Caroline Abidoun Adeola ANJORIN
⟨ ca106@doc.ic.ac.uk ⟩

Department of Computing, Imperial College London

JADA

A Modelling Language for Decision-making Under Uncertainty

MENG INDIVIDUAL FINAL YEAR PROJECT

Supervisor
Dr. Daniel KUHN
⟨ dkuhn@doc.ic.ac.uk ⟩

Second Marker
Professor Berç RUSTEM
⟨ br@doc.ic.ac.uk ⟩

Abstract

In the context of linear optimisation with unknown parameters, George B. Dantzig first introduced *Stochastic Programming* in his 1955 paper[1] as a framework for modelling optimisation problems characterised by recourse decisions. Since its conception, the framework has become the progressive approach to decision-making under uncertainty.

Stochastic programming problems by nature are dynamic, which makes the computational complexity of the algorithms to solve such problems #P-hard[3]. An innovative approach, to overcome the computational intractability of stochastic programs, has been to radically simplify the recourse decisions to affine functional forms termed *linear decision rules.* Despite providing scalability to multi-stage models, this method is not widely used. The prevailing issue is the hindrance placed on the modeller to derive the conic program approximations.

This project is driven by the aim to alleviate the burden placed on the modeller, and to make the linear decision rule approximation approach widely accessible to industrial modellers, whereby system-specific knowledge should be sufficient for all intents and purposes. To achieve this, we propose to design an algebraic modelling language for intuitively describing stochastic programming models in an expressible format. A parser will also be written to read models specified in this standardised format. By representing the parsed input in a highly condensed and efficient structure, we can efficiently generate and solve the conservative and progressive conic programming instances. The computed solutions of these auto-generated linear programs will provide the modeller with the upper and lower bounds of the true optimal decisions, which can then be used to quantify the loss of optimality incurred.

Project Archive: `http://www.doc.ic.ac.uk/~ca106/jada.tgz/`

Acknowledgements

I would like to thank Dr. Daniel Kuhn for his continuous encouragement, help and guidance during the completion of this project. As well as taking the time to ensure I understood the different complex mathematical concepts, his challenging questions, recommended readings, and anecdotal examples have helped me develop a growing interest in the field of optimisation theory.

I also wish to acknowledge Angelos Georghiou and Wolfram Wiesemann for their patience, generous support, and critical review of the technical implementation.

I would also like to thank Professor Berç Rustem for reviewing my discussion of the background material, and for giving me informative advice concerning the proposed methodology for evaluating the final product.

Lastly, I wish to thank my mom - my support system and motivation. There are not enough words to express how grateful I am for her unconditional love and unwavering belief in me. I dedicate this project to her.

Fi gbogbo àyà rẹ gbẹkẹle Oluwa; mà si ṣe tẹ̀ si ìmọ̀ ara rẹ.

(*Trust in the Lord with all your heart and lean not on your own understanding;*)

Mọ ọn ni gbogbo ọ̀nà rẹ: oun o si maa tọ ipa-ọ̀nà rẹ.

(*in all your ways acknowledge Him, and He will make your paths straight.*)

Ówe (*Proverbs*) 3:5-6

Table of Contents

1 Introduction

We often regard decision-making as an optimisation problem that takes place over finite time. Approaches to selecting the optimal decisions for such a problem often begin with formulating the underlying model as a deterministic optimisation problem, whereby all the parameters are known to the decision-maker. While this methodology is plausible, it will obviously be limited in its applicability to real, every-day decision-making problems. In reality, we as human-beings are required to choose the best-course of action in situations subject to an indeterminism that emerges from inaccurate measurements, unavailable data or impalpable outcomes. Thus, our choices become measurable functions of the random elements, and we term such choices as *recourse decisions* or *decision rules*.

1.1 A Historial Perspective

In the context of linear optimisation with unknown parameters, George B. Dantzig first introduced *Stochastic Programming* in his 1955 paper [1] as a framework for modelling optimisation problems characterised by recourse decisions. In 1962, Benders formalised a method for optimisation under uncertainty[11]. Since its conception, stochastic programming has become the *modus operandi* for decision-making under uncertainty, and has been expended in a miscellany of application areas such as finance[5], manufacturing[13], transportation[14] and economic policy[15], to name a few. When compared with modelling techniques such as statistical decision analysis, in a large number of application areas, stochastic programming has been proven to be a far superior paradigm to its deterministic predecessors for modelling optimisation problems.

Stochastic optimisation problems are characteristically dynamic, which makes the computational complexity of the algorithms to solve such problems #P-hard[3]. In light of this, the traditional approach has been to substitute the underlying process with a discretized approximation known as a *scenario-tree*. In this compact representation, the decision-maker is able represent the multitude of possible paths as branches that spurt from observations of random events. However, this method heavily relies on the modeller possessing the exact knowledge of the underlying probabilities, and reality informs us that these scenario probabilities are considerably complex to calculate accurately[6]. While we could use prior knowledge or heuristics to specify a distribution, the variables representing the optimal decision policies might be perturbed by such assumptions. Another limitation of the scenario-tree approximation is that scenario trees grow exponentially with the number of stages.

In consideration of the above, rather than replacing the underlying process with a discrete stochastic process, the functional form of the decision rules can be radically simplified to *linear decision rules*. One of the advantages of performing this simplification is that the problem formulation can now be scaled to decision-making problems with multiple stages. However, this benefit arises at the expense of accuracy. By constraining the feasible region to those decision rules that are affinely dependent on the random parameters, we impede our opportunities of finding the actual optimal decision. As a result, the best solution found using linear decision rules may not be reflective of the true optimality with respect to the original problem.

1.2 Recent Applications of Optimisation Under Uncertainty

Early applications of stochastic programming include Ferguson and Dantzig's airline fleet-assignment model with stochastic demand[4], which was developed after previously formulating the same model for deterministic demand. This stochastic programming framework models how one should designate particular planes to routes of scheduled flights. In this model, the objective is to maximise the expected revenue in accordance with a probability distribution representing the demand for passengers on each journey. Since this first application, stochastic programming has manifested in different sectors of industry. We briefly present a few examples below.

Sport (Batting Order)

Stochastic programming also has an application in baseball. The optimisation problem in question relates to how we can compute the optimal order in which a group of nine outfield-players should bat, where such an optimal order can increase a team's total number of wins by three per season. To contexualise the significance of this, the Major League Baseball playoffs provide huge financial rewards and remarkably 10% of the teams missed the 1998 playoffs by three or less wins[9]. Thus, baseball stochastic models capturing the uncertainty in skill measurement can been used to robustly optimise a heuristic for batting sequences[9].

Aeronautics (Transonic Airfoil Optimization for Low Drag)

In 2002, a group of aerodynamicists at NASA initiated a research effort to decrease the drag of an airfoil while simultaneously maintaining lift[8]. The parameter of uncertainty in this application is the ratio of the velocity of an aircraft to the velocity of sound in the gas. This ratio is also known as the *Mach number*[7]. The objective depended on minimising the mean and the standard deviation of the drag, a function of the Mach number, while the variables representing the lift were deterministically constrained.

Health Care (Surgery Scheduling)

The uncertainty in surgery durations means that scheduling operating rooms can be a complex problem. So, if surgery operations exceed the expected durations, then potentially all the surgeries scheduled for that day can be impacted with delayed starts. Bearing in mind that operating rooms generate the most expenditures, but also the greatest revenues for hospitals, consequences of delayed surgeries can manifest by extra costs in hiring staff for overtime. For this reason, stochastic optimisation models have been used, in conjunction with practical-experience-based techniques, to determine operating room schedules to mitigate the consequences of uncertain surgery durations[10].

1.3 Motivation

Within this section, we aim to address the main idea that underpins this project by describing the problem and by discussing the current state-of-the-art approaches to tackling the said problem.

1.3.1 The Problem

The decision rule approximation approach provides the ultimate benefit of scalability, but despite some automation being achieved, the ultimate drawback is that a large part of the model processing is still required to be manually performed by the modeller. For example, the manual processing requires an in-depth knowledge about probability and optimisation theory to formulate the decision rule approximations. This is in addition to a computer science background to guide the algorithmic design, code implementation and use of optimisation software to formulate and solve instances of the generated tractable conic programs. The requirement for the modeller to have expert knowledge in all of the mentioned disciplines to achieve their end goal is unrealistic and discouraging. The reality is that the industrial modellers only possess expert knowledge of the *physical* system to be modelled, which theoretically speaking should be sufficient.

1.3.2 Current State of Affairs for Algebraic Modelling Languages

Stochastic programming on a large-scale goes beyond applying an algorithm to solve an optimisation problem. Prior to optimising an objective function, the underlying model must be converted to an internal form that is communicable to a solver routine. In the mathematical programming community, *algebraic modelling languages* (AML) have been well received as a vocabulary for expressing these underlying models, whereby stochastic programming models can be formulated by directly defining their equivalent deterministic model. Besides the possible knowledge barrier and errancies due to manually describing such a model, stating the deterministic equivalent can be cumbersome and resource consumptive. This is owing to the fact that the sizes of stochastic programming problems are exponentially proportional to the number of random variables and stages. In the subsequent sections, we discuss two AML implementations named SMPS and AMPL. Models have been previously specified in these data formats and then solved by external tools interfaced via the web-based stochastic tool *NEOS Solver*[19].

SMPS

Birge *et al* introduced *SMPS*, an extension of the *Mathematical Programming System* (MPS) format[1], for standardising the format for inputing multi-stage stochastic programs. The data

[1]MPS is a file format for representing and persisting linear programming and mixed integer programming problems[16].

format has been designed for the ease of compactly describing those large-scale stochastic programs characterised by scenario-based recourse decisions. The is currently achieved via use of three text files[17].

- The `core` file specifies all the deterministic information of the problem in the MPS format.
- The `time` file serves to decompose the data in the `core` file into nodes corresponding to discretised stages.
- The `stochastics` file contains meta-information about random variables, which the solver can use to build a deterministic equivalent of the stochastic model.

AMPL

AMPL is *A Modelling Language for Mathematical Programming* developed at Bell Laboratories. It offers a formal vocabulary for linear and non-linear optimisation problems through a pseudo-symbolic algebraic and indexing notation. Optimisation problems are described via two files[18].

- The `model` file uses particular language constructs to declare variables for constant parameters. This is in addition to variables for recourse decisions, which allow for a minimised or maximised objective function and a set of model constraints to be expressed.
- The `data` file encapsulates the numerical values for constants and the costs for the decisions declared in the model file.

Integrated Environments for Decision-making Under Uncertainty

With the provision of an input format for initially specifying the model, all that remains is to convert the input data into an intermediate representation that an invoked solver can easily manipulate. There are many integrated environments that combine the modelling, solving and results analysis components as sub-systems. An example of such is the *Stochastic Programming Integrated Environment* (SPInE) available from OptiRisk Systems[20].

Discussion

SMPS is a good modelling tool but as pointed out by Gassmann and Schweitezer[22], this particular data format has some limitations. In particular, the `subroutine` construct to allow the user to specify distribution information has never been properly developed. In addition, the dependency structure makes the order of processing important[17], which not only complicates the parsing routine[21] but also restricts the modeller's choice as to how the model is described.

AMPL's symbolic notation offers the benefit of being intuitive for mathematically-inclined modellers and being consistent and formal enough for direct manipulation by a computer system. However, for modellers who only have system-specific knowledge, this notation is too verbose and complicated. Besides the learning curve of using such a language, AMPL has no notion of random parameters or variables, thereby making it impossible to model recourse problems. Consequently, this limitation has been addressed in SAMPL, which is the stochastic extension of the AMPL language co-developed by CARISMA and OptiRisk Systems[48].

The fundamental limitation of using or even extending current modelling languages like AMPL, SAMPL and SMPS is that they are biased towards scenario-based recourse problems, whereas our focus is on distribution-based recourse problems.

1.4 Overview of Report

Chapter 2 lays the foundation for this project by first reviewing fundamental concepts relating to optimisation and probability theory. Following this material, we go on to revise stochastic programming by formalising a definition of uncertainty and exploring the characteristics and types of stochastic programs. By re-introducing two particular recourse models, we can go further to investigate how such models can be solved in light of two-stage and multi-stage stochastic programs.

Chapter 3 re-presents important results from the research paper *Primal and Dual Linear Decision Rules in Stochastic and Robust Optimisation*[2]. We explain to the reader how we can reformulate intractable stochastic programming problems to compute conservative and progressive approximations. Additionally, we work through an example application of the linear decision rule approximation to demonstrate how the matrix components of the tractable conic programs are generated.

Chapter 4 discusses algebraic modelling languages with respect to a particular stochastic programming framework. We look at the language constructs required to model distribution-based decision problems, and we discuss an existing C++ implementation of this project. We then conclude this chapter, with a specification of the new input format, JADA. We aim aim to explain and justify our choice for the syntax and the semantics of JADA's algebraic modelling language.

Chapter 5 details our design and implementation of JADA. We discuss the development environment and justify our choices of the implementation language. As a high-level description, we explain the overall system architecture. We then decompose the system into the four main modules respectively responsible for parsing the input file, generating the matrix components of the linear program, generating the objective function and constraints of the conic programming instances to be solved, and finally rendering the optimal decision rules of the solved conservative and progressive linear programs.

Chapter 6 explains an extended implementation to that outlined in chapter 5 by considering notation for stochastic processes to facilitate highly compact descriptions.

Chapter 7 methodically describes a numerical evaluation of the final deliverable by considering a simple supply-demand stochastic program that we have previously manually solved, and a real-life complex model concerning the capacity expansion of public infrastructure. We present our results in terms of the generated linear programs and the interpreted optimal decision rules. We also discuss the gaps in optimality for the two decision-making problems.

Chapter 8 concludes this report by summarising our overall contributions, by presenting a qualitative evaluation of JADA, and lastly by discussing the directions for further development.

2
Background

The aim of this chapter is to introduce some theoretical material that we feel are important for understanding the nature and implementation of this project. We outline some introductory concepts to probability and measure theory (see section 2.1)[23] in order to understand how uncertainty is modelled for structuring stochastic programs. Section 2.2 encompasses a basic mathematical review of optimisation theory in order to appreciate the differences between deterministic and stochastic programming. We assume no prior knowledge of stochastic programming and thus we explain in detail what stochastic programming problems are, how they are formulated and also how they are solved. By exploring how solutions to stochastic programming problems are computed, we hope this will lay the foundation for chapter 3, where we re-introduce some key notions for applying linear decision rules for computational tractability[2].

2.1 Probability and Measure Theory

2.1.1 Sigma-algebra

Suppose ψ denotes a set and a set $\mathcal{F}$ representing subsets of ψ, then $\mathcal{F}$ is a *σ-algebra* of subsets of ψ if the following hold:

1. $\forall\, S_1, S_2 \in \mathcal{F}$,
    - $\mathcal{F}$ is closed under finite intersection, $S_1 \bigcap S_2 \in \mathcal{F}$.
    - $\mathcal{F}$ is closed under finite union, $S_1 \bigcup S_2 \in \mathcal{F}$.
    - $\mathcal{F}$ is closed under complementation, $S_1 \backslash S_2 \in \mathcal{F}$.
2. $\psi \in \mathcal{F}$.
3. $\forall\, S_i \in \mathcal{F}$, $\left(\bigcup_{i\in\mathbb{N}} S_i\right) \in \mathcal{F}$.

Sample Space

The set ψ that has a σ-algebra $\mathcal{F}$ is acknowledged as a *sample space* and is symbolically defined as the tuple $(\psi, \mathcal{F})$.

Probability Measure

A non-negative function $\mu : \mathcal{F} \longrightarrow \mathbb{R}$ is called a *measure* on $(\psi, \mathcal{F})$ if $\forall\ S_i \in \mathcal{F}$ and $i \in \mathbb{N}$ where $i \neq j$ and $S_i \bigcap S_j = \emptyset$:

$$\mu(\bigcup_{i\in\mathbb{N}} S_i) = \sum_{i\in\mathbb{N}} \mu(S_i) \tag{2.1.1.1}$$

A measure $\mathbb{P}$ is called a *probability measure* if $\mathbb{P}(\psi) = 1$, where:

1. $\mathbb{P}(\emptyset) = 0$.
2. $0 \leq \mathbb{P}(S) \leq 1$.
3. $\mathbb{P}(S_1 \bigcup S_2) = \mathbb{P}(S_1) + \mathbb{P}(S_2) - \mathbb{P}(S_1 \bigcap S_2)$.

Probability Space

A sample space that has a probability measure $\mathbb{P}$ is formally known as a *probability space*, and is represented by the tuple $(\psi, \mathcal{F}, \mathbb{P})$.

Borel Sigma-algebra

Let $\psi = \mathbb{R}$ and $\mathcal{F}$ semantically denote the collection of all intervals in $\mathbb{R}$. Then this collection must generate a σ-algebra of subsets of $\mathbb{R}$. This σ-algebra of subsets of $\mathbb{R}$ is the *Borel σ-algebra* in $\mathbb{R}$, which is denoted as $\mathcal{B}(\mathbb{R})$. This is the smallest σ-algebra containing $\mathcal{F}$[24].

1. Any subset A of $\mathbb{R}$ such that $A \in \mathcal{B}(\mathbb{R})$ is called a *Borel set* in $\mathbb{R}$.
2. A function $f : \psi \longrightarrow \mathbb{R}$, which has the inverse image $f^{-1}(A) = \{\omega \in \psi : f(\omega) \in A\}$, is called a *Borel measure*.

Support of Probability Measure

The support of probability measure $\mathbb{P}$ is the smallest closed set $\Xi \subset \mathbb{R}$ where $\mathbb{P}(\ \Xi\ ) = 1$. Thus, the probability measure $\mathbb{P}$ defined over the measurable space $(\Xi, \mathcal{B}(\mathbb{R}))$ yields the probability space $(\Xi, \mathcal{B}(\mathbb{R}), \mathbb{P})$. Thus, we can consider $\xi \in \Xi$ as a particular realisation of a random data vector.

Essential Supremum

Assuming the measurable space $(\mathbb{R}^k, \mathcal{B}(\mathbb{R}^k), \mathbb{P})$ and a function $f : \psi \longrightarrow \mathbb{R}$, an element $\alpha \in \mathbb{R}$ is called an *essential supremum* (ess-sup) for f if $\forall x \in X$, we have $f(x) \leq \alpha$.

2.1.2 Random Variables

A random variable X is a Borel measurable function $X : (\psi, \mathcal{F}) \longrightarrow (\mathbb{R}, \mathcal{B}(\mathbb{R}))$. The probability measure $\mathbb{P}_X$ for this random variable is:

$$\mathbb{P}_X(Y) = \mathbb{P}_X(\omega : X(\omega) \in Y), \quad Y \in \mathcal{B}(\mathbb{R}). \tag{2.1.2.1}$$

Distribution Functions

The *cumulative distribution function* (CDF) of a random variable X is defined as:

$$F_X(x) = \mathbb{P}_X(\omega : X(\omega) \leq x), \quad x \in \mathbb{R} \tag{2.1.2.2}$$

where $F_X : \mathbb{R} \longrightarrow [0, 1]$.

A random variable X that is continuously distributed on the set $\mathbb{R}$ is called a continuous random variable. X has a Borel measure f_X known as the *probability density function* (PDF), for $x \in \mathbb{R}$:

$$F_X(x) = \int_{-\infty}^{x} f_X(t)\, dt \tag{2.1.2.3}$$

where the distribution $F_X(X)$ is such that $\lim_{x \longrightarrow -\infty} F_X(x) = 0$ and $\lim_{x \longrightarrow +\infty} F_X(x) = 1$.

Additionally, the probability of X belonging to the interval $[a, b]$, where $a, b \in \mathbb{R}$, is defined as:

$$\mathbb{P}_X(a \leq X \leq b) = \int_a^b f_X(x)\, dx. \tag{2.1.2.4}$$

Independence

Random variables $Y_1, Y_2, \cdots, Y_n$, which are equipped with the probability space $(\psi, \mathcal{F}, \mathbb{P}_{Y_i})$ are independently distributed if the probability of random variable $X = \bigcap_{i=1}^{n} Y_i$ is:

$$\mathbb{P}_X(X) = \mathbb{P}_X\left(\bigcap_{i=1}^{n} Y_i\right) = \prod_{i=1}^{n} \mathbb{P}_{Y_i}(Y_i), \tag{2.1.2.5}$$

where $Y_i \in B_i$ and Borel set $B_i \in \mathcal{B}(\mathbb{R})$.

The distribution function of the random variable $X = \bigcap_{i=1}^{n} Y_i$ is:

$$F_X(X) = F_X\left(\bigcap_{i=1}^{n} Y_i\right) = \prod_{i=1}^{n} F_{Y_i}(Y_i) \tag{2.1.2.6}$$

where independent random variables $Y_1 \in B_1, Y_2 \in B_2, \cdots, Y_n \in B_n$ are equipped with CDFs F_{Y_1}, F_{Y_2}, $\cdots$, F_{Y_n} respectively.

Expectation and Variance

If the random variable X is equipped with the probability space $(\psi, \mathcal{F}, \mathbb{P}_X)$, then the expected value of X, denoted by $\mathbb{E}_X$, is computed as:

$$\mathbb{E}_X[X] = \int_{\psi} X\, d\,\mathbb{P}_X \tag{2.1.2.7}$$

and its variance, denoted by $\mathbb{V}ar_X$, is defined as:

$$\mathbb{V}ar_X[X] = \mathbb{E}_X\left[(X - \mathbb{E}_X[X])^2\right]. \tag{2.1.2.8}$$

If $Y_1, Y_2, \cdots, Y_n$ are independently distributed random variables with respective expected values $\mathbb{E}[Y_1]$, $\mathbb{E}[Y_2]$, $\cdots$, $\mathbb{E}[Y_n]$, then the expectation of random variable $X = \bigcap_{i=1}^{n} Y_i$ is:

$$\mathbb{E}_X(X) = \mathbb{E}_X\left[\bigcap_{i=1}^{n} Y_i\right] = \prod_{i=1}^{n} \mathbb{E}_{Y_i}[Y_i]. \tag{2.1.2.9}$$

If X is a $m \times n$ matrix, then the expectation of X is computed as:

$$\mathbb{E}_X[X] = \mathbb{E}_X\left[\begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix}\right] = \begin{pmatrix} \mathbb{E}_X[x_{1,1}] & \mathbb{E}_X[x_{1,2}] & \cdots & \mathbb{E}_X[x_{1,n}] \\ \mathbb{E}_X[x_{2,1}] & \mathbb{E}_X[x_{2,2}] & \cdots & \mathbb{E}_X[x_{2,n}] \\ \vdots & \vdots & \ddots & \vdots \\ \mathbb{E}_X[x_{m,1}] & \mathbb{E}_X[x_{m,2}] & \cdots & \mathbb{E}_X[x_{m,n}] \end{pmatrix}. \tag{2.1.2.10}$$

Covariance

The covariance quantifies how much two random variables X and Y change together, and is denoted by $\mathbb{C}ov[X, Y]$. The covariance is given as:

$$\mathbb{C}ov[X, Y] = \mathbb{E}_{X,Y}\left[(X - \mathbb{E}_X[X])^2 (Y - \mathbb{E}_Y[Y])^2\right] \tag{2.1.2.11}$$

which can further be simplified to:

$$\begin{aligned} \mathbb{C}ov[X, Y] &= \mathbb{E}_{X,Y}\left[XY - X\mathbb{E}_Y[Y] - Y\mathbb{E}_X[X] + \mathbb{E}_X[X]\mathbb{E}_Y[Y]\right] \\ &= \mathbb{E}_{X,Y}[XY] - \mathbb{E}_{X,Y}[X\mathbb{E}_Y[Y]] - \mathbb{E}_{X,Y}[Y\mathbb{E}_X[X]] + \mathbb{E}_{X,Y}[\mathbb{E}_X[X]\mathbb{E}_Y[Y]] \\ &= \mathbb{E}_{X,Y}[XY] - \mathbb{E}_X[X]\mathbb{E}_Y[Y] \end{aligned} \tag{2.1.2.12}$$

Correlation

The *Pearson product-moment correlation co-efficient* $\rho_{X,Y}$ of two random variables X and Y quantifies the extent of their dependence, and is defined as:

$$\rho_{X,Y} = \frac{\mathbb{C}ov[X, Y]}{\sqrt{\mathbb{V}ar_X(X)}\sqrt{\mathbb{V}ar_Y(Y)}} \in [-1, +1]. \tag{2.1.2.13}$$

If X and Y are independent random variables, then they are said to be *uncorrelated* since $\rho_{X,Y} = 0$ as $\mathbb{E}_{X,Y}[XY] = \mathbb{E}_X[X]\mathbb{E}_Y[Y]$ implies $\mathbb{C}ov[X, Y] = 0$. If $\rho_{X,Y} = \pm 1$ then X and Y are totally correlated.

k^{th} Central Moment

The kth central moment of a random variable X, denoted by μ_k, is quantified as

$$\mu_k = \mathbb{E}_X\left[(X - \mathbb{E}_X[X])^k\right], \tag{2.1.2.14}$$

where $\mu_0 = 1$ and $\mu_1 = 0$. Intuitively, we interpret μ_2 as the variance eq. (2.1.2.8).

Conditional Probability and Expectation

Assuming a probability space $(\psi, \mathcal{F}, \mathbb{P})$ and sets $A_1, A_2 \in \mathcal{F}$ where $\mathbb{P}(A_2) > 0$, the conditional probability of A_1 given A_2 is:

$$\mathbb{P}(A_1|A_2) = \frac{\mathbb{P}(A_1 \cap A_2)}{\mathbb{P}(A_2)}. \tag{2.1.2.15}$$

For a random variable X defined over the probability space $(\psi, \mathcal{F}, \mathbb{P})$, the conditional expectation of X given $\mathcal{B}(\mathbb{R})$ is a Borel measurable function $\mathbb{E}_X[X|\mathcal{B}(\mathbb{R})] : \psi \longrightarrow \mathbb{R}$ such that $\forall B \in \mathcal{B}(\mathbb{R})$:

$$\int_B \mathbb{E}_X[X|\mathcal{B}(\mathbb{R})]\, d\,\mathbb{P} = \int_B X\, d\,\mathbb{P} \tag{2.1.2.16}$$

2.1.3 Random Vectors

Supposing $X_1, \cdots, X_n$ are random variables with the same probability space $(\mathbb{R}^k, \mathcal{B}(\mathbb{R}^k), \mathbb{P})$, then a multivariate random variable $X = (X_1, \cdots, X_n)$ is termed a *random vector*.

Joint Distribution Functions

The random vector $(X_1, \cdots, X_n)$ generates a probability measure on the space $\mathbb{R}^n$ with respect to the Borel σ-algebra. This Borel measurable function $f_{X_1,\cdots,X_n}$ is known as its *joint probability density distribution*. The joint PDF is:

$$\begin{aligned} f_{X_1,\cdots,X_n}(x_1,\cdots,x_n) &= f_{X_n \mid x_1,\cdots,x_{n-1}}(x_n \mid x_1,\cdots,x_{n-1}) \cdot f_{X_1,\cdots,X_{n-1}}(x_1,\cdots,x_{n-1}) \\ &= \prod_{i=1}^{n} f_{X_i \mid X_1,\cdots,X_{n-1}}(x_i \mid x_1,\cdots,x_{n-1}) \end{aligned} \tag{2.1.3.1a}$$

where

$$\begin{aligned} f_{X_i \mid X_1,\cdots,X_{i-1}}(x_i \mid x_1,\cdots,x_{i-1}) &= \frac{f_{X_1,\cdots,X_i}(x_1,\cdots,x_i)}{\int f_{X_1,\cdots,X_i}(x_1,\cdots,x_{i-1},t_i)\, dt_i} \\ &= \frac{\int\cdots\int f_{X_1,\cdots,X_n}(x_1,\cdots,x_i,t_{i+1},\cdots,t_n)\, dt_{i+1}\cdots dt_n}{\int\cdots\int\int f_{X_1,\cdots,X_n}(x_1,\cdots,x_{i-1},t_i,\cdots,t_n)\, dt_i\, dt_{i+1}\cdots dt_n} \end{aligned} \tag{2.1.3.1b}$$

and

$$f_{X_1,\cdots,X_i}(x_1,\cdots,x_i) = \int\cdots\int f_{X_1,\cdots,X_n}(x_1,\cdots,x_i,x_{i+1},\cdots,x_n)\, dx_{i+1}\cdots dx_n. \tag{2.1.3.1c}$$

We define the joint CDF of random vector X by:

$$F_{X_1,\cdots,X_n}(x_1,\cdots,x_n) = \int_{-\infty}^{x_1}\cdots\int_{-\infty}^{x_n} f_{X_1,\cdots,X_n}(t_1,\cdots,t_n)\, d\,t_1\cdots d\,t_n. \tag{2.1.3.2}$$

Conditional Distribution

We note that the conditional distribution function of random vector X is:

$$F_{X_i \mid X_1,\cdots,X_{i-1}}(x_i \mid x_1,\cdots,x_{i-1}) = \frac{\int\limits_{-\infty}^{x_i} f_{X_1,\cdots,X_i}(x_1,\cdots,x_{i-1},t_i)\,d\,t_i}{\int\limits_{-\infty}^{+\infty} f_{X_1,\cdots,X_i}(x_1,\cdots,x_{i-1},t_i)\,d\,t_i}$$

$$= \frac{\int\limits_{-\infty}^{+\infty}\cdots\int\limits_{-\infty}^{+\infty}\int\limits_{-\infty}^{x_i} f_{X_1,\cdots,X_i}(x_1,\cdots,x_{i-1},t_i,\cdots,t_n)\,d\,t_i\,t_i\,\cdots\,d\,t_n}{\int\limits_{-\infty}^{+\infty} f_{X_1,\cdots,X_i}(x_1,\cdots,x_{i-1},t_i,\cdots,t_n)\,d\,t_i\,t_i\,\cdots\,d\,t_n} \tag{2.1.3.3}$$

Marginal Distribution

The probability distributions of each of the random variables X_i from the random vector X is called the *marginal distribution.* Assuming a distribution function $f(x_1,\cdots,x_n)$, the marginal PDF is:

$$f_{X_i}(x_i) = \int\cdots\int\int\cdots\int f_{X_1,\cdots,X_n}(x_1,\cdots,x_{i-1},x_i,x_{i+1},\cdots,x_n)\,dx_1\cdots dx_{i-1}\,dx_{i+1}\cdots dx_n. \tag{2.1.3.4}$$

Covariance

Given a random vector $X = (X_1, X_2, \cdots, X_n)^\mathsf{T}$, the covariance matrix $\Omega \in \mathbb{R}^{n\times n}$ for X is a matrix of covariances between the elements of X, where $\Omega_{X,i,j} = \mathbb{C}ov[X_i, X_j] = \mathbb{E}_{X_i,X_j}[X_iX_j] - \mathbb{E}_{X_i}[X_i]\mathbb{E}_{X_j}[X_j]$:

$$\Omega_X = \begin{pmatrix} \mathbb{C}ov[X_1,X_1] & \mathbb{C}ov[X_1,X_2] & \cdots & \mathbb{C}ov[X_1,X_n] \\ \mathbb{C}ov[X_2,X_1] & \mathbb{C}ov[X_2,X_2] & \cdots & \mathbb{C}ov[X_2,X_n] \\ \vdots & \vdots & \ddots & \vdots \\ \mathbb{C}ov[X_n,X_1] & \mathbb{C}ov[X_n,X_2] & \cdots & \mathbb{C}ov[X_n,X_n] \end{pmatrix}. \tag{2.1.3.5}$$

2.1.4 Continuous Uniform Distribution

The continuous uniform distribution $U(a,b)$ is a family of probability distributions such that for each member of the family, all intervals of identical length on the distribution's support are equally probable. The support is parameterised by $a, b \in \mathbb{R}$, which correspond to the minimum and maximum values respectively.

Distribution Functions

For a random variable X following a continuous uniform distribution $U(a, b)$, its PDF is:

$$f_X(x) = \begin{cases} \dfrac{1}{b-a}, & a \leq x \leq b \\ \\ 0, & \text{otherwise} \end{cases} \tag{2.1.4.1}$$

and its CDF is:

$$F_X(x) = \begin{cases} 0, & x < a \\ \\ \dfrac{x-a}{b-a}, & a \leq x \leq b \\ \\ 1, & x > b \end{cases} \tag{2.1.4.2}$$

Mean and Variance

By eqs. (2.1.2.7), (2.1.2.8) and (2.1.4.1), we state the expectation and variance of a random variable $X \sim U(a, b)$ as:

$$\mathbb{E}_X[X] = \frac{a+b}{2} \quad \text{and} \quad \mathbb{V}ar_X[X] = \frac{(b-a)^2}{12} \tag{2.1.4.3}$$

respectively.

2.1.5 Sample Mean and Variance

Assuming a random sample $x_1, x_2, \cdots, x_N$ from an n-dimensional random variable X, the sample mean is given by:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{2.1.5.1}$$

and the sample variance is computed as:

$$\sigma^2 = \frac{1}{N-1} \sum_{i=1}^{N} (x_i - \bar{x})^2. \tag{2.1.5.2}$$

2.1.6 Modelling Uncertainty

Uncertainty can be modelled by a probability space $(\mathbb{R}^k, \mathcal{B}(\mathbb{R}^k), \mathbb{P})$, where the elements of the sample space $\mathbb{R}^k$ are represented by ξ[2]. Elements of uncertainty in a stochastic programming problem can be modelled by random vectors.

2.2 Basic Concepts of Optimisation Theory

In the operations research community, *optimisation* is an umbrella-term for selecting the best decision policy or strategy, from a group of possible alternatives, in order to maximise or minimise a real-valued affine function f.

2.2.1 Deterministic Linear Programming (LP)

The general formulation of a linear program consists of a linear function to be minimised, a set of problem constraints and a specification of positive variables:

$$\begin{aligned} \underset{x \in \mathbb{R}}{\mathit{minimise}} \quad & c^{\mathsf{T}} x \\ \mathit{subject\ to} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{2.2.1.1}$$

where $A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$, $x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$, $c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}$ and $b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$.

- Dimensions m and n are the number of constraints and decision variables respectively.
- $x \in \mathbb{R}^n$ is a vector of unknown decision variables.
- $A \in \mathbb{R}^{m \times n}$ is a matrix of known co-efficients representing the left-hand-side of the constraints.
- $b \in \mathbb{R}^m$ is a vector of known co-efficients representing the right-hand-side of the constraints.
- $c \in \mathbb{R}^n$ is a vector of known co-efficients representing the costs of the unknown variables in the objective.

The expression to be minimised or maximised, $c^{\mathsf{T}} x$, is called the objective function, and the system of equations $Ax \leq b$ and $x \geq 0$ are the constraints which give a bounded convex polytope (the feasible region) over which the objective function is to be optimised.

Supposing $\alpha \leq \beta$, we use the inequality operator to symbolise *componentwise inequality* if α and β are vectors and matrix inequality when α and β are matrices

2.2.2 Duality

The linear programming problem given in eq. (2.2.1.1) is known as the *primal problem* (P-LP), which can also be converted into a *dual problem* (D-LP). It is formulated as:

$$\begin{aligned} \underset{y \in \mathbb{R}}{\mathit{maximise}} \quad & b^{\mathsf{T}} y \\ \mathit{subject\ to} \quad & A^{\mathsf{T}} y \leq c \\ & y \text{ unrestricted} \end{aligned} \tag{2.2.2.1}$$

where $y \in \mathbb{R}^m = [y_1, y_2, \cdots, y_m]^\mathsf{T}$ is a vector of unknown dual decision variables, and A, b and c assume the same definitions as eq. (2.2.1.1).

Duality Theory

There are two fundamental notions that underpin duality theory.

1. The D-LP provides a lower bound to the computed optimal value of the primal linear program, $c^\mathsf{T}x \geq b^\mathsf{T}y$.

2. If $c^\mathsf{T}x' = c^\mathsf{T}y'$, where x' and y' are feasible solutions to the primal linear program eq. (2.2.1.1) and the dual linear program eq. (2.2.2.1) respectively, then x' and y' are also the optimal solutions to their respective linear programs.

2.2.3 Standardisation

We must convert all linear programs into its standard form, which may require transforming maximisation objectives, negative co-efficients on the right-hand-side of the constraints, inequality constraints and unrestricted variables.

Maximisation Objective

Maximisation objective functions can be converted into an equivalent minimisation objective function type by noting $\underset{x\in\mathbb{R}}{maximise}\ f(x) \equiv \underset{x\in\mathbb{R}}{minimise}\ -f(x)$.

Negative Co-efficients on the Constraints' RHS

For an LP to be standardised, the constraints' right-hand-side vector b must be non-negative. When $b \leq 0$, we can multiply both sides of the constraint by -1, which will consequently reverse the direction of the inequality.

