim/
26See https://github.com/kibaekkim/StochJuMP.jl
27See https://github.com/martinbiel/StochasticPrograms.jl
28See https://odow.github.io/SDDP.jl/stable/
29See https://github.com/reganbaucke/JuDGE.jl
30See https://github.com/coin-or/Smi from COIN-OR.
34See https://www.gams.com/latest/docs/UG_EMP_SP.html.
36See https://github.com/g0900971/RobustOptimization.
Acknowledgments
This work was supported in part by the Operations Engineering Program of the National Science Foundation under NSF Award No. 1763108. The authors are grateful to Dimitrios Elissaios, Daniel Kuhn, Berç Rustem, and Wolfram Wiesemann, for valuable discussions that helped shape this work.

References


The strategy pattern is a behavioral design pattern used to, different ways at different times.


**E-Companion**

**EC.1. Companion to Section 5: Additional Examples**

**EC.1.1. Retailer-Supplier Flexible Commitment Contracts (RSFC)**

We model the two-echelon, multiperiod supply chain problem, known as the retailer-supplier flexible commitment (RSFC) problem from Ben-Tal et al. (2005) in ROC++.

**EC.1.1.1. Problem Description** We consider a finite planning horizon of $T$ periods, $\mathcal{T} := \{1, \ldots, T\}$. At the end of each period $t \in \mathcal{T}$, the demand $\xi_t \in \mathbb{R}_+$ for the product faced during that period is revealed. We collect the demands for all periods in the vector $\xi := (\xi_1, \ldots, \xi_T)$. We assume that the demand is known to belong to the box uncertainty set

$$\Xi := \{\xi \in \mathbb{R}^T : \xi \in [\xi(1-\rho), \xi(1+\rho)]\},$$

where $\bar{\xi} := e\xi$, $\xi$ is the nominal demand, and $\rho \in [0, 1]$ is the budget of uncertainty.

As the time of revelation of the uncertain parameters is exogenous, the information base, encoded in the vectors $w_t \in \{0, 1\}^T$, $t = 0, \ldots, T$, is an input of the problem (data). In particular, it is defined through $w_0 := 0$ and $w_t := \sum_{\tau=1}^t e_\tau$ for each $t \in \mathcal{T}$: the information base for time $t+1$ only contains the demand for the previous periods $\tau \in \{1, \ldots, t\}$.

At the beginning of the planning horizon, the retailer holds an inventory $x^i_1$ of the product (assumed to be known). At that point, they must specify their commitments $y^c := (y^c_1, \ldots, y^c_T)$, where $y^c_t \in \mathbb{R}_+$ represents the amount of the product that they forecast to order at the beginning of time $t \in \mathcal{T}$ from the supplier. A penalty cost $c^i_{t+}$ (resp. $c^i_t-$) is incurred for each unit of increase (resp. decrease) between the amounts committed for times $t$ and $t-1$. The amount committed for the last order before the beginning of the planning horizon is given by $y^c_0$. At the beginning of each period, the retailer orders a quantity $y^o_t(\xi) \in \mathbb{R}_+$ from the supplier at unit cost $c^o_t$. These orders may deviate from the commitments made at the beginning of the planning horizon; in this case, a cost $c^p_{t+}$ (resp. $c^p_t-$) is incurred for each unit that orders $y^o_t(\xi)$ overshoot (resp. undershoot) the plan $y^c_t$.

Given this notation, the inventory of the retailer at time $t+1$, $t \in \mathcal{T}$, is expressible as

$$x^i_{t+1}(\xi) = x^i_t(\xi) + y^o_t(\xi) - \xi_t.$$

A holding cost $c^h_{t+1}$ is incurred for each unit of inventory held on hand at time $t+1$, $t \in \mathcal{T}$. A shortage cost $c^s_{t+1}$ is incurred for each unit of demand lost at time $t+1$, $t \in \mathcal{T}$. 


The amounts of the product that can be ordered in any given period are constrained by lower and upper bounds denoted by \( y_o^t \) and \( y_c^t \), respectively. Similarly, cumulative orders up to and including time \( t \in T \) are constrained to lie in the range \( y_{co}^t \) to \( y_{co}^t \). Thus, we have:

\[
y_o^t \leq y_o^t(\xi) \leq \overline{y}_o^t \quad \text{and} \quad y_{co}^t \leq \sum_{\tau=1}^{t} y_{\tau}^o(\xi) \leq \overline{y}_{co}^t.
\]

The objective of the retailer is to minimize their worst-case (maximum) costs. We introduce three sets of auxiliary variables used to linearize the objective function. For each \( t \in T \), we let \( y_{dc}^t \) represent the smallest number that exceeds the costs of deviating from commitments, i.e.,

\[
y_{dc}^t \geq c_{dc}^t + (y_c^t - y_{c}^{t-1}) \quad \text{and} \quad y_{dc}^t \geq c_{dc}^t - (y_{c}^{t-1} - y_{c}^{t}).
\]

For each \( t \in T \) and \( \xi \in \Xi \), we denote by \( y_{dp}^t(\xi) \) the smallest number that exceeds the deviations from the plan for time \( t \), i.e.,

\[
y_{dp}^t(\xi) \geq c_{dp}^t + (y_o^t(\xi) - y_c^t) \quad \text{and} \quad y_{dp}^t(\xi) \geq c_{dp}^t - (y_c^t - y_o^t(\xi)).
\]

Similarly, for each \( t \in T \) and \( \xi \in \Xi \), we denote by \( y_{hs}^{t+1}(\xi) \) the smallest number that exceeds the overall holding and shortage costs at time \( t + 1 \), i.e.,

\[
y_{hs}^{t+1}(\xi) \geq c_{hs}^{t+1} x_{i+1}^t(\xi) \quad \text{and} \quad y_{hs}^{t+1}(\xi) \geq -c_{hs}^{t+1} x_{i+1}^t(\xi).
\]

The objective of the retailer is then expressible compactly as

\[
\min \max_{\xi \in \Xi} \sum_{t \in T} c_{i}^o y_{o}^t(\xi) + y_{dc}^t(\xi) + y_{dp}^t(\xi) + y_{hs}^{t+1}(\xi).
\]

The full model for this problem can be found in Electronic Companion EC.2.1.

**EC.1.1.2. Model in ROC++** We now present the ROC++ model of the RSFC problem. We assume that the data of the problem have been defined in C++. The C++ variables associated with the problem data are detailed in Table EC.1. For example, the lower bounds on the orders \( y_o^t, t \in T \), are stored in the map \texttt{OrderLB} that maps each time period to the double representing the lower bound for that period. We discuss how to construct the key elements of the problem here. The full code can be found in Electronic Companion EC.2.2.

The RSFC problem is a multi-stage robust optimization problem involving only exogenous uncertain parameters. We begin by creating a model, \texttt{RSFModel}, in ROC++ that will
Table EC.1  List of model parameters and their associated C++ variables for the RSFC problem.

<table>
<thead>
<tr>
<th>Parameter/Index</th>
<th>C++ Name</th>
<th>C++ Type</th>
<th>C++ Map Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T (t) )</td>
<td>( T (t) )</td>
<td>uint</td>
<td>NA</td>
</tr>
<tr>
<td>( x_1^i )</td>
<td>InitInventory</td>
<td>double</td>
<td>N/A</td>
</tr>
<tr>
<td>( y_0^c )</td>
<td>InitCommit</td>
<td>double</td>
<td>N/A</td>
</tr>
<tr>
<td>( \xi )</td>
<td>NomDemand</td>
<td>double</td>
<td>N/A</td>
</tr>
<tr>
<td>( \rho )</td>
<td>rho</td>
<td>double</td>
<td>N/A</td>
</tr>
<tr>
<td>( y_t^c, t \in \mathcal{T} )</td>
<td>OrderLB</td>
<td>map&lt;uint,double&gt;</td>
<td>t=1...T</td>
</tr>
<tr>
<td>( y_t^{co}, t \in \mathcal{T} )</td>
<td>CumOrderLB</td>
<td>map&lt;uint,double&gt;</td>
<td>t=1...T</td>
</tr>
<tr>
<td>( y_t^{co}, t \in \mathcal{T} )</td>
<td>CumOrderUB</td>
<td>map&lt;uint,double&gt;</td>
<td>t=1...T</td>
</tr>
<tr>
<td>( c_t^h, t \in \mathcal{T} )</td>
<td>OrderCost</td>
<td>map&lt;uint,double&gt;</td>
<td>t=1...T</td>
</tr>
<tr>
<td>( c_{t+1}^h, t \in \mathcal{T} )</td>
<td>HoldingCost</td>
<td>map&lt;uint,double&gt;</td>
<td>t=2...T+1</td>
</tr>
<tr>
<td>( c_{t+1}^s, t \in \mathcal{T} )</td>
<td>ShortageCost</td>
<td>map&lt;uint,double&gt;</td>
<td>t=2...T+1</td>
</tr>
<tr>
<td>( c_t^{dc+}, t \in \mathcal{T} )</td>
<td>CostDCp</td>
<td>map&lt;uint,double&gt;</td>
<td>t=1...T</td>
</tr>
<tr>
<td>( c_t^{dc-}, t \in \mathcal{T} )</td>
<td>CostDCm</td>
<td>map&lt;uint,double&gt;</td>
<td>t=1...T</td>
</tr>
<tr>
<td>( c_t^{dp+}, t \in \mathcal{T} )</td>
<td>CostDPp</td>
<td>map&lt;uint,double&gt;</td>
<td>t=1...T</td>
</tr>
<tr>
<td>( c_t^{dp-}, t \in \mathcal{T} )</td>
<td>CostDp</td>
<td>map&lt;uint,double&gt;</td>
<td>t=1...T</td>
</tr>
</tbody>
</table>

contain our formulation. All models are pointers to the interface class ROCPPOptModelIF. In this case, we instantiate an object of type ROCPPUncOptModel that is derived from ROCPPOptModelIF and which can model multi-stage optimization problems affected by exogenous uncertain parameters only. The first parameter of the ROCPPUncOptModel constructor corresponds to the maximum time period of any decision variable or uncertain parameter in the problem: in this case, \( T + 1 \). The second parameter of the ROCPPUncOptModel constructor is of the enumerated type uncOptModelObjType that admits two possible values: robust, which indicates a min-max objective; and, stochastic, which indicates an expectation objective. The robust RSFC problem can be initialized as:

```cpp
// Create an empty robust model with T + 1 periods for the RSFC problem
ROCPPOptModelIF_Ptr RSFCModel (new ROCPPUncMSOptModel(T+1, robust));
```

We note that in ROC++ all optimization problems are minimization problems.

Next, we discuss how to create the ROC++ variables associated with uncertain parameters and decision variables in the problem. The correspondence between variables is summarized in Table EC.2 for convenience.
<table>
<thead>
<tr>
<th>Variable</th>
<th>C++ Name</th>
<th>C++ Type</th>
<th>C++ Map Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>ξₜ, t ∈ T</td>
<td>Demand</td>
<td>map&lt;uint,ROCPPUnc_Ptr&gt;</td>
<td>t=1...T</td>
</tr>
<tr>
<td>yₜᶜ, t ∈ T</td>
<td>Orders</td>
<td>map&lt;uint,ROCPPVarIF_Ptr&gt;</td>
<td>t=1...T</td>
</tr>
<tr>
<td>yₜᶜ, t ∈ T</td>
<td>Commits</td>
<td>map&lt;uint,ROCPPVarIF_Ptr&gt;</td>
<td>t=1...T</td>
</tr>
<tr>
<td>yₜᵈₑ, t ∈ T</td>
<td>MaxDC</td>
<td>map&lt;uint,ROCPPVarIF_Ptr&gt;</td>
<td>t=1...T</td>
</tr>
<tr>
<td>yₜᵈₚ, t ∈ T</td>
<td>MaxDP</td>
<td>map&lt;uint,ROCPPVarIF_Ptr&gt;</td>
<td>t=1...T</td>
</tr>
<tr>
<td>yₜʰₛ, t ∈ T</td>
<td>MaxHS</td>
<td>map&lt;uint,ROCPPVarIF_Ptr&gt;</td>
<td>t=2...T+1</td>
</tr>
<tr>
<td>xₜ₊₁, t ∈ T</td>
<td>Inventory</td>
<td>ROCPPExp_Ptr</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table EC.2 List of model variables and uncertainties and their associated ROC++ variables for the RSFC problem.

The uncertain parameters of the RSFC problem are ξₜ, t ∈ T. We store these in the Demand map, which maps each period to the associated uncertain parameter. Each uncertain parameter is a pointer to an object of type ROCPPUnc. The constructor of the ROCPPUnc class takes two input parameters: the name of the uncertain parameter and the period when that parameter is revealed (first time stage when it is observable).

```cpp
// Create the Demand map to store the uncertain parameters of the problem
map<uint,ROCPPUnc_Ptr> Demand;

// Iterate over all time periods when there is uncertainty
for (uint t=1; t<T; t++)
    // Create the uncertain parameters, and add them to Demand
    Demand[t] = ROCPPUnc_Ptr(new ROCPPUnc("Demand_"+to_string(t),t+1));
```