Inequality Constraints

To bring any LP to standard form, we replace all inequality constraints with equalities by the introduction of slack or surplus variables, $s \in \mathbb{R}^m$.

- If a_i represents the i^{th} row of the matrix A, then we transform the inequality $a_ix \leq b_i$ to $a_ix + s_i = b_i$ by adding a slack variable s_i for the i^{th} row.

- Similarly, we transform the inequality $a_ix \geq b_i$ to $a_ix - s_i = b_i$ by subtracting a surplus variable s_i for the i^{th} row.

Unrestricted Variables

The standard form for an LP imposes that all variables must be positive. Supposing the constraint $x' \geq 0$ is absent from the LP, then the variable x' can take positive or negative values, which is the same as saying x' is *unrestricted*. Thus, we substitute x' with $x' = u' - v'$, where $u, v \geq 0$.

2.2.4 Solutions

Linear programming problems can be solved using the SIMPLEX algorithm. The method traces the perimeter of the convex polytope (feasible region), given by the system of linear constraints, to search for the optimal solution.

- A *solution* to a linear program of the form eq. (2.2.1.1) is a vector x that satisfies the system of linear constraints $Ax = b$.
- A *feasible solution* is the solution x where $x \geq 0$.
- An *optimal solution* is the feasible solution $x*$ such that for all feasible solutions x', we have $c^\mathsf{T} x* \leq c^\mathsf{T} x'$.

2.2.5 Semidefinite Programming (SDP)

Semidefinite programming is a general form of linear programming and is used in the context of linear matrix inequalities (LMI). As convex optimisation problems, SDPs aim to minimise a linear function subject to an affine combination of positive semidefinite symmetric matrices.

An SDP is solved using the *interior point method* which, in contrast to the SIMPLEX technique, attempts to find an optimal solution by tracing the interior of the feasible region[30].

We note the following definitions:

- For a matrix $A \in \mathbb{R}^{m \times m}$ to be *positive semidefinite*, all of its eigenvalues must be non-negative or this can be succinctly put as $\forall z \in \mathbb{R}^m, z^\mathsf{T} Az \geq 0$. We denote matrix A is positive semidefinite by $A \succeq 0$.
- The *trace* of matrix $A \in \mathbb{R}^{m \times m}$ is defined to be $\mathbb{T}race(A) = a_{1,1} + a_{2,2} + \cdots + a_{m,m} = \sum_{i=1}^{m} a_{i,i}$.
- We define $\mathbb{S}^k$ to be the space $\mathbb{R}^{k \times k}$ of all symmetric matrices.

Primal Semidefinite Program (P-SDP)

We can formulate the P-SDP as:

$$\begin{aligned} &\underset{x \in \mathbb{R}}{minimise} \;\; c^\mathsf{T} x \\ &subject\ to \;\; F_0 + \sum_{i=1}^{m} x_i F_i \succeq 0 \end{aligned} \tag{2.2.5.1}$$

where $F_i \in \mathbb{S}^k$ for $i = 0, \cdots, m$. Vectors c and x assume the same dimensions and semantics as before. We acknowledge the constraint $F_0 + \sum_{i=1}^{m} x_i F_i \succeq 0$ as an LMI.

Dual Semidefinite Program (D-SDP)

Similarly, we can define D-SDP as:

$$\begin{aligned} \underset{Y \in \mathbb{R}^{m \times m}}{maximise} \quad & -\mathbb{T}race(F_0 Y) \\ subject\ to \quad & \mathbb{T}race(F_i Y) = c_i, \quad \forall i \in \{1, \cdots, n\} \\ & Y \succeq 0 \end{aligned} \tag{2.2.5.2}$$

where matrix $Y = Y^{\mathsf{T}} \in \mathbb{R}^{m \times m}$ is the dual variable and the objective function is a linear combination of Y.

2.3 Stochastic Programming

Deterministic linear programs of the form eq. (2.2.1.1) are intended for modelling optimisation problems where all of the underlying data elements are known to the decision-maker. However, if we are modelling real-life decision-making problems, it is perhaps naive to assume all of the problem data is indeed known, and instead consider that many data elements may be subject to some degree of uncertainty. Thus, we observe the need to utilise the stochastic programming framework. which is the state-of-the-art approach to optimising decision problems under uncertainty.

For stochastic programming, the modeller is required to apply a variety of statistical techniques and procedures from the operations research toolbox.

2.3.1 Characteristics of the Stochastic Programming Framework

Below we qualify the prominent attributes of stochastic programming.

Recourse Models

The term *recourse* refers to the opportunity to re-strategise or adapt a solution in response to information from an observation[26].

In recourse models, some decisions can only be made after uncertainty has been revealed. Thus, before information applicable to the uncertainties is disclosed, some of the decisions must be anchored and some decisions must be postponed until after some random experiment. There are two distinct cases of the recourse model, namely fixed recourse and random recourse.

Decision Stages

The set of decisions made can be generally categorised into two groups[27].

- *First-stage decisions* are those decisions which occur in the first period of the model, known as the first-stage, and consequently have to be made before a random experiment takes place.

- *Second-stage decisions* are those decisions that are made after the aforementioned random experiment has been carried out. The period for when these decisions are taken is known as the second stage.

Non-anticipativity

We enforce a rule on the recourse decisions called *non-anticipativity.* This effectively means that although recourse decisions can respond to past observations, they are not allowed to be influenced by future observations that have yet to occur.

2.3.2 Components of a Stochastic Program (SP)

In this section we present the basic components of a stochastic programming problem[28].

An Underlying Process

Fundamentally, a stochastic program can be considered as a finite process of interleaving decisions and observations in stages.

The first-stage of the process revolves around the selection of an initial decision x_1, which is then succeeded by $N-1$ recourse stages. Each of these recourse stages involve observations of random variables after a random experiment has occurred, from which a choice of a new decision is made in reaction to the observation.

At termination the process produces an outcome modelled as the tuple $(x, \xi) \in \mathbb{R}^n \times \psi$. The vector $x = [x_1, x_2, \cdots, x_N] \in \mathbb{R}^n$ for $n = \sum_{1=1}^{N} n_i$ is the trace of the decision-maker's pattern of action, and $\xi = [\xi_2, \cdots, \xi_N] \in \psi = \psi_2 \times \cdots \times \psi_N$ is the historical record of observations made.

Cost of Outcomes

The cost attributed to the eventual outcome of this process can be described by the affine function c on the domain $\mathbb{R}^n \times \psi$ such that $c(x, \xi) = c(x_1, x_2, \cdots, x_N, \xi_2, \cdots, \xi_N)$.

$x_1 \in \mathbb{R}^{n_1}$	first-stage decision
	random experiment
$\xi_2 \in \psi_2$	observations
$x_2 \in \mathbb{R}^{n_2}$	second-stage decision
$\vdots$	
	random experiment
$\xi_N \in \psi_N$	observations
$x_N \in \mathbb{R}^{n_N}$	final-stage decision

Figure 2.1: Underlying process of a stochastic program.

Probability Structure

The random vector $\xi = [\xi_1, \cdots, \xi_N]$ has a general probability distribution which is given with the space ψ. The random variables or components ξ_i of ξ may or may not be independently distributed of each other.

Evolution of Information

The elements of uncertainty have a *prior* and *posterior* mode. In the prior mode, only probabilistic information about the random variable ξ_i is available, but in the posterior mode ξ_i becomes static data. We refer to an *observation* as the transition between this prior and posterior mode.

In the first-stage, when the decision-maker is required to select an initial decision x_1, no information is available about the uncertain elements in the data as no observation can yet be made. Without loss of generality, we can model ξ_1 as a degenerate dummy outcome such that $\xi_1 = 1$. On the contrary, for recourse decision x_i taken in the second stage and beyond, some of the uncertainty has been revealed through observations of $[\xi_2, \cdots, \xi_i]$.

The random vector $\xi = [\xi_2, \cdots, \xi_N]$ is now partitioned into $[\xi_2, \cdots, \xi_i]$ and $[\xi_{i+1}, \cdots, \xi_N]$, which respectively represent the *current information* and *residual uncertainty*. Additionally, the probability space has now been truncated to $\psi_{i+1} \times \cdots \times \psi_N$, and the probability distribution for $[\xi_{i+1}, \cdots, \xi_N]$ in this reduced space is its conditional probability distribution given $[\xi_2, \cdots, \xi_i]$.

Recourse Functions

It is mandatory that the recourse decision in stage i is modelled as a function, as opposed to a constant vector. This allows us to capture the ability of the decision x_i to adapt itself to the current information. Thus, the decision-maker is not simply selecting a vector in $\mathbb{R}^{n_i}$, but is instead selecting a recourse function $x_i : [\xi_2 \cdots, \xi_N] \longrightarrow x_i(\xi_2, \cdots, \xi_N) \in \mathbb{R}^{n_i}$ defined over the space $\psi_2 \times \cdots \times \psi_i$ in order to state in advance how the decision-maker intends to respond to all outcomes of the first i observations.

We define a *square integrable function* to be a function f such that the integral $\int_{-\infty}^{+\infty} |f(x)|^2 \, dx$ is finite. We represent the space of all Borel measurable, square-integrable functions from $\mathbb{R}^k$ to $\mathbb{R}^n$ by $\mathcal{L}^2_{k,n} = \mathcal{L}^2(\Gamma, \mathbb{R}^n)$, where Γ denotes the probability space $(\mathbb{R}^k, \mathcal{B}(\mathbb{R}^k), \mathbb{P})$. Thus, we can alternatively model the recourse function as $x_i \in \mathcal{L}^2_{k_i,n_i}$.

Policies

A *policy* refers to the selection of the first-stage decision x_1 and the recourse decision $x_i : \psi \longrightarrow \mathbb{R}^n$ for $i = 2, \cdots, N$ such that $x(\xi) = [x_1, x_2(\xi_2), \cdots, x_N(\xi_2, \cdots, \xi_N)]$. This set of non-anticipative functions x is called the *policy space*.

2.3.3 One-stage Stochastic Programs (SP)

In this section we revise one-stage stochastic programs, which are instances of optimisation problems under uncertainty with a single time period or stage. Stochastic programs of this type involve an initial observation of a random variable ξ from the sample space ψ. The decision-maker then chooses a recourse decision $x(\xi) \in \mathbb{R}^n$, with an associated cost of $c(\xi)^\mathsf{T} x(\xi)$, satisfying the constraint system $A(\xi)x(\xi) \leq b(\xi)$.

The stochastic program models an objective to minimise the expected cost $\mathbb{E}[c(\xi)^\mathsf{T} x(\xi)]$ by selecting a recourse function $x \in \mathcal{L}^2_{k,n}$. We formulate the corresponding stochastic program as:

$$\begin{aligned} &\underset{x \in \mathcal{L}^2_{k,n}}{\textit{minimise}} \;\; \mathbb{E}[c(\xi)^\mathsf{T} x(\xi)] \\ &\textit{subject to} \;\; A(\xi)x(\xi) \leq b(\xi), \quad \mathbb{P}-a.s. \end{aligned} \tag{2.3.3.1}$$

Standard Form

We bring the primal problem eq. (2.3.3.1) into standard form by augmenting the stochastic program with slack variables $s \in \mathcal{L}^2_{k,m}$ to eliminate inequality constraints:

$$\begin{aligned} &\textit{minimise} \;\; \mathbb{E}[c(\xi)^\mathsf{T} x(\xi)] \\ &\textit{subject to} \;\; x \in \mathcal{L}^2_{k,n},\; s \in \mathcal{L}^2_{k,m} \\ &\qquad\qquad \left.\begin{aligned} &A(\xi)x(\xi) + s(\xi) = b(\xi) \\ &s(\xi) \geq 0 \end{aligned}\right\} \mathbb{P}-a.s. \end{aligned} \tag{2.3.3.2}$$

and it's dual form is[1]:

$$\begin{aligned} &\textit{minimise} \;\; \mathbb{E}[c(\xi)^\mathsf{T} x(\xi)] \\ &\textit{subject to} \;\; x \in \mathcal{L}^2_{k,n},\; s \in \mathcal{L}^2_{k,m} \\ &\qquad\qquad \left.\begin{aligned} &\mathbb{E}[(A(\xi)x(\xi) + s(\xi) - b(\xi))\xi^\mathsf{T}] = 0 \\ &s(\xi) \geq 0 \end{aligned}\right\} \mathbb{P}-a.s. \end{aligned} \tag{2.3.3.3}$$

Well-definition

For well-definition of eq. (2.3.3.2), we assume vectors $c(\xi)$ and $b(\xi)$ are linear combinations of the random elements ξ. Therefore, we can assume without proof that $\exists C \in \mathbb{R}^{n\times k}$ such that $c(\xi) = C\xi$ and $\exists B \in \mathbb{R}^{m\times k}$ such that $b(\xi) = B\xi$.

Fixed Recourse

A *fixed recourse* problem assumes that the constraints matrix $A(\xi)$ is not subject to uncertainty. To specify that $A(\xi)$ does not depend on ξ, we indicate the equivalence $A(\xi) \equiv A \in \mathbb{R}^{m\times n}$.

The support of the probability measure $\mathbb{P}$, which we assume to span the whole of the sample space ψ, is given by a bounded non-empty set that is defined as:

$$\Xi = \{\; \xi \in \mathbb{R}^k : W\xi \geq h \;\} \tag{2.3.3.4a}$$

given that

$$W = [e_1, -e_1, \hat{W}]^\mathsf{T} \in \mathbb{R}^{l\times k} \text{ and } h = [1, -1, \underbrace{0, \cdots, 0}_{l-2}] \in \mathbb{R}^l \tag{2.3.3.4b}$$

[1] We refer the reader to the research paper[2] for its derivation.

where sub-matrix $\hat{W} \in \mathbb{R}^{(l-2)\times k}$ and basis vector $e_1 = [1, \underbrace{0, \cdots, 0}_{l-1}] \in \mathbb{R}^k$. The consequence of this definition is that for all $\xi \in \Xi$, we have $\xi_1 = 1$.

Random Recourse

For one-stage stochastic programs with random recourse, we assume that the constraints matrix $A(\xi)$ is indeed parameterised by uncertainty. We let $\xi^\mathsf{T} A_\mu$ represent the μ^{th} row of $A(\xi)$ where matrix $A_\mu \in \mathbb{R}^{k\times n}$ for $\mu = 1, \cdots, m$. We also define the μ^{th} row of matrix B as b_μ^T.

The polyhedral support for probability measure $\mathbb{P}$ is now described as:

$$\Xi = \{\ \xi \in \mathbb{R}^k : e_1^\mathsf{T}\xi = 1,\ \xi^\mathsf{T} W_\ell \xi \geq 0,\ \ell = 1, \cdots, l\ \} \tag{2.3.3.5}$$

where matrices W_ℓ are from $\mathbb{S}^k$, the space $\mathbb{R}^{k\times k}$ of all symmetric matrices.

2.3.4 Multi-stage Stochastic Programs (MSP)

So far, we have considered one-stage stochastic problems with recourse, in which the decision maker observes a random variable from the sample space, and then chooses a recourse decision x. In reality, most practical optimisation problems are actually sequential decision processes. In this section, we review stochastic recourse problems of this kind called *multi-stage stochastic programs.*

Temporal Structure

To capture the fact that the decision-maker now chooses multiple decisions that adapt to observations that evolve over time, we introduce a temporal structure through the indices $t \in T = \{\ 1, \cdots, T\ \}$ to denote the stages of the model. It is important to note that although the values $t \in T$ are strongly related to the temporal structure, they may not correspond exactly to the time periods. This is the case when time periods, at which no observations can be made, are aggregated with preceding periods to form one stage.

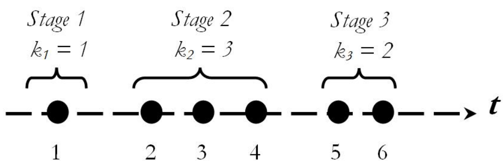


Figure 2.2: An example of three-stage aggregation.

Specifically, we denote the uncertain elements as $\xi = [\xi_1, \cdots, \xi_T]$, where sequential observations of the random sub-vectors $\xi_t \in \mathbb{R}^{k_t}$ are indexed by time points $t \in T$. The dimension k_t indicates the size of the current information for stage t. We further assume $k_1 = 1$ to impose that $\forall \xi \in \Xi, \xi_1 = 1$. The historical record of observations made up to the time point t is representable as $\xi^t = [\xi_1, \cdots, \xi^t]$ such that $k^t = \sum_{s=1}^t k_s$. For consistency, we stipulate that $\xi^T = \xi$ and $k^T = k$.

Temporal Operators

For $t \in T$, we define truncation operators P_t:

$$P_t : \mathbb{R}^k \longrightarrow \mathbb{R}^{k^t},\ \xi \longmapsto \xi^{\,t} \tag{2.3.4.1a}$$

Informally, we can think of $P_t \in \mathbb{R}^{k^t \times k}$ as the following matrix:

$$P_t = \left.\overbrace{\underbrace{\begin{bmatrix} 1_1 & 0 & \cdots & 0 \\ 0 & 1_{k_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1_{k_t} \end{bmatrix}}_{k^t = \sum_{s=1}^{t} k_s} \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}}^{k \,=\, k^T = \sum_{s=1}^{T} k_s} \right\} k^t = \sum_{s=1}^{t} k_s \tag{2.3.4.1b}$$

Uncertainty Model

We again assume that uncertainty is modelled by the probability space $(\mathbb{R}^k, \mathcal{B}(\mathbb{R}^k), \mathbb{P})$. We focus on fixed recourse programs and thus define the support for probability measure $\mathbb{P}$ to be identical to eq. (2.3.3.4).

General Formulation

A multi-stage stochastic program involves choosing at time t a decision $x_t(\xi^{\,t}) \in \mathbb{R}^{n_t}$ given the current information $\xi^{\,t}$ and residual uncertainty $\{\ \xi_s \mid s \geq t\ \}$. Thus, the objective is to minimise a linear expected cost function by selecting a series of policies $x_t \in \mathcal{L}^2_{k^t,n_t}$, using only the available observations $\xi^{\,t}$, such that particular linear constraints are satisfied.

The primal formulation for decision problems of this type is:

$$\begin{aligned} \textit{minimise}\ \ & \mathbb{E}\left[\sum_{t=1}^{T} c_t(\xi^{\,t})^\mathsf{T} x_t(\xi^{\,t})\right] \\ \textit{subject to}\ \ & x_t \in \mathcal{L}^2_{k^t,n_t},\ \forall t \in \tau \\ & \mathbb{E}\left[\sum_{s=1}^{T} A_{t,s} x_s(\xi^{\,s})\right] \leq b_t(\xi^{\,t}) \qquad \mathbb{P}-a.s. \quad \forall t \in T \end{aligned} \tag{2.3.4.2}$$

Standard Form

We augment the primal problem eq. (2.3.4.2) with a sequence of non-anticipative slack variables $s_t \in \mathcal{L}^2_{k^t,m_t}$ for all $t \in T$ to yield the following standard form:

$$\begin{aligned} &\textit{minimise} \;\; \mathbb{E}\left[\sum_{t=1}^{T} c_t(\xi^t)^\intercal x_t(\xi^t)\right] \\ &\textit{subject to} \;\; x_t \in \mathcal{L}^2_{k^t,n_t},\; s_t \in \mathcal{L}^2_{k^t,m_t} \;\; \forall t \in T \\ &\qquad \left.\begin{aligned} &\mathbb{E}\left[\textstyle\sum_{s=1}^{T} A_{t,s} x_s(\xi^s)\right] + s_t(\xi^t) = b_t(\xi^t) \\ &s_t(\xi^t) \geq 0 \end{aligned}\right\} \;\; \mathbb{P} - a.s. \;\; \forall t \in T \end{aligned} \tag{2.3.4.3}$$

Its dual form is[2]:

$$\begin{aligned} &\textit{minimise} \;\; \mathbb{E}\left[\sum_{t=1}^{T} c_t(\xi^t)^\intercal x_t(\xi^t)\right] \\ &\textit{subject to} \;\; x_t \in \mathcal{L}^2_{k^t,n_t},\; s_t \in \mathcal{L}^2_{k^t,m_t} \;\; \forall t \in T \\ &\qquad \left.\begin{aligned} &\mathbb{E}\left[\textstyle\sum_{s=1}^{T} (A_{t,s} x_s(\xi^s) + s_t(\xi^t) - b_t(\xi^t)\right] (P_t \xi)^\intercal = 0 \\ &s_t(\xi^t) \geq 0 \end{aligned}\right\} \;\; \mathbb{P} - a.s. \;\; \forall t \in T \end{aligned} \tag{2.3.4.4}$$

Well-definition

For well-definition of eq. (2.3.4.3), we ascertain that vectors $c_t(\xi^t)$ and $b_t(\xi^t)$ are linear non-anticipative combinations of the uncertain elements ξ^t. For that reason, we can assume without proof that $\exists C_t \in \mathbb{R}^{n_t \times k^t}$ such that $c_t(\xi^t) = C_t P_t \xi$ and $\exists B_t \in \mathbb{R}^{m_t \times k^t}$ such that $b_t(\xi^t) = B_t P_t \xi$. We focus on fixed recourse for multi-stage stochastic programs, and therefore assume the recourse matrices $A_{t,s} \in \mathbb{R}^{m_t \times n_s}$ to not depend on ξ. In addition, we presume that the random variables $\{\ \xi_t\ \}_{t \in T}$ are independent, which implies that $\mathbb{E}_{\xi,t}(\xi)$ is affinely dependent on the uncertain parameters. For notational semantics, we point out that n_t and m_t determine the number of decisions taken up to time t and the number of constraints for time t respectively.

2.3.5 Worst-case Stochastic Program (WCSP)

Worst-case optimisation closely models decision-making under uncertainty where the decision-maker has insufficient information about the probability distribution of the underlying problem's uncertain data elements. For this class of optimisation problems, we are unable to formulate an optimisation model that aims to minimise the expected cost of the decision-maker's policy selections.

[2] We refer the reader to the research paper[2] for its derivation.

General Formulation I

If we assume that there is an identifiable family $\mho$ of fitting probability distributions for the uncertain parameter ξ, then the generic formulation of the worst-case mini-max optimisation problem is[31]:

$$\begin{aligned} &\underset{x \in \mathcal{L}^2_{k,n}}{\textit{minimise}} \ \sup_{\mathbb{P} \in \mho} \{ \, \mathbb{E}[c(x, \xi)] \, \} \\ &\textit{subject to} \ \ A(\xi)x(\xi) \leq b(\xi), \qquad \mathbb{P} - a.s. \end{aligned} \tag{2.3.5.1}$$

Kuhn et al[2] investigated a generalised stochastic programming model in which the probability distribution for some of the random vectors are known, and for the remaining random vectors only the polyhedral support of their distributions are known. In this situation, the goal is to minimise the expected value of the worst-case cost function $c(x, \xi)$ with respect to the expectation for the known random vectors. The worst-case is determined with respect to the finite support of type eq. (2.3.3.4) for a partly-unknown probability measure $\mathbb{P}$.

Uncertainty Model

We now introduce parameters $\eta \in \mathbb{R}^{k_\eta}$ and $\zeta \in \mathbb{R}^{k_\zeta}$, where $k_\eta + k_\zeta = k$, to model the random vectors ξ as the tuple (η, ζ). We assume the marginal distribution of η is fully known. We note that $k_\eta \geq 1$ since we know the marginal distribution of ξ_1 as the Dirac measure[3] concentrated at 1.

Furthermore, we suppose that the conditional distribution of ζ given η is unknown, but its conditional polyhedral support is available to the modeller:

$$\mathcal{Z}(\eta) = \{ \, \zeta \in \mathbb{R}^{k_\zeta} : (\eta, \zeta) \in \Xi \, \} \tag{2.3.5.2}$$

Risk-Averse General Formulation II

The robust form, using our new model for uncertainty, of the one-stage stochastic program as introduced in section 2.3.3 is:

$$\begin{aligned} &\textit{minimise} \ \ \mathbb{E}\left[\underset{\zeta \in \mathcal{Z}(\eta)}{\text{ess-sup}} \{ \, c^\intercal x(\eta, \zeta) \, \} \right] \\ &\textit{subject to} \ \ x \in \mathcal{L}^2_{k_{\eta n} + k_{\zeta, n}} \\ &\qquad\qquad\quad Ax(\eta, \zeta) \leq b(\eta, \zeta), \qquad \mathbb{P} - a.s. \end{aligned} \tag{2.3.5.3}$$

[3] a Dirac measure is a measure δ_x on a set X, with any σ-algebra of subsets of X, such that $\delta_x(\{ \, x \, \}) = 1$ for an arbitrarily chosen $x \in X$.

Standard Form

We standardise the primal problem eq. (2.3.5.3) by adding slack variables $s \in \mathcal{L}^2_{k_{\eta n}+k_{\zeta,m}}$ to obtain:

$$
\begin{aligned}
&\textit{minimise} \;\; \mathbb{E}[x_0(\eta)] \\
&\textit{subject to} \;\; x \in \mathcal{L}^2_{k_{\eta n}+k_{\zeta,n}},\; s \in \mathcal{L}^2_{k_{\eta n}+k_{\zeta,m}} \\
&\qquad \left.\begin{aligned}
&c^\mathsf{T} x(\eta,\zeta) + s_0(\eta,\zeta) = x_0(\eta) \\
&Ax(\eta,\zeta) + s(\eta,\zeta) = b(\eta,\zeta) \\
&s_0(\eta,\zeta) \geq 0 \\
&s(\eta,\zeta) \geq 0
\end{aligned}\right\} \mathbb{P}-a.s.
\end{aligned}
\tag{2.3.5.4}
$$

To justify the equivalence of eq. (2.3.5.4) and eq. (2.3.5.3), we point out that x_0 is independent of the unknown random parameters ζ and only depends on η for which its distribution is fully known. We repeat the remark from the paper[2] that the conditions

$$
\left.\begin{aligned}
x_0 * (\eta) \quad &= \operatorname*{ess-sup}_{\zeta \in \mathcal{Z}(\eta)} \{\, c^\mathsf{T} x * (\eta,\zeta) \,\} \\
s_0(\eta,\zeta) \quad &= x_0 * (\eta) - c^\mathsf{T} x * (\eta,\zeta)
\end{aligned}\right\} \mathbb{P}-a.s.
\tag{2.3.5.5}
$$

constrain any optimal solution $(x*, s*, x_0*, s_0*)$ to eq. (2.3.5.4).

Its dual form is[4]:

$$
\begin{aligned}
&\textit{minimise} \;\; \mathbb{E}[x_0(\eta)] \\
&\textit{subject to} \;\; x \in \mathcal{L}^2_{k_{\eta n}+k_{\zeta,n}},\; s \in \mathcal{L}^2_{k_{\eta n}+k_{\zeta,m}} \\
&\qquad \left.\begin{aligned}
&\mathbb{E}[c^\mathsf{T} x(\eta,\zeta) + s_0(\eta,\zeta) - x_0(\eta)] = 0 \\
&\mathbb{E}[Ax(\eta,\zeta) + s(\eta,\zeta) - b(\eta,\zeta)] = 0 \\
&s_0(\eta,\zeta) \geq 0 \\
&s(\eta,\zeta) \geq 0
\end{aligned}\right\} \mathbb{P}-a.s.
\end{aligned}
\tag{2.3.5.6}
$$

Well-definition

For the stochastic program eq. (2.3.5.3) to be well-defined, the valuation

$$
n \longmapsto \operatorname*{ess-sup}_{\zeta \in \mathcal{Z}(\eta)} \{\, c^\mathsf{T} x(\eta,\zeta) \,\}
\tag{2.3.5.7}
$$

[4]We refer the reader to section 2.3.3 and the research paper[2] for an explanation of its derivation.

must be a measurable function with an integrable minorant[5] $\forall x \in \mathcal{L}_{k_{\eta n}+k_{\zeta,n}{}^2}$.

[5]If $\exists \beta \in B$ such that $\forall \alpha \in A$, $\beta \leq \alpha$, where $A \subset B$ and B is an ordered set, then β is the *minorant* of A.

3

Decision Rule Approximation

3.1 Computational Intractability of Recourse Problems

Stochastic linear programming problems are considerably much more difficult to solve than their deterministic counterparts. When the random data follows a continuous distribution, multivariate integration must be performed in order to compute the expected costs of each stage.

Dyer and Stougie formally verified the complexity associated with dynamic decision problems under uncertainty. By assuming stochastic parameters are independently distributed, they were able to theoretically qualify one-stage stochastic programming problems as #P-hard and multi-stage stochastic programming problems as #PSPACE-hard in computational complexity[3].

Another complication of stochastic problems is the requirement for the exact probability distribution of uncertain elements to be supplied for random sampling. For real-life decision-making problems, we can appreciate that defining such exact distributions is not always possible.

3.2 Linear Approximations of Recourse Problems

3.2.1 Recourse-constrained One-stage Stochastic Program

Thus far we have considered recourse decisions of the form $x(\xi) \in \mathbb{R}^n$ such that $x \in \mathcal{L}^2_{k,n}$ for one-stage recourse programs. By introducing linear decision rules, we restrict the functional form of $x(\xi)$ to be linear combinations of ξ. Thus we further truncate the feasible region to those solutions which are of the form $x(\xi) = X\xi$ for a $X \in \mathbb{R}^{n \times k}$. For fixed recourse problems we require $s(\xi) = S\xi$ for a $S \in \mathbb{R}^{m \times k}$. However, for random recourse problems we will instead have $s_\mu(\xi) = \xi^\mathsf{T} S_\mu \xi$ for a $S_\mu \in \mathbb{S}^k$, where $\mu = 1, \cdots, m$.

3.2.2 Multi-stage Stochastic Program with Fixed Recourse

When we consider multi-stage stochastic programs with fixed recourse, we reduce the region of admissible decision rules to those of the form $x_t(\xi) = X_t P_t \xi$ for a $X_t \in \mathbb{R}^{n_t \times k^t}$ and $s_t(\xi) = S_t P_t \xi$ for a $S_t \in \mathbb{R}^{m_t \times k^t}$, where $t \in T$.

3.2.3 One-stage Worst-case Stochastic Program with Fixed Recourse

For worst-case optimisation of one-stage fixed recourse problems, we make the following decision rule linearisations:

- $\exists X \in \mathbb{R}^{n\times k}$, $x(\xi) = X\xi$.
- $\exists S \in \mathbb{R}^{m\times k}$, $s(\xi) = S\xi$.
- $\exists \chi \in \mathbb{R}^{k_\eta}$, $x_0(\eta) = \chi^\mathsf{T} P_\eta \xi$.
- $\exists \sigma \in \mathbb{R}^{k}$, $s_0(\eta) = \sigma^\mathsf{T} \xi$.

3.3 Computational Benefits of Linear Decision Rules

The benefit of using linear decision rules is that the stochastic program now has a finite number of decision variables. However, the problem still has a semi-infinite number of constraints, which means it is still not easily solved. Through the use of robust optimisation techniques, we can reduce the number of constraints to a finite set for semidefinite programs. In the following section we only present the final results of linearising the decision rules, we refer the reader to the research paper[2] for the step-by-step derivations.

3.4 Tractable Approximations for Recourse-Constrained Stochastic Programs

By applying linear decision rules as a standard robust optimisation technique, we can define a conservative approximation, which is the primal formulation‘ of a stochastic program. Similarly, by imposing linear decision rules on the dual of the original problem, we can form a semidefinite program representing the progressive approximation.

For the following sections, we introduce the matrix $M = \mathbb{E}[\xi\xi^\mathsf{T}]$ as the second-order moment matrix equipped with the probability measure $\mathbb{P}$.

3.4.1 One-stage Stochastic Program with Fixed Recourse

Below we present the respective conservative and progressive approximations for fixed recourse problems with one-stage.

$$\begin{aligned} \textit{minimise} \quad & \mathbb{T}race(MC^\mathsf{T} X) \\ \textit{subject to} \quad & X \in \mathbb{R}^{n\times k},\ \Lambda \in \mathbb{R}^{m\times l} \\ & AX + \Lambda W = B \\ & \Lambda h \geq 0 \\ & \Lambda \geq 0 \end{aligned} \tag{Cons-SP$_{fixed}$}$$

where Λ is a matrix of decision vectors.

$$\begin{aligned} minimise\ & \mathbb{T}race(MC^{\mathsf{T}}X) \\ subject\ to\ & X \in \mathbb{R}^{n\times k},\ S \in \mathbb{R}^{m\times k} \\ & AX + S = B \\ & (W - he_1^{\mathsf{T}})MS^{\mathsf{T}} \geq 0 \\ & SMe_1 \geq 0 \end{aligned} \tag{Prog-SP$_{fixed}$}$$

where S is a matrix of decision vectors.

3.4.2 One-stage Stochastic Program with Random Recourse

The semidefinite program representing the conservative approximation for random recourse problems with one stage is:

$$\begin{aligned} minimise\ & \mathbb{T}race(MC^{\mathsf{T}}X) \\ subject\ to\ & X \in \mathbb{R}^{n\times k},\ S = [S_1, \cdots, S_m] \in \mathbb{S}^m,\ \Lambda \in \mathbb{R}^{m\times l} \\ & \left.\begin{aligned} & \tfrac{1}{2}(\xi^{\mathsf{T}}A_\mu X\xi + X^{\mathsf{T}}A_\mu^{\mathsf{T}}) + S_\mu = \tfrac{1}{2}(e_1 b_\mu^{\mathsf{T}} + b_\mu e_1^{\mathsf{T}}) \\ & S_\mu - \textstyle\sum_{\ell=1}^{l} \Lambda_{\mu,l} W_\ell \succeq 0 \end{aligned}\right\} \quad \forall \mu \in \{\ 1, \cdots, m\ \} \\ & \Lambda \geq 0 \end{aligned} \tag{Cons-SP$_{random}$}$$

where X and Λ are the matrices of decision vectors.

The semidefinite program for the progressive approximation is:

$$\begin{aligned} minimise\ & \mathbb{T}race(MC^{\mathsf{T}}X) \\ subject\ to\ & X \in \mathbb{R}^{n\times k},\ S = [S_1, \cdots, S_m] \in \mathbb{S}^m \\ & \left.\begin{aligned} & \tfrac{1}{2}(\xi^{\mathsf{T}}A_\mu X\xi + X^{\mathsf{T}}A_\mu^{\mathsf{T}}) + S_\mu = \tfrac{1}{2}(e_1 b_\mu^{\mathsf{T}} + b_\mu e_1^{\mathsf{T}}) \\ & \mathbb{T}race(W_\ell Q(S_\mu)) \geq 0 \\ & Q(S_\mu) \succeq 0 \end{aligned}\right\} \quad \forall \mu, \ell \in \{\ 1, \cdots, m\ \} \end{aligned} \tag{Prog-SP$_{random}$}$$

where X and S_μ are the matrices of decision vectors. The linear function $Q : \mathbb{R}^{k\times k} \longrightarrow \mathbb{R}^{k\times k}$ is a symmetric tensor of all moments of probability measure $\mathbb{P}$ up to the fourth order:

$$e_\alpha^{\mathsf{T}} Q(e_\beta e_\gamma^{\mathsf{T}}) e_\delta = \mathbb{E}[\xi_\alpha\, \xi_\beta\, \xi_\gamma\, \xi_\delta], \quad \forall \alpha, \beta, \gamma, \delta \in \{\ 1, \cdots, k\ \}. \tag{3.4.2.1}$$

In eq. (3.4.2.1), the set $\{\ e_\alpha\ \}_{\alpha=1}^{k}$ represents the standard basis of the real space $\mathbb{R}^k$.

3.4.3 Multi-stage Stochastic Program

We extend the linear decision rule approximations of one-stage fixed recourse problems for sequential decision-making processes that evolve over several time periods.

For the following approximations we introduce $M_t \in Re^{k \times k^t}$ as the conditional second-order moment matrix for stage t, and is defined through $\mathbb{E}_\xi[\xi \mid \xi^t] = M_t P_t \xi$. Additionally, we note that the sizes of the linearised stochastic programs are now polynomial in $k, l, m = \sum_{t=1}^{T} m_t$ and $n = \sum_{t=1}^{T} n_t$.

The conservative approximation is:

$$\begin{array}{ll} \textit{minimise} & \sum_{t=1}^{T} \mathbb{T}race(P_t M P_t^\intercal C_t^\intercal X_t) \\ \textit{subject to} & \left.\begin{array}{l} X_t \in \mathbb{R}^{n_t \times k^t},\ \Lambda_t \in \mathbb{R}^{m_t \times l} \\ \sum_{t=1}^{T} A_{t,s} X_s P_s M_t P_t + \Lambda_t W = B_t P_t \\ \Lambda_t h \geq 0 \\ \Lambda_t \geq 0 \end{array}\right\} \quad \forall t \in T \end{array} \tag{Cons-MSP$_{fixed}$}$$

where X_t and Λ_t are the matrices of decision vectors for stage t.