The main decision variables of the RSFC problem are yᶜₜ and yₒₜ, t ∈ T. The commitment variables yᶜₜ are all static. We store these in the Commits map that maps each time period to the associated commitment decision. The order variables yₒₜ are allowed to adapt to the history of observed demand realizations. We store these in the Orders map that maps the time period at which the decision is made to the order decision for that period. In ROC++, the decision variables are pointers to objects of type ROCPPVarIF. Real-valued static (adaptive) decision variables are modelled using objects of type ROCPPStaticVarReal (ROCPPAdaptVarReal). The constructor of ROCPPStaticVarReal admits three input parameters: the name of the variable, its lower bound, and its upper bound. The constructor of ROCPPAdaptVarReal admits four input parameters: the name of the variable, the time period when the decision is made, and the lower and upper bounds.

```cpp
// Create maps to store the decision variables of the problem
map<uint,ROCPPVarIF_Ptr> Orders, Commits;  // Quantity ordered, Commitments made

// Iterate over all time periods from 1 to T
for (uint t=1; t<T; t++) {
```
The RSFC problem also involves the auxiliary variables $y_{t}^{dc}$, $y_{t}^{dp}$, and $y_{t+1}^{hs}$, $t \in \mathcal{T}$. We store the $y_{t}^{dc}$ variables in the map MaxDC. These variables are all static. We store the $y_{t}^{dp}$ variables in the map MaxDP. We store the $y_{t}^{hs}$ variables in the map MaxHS. Since the orders placed and inventories held change based on the demand realization, the variables stored in MaxDP and MaxHS are allowed to adapt to the history of observations. All maps map the index of the decision to the actual decision variable. The procedure to build these maps exactly parallels the approach above and we thus omit it. We refer the reader to lines 20-34 in Section EC.2.2 for the code to build these maps.

Having defined the model, the uncertain parameters, and the decision variables of the problem, we are now ready to formulate the constraints. To express our constraints in an intuitive fashion, we create an expression variable (type ROCPPExpr), which we call Inventory that stores the amount of inventory held at the beginning of each period. This is computed by adding to the initial inventory InitInventory the amount ordered at each period and subtracting the demand faced. Similarly, we create an ROCPPExpr to store the cumulative orders placed. This is obtained by adding orders placed at each period. Constraints can be created using the operators “$<$”, “$>$”, or “$==$” and added to the problem using the add_constraint() function. We show how to construct the cumulative order constraints and the lower bounds on the shortage and holding costs. The code to add the remaining constraints can be found in lines 53-67 of Section EC.2.2.

```c++
// Create the commitment variables (these are static)
Commits[t] = ROCPPVarIF_Ptr(new ROCPPStaticVarReal("Commit_{t}+to_string(t),0.));
if (t==1) // In the first period, the order variables are static
    Orders[t] = ROCPPVarIF_Ptr(new ROCPPStaticVarReal("Order_{t}+to_string(t),
        OrderLB[t],OrderUB[t]);
else // In the other periods, the order variables are adaptive
    Orders[t] = ROCPPVarIF_Ptr(new ROCPPAdaptVarReal("Order_{t}+to_string(t),t,
        OrderLB[t],OrderUB[t]);
}

// Create the constraints of the problem
// Create an expression for the amount of inventory held and initialize it
ROCPPExpr_Ptr Inventory = new ROCPPExpr();
Inventory = Inventory + InitInventory;
// Create an expression for the cumulative amount ordered
ROCPPExpr_Ptr CumOrders = new ROCPPExpr();
// Iterate over all time periods and add the constraints to the problem
for (uint t=1; t<T; t++) {
    // Create the upper and lower bounds on the cumulative orders
    CumOrders = CumOrders + Orders[t];
    RSFCModel->add_constraint(CumOrders >= CumOrderLB[t]);
```
The objective function of the RSFC problem consists in minimizing the sum of all costs over time. We create the ROCPPExpr expression RSFCObj to which we add all terms by iterating over time. We then set RSFCObj as the objective function of the RSFCModel model by using the set_objective() function.

Finally, we create a box uncertainty set for the demand.

Having formulated the RSFC problem in ROC++, we turn to solving it.

**EC.1.1.3. Solution in ROC++** From Ben-Tal et al. (2005), LDRs are optimal in this case. Thus, it suffices to approximate the real-valued adaptive variables in the problem by linear decision rules, then robustify the problem using duality theory, and finally solve it using an off-the-shelf deterministic solver. This process is streamlined in ROC++.
We consider the instance of RSFC detailed in Electronic Companion EC.2.3. The following output is displayed when executing the above code on this instance.

```
===========================================================================
======================== APPROXIMATING USING LDR ==========================
===========================================================================
Total time to approximate: 0 seconds
===========================================================================
ROBUSTIFYING===========================================================================
11 of 119 constraints robustified ...
110 of 119 constraints robustified
Total time to robustify: 1 seconds
===========================================================================
```

This states that the problem has 119 constraints in total, that the time it took to approximate it was under half a second, and that under 1 second was needed to obtain its robust counterpart. Next, we showcase how optimal solutions to the problem can be retrieved in ROC++.

```
// Retrieve the optimal solution from the solver
map<string,double> optimalSln(pSolver->getSolution());
// Print the optimal decision (from the original model)
pLDRApprox->printOut(RSFCModelLDR, optimalSln, Orders[10]);
```

The following output is displayed when executing the above code.

```
Order_10 = + 0\times Demand_1 + 0\times Demand_2 + 0\times Demand_3 + 0\times Demand_4 + 0\times Demand_5 + 0\times Demand_6 + 0\times Demand_7 + 0\times Demand_8 + 1\times Demand_9 - 0.794
```

Thus, the optimal linear decision rule for the amount to order at stage 10 is \( y_{10}^o(\xi) = \xi_9 - 0.794 \) for this specific instance. To get the optimal objective value of RSFCModelLDR, we can use the following command, which returns 13531.7 in this instance.

```
double optVal(pSolver->getOptValue());
```

**EC.1.1.4. Variant: Ellipsoidal Uncertainty Set** In Ben-Tal et al. (2005), the authors also investigated ellipsoidal uncertainty sets for the demand. These take the form

\[ \Xi := \{ \xi \in \mathbb{R}_+^T : \| \xi - \bar{\xi} \|_2 \leq \Omega \}, \]
where $\Omega$ is a safety parameter. Letting $\Omega$ represent $\Omega$, this ellipsoidal uncertainty set can be used instead of the box uncertainty set by replacing lines 74-78 in the ROC++ code for the RSFC problem with the following code:

```cpp
// Create a vector that will contain all the elements of the norm term
vector<ROCPPExpr_Ptr> EllipsoidElements;
// Populate the vector with the difference between demand and nominal demand
for (uint t=1; t<T; t++)
    EllipsoidElements.push_back(Demand[t+1] - NominalDemand);
// Create the norm term
boost::shared_ptr<ConstraintTermIF> EllipsTerm(new NormTerm(EllipsoidElements));
// Create the ellipsoidal uncertainty constraint
RSFCModel->add_constraint_uncset(EllipsTerm <= Omega);
```

The solution approach from Section EC.1.1.3 applies as is with this ellipsoidal set. The time it takes to robustify the problem is again under half a second. In this case, the optimal objective value under LDRs is 14,814.3. The optimal linear decision rule is given by:

```
Order_10 = + 0.0305728*Demand_1 + 0.0567*Demand_2 + 0.0739*Demand_3 + 0.0887*
          Demand_4 + 0.101*Demand_5 + 0.115*Demand_6 + 0.142*Demand_7 + 0.179*Demand_8 +
          0.231*Demand_9 - 3.33
```

**EC.1.2. Stochastic Best Box Problem with Uncertain Observation Costs**

We consider a variant of Pandora’s Box problem, known as Best Box (BB), in which observation costs are uncertain and subject to a budget constraint. We assume that the decision-maker is interested in maximizing the expected value of the box kept.

**EC.1.2.1. Problem description**

There are $I$ boxes indexed in $I := \{1, \ldots, I\}$ that we can choose or not to open over the planning horizon $T := \{1, \ldots, T\}$. Opening box $i \in I$ incurs an uncertain cost $\xi^c_i \in \mathbb{R}_+$. Each box has an unknown value $\xi^v_i \in \mathbb{R}$, $i \in I$. The value of each box and the cost of opening it will only be revealed if the box is opened. The total cost of opening boxes cannot exceed budget $B$. At each period $t \in T$, we can either open a box or keep one of the opened boxes, earn its value (discounted by $\theta^{t-1}$), and stop the search.

We assume that box values and costs are uniformly distributed in the set $\Xi := \{\xi^v \in \mathbb{R}_+^I, \xi^c \in \mathbb{R}_+^I : \xi^v \leq \bar{\xi}^v, \xi^c \leq \bar{\xi}^c\}$, where $\bar{\xi}^v, \bar{\xi}^c \in \mathbb{R}^I$.

In this problem, the box values and costs are endogenous uncertain parameters whose time of revelation can be controlled by the box open decisions. For each $i \in I$, and $t \in T$, we let $w^v_{t,i}(\xi)$ and $w^c_{t,i}(\xi)$ indicate if parameters $\xi^v_i$ and $\xi^c_i$ have been observed on or before
time $t$. In particular $w_{t,i}(\xi) = w_{t,i}(\xi)$ for all $i, t$, and $\xi$. We assume that $w_0(\xi) = 0$ so that no box is opened before the beginning of the planning horizon. We denote by $z_{t,i}(\xi) \in \{0, 1\}$ the decision to keep box $i \in I$ and stop the search at time $t \in T$. The requirement that at most one box be opened at each time $t \in T$ and that no box be opened if we have stopped the search can be captured in a manner that parallels constraint (5). Similarly, the requirement that only one of the opened boxes can be kept can be modeled using a constraint similar to (6). The budget constraint and objective can be expressed compactly as

$$\sum_{i \in I} \xi_{i}^{c} w_{T,i}(\xi) \leq B, \quad \text{and} \quad \max \mathbb{E} \left[ \sum_{t \in T} \sum_{i \in I} \theta^{t-1} \xi_{i}^{v} z_{t,i}(\xi) \right],$$

respectively. The full model for this problem can be found in Electronic Companion EC.4.1.

### EC.1.2.2. Model in ROC++

We assume that the data, decision variables, and uncertain parameters of the problem have been defined as in Tables EC.3 and EC.4.

<table>
<thead>
<tr>
<th>Model Parameter</th>
<th>C++ Variable Name</th>
<th>C++ Variable Type</th>
<th>C++ Map Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>$B$</td>
<td>double</td>
<td>NA</td>
</tr>
<tr>
<td>$\xi_{i}^{c}$, $i \in I$</td>
<td>CostUB</td>
<td>map&lt;uint,double&gt;</td>
<td>i=1...I</td>
</tr>
<tr>
<td>$\xi_{i}^{v}$, $i \in I$</td>
<td>ValueUB</td>
<td>map&lt;uint,double&gt;</td>
<td>i=1...I</td>
</tr>
</tbody>
</table>

**Table EC.3** List of model parameters and their associated C++ variables for the BB problem. The parameters $T(t)$ and $I(i)$ are as in Table 1 and we thus omit them here.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>C++ Nm.</th>
<th>C++ Type</th>
<th>C++ Map Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_{t,i}^{c}$, $i \in I, t \in T$</td>
<td>MVcost</td>
<td>map&lt;uint,map&lt;uint,ROCPPVarIF_Ptr&gt;&gt;</td>
<td>1...T, 1...I</td>
</tr>
<tr>
<td>$w_{t,i}^{v}$, $i \in I, t \in T$</td>
<td>MVval</td>
<td>map&lt;uint,map&lt;uint,ROCPPVarIF_Ptr&gt;&gt;</td>
<td>1...T, 1...I</td>
</tr>
<tr>
<td>$\xi_{i}^{c}$, $i \in I$</td>
<td>Cost</td>
<td>map&lt;uint,ROCPPUnc_Ptr&gt;</td>
<td>i=1...I</td>
</tr>
<tr>
<td>$\xi_{i}^{v}$, $i \in I$</td>
<td>Value</td>
<td>map&lt;uint,ROCPPUnc_Ptr&gt;</td>
<td>i=1...I</td>
</tr>
</tbody>
</table>

**Table EC.4** List of model variables and uncertainties and their associated C++ variables for the BB problem. The variables $z_{i,t}, i \in I, t \in T$, are as in Table 2 and we thus omit them here.

We create a stochastic model with decision-dependent information discovery as follows.

```cpp
// Create an empty stochastic model with T periods for the BB problem
ROCPPOptModelIF_Ptr BBModel = new ROCPPOptModelDDID(T, stochastic);
```

To model the requirement that $\xi_{i}^{c}$ and $\xi_{i}^{v}$ must be observed simultaneously, the function `pair_uncertainties()` may be employed in ROC++.
To build the budget constraint we use the ROC++ expression AmountSpent.

```
for (uint i = 1; i <= I; i++)
    BBModel->pair_uncertainties(Value[i], Cost[i]);
```

```
// Constraint on the amount spent
ROCPEXPx_Ptr AmountSpent(new ROCCPExp());
for (uint i = 1; i <= I; i++)
    AmountSpent = AmountSpent + Cost[i] * MVval[T][i];
BBModel->add_constraint(AmountSpent <= B);
```

The construction of the remaining constraints and of the objective parallels that for the Pandora Box problem and we thus omit it. We refer to EC.4.2 for the full code.