The progressive approximation is:

$$\begin{array}{ll} \textit{minimise} & \sum_{t=1}^{T} \mathbb{T}race(P_t M P_t^\intercal C_t^\intercal X_t) \\ \textit{subject to} & \left.\begin{array}{l} X_t \in \mathbb{R}^{n_t \times k^t},\ S_t \in \mathbb{R}^{m_t \times k^t} \\ \sum_{t=1}^{T} A_{t,s} X_s P_s N_t P_t + S_t P_t = B_t P_t \\ (W - h e_1^\intercal) M P_t^\intercal S_t^\intercal \geq 0 \\ S_t P_t M e_1 \geq 0 \end{array}\right\} \quad \forall t \in T \end{array} \tag{Prog-MSP$_{fixed}$}$$

where $N_t = M P_t^\intercal (P_t M P_t^\intercal)^{-1}$. X_t and S_t are the matrices of decision vectors for stage t.

3.4.4 Worst-case Stochastic Program (WCSP)

We also consider the probable situation where the modeller does not have a complete knowledge of the probability distribution of the random vector ξ. In this section we present the linear decision rule approximations to the worst-case optimisation problems. We refer the reader to the previous sections for further explanatory details on the derivation of the subsequent approximations.

In the stochastic programs below, we introduce the new truncation operator P_η:

$$P_\eta : \mathbb{R}^k \longrightarrow \mathbb{R}^{k_\eta},\ (\eta, \zeta) \longmapsto \eta \tag{3.4.4.1}$$

The conservative approximation is:

$$\begin{aligned}
\textit{minimise}\ \ & \chi^\intercal P_\eta M e_1 \\
\textit{subject to}\ \ & X \in \mathbb{R}^{n\times k},\ \Lambda \in \mathbb{R}^{m\times l} \\
& \chi \in \mathbb{R}^{k_\eta},\ \lambda \in \mathbb{R}^{l} \\
& c^\intercal X + \lambda^\intercal W = \chi^\intercal P_\eta \\
& AX + \Lambda W = B \\
& \Lambda h \geq 0 \\
& \lambda^\intercal h \geq 0 \\
& \Lambda \geq 0 \\
& \lambda \geq 0
\end{aligned} \tag{Cons-WCSP$_{fixed}$}$$

where X and Λ are the matrices of decision vectors.

The progressive approximation is:

$$\begin{aligned}
\textit{minimise}\ \ & \chi^\intercal P_\eta M e_1 \\
\textit{subject to}\ \ & X \in \mathbb{R}^{n\times k},\ S \in \mathbb{R}^{m\times k} \\
& \chi \in \mathbb{R}^{k_\eta},\ \sigma \in \mathbb{R}^{k} \\
& c^\intercal X + \sigma^\intercal = \chi^\intercal P_\eta \\
& AX + S = B \\
& (W - h e_1^\intercal) M S^\intercal \geq 0 \\
& (W - h e_1^\intercal) M \sigma \geq 0 \\
& S M e_1 \geq 0 \\
& \sigma M e_1 \geq 0
\end{aligned} \tag{Prog-WCSP$_{fixed}$}$$

where X and S are the matrices of decision vectors.

3.5 Loss of Optimality

The solutions computed from linearising the decision rules are seldom optimal due to the inherent approximation errors. However, we appreciate that this a trade-off for tractability. This is remarked in Shapiro and Nemirovski's paper *On Complexity of Stochastic Programming Problems*[31]:

> *The only reason for restricting ourselves with affine decision rules stems from the desire to end up with a computationally tractable problem. We do not pretend that affine decision rules approximate well the optimal ones - whether it is so or not, it depends on the problem, and we usually have no possibility to understand how good in this respect is*

a particular problem we should solve. The rationale behind restricting to affine decision rules is the belief that in actual applications it is better to pose a modest and achievable goal rather than an ambitious goal which we do not know how to achieve.

For this reason, we quantify the differences in the primal and dual optimal solutions of the linearised stochastic programs to measure the loss of optimality incurred by the linear decision rule approximation.

3.6 An Illustrative Example - The Newsvendor Problem

The *newsvendor problem* is probably the most simplest form of a stochastic program. We repeat this example from the paper[34] to demonstrate the linear decision rules approach.

3.6.1 Description

A newspaper vendor faces the dilemma of deciding how many newspapers to order from an external supplier before knowing the actual demand, which itself is non-deterministic.

3.6.2 Problem Set-up

We denote the cost per x units of newspapers as c'. and the retail price stipulated by the newspaper vendor as p, where we insist $p > c'$ for profitability.

The demand d subject to uncertainty is a function of a random variable ξ equipped with a probability measure $\mathbb{P}$ and support Ξ. The random demand is representable as $d(\xi) = \xi$. We assume ξ has mean μ and variance σ^2. We represent the demand satisfied from the inventory as $-x'(\xi)$. To further simplify the model, we stipulate that newspapers ordered in excess of the demand have no salvage value and are therefore thrown away.

3.6.3 Stochastic Optimisation Formulation

In this problem, the goal is to increase profit. We can formulate the newsvendor model as:

$$\begin{aligned} \textit{minimise} \quad & c'x + \mathbb{E}_\xi[\,px'(\xi)\,] \\ \textit{subject to} \quad & x'(\xi) \geq -x \\ & x'(\xi) \geq -d(\xi) \\ & x \geq 0 \\ & x'(\cdot) \in \mathcal{Y} \end{aligned} \tag{3.6.3.1}$$

where the set $\mathcal{Y}$ denotes the space of linear functions from $\mathbb{R}^n$ to $\mathbb{R}^{n_2}$. We remind the reader that $n = \sum_{t=1}^{T} n_t$.

We can standardise eq. (3.6.3.1) by adding non-anticipative slack variables $\alpha(\xi), \beta(\xi)$ and $\gamma(\xi)$

to obtain the equivalent form:

$$\begin{aligned}
&\mathit{minimise}\ \ c^\top x + \mathbb{E}_\xi[p\,x'(\xi)] \\
&\mathit{subject\ to}\ \ \left.\begin{aligned}
&-x+\alpha(\xi) && = 0 \\
&x + x'(\xi) - \beta(\xi) && = 0 \\
&x'(\xi) - \gamma(\xi) && = -d(\xi) \\
&\alpha(\xi), \beta(\xi), \gamma(\xi) \geq 0 && \\
&x'(\cdot), \alpha(\xi), \beta(\xi), \gamma(\xi) \in \mathcal{Y} &&
\end{aligned}\right\}\ \mathbb{P}-a.s.
\end{aligned} \tag{3.6.3.2}$$

3.6.4 Multi-stage Stochastic Program Formulation

We observe that eq. (3.6.3.1) is an instance of a multi-stage stochastic program with fixed recourse.

$$\begin{aligned}
&\mathit{minimise} && \mathbb{E}_\xi\left[\sum_{t=1}^{T} c_t(\xi^t)^\top x_t(\xi^t)\right] \\
&\mathit{subject\ to} && x_1 \in \mathcal{L}^2_{k^1,n_1}, x_2 \in \mathcal{L}^2_{k^2,n_2},\ s_1 \in \mathcal{L}^2_{k^1,m_1}, s_2 \in \mathcal{L}^2_{k^2,m_2} \\
& && \left.\begin{aligned}
&A_{1,1}x_1(\xi^t) + s_1(\xi^t) && = b_1(\xi^t) \\
&A_{2,1}x_1(\xi^t) + A_{2,2}x_2(\xi^t) + s_2(\xi^t) && = b_2(\xi^t) \\
&s_1(\xi), s_2(\xi) \geq 0 &&
\end{aligned}\right\}\ \mathbb{P}-a.s.
\end{aligned} \tag{3.6.4.1a}$$

More specifically, for $T = 2$, the components of eq. (3.6.4.1a) are:

t	$\xi^{\mathbf{t}}$	$\mathbf{x_t}(\xi^{\mathbf{t}})$	$\mathbf{s_t}(\xi^{\mathbf{t}})$	$\mathbf{b_t}(\xi^{\mathbf{t}})$	$\mathbf{c_t}(\xi^{\mathbf{t}})$
1	ξ_1	x	α	0	c'
2	$[1, \xi_2]^\top$	$x'(\xi^t)$	$[\beta(\xi), \gamma(\xi)]^\top$	$[0, d(\xi_2)]^\top$	p

and the generated matrix components are:

t	$\mathbf{A_{t,1}}$	$\mathbf{A_{t,2}}$	$\mathbf{C_t}$	$\mathbf{B_t}$	$\mathbf{k^t}$	$\mathbf{n_t}$	$\mathbf{m_t}$
1	-1	0	c	0	1	1	1
2	$\begin{bmatrix}-1\\0\end{bmatrix}$	$\begin{bmatrix}-1\\-1\end{bmatrix}$	$\begin{bmatrix}p & 0\end{bmatrix}$	$\begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}$	2	2	2

We assume that the uncertain elements of the underlying newsvendor model follow a continuous uniform distribution, $\xi \sim U(a,b)$. Its probability density function is given as:

$$f_X(x) = \begin{cases} \dfrac{1}{b-a} & a \leq x \leq b \\ \\ 0 & \text{otherwise} \end{cases} \tag{3.6.4.1b}$$

where $a = 1$ and $b = 2$.

Noting that $\xi_1 = 1$ and $\xi \in [a,b]$, we can now define the support Ξ for the probability measure $\mathbb{P}$ as:

$$\begin{aligned} \Xi &= \{\ \xi = [\xi_1, \xi_2]^{\mathsf{T}} \in \mathbb{R}^k : \xi_1 = 1, a \leq \xi_2 \leq b\ \} \\ &= \{\ \xi = [\xi_1, \xi_2]^{\mathsf{T}} \in \mathbb{R}^k : W\xi \geq h\ \} \end{aligned} \tag{3.6.4.1c}$$

The inequality $W\xi \geq h$ expands to the following:

$$\begin{bmatrix} \xi_1 & & \\ -\xi_1 & & \\ -a\xi_1 & + & \xi_2 \\ b\xi_1 & - & \xi_2 \end{bmatrix} = \begin{bmatrix} \xi_1 & + & 0 \cdot \xi_2 \\ -\xi_1 & + & 0 \cdot \xi_2 \\ -a\xi_1 & + & \xi_2 \\ b\xi_1 & - & \xi_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -a & 1 \\ b & -1 \end{bmatrix} \xi \ \geq \ \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \end{bmatrix} \tag{3.6.4.1d}$$

and we can identity the components of the support as $W = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -a & 1 \\ b & -1 \end{bmatrix}$ and $h = \begin{bmatrix} 1 & -1 & 0 & 0 \end{bmatrix}^{\mathsf{T}}$.

3.6.5 Application of Linear Decision Rules

Informally speaking, we can generically represent the policy space by a super-vector that is row-wise indexable by time points t.

$$\left[x_1(\xi^1), x_2(\xi^2), \cdots, x_{T-1}(\xi^{T-1}), x_T(\xi^T)\right]^{\mathsf{T}} = \left[x_1^{\mathsf{T}}\xi^1, x_2^{\mathsf{T}}\xi^2, \cdots, x_{T-1}^{\mathsf{T}}\xi^{T-1}, x_T^{\mathsf{T}}\xi^T\right]^{\mathsf{T}} \tag{3.6.5.1a}$$

which we can expand as:

$$\begin{bmatrix} x_{1,1,1} + \sum_{j=2}^{k^1} x_{1,1,j}\,\xi_j \\ x_{2,1,1} + \sum_{j=2}^{k^2} x_{2,1,j}\,\xi_j \\ \vdots \\ x_{T-1,1,1} + \sum_{j=2}^{k^{T-1}} x_{T-1,1,j}\,\xi_j \\ x_{T,1,1} + \sum_{j=2}^{k} x_{T,1,j}\,\xi_j \end{bmatrix}. \tag{3.6.5.1b}$$

For the newsvendor problem, we have the following policy space:

$$\begin{bmatrix} x_{1,1,1} + \sum_{j=2}^{k^1} x_{1,1,j}\, \xi_{\ j} \\ x_{2,1,1} + \sum_{j=2}^{k^2} x_{2,1,j}\, \xi_{\ j} \end{bmatrix}. \tag{3.6.5.1c}$$

Thus, we have the linear decision rules:

$$\begin{aligned} x_1(\xi^{\,1}) &= X_1 P_1 \xi = x_{1,1,1} \\ x_2(\xi^{\,2}) &= X_2 P_2 \xi = x_{2,1,1} + x_{2,1,2} \end{aligned} \tag{3.6.5.1d}$$

where $X_1 = x_{1,1,1}$ and $X_2 = (x_{2,1,1}, x_{2,1,2})$.

We assume that the random variables $\{\ \xi_t\ \}_{t\in T}$ are independently distributed of each other. Thus we compute the expectation of ξ as:

$$\bar{\xi} = \mathbb{E}_\xi[\xi] = \mathbb{E}_\xi\left[\begin{pmatrix} \xi_1 \\ \xi_2 \end{pmatrix}\right] = \begin{pmatrix} \mathbb{E}_\xi[\xi_1] \\ \mathbb{E}_{\xi_2}[\xi_2] \end{pmatrix} = \begin{pmatrix} 1 \\ \mu \end{pmatrix} \tag{3.6.5.1e}$$

and the covariance matrix of ξ is calculated as:

$$\begin{aligned} \Omega_\xi = &\begin{pmatrix} \mathbb{C}ov[\xi_1,\xi_1] & \mathbb{C}ov[\xi_1,\xi_2] \\ \mathbb{C}ov[\xi_2,\xi_1] & \mathbb{C}ov[\xi_2,\xi_2] \end{pmatrix} \begin{pmatrix} \mathbb{E}_{\xi_1,\xi_1}[\xi_1\xi_1] - \mathbb{E}_{\xi_1}[\xi_1]\mathbb{E}_{\xi_1}[\xi_1] & \mathbb{E}_{\xi_1,\xi_2}[\xi_1\xi_2] - \mathbb{E}_{\xi_1}[\xi_1]\mathbb{E}_{\xi_2}[\xi_2] \\ \mathbb{E}_{\xi_2,\xi_1}[\xi_2\xi_1] - \mathbb{E}_{\xi_2}[\xi_2]\mathbb{E}_{\xi_1}[\xi_1] & \mathbb{E}_{\xi_2,\xi_2}[\xi_2\xi_2] - \mathbb{E}_{\xi_2}[\xi_2]\mathbb{E}_{\xi_2}[\xi_2] \end{pmatrix} \\ = &\begin{pmatrix} 0 & 0 \\ 0 & \sigma^2 \end{pmatrix} \end{aligned} \tag{3.6.5.1f}$$

which gives the second moment matrix M as:

$$M = \mathbb{E}_\xi[\xi\,\xi^{\mathsf{T}}] = \mathbb{E}_\xi\left[\begin{pmatrix} \xi_1\xi_1 & \xi_1\xi_2 \\ \xi_2\xi_1 & \xi_2\xi_2 \end{pmatrix}\right] = \mathbb{E}_\xi\left[\begin{pmatrix} 1 & \xi_2 \\ \xi_2 & {\xi_2}^2 \end{pmatrix}\right] = \begin{pmatrix} \mathbb{E}_\xi[{\xi_1}^2] & \mathbb{E}_\xi[\xi_2] \\ \mathbb{E}_\xi[\xi_2] & \mathbb{E}_\xi[{\xi_2}^2] \end{pmatrix}. \tag{3.6.5.1g}$$

We point out that $\mathbb{V}ar_\xi[\xi_2] = \mathbb{E}_\xi[(\xi_2 - \mathbb{E}_\xi[\xi_2])] = \mathbb{E}_\xi[\xi^2] - \mathbb{E}_\xi[\xi]^2$, which implies $\mathbb{E}_\xi[\xi^2] = \mathbb{V}ar_\xi[\xi_2] + \mathbb{E}_\xi[\xi]^2 = \sigma^2 + \mu^2$. Therefore,

$$M = \begin{pmatrix} 1 & \mu \\ \mu & \sigma^2 + \mu^2 \end{pmatrix}. \tag{3.6.5.1h}$$

As $M_t P_t \xi = M_t \xi^t = \mathbb{E}_\xi[\xi \mid \xi^{\,t}]$, we have:

$$
\begin{aligned}
M_1 \xi^1 &= \mathbb{E}_\xi[\xi \mid \xi^1] \\
&= \mathbb{E}_\xi\left[\begin{pmatrix}\xi_1, \xi_2\end{pmatrix}^\mathsf{T} \,\middle|\, \xi_1\right] \\
M_1 \xi_1 &= \begin{pmatrix}\mathbb{E}_\xi[\xi_1 \mid \xi_1] \\ \mathbb{E}_\xi[\xi_1 \mid \xi_2]\end{pmatrix} \\
M_1 &= \begin{pmatrix}1 \\ \mu\end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
M_2\, \xi_2^\mathsf{T} &= \mathbb{E}_\xi[\xi \mid \xi^2] \\
&= \mathbb{E}_\xi\left[\begin{pmatrix}\xi_1, \xi_2\end{pmatrix}^\mathsf{T} \,\middle|\, \begin{pmatrix}\xi_1, \xi_2\end{pmatrix}^\mathsf{T}\right] \\
M_2 \begin{pmatrix}\xi_1 & \xi_2\end{pmatrix} &= \begin{pmatrix}\mathbb{E}_\xi[\xi_1 \mid \xi_1 \cap \xi_2] \\ \mathbb{E}_\xi[\xi_2 \mid \xi_1 \cap \xi_2]\end{pmatrix} \\
M_2 \begin{pmatrix}\xi_1 & \xi_2\end{pmatrix} &= \begin{pmatrix}\xi_1 \\ \xi_2\end{pmatrix} \\
M_2 &= \begin{pmatrix}1 & 0 \\ 0 & 1\end{pmatrix}.
\end{aligned}
\tag{3.6.5.1i}
$$

4 Algebraic Modelling Languages

In this section, we discuss algebraic modelling languages with respect to a particular stochastic programming framework, and we define the vocabulary or language constructs required to describe stochastic programming problems. We then briefly discuss the current implementation, and then detail our own approach.

4.1 Re-defining the Stochastic Programming Framework

We remind the reader that, although the taxonomy of stochastic programming models extends beyond recourse problems, this project is steered in the direction of distribution-based recourse-constrained multi-stage stochastic programming problems. Thus, we can identify the stochastic framework for such problems necessitates language constructs for describing the temporal structure and the uncertain data elements (see table 4.1)[48].

Table 4.1: Algebraic modelling language requirements for distribution-based recourse-constrained multi-stage stochastic optimisation problems.

Model Data Components	**AML Requirements**
Meta-information for the stages that capture temporal structure of problem.	Explicit mappings of decision variables and constraints to individual stages.
Random entities which represent the underlying model's uncertain parameters.	Description of probability distributions and supports of probability measures for the random entities.

In figure (4.1), we have adapted the language constructs required for this particular stochastic framework from those used in the SAMPL platform[48].

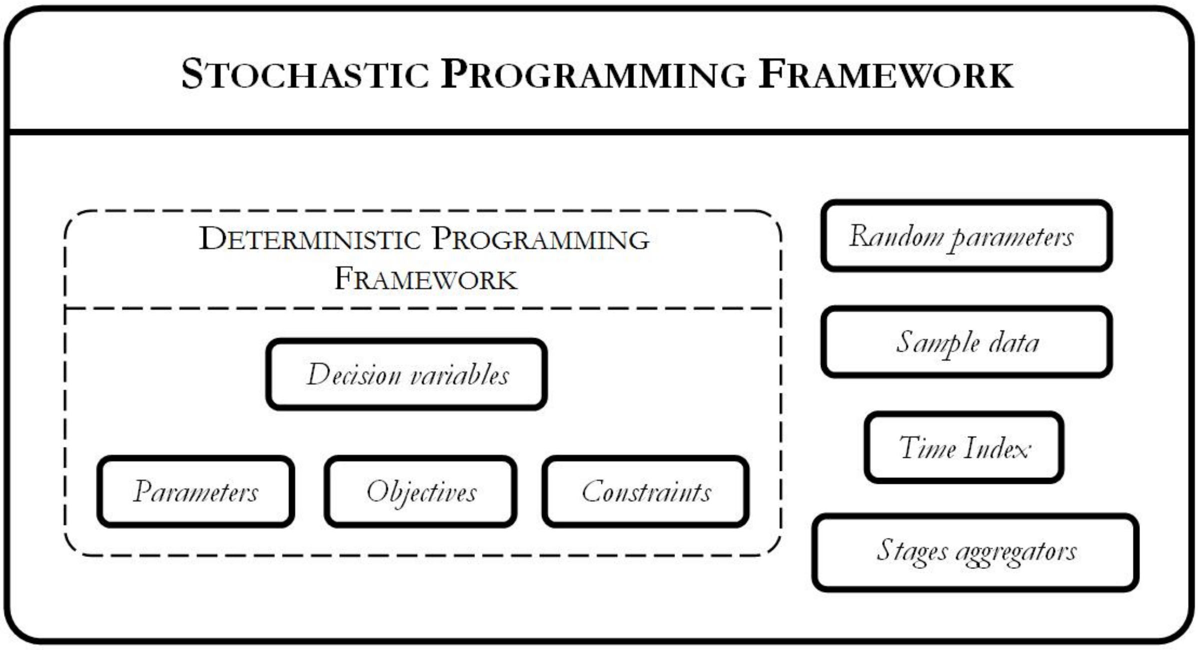


Figure 4.1: Language constructs for Stochastic Programming framework.

4.2 Related Work

The problem specification has been previously prototyped in C++[1]. However, the code-base for this solution is quite complicated due to lack of internal support for numerical computing in C++. Consequently, custom data structures and operations for respectively representing and manipulating matrices, affine functions and constraints have compulsorily been written from scratch. In addition, the scope for expressibility could be widened by introducing vocabulary for complex arithmetic expressions involving multiplication and nested parentheses.

For demonstration purposes, we refer the reader to listing 4.1, which presents a description of the newsvendor problem (see section 3.6) using the current C++ implementation[2]. We point out to the reader that the language constructs are those of the form `[<keyword>]`, and that the random variables are parameterised with two real-values which specify the default shape of its support. For clarity:

- the construct `[support]` describing the random variables' uncertainty sets,
- the paramterised random variables `<variable>[a,b]` which specify their distributions,
- and the construct `[samples_file]` for supplying information to sample these distributions

facilitate the decision-maker's modelling of his or her decision-making under uncertainty.

[1]Original design and implementation was by W. Wiesemann and A. Georghiou from Department of Computing, Imperial College London

[2]Re-produced with permission of W. Wiesemann and A. Georghiou.

Listing 4.1: Specification of the Newsvendor problem using the legacy system's algebraic modelling language.

```
! Newsvendor Model

[general]
no_periods 2;

[random_variables]
demand [5,10] known_at 2 breakpoints{};

[support]
demand <= 10;
5 <= demand;

[samples_file]
file samples.txt;

[decision_variables]
x at_period 1   with_objective 5;       ! cost = 5
w at_period 2   with_objective 10;      ! price = 10

[constraints]
-w - x <= 0;
-w <= demand;
-x <= 0;

[end]
```

4.3 Our Approach: The JADA Input Format

JADA provides a standardised input format for describing stochastic programming problems. In this section we aim to explain and justify our choice for the syntax and the semantics of JADA's algebraic modelling language.

Fundamentally, a `JADA` file consists of a single `model`, which is formally described via

- `general` meta-information,
- declarations of decision and random `variables`,
- `support` constraints for the random variables for the representation of uncertainty,
- data `samples` for the random variables for sampling their distributions,
- `recourse constraints`, and the
- `objective` function.

For user convenience, the language has been carefully designed to reflect the structure and mathematical notation of a linear programming problem. Although some of the subsections of the model could be merged, such as the recourse and support constraints, we feel that clarity is more important than brevity. Additionally, we have adopted the capability for source code documentation, as present in all programming languages, to allow the user to add in-lined or block comments to document their models.

Listing 4.2: Code listing showing the template of the JADA file for the *Newsvendor* problem.

```
1  //The Newsvendor Model Example
2  
3  Model
4  {
5  
6   /*
7    Specify the name of model and the number of stages.
8   */
9   General{...}
10 
11  /*
12   Specify the decision variables for how many newspapers to
13   buy from the supplier, and the random variable representing
14   the stochastic demand.
15  */
16  Variables{...}
17 
18 /*
19   Specify more restrictive constraints for the random variables
20   e.g. stochastic demand should be between five and ten units.
21  */
22  Support{...}
23 
24  /*
25   Specify the text file containing the sample data to derive
26   the expectation and second order moment from.
27  */
28  Samples{...}
29 
30  /*
31   Specify the constraints for the recourse decisions.
32  */
33  Constraints{...}
34 
35  /*
36   Specify the objective to minimise the expected wastage, from
37   overestimating the demand, in order to maximise profit.
38  */
39  Objective{...}
40 
41 }
42 
43 //End of model
```

The General Language Construct

The `General` language construct allows the modeller to declare administrative and temporal information such as the name of the model (line 5) and the number of stages (line 8).

Listing 4.3: Code listing showing the `General` subsection of the JADA file for the *Newsvendor* problem.

```
...
General
{
 //A descriptive name for the model
 name("Newsvendor Model - Experimental Example");

 //The multi-stage stochastic programming problem has two stages.
 stages(2);
}
...
```

Specification of the model's name is deemed important to achieve meaningful name mangling for the auto-generated results and log files, and to also permit the modeller to easily identify them in their temporary directory. Although the number of stages could be easily inferred from the declaration of the decision and random variables, an explicit declaration of the intended number of stages allows for efficient cross-validation and internal initialisation of the parser. Both the `name` and `stages` attributes are mandatory.

The Variables Language Construct

The `Variables` subsection consists of declarations of the linear program's decision and random parameters.

Listing 4.4: Code listing showing the `Variables` subsection of the JADA file for the *Newsvendor* problem.

```
1  ...
2  Variables
3  {
4   //Decision variable for no. newspapers to buy from the supplier to sell
        today
5   decision(x,1);
6
7   //Decision variable for no. newspapers to buy from the supplier to sell
        tomorrow
8   decision(w,2);
9
10  //Random variable representing the stochastic demand
11  random(demand,2,5,10);
12 }
13 ...
```

Declarations of the decision and random variables, using the reserved keywords `decision` (line 5) and `random` (line 11) respectively, are compulsorily parameterised by

- a unique alpha-numeric identifier for the variable (first parameter), and
- a natural number representing the stage to which the decision corresponds to (second parameter).

The random variables receive additional non-optional arguments representing the minimum (third parameter) and maximum (fourth parameter) values that a random variable can adopt.

These values define the support of the random variable's probability distribution.

The Constraints Language Construct

The `Constraint` language construct facilitates the declaration of the recourse constraints.

Listing 4.5: Code listing showing the `Constraints` subsection of the JADA file for the *Newsvendor* problem.

```
...
Constraints
{
  w + x >= 0;
  w >= -demand;
  x >= 0;
}
...
```

The modeller is not required to standardise the constraints, as the standardisation for the linear program is handled by JADA. Thus the constraints can be equalities or inequalities. In the former case, JADA replaces the equality constraints by a less-than-or-equal-to and a greater-than-or-equal to inequality. Ultimately, all greater-than-or-equal-to inequalities will be converted to less-than-or-equal-to inequalities by negation.

The Support Language Construct

The `Support` language construct permits the modeller to further restrict the support of the random variables by specifying additional constraints.

Listing 4.6: Code listing showing the `Support` subsection of the JADA file for the *Newsvendor* problem.

```
 ...
 Support
 {
  //Stochastic demand is between five and ten units inclusive.
  demand <= 10;
  5 <= demand;
 }
 ...
```

The minimum and maximum values given for the random variables provide the default support of their distribution, hence this subsection is optional.

The Samples Language Construct

The `Samples` subsection contains the sample data for generating the expectation and moments matrices. The modeller provides this information by stating an absolute path to a text-file containing real-values for the sample data points.

Listing 4.7: Code listing showing the `Samples` subsection of the JADA file for the *Newsvendor* problem.

```
...
Samples
{
  file("C:/optimisation/Newsvendor/samples.txt");
}
...
```

The intial implementation assumed that the user would specify the sample data for all the random variables in one file. However, we had not provided an explicitly structured format for doing this. Instead, we assumed that there was an equal number of sample data points for each random variable. Furthermore, the system assumed that the sample data points for each random variable are assigned to the random variables in the order that they are declared in the input file.

For clarification, suppose a JADA file contained just two random variables `rand1` and `rand2` such that `rand1` had been declared before `rand2` in the `Variables` subsection. Then the text-file containing the sample data points for the two random variables would have the implicit structure indicated in figure (4.2).

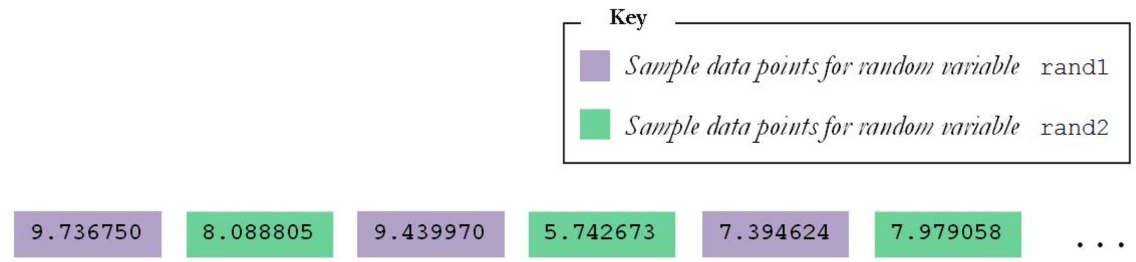


Figure 4.2: Diagram showing the assumed structure of the sample data file.

By mandating that header information be supplied, we are able to refactor the original format of the samples data file to eliminate ambiguity and as many syntactical errors as possible.

Listing 4.8: Code listing showing the sample data file for the *Newsvendor* problem.

```
SampleData
{

 Header
 {
   population(1);

   samplesize(1000);

   //We only have sample data for the stochastic demand
   variables(demand);
 }

 Data
 {
   9.736750, 8.088805, 9.439970, 5.742673,
   7.394624, 7.979058, 8.002340, 6.280136,
   ...
   ...
   ...
```

```
21      5.232897, 5.998800, 7.677094, 7.566908;
22    }
23
24  }
```

The header information contains

- the number of random variables for which the file provides sample data for (line 6),
- the number of sample data points per random variable (line 8), and
- the random variables in the order that their sample data has been declared (line 11).

We divide the samples data file into the header information, as described above, and the actual data. The syntax for the sample data points differ slightly from the original format in the sense that commas are used to delimit the sample data points rather than whitespaces, and the declared sample data points are terminated by a semi-colon.

The `variables` keyword in the header of the sample data file is included to not only disambiguate the order in which the sample data have been declared for the random variables, but to also indicate which random variables the sample data is applicable to. This allows the modeller to split the sample data across several text files for the random variables. Hence one text-file could hold sample data for random variable `rand1` and the other text-file for random variable `rand2`. Having said this, we do encourage use of a small number of sample data files to decrease the overhead of opening, reading and validating several sample data files. The second functionality of the `variables` keyword is that it permits the parser to check which random variables do not intentionally have any sample data. In this case, we sample the probability distribution of the random variable by using the minimum and maximum parameters of its support.

The Objective Language Construct

The modeller defines the objective function by providing the costs of the decision variables. The original design followed a more declarative style whereby the keyword `goal` was used to indicate the modeller's intention to either `maximise` or `minimise` the objective function (line 4). Moreover, the keyword `cost` was used to associate a real value with each decision variable declared (lines 5 and 6).

Listing 4.9: Code listing showing the initial design of the `Objective` subsection of the JADA file for the *Newsvendor* problem.

```
1  ...
2   Objective
3   {
4     goal(minimise);
5     cost(x,5);
6     cost(w,10);
7   }
8  ...
```

Although this form is explicit and intuitive, it has been deemed too verbose and tedious to use. Instead, we have made a decision for the objective function to adopt the same format as a linear program. Consequently, we allow the user to specify their objective function as an affine expression.

Listing 4.10: Code listing showing the final design of the `Objective` subsection of the JADA file for the *Newsvendor* problem.

```
...
 Objective
 {
   minimise expectation x[5] + w[10];
 }
...
```

This form is notably simpler and exhibits a certain degree of brevity that the original format lacked. The linear expression is a summation of variables multiplied by their costs. This multiplication is implied by the square brackets, which contain the cost expressions for the individual decision variables. The cost expressions are affinely dependent on the random variables, in this case the decisions `x` and `w` are functions of the degenerate random variable[3].

In extending the design for the `Objective` subsection, we assume the modeller will stipulate whether the objective function is to be minimised or maximised with respect to a statistical measure. Currently, JADA implements the algorithms for minimising or maximising the *expectation* of some linear function. However, JADA can be extended to consider optimising the variance of a linear expression, which will involve algorithms based on quadratic programming.

[3]We remind the reader that the degenerate dummy outcome ξ_1 is equal to 1. Hence the objective function is equivalent to `minimise expectation x[5*`ξ_1`] + w[10*`ξ_1`]`

5 Design and Implementation

In this chapter, we explain our choice of implementation language and state the uses of external libraries to realise the goals of this project. Subsequently, we present some graphical notation to visualize the architectural blueprint of the system, which includes the interfacing and the structure of the four different modules for:

- parsing and validating the input file,
- generating the temporal and matrix components of the *linear program* (LP),
- solving the generated LP, and
- rendering the results.

In explaining the design and implementation of these modules, we aim to justify our design choices, and explain any problems encountered and how we solve them.

5.1 Development Environment

Implementation Language and External Libraries

Due to the breadth of internal support for technical mathematical computing, we have chosen to implement the system in MATLAB. The latest versions of MATLAB (R2008a onwards) facilitate object-oriented programming to take advantages of code re-usability, inheritance, polymorphism, encapsulation and reference behaviour.

Initially, the aim has been to implement the entire system in MATLAB to allow for a maximally consistent code-base. However, as we detail in section 5.3, implementation of the parser using MATLAB's regular expressions library is inadequate for reading in the model. This limitation was called to our attention during our attempts to extend the JADA syntax to permit more complex mathematical expressions involving nested parentheses. Our solution is to specify the entire algebraic modelling language using a context free grammar which is itself expressed using Extended Backus Naur Form (EBNF).

YALMIP and the LMI Control Toolbox

The legacy system interfaces with the ILOG CPLEX optimisation software package, which is based on the SIMPLEX technique, to solve the linear programming models. To communicate with CPLEX, the legacy system passes the generated LP to a CPLEX solver interface as a `.lp` file. The optimal solutions are then extracted using regular expressions from a file generated by CPLEX, and are subsequently interpreted to present the results to the user. The overhead with performing I/O routines to communicate with CPLEX and the dependency on the assumed output format of the CPLEX solver is a prevailing issue. Additionally, the implementation of the legacy system limits its applicability by only catering to users of the CPLEX solvers.

For the reasons stated above, the JADA solver sub-system utilises YALMIP, a convex optimisation framework, to provide interfaces to a miscellany of popular solvers such as CPLEX, SeDuMi, CSDP, SDPA. YALMIP provides a variety of benefits, such as allowing for the low-level processing, that is required to simplify the models for soundness and efficiency, to be delegated to its internal routines. As a result, we can just focus on specifying the objective function and constraints, which are to be submitted via the `solvesdp` function. The decision matrices to be solved for are representable as YALMIP's multi-dimensional symbolic decision variables (`sdpvar`), and the numerical values of the declared decision variables, as well as the residual quantities of the constraints, can be accessed via the `double` command.

The linear matrix inequalities (LMI) in the constraints system of the approximation models characterise the linear programs as instances of semi-definite programs. The LMI control toolbox provides the linear matrix inequality variable (`lmivar`) to incrementally specify these systems of LMIs.

5.2 The Overall Design of JADA

In following good software engineering practices, we decouple the overall system architecture into six main components to facilitate modularity and extensibility. The primary packages illustrated in figure (5.1) are briefly described below.

The `parser` package contains the ANTLR implementation of the `ParserEngine` for reading and validating an optimisation problem specified in the JADA format. It includes an implementation of the JADAModel which represents the minimal data extracted from the supplied JADA file in order to solve the stochastic programming problem.

The `generator` package encapsulates static classes for generating the matrix components of the conservative and progressive linear programs.

The `approximator` package comprises the classes required for computing the objective function and constraints particular to the conservative and progressive approximations of the original stochastic program. Additionally, it provides an interface for interacting with the YALMIP convex optimisation framework to communicate with a variety of popular external solvers.