**EC.1.2.3. Solution in ROC++** The BB problem is a multi-stage stochastic problem with decision-dependent information discovery, see Vayanos et al. (2011). We thus propose to solve it using PWC decision rules. We consider the instance of BB detailed in Electronic Companion EC.4.3, which has $T = 4$ and $I = 5$. To employ a breakpoint configuration $r = (1, 1, 1, 1, 3, 3, 1, 3, 1)$ for the PWC approximation, we use the following code.

```
// Construct the reformulation orchestrator
ROCPPOrchestrator_Ptr pOrch(new ROCPPOrchestrator());
// Construct the piecewise linear decision rule reformulation strategy
// Build the map containing the breakpoint configuration
map < string, uint > BPconfig;
    BPconfig["Value_1"] = 3;
    BPconfig["Value_2"] = 3;
    BPconfig["Value_4"] = 3;
ROCPPStrategy_Ptr pPWApprox(new ROCPPPWDR(BPconfig));
// Construct the robustify engine reformulation strategy
ROCPPStrategy_Ptr pRE (new ROCPPRobustifyEngine());
// Approximate the adaptive decisions using the linear/constant decision rule approximator and robustify
vector<ROCPPStrategy_Ptr> strategyVec {pPWApprox, pRE};
ROCPPOptModelIF_Ptr BBModelPWCFinal = pOrch->Reformulate(BBModel, strategyVec);
```

Under this breakpoint configuration, the optimal profit is 934.2, compared to 792.5 for the static decision rule. The optimal solution can be printed to screen using the `printOut` function, see lines 92-97 in EC.4.2. Part of the resulting output is

```
On subset 1111112131: Keep_4_4 = 1
Uncertain parameter Value_4 on subset 1111112131 is observed at stage 2
```

Thus, on subset $s = (1, 1, 1, 1, 1, 2, 1, 3, 1)$, the forth box is kept at time 4. On subset $s = (1, 1, 1, 1, 1, 2, 1, 3, 1)$, box 5 is opened at time 2 (resp. 3).
EC.2. Supplemental Material: Retailer-Supplier Problem

EC.2.1. Retailer-Supplier Problem: Mathematical Formulation

Using the notation introduced in Section EC.1.1, the robust RSFC problem can be expressed mathematically as:

\[
\begin{align*}
\text{minimize} & \quad \max_{\xi \in \Xi} \sum_{t \in T} c_t^o y_t^o(\xi) + y_t^{dc}(\xi) + y_t^{dp}(\xi) + y_{t+1}^{hs}(\xi) \\
\text{subject to} & \quad y_t^o \in \mathbb{R}_+ \quad \forall t \in T \\
& \quad y_t^o, y_t^{dc}, y_t^{dp}, y_{t+1}^{hs} \in \mathcal{L}_T^1 \quad \forall t \in T \\
& \quad x_t^{i+1}(\xi) = x_t^i(\xi) + y_t^o(\xi) - \xi_{t+1} \\
& \quad y_t^o(\xi) \leq y_t^o(\xi) \leq \bar{y}_t^o, \quad \bar{y}_t^{co} \leq \sum_{t=1}^T y_t^o(\xi) \leq \bar{y}_t^{co} \\
& \quad y_t^{dc} \geq c_t^{dc+}(y_t^o - y_{t-1}^c) \\
& \quad y_t^{dc} \geq c_t^{dc-}(y_{t-1}^c - y_t^c) \\
& \quad y_t^{dp} \geq c_t^{dp+}(y_t^c - y_t^c) \\
& \quad y_t^{dp}(\xi) \geq c_t^{dp-}(y_t^c - y_t^c(\xi)) \\
& \quad y_{t+1}^{hs}(\xi) \leq c_t^{h} x_{t+1}(\xi) \\
& \quad y_{t+1}^{hs}(\xi) \geq -c_t^{s} x_{t+1}(\xi) \\
& \quad y_t^o(\xi) = y_t^o(\xi') \\
& \quad y_t^{dc}(\xi) = y_t^{dc}(\xi') \\
& \quad y_t^{dp}(\xi) = y_t^{dp}(\xi') \\
& \quad y_{t+1}^{hs}(\xi) = y_{t+1}^{hs}(\xi') \\
\end{align*}
\]

\[\forall \xi, \xi' \in \Xi: w_{t-1} \circ \xi = w_{t-1} \circ \xi', \forall t \in T\]

The last set of constraints corresponds to non-anticipativity constraints. The other constraints are explained in Section EC.1.1.1.

EC.2.2. Retailer-Supplier Problem: Full ROC++ Code

1 // Create an empty robust model with T + 1 periods for the RSFC problem
2 ROcppOptModelIF_Ptr RSFCModel(new ROcppUncMSOptModel(T+1, robust));
3 // Create the Demand map to store the uncertain parameters of the problem
4 map<uint,ROcppUnc_Ptr> Demand;
5 // Iterate over all time periods when there is uncertainty
6 for (uint t =1; t<T; t++)
7  // Create the uncertain parameters, and add them to Demand
8   Demand[t] = ROcppUnc_Ptr(new ROcppUnc("Demand_"+to_string(t),t+1));
9 // Create maps to store the decision variables of the problem
10 map<uint,ROcppVarIF_Ptr> Orders, Commits; // Quantity ordered, Commitments made
11 // Iterate over all time periods from 1 to T
12 for (uint t =1; t<T; t++) {
13   // Create the commitment variables (these are static)
14   Commits[t] = ROcppVarIF_Ptr(new ROcppStaticVarReal("Commit_"+to_string(t),0.));
15   if (t==1) // In the first period, the order variables are static
Orders[\(t\)] = ROCPPVarIF_Ptr(new ROCPPStaticVarReal("Order_"+to_string(\(t\)), OrderLB[\(t\)], OrderUB[\(t\)]));

\textbf{else} // In the other periods, the order variables are adaptive
Orders[\(t\)] = ROCPPVarIF_Ptr(new ROCPPAdaptVarReal("Order_"+to_string(\(t\)), \(t\), OrderLB[\(t\)], OrderUB[\(t\)]));
}

map<\textit{uint},\textit{ROCPPVarIF_Ptr}> MaxDC; // Upper bound on deviation between commitments
map<\textit{uint},\textit{ROCPPVarIF_Ptr}> MaxDP; // Upper bound on deviation from plan
map<\textit{uint},\textit{ROCPPVarIF_Ptr}> MaxHS; // Upper bound on holding and shortage costs

\textbf{// Iterate over all time periods 1 to T}
for (\textit{uint} \(t=1; t<=\textit{T}; t++\)) {

\textbf{// Create upper bounds on the deviation between successive commitments}
MaxDC[\(t\)] = ROCPPVarIF_Ptr(new ROCPPStaticVarReal("MaxDC_"+to_string(\(t\))));

\textbf{// Create upper bounds on the deviation of orders from commitments}
\textbf{if} (\(t=1\)) // In the first period, these are static
MaxDP[\(t\)] = ROCPPVarIF_Ptr(new ROCPPStaticVarReal("MaxDP_"+to_string(\(t\))));
\textbf{else} // In the other periods, these are adaptive
MaxDP[\(t\)] = ROCPPVarIF_Ptr(new ROCPPAdaptVarReal("MaxDP_"+to_string(\(t\)), \(t\)));

\textbf{// Create upper bounds on holding and shortage costs (these are adaptive)}
MaxHS[\(t+1\)] = ROCPPVarIF_Ptr(new ROCPPAdaptVarReal("MaxHS_"+to_string(\(t+1\)), \(t+1\)));
}

\textbf{// Create the constraints of the problem}
\textbf{ROCPPExpr_Ptr} Inventory(new ROCPPExpr());
Inventory = Inventory + InitInventory;

\textbf{ROCPPExpr_Ptr} CumOrders(new ROCPPExpr());

\textbf{// Iterate over all time periods and add the constraints to the problem}
for (\textit{uint} \(t=1; t<=\textit{T}; t++\)) {