The `model` package contains the internal representation of the generated linear program. It com-

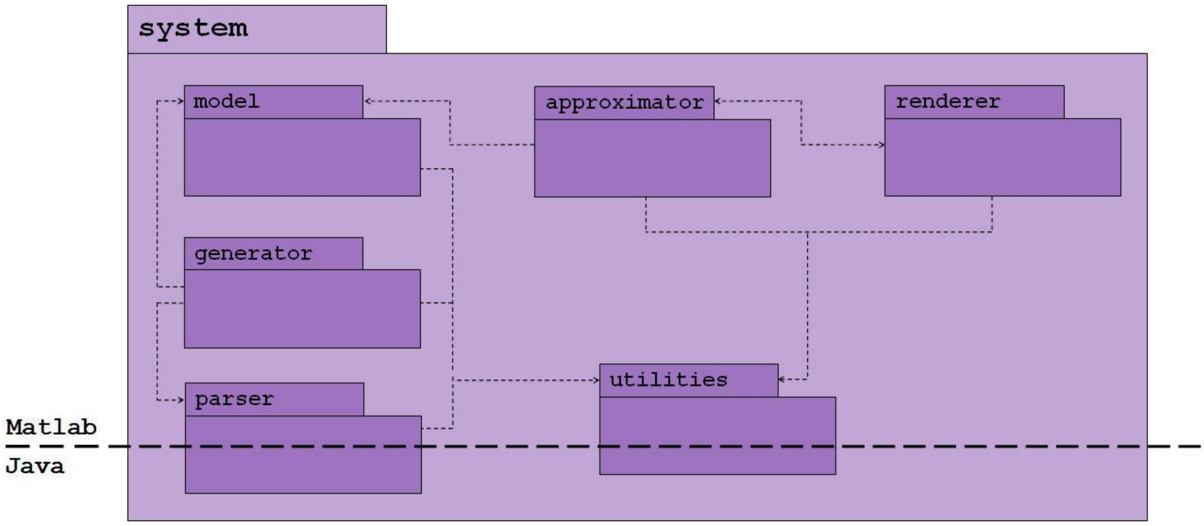


Figure 5.1: UML diagram showing the package structure, where the dashed arrows indicate the package dependencies. The separation between the Java implementation and MATLAB implementation is emphasised by the more solid dashed line.

pacts the `JADAModel` generated by the `Parser` and the `LPModel` constructed by the `LPGenerator` to produce the `OptimisationModel`. This merged model is then augmented to allow storage of computations specific to the `ConservativeApproximator` and `ProgressiveApproximator` classes.

The `renderer` package handles the presentation to the user of the generated linear program, the values of the solved variables and the optimal decisions as linear functions of their random variables.

The `utilities` package defines a set of re-usable functions and global static attributes for maintaining system-wide properties, performing exception handling and error propagation, implementing MATLAB's arithmetic operators in an object-oriented fashion, formatting text, and for manipulating generic data structures.

5.2.1 Pattern of Interaction

Communication between the user and the system is achieved via the JADA interface, which exposes functionality to solve a stochastic programming problem described in the JADA format. After initialising the JADA system, the interface provides functionality for parsing the model, generating the LP and computing the conservative and progressive solutions to the optimisation problem. Figure (5.2) illustrates the sequence of system interactions that occur during this process.

5.3 Parser

Having defined an intuitive and standardised format for specifying a stochastic programming problem, the next step is to design and implement a parser to

- read and syntactically analyse the supplied JADA file defined using the algebraic modelling language as described in section 4.3,
- extract data associated with a pre-defined set of tokens,
- validate the JADA file for correct syntax and semantics, and to
- generate an internal representation of the specified stochastic programming problem for conic programming instances to be generated.

5.3.1 Regular Expressions Implementation

Initially, the JADA parsing technology had been written solely in MATLAB to keep the implementation language consistent across the whole of the code-base. As programmtically explained in listing 5.1, the main function of the `Parser` class is to take as input an absolute path to a JADA file, which contains the stochastic programming problem, and to delegate extraction of the file contents to the `JADAFileReader` (line 7). The file reader's output is then piped to the `Tokeniser` to build the `JADAModel` (line 10). The relationships between these classes are diagrammatically explained in figure (5.3).

Listing 5.1: Code listing showing the `parseFile(...)` method defined in `Parser.m`.

```
1  % + Function Description: parses a JADA file
2  % + Function Input:       string representing absolute filepath to JADA file
3  % + Function Output:      a JADAModel
4  jadaModel = function parseFile(self, filePath)
5
6    % Get contents of file
7    fileContents = AMLFileReader.getFileContents(filePath);
8
9    % Extract tokens and build JADAModel
10   jadaModel = self.tokeniser.generateJADAModel(fileContents);
11
12 end %parseFile
```

Tokenisation

The `Tokeniser` class had been written to follow the *delegation* design pattern for object-oriented programming. Thus, by inversion of responsibility, the Tokeniser class (the delegate) has evolved to be a composition of several sub-tokenisers as illustrated in figure (5.4). These composite classes are responsible for extracting the tokens related to one of the six language constructs for specifying an instance of a multi-stage stochastic optimisation problem.

The `generateJADAModel()` method implemented by the Tokeniser iterates over the construct tokenisers to sequentially process a section of the JADA file. The details of this logic are given in listings A.1 and A.2. Each of these construct tokenisers provide its own implementation of the `IConstructTokeniser` interface, which specifies functionality for

(a) retrieving the construct's regular expression (`getRegex()`), and for

(b) processing the tokens extracted, having applied the regular expression to the contents of the file, to update an instance of a JADAModel (`processTokens(...)`).

Regular Expressions

As shown in the second column of tables 5.2 to 5.4, each sub-tokeniser formulates a regular expression to match a particular section of the contents of the JADA file. To assist with the construction of these expressions, a set of utility regexes had been pre-defined (see table 5.1).

Construction of the Internal Model

The `processTokens(...)` method defined in the `IConstructTokeniser` interface provides the functionality for building an internal representation of the contents of the JADA file . The third column of tables 5.2 to 5.4 briefly outlines the incremental construction of the JADA model with respect to each of the construct tokenisers.

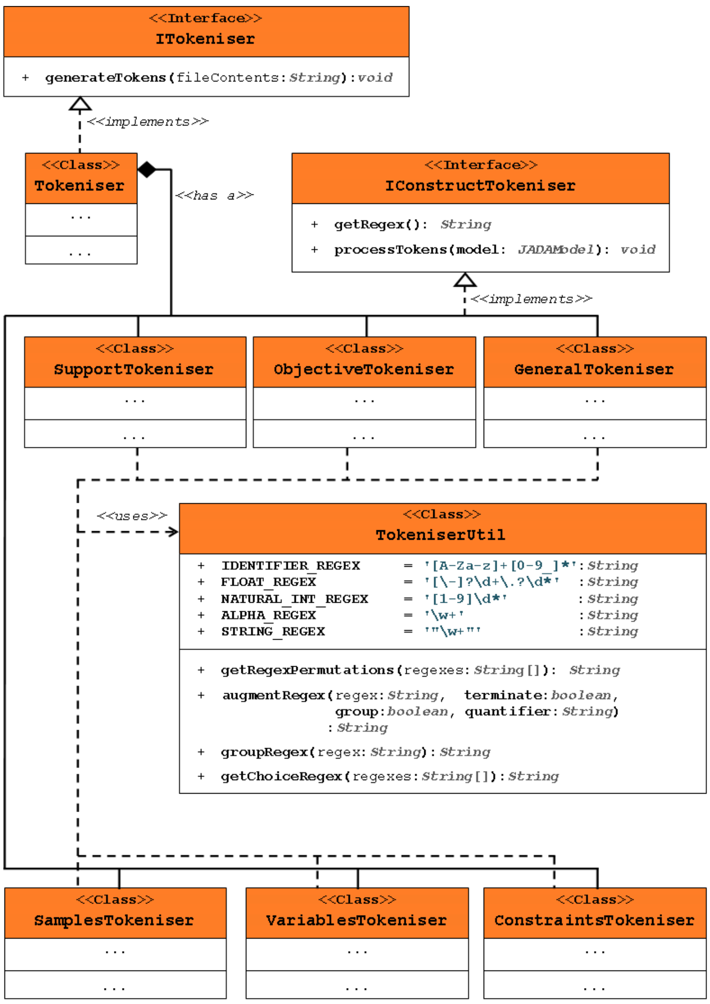


Figure 5.4: UML class diagram showing the structural implementation of the tokenisation component for the `Parser` class.

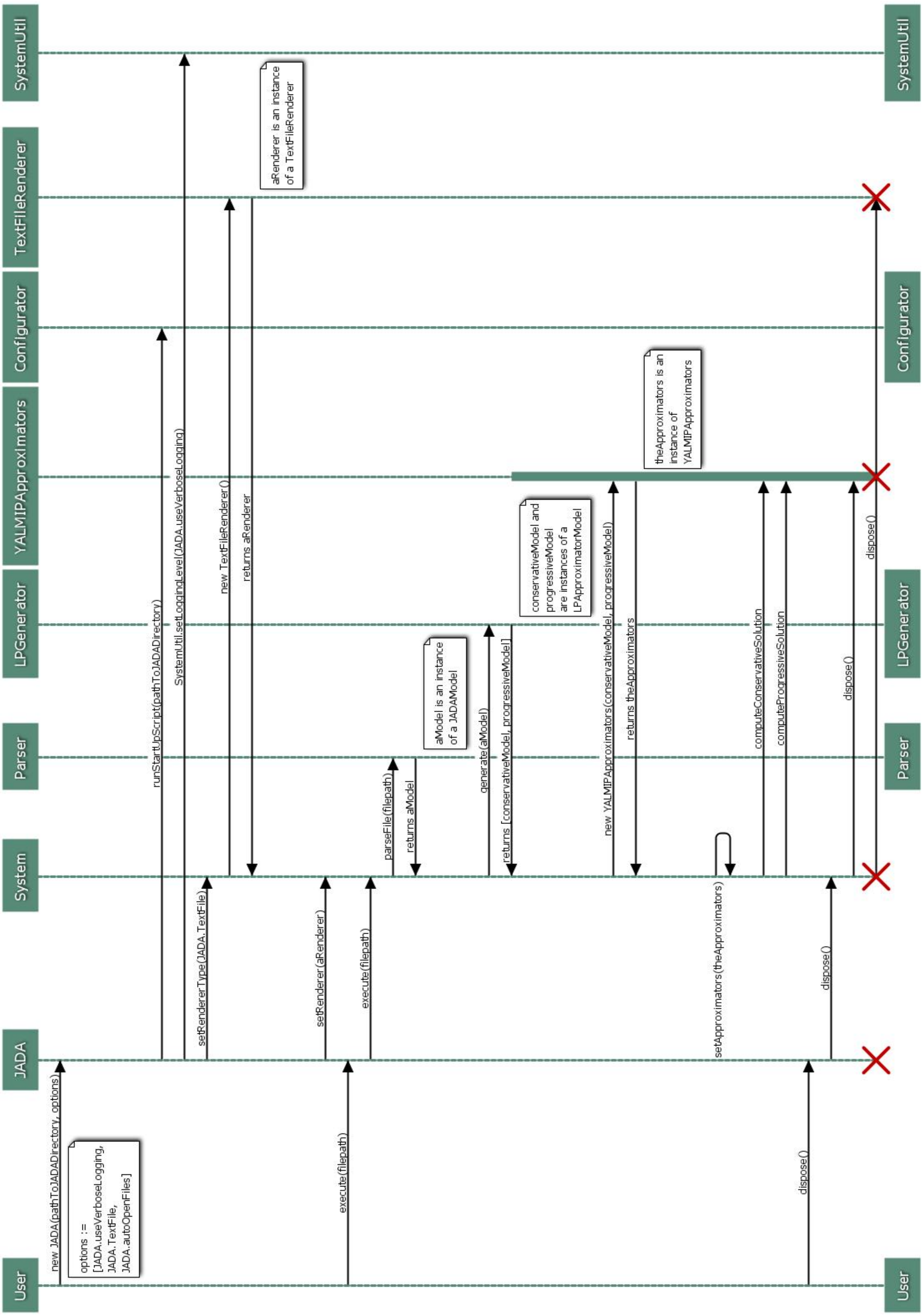


Figure 5.2: Sequence diagram explaining the pattern of the user-system interaction for JADA.

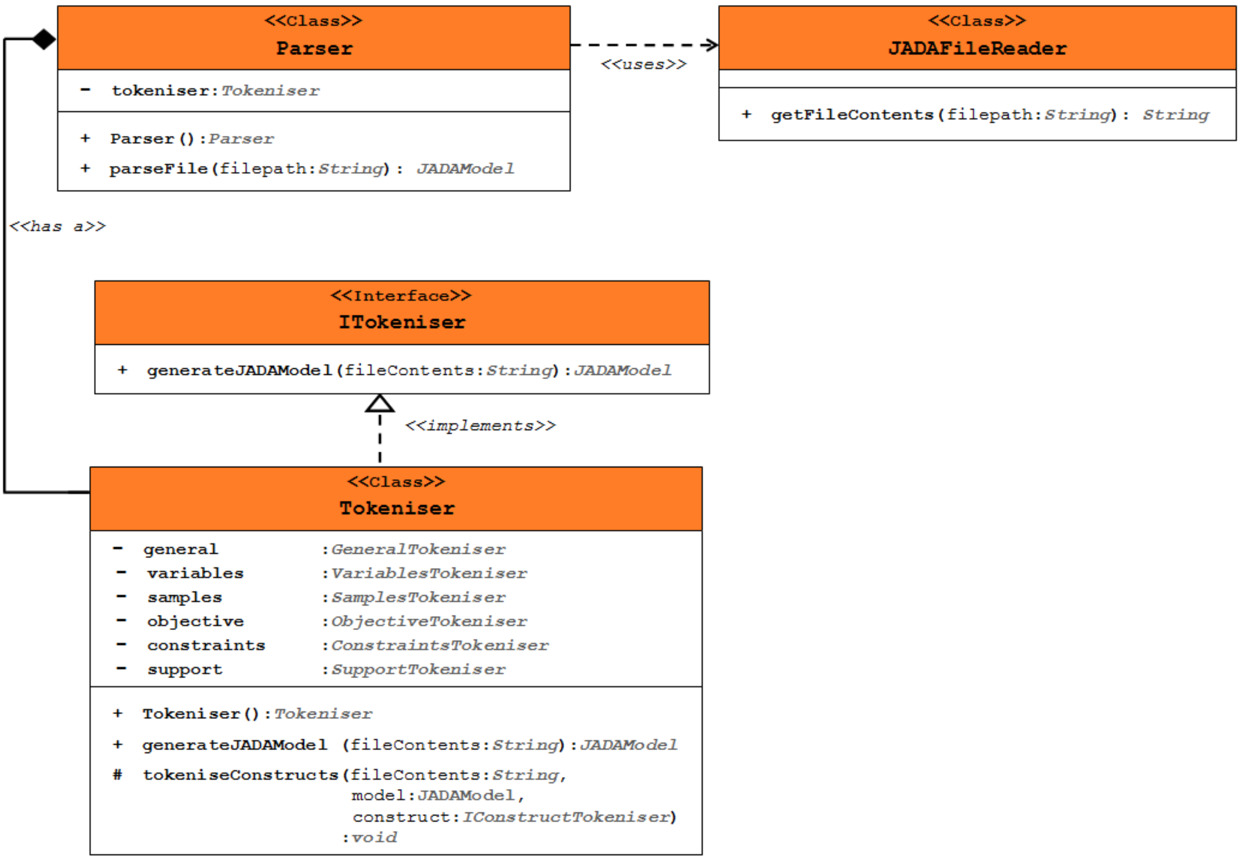


Figure 5.3: UML class diagram showing the structural implementation of the `parser` using MATLAB's regular expressions library.

Denotation	**Regex**	**Description**
`STRING`	"\w+"	An arbitrary string of ASCII characters.
`INT`	[\-]?\d+	A positive or negative integer.
`NAT_INT`	[1-9]\d*	A positive, non-zero integer.
`FLOAT`	[\-]?\d+\.\d*	A positive or negative real number.
`IDENT`	[A-Za-z]+[0-9_]*	An identifier for a variable.
`NUM`	[\-]?\d+\.?\d*	An identifier for a variable.
`TIMES_OP`	(`NUM`*)?`IDENT`	A variable scaled by linear multiplication.
`LINEAR_EXPR`	`TIMES_OP` ((\+|\-) `TIMES_OP`)*	A basic linear expression involving only summations of (scaled) variables with no parentheses.

Table 5.1: Utility Regular Expressions

Table 5.2: IConstructTokeniser implementations for the `General` and `Variables` constructs of JADA's algebraic modelling language.

	Implementation of `IConstructTokeniser`	
Auxiliary Tokeniser	**`getRegex()`**	**`processTokens(tokens,jadaModel)`**

`GeneralTokeniser`	'General
	{
	name(`STRING`);
	stages(`NAT_INT`);
	}'
		Applies a regular expression to the `tokens` to extract the name of the model.
		Invokes `jadaModel.setName(...)` to update the `JADAModel` instance.

`VariablesTokeniser`	'Variables { (decision(`IDENT`, `NAT_INT`);)+ (random(`IDENT`, `NAT_INT`, `FLOAT`, `FLOAT`);)+ }'	Applies regular expressions to the `tokens` to extract the decision and random variables declarations respectively. The parameters of the variables are further extracted. Invokes `jadaModel.addDecisionVariable(...)` and `jadaModel.addRandomVariable(...)` as appropriate to update the `JADAModel` instance with each occurrence.

Table 5.3: IConstructTokeniser implementations for the `Constraints` and `Support` constructs of JADA's algebraic modelling language.

<table>
<tr><th rowspan="2">Auxiliary Tokeniser</th><th colspan="2">Implementation of <code>IConstructTokeniser</code></th></tr>
<tr><th><code>getRegex()</code></th><th><code>processTokens(tokens,jadaModel)</code></th></tr>
<tr><td><code>ConstraintsTokeniser</code></td><td>‘Constraints
{
(LINEAR_EXPR(=|>=|<=) LINEAR_EXPR;)+;
}’</td><td>Applies a regular expression to the <code>tokens</code> to extract equality or inequality arithmetic expressions.

Each expression extracted is further decomposed by applying a series of regular expressions to isolate the identifiers for the variables and their coefficients.

Invokes <code>jadaModel.addRecourseConstraint(...)</code> to persist the meta-information obtained for the recourse constraints.</td></tr>
<tr><td><code>SupportTokeniser</code></td><td>‘Support
{
(LINEAR_EXPR(=|>=|<=) LINEAR_EXPR;)*
}’</td><td>As reiterated in section 4.3, specification of constraints for the <code>Support</code> is optional, hence the use of the quantifier ‘*’ to indicate zero or multiple occurrences of equality or inequality expressions.

The processing logic is similar to that implemented for the recourse constraints. The only difference is that the support constraints are linear equalities and/or inequalities involving only random variables.

After extracting each constraint, the method <code>addSupportConstraint(...)</code> is called on the given instance of a <code>JADAModel</code>.</td></tr>
</table>

Table 5.4: IConstructTokeniser implementations for the `Samples` and `Objectove` constructs of JADA's algebraic modelling language.

<table>
<tr><th rowspan="2">Auxiliary Tokeniser</th><th colspan="2">Implementation of <code>IConstructTokeniser</code></th></tr>
<tr><th><code>getRegex()</code></th><th><code>processTokens(tokens,jadaModel)</code></th></tr>
<tr><td><code>SamplesTokeniser</code></td><td>‘Samples
{
(file(<code>STRING</code>);)+
}’</td><td>Having applied a regular expression to identify the absolute paths to the sample data files, the <code>processTokens(...)</code> function needs to incorporate an additional I/O routine to read in the sample data file. Regular expressions are again used to derive its sample data points and the random variables to which the sample data corresponds to.

The obtained sample data is then passed to the <code>JADAModel</code> instance by invoking <code>jadaModel.addSampleData(...)</code>.</td></tr>
<tr><td><code>ObjectiveTokeniser</code></td><td>‘Objective
{
(min|max)imise
exp
<code>IDENT</code>[<code>LINEAR_EXPR</code>]
(\+ <code>IDENT</code>[<code>LINEAR_EXPR</code>])* ;
}’</td><td>When the regular expression is applied to the tokens, it determines a minimisation (or else maximisation) objective, the statistical measure (expectation or else the variance) and the cost expressions of the decision variables.

The cost expressions are subjected to further processing to identify the random variables and their coefficients.

Each of the aforementioned meta-data are then used to update the JADAModel instance by calling

• <code>setIsMinimisation(...)</code>,
• <code>setIsExpectation(...)</code>, and
• <code>setObjectiveFunction(...)</code>

as accordingly.</td></tr>
</table>

Discussion of Limitations

Although this implementation of the parser is sufficient to specify a basic stochastic optimisation problem, it has many limitations in its applicability. For expressibility, we require a more complex modelling for linear expressions to allow for a flexibility and convenience in specifying the recourse constraints, support constraints and the objective function. This includes use of parentheses to accommodate nested linear expressions.

Consideration of nested parentheses necessitates us to ensure that the parentheses are balanced. If the level of nested parentheses is no more than one level, then the regex can be easily modified to incorporate parentheses. However, for multiple levels of nesting, this is impossible since regular expressions do not support the notion of recursion. In general, regular expressions are not apposite for parsing arbitrarily nested text. Ultimately, a metasyntax like Backus Naur Form (BNF) is required to achieve our goals.

While attempting to implement the parser using regular expressions, we discovered an inherent awkwardness with modifying the JADA syntax to perform augmentations or modifications. This is undesirable, since one of the primary architectural requirements for JADA is extensibility. If the code is difficult to read then it cannot be easily maintained, and if it cannot be easily maintained then it cannot be easily extended which would make the system redundant when considering the long-term goals of the project.

Lastly, although validation has not yet been implemented, we are able to discern that efficient error reporting would be made more difficult with the approach to use regular expressions. The `parseFile(...)` routine defined in the `Parser` class uses the `JADAFileReader` to read a JADA file, which collapses the file contents into a single string with no comments or newlines. As a result, any information that could be used to infer line locations are lost. A tactical solution is to persist a copy of the original file contents. However, when a regular expression cannot not yield any matches, there are no output tokens, thus we cannot not indicate to the user a specific location of the syntactical error. In fact, only a general location relative to the containing language construct can be used in the error message, which is not useful nor convenient for a JADA file containing a large model.

5.3.2 ANTLR v3.0 Implementation

The limitations of the approach, to use regular expressions to implement the parser module, has steered us towards the direction of using a metasyntax to specify the JADA modelling language as a context free grammar. We have investigated several context-free languages like `ANTLR`, `Spirit`, and `YACC++` which not only provide a metasyntax for formally defining programmining or natural languages but also automatically generate the code for the parser engine.

The legacy system has been programmed in C++ and uses the `Spirit` Parser Framework as the parser generator for its standardized input format. It is a relatively good choice, since the expression templates[1] allow the developer to approximate the syntax of Extended Backus Naur Form entirely in C++. However, apart from the syntax being too heavy-weight, the `Spirit`

[1]Expression templates is a metaprogramming technique specific to C++. It permits templates to be used to denote composites of an expression.

parser generator framework is commonly limited to moderately sized parsers, which is owing to the fact that a parser for a full language requires a longer time for compilation. Additionally, we acknowledge that expression templates have many benefits, but the heavy template usage usually corresponds to an increase in code size. Lastly, the lack of static verification for the grammar is a problem. From a developer's perspective, instances of excessive lookahead and usages of left recursion are the two main issues when using a context-free language to design a domain specific language. In the former case, problems of exponential parsing times arise, and in the latter case infinite recursion becomes a possible occurrence. Thus detection and error reporting for occurrences of these two problems are very important for a high-quality implementation[51].

YACC++ is a suitable option, however an unfamiliarity with the language means that there is an associated learning curve, which is further steepened by the lack of an IDE for assistance. For the reasons previously mentioned, the new approach adopts *ANother Tool for Language Recognition* (ANTLR). This choice is justified by

- the desirable provision of static checking for the grammar,
- support for tree construction facilities to build efficient data structures which represented a high-condensed version of the parsed input,
- ease at resolving grammar ambiguities,
- extensive documentation, and
- integrable tools for IDEs, such as the ANTLR plugin for Eclipse, to add internal support for the ANTLR parser generator [52].

ANTLR takes as input the context-free grammar specifying JADA's domain specific language and generates Java code for the parser engine using LL(*)[2] parsing.

Architectural Design

The parser module consists of several components, the bulk of which is implemented in Java with wrapper classes implemented in MATLAB to interface with their Java counterparts.
The sub-modules, as seen in figure (5.5) are enlisted below.

- The `grammar` package defines the parsing rules for the standardised input format.
- The `model` package contains an implementation for the internal representation of the parsed input.
- The `validation` package provides functionality for checking the parsed input for syntactical and semantical errors.
- The `processors` package comprises the auto-generated code resulting from the compilation of the ANTLR grammar source code.
- The `tokens` package persists the tokens exported from compiling the grammar.

[2]An LL parser is a top-down parser for a subset of Backus Naur Form (BNF) grammars. It operates by parsing the input from *left* to right, and builds a *leftmost* derivation of the input string.

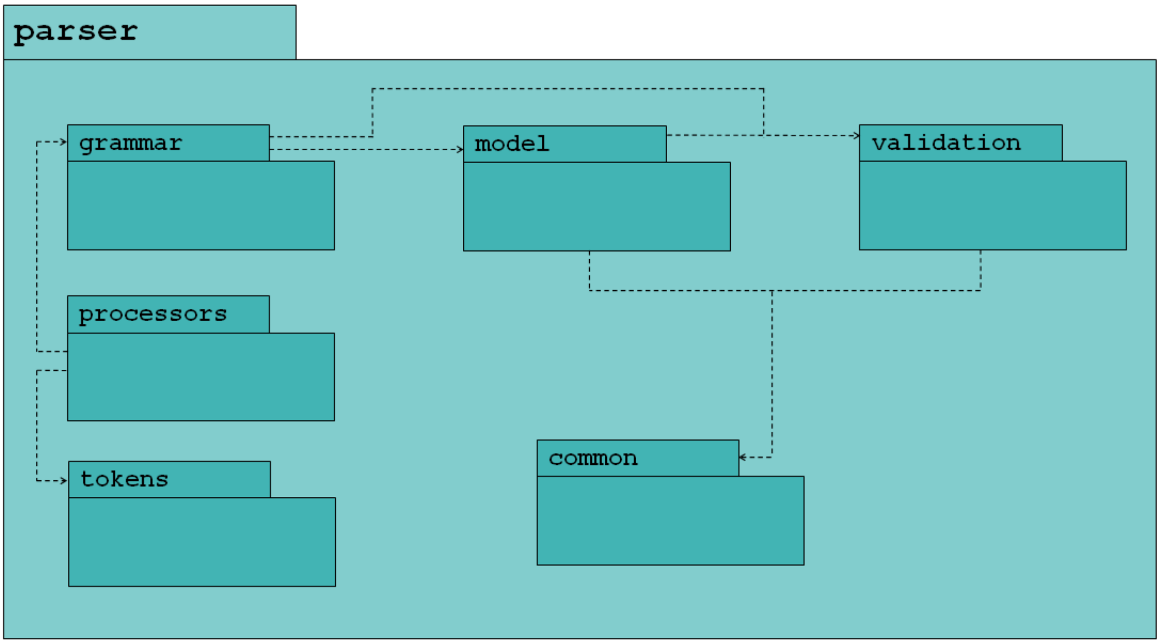


Figure 5.5: UML diagram showing the structure of the parser package, where the dashed arrows indicate the sub-package dependencies.

- The `common` package consists of utility bean classes that can be reused across the different parser sub-packages.

The JADA Grammar

The grammar for JADA's algebraic modelling language is distributed across five different files. Their primary purposes are briefly described below.

`JADALexer.g` is the main lexer source file and provides rules defining literals such as definitions for reserved keywords, alpha-numerical text, numbers (integers and floats), identifiers for variables, escape sequences, strings, whitespace, comments (in-lined and block), symbols and mathematical operators. The syntax diagrams for the lexer rules are given in section B.3.

`JADAParser.g` is the main parser source file. It imports the token vocabulary, as defined by `JADALexer.g`, to specify the parsing rules for recognising a stochastic programming problem declared using the JADA input format. As well as checking for syntax errors, it defines *rewrite rules* which it uses to build an abstract syntax tree. The AST it generates is used to represent the input in an efficiently structured and compact format that can be later traversed. The syntax diagrams for the parser rules are given in section B.4.

`JADATree.g` is the tree parser source file. It provides rules for *walking* over the abstract syntax tree to interpret expressions and populate and `ImmutableJADAModel` instance.

`SampleDataLexer.g` is the lexer source file for the sample data input format. It provides rules similar to those defined in `JADALexer.g`. The additional syntax diagrams specific to lexical analysis of the sample data file format are given in section B.3.

`SampleDataParser.g` is the parser source file that we use to parse the sample data file. It performs some validation routines and, rather than constructing an abstract syntax tree, it interprets the expressions as the input is being parsed. The syntax diagrams specific to the sample data input format are given in section B.4.

Co-ordination of the Auto-generated Parser Classes

Compilation of the grammar files `JADALexer.g`, `JADAParser.g`, and `JADATree.g` initiate an automatic generation of their respective Java classes `JADALexer`, `JADAParser` and `JADATree`. These classes are co-ordinated by the `ParserEngine` class as explained in the code listing B.1 in section B.2. Essentially, the `parseFile(...)` method defined in the `ParserEngine` performs the following steps:

(i) It constructs a file reader to read the JADA file at the given filepath.

(ii) A JADALexer is instantaited with an input stream reader that is built using the file reader.

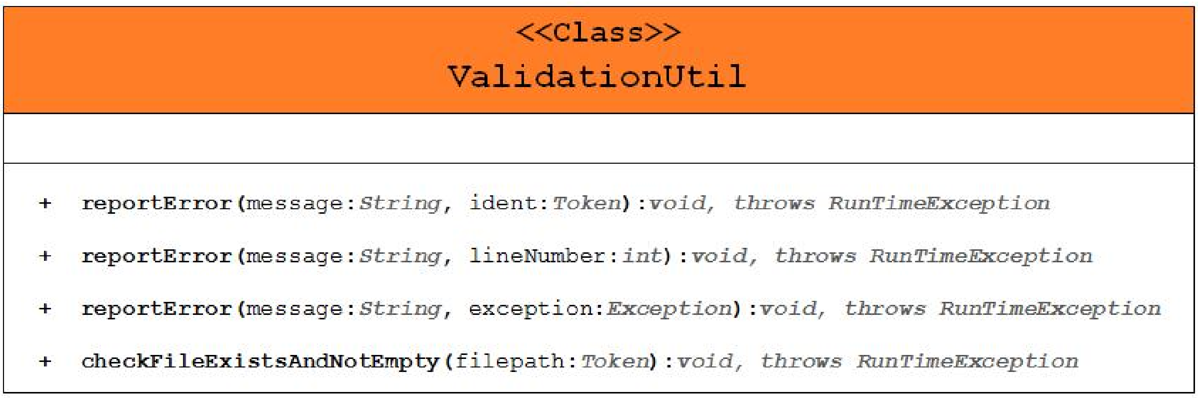
(iii) It subsequently creates a token stream object using the JADALexer instance, which is then passed to the constructor of the JADAParser class.

(iv) Tokenisation of the JADA file is then commenced by invoking the start rule on the JADAParser object. The `ParserEngine` class then uses the return result of the tokenisation to obtain the AST.

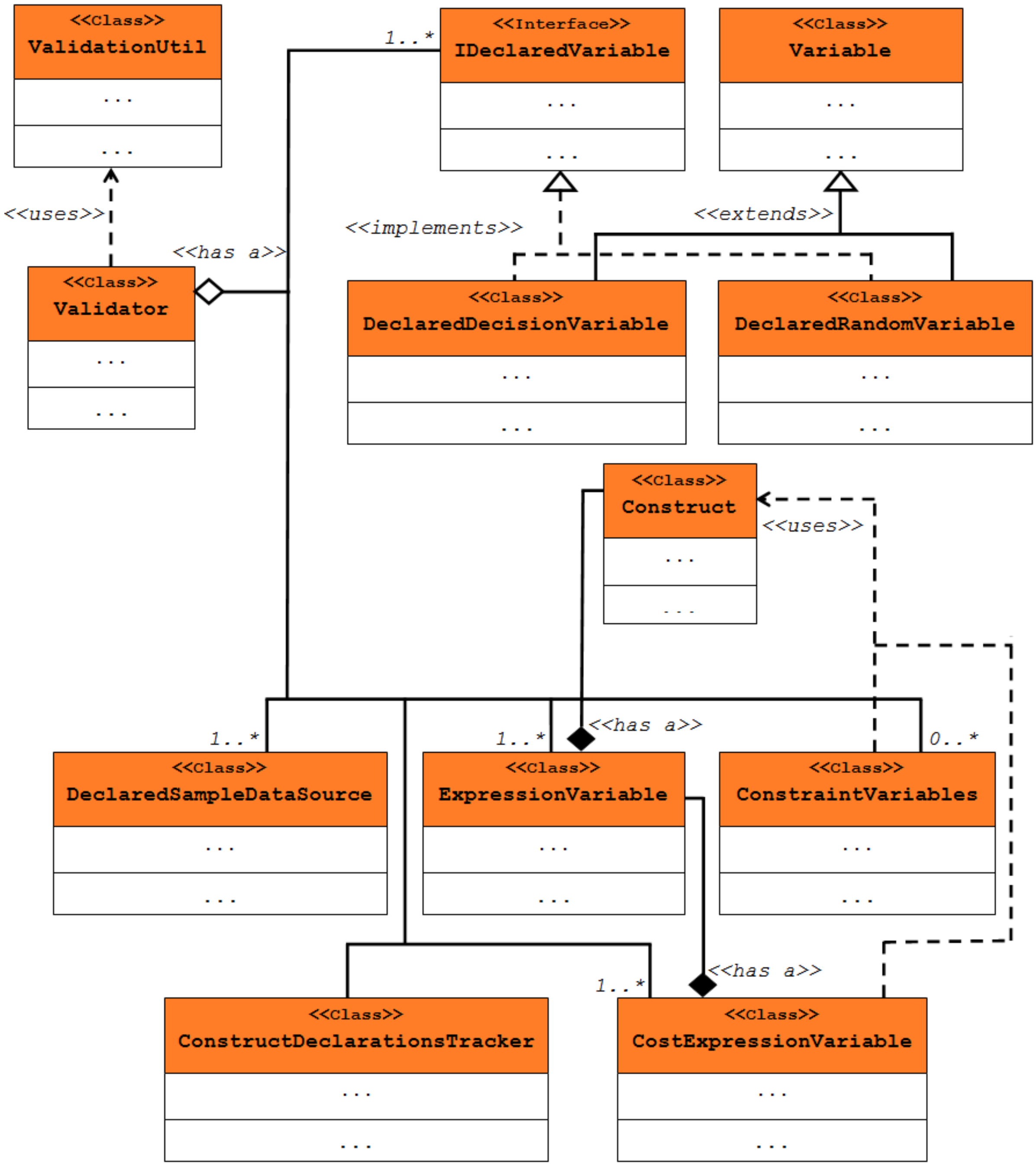
(v) Finally, it instantites a JADATree object using the AST, and traverses the generated tree to populate an `ImmutableJADAModel` object.

Validation

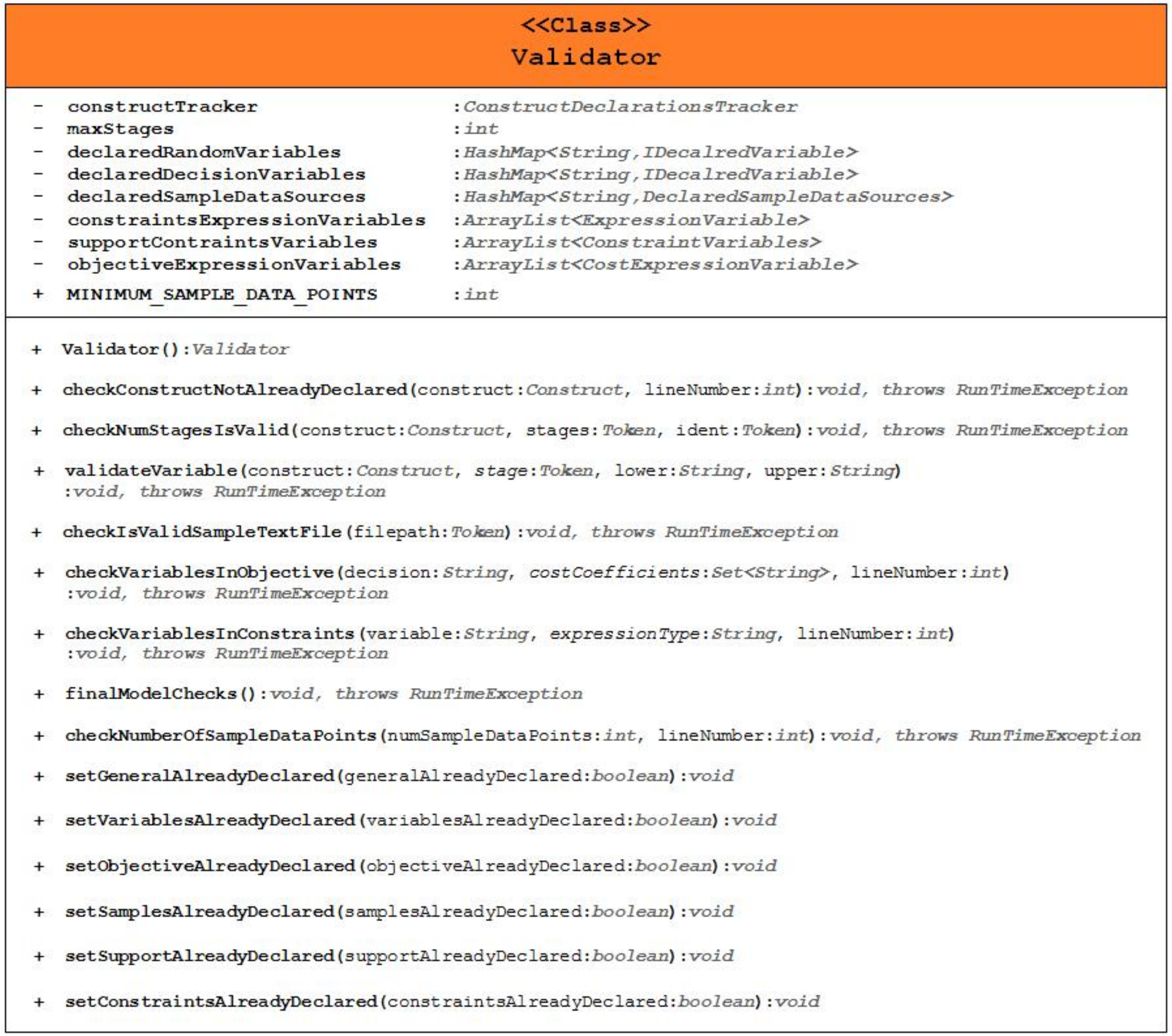
The parser is responsible for the validation of the syntax and semantics of the input format. Most of the model checking is concentrated in JADAParser.g by using fragments of in-lined Java code and the `Validator` class (see fig. 5.6b). Table 5.5 summarises the cases to be checked and the potential error messages that can be propagated.



(a) UML class diagram specifying the utility functions provided by the `ValidationUtil` class.



(b) UML class diagram illustrating the structural composition and dependencies of the `Validator` class.



(c) UML class diagram explaining the functional behaviour of the `Validator` class.

Figure 5.5: UML class diagrams delineating the architecture of, and the relationships between, the classes in the `validation` package.

Table 5.5: Case table for validating the input defined in the JADA format.

Construct	Case	Method	Error Message
All	Each construct must only be defined once.	The special ANTLR directive `@after{...}` allows us to execute code after running the code for the rule. Thus, after invoking each rule for a construct, we send a message to the `Validator` class to note that the construct in question has been defined. Immediately after matching the name of the construct, we use the `Validator` instance to check for multiple declarations of a construct. The `Validator` class utilises the `ConstructDeclarationsTracker` class to assist with determining these duplicate definitions. As an example, the code listing B.2 demonstrates this logic for the 'General' language construct. We explain to the reader that this check has to be performed since we do not restrict the modeller to define the constructs in a particular sequence, otherwise we could have bypassed this validation check. If we had constrained the order in which a JADA file must be specified, then duplicate definitions of constructs would have been captured as syntactical errors since the whole model would not have been matched by the starting parser rule.	E.g. "Multiple definitions of the 'General' construct have been found (line 7)."

Continued on the next page

Table 5.5 ***continued from previous page.***

Construct	Case	Method	Error Message
	All references to variables should be formally declared via the Variables construct.	The `Validator` class maintains a hash table of `IDeclaredVariable` objects indexed by their variable identifier as declared in the `Variables` section. When a variable identifier is matched by any of the parsing rules, the `Validator` is called to determine whether the hash table of declared variables contains this identifier in its key-set.	E.g. "The cost function for the objective refers to the unknown variable 'anUndeclaredVar' (line 14)." E.g. "A recourse constraint refers to the unknown variable 'anUndeclaredVar' (line 19)." E.g. "A support constraint refers to the unknown random variable 'anUndeclaredRandomVar' (line 122)." E.g. "The sample data file 'samples.txt' refers to the unknown random variable 'anUndeclaredRandomVar' (line 5)."
General	The number of stages must be greater than zero.	The extracted text representing the number of stages is converted to an integer to determine whether it is zero-valued.	"The number of stages declared must be greater than zero."

Continued on the next page

Table 5.5 ***continued from previous page.***

Construct	Case	Method	Error Message
	The name given for the model must not be too long.	The length of the string representing the model's name is calculated and ascertained to be between the range [1, 120]. A maximum length is imposed since we use the name of the model for name mangling any generated files. Since, most file systems have a maximum filename length of 256 characters, we allocate to ourselves just over 50% of these characters for our own purposes.	"The length of the model's name must be between 1 and 120 characters."
Variables	The identifiers of declared variables must not conflict with any of the reserved keywords.	A static list of reserved keywords is maintained. As variable identifiers are parsed, the `Validator` verifies whether the variable identifier conflicts with any of the reserved keywords as defined in the lexer source files.	E.g. "the variable identifier 'decision' is illegal as it conflicts with the reserved keyword 'decision' (line 64)."
	Variable declarations must be unique with respect to their identifiers.	When a decision or random variable identifier is matched, the `Validator` checks that the maintained hash table of `IDeclaredVariable` objects does not already contain a variable with the same identifier.	E.g. "Duplicate declarations of the decision variable 'aDuplicateVar' were found (line 43)."

Continued on the next page

Table 5.5 ***continued from previous page.***

Construct	Case	Method	Error Message
	The stage attribute of a variable declaration must be within the range [1, *maxStages*], where *maxStages* is the declared number of stages.	Due to the fact that we do not stipulate an ordering for how the constructs should be specified, we might be able to validate this requirement immediately or postpone the check. In the former case, the `General` construct must have been defined earlier, and thus the maximum number of stages has been declared. Consequently, we can corroborate that the stage to which the decision or random variable belongs to is indeed within the mandatory range. However, if say the `Variables` information was the first section to be delineated then we need to persist the stage of the variable in the hash table of declared variables. Thus, when the number of stages is known, we are obligated to iterate through the collection of declared variables to perform the required validation.	E.g. "The 'stage' attribute (second parameter) for a random variable must be greater than zero (line 23)." E.g. "The declared stage (second parameter) for which the decision variable 'x' corresponds to must be from the set {1, ..., *maxStages*} (line 56)."
	The bounds given for declared random variables must be numerically consistent.	When a random variable declaration is extracted, the lower and upper bound parameters are converted to the `double` primitive type. The `Validator` then checks that the lower bound value is indeed smaller than upper bound value.	E.g. "The lower bound (third parameter) for random variable 'y' must be smaller than the upper bound value (fourth parameter) (line 43)."
Support	All the variables referred to in the support constraints must be random variables.	The `Validator` uses the matched identifiers in the equalities and/or inequalities to check whether they are elements of the key-set belonging to the hash table of random variables.	E.g. "A support constraint refers to the variable 'x' which has not been declared as a random variable."

Continued on the next page

Table 5.5 ***continued from previous page.***

Construct	Case	Method	Error Message
Samples	All declarations of the sample files must be unique.	The `Validator` class possesses a hash table of `DeclaredSampleDataSource` objects that are indexed by their corresponding filepaths to the sample data file. When the absolute filepath to the sample data file is obtained, the `Validator` class determines whether the key-set for this hash table contains the parsed filepath.	E.g. "A duplicate declaration of the sample data file 'C:/samples.txt' has been found (line 99).
	All sample data file declarations must reference existent files.	The `Validator` verifies that the filepath given points to an existent file by invoking the `java.io.File.exists()` method.	"The sample data file at the specified location 'C:/samples.txt' does not exist (line 77)."
	All sample data file declarations must reference a non-empty file.	The `Validator` verifies that the file at the supplied filepath is not empty by checking the file length, in bytes, is non-zero.	"The sample data file at the specified location 'C:/samples.txt' is empty (line 44)."
	All sample data file declarations must reference a file with a valid extension type.	Currently, JADA only considers sample data given as text files. As a result, the `Validator` class verifies that the specified filepath has a `.txt` extension.	"The sample data file at the specified location 'C:/samples.doc' does not have a '.txt' extension type (line 11)."

Continued on the next page

Table 5.5 ***continued from previous page.***

Construct	Case	Method	Error Message
	All variables referred to in the sample data file must have been formally declared as random variables.	This check is initiated by the `SampleDataParser`. The `ISampleDataValidator`, which is passed to the constructor of the `SampleDataParser`, checks that the variables to be sampled are members of a hash table maintained for declared random variables and/or not members of a hash table for declared decision variables. The `ISampleDataValidator` is implemented by the `JADAModel` class, which allows the `SampleDataParser` indirect access to the contents of the parsed JADA file.	E.g. "A reference to a random variable 'y' that has not been declared has been found for a sample data source declaration (line 7)." E.g. "A sample data source has been specified for the parameter 'y', which is not a random variable (line 10)."
	A maximum of one sample data source can be defined for each random variable.	The `SampleDataParser` indirectly delegates this check to the `JADAModel`, as it implements the `ISampleDataValidator` interface. The logic used to perform this validation involves retrieving the metadata of the random variable in question and determining whether the boolean flag indicating whether the random variable has a sample data has been set.	E.g. "The random variable 'y' has been associated with multiple sample data sources (line 15)."
	The random variables must have a number of sample data points equal to the specified sample size in the sample data file.	The `SampleDataParser` keeps a hash table of the sample data points as a collection of real-valued lists, which is indexed by the identifier of the random variable. Having extracted the sample size from the header information, the `SampleDataParser` is able to verify that the number of sample data points for each variable is consistent with the explicitly stated sample size.	E.g. "The number of sample data points in the sample data file 'C:/samples.txt' for the random variable 'y' is 999, which is not equal to the declared sample size of 1000 (line 23)."

Continued on the next page

Table 5.5 ***continued from previous page.***

Construct	Case	Method	Error Message
Objective	The objective function must be linear in the random variables and the cost coefficients for the decision variables must only refer to declared random variables or real numbers.	The cost coefficient of a decision variable is represented as a linear expression. When the variables in this linear expression are isolated by `JADAParser`, the `Validator` is called to check that all the variables have actually been declared as random variables. This is achieved by using the identifier of the variable to check its membership in the hash table of declared random variables.	"The objective function refers to the decision variable 'x' as if it were a random variable. Decision variables are not allowed to be used in the cost coefficient of a decision variable (line 100)."
	The objective function must abide by the non-anticipativity property for decision variables (see section 2.3.1).	The `Validator` ascertains that any random variable referred to in a decision variable's cost expression is known before or at the stage the decision variable is known. To do this, the `Validator` uses the metadata stored for the `IDeclaredVariable` objects to obtain the corresponding stages of the decision and random variables respectively, which are then compared.	"The decision variable 'x' in the objective function has a cost coefficient that depends on the random variable 'y', which is not known by the time 'x' is known (line 52)."
	In the objective function, random variables cannot appear outside the square parentheses.	The `Validator` checks whether a random variable has been used as decision variable. To perform this check, the JADAParser passes the identifier of the variable that prefixes the left square parenthesis. The `Validator` instance is then able to use the hash table of declared random variables to determine if the identifier it receieves belongs to a random variable instead of a decision variable.	"The objective function refers to the random variable 'y' as if it were a decision variable. Random variables can only be used in the cost coefficient of a decision variable (line 8)."

Continued on the next page

Table 5.5 ***continued from previous page.***

Construct	**Case**	**Method**	**Error Message**
Constraints Support Objective	All arithmetic expressions must be linear in the random variables.	JADAParser defines parser rules for arithmetic expressions, and uses two hash sets to accumulate variables in the left-hand-side and right-hand-side of the multiplication and division operations. When processing the parser rule for these two arithmetic operations, in-lined Java code is used to check that the hash sets representing the variables referenced in the operands operands do not *both* contain elements.	E.g. "An arithmetic expression in the Support is not linearly dependent on the random variables (line 29)."

5.3.3 JADA Model

As explained in section 5.3.2, the `JADATree` class traverses the abstract syntax tree generated by the `JADAParser` class to assist with populating an instantiation of a `JADAModel`. While the `JADATree` extracts the physical data represented in the abstract syntax tree, the `JADAModel` is actually responsible for pre-processing the data it receives to store and derive the necessary metadata. Some of the tasks performed include:

- stage-wise aggregation of the variables, recourse constraints and support constraints,
- generating additional support constraints implied by a random variable's lower and upper bound values,
- standardising the recourse constraints, support constraints and the objective functions,
- factorising the objective function,
- extracting and validating the sample data, and
- enumerating the positions of the decision and random variables for the matrix components of the linear program.

The `JADAModel` class has been designed with the intention of compactly representing a highly condensed version of the input, which can be efficiently queried by the classes in the `generator`, `approximator` and `renderer` packages. The `JADAModel` class exposes particular methods to its client packages for it be efficiently and conveniently post-processed. These functionalities include:

- retrieving variables and constraints corresponding to a particular stage,
- sorting variables by their corresponding time period,
- obtaining the objective function as a formal mathematical expression,
- determining the vector position of the variables in the decision and random vector, and
- generating a string representation of the model, which includes the derived parameters.

To restrict modifications to the state of the `JADAModel` after being populated, we mandate that the `parseFile(...)` method defined in `ParserEngine` returns an `ImmutableJADAModel`. The `ImmutableJADAModel` class implements the *decorator* design pattern to wrap a `JADAModel` object and only expose accessor methods.

5.3.4 Invocation from MATLAB

MATLAB provides the capability for bringing Java classes and methods into its workspace by generating class definitions in `.java` files and using a Java compiler to produce the .class files from them[37]. We currently automate this procedure with a XML-based build script that is executed with the Apache Ant ('Another Neat Tool') open-source software tool.

The Dynamic Class-path

MATLAB imports Java class definitions from files that are present on the Java *class-path*, which is a list of files and directories that MATLAB uses to find class definitions[37]. For our own purposes, we are required to update the dynamic class-path, which is loadable at all times during a MATLAB software session using the `javaclasspath` function. It is also modifiable using the `javaaddpath` and `javarmpath` functions, and refreshable using the command `clear('java')` without needing to restart the MATLAB session.

Making the Java-based Parser Implementation Available in the MATLAB Workspace

We distribute the library of classes and functions as an aggregated or archived format, known as a *Java Archive* (JAR) file, by also using the aforementioned Appache ANT build script. We then make the multiple class definitions, which had been compressed into the `JAR` file, available for use by declaring the absolute file-path to the `JAR` file and placing it on the dynamic class-path. Thus, to initiate the parsing process implemented in Java, we can invoke the public methods in the `JAR` file.

5.4 Linear Program Generator

In section 3.4.3, we presented to the reader the decision rule approximations developed as a result of the research paper *Primal and Dual Linear Decision Rules in Stochastic and Robust Optimization*[2]. By using the decision rule approximations eqs. (Cons-MSP$_{fixed}$) and (Prog-MSP$_{fixed}$), we can convert a stochastic programming problem given in the JADA standard input format, as specified in section 4.3, to instances of a linear program that can be communicated to external solvers.

As explained in section 3.5, the conversion will be an approximation since stochastic programming models are traditionally formulated as optimisation problems with infinite dimensions, which are inherently computationally intractable. Thus, the linear programs eqs. (Cons-MSP$_{fixed}$) and (Prog-MSP$_{fixed}$) will provide an overestimate and underestimate of the actual optimal solution. In this section, we only outline our implementation of the generation of the matrix components used by these linear programs, and in section 5.5 we discuss the design of the conservative and progressive approximation routines that were previously referred to.

5.4.1 Pattern of Interaction

The `LPGenerator` class defines a single public method `generate(...)`, which co-ordinates the generation of the recourse constraints matrices, the costs matrices, the support matrices and the second-order and conditional moments matrices. The `generate(...)` method takes as input an instance of the `ImmutableJADAModel` class representing the contents of the parsed JADA file. It then delegates the generation of the individual components to other matrix generators as shown in the sequence diagram of figure (5.6).

The `LPGenerator` compacts the generated matrices into an `LPModel` object, which we then compose with the given `ImmutableJADAModel` instance to instantiate an `OptimisationModel` object. The `LPGenerator` returns from this method with two initialised `LPApproximatorModel` objects, which encapsulate the decision rule matrices and a copy of the `OptimisationModel` instance, to represent the models for `ConservativeApproximator` and `ProgressiveApproximator` respectively.

5.4.2 Algorithms

We use pseudo-code notation to explicate our development of the module responsible for automating the generation of the matrix components appertain to eqs. (Cons-MSP$_{fixed}$) and (Prog-MSP$_{fixed}$).

Generation of the Decision Matrices

The MATLAB class `DecisionMatricesGenerator` is responsible for the generation of the decision rule matrices. The generation logic is algorithmically explained using the pseudo-code of 5.1 and 5.2. We refer the reader to table 5.6 for a decipherment of the notation used.

Table 5.6: Notation for algorithms 5.1 and 5.2, which explain the generation of the decision rule matrix components of the linear programs given by eqs. (Cons-MSP$_{fixed}$) and (Prog-MSP$_{fixed}$).

$T \in \mathbb{N}$	$\stackrel{\text{def}}{=}$	is the maximum number of stages in the multi-stage stochastic programming problem.
$n_t \in \mathbb{Z}^+$	$\stackrel{\text{def}}{=}$	is a dimension that denotes the number of decisions to be made at time t.
$k^t \in \mathbb{N}$	$\stackrel{\text{def}}{=}$	is a dimension that denotes the cumulative number of observed outcomes at time t.
$m_t \in \mathbb{Z}^+$	$\stackrel{\text{def}}{=}$	is a dimension that denotes the number of recourse conditions that constrain the decisions at time t.
$\xi \in \mathbb{R}_T$	$\stackrel{\text{def}}{=}$	is the vector of uncertain parameters $[\xi_1, \xi_2, \cdots, \xi_T]$.
$\Upsilon : string \mapsto$ `cell array`	$\stackrel{\text{def}}{=}$	is a hash table that maps auto-generated variable identifiers for elements of the decision rule matrices X_t, Λ_t and S_t to a cell-array containing a decision variable and the random variable that it is affinely dependent on.
$\mathcal{Y}$	$\stackrel{\text{def}}{=}$	denotes the data type `sdpvar` from the YALMIP convex programming framework, which allows us to define symbolic decision variables.
$\mathcal{V}$	$\stackrel{\text{def}}{=}$	denotes the data type `VariableTerm` from our proprietary mathematical expressions library, which allows us to represent arithmetic expressions as objects.
$X_t \in \mathcal{Y}^{n_t \times k^t}$	$\stackrel{\text{def}}{=}$	is a 2-D array that represents the originally unknown linear decision rule matrix for the time period t.

$X_{symbolic,t} \in \mathcal{V}^{n_t \times k^t}$ $\stackrel{\text{def}}{=}$ is a 2-D array of type `VariableTerm` that symbolically represents the originally unknown linear decision rule matrix for the time period t.

$\Lambda_t \in \mathcal{Y}^{n_t \times k^t}$ $\stackrel{\text{def}}{=}$ is a 2-D array of `sdpvar` variables that represents the unknown linear slack decision rule matrix at stage t for the conservative LP eq. (Cons-MSP$_{fixed}$).

$\Lambda_{symbolic,t} \in \mathcal{V}^{n_t \times k^t}$ $\stackrel{\text{def}}{=}$ is a 2-D array of type `VariableTerm` that symbolically represents the unknown linear slack decision rule matrix at stage t for the conservative LP eq. (Cons-MSP$_{fixed}$).

$S_t \in \mathcal{Y}^{n_t \times k^t}$ $\stackrel{\text{def}}{=}$ is a 2-D array of `sdpvar` variables that represents the unknown linear slack decision rule matrix at stage t for the progressive LP eq. (Cons-MSP$_{fixed}$).

$S_{symbolic,t} \in \mathcal{V}^{n_t \times k^t}$ $\stackrel{\text{def}}{=}$ is a 2-D array of type `VariableTerm` that symbolically represents the unknown linear slack decision rule matrix at stage t for the progressive LP eq. (Prog-MSP$_{fixed}$).

Π $\stackrel{\text{def}}{=}$ is an instance of the Java class `ImmutableJADAModel` which contains a compact representation of the parsed input file.

Algorithm 5.1 `generate`(Π)

$l \leftarrow 2 + |\Pi[\text{‘support constraints’}]\ |$
$\Upsilon \leftarrow \emptyset$
$X \equiv \{\texttt{cell}(X_t\,, X_{symbolic,t})\}_{t=1}^{T} \wedge X \leftarrow \emptyset$
$\Lambda\ \equiv \{\texttt{cell}(\Lambda_t\,, \Lambda_{symbolic,t})\}_{t=1}^{T} \wedge \Lambda \leftarrow \emptyset$
$S\ \equiv \{\texttt{cell}(S_t\,, S_{symbolic,t})\ \}_{t=1}^{T} \wedge S \leftarrow \emptyset$
$T \leftarrow \Pi[\text{‘maximum stages’}]$
for $t \in 1, 2, \cdots, T$ **do**
    $n_t \leftarrow \Pi[\text{‘decisions aggregator’}, t]$
    $k^t \leftarrow \Pi[\text{‘uncertainty aggregator’}, t]$
    $m_t \leftarrow \Pi[\text{‘recourse constraints aggregator’}, t]$
    $[X_t, X_{symbolic,t}, \Upsilon] \leftarrow$ `createSDPVARMatrix`(Π, n_t, k^t, t, false, Υ, ‘x’)
    $X[t] \leftarrow [X_t, X_{symbolic,t}]$
    $[\Lambda_t, \Lambda_{symbolic,t}, \Upsilon] \leftarrow$ `createSDPVARMatrix`(Π, m_t, l, t, true, Υ, ‘lambda’)
    $\Lambda[t] \leftarrow [\Lambda_t, \Lambda_{symbolic,t}]$
    $[S_t, S_{symbolic,t}, \Upsilon_t] \leftarrow$ `createSDPVARMatrix`(Π, m_t, k^t, t, true, Υ, ‘s’)
    $S[t] \leftarrow [S_t, S_{symbolic,t}]$
end for
conservativeDecisionRules $\leftarrow$ *new* `DecisionRules`(X, Λ, Υ, ‘lambda’)
progressiveDecisionRules $\leftarrow$ *new* `DecisionRules`(X, Λ, Υ, ‘s’)
return [*conservativeDecisionRules*, *progressiveDecisionRules*]

The method `createSDPVARMatrix(...)` is a private static method defined in the MATLAB class `DecisionMatricesGenerator`. It is invoked to assist with generating the 2-D arrays of `sdpvar` variables and `VariableTerm` objects, and the incremental construction of the decision rule mappings (see algorithm 5.2).

Algorithm 5.2 `createSDPVARMatrix`(Π, $\#rows$, $\#cols$, t, isSlack, Υ, decision$_{symbol}$)

if $\neg$isSlack **then**
    $D_t \leftarrow \Pi$[‘decision variables’, t, ‘sorted’]
    $R_t \leftarrow \Pi$[‘random variables’, t, ‘sorted’]
end if
Matrix$_{symbolic}$ $\leftarrow$ `VariableTerm.empty`($\#rows$, 0)
if $\#rows = 0 \vee \#cols = 0$ **then**
    Matrix$_{sdpvar}$ $\leftarrow \mathbf{0} \in \mathbb{R}^{\#\text{rows} \times \#\text{cols}}$
    return
end if
Matrix$_{sdpvar}$ $\leftarrow$ *new* `sdpvar`(#rows, #cols)
for $i \in 1, 2, \cdots,$ #rows **do**
    if $\neg$isSlack **then**
        decision$_{ID}$ $\leftarrow D_t$[‘identifier’, i]
    end if
    for $j \in 1, 2, \cdots,$ #cols **do**
        decision$_{ID}$ $\leftarrow$ `getDecisionVariableID`(t, i, j, decision$_{symbol}$)
        if $\neg$isSlack **then**
            if $j = 1$ **then**
                random$_{ID}$ $\leftarrow$ *empty string*
            else
                random$_{ID}$ $\leftarrow R_t$[‘identifier’, $j - 1$]
            end if
            Υ[decision$_{ID}$] $\leftarrow$ `cell`(decision$_{ID}$, random$_{ID}$)
        end if
        Matrix$_{symbolic}$(i, j) $\leftarrow$ `VariableTerm`(decision$_{ID}$, 1.0)
    end for
end for
return [Matrix$_{sdpvar}$, Matrix$_{symbolic}$, Υ]

The creation of the decision rule mappings, denoted by Υ is a tactical solution for maintaining the logical mappings of the `sdpvar` variables with their associated affine decision functions $x(\xi)$. Originally, we relied on an implementation of the `sdisplay()` function provided by the `sdpvar` class. However when it came to rendering the individual decision variables with the names that were dynamically assigned to them, we encountered several problems due to issues with *variable scope* in MATLAB. This is discussed further in section 5.6.1.

Generation of the Costs Matrices

We abstract the generation of the costs matrices into the class `CostsMatricesGenerator`. The implemented `generate(...)` method takes as input an instance of an `ImmutableJADAModel` and outputs a cell-array of real-valued matrices corresponding to the decision costs for each time period. The implementation of this method is described by algorithm 5.3. In addition to the notation given in table 5.6, we also provide the following notation in table 5.7 for algorithm 5.3.

Table 5.7: Notation for algorithms 5.3, which explain the generation of the decision costs matrices for use by the linear programs as defined in eqs. (Cons-MSP$_{fixed}$) and (Prog-MSP$_{fixed}$).

$C_t \in \mathbb{R}^{n_t \times k^t}$	$\stackrel{\text{def}}{=}$	is the matrix of decision costs pertaining to the time period t.
$\mathcal{D}$	$\stackrel{\text{def}}{=}$	denotes the custom Java class `IVariable` used to treat our representation of the decision and random variables in a uniform manner.
$D \in \mathcal{D}^{n \times 1}$	$\stackrel{\text{def}}{=}$	is a list of decision variable objects of type `IVariable`.
$R_{map} : string \mapsto \mathcal{D}$	$\stackrel{\text{def}}{=}$	is a hash table that maps the identifiers of random variables to their metadata, which is encapsulated in a `IVariable` instance.

Algorithm 5.3 `generate`(Π)

$T \leftarrow \Pi$[‘maximum stages’]
$D \leftarrow \Pi$[‘decision variables’, ‘sorted’]
$R_{map} \leftarrow \Pi$[‘random variables’, ‘as map’]
$C \leftarrow$ `cell`$(1, T)$
for $t \in 1, 2, \cdots, T$ **do**
  $n_t \leftarrow \Pi$[‘decision aggregator’, t]
  $k^t \leftarrow \Pi$[‘uncertainty aggregator’, t]
  $m_t \leftarrow \Pi$[‘recourse constraints aggregator’, t]
  $C_t \leftarrow \mathbf{0} \in \mathbb{R}^{n_t \times k^t}$
  for $i \in 1, 2, \cdots, |D|$ **do**
    decision $\leftarrow D[i]$
    decision$_{stage} \leftarrow$ decision[‘stage known at’]
    if decision$_{stage} = t$ **then**
      $C_t \leftarrow$ `processDecisionCost`(decision[‘position’], C_t, decision, R_{map})
    end if
  end for
  $C[t] \leftarrow$ `cell`(C_t)
end for
return C

The method `processDecisionCost`(...) (see algorithm 5.4) is an auxiliary function defined in the class `CostsMatricesGenerator`. It generates a row of the decision cost matrix at time period t, such that the row vector created is representative of the decision cost coefficients as linear combinations of the uncertain elements. The method is parameterised by

- an integer that represents the row-wise vector position of the decision variable,
- an intermediate cost matrix,
- an `IVariable`, which compacts the meta-information about the decision variable currently being processed, and
- a hash table that maps the identifiers of the random variables to `IVariable` objects.

For each decision variable, the method `processDecisionCost`(...) obtains the variable's cost coefficient. The cost coefficient can be a constant term or a linear expression in the random variables. In the former case, the function `processDecisionCost`(...) associates this cost coefficient with the dummy outcome ξ_1, and uses this constant value to update the entry of the cost matrix C_t. The entry is given by the vector position of the decision variable and the vector position of the random variable[3]. The latter case is more complex. It necessitates iterating through the random variables in the linear expression. The processing of each random variable is then handled by the method `processRandomVariableTerm`(...) (see algorithm 5.5), which also identifies two cases as explained below.

(a) If the random variable is a constant term, then the function determines that it belongs to the first column of the cost matrix.

(b) If the random variable is indeed a variable term, then function obtains the identifier of the variable and its real-valued coefficient. Using the found identifier, it indexes the map of random variables to obtain the associated `IVariable` object, which it can then use to retrieve the vector position of the random variable. This vector position determines the column of the cost matrix that the random variable belongs to.

Finally, the method `processRandomVariableTerm`(...) completes its processing by updating the cost matrix with the constant term or else with the coefficient of the random variable.

[3]In this case, the vector position of the random variable is 1.

Algorithm 5.4 `processDecisionCost`(decision$_{position}$, C_t, decision, R_{map})

$c_\xi \leftarrow$ *decision*[‘cost coefficient’]
if c_ξ *is a constant* **then**
  C_t(decision$_{position}$, 1) $\leftarrow c_\xi$[‘value’]
else if c_ξ *is a linear expression* **then**
  for each term $\in c_\xi$[‘terms’] **do**
    $C_t \leftarrow$ `processRandomVariableTerm`(decision$_{position}$, C_t, term, R_{map})
  end for each
else c_ξ *is a linear expression* **then**
  $C_t \leftarrow$ `processRandomVariableTerm`(decision$_{position}$, C_t, c_ξ, R_{map})
end if
return C_t

Algorithm 5.5 `processRandomVariableTerm`(decision$_{position}$, C_t, random$_{\mathcal{D}-term}$, R_{map})

if random$_{\mathcal{D}-term}$ *is a constant* **then**
  random$_{position} \leftarrow 1$
  $c_\xi \leftarrow$ random$_{\mathcal{D}-term}$
else
  random $\leftarrow R_{map}$[random$_{\mathcal{D}-term}$[‘identifier’]]
  random$_{position} \leftarrow$ random[‘position’]
  $c_\xi \leftarrow$ random$_{\mathcal{D}-term}$[‘coefficient’]
end if
C_t(decision$_{position}$, random$_{position}$) $\leftarrow c_\xi$
return C_t

Generation of the Support Matrices

The class `SupportMatricesGenerator` provides its own implementation of the method `generate(...)`, which takes an `ImmutableJADAModel` object, and generates a `SupportMatrices` object. This output object contains the matrix components for the support constraints as defined by the matrix inequality of eq. (2.3.3.4). We refer the reader to the notation itemised in tables 5.6 and 5.8 in order to understand our explanation of the generation logic given by algorithms 5.6, 5.7, and 5.8.

Table 5.8: Notation for algorithms 5.6-5.8, which explain the generation of the support matrices for use by the linear programs as defined in eqs. (Cons-MSP$_{fixed}$) and (Prog-MSP$_{fixed}$).

$\hat{W} \in \mathbb{R}^{(l-2)\times k}$	$\stackrel{\text{def}}{=}$	is the sub-matrix $\hat{W}$ of the support coefficient matrix $W = [e_1, -e_1, \hat{W}]^\mathsf{T} \in \mathbb{R}^{l\times k}$ as defined by eq. (2.3.3.4)
$e_1 \in \mathbb{R}^k$	$\stackrel{\text{def}}{=}$	is the basis vector $[1, 0, \cdots, 0] \in \mathbb{R}^k$

$h \in \mathbb{R}^{l \times 1}$	$\stackrel{\text{def}}{=}$	is the vector $[1, -1, 0, \cdots, 0]$, which denotes the right-hand-side of the inequality defined by eq. (2.3.3.4)
$\mathcal{C}$	$\stackrel{\text{def}}{=}$	denotes the custom Java class `IConstraint` used to represent the support constraints as objects.
$\text{constraints}_{support} \in \mathcal{C}^{(l-2) \times 1}$	$\stackrel{\text{def}}{=}$	is the list of `IConstraint` objects representing the inequalities $\hat{W}\xi \leq \mathbf{0}$ in an object-oriented manner. Each constraint has been standardised such that the right-hand-side of the less-than-or-equal-to inequality is 0
$k \in \mathbb{N}$	$\stackrel{\text{def}}{=}$	is a dimension that denotes the cumulative number of observed outcomes by the last stage T, such that $k = k^T$.
$\mathcal{L}$	$\stackrel{\text{def}}{=}$	denotes the custom Java class `ILinearTerm` used to generically represent linear arithmetic expressions as objects.

Algorithm 5.6 `generate`(Π)

$\text{constraints}_{support} \leftarrow \Pi$[‘support constraints’, ‘standardised’]
$l \leftarrow 2 + |\ \text{constraints}_{support}|$
$R_{map} \leftarrow \Pi$[‘random variables’, ‘as map’]
$T \leftarrow \Pi$[‘maximum stages’]
$k \leftarrow \Pi$[‘uncertainty aggregator’, T]
$W \leftarrow \mathbf{0} \in \mathbb{R}^{l \times k}$
$W(1,1) \leftarrow 1$
$W(2,1) \leftarrow -1$
for $i \in 3, 4, \cdots$, l **do**
    constraint $\leftarrow \text{constraints}_{support}[i$ - 2]
    $\text{expr}_{LHS\mathcal{C}-term} \leftarrow$ constraint[‘LHS term’]
    $W \leftarrow$ `processRandomVariableConstraint`($\text{expr}_{LHS\,\mathcal{C}-term}$, R_{map}, W, i)
end for
$h \leftarrow \mathbf{0} \in \mathbb{R}^{l \times 1}$
h(1,1) $\leftarrow 1$
h(2,1) $\leftarrow -1$
supportMatrices $\leftarrow$ *new* SupportMatrices(W, h)
return supportMatrices

As explained, the `SupportMatrices` object finally outputted from the `generate(...)` method contains two matrices that represent the matrix components in eq. (2.3.3.4). The generation of these matrices involves a systematic processing of the parsed support constraints, which are retrievable as a list of `IConstraint` objects from the `ImmutableJADAModel` instance.

For clarity, we compute the row-dimension l of the support matrix W as the size of this list of `IConstraint` objects plus the additional support constraints for the degenerate observed outcome at the first stage. This is compactly represented as $-1 \leq \xi_1 \leq 1$.

The calcuation of the vector h is straightforward, as shown in lines 14-16 of algorithm 5.6. We simply initialise the matrix to all zeroes. Subsequently, we then set the entries at positions (1,1) and (2,1) to -1 and 1 to respectively denote the constant parts of the respective inequalities $-1 \leq \xi_1$ and $\xi_1 \leq 1$.

On the other hand, the derivation of the coefficient matrix W involves considerably more computational processing. We are required to iterate through all the support constraints to incrementally build the matrix W, one row vector at a time, using the utility method `processRandomVariableConstraint(...)` as provided by the class `LPGeneratorUtil`. We point out to the reader that the support constraints have been standardised such that the right-hand-side expression is zero-valued and all the constraints are less-than-or-equal-to inequalities.

Algorithm 5.7 `processRandomVariableConstraint`(expr, R_{map}, W, i)

if expr *is a linear expression* **then**
  terms $\leftarrow$ expr['terms']
  for $j \in 1, 2, \cdots, |$ terms $|$ **do**
    term$_j$ $\leftarrow$ terms[j]
    $W \leftarrow$ `processRandomVariableConstraintTerm`(term$_j$, R_{map}, W, i)
  end for
else
    $W \leftarrow$ `processRandomVariableConstraintTerm`(expr, R_{map}, W, i)
end if
return W

The utility function `processRandomVariableConstraint(...)` takes as input

- an `ILinearTerm` object representing the left-hand-side of the support constraint,
- a map of the identifiers of the declared random variables versus their corresponding `IVariable` objects,
- a partially computed matrix representing the coefficient matrix W, and
- an integer enumerating the support constraint being processed, which will correspond to a row index of the matrix W.