\textbf{// Create the upper and lower bounds on the cumulative orders}
CumOrders = CumOrders + Orders[\(t\)];
RSFCModel->add_constraint(CumOrders >= CumOrderLB[\(t\)]);
RSFCModel->add_constraint(CumOrders <= CumOrderUB[\(t\)]);

\textbf{// Update the inventory}
Inventory = Inventory + Orders[\(t\)] - Demand[\(t\)];

\textbf{// Create upper bound on shortage/holding costs}
RSFCModel->add_constraint(MaxHS[\(t+1\)] >= HoldingCost[\(t+1\)]*Inventory);
RSFCModel->add_constraint(MaxHS[\(t+1\)] >= (-1.*ShortageCost[\(t+1\)]*Inventory));
}

\textbf{// Iterate over all time periods and add the constraints to the problem}
for (\textit{uint} \(t=1; t<=\textit{T}; t++\)) {

\textbf{// Create upper bound on deviations from commitments}
RSFCModel->add_constraint(MaxDP[\(t\)] >= CostDPp*(Orders[\(t\)]-Commits[\(t\)]));
RSFCModel->add_constraint(MaxDP[\(t\)] >= -CostDPm*(Orders[\(t\)]-Commits[\(t\)]));

\textbf{// Create upper bound on deviations between commitments}
\textbf{if} (\(t=1\)) {
RSFCModel->add_constraint(MaxDC[\(t\)] >= CostDCp*(Commits[\(t\)]-InitCommit));
RSFCModel->add_constraint(MaxDC[\(t\)] >= -CostDCm*(Commits[\(t\)]-InitCommit));
}
\textbf{else} {
RSFCModel->add_constraint(MaxDC[\(t\)] >= CostDCp*(Commits[\(t\)]-Commits[\(t-1\)]));
RSFCModel->add_constraint(MaxDC[\(t\)] >= -CostDCm*(Commits[\(t\)]-Commits[\(t-1\)]));
}

\textbf{// Create an expression that will contain the objective function}
ROCPPExpr_Ptr RSFCObj(new ROCPPExpr());
// Iterate over all periods and add the terms to the objective function
for ( uint t=1; t<=T; t++)
    RSFCObj = RSFCObj + OrderCost*Orders[t] + MaxDC[t] + MaxDP[t] + MaxHS[t+1];
RSFCModel->set_objective(RSFCObj); // Add the objective to the problem
for ( uint t=1; t<=T; t++) {
    // Add the upper and lower bounds on the demand to the uncertainty set
    RSFCModel->add_constraint_uncset(Demand[t] >= NomDemand*(1.0-rho));
    RSFCModel->add_constraint_uncset(Demand[t] <= NomDemand*(1.0+rho));
}
// Construct the reformulation orchestrator
ROCPPOrchestrator_Ptr pOrch(new ROCPPOrchestrator());
// Construct the linear/constant decision rule reformulation strategy
ROCPPStrategy_Ptr pLDR(new ROCPPLinearDR());
// Construct the robustify engine reformulation strategy
ROCPPStrategy_Ptr pRE(new ROCPPRobustifyEngine());
// Approximate the adaptive decisions using the linear/constant decision rule
   approximator and robustify
vector<ROCPPStrategy_Ptr> strategyVec {pLDR, pRE};
ROCPPOptModelIF_Ptr RSFCModelLDRFinal = pOrch->Reformulate(RSFCModel, strategyVec);
// Construct the solver (in this case, use gurobi as deterministic solver)
ROCPPSolverInterface_Ptr pSolver (new ROCPPGurobi(SolverParams()));
// Solve the problem
pSolver->solve(RSFCModelLDRFinal);
// Retrieve the optimal solution from the solver
map<string, double> optimalSln (pSolver->getSolution());
// Print the optimal decision (from the original model)
pLDAprox->printOut(RSFCModelLDR, optimalSln, Orders[10]);
// Get the optimal objective value
double optVal (pSolver->getOptValue());

EC.2.3. Retailer-Supplier Problem: Instance Parameters
The parameters for the instance of the problem that we solve in Section EC.1.1.3 are provided in Table EC.5. They correspond to the data from instance W12 in Ben-Tal et al. (2005).

<table>
<thead>
<tr>
<th></th>
<th>x_i</th>
<th>y_i</th>
<th>ρ</th>
<th>y_i^o</th>
<th>y_i^co</th>
<th>y_i^c</th>
<th>c_t</th>
<th>c_{t+1}</th>
<th>c_{t+dc}</th>
<th>c_{t+dp}</th>
<th>c_{t+dp-}</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>12</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>10%</td>
<td>0</td>
<td>200</td>
<td>0</td>
<td>200t</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Table EC.5 Parameters for the instance of the RSFS problem that we solve in Section EC.1.1.3.
EC.3. Supplemental Material: Robust Pandora’s Box Problem

EC.3.1. Robust Pandora’s Box Problem: Mathematical Formulation

Using the notation introduced in Section 5.1, the robust PB problem can be expressed mathematically as:

\[
\begin{align*}
\text{maximize} & \quad \min_{\xi \in \Xi} \sum_{t \in T} \sum_{i \in I} \theta_t^{-1} \xi_t z_{t,i}(\xi) - c_i(w_{t,i}(\xi) - w_{t-1,i}(\xi)) \\
\text{subject to} & \quad z_{t,i}, w_{t,i} \in \{0, 1\} \quad \forall t \in T, \forall i \in I \\
& \quad \sum_{i \in I} (w_{t,i}(\xi) - w_{t-1,i}(\xi)) \leq 1 - \sum_{\tau=1}^{t} z_{\tau,i}(\xi) \\
& \quad z_{t,i}(\xi) \leq w_{t-1,i}(\xi) \quad \forall i \in I \\
& \quad z_{t,i}(\xi) = z_{t,i}(\xi') \quad \forall \xi, \xi' \in \Xi : w_{t-1,i}(\xi) = w_{t-1,i}(\xi'), \forall i \in I, \forall t \in T.
\end{align*}
\]

The last set of constraints in this problem are non-anticipativity constraints. The other constraints are explained in Section 5.1.

EC.3.2. Robust Pandora’s Box Problem: Full ROC++ Code

```cpp
// Create an empty robust model with T periods for the PB problem
ROCPPOptModelIF_Ptr PBModel(new ROCPPOptModelDDID(T, robust));
// Create empty maps to store the uncertain parameters
map <uint, ROCPPUnc_Ptr> Value, Factor;
for (uint i = 1; i <= I; i++)
    // Create the uncertainty associated with box i and add it to Value
    Value[i] = ROCPPUnc_Ptr (new ROCPPUnc("Value_"+to_string(i)));
for (uint m = 1; m <= M; m++)
    // The risk factors are not observable
    Factor[m] = ROCPPUnc_Ptr (new ROCPPUnc("Factor_"+to_string(m) ,1, false));
map <uint, map <uint, ROCPPVarIF_Ptr>> MeasVar;
for (uint i = 1; i <= I; i++) {
    // Create the measurement variables associated with the value of box i
    PBModel->add_ddu(Value[i], 1, T, obsCost[i]);
    // Get the measurement variables and store them in MeasVar
    for (uint t = 1; t <= T; t++)
        MeasVar[t][i] = PBModel->getMeasVar(Value[i]->getName(), t);
}
map <uint, map <uint, ROCPPVarIF_Ptr>> Keep;
for (uint t = 1; t <= T; t++) {
    for (uint i = 1; i <= I; i++) {
        if (t == 1) // In the first period, the Keep variables are static
            Keep[t][i] = ROCPPVarIF_Ptr (new ROCPPStaticVarBool("Keep_"+to_string(t) +"_"+to_string(i)));
        else // In the other periods, the Keep variables are adaptive
            Keep[t][i] = ROCPPVarIF_Ptr (new ROCPPAdaptVarBool("Keep_"+to_string(t) +"_"+to_string(i), t));
    }
}
```
// Create the constraints and add them to the problem
ROCPPExpr_Ptr StoppedSearch(new ROCPPExpr());
for (uint t = 1; t <= T; t++) {
    // Create the constraint that at most one box be opened at t (none if the
    // search has stopped)
    ROCPPExpr_Ptr NumOpened(new ROCPPExpr());
    // Update the expressions and and the constraint to the problem
    for (uint i = 1; i <= I; i++) {
        StoppedSearch = StoppedSearch + Keep[t][i];
        if (t>1)
            NumOpened = NumOpened + MeasVar[t][i] - MeasVar[t-1][i];
        else
            NumOpened = NumOpened + MeasVar[t][i];
    }
    PBModel->add_constraint ( NumOpened <= 1. - StoppedSearch );
    // Constraint that only one of the open boxes can be kept
    for (uint i = 1; i <= I; i++)
        PBModel->add_constraint ( (t>1) ? (Keep[t][i] <= MeasVar[t-1][i]) : (Keep[t ][i] <= 0.));
}
// Create the uncertainty set constraints and add them to the problem
// Add the upper and lower bounds on the risk factors
for (uint m = 1; m <= M; m++) {
    PBModel->add_constraint_uncset(Factor[m] >= -1.0);
    PBModel->add_constraint_uncset(Factor[m] <= 1.0);
}
// Add the expressions for the box values in terms of the risk factors
for (uint i = 1; i <= I; i++) {
    ROCPPExpr_Ptr ValueExpr(new ROCPPExpr());
    for (uint m = 1; m <= M; m++)
        ValueExpr = ValueExpr + RiskCoeff[i][m]*Factor[m];
    PBModel->add_constraint_uncset ( Value[i] == (1.+0.5*ValueExpr) * NomVal[i] );
}
// Create the objective function expression
ROCPPExpr_Ptr PBObj(new ROCPPExpr());
for (uint t = 1; t <= T; t++) {
    for (uint i = 1; i <= I; i++)
        PBObj = PBObj + pow(theta,t-1)*Value[i]*Keep[t][i];
    // Set objective (multiply by -1 for maximization)
    PBModel->set_objective(-1.0*PBObj);
}
// Construct the reformulation orchestrator
ROCPPOrchestrator_Ptr pOrc(new ROCPPOrchestrator());
// Construct the finite adaptability reformulation strategy with 2 candidate
// policies in the each time stage
ROCPPStrategy_Ptr pKadaptStrategy(new ROCPPKAdapt(Kmap));
// Construct the robustify engine reformulation strategy
ROCPPStrategy_Ptr pRE (new ROCPPRobustifyEngine());
// Construct the linearization strategy based on big M constraints
ROCPPStrategy_Ptr pBTR (new ROCPPBTR_bigM());
// Approximate the adaptive decisions using the linear/constant decision rule
// approximator and robustify
vector<ROCPPStrategy_Ptr> strategyVec {pKadaptStrategy, pRE, pBTR};
ROCPPOptModelIF_Ptr PBModelKAadapt = pOrc->Reformulate(PBModel, strategyVec);
// Construct the solver and solve the problem
ROCPPSolver_Ptr pSolver (new ROCPPGurobi(SolverParams()));
pSolver->solve(PBModelKAadapt);
// Retrieve the optimal solution from the solver
map<string, double> optimalSln(pSolver->getSolution());

// Print decision rules for variable Keep_4_2 from the original problem automatically
ROCPPKAdapt_Ptr pKadapt = static_pointer_cast<ROCPPKAdapt>(pKadaptStrategy);
pKadapt->printOut(PBModel, optimalSln, Keep[4][2]);

// Prints the observation decision for uncertain parameter Value_2
pKadapt->printOut(PBModel, optimalSln, Value[2]);

EC.3.3. Robust Pandora’s Box Problem: Instance Parameters

The parameters for the instance of the robust Pandora’s box problem that we solve in Section 5.3 are provided in Table EC.6.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\theta,T,I,M)$</td>
<td>(1,4,5,4)</td>
</tr>
<tr>
<td>$c$</td>
<td>(0.69,0.43,0.01,0.91,0.64)</td>
</tr>
<tr>
<td>$\xi$</td>
<td>(5.2,8,19.4,9.6,13.2)</td>
</tr>
<tr>
<td>$\Phi$</td>
<td>$\begin{bmatrix} 0.17 &amp; -0.7 &amp; -0.13 &amp; -0.6 \ 0.39 &amp; 0.88 &amp; 0.74 &amp; 0.78 \ 0.17 &amp; -0.6 &amp; -0.17 &amp; -0.84 \ 0.09 &amp; -0.07 &amp; -0.52 &amp; 0.88 \ 0.78 &amp; 0.94 &amp; 0.43 &amp; -0.58 \end{bmatrix}$</td>
</tr>
</tbody>
</table>

Table EC.6 Parameters for the instance of the PB problem that we solve in Section 5.3.
EC.4. Supplemental Material: Stochastic Best Box Problem

EC.4.1. Stochastic Best Box: Problem Formulation

Using the notation introduced in Section EC.1.2, the BB problem can be expressed mathematically as:

\[
\text{maximize } \mathbb{E} \left[ \sum_{t \in T} \sum_{i \in I} \theta^{t-1} \xi_{t,i}(\xi) \right] \\
\text{subject to } \begin{cases} 
  z_{t,i}, w_{t,i}^c, w_{t,i}^v \in \{0, 1\} & \forall t \in T, \forall i \in I \\
  w_{t,i}^c(\xi) = w_{t,i}^v(\xi) & \forall t \in T, \forall i \in I \\
  \sum_{i \in I} \xi_i^c w_{t,i}^y(\xi) \leq B \\
  \sum_{i \in I} (w_{t,i}^y(\xi) - w_{t-1,i}^y(\xi)) \leq 1 - \sum_{\tau=1}^{t} z_{t,i}(\xi) & \forall t \in T, \xi \in \Xi \\
  z_{t,i}(\xi) \leq w_{t-1,i}^y(\xi) & \forall i \in I \\
  z_{t,i}(\xi) = z_{t,i}(\xi') & \forall i \in I, \forall t \in T. 
\end{cases}
\]

The first set of constraints stipulates that \(\xi_i^c\) and \(\xi_i^v\) must be observed simultaneously. The second set of constraints is the budget constraint. The third set of constraints stipulates that at each stage, we can either open a box or stop the search, in which case we cannot open a box in the future. The fourth set of constraints ensures that we can only keep a box that we have opened. The last set of constraints correspond to decision-dependent non-anticipativity constraints.