Moreover, it identifies whether the obtained left-hand-side expression of the constraint is a linear combination of more than one of the random variables $\xi_1, \xi_2, \cdots, \xi_T$ or whether it is just a single random variable term ξ_i. In the former case, we iterate over the terms to process each random variable term individually. However, the latter case is more simple, and we simply process the left-hand-side term as it is without any additional pre-processing or data extraction. The actual processing of a random variable is then delegated to a private helper function `processRandomVariableConstraintTerm(...)`, which updates the jth row of matrix W to represent the random variable's participation in the jth support constraint (see algorithm 5.7).

Algorithm 5.8 `processRandomVariableConstraintTerm(random`$_{term}$`,` R_{map}`, matrix,` i`)`

1. **if** random$_{term}$ *is a constant* **then**
2. matrix(i,1) $\leftarrow$ random$_{term}$[‘value’]
3. **else**
4. random$_{ID}$ $\leftarrow$ random$_{term}$[‘identifier’]
5. random$_{position}$ $\leftarrow$ R_{map}[random$_{ID}$]
6. matrix(i, random$_{position}$) $\leftarrow$ (random$_{term}$[‘coefficient’])[‘value’]
7. **end if**
8. **return** matrix

Generation of the Recourse Constraints Matrices

We attribute the responsibility for generating the recourse constraints matrices to the MATLAB class `ConstraintsMatricesGenerator`. Its `generate(...)` method is parameterised by an `ImmutableJADAModel` object and returns a `RecourseConstraintsMatrices` object. We outline the logic of this processing in algorithms 5.9 and 5.10, and we further augment the notation given by tables 5.6 to 5.8 with that of table 5.9.

Table 5.9: Notation for algorithms 5.9 and 5.10, which explain the generation of the recourse constraints matrices for use by the linear programs as defined in eqs. (Cons-MSP$_{fixed}$) and (Prog-MSP$_{fixed}$).

$A_{t,s} \in \mathbb{R}^{m_t \times n_s}$	$\stackrel{\text{def}}{=}$	is a real-valued matrix that pre-multiplies the vector of decision variables associated with the time period t.
$A_t \in \texttt{cell}^{m_t \times n_t}$	$\stackrel{\text{def}}{=}$	is a cell-array of matrices $A_{t,s} \in \mathbb{R}^{m_t \times n_s}$ for each time period t.
$B_t \in \mathbb{R}^{M_t \times k^t}$	$\stackrel{\text{def}}{=}$	is a real-valued matrix that pre-multiplies the vector of random variables associated with the time period t.
$D_{map} : string \mapsto \mathcal{D}$	$\stackrel{\text{def}}{=}$	is a hash table that maps the identifiers of decision variables to their metadata, which is encapsulated in a `IVariable` instance.
constraints$_{recourse} \in \mathcal{C}^{m \times 1}$	$\stackrel{\text{def}}{=}$	is the list of `IConstraint` objects representing the inequalities $\mathbb{E}\left[\sum_{s=1}^{T} A_{t,s} x_s(\xi^s)\right] \leq b_t(\xi^t)$ in an object-oriented manner. Each constraint has been standardised such that the right-hand-side of the inequality contains only the uncertain parameters, while the left-hand-since is a deterministic expression (see section 2.3.4) involving only the decision variables.

Algorithm 5.9 `generate`(Π)

$T \leftarrow \Pi$[‘maximum stages’]
$D_{map} \leftarrow \Pi$[‘decision variables’, ‘as map’]
$R_{map} \leftarrow \Pi$[‘random variables’, ‘as map’]
$A, B \leftarrow$ `cell`$(1, T)$
for $t \in 1, 2, \cdots, T$ **do**
  $m_t \leftarrow \Pi$[‘recourse constraints aggregator’, t]
  $k^t \leftarrow \Pi$[‘uncertainty aggregator’, t]
  $A_t \leftarrow$ `cell`$(1, T)$
  $A_{t,s} \leftarrow \mathbf{0} \in \mathbb{R}^{m_t \times n_s}$, where $n_s \leftarrow \Pi$[‘decision aggregator’, s], $\forall s \in 1, 2, \cdots, T$
  $B_t \leftarrow \mathbf{0} \in \mathbb{R}^{m_t \times k^t}$
  constraints$_{recourse} \leftarrow \Pi$[‘recourse constraints’, ‘standardised’, t]
  for $m \in 1, 2, \cdots, \mid$ constraints$_{recourse} \mid$ **do**
    constraint $\leftarrow$ constraints$_{recourse}[m]$
    exprLHS $\leftarrow$ constraint[‘LHS term’]
    for $m \in 1, 2, \cdots, \mid$ constraints$_{recourse} \mid$ **do**
      if exprLHS *is a linear expression* **then**
        terms $\leftarrow$ exprLHS[‘terms’]
        for $i \in 1, 2, \cdots, \mid$ terms $\mid$ **do**
          term$_i \leftarrow$ terms$[i]$
          $A_t \leftarrow$ `processLHSConstraintTerm`(term$_i$, D_{map}, A_t, m)
        end for
      else
        $A_t \leftarrow$ `processLHSConstraintTerm`(exprLHS, D_{map}, A_t, m)
      end if
    end for
    exprRHS $\leftarrow$ constraint[‘RHS term’]
    $B_t \leftarrow$ `processRandomVariableConstraint`(exprRHS, R_{map}, B_t, m)
  end for
  $A[t] \leftarrow A_t$
  $B[t] \leftarrow B_t$
end for
return `RecourseConstraintsMatrices`(A, B)

The `generate(...)` method *walks through* the different time stages, and processes the constraints that the `ImmutableJADAModel` instance has assigned to a particular period t. Having obtained the recourse constraints for a period t, the method then initialises the matrices $A_t \equiv \{A_{t,s} \in R^{m_t \times n_s}\}_{s=1}^{T}$ and $B_t \in R^{m_t \times k^t}$ to zero-valued matrices.

Each of the retrieved constraints for stage t is then manipulated by extracting the individual `ILinearTerm` expressions that respectively correspond to the sides of the inequality. The left-hand-side expression exprLHS is handled by the auxiliary method `processLHSConstraintTerm(...)` defined in the `ConstraintsMatricesGenerator`. Furthermore, the processing of the right-hand-side expression exprRHS is delegated to the utility method `processRandomVariableConstraint(...)` provided by the `LPGeneratorUtil` class (see algorithms 5.7 and 5.8). The derived matrices are then stored in their respective A and B cell-arrays.

Algorithm 5.10 `processLHSConstraintTerm(`decision_{term}, D_{map}, A_t, m`)`

1. decision $\leftarrow D_{map}[decision_{term}[\text{'identifier'}]]$
2. $\text{decision}_{stage} \leftarrow$ decision['stage known at']
3. $\text{decision}_{position} \leftarrow$ decision['position']
4. $A_{t,s} \leftarrow A_t[\text{decision}_{stage}]$
5. $A_{t,s}(m, \text{decision}_{position}) \leftarrow (\text{decision}_{term}[\text{'coefficient'}])[\text{'value'}]$
6. $A_t[\text{decision}_{stage}] \leftarrow \texttt{cell}(A_{t,s})$
7. **return** A_t

The `processLHSConstraintTerm(...)` incrementally builds the collection of matrices $A_{t,s}$ as it processes each decision variable encountered in the left-hand-side expression of a recourse constraint. The method is supplied the following arguments:

- an `ILinearTerm` object representing the decision variable that participates in the left-hand-side expression of the recourse constraint,
- a map of the identifiers of the declared decision variables versus their corresponding `IVariable` objects,
- a cell-array of partially generated matrices representing the recourse constraints for a specific time period, and
- an integer enumerating the recourse constraint being processed and thus a row index of the matrix $A_{t,s}$, $\forall s = 1, 2, \cdots, T$.

The method uses the identifier of the decision variable term to index the hash-table D_{map}, and thus access information such as the vector position of the decision variable and the stage it corresponds to. We use the latter attribute to obtain the relevant constraint matrix $A_{t,s}$, whose entry at the given row index i and the decision's vector position is updated with a value. This value quantifies the decision variable's participation in the ith recourse constraint at time t.

Generation of the Moments Matrices

The generation of the second-order moments and conditional moments matrices is explained in algorithm 5.11. The `generate(...)` method, which is implemented by the class `MomentsMatricesGenerator`, has been written to delegate the generation logic rather than to directly handle the processing. We refer the reader to the notation given in table 5.10, in addition to that provided thus far.

Table 5.10: Notation for algorithms 5.11-5.13, which explain the generation of the second order moments matrix and the conditional moments matrices for use by the linear programs as defined in eqs. (Cons-MSP$_{fixed}$) and (Prog-MSP$_{fixed}$).

$\bar{\xi} \in \mathbb{R}^T$	$\stackrel{\text{def}}{=}$	is the vector of the individual uncertain parameters for all stages of the multi-stage stochastic programming problem.
$\Sigma \in \mathbb{R}^{k \times k}$	$\stackrel{\text{def}}{=}$	is the covariance matrix for the random vector $\xi \in \mathbb{R}^T$.
$M_{\mathbb{E}[\xi\xi^\top]} \in \mathbb{R}^{k \times k}$	$\stackrel{\text{def}}{=}$	is a real-valued matrix representing the second order moments $\mathbb{E}[\xi\xi^\top]$.
$M_t \in \texttt{cell}^{k \times k^t}$	$\stackrel{\text{def}}{=}$	is the real-valued matrix representing the conditional moments matrix computed as, $M_{\mathbb{E}[\xi \mid \xi^t]}$, for time period t.
$M_{\mathbb{E}_t[\xi]} \in \texttt{cell}^{k \times k}$	$\stackrel{\text{def}}{=}$	is a cell-array of the conditional moments matrices M_t for each time period t.

Algorithm 5.11 `generate`(Π)

1. $[\bar{\xi}, \Sigma] \leftarrow$ `computeExpectationsCovariances`(Π)
2. $[M_{\mathbb{E}[\xi\xi^\top]}, M_{\mathbb{E}_t[\xi]}] \leftarrow$ `computeExpectationsCovariances`(Π)
3. **return** `MomentsMatrices`$(M_{\mathbb{E}[\xi\xi^\top]}, M_{\mathbb{E}_t[\xi]})$

We define an auxiliary function to facilitate the computation of the expectation vector and the covariance matrix, which uses either the support parameters or the sample data(see algorithms 5.12).

The sample expectations and covariances are then supplied to another method to initiate the derivation of the moments matrices (see algorithm 5.13).

The computation of the second order moments matrix $M_{\mathbb{E}[\xi\xi^\top]}$ involves calculating the expectation matrix

$$\mathbb{E}[\xi\xi^\top] = \begin{pmatrix} \mathbb{E}[\xi_1^2] & \mathbb{E}[\xi_1\ \xi_2] & \cdots & \mathbb{E}[\xi_1\ \xi_T] \\ \mathbb{E}[\xi_2\ \xi_1] & \mathbb{E}[\xi_2^2] & \cdots & \mathbb{E}[\xi_2\ \xi_T] \\ \vdots & \vdots & \ddots & \vdots \\ \mathbb{E}[\xi_T\ \xi_1] & \vdots & \cdots & \mathbb{E}[\xi_T^2] \end{pmatrix}. \tag{5.4.2.1}$$

We note that an independent random variable ξ_i has a variance defined by $\mathbb{V}ar[\xi_i] = E[\xi_i - E[\xi_i]] = E[\xi_i^2] - E[\xi_i]^2$, which implies that $\mathbb{E}[\xi_i^2] = Var[\xi_i] + E[\xi_i]^2$.

Consequently, if $\forall i, j = 1, 2, \cdots, k$ we let $\mathbb{V}ar[\xi_i]$, $\mathbb{E}[\xi_i]$, $\mathbb{V}ar[\xi_j]$, $\mathbb{E}[\xi_j]$, and $\mathbb{C}ov[\xi_i, \xi_j] =$

$\mathbb{C}ov[\xi_j, \xi_i]$ be respectively denoted by σ_i^2, $\bar{\xi}_i$, σ_j^2, $\bar{\xi}_j$, and $\sigma_{i,j}$, then

$$M_{\mathbb{E}\,[\xi\xi^{\top}]} = \begin{pmatrix} \sigma_1 + \bar{\xi_1}^2 & \sigma_{1,2} + \bar{\xi_1}\ \bar{\xi_2} & \cdots & \mathbb{E}\,[\xi_1\ \xi_T] \\ \sigma_{2,1} + \bar{\xi_2}\ \bar{\xi_1} & \sigma_2 + \bar{\xi_2}^2 & \cdots & \sigma_{2,T} + \bar{\xi_2}\ \bar{\xi_T} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{T,1} + \bar{\xi_T}\ \bar{\xi_1} & \sigma_{T,2} + \bar{\xi_T}\ \bar{\xi_2} & \cdots & \sigma_T + \bar{\xi_T}^2 \end{pmatrix}. \tag{5.4.2.2}$$

Algorithm 5.12 `computeExpectationsCovariances(Π)`

$k \leftarrow \Pi$[‘uncertainty aggregator’, Π[‘maximum stages’]]
$R \leftarrow \Pi$[‘random variables’, ‘sorted’]
$\bar{\xi} \leftarrow \mathbf{0} \in \mathbb{R}^{k\times 1}$
$\bar{\xi}(1,1) \leftarrow 1$
$\Sigma \leftarrow \mathbf{0} \in \mathbb{R}^{k\times k}$
for $i \in 1, 2, \cdots, \mid R \mid$ **do**
  random$_i \leftarrow R[i]$
  if random$_i$ *has sample data* **then**
    samples$_i \leftarrow$ random$_i$[‘sample data’]
    $\bar{\xi}_i \leftarrow$ `mean`(samples$_i$)
    $\sigma_i^2 \leftarrow$ `var`(samples$_i$)
  else
    $a =$ random$_i$[‘lower bound support parameter’]
    $b =$ random$_i$[‘upper bound support parameter’]
    $\bar{\xi}_i \leftarrow \frac{1}{2}(a+b)$
    $\sigma_i^2 \leftarrow \frac{1}{12}(b-a)^2$
  end if
  $\bar{\xi}(i{+}1,1) \leftarrow \bar{\xi}_i$
  $\Sigma(i{+}1,i{+}1) \leftarrow \sigma_i^2$
end for
return $[\bar{\xi}, \Sigma]$

Algorithm 5.13 `computeMoments(`$\bar{\xi}$`, `Σ`, `Π`)`

$M_{\mathbb{E}[\xi\xi^\top]} \leftarrow \Sigma + \bar{\xi}\bar{\xi}^\top$
if $M_{\mathbb{E}[\xi\xi^\top]}$ *is singular* **then**
  throw `MEException`
end if
$T \leftarrow \Pi$['maximum stages']
$M_{\mathbb{E}_t[\xi]} \leftarrow$ `cell`$(1,\ T)$
$k \leftarrow \Pi$['uncertainty aggregator', T]
$R \leftarrow \Pi$['random variables', 'sorted']
for $t \in 1,\ 2,\ \cdots,\ T$ **do**
  $k^t \leftarrow \Pi$['uncertainty aggregator', t]
  $M_t \leftarrow \mathbf{0} \in \mathbb{R}^{k \times k^t}$
  $M_t(1{,}1) \leftarrow 1$
  for $j \in 1,\ 2,\ \cdots,\ \mid R \mid$ **do**
    if $(R[i])$['stage known at'] $\leq t$ **then**
      $M_t(j{,}j) \leftarrow 1,\ \forall j = 1, 2, \cdots, k^t$
    else
      $M_t(j,\ 1) \leftarrow \bar{\xi}(\mathrm{j}),\ \forall j = 1, 2, \cdots, k$
    end if
  end for
$M_{\mathbb{E}_t[\xi]}[t] \leftarrow M_t$
end for
return $[M_{\mathbb{E}[\xi\xi^\top]},\ M_{\mathbb{E}_t[\xi]}]$

5.4.3 Problems Encountered

Determining the Singularity of the Second-order Moments Matrix

The second-order moments matrix needs to be inverted for computing the constraints for the progressive linear program. However, this cannot be done if the symmetric matrix is singular due to insufficient sample data points. Below we state the approaches explored for determining matrix singularity for potentially large matrices.

(i) A naive solution is to attempt to inverse such a matrix in a try-catch brace and re-throw the caught exception with a more intuitive error message. However, we discovered that MATLAB does not throw an exception in such cases. It merely displays a silent warning message 'Matrix is singular to working precision', with the entries in the resulting matrix having the IEEE arithmetic representation for positive infinity.

(ii) Linear algebra theory tells us that a matrix is invertible if its determinant is zero-valued. However, such computation is only appropriate for small matrices since the values of the determinants increase to large unrepresentable values very quickly.

(iii) Another approach involves determining if the first smallest eigenvalue of the symmetric matrix is zero. Although, MATLAB's `eigs(...)` function for computing the eigenvalues is relatively fast, the potential computational memory requirements is a cause of concern. This is also our justification for not considering an application of a QR decomposition to find the absolute value of the determinant.

(iv) The last alternative is to check the rank of the matrix, which itself can be quite time-intensive but is possibly the most efficient in terms of its memory use.

Thus, we proceed with assessing non-singularity by checking that the matrix has full rank using the function `spnrank(...)`, from the external MATLAB framework *SJsingular* developed by *San Jose State University, Mathematics Department*[53].

Listing 5.2: Utilisation of the **spnrank**(...) routine to check the singularity of the second-order moments matrix.

```
...
if spnrank(secondOrderMoments) ~= min(size(secondOrderMoments))
    message = 'The second order moments matrix M is not invertible due to
        insufficient sample data points.';
    throw(MEException('MomentMatricesGenerator:computeMoments:
        singularSecondOrderMomentsMatrix', message));
end %if
```

The **spnrank**(...) routine implements an algorithm based on Sylvester's low of inertia[55]. Apart from providing the benefits of a sufficiently high degree of accuracy, the method also acquires a required robustness to handle matrices up to dimensions of 682,712 by 682,712[54].

5.5 Approximator

We make the decision to abstract the actual generation of the *conservative* and *progressive* linear programs to reduce the responsibilities of the `generator` package. In doing so, we also reduce the overall system coupling. As illustrated by the sequence diagram figure (5.2), the `System` initialises the `YALMIPApproximators` class with the conservative and progressive `LPApproximatorModel` instances generated by the `LPGenerator` class. Using these models, the linear programs can then be generated to compute the lower and upper bounds of the optimal solution to the original stochastic programming problem.

5.5.1 Design Structure

The `YALMIPApproximators` class behaves like a co-ordinator and appropriately distributes the tasks of generating the objective function and constraints for the conservative and progressive linear programming problems. In this section, we describe the architectural design of this module, the general algorithm for computing the objective function and constraints, and how this module interfaces with the `YALMIP` convex programming framework to invoke the exposed external solvers.

The `approximator` module has been designed to maximise code re-usability and encapsulation of the underlying models of the parsed and derived meta-information. Consequently, the `LPApproximator` class provides the majority of the constraints and objective function computation, and invokes methods to be overriden by functionality particular to the `ConservativeApproximator` and `ProgressiveApproximator` classes. Additionally, the `LPApproximator` class is implemented as a wrapper for the `LPApproximatorModel`, which receives queries to retrieve or modify specific data and delegates the requests to the internal model.

The `LPApproximatorModel`, as diagrammatically explained by the UML class diagram figure (5.7) is an abstraction of the parsed input file (`ImmutableJADAModel`), the generated LP matrix components (`LPModel`), and dictionary structures to persist the computed objective function and the constraints for solving and presenting the results.

5.5.2 Algorithms

In this section, we expound on the general computation of the objective function and the constraints. For the pseudo-code presentation of the logic for the `approximator` classes, we direct the attention of the reader to tables 5.6 to 5.11 as points of reference for the notation used in algorithms 5.14 and 5.15, and sections C.1.1 and C.1.2.

Table 5.11: Notation for algorithms 5.14 and 5.15, which explain the generation of the linear programs eqs. (Cons-MSP$_{fixed}$) and (Prog-MSP$_{fixed}$) that serve to approximate the original optimisation problem.

$P_t : \mathbb{R}^k \mapsto \mathbb{R}^{k^t}, t \in \mathbb{T}$	$\stackrel{\text{def}}{=}$	is the truncation operator.
Γ	$\stackrel{\text{def}}{=}$	is an instance of the MATLAB class `LPApproximatorModel` which contains a compact representation of the parsed input file and the derived matrices to be used for computing the objective function and the constraints.
f	$\stackrel{\text{def}}{=}$	is an `sdpvar` variable that represents the resulting computation of the objective function using the `sdpvar` decision rule matrices.
$f_{symbolic}$	$\stackrel{\text{def}}{=}$	is an ILinearTerm object that represents the resulting computation of the objective function using the symbolic decision rule matrices.
$g_C(X_{sdpvar}, X_{symbolic})$	$\stackrel{\text{def}}{=}$	denotes the computed feasibility equality-constraint for the conservative linear program.
$g_P(X_{sdpvar}, X_{symbolic})$	$\stackrel{\text{def}}{=}$	denotes the computed feasibility equality-constraint for the progressive linear program.
$h_C(\Lambda_{sdpvar}, \Lambda_{symbolic})$	$\stackrel{\text{def}}{=}$	denotes the computed slack decision rule bounds for the conservative linear program.
$h_P(S_{sdpvar}, S_{symbolic})$	$\stackrel{\text{def}}{=}$	denotes the computed slack decision rule bounds for the progressive linear program.

Computing the Objective Function

To compute the objective function, we are required to iterate through the stages $1, \cdots, T$ of the stochastic model, which permits us to calculate the summation of the iterated expression $\mathbb{T}race(P_t M_{\mathbb{E}[\xi\xi^\top]} P_t^\top C_t^\top X_t)$ (see 5.14).

Algorithm 5.14 `computeObjectiveFunction()`

---

1. $X \leftarrow \Gamma$[‘decision matrices’, ‘sdpvar’]
2. $X_{symbolic} \leftarrow \Gamma$[‘decision matrices’, ‘symbolic’]
3. $f \leftarrow 0.0$
4. $f_{symbolic} \leftarrow$ *new* `ConstantTerm`(0.0)
5. **for** $t \in 1, 2, \cdots, \Gamma$[‘maximum stages’] **do**
6. $C_t \leftarrow \Gamma$[‘costs matrices’, t]
7.   **if** $\neg C_t$ *empty* **do**
8.   **for each** $(\hat{X}_t,) \in \{(X[t], f), (X_{symbolic}[t], f_{symbolic})\} \wedge \neg\, X_t$ *empty* **do**
9. $\hat{f} \leftarrow + \mathbb{T}race(\mathrm{P}_t(\Gamma[\text{‘second order moments’}]) P_t^\top C_t^\top \hat{X}_t)$
10.   **end for each**
11.   **end if**
12. **end for**
13. Γ[‘objective function’] $\leftarrow [f_{sdpvar}, f_{symbolic}]$

We clarify that the computation of the expression $\mathbb{T}race(P_t M_{\mathbb{E}[\xi\xi^\top]} P_t^\top C_t^\top X_t)$ at time period t is only done if there exists a cost matrix C_t (line 7) and a decision rule matrix X_t (line 8) for that time period. Thus, in the case where the modeller is not required to make any decision at, for example, the first stage, then for iteration $t = 1$ we need not compute any values.

Computing the Constraints

We intentionally implement the method `computeConstraints()` in the `LPApproximator` class to generalise the calculation of the constraints. This involves abstracting away from whether we are deriving the constraints for the conservative or progressive LP, and then handling variable points in the code by using abstract methods as placeholders. We refer the reader to sections C.1.1 and C.1.2 in section C.1 for further details of how the abstract methods are overridden by the derived MATLAB classes `ConservativeApproximator` and `ProgressiveApproximator`.

Algorithm 5.15 `computeConstraints`()

$T \leftarrow \Gamma$[‘maximum stages’]
$X \leftarrow \Gamma$[‘decision rules’, ‘sdpvar’]
$X_{symbolic} \leftarrow \Gamma$[‘decision matrices’, ‘symbolic’]
for $t \in 1, 2, \cdots, T$ **do**
  $A_t \leftarrow \Gamma$[‘LHS recourse constraints matrices’, t]
  $B_t \leftarrow \Gamma$[‘RHS recourse constraints matrices’, t]
  $U_t \leftarrow$ `getDecisionRulesOuterFactor`(t)
  for each $(\text{expr}_{LHS}, \hat{X}) \in \{(\text{expr}, X), (\text{expr}_{symbolic}, X_{symbolic})\}$ **do**
    $\text{expr}_{LHS} \leftarrow \sum_{s=1, \neg A_t[\text{s}]\, empty}^{T} A_t[\text{s}]\hat{X}_s P_s U_t P_t$
    $\text{expr}_{LHS} \leftarrow$ `getStandardisedFeasibilityCondition`(expr)
  end for each
  $\text{expr}_{RHS} \leftarrow B_t P_t$
  $\text{constraint}_{LHS} \leftarrow$ `cell`(expr, $\text{expr}_{symbolic}$)
  $\text{constraint}_{RHS} \leftarrow$ `cell`(expr_{RHS}, expr_{RHS})
  $\text{constraint}_{quantifier} \leftarrow$ ConstraintQuantifier.EQ
  $g_{(\cdot)}(X, X_{symbolic}) \leftarrow [\text{constraint}_{LHS}, \text{constraint}_{quantifier}, \text{constraint}_{RHS}]$
  $h_{(\cdot)}(X, X_{symbolic}) \leftarrow$ `getPositiveSlacknessCondition`(t)
  Γ[‘computed constraints’, t] $\leftarrow$ `cell`($g_{(\cdot)}(X, X_{symbolic})$, $h_{(\cdot)}(X, X_{symbolic})$)
end for

5.5.3 Interfacing the External Solvers

Once the linear program approximator classes have completed their tasks, the abstract class `Approximators` is able to communicate their computed objective functions and the constraints to the YALMIP framework. JADA interfaces with the external solvers at a single point in the system, where it invokes the `solvesdp(...)` command to solve the optimisation problem (see code listing 5.3).

Listing 5.3: Interfacing with the available external solvers via YALMIP convex programming framework.

```
...
    %Settings
    options = sdpsettings('verbose', 0, 'cachesolvers', true);

    % Function 'solvesdp(F,h,options)' is the common command to solve
    % optimization problems of the following kind:
    % 'min{ h | F >(=) 0}'
    solvesdp(constraints, objectiveFunction, options);
...
```

5.6 Renderer

The `renderer` package behaves as a custom reporting engine to present the results to the modeller. Currently, JADA generates three files for each LP approximation, which amounts to six files generated in total within a folder ‘`.../JADA/results`’ created in the user's *temp* directory. The files produced include

- the `.lpmodel` files which represent the generated linear programs by the `ConservativeApproximator` and `ProgressiveApproximator` classes,
- the `.optimality` files which contain the optimal values of the decision and slack variables that were represented as `sdpvar` objects , and
- the `.rules` files which specify the optimal decisions as functions of the random variables.

In figs. 5.6d, 5.7c, 5.8a and 5.8b, we use the results of the conservative approximation for the newsvendor optimisation problem to present examples of the aforementioned files.

5.6.1 Problems Encountered

Variable Scope

We briefly explained in section 5.4.2 the need to maintain a hash-table, which would keep track of the logical mappings of the `sdpvar` variables with the declared decision variables. The reason for this is related to the requirement to present to the user the generated linear program and the optimal solutions. These representations are both parameterised by these matrices of `sdpvar` variables. For clarification, we point out that to display an `sdpvar` object in symbolic MATLAB form, the method `sdisplay(...)` can be invoked with the object supplied as the argument.

Listing 5.4: Example declaration and symbolic display of `sdpvar` objects from the YALMIP convex programming framework.

```
>> x = sdpvar(1,1)
>> y = sdpvar(1,1)
>> f = [x; 7*x + 2*y + 2*x + 7*y]

Linear matrix variable 2x1 (full, real, 2 variables)

>> sdisplay(f)

ans =
    ‘x’
    ‘9*x + 9*y’
```

The desired identifier for the entry (i,j) of the decision rule matrix X_t at stage t is a mangled string of the form `x_t_i_j`, however the `sdpvar` object provides no functionality for assigning these variables a specific name to be denoted by. Instead, the object is dynamically assigned an identifier that corresponds to the name of its declaration. Hence, a workaround is to use MATLAB's `eval(...)` function to execute a string containing the desired assignment expression.

Listing 5.5: Using MATLAB's `eval(...)` to declare an `sdpvar` object with a specific name.

```
...
for t=1:numStages
  for i: decisionsAggregator(t)
       for i: decisionsAggregator(t)
          ...
          identifier = [‘x_’, num2Str(t), ‘_’, num2Str(i), ‘_’, num2Str(j)];
          eval([identifier, ‘ = sdpvar(1,1)’]);
          decisionMatrix_t(i,j) = eval(identifier);
          ...
       end %for j=1:uncertaintyAggregator(t)
  end %for i=1:decisionsAggregator(t)
end %for t=1:numStages
..
```

Besides this approach being very awkward and heavily inefficient, we encountered a serious problem when it came to rendering the `sdpvar` objects outside the scope that they were defined. Thus, rather than displaying the mangled identifier, the symbolic display function displayed the variable name ‘internal’ when invoked outside the `DecisionMatricesGenerator` class. As a result, we had to develop an expressions utility library that represented and performed the linear operations involving variables, vectors and matrices in an object-oriented manner by overloading MATLAB's built-in functions.

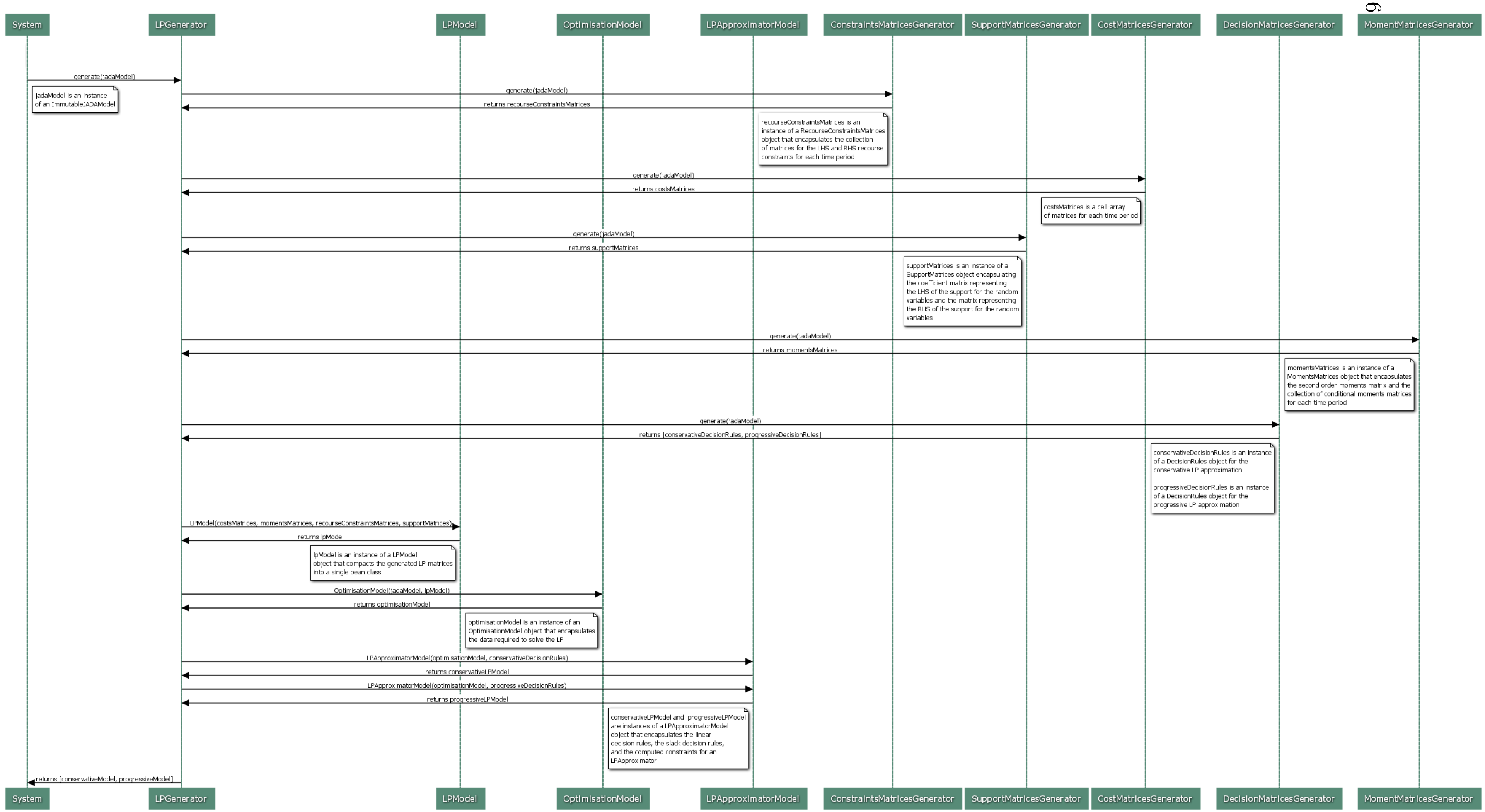


Figure 5.6: UML sequence diagram illustrating the pattern of interaction to achieve generation of the components of the *conservative* and *progressive* LP.

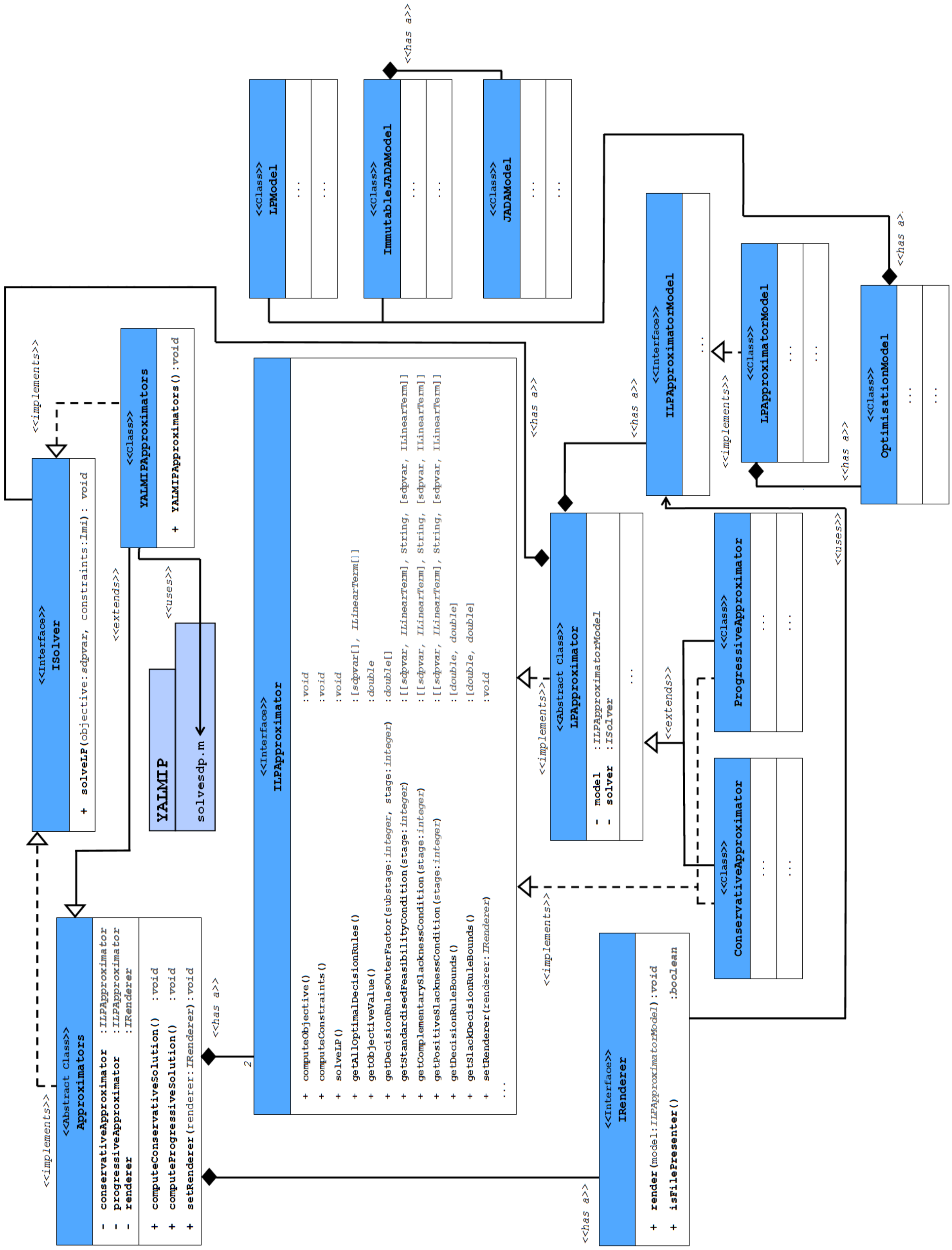
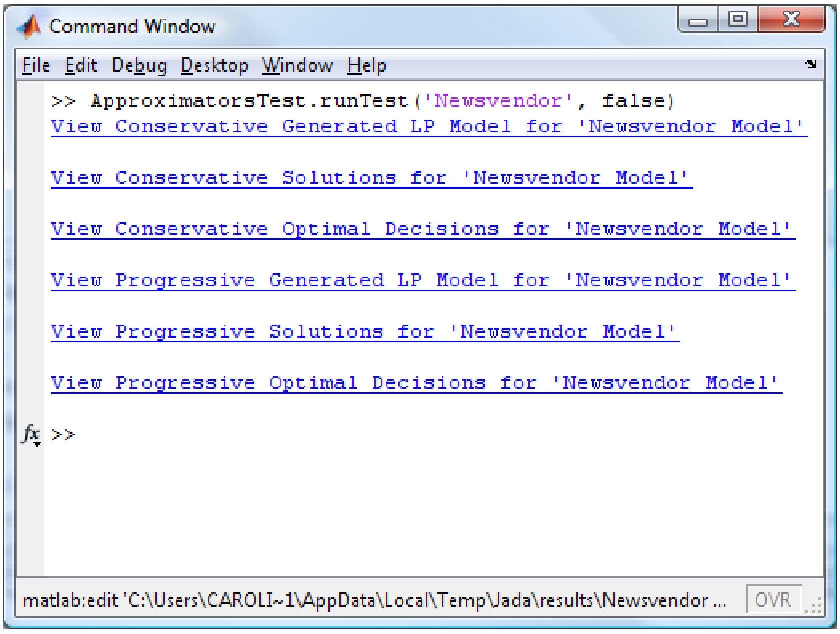
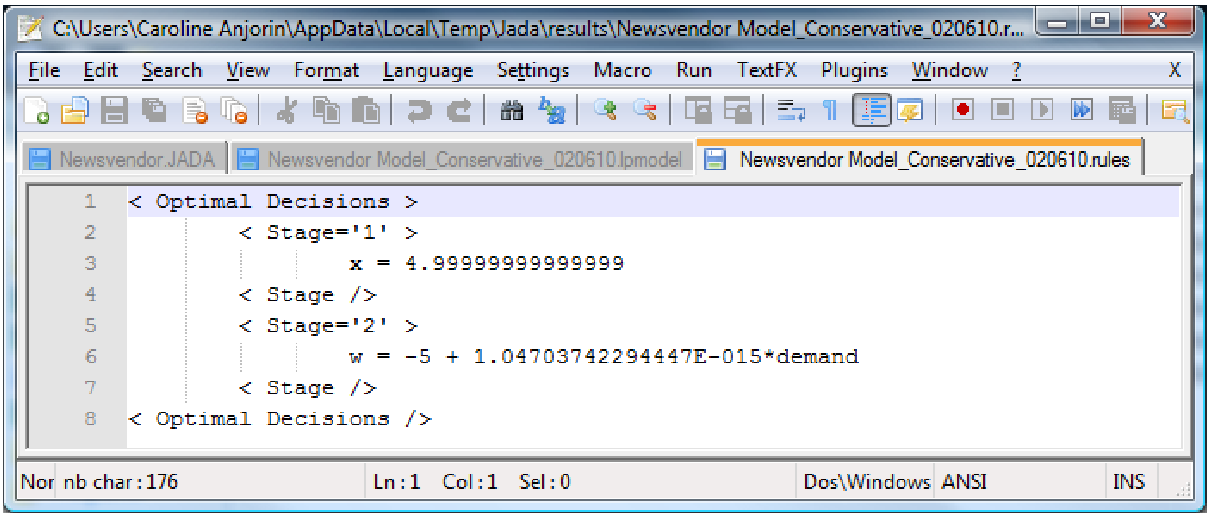


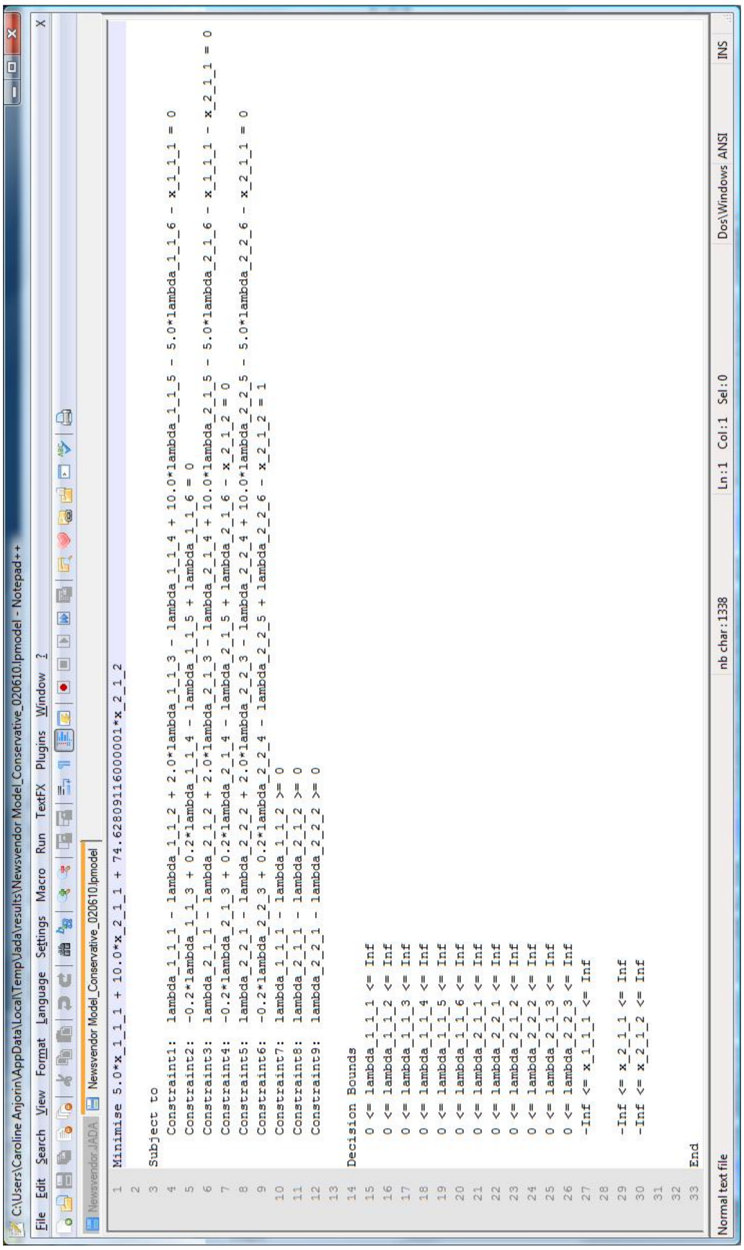
Figure 5.7: UML class diagram showing the structural dependencies of the classes in the `approximator` package, as well as the point of interfacing of the JADA system with the YALMIP convex programming framework to access a miscellany of external solvers.



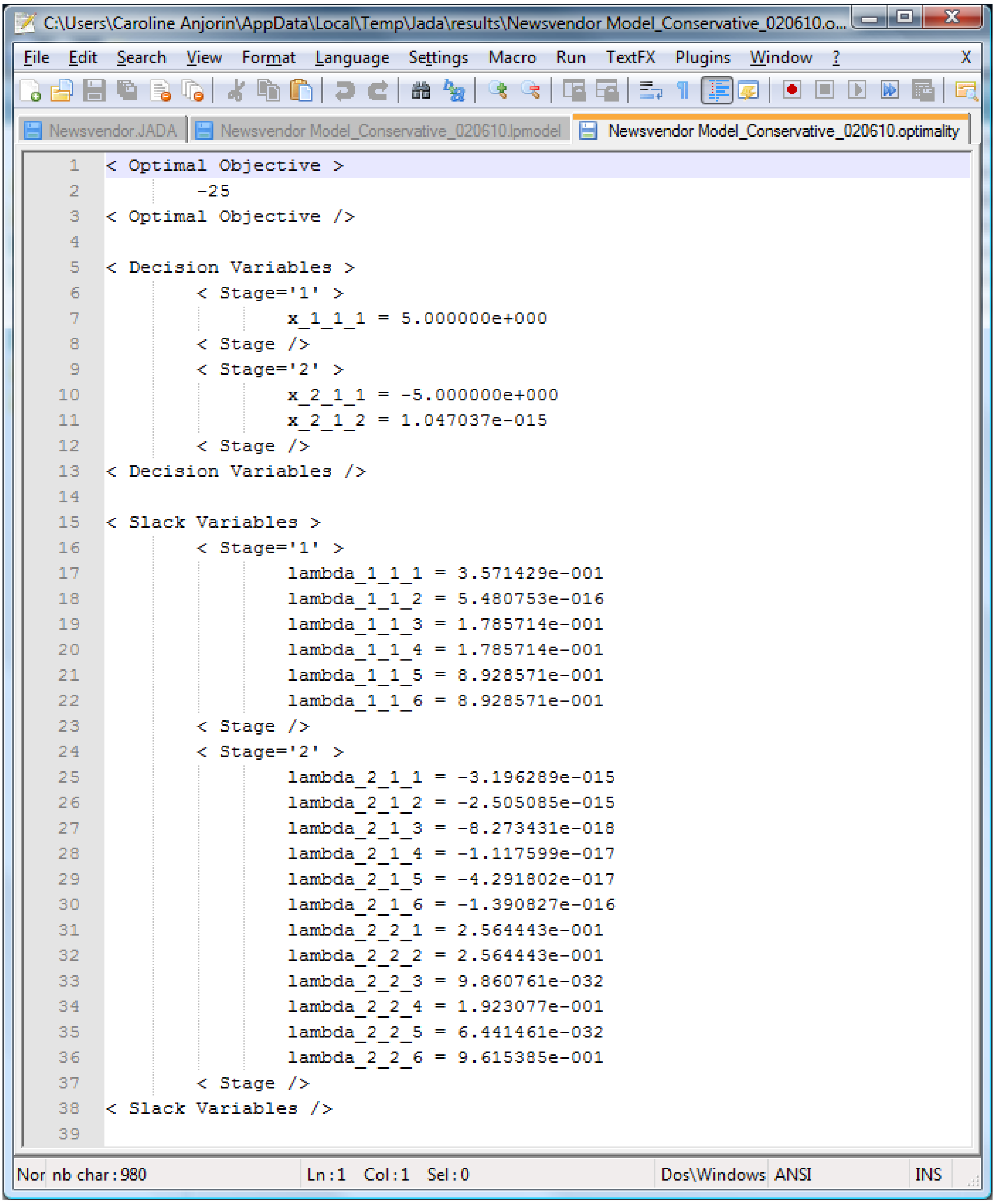
(a) Notification of generated output files as hyper-links in the MATLAB command window.



(b) The `.rules` file for the conservative approximation of the newsvendor problem.



(c) The `.lpmodel` file for the conservative approximation of the newsvendor problem.




```
< Optimal Objective >
        -25
< Optimal Objective />

< Decision Variables >
        < Stage='1' >
                x_1_1_1 = 5.000000e+000
        < Stage />
        < Stage='2' >
                x_2_1_1 = -5.000000e+000
                x_2_1_2 = 1.047037e-015
        < Stage />
< Decision Variables />

< Slack Variables >
        < Stage='1' >
                lambda_1_1_1 = 3.571429e-001
                lambda_1_1_2 = 5.480753e-016
                lambda_1_1_3 = 1.785714e-001
                lambda_1_1_4 = 1.785714e-001
                lambda_1_1_5 = 8.928571e-001
                lambda_1_1_6 = 8.928571e-001
        < Stage />
        < Stage='2' >
                lambda_2_1_1 = -3.196289e-015
                lambda_2_1_2 = -2.505085e-015
                lambda_2_1_3 = -8.273431e-018
                lambda_2_1_4 = -1.117599e-017
                lambda_2_1_5 = -4.291802e-017
                lambda_2_1_6 = -1.390827e-016
                lambda_2_2_1 = 2.564443e-001
                lambda_2_2_2 = 2.564443e-001
                lambda_2_2_3 = 9.860761e-032
                lambda_2_2_4 = 1.923077e-001
                lambda_2_2_5 = 6.441461e-032
                lambda_2_2_6 = 9.615385e-001
        < Stage />
< Slack Variables />
```

(d) The `.optimality` file for the conservative approximation of the newsvendor optimisation problem.

Figure 5.6: Generated results files for the conservative approximation of the newsvendor problem

6

Incorporating Stochastic Processes for a More Expressible AML

In chapter 5, we have methodically explained a basic implementation that will allow us to meet the fundamental requirements of this project. In this chapter we describe an implemented extension to maximise the expressibility and flexibility associated with specifying a problem in our input format.

6.0.2 Motivation

As we explained in section 2.3.2, we can consider decision-making under uncertainty as a finite process, which consists of interleaving decisions and observations that occur over time stages. Therefore, we can model our decisions and observations as indexable sets.

To extend the JADA syntax, we intend to introduce a notational representation of decision-making as a stochastic *process*. Our primary motivation, for enriching the language in this way, stems from a requirement to allow for the modeller to specify their optimisation problems in a succinct and flexible manner. For example, consider the abstracted decision-making problem in eq. (6.0.2.1).

$$
\begin{aligned}
&\textit{minimise}\ \mathbb{E}\left[\textstyle\sum_{t=1}^{T}\sum_{i=1}^{N} c_{i,t}\, x_{i,t}(\xi^{\,t})\right] \\
&\textit{subject to} \\
&\qquad \left.\begin{array}{l}
x_{i,t} \in \mathcal{L}^2_{t,1} \\[2ex]
x_{i,t}^{min} \leq x_{i,t}(\xi^{\,t}) \leq x_{i,t}^{max} \\[2ex]
\sum_{t=1}^{T} x_{i,t}(\xi^{\,t}) \leq x_t^{total}
\end{array}\right\} \quad \forall i \in N, \forall t \in \mathbb{T}
\end{aligned}
\tag{6.0.2.1}
$$

A formulation of the equivalent linear program, using our current algebraic modelling language, requires explicit declarations of the individual decision variables, random variables, as well as the individual terms in the objective function and constraints. Thus before the modeller can use our input format, eq. (6.0.2.1) must undergo a preliminary transformation to expand the summations $\sum_{t,i}$ and the universal quantifiers $\forall$. This is obviously tedious and inconvenient if the JADA file cannot be generated programmatically. To eliminate the need to re-translate the optimisation problem, we aim to introduce syntax for

- declaraing decision and random processes,
- declaring single-valued and vector-valued constants,
- indexing expressions, and
- iterating over indexed expressions using summation or universal quantifiers.

6.0.3 New Language Features

Decision and Random Processes

We amend our syntax for specifying the optimisation problem's decision variables and random variables to allow for a more succinct representation. A family of decision variables need not be declared individually, but can instead be declared as a decision process that is parameterised by

- a string which uniquely identifies the decision process,
- an index range, given by two integers, which specify the stage at which the decision process commences and terminates,
- an integer that quantifies the number decisions at each stage of the process.

Similary, we can describe a random process by giving

- a string which uniquely identifies the random process,
- an range, given by two integers, which specify the stage at which the decision process commences and terminates, and
- a list of comma-delimited ranges which specify the shape of each of the random variables' probability distribution.

Thus, if we let $T = 3$ and $N = 3$, we can declare the decision and random variables required for eq. (6.0.2.1) by two single declarations, as shown in code listing 6.1.

Listing 6.1: Modified design for the `Variables` subsection of the JADA file using eq. (6.0.2.1) as the motivation example.

```
...
Variables
{
   decision(x,1:3,3);
   random(y,2:3,0:1000000,0:1000000);
}
...
```

We can contrast this with the initial design of the input format, which we specified in section 4.3 and demonstrate in code listing 6.2.

Listing 6.2: Initial design for the `Variables` subsection of the JADA file using eq. (6.0.2.1) as the motivational example.

```
...
Variables
{
   decision(x1,1);
   decision(x2,2);
   decision(x3,3);

   random(y2,2,0,1000000);
   random(y3,3,0,1000000);
 }
...
```

We point out to the reader that we require an amended modelling of the distribution of the random variables. To reduce the scope for ambiguity, we use the notation `<minimum>:<maximum>` to specify the bounds, rather than use commas to delimit the values of the lower and upper bounds.

Constant Parameters

To facilitate the modeller's specification of his or her decision-making problem, we introduce special variables with known, static values which can be referred to in symbolic expressions and in definitions of other constants. We use the reserved keyword `constant` to make an explicit semantic distinction between the model variables and the constants. The syntax for declaring a constant parameter is shown in code listing 6.3.

Listing 6.3: Modified design for the `General` subsection of the JADA file, to introduce declarations of *constant* parameters.

```
...
General
{
   ...
   constant(default_cost,10);
   constant(cost_factor, 1.0,2.5,3.0);
   constant(additional_cost,1/45+3.0,5+12.08765,9.0-3);

   constant(total_cost, additional_cost#1 + default_cost*cost_factor#1,
                        additional_cost#2 + default_cost*cost_factor#2,
                        additional_cost#3 + default_cost*cost_factor#3);

   constant(min,10,17,13);
   constant(max,20);

   constant(total,100,75);
   ...
 }
...
```

Constants are defined in the `General` construct and their declarations are syntactically parameterised by a unique identifier and a list of values. In code listing 6.3, we would like to draw the reader's attention to the variations in the manner that constants can be declared.

- Constants can be single-valued (line 5) or vector-valued (line 6).
- Constants can be defined using arithmetic expressions (line 7).
- Declarations of constants can refer to previously defined single-valed constants (line 5) or vector-valued constants (line 11).

Sigma Notation

We utilise eq. (6.0.2.1) as a motivational example for introducing syntactical constructs for specifying iterated addition. This allows the decision-maker to compactly and precisely express any sequence of linear terms to be added. We make the choice to model summation using the *Sigma Notation*, and in the generic expression $\sum_{i=\alpha}^{\beta} f(i)$, we identify the following components[56]:

- the letter k is denotes the *index variable* or the *index of summation* and adopts integer values in the range $[\alpha, \beta]$,
- the values α and β are the starting and ending index of summation, and
- $f(i)$ is the iterated expression that specifies each individudal term in the final sum.

Code listing 6.4 illustrates the syntactic declaration of a summation to specify the objective function of eq. (6.0.2.1).

Listing 6.4: Declaration of the objective function the 'sum' construct.

```
...
 Objective
 {
   minimise expectation sum(t=1:3, i=1:3)(total_cost#i * x#i#t * y#t);
 }
...
```

In the initial design, we explained that we modelled multiplication of the decision variables with the random variables using the square parentheses. Upon further discussion, we noted that this syntax is not as intuitive as using the normal multiplication symbol '*'. Although, the initial design makes it easier to extract the costs of the decision variables, since it encourages the modeller to factorise the objective function, we feel familiarity and convenience is more important.

Universal Quantification

In predicate logic, universal quantification formalises the notion that a *logical predicate* or *proposition* is true for the *universe of discourse*, which is the set of objects of interest. In symbolic logic, the universal quantifier $\forall$ is used to denote universal quantification, and is informally read as 'for all'. For the quantified formula $\forall P(x)$,

- $P(x)$ denotes the *predicate* or *atomic formula*, and
- x is an object in the universe of discourse.

For our own purposes, we will introduce universal quantification to aid the modeller's specification of the recourse constraints and the support constraints. Its syntax is similar to the summation construct, except that the iterated expression is a constraint. This is illustrated in code listing 6.5 for modelling the recourse constraints of eq. (6.0.2.1).

Listing 6.5: Declaration of the recourse constraints using the 'forall' construct.

```
...
Constraints
{
  forall(t=1:3, i=1:3)(min#i <= x#i#t);

  forall(t=1:3, i=1:3)(x#i#t <= max);

  forall(t=1:3, i=1:3)(sum(t=1:3)(x#i#t) <= total#t);
}
...
```

Indexing expressions

For vector-valued variables and constants, we allow the user to reference the components of a declared process or vector using the hash symbol '#'. Additionally, we permit references to the index variables in the iterated expressions using the hash symbol. Thus, the example expression $\sum_{t=1}^{T} 3 * t$ can be represented as `sum(t=1:2)(3*#t)`.

6.0.4 Parser Modifications

To implement the notation for stochastic processes, we are only required to re-implement parts of the parser.

- We modify the lexer grammar file `JADALexer.g` to introduce new lexical tokens for the reserved keywords 'sum' and 'forall',
- We augment the rules in the parser file `JADAParser.g` to capture our new representation of constants, decision and random processes, indexable expressions, iterated addition, universal quantification for specifying constraints, and finally the use of explicit multiplication in the objective function. Additionally, we ensure that the `SampleDataParser` permits implicit references to random variables, using the stochastic process notation, to specify the sample data.
- We re-factor the tree parser grammar file `JADATreeParser.g` to accommodate the modified abstract syntax trees we now generate. Furthermore, we perform the expansion of the iterated expressions at this point in the code-base.

In section D.1 we present some syntax diagrams, which serve to explain the formation of our parser rules to parse the declarations of constants, declarations and references to decision and random processes, explicit multiplication in the objective function, and the iterated expressions.

In addition to the aforementioned syntax diagrams, we refer the reader to the syntax digrams for the unmodified grammar in sections B.3 and B.4 to understand how expressions are interpreted for 'sum' construct (see figure (6.1)) and 'forall' construct (see figure (6.2)).

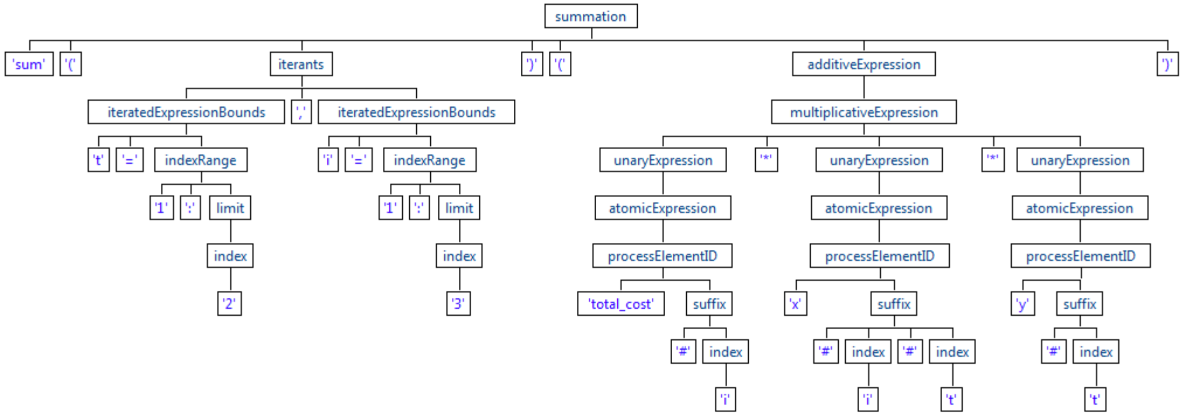


Figure 6.1: Diagram showing the interpretation of the *summation* parser rule for code listing 6.4 (line 4).

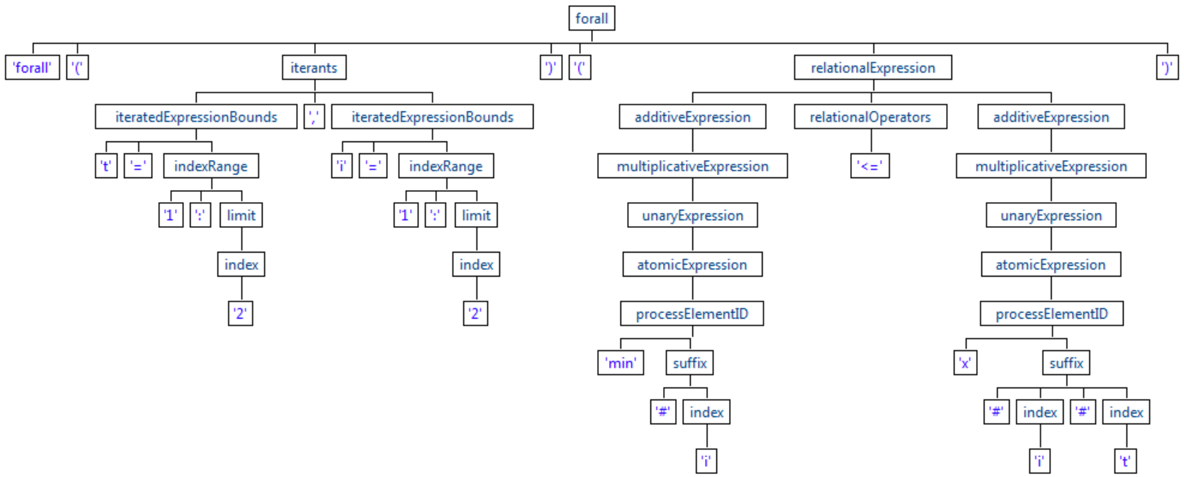


Figure 6.2: Diagram showing the interpretation of the *universal quantification* parser rule for code listing 6.5 (line 4).

7 Numerical Evaluation

In this chapter, we consider two different stochastic programming problems to supplement the evaluation of the implemented system. For each case-study, we qualify the decision-making problem by giving a qualitative description and a mathematical formulation of the underlying optimisation problem. We then specify the decision-making problem using our standardised input format. Finally, we import the JADA library into MATLAB to parse the specified model, and to compute the solutions of the generated *conservative* and *progressive* linear programs.

Our simulation environment consists of MATLAB R2008a software running on a 32-bit Windows Vista™ Home Premium machine with a 1.90GHz AMD Turion™ 64 X2 TL-58 processor, which uses dual-core mobile technology and has 1.918GB RAM.

7.1 Case Study A: The Newsvendor Problem

7.1.1 Description

In section 3.6 we discuss a very simple stochastic programming problem to demonstrate the linear decision rules approximation. We remind the reader that the *newsvendor problem* is centered around a newspaper vendor who faces the dilemma of deciding how many newspapers to order from an external supplier today. The element of uncertainty is characterised by the non-determinism of the customers' demand for newspapers tomorrow.

7.1.2 JADA Formulation Using Explicit Syntax

In listing 7.1, we illustrate our specification of the *newsvendor problem* using our input format, JADA.

Listing 7.1: A formulation of the *newsvendor problem*.

```
Model
{
  General
  {
    name("Newsvendor Problem");
    stages(2);
```

```
  }

  Variables
  {
    random(demand,2,5:10);
    decision(x,1,1);
    decision(w,2,1);
  }

  Support
  {
    5 <= demand; demand <= 10;
  }
  Samples{file("C:Workspace/JADA/tests/examples/newsvendorsamples.txt");}

  Constraints
  {
    w + x >= 0;
    w >= -demand;
    x >= 0;
  }

  Objective{minimise expectation 5*x + 10*w;}
}
```

The sample data file required for this problem is produce by randomly generating numbers between 5 and 10 using *inversion transform sampling*.

7.1.3 Generated Matrices

We mention in our discussion of the implementation that we abstract the generation of the matrices, as required by eqs. (Cons-MSP$_{fixed}$) and (Prog-MSP$_{fixed}$) into the `generator` module. In this section we list the components generated by the matrices generator classes.

Decision Rules

In this section we present the decision rule matrices generated by the `DecisionRuleMatricesGenerator` for both the *conservative* and *progressive* linear program (see table 7.1).

Decision Costs

The costs matrices generated by the `CostsMatricesGenerator` are given in table 7.2.

$C_1 \in \mathbb{R}^{1\times1}$	5.0
$C_2 \in \mathbb{R}^{1\times2}$	$\begin{pmatrix} 10.0 & 0.0 \end{pmatrix}$

Table 7.2: Generated costs matrices for the *conservative* and *progressive* approximations.

$X_1 \in \mathbb{R}^{1\times 1}$	$x_{1,1,1}$
$X_2 \in \mathbb{R}^{1\times 2}$	$\begin{pmatrix} x_{2,1,1} & x_{2,1,2} \end{pmatrix}$
$\Lambda_1 \in \mathbb{R}^{1\times 6}$	$\begin{pmatrix} \Lambda_{1,1,1} & \Lambda_{1,1,2} & \Lambda_{1,1,3} & \Lambda_{1,1,4} & \Lambda_{1,1,5} & \Lambda_{1,1,6} \end{pmatrix}$
$\Lambda_2 \in \mathbb{R}^{2\times 6}$	$\begin{pmatrix} \Lambda_{1,1,1} & \Lambda_{1,1,2} & \Lambda_{1,1,3} & \Lambda_{1,1,4} & \Lambda_{1,1,5} & \Lambda_{2,1,6} \\ \Lambda_{2,2,1} & \Lambda_{2,2,2} & \Lambda_{2,2,3} & \Lambda_{2,2,4} & \Lambda_{2,2,5} & \Lambda_{2,2,6} \end{pmatrix}$
$S_1 \in \mathbb{R}^{1\times 1}$	$s_{1,1,1}$
$S_2 \in \mathbb{R}^{2\times 2}$	$\begin{pmatrix} s_{2,1,1} & s_{2,1,2} \\ s_{2,2,1} & s_{2,2,2} \end{pmatrix}$

Table 7.1: Generated decision rule matrices for the *conservative* and *progressive* approximations.

Recourse Constraints

The post-processing of the parsed recourse constraints involves generation of the matrices $\{A_{t,s}\}_{s=1,t=1}^{2}$ and $\{B_t\}_{t=1}^{2}$. These components represent the derived parameters for the generalised inequality constraint $\mathbb{E}\left[\sum_{s=T}^{2} A_{t,s} x_s(\xi^{s})\right] \leq b_t(\xi^{t})$. In table 7.3 we illustrate the values of these matrices, as derived by the `ConstraintsMatricesGenerator`, for this optimisation problem.

$A_{1,1} \in \mathbb{R}^{1\times 1}$	-1.0
$A_{1,2} \in \mathbb{R}^{1\times 1}$	0.0
$A_{2,1} \in \mathbb{R}^{2\times 1}$	$\begin{pmatrix} -1.0 & 0.0 \end{pmatrix}^{\mathsf{T}}$
$A_{2,2} \in \mathbb{R}^{2\times 1}$	$\begin{pmatrix} -1.0 & -1.0 \end{pmatrix}^{\mathsf{T}}$

$B_1 \in \mathbb{R}^{1\times 1}$	0.0
$B_2 \in \mathbb{R}^{2\times 2}$	$\begin{pmatrix} 0.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix}$

Table 7.3: Generated matrices for the recourse constraints to be utilised by the *conservative* and *progressive* approximations.

Support Constraints

The matrices W and h are defined through eq. (2.3.3.4), and are derived by processing the declared support constraints and the bound parameters specified for the random variables. Their values, as generated by the `SupportMatricesGenerator`, are stated in table 7.4.

$W \in \mathbb{R}^{6\times 2}$	$\begin{pmatrix} 1.0 & -1.0 & 10.0 & -5.0 & 10.0 & -5.0 \\ 0.0 & 0.0 & -1.0 & 1.0 & -1.0 & 1.0 \end{pmatrix}^{\mathsf{T}}$
$h \in \mathbb{R}^{6\times 1}$	$\begin{pmatrix} 1.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}^{\mathsf{T}}$

Table 7.4: Generated support matrices for the *conservative* and *progressive* approximations.

Moments

Finally, in table 7.5 we give the second-order moments M and conditional moments matrices $\{M_t\}_{t=1}^2$ generated by the `MomentMatricesGenerator`.

$M_1 \in \mathbb{R}^{2\times 1}$	$\begin{pmatrix} 1.0 & 7.4628 \end{pmatrix}^{\mathsf{T}}$
$M_2 \in \mathbb{R}^{2\times 2}$	$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix}$

$M \in \mathbb{R}^{2\times 2}$	$\begin{pmatrix} 1.0 & 7.4628 \\ 7.4628 & 57.8368 \end{pmatrix}$

Table 7.5: Generated moments matrices to be used by the *conservative* and *progressive* approximations.

7.1.4 Generated Linear Programs

Using the generated matrix components (see section 7.1.3), the classes in the `approximator` module are able to generate the conservative (see listing 7.2) and progressive (see listing 7.3) approximations of the original stochastic program.

Listing 7.2: The generated *conservative* approximation linear program for the newsvendor problem.

```
Minimise 5.0x_{1,1,1} + 10.0x_{2,1,1} + 74.62809116000001x_{2,1,2}

Subject to
    C1:  Λ_{1,1,1} - Λ_{1,1,2} + 10.0Λ_{1,1,3} -  5.0Λ_{1,1,4} + 10.0Λ_{1,1,5} - 5.0Λ_{1,1,6} - x_{1,1,1} = 0
    C2: -Λ_{1,1,3} + Λ_{1,1,4} - Λ_{1,1,5} + Λ_{1,1,6} = 0
    C3:  Λ_{2,1,1} - Λ_{2,1,2} + 10.0Λ_{2,1,3} - 5.0Λ_{2,1,4} + 10.0Λ_{2,1,5} - 5.0Λ_{2,1,6} - x_{1,1,1} -
        x_{2,1,1} = 0
    C4: -Λ_{2,1,3} + Λ_{2,1,4} - Λ_{2,1,5} + Λ_{2,1,6} - x_{2,1,2} = 0
    C5:  Λ_{2,2,1} - Λ_{2,2,2} + 10.0Λ_{2,2,3} - 5.0Λ_{2,2,4} + 10.0Λ_{2,2,5} - 5.0Λ_{2,2,6} - x_{2,1,1} = 0
    C6: -Λ_{2,2,3} + Λ_{2,2,4} - Λ_{2,2,5} + Λ_{2,2,6} - x_{2,1,2} = 1
    C7:  Λ_{1,1,1} - Λ_{1,1,2} >= 0
    C8:  Λ_{2,1,1} - Λ_{2,1,2} >= 0
    C9:  Λ_{2,2,1} - Λ_{2,2,2} >= 0

Decision Bounds
```

```
      0   <=  Λ_{1,1,1}  <=  Inf     0   <=  Λ_{1,1,2}  <=  Inf     0   <=  Λ_{1,1,3}  <=  Inf
      0   <=  Λ_{1,1,4}  <=  Inf     0   <=  Λ_{1,1,5}  <=  Inf     0   <=  Λ_{1,1,6}  <=  Inf
      0   <=  Λ_{2,1,1}  <=  Inf     0   <=  Λ_{2,1,2}  <=  Inf     0   <=  Λ_{2,1,3}  <=  Inf
      0   <=  Λ_{2,1,4}  <=  Inf     0   <=  Λ_{2,1,5}  <=  Inf     0   <=  Λ_{2,1,6}  <=  Inf
      0   <=  Λ_{2,2,1}  <=  Inf     0   <=  Λ_{2,2,2}  <=  Inf     0   <=  Λ_{2,2,3}  <=  Inf
      0   <=  Λ_{2,2,4}  <=  Inf     0   <=  Λ_{2,2,5}  <=  Inf     0   <=  Λ_{2,2,6}  <=  Inf

    -Inf  <=  x_{1,1,1}  <=  Inf
    -Inf  <=  x_{2,1,1}  <=  Inf
    -Inf  <=  x_{2,1,2}  <=  Inf

End
```

Listing 7.3: The generated *progressive* approximation linear program for the newsvendor problem.

```
Minimise 5.0x_{1,1,1} + 10.0x_{2,1,1} + 74.62809116000001x_{2,1,2}

Subject to
     C1:  s_{1,1,1} - x_{1,1,1} = 0
     C2:  s_{2,1,1} - x_{1,1,1} - x_{2,1,1} = 0
     C3:  s_{2,1,2} - x_{2,1,2} = 0
     C4:  s_{2,2,1} - x_{2,1,1} = 0
     C5:  s_{2,2,2} - x_{2,1,2} = 1
     C6:  2.5371908839999993s_{1,1,1} >= 0
     C7:  2.4628091160000007s_{1,1,1} >= 0
     C8:  2.5371908839999993s_{1,1,1} >= 0
     C9:  2.4628091160000007s_{1,1,1} >= 0
     C10: 2.5371908839999993s_{2,1,1} + 16.791328145719817s_{2,1,2} >= 0
     C11: 2.5371908839999993s_{2,2,1} + 16.791328145719817s_{2,2,2} >= 0
     C12: 2.4628091160000007s_{2,1,1} + 20.52271743428019s_{2,1,2} >= 0
     C13: 2.4628091160000007s_{2,2,1} + 20.52271743428019s_{2,2,2} >= 0
     C14: 2.5371908839999993s_{2,1,1} + 16.791328145719817s_{2,1,2} >= 0
     C15: 2.5371908839999993s_{2,2,1} + 16.791328145719817s_{2,2,2} >= 0
     C16: 2.4628091160000007s_{2,1,1} + 20.52271743428019s_{2,1,2} >= 0
     C17: 2.4628091160000007s_{2,2,1} + 20.52271743428019s_{2,2,2} >= 0
     C18: s_{1,1,1} >= 0
     C19: s_{2,1,1} + 7.462809116000001s_{2,1,2} >= 0
     C20: s_{2,2,1} + 7.462809116000001s_{2,2,2} >= 0

Decision Bounds
      0   <=  s_{1,1,1}  <=  Inf
      0   <=  s_{2,1,1}  <=  Inf
      0   <=  s_{2,1,1}  <=  Inf

    -Inf  <=  x_{1,1,1}  <=  Inf
    -Inf  <=  x_{2,1,1}  <=  Inf
    -Inf  <=  x_{2,1,2}  <=  Inf

End
```

7.1.5 Computed Solutions

The `approximator` module interfaces the YALMIP convex programming framework. JADA is able to communicate instances of the generated approximation linear programs by passing two parameters, which respectively represent the objective function and the constraints.

```
< Optimal Objective >
     -25.0
< Optimal Objective />

< Decision Variables >
     < Stage='1' >
          x_1_1_1 = 5.000000e+000
     < Stage />
     < Stage='2' >
          x_2_1_1 = -5.000000e+000
          x_2_1_2 = 5.178669e-016
     < Stage />
< Decision Variables />

< Slack Variables >
     < Stage='1' >
          lambda_1_1_1 = 1.923077e-001
          lambda_1_1_2 = -5.496885e-017
          lambda_1_1_3 = 4.807692e-001
          lambda_1_1_4 = 4.807692e-001
          lambda_1_1_5 = 4.807692e-001
          lambda_1_1_6 = 4.807692e-001
     < Stage />
     < Stage='2' >
          lambda_2_1_1 = 7.236667e-017
          lambda_2_1_2 = 1.128462e-016
          lambda_2_1_3 = 2.205772e-019
          lambda_2_1_4 = -1.648091e-016
          lambda_2_1_5 = -3.980097e-017
          lambda_2_1_6 = -9.453507e-017
          lambda_2_2_1 = -3.953631e-016
          lambda_2_2_2 = 5.399349e-015
          lambda_2_2_3 = -1.110223e-016
          lambda_2_2_4 = 1.000000e+000
          lambda_2_2_5 = -5.690998e-017
          lambda_2_2_6 = -1.572382e-015
     < Stage />
< Slack Variables />
```

Figure 7.1: Computed solutions to the newsvendor problem's *conservative* LP.

```
< Optimal Objective >
     -32.9795738681331
< Optimal Objective />

< Decision Variables >
     < Stage='1' >
          x_1_1_1 = 8.107933e+000
     < Stage />
     < Stage='2' >
          x_2_1_1 = -8.687333e-001
          x_2_1_2 = -8.687333e-001
     < Stage />
< Decision Variables />

< Slack Variables >
     < Stage='1' >
        s_1_1_1 = 8.107933e+000
     < Stage />
     < Stage='2' >
        s_2_1_1 = 7.239200e+000
        s_2_1_2 = -8.687333e-001
        s_2_2_1 = -8.687333e-001
        s_2_2_2 = 1.312667e-001
     < Stage />
< Slack Variables />
```

Figure 7.2: Computed solutions to the newsvendor problem's *progressive* LP.