EC.4.2. Stochastic Best Box Problem: Full ROC++ Code

```cpp
// Create an empty stochastic model with T periods for the BB problem
ROCPPoptModelIF_Ptr BBModel(new ROCPPoptModelDDID(T, stochastic));
map<uint, ROCPPUnc_Ptr> Value, Cost;
for (uint i = 1; i <= I; i++) {
    // Create the value and cost uncertainties associated with box i
    Value[i] = ROCPPUnc_Ptr(new ROCPPUnc("Value_"+to_string(i)));
    Cost[i] = ROCPPUnc_Ptr(new ROCPPUnc("Cost_"+to_string(i)));
}
// Create the measurement decisions and pair the uncertain parameters
map<uint, map<uint, ROCPPVarIF_Ptr>> MVcost, MVval;
for (uint i = 1; i <= I; i++) {
    // Create the measurement variables associated with the value of box i
    BBModel->add_ddu(Value[i], 1, T, obsCost);
    // Create the measurement variables associated with the cost of box i
    BBModel->add_ddu(Cost[i], 1, T, obsCost);
}
// Get the measurement variables and store them in MVval and MVcost
```
```cpp
for (uint t = 1; t <= T; t++) {
    MVval[t][i] = BBModel->getMeasVar(Value[i]->getName(), t);
    MVcost[t][i] = BBModel->getMeasVar(Cost[i]->getName(), t);
}

// Pair the uncertain parameters to ensure they are observed at the same time
for (uint i = 1; i <= I; i++)
    BBModel->pair_uncertainties(Value[i], Cost[i]);

// Create the keep decisions
map< uint , map< uint , ROCPPVarIF_Ptr > > Keep;
for (uint t = 1; t <= T; t++) {
    for (uint i = 1; i <= I; i++)
        if (t == 1) // In the first period, the Keep variables are static
            Keep[t][i] = ROCPPVarIF_Ptr(new ROCPPStaticVarBool("Keep_"+to_string(t)+"_"+to_string(i)));
        else // In the other periods, the Keep variables are adaptive
            Keep[t][i] = ROCPPVarIF_Ptr(new ROCPPAdaptVarBool("Keep_"+to_string(t)+"_"+to_string(i), t));
}

// Create the constraints and add them to the problem
ROCPPExpr_Ptr StoppedSearch(new ROCPPExpr());
for (uint t = 1; t <= T; t++) {
    // Create the constraint that at most one box be opened at t (none if the search has stopped)
    ROPCPExpr_Ptr NumOpened(new ROCPPExpr());
    // Update the expressions and the constraint to the problem
    for (uint i = 1; i <= I; i++)
        if (t>1) NumOpened = NumOpened + MVval[t][i] - MVval[t-1][i];
        else NumOpened = NumOpened + MVval[t][i];
    BBModel->add_constraint(NumOpened <= 1. - StoppedSearch);
    // Constraint that only one of the open boxes can be kept
    for (uint i = 1; i <= I; i++)
        BBModel->add_constraint((t>1) ? (Keep[t][i] <= MVval[t-1][i]) : (Keep[t][i] <= 0.));
}

// Constraint on the amount spent
ROCPPExpr_Ptr AmountSpent(new ROCPPExpr());
for (uint i = 1; i <= I; i++)
    AmountSpent = AmountSpent + Cost[i] * MVval[T][i];
BBModel->add_constraint(AmountSpent <= B);

// Create the uncertainty set constraints and add them to the problem
for (uint i = 1; i <= I; i++) {
    // Add the upper and lower bounds on the values
    BBModel->add_constraint_uncset(Value[i] >= 0.);
    BBModel->add_constraint_uncset(Value[i] <= ValueUB[i]);
    // Add the upper and lower bounds on the costs
    BBModel->add_constraint_uncset(Cost[i] >= 0.);
    BBModel->add_constraint_uncset(Cost[i] <= CostUB[i]);
}

// Create the objective function expression
ROCPPExpr_Ptr BBObj(new ROCPPExpr());
```

for (uint t = 1; t <= T; t++)
    for (uint i = 1; i <= I; i++)
        BBObj = BBObj + pow(theta, t-1) * Value[i] * Keep[t][i];
// Set objective (multiply by -1 for maximization)
BBModel->set_objective(-1.0*BBObj);
// Construct the reformulation orchestrator
ROCPPOrchestrator_Ptr pOrch(new ROCPPOrchestrator());
// Construct the piecewise linear decision rule reformulation strategy
// Build the map containing the breakpoint configuration
map<string,uint> BPconfig;
BPconfig["Value_1"] = 3;
BPconfig["Value_2"] = 3;
BPconfig["Value_4"] = 3;
ROCPPStrategy_Ptr pPWApprox(new ROCPPPWDR(BPconfig));
// Construct the robustify engine reformulation strategy
ROCPPStrategy_Ptr pRE(new ROCPPRobustifyEngine());
// Approximate the adaptive decisions using the linear/constant decision rule approximator and robustify
vector<ROCPPStrategy_Ptr> strategyVec {pPWApprox, pRE};
ROCPPOptModelIF_Ptr BBModelPWCFinal = pOrch->Reformulate(BBModel, strategyVec);
// Construct the solver; in this case, use the gurobi solver as a deterministic solver
ROCPPSolverInterface_Ptr pSolver(new ROCPPGurobi(SolverParams()));
// Solve the problem
pSolver->solve(BBModelPWCFinal);
// Retrieve the optimal solution from the solver
map<string,double> optimalSln(pSolver->getSolution());
// Print the optimal decision (from the original model)
ROCPPPWDR_Ptr pPWApproxDR = static_pointer_cast<PiecewiseDecisionRule>(pPWApprox);
pPWApproxDR->printOut(BBModel, optimalSln, Keep[3][5]);
pPWApproxDR->printOut(BBModel, optimalSln, Value[5]);

EC.4.3. Stochastic Best Box: Instance Parameters
The parameters for the instance of the stochastic best box problem that we solve in Section EC.1.2.3 are provided in Table EC.7.

<table>
<thead>
<tr>
<th>T</th>
<th>I</th>
<th>B</th>
<th>θ</th>
<th>ξ^c</th>
<th>ξ^v</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>163</td>
<td>1</td>
<td>(40,86,55,37,30)</td>
<td>(1030,1585,971,971,694)</td>
</tr>
</tbody>
</table>

Table EC.7 Parameters for the instance of the BB problem that we solve in Section EC.1.2.3.