For this case-study, the solutions and interpreted optimal decisions rules to the linear programs displayed in listings 7.2 and 7.3 are given in figs. 7.1 to 7.3.

```
< Optimal Decisions >
     < Stage='1' >
          x = 5.0
     < Stage />
     < Stage='2' >
          w = -5.00000000000001 +
              5.17866944988629E-016*demand
     < Stage />
< Optimal Decisions />
```

Figure 7.3: Interpreted optimal decisions rules for the newsvendor problem's *conservative* LP.

```
< Optimal Decisions >
     < Stage='1' >
          x = 8.1079334294196
     < Stage />
     < Stage='2' >
          w = -0.868733301289211 -
              0.868733301289211*demand
     < Stage />
< Optimal Decisions />
```

Figure 7.4: Interpreted optimal decisions rules for the newsvendor problem's *progressive* LP.

7.1.6 Loss of Optimality

We previously stated in section 3.5 that the solutions computed from using linear decision rules are rarely truly optimal. This is a by-product of restricting the feasible region to those policies that are affinely dependent on the uncertain elements. Since the optimal decision rule may not be linear in the random variables, we may incur losses in optimality from performing this drastic reduction of the policy space.

	Conservative Approximation (UB)	Progressive Approximation Approximation (LB)	Percentage Gap $\frac{UB-LB}{LB}$ (%)
Objective Value	-25.0	-32.97957387	-24.196
x_1_1_1	5.0	8.107933	n/a
x_2_1_1	-5.0	-0.8687333	n/a
x_2_1_2	5.178669E-16	-0.8687333	n/a

Table 7.6: Loss of optimality incurred by using linear decision rule for the newsvendor problem.

The conservative and progressive approximations provide the upper and lower bounds for the actual optimal decision rules, and we can use these values to estimate the loss of optimality incurred as a result of enforcing computational tractability. The results of the conservative approximation suggest that the newsvendor should purchase 5.0 units of newspapers from the external vendor today, and then sell 5 units tomorrow. Unfortunately, we cannot numerically compare the decision rules computed by the conservative and progressive approximation since the progressive decision rules are only constrained in expectation. However, we can estimate the percentage gap between the objectives value computed by the conservative and progressive approximations as -24.196%, which is quite significant. Hence, it might be too optimistic of the decision-maker to select the upper bound decision rule.

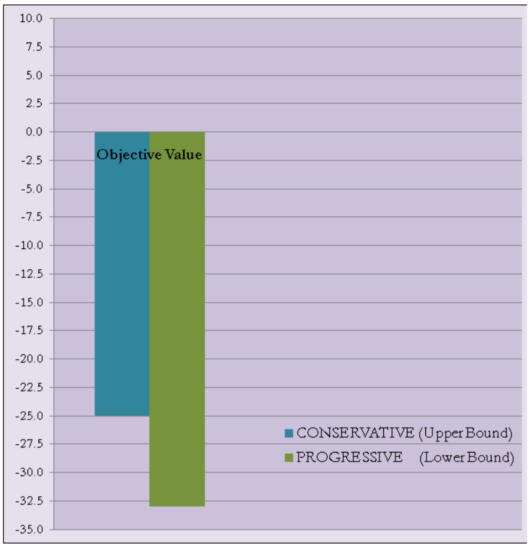


Figure 7.5: Linear decision rule-based bounds for the *newsvendor problem*

7.2 Case Study B: Capacity Expansion for an Electricity Power Plant

7.2.1 Description

Stochastic programming has manifold applications such as public infrastructure investment planning. As an example, we study an adaptation of a capacity expansion model, proposed by Kuhn et al[58][57], for a power system comprising of generators, regional consumers, and transmission lines such that future regional demand and energy production costs are uncertain. The problem is described by a multi-stage stochastic programming problem consisting of two-stages. In the first stage, the capacity of the current infrastructure is expanded, which yields investment costs. During the second stage, operational costs are incurred following the execution of the upgraded electric power system. The improved system is required to satisfy the total demand across all the regions.

Consequently, the objective is to minimise the expected expansion expenditures and operational costs. To formalise the description of the problem, we consider that the power grid consists of $R = \{1, \cdots, \bar{r}\}$, regions that depend on the electricity supply. Additionally, we suppose that there are $G = \{1, \cdots, \bar{g}\}$ electricity generators constrained by a production capacity of $\bar{x}_g$, and $L = \{1, \cdots, \bar{\ell}\}$ electricity transmission lines, which can carry up to $\bar{u}_\ell$ units of electricity.

Each electricity generator is allocated to one of regions $r \in R$, and the group of generators supplying the electricity to region r is denoted by the relation $G(r) \subset G$. The flow of electricity for region r is represented by the relations $L(r, in) \subset L$ and $L(r, out) \subset L$, which respectively symbolise the set of transmission lines carrying electricity into and out of the region. The sum of the number of power plants $y_g, g \in G(r)$, number of electricity transmission lines $t_\ell, \ell \in L(r, in)$ going into region r, and the number of electricity transmission lines $t_\ell, \ell \in L(r, out)$ going out of region r must at least meet the stochastic demand for electricity to achieve nodal load balance.

We use the quantities c'_g and c''_ℓ to respectively denote the cost of expanding power plant y_g by an amount α_g, and the cost of expanding transmission line t_ℓ by an amount β_ℓ. The first-stage decisions α_g and β_ℓ are defined such that the capacity expansion of the generators and electricity transmission can at most double. The expected operational costs of power plant p_g and the unknown electricity demand for region r are respectively represented by the stochastic elements ξ'_g and ξ''_r. Thus, the total cost of the first-stage capacity expansion is $\sum_{g\in G} c'_g\alpha_g + \sum_{\ell\in L} c''_\ell\beta_\ell$, and the expected total operational cost for the second stage is $\sum_{g\in G}\mathbb{E}\left[\xi'_g y_g\right]$.

Using the aforementioned notation, the decision-making problem can be specified by eq. (7.2.1.1).

$$
\begin{array}{ll}
\textit{minimise} & \sum_{g\in G} c'_g\alpha_g + \sum_{\ell\in L} c''_\ell\beta_\ell + \mathbb{E}\left[\sum_{g\in G}\xi'_g y_g\right] \\
& \left.\begin{array}{ll}
\alpha_g \in \mathbb{R},\ \beta_\ell \in \mathbb{R},\ y_g \in \mathcal{L}^2_{k,k},\ t_\ell \in \mathcal{L}^2_{k,k} & \\
\textit{subject to}\quad 1 \le \alpha_g \le 2 & \forall g \in G \\
1 \le \beta_\ell \le 2 & \forall \ell \in L \\
0 \le y_g \le \bar{u}_g\alpha_g & \forall g \in G \\
|t_\ell| \le \bar{u}_\ell\beta_\ell & \forall \ell \in L \\
\sum_{g\in G} p_g + \sum_{\ell\in L(r,in)} t_\ell - \sum_{\ell\in L(r,out)} t_\ell \ge \xi''_r & \forall r \in R
\end{array}\right\}\ \mathbb{P}-a.s.
\end{array}
\tag{7.2.1.1}
$$

JADA Formulation Using Stochastic Processes

For purposes of experimentation, we first contextualise eq. (7.2.1.1) with numerical values. We begin by assuming that the power grid consists of five regions $R = 1, 2, 3, 4, 5$ such that the stochastic regional demands are

- $\xi''_1 = \hat{\xi}''_1$,
- $\xi''_2 = 30 + 1.2\hat{\xi}''_2$,
- $\xi''_3 = 30 + 1.4\hat{\xi}''_3$,
- $\xi''_4 = 30 + 1.6\hat{\xi}''_4$, and
- $\xi''_3 = 30 + 1.8\hat{\xi}''_5$,

where $\forall r \in R, \hat{\xi}''_r \in [0, 120]$. Secondly, we assume that there are three electricity generators $G = 1, 2, 3$ and thus three power plants y_1, y_2, y_3. The power plants incur respective uncertain operational costs of

- $\xi'_1 = 20 + \hat{\xi}'_2$,
- $\xi'_2 = 100 + \hat{\xi}'_3$, and
- $\xi'_3 = 20 + \hat{\xi}'_2$,

where $\xi_1' \in [0, 80]$, $\xi_2' \in [0, 80]$, and $\xi_3' \in [0, 100]$. Additionally, we suppose that

- the cost of expanding the capacity of power plant y_1 by α_1 is $c_1' = 100$,
- the cost of expanding the capacity of power plant y_2 by α_2 is $c_2' = 40$, and
- the cost of expanding the capacity of power plant y_3 by α_3 is $c_3' = 150$.

We also define costs of expanding the capacity of five transmission lines t_1, t_2, t_3, t_4, t_5 by amounts $\beta, \beta_2, \beta_3, \beta_4, \beta_5$ to be $c_1'' = 500$, $c_2'' = 20$, $c_3'' = 400$, $c_4'' = 60$, $c_5'' = 10$ respectively.

Furthermore, we impose the maximum production capacity of the generators to be $\bar{u}_g = 350$, and the maximum capacity of the transmission line to be $\bar{u}_\ell = 350$.

Finally, to model the flow of electricity, we require that

$$G(r) = \begin{cases} \{3\}, & r = 1 \\ \emptyset, & r = 2 \\ \{2\}, & r = 3 \\ \emptyset, & r = 4 \\ \{1\}, & r = 5 \end{cases} \quad L(r, in) = \begin{cases} \{1\}, & r = 1 \\ \{2,4\}, & r = 2 \\ \emptyset, & r = 3 \\ \{3,5\}, & r = 4 \\ \emptyset, & r = 5 \end{cases} \quad L(r, out) = \begin{cases} \{2\}, & r = 1 \\ \emptyset, & r = 2 \\ \{1,3\}, & r = 3 \\ \{4\}, & r = 4 \\ \{5\}, & r = 5 \end{cases}$$

Thus, we can formulate the program in our standardised input format by the description given in listing 7.4. The numerical data the sample file required for this problem are similarly generated using the *inversion sampling* method.

Listing 7.4: A formulation of the *electricity power plant capacity expansion problem.*

```
Model
{
     General
     {
          name("Electricity Power Plant Expansion");
          stages(2);

          constant(line_capacity,         350);
          constant(generator_capacity,    350);
          constant(line_expansion_cost,   500, 20, 400, 60, 10);
          constant(plant_expansion_cost,  100, 40, 150);
          constant(default_op_cost,        20, 20, 100);
     }

     Variables
     {
          random(demand_1,2,0:120);
          random(demand_2,2,0:120);
          random(demand_3,2,0:120);
          random(demand_4,2,0:120);
          random(demand_5,2,0:120);

          random(op_cost_1,2,0:80);
```

```
            random(op_cost_2,2,0:80);
            random(op_cost_3,2,0:100);

            decision(plant_expansion,1,3);
            decision(line_expansion,1,5);

            decision(plant,2,3);
            decision(line,2,5);
      }

      Support{}

      Samples
      {
            file("C:/Workspace/JADA/tests/examples/electricitysamples.txt");
      }

      Constraints
      {
            forall(i=1:5)(line_expansion#i#1 >= 1);
            forall(i=1:5)(line_expansion#i#1 <= 2);

            forall(i=1:3)(plant_expansion#i#1 >= 1);
            forall(i=1:3)(plant_expansion#i#1 <= 2);

            forall(i=1:3)(plant#i#2 >= 0);
            forall(i=1:3)(plant#i#2 <= generator_capacity*plant_expansion#i#1);

            forall(i=1:5)(-line#i#2 <= line_capacity*line_expansion#i#1);
            forall(i=1:5)(+line#i#2 <= line_capacity*line_expansion#i#1);

            plant#3#2 - line#2#2 + line#1#2 = demand_1;
            line#2#2  + line#4#2 = 30 + 1.2*demand_2;
            plant#2#2 - line#1#2 - line#3#2 = 30 + 1.4*demand_3;
            line#3#2  + line#5#2 - line#4#2 = 30 + 1.6*demand_4;
            plant#1#2 - line#5#2 = 30 + 1.8*demand_5;
      }

      Objective
      {
            minimise expectation
                  sum(i=1:3)(plant_expansion_cost#i*plant_expansion#i#1) +
                  sum(i=1:3)(line_expansion_cost#i*line_expansion#i#1) +
                  sum(i=1:3)(plant#i#1*default_op_cost#i) +
                  plant#1#2*op_cost_1 +
                  plant#2#2*op_cost_2 +
                  plant#3#2*op_cost_3 ;
      }

}
```

7.2.2 Generated Matrices

In this section we state the matrix components derived by the classes in the `generator` package. We point out to the reader that this problem is significantly more complex and larger than the *newsvendor problem*, which necessitates some mathematical abbreviation in our presentation of the matrices.

Decision Rules

The decision rule matrices generated by the `DecisionRuleMatricesGenerator` for both the *conservative* LP and *progressive* LP are illustrated in table 7.7.

$X_1 \in \mathbb{R}^{8\times 1}$	$\begin{pmatrix} x_{1,1,1} & x_{1,1,2} & \cdots & x_{1,1,8} \end{pmatrix}^\mathsf{T}$	$S_1 \in \mathbb{R}^{16\times 1}$	$\begin{pmatrix} s_{1,1,1} & s_{1,1,2} & \cdots & s_{1,1,16} \end{pmatrix}^\mathsf{T}$
$X_2 \in \mathbb{R}^{8\times 9}$	$\begin{pmatrix} x_{2,1,1} & x_{2,1,2} & \cdots & x_{2,1,9} \\ x_{2,2,1} & x_{2,2,2} & \cdots & x_{2,2,9} \\ \vdots & \vdots & \ddots & \vdots \\ x_{2,8,1} & x_{2,8,2} & \cdots & x_{2,8,9} \end{pmatrix}$	$S_2 \in \mathbb{R}^{26\times 9}$	$\begin{pmatrix} s_{2,1,1} & s_{2,1,2} & \cdots & s_{2,1,9} \\ s_{2,2,1} & s_{2,2,2} & \cdots & s_{2,2,9} \\ \vdots & \vdots & \ddots & \vdots \\ s_{2,26,1} & s_{2,26,2} & \cdots & s_{2,26,9} \end{pmatrix}$

$\Lambda_1 \in \mathbb{R}^{16\times 18}$	$\begin{pmatrix} \Lambda_{1,1,1} & \Lambda_{1,1,2} & \cdots & \Lambda_{1,1,18} \\ \Lambda_{1,2,1} & \Lambda_{1,2,2} & \cdots & \Lambda_{1,2,18} \\ \vdots & \vdots & \ddots & \vdots \\ \Lambda_{1,16,1} & \Lambda_{1,16,2} & \cdots & \Lambda_{1,16,18} \end{pmatrix}$
$\Lambda_2 \in \mathbb{R}^{26\times 18}$	$\begin{pmatrix} \Lambda_{2,1,1} & \Lambda_{2,1,2} & \cdots & \Lambda_{2,1,18} \\ \Lambda_{2,2,1} & \Lambda_{2,2,2} & \cdots & \Lambda_{2,2,18} \\ \vdots & \vdots & \ddots & \vdots \\ \Lambda_{2,26,1} & \Lambda_{2,26,2} & \cdots & \Lambda_{2,26,18} \end{pmatrix}$

Table 7.7: Generated decision rule matrices for the *conservative* and *progressive* approximations.

Decision Costs

The costs matrices generated by the `CostsMatricesGenerator` are given in table 7.8.

$C_1 \in \mathbb{R}^{8\times 1}$	$\begin{pmatrix} 500.0 & 20.0 & 400.0 & 0.0 & 0.0 & 100.0 & 40.0 & 150.0 \end{pmatrix}^\mathsf{T}$	$C_2 \in \mathbb{R}^{8\times 9}$	$\begin{pmatrix} \mathbf{0} \in \mathbb{R}^{5\times 6} & \mathbf{0} \in \mathbb{R}^{5\times 3} \\ \mathbf{0} \in \mathbb{R}^{3\times 6} & \mathbf{I}_3 \end{pmatrix}$

Table 7.8: Generated costs matrices for the *conservative* and *progressive* approximations.

Recourse Constraints

We repeat for emphasis that the MATLAB class `ConstraintsMatricesGenerator` is solely responsible for generating the recourse matrices $\{A_{t,s}\}_{s=1,t=1}^{2}$ and $\{B_t\}_{t=1}^{2}$. We tabulate their values for the capacity expansion optimisation problem in table 7.9.

Support Constraints

The support matrices W and h, as generated by the `SupportMatricesGenerator`, are given in table 7.10.

Moments

In table 7.11 we provide the second-order moments M and conditional moments matrices $\{M_t\}_{t=1}^{2}$ generated by the `MomentMatricesGenerator`.

$M_1 \in \mathbb{R}^{9\times 1}$	$\begin{pmatrix} 1.0000 & 60.3017 & 60.1545 & 59.8222 & 59.5402 & 60.1717 & 40.1614 & 39.9090 & 50.2953 \end{pmatrix}^\mathsf{T}$
$M_2 \in \mathbb{R}^{9\times 9}$	$\mathbf{I}_9$
$M \in \mathbb{R}^{2\times 2}$	$\begin{pmatrix} 1.0000 & 60.3017 & 60.1545 & 59.8222 & 59.5402 & 60.1717 & 40.1614 & 39.9090 & 50.2953 \\ 60.3017 & 4821.4002 & 3627.4194 & 3607.3796 & 3590.3749 & 3628.4561 & 2421.7972 & 2406.5777 & 3032.8886 \\ 60.1545 & 3627.4194 & 4829.908 & 3598.5785 & 3581.6154 & 3619.6036 & 2415.8886 & 2400.7063 & 3025.4892 \\ 59.8222 & 3607.3796 & 3598.5785 & 4787.373 & 3561.8286 & 3599.607 & 2402.542 & 2387.4435 & 3008.7747 \\ 59.5402 & 3590.3749 & 3581.6154 & 3561.8286 & 4764.7501 & 3582.639 & 2391.2168 & 2376.1895 & 2994.5919 \\ 60.1717 & 3628.4561 & 3619.6036 & 3599.607 & 3582.639 & 4831.3749 & 2416.5791 & 2401.3924 & 3026.3538 \\ 40.1614 & 2421.7972 & 2415.8886 & 2402.542 & 2391.2168 & 2416.5791 & 2144.4994 & 1602.7989 & 2019.9267 \\ 39.9090 & 2406.5777 & 2400.7063 & 2387.4435 & 2376.1895 & 2401.3924 & 1602.7989 & 2129.0338 & 2007.2327 \\ 50.2953 & 3032.8886 & 3025.4892 & 3008.7747 & 2994.5919 & 3026.3538 & 2019.9267 & 2007.2327 & 3361.9614 \end{pmatrix}$

Table 7.11: Generated moments matrices to be used by the *conservative* and *progressive* approximations.

7.2.3 Computed Solutions

The solutions obtained via the YALMIP framework are shown in figs. 7.6 and 7.8. In figs. 7.7 and 7.9 we present the interpreted optimal decisions rules to the approximation linear programs for the electricity capacity expansion problem.

```
< Optimal Objective >
    25772.3478372735
< Optimal Objective />

< Decision Variables >
    < Stage='1' >
        x_1_1_1 = 1.000000e+0000          x_1_2_1 = 1.000000e+000
        x_1_3_1 = 1.000000e+000           x_1_4_1 = 1.000000e+000
        x_1_5_1 = 1.000000e+000           x_1_6_1 = 1.000000e+000
        x_1_7_1 = 1.000000e+000           x_1_8_1 = 1.588571e+000
    < Stage />
    < Stage='2' >
        x_2_1_1 = -2.000000e+002          x_2_1_2 = -2.971647e-015
        x_2_1_3 = 1.200000e+000           x_2_1_4 = -1.250000e+000
        x_2_1_5 = -2.769181e-014          x_2_1_6 = -1.512108e-015
        x_2_1_7 = -1.653449e-013          x_2_1_8 = 6.438113e-014
        x_2_1_9 = -1.000142e-014          x_2_2_1 = -2.000000e+002
        x_2_2_2 = -1.000000e+000          x_2_2_3 = 1.200000e+000
        x_2_2_4 = -1.904012e-014          x_2_2_5 = 6.833333e-001
        x_2_2_6 = 2.700000e+000           x_2_2_7 = -5.416062e-015
        x_2_2_8 = 4.170734e-014           x_2_2_9 = 3.521231e-014
        x_2_3_1 = 1.700000e+002           x_2_3_2 = -3.598754e-015
        x_2_3_3 = -9.354732e-015          x_2_3_4 = -2.047225e-014
        x_2_3_5 = -1.529152e-014          x_2_3_6 = 1.000000e-001
        x_2_3_7 = 6.235177e-014           x_2_3_8 = -3.150353e-014
        x_2_3_9 = -1.171583e-014          x_2_4_1 = 2.300000e+002
        x_2_4_2 = 1.000000e+000           x_2_4_3 = -1.403247e-014
        x_2_4_4 = 7.072075e-015           x_2_4_5 = -6.833333e-001
        x_2_4_6 = -2.700000e+000          x_2_4_7 = 8.002106e-015
        x_2_4_8 = -5.272765e-014          x_2_4_9 = -3.927626e-014
        x_2_5_1 = 9.000000e+001           x_2_5_2 = 1.000000e+000
        x_2_5_3 = 3.433845e-014           x_2_5_4 = -4.463825e-016
        x_2_5_5 = 9.166667e-001           x_2_5_6 = -2.800000e+000
        x_2_5_7 = -2.667339e-014          x_2_5_8 = -4.261779e-014
        x_2_5_9 = -1.832792e-014          x_2_6_1 = 1.200000e+002
        x_2_6_2 = 1.000000e+000           x_2_6_3 = 4.960989e-014
        x_2_6_4 = 4.149621e-014           x_2_6_5 = 9.166667e-001
        x_2_6_6 = -1.000000e+000          x_2_6_7 = -4.367358e-014
        x_2_6_8 = 5.208795e-015           x_2_6_9 = -2.914600e-014
        x_2_7_1 = 9.984410e-012           x_2_7_2 = -7.019332e-015
        x_2_7_3 = 1.200000e+000           x_2_7_4 = 1.500000e-001
        x_2_7_5 = -2.456491e-016          x_2_7_6 = 1.000000e-001
        x_2_7_7 = -4.494931e-014          x_2_7_8 = 6.796603e-014
        x_2_7_9 = -2.667757e-014          x_2_8_1 = -7.725711e-012
        x_2_8_2 = -3.167493e-015          x_2_8_3 = 9.071152e-015
        x_2_8_4 = 1.250000e+000           x_2_8_5 = 6.833333e-001
        x_2_8_6 = 2.700000e+000           x_2_8_7 = 1.799048e-013
        x_2_8_8 = -6.421585e-016          x_2_8_9 = 2.352973e-014
    < Stage />
< Decision Variables />

< Slack Variables >
        ...
< Slack Variables />
```

Figure 7.6: Computed solutions to the *conservative* linear program for the electricity capacity expansion model.

```
< Optimal Decisions >
    < Stage='1' >
        line_expansion#1#1  = 0.999999999985026
        line_expansion#2#1  = 1.00000000000404
        line_expansion#3#1  = 0.99999999999651
        line_expansion#4#1  = 0.999999999988034
        line_expansion#5#1  = 0.99999999999162
        plant_expansion#1#1 = 0.999999999999997
        plant_expansion#2#1 = 0.999999999992891
        plant_expansion#3#1 = 1.58857142856044
< Stage />
< Stage='2' >
        line#1#2 =  -199.999999995828 - 2.97164688708029E-015*demand_1 +
                    1.19999999999998*demand_2 - 1.24999999999096*demand_3 -
                    2.76918118518227E-014*demand_4 - 1.51210766305398E-015*demand_5 -
                    1.65344938538876E-013*op_cost_1 + 6.43811323763965E-014*op_cost_2 -
                    1.00014201888485E-014*op_cost_3
        line#2#2 =  -199.999999995834 - 0.999999999999979*demand_1 +
                    1.19999999999999*demand_2 - 1.90401205587901E-014*demand_3 +
                    0.683333333333035*demand_4 + 2.69999999997741*demand_5 -
                    5.41606173904003E-015*op_cost_1 + 4.17073405506132E-014*op_cost_2 +
                    3.52123107909882E-014*op_cost_3
        line#3#2 =  169.999999995851 - 3.59875387316148E-015*demand_1 -
                    9.35473248342136E-015*demand_2 - 2.04722523658326E-014*demand_3 -
                    1.52915225192612E-014*demand_4 + 0.100000000022777*demand_5 +
                    6.23517691950487E-014*op_cost_1 - 3.15035298291395E-014*op_cost_2 -
                    1.17158302262111E-014*op_cost_3
        line#4#2 =  229.999999995824 + 0.999999999999994*demand_1 -
                    1.40324662485488E-014*demand_2 + 7.07207482780756E-015*demand_3 -
                    0.683333333332958*demand_4 - 2.69999999997744*demand_5 +
                    8.00210551188517E-015*op_cost_1 - 5.27276463830384E-014*op_cost_2 -
                    3.92762587950921E-014*op_cost_3
        line#5#2 =  89.9999999999728 + 0.999999999999969*demand_1 + 3.43384459301636E-014*demand_2 -
                    4.46382454598795E-016*demand_3 + 0.916666666667051*demand_4 -
                    2.79999999999989*demand_5 - 2.66733897049477E-014*op_cost_1 -
                    4.26177864307544E-014*op_cost_2 - 1.83279153683046E-014*op_cost_3
        plant#1#2 = 119.999999999948 + 0.999999999999993*demand_1 +
                    4.9609893258212E-014*demand_2 + 4.1496208083337E-014*demand_3 +
                    0.916666666667045*demand_4 - 0.999999999999552*demand_5 -
                    4.36735803496969E-014*op_cost_1 + 5.20879507970329E-015*op_cost_2 -
                    2.91460033918584E-014*op_cost_3
        plant#2#2 = 9.98440989626027E-012 - 7.01933197017602E-015*demand_1 +
                    1.19999999999996*demand_2 + 0.150000000009056*demand_3 -
                    2.45649134601739E-016*demand_4 + 0.100000000022789*demand_5 -
                    4.4949309925928E-014*op_cost_1 + 6.79660313719804E-014*op_cost_2 -
                    2.66775703185374E-014*op_cost_3
        plant#3#2 = -7.72571103288292E-012 - 3.16749276514095E-015*demand_1 +
                    9.0711521825775E-015*demand_2 + 1.24999999999094*demand_3 +
                    0.683333333333024*demand_4 + 2.69999999997742*demand_5 +
                    1.79904753795269E-013*op_cost_1 - 6.42158533279032E-016*op_cost_2
                    + 2.35297306289402E-014*op_cost_3
< Stage />
< Optimal Decisions />
```

Figure 7.7: Interpreted optimal decisions rules for the *conservative* approximation of the electricity capacity expansion model.

```
< Optimal Objective >
    24529.9855379761
< Optimal Objective />

< Decision Variables >
    < Stage='1' >
        x_1_1_1 = 1.000000e+0000          x_1_2_1 = 1.000000e+000
        x_1_3_1 = 1.000000e+000           x_1_4_1 = 1.000000e+000
        x_1_5_1 = 1.000000e+000           x_1_6_1 = 1.000000e+000
        x_1_7_1 = 1.000000e+000           x_1_8_1 = 1.000000e+000
    < Stage />
    < Stage='2' >
        x_2_1_1 = -6.857143e+000          x_2_1_2 = 2.857143e-001
        x_2_1_3 = 1.371429e-001           x_2_1_4 = -3.600000e-001
        x_2_1_5 = -9.142857e-002          x_2_1_6 = -5.142857e-002
        x_2_1_7 = 3.005970e-011           x_2_1_8 = -1.961303e-011
        x_2_1_9 = 9.450557e-012           x_2_2_1 = 2.742857e+001
        x_2_2_2 = -1.428571e-001          x_2_2_3 = 6.514286e-001
        x_2_2_4 = 4.000000e-002           x_2_2_5 = 3.657143e-001
        x_2_2_6 = 2.057143e-001           x_2_2_7 = 1.930458e-011
        x_2_2_8 = -4.529188e-012          x_2_2_9 = -2.171491e-012
        x_2_3_1 = 1.800000e+001           x_2_3_2 = -8.149550e-012
        x_2_3_3 = 2.400000e-001           x_2_3_4 = -2.800000e-001
        x_2_3_5 = 6.400000e-001           x_2_3_6 = 3.600000e-001
        x_2_3_7 = -1.055264e-011          x_2_3_8 = 6.031963e-012
        x_2_3_9 = 1.212859e-012           x_2_4_1 = 2.571429e+000
        x_2_4_2 = 1.428571e-001           x_2_4_3 = 5.485714e-001
        x_2_4_4 = -4.000000e-002          x_2_4_5 = -3.657143e-001
        x_2_4_6 = -2.057143e-001          x_2_4_7 = -1.823779e-011
        x_2_4_8 = 5.524008e-012           x_2_4_9 = 2.915237e-012
        x_2_5_1 = 1.457143e+001           x_2_5_2 = 1.428571e-001
        x_2_5_3 = 3.085714e-001           x_2_5_4 = 2.400000e-001
        x_2_5_5 = 5.942857e-001           x_2_5_6 = -5.657143e-001
        x_2_5_7 = -8.996728e-012          x_2_5_8 = -1.559486e-012
        x_2_5_9 = 2.373522e-012           x_2_6_1 = 4.457143e+001
        x_2_6_2 = 1.428571e-001           x_2_6_3 = 3.085714e-001
        x_2_6_4 = 2.400000e-001           x_2_6_5 = 5.942857e-001
        x_2_6_6 = 1.234286e+000           x_2_6_7 = -9.953358e-012
        x_2_6_8 = -2.480769e-012          x_2_6_9 = 1.542871e-012
        x_2_7_1 = 4.114286e+001           x_2_7_2 = 2.857143e-001
        x_2_7_3 = 3.771429e-001           x_2_7_4 = 7.600000e-001
        x_2_7_5 = 5.485714e-001           x_2_7_6 = 3.085714e-001
        x_2_7_7 = 1.940364e-011           x_2_7_8 = -1.355301e-011
        x_2_7_9 = 1.060338e-011           x_2_8_1 = 3.428571e+001
        x_2_8_2 = 5.714286e-001           x_2_8_3 = 5.142857e-001
        x_2_8_4 = 4.000000e-001           x_2_8_5 = 4.571429e-001
        x_2_8_6 = 2.571429e-001           x_2_8_7 = -1.077438e-011
        x_2_8_8 = 1.503961e-011           x_2_8_9 = -1.173440e-011
    < Stage />

< Slack Variables >
        ...
< Slack Variables />
```

Figure 7.8: Computed solutions to the *progressive* linear program for the electricity capacity expansion model.

```
< Optimal Decisions >
    < Stage='1' >
        line_expansion#1#1 = 0.999999999999921
        line_expansion#2#1 = 0.999999999999927
        line_expansion#3#1 = 0.999999999999927
        line_expansion#4#1 = 0.99999999999992
        line_expansion#5#1 = 0.999999999999911
        plant_expansion#1#1 = 0.999999999999935
        plant_expansion#2#1 = 0.999999999999932
        plant_expansion#3#1 = 0.999999999999919
< Stage />
< Stage='2' >
        line#1#2  = -6.8571428571269 + 0.285714285719237*demand_1 +
                    0.137142857134923*demand_2 - 0.359999999960832*demand_3 -
                    0.0914285714451321*demand_4 - 0.0514285714218868*demand_5 +
                    3.00597046707533E-011*op_cost_1 - 1.96130335159752E-011*op_cost_2 +
                    9.45055716384964E-012*op_cost_3
        line#2#2  = 27.428571428698 - 0.142857142836771*demand_1 + 0.651428571438912*demand_2 +
                    0.039999999977285*demand_3 + 0.365714285708921*demand_4 +
                    0.205714285713234*demand_5 + 1.9304581215532E-011*op_cost_1 -
                    4.52918805442036E-012*op_cost_2 - 2.17149115403608E-012*op_cost_3
        line#3#2  = 18.000000000186 - 8.14954976076009E-012*demand_1 +
                    0.239999999984477*demand_2 - 0.280000000010678*demand_3 +
                    0.639999999992917*demand_4 + 0.359999999993548*demand_5 -
                    1.05526407498919E-011*op_cost_1 + 6.03196341847378E-012*op_cost_2 +
                    1.21285864728281E-012*op_cost_3
        line#4#2  = 2.57142857138849 + 0.142857142836145*demand_1 +
                    0.548571428560489*demand_2 - 0.0399999999778181*demand_3 -
                    0.365714285709523*demand_4 - 0.20571428571385*demand_5 -
                    1.8237785288433E-011*op_cost_1 + 5.52400769935196E-012*op_cost_2 +
                    2.91523656064935E-012*op_cost_3
        line#5#2 =  14.5714285714449 + 0.142857142843594*demand_1 + 0.308571428575262*demand_2 +
                    0.240000000031993*demand_3 + 0.594285714296922*demand_4 -
                    0.565714285708092*demand_5 - 8.99672777246986E-012*op_cost_1 -
                    1.5594856776024E-012*op_cost_2 + 2.37352215518716E-012*op_cost_3
        plant#1#2 = 44.5714285717671 + 0.142857142842949*demand_1 + 0.308571428574587*demand_2 +
                    0.240000000031359*demand_3 + 0.594285714296364*demand_4 +
                    1.23428571429126*demand_5 - 9.9533575221588E-012*op_cost_1 -
                    2.48076890469225E-012*op_cost_2 + 1.54287054126588E-012*op_cost_3
        plant#2#2 = 41.1428571430798 + 0.285714285710988*demand_1 + 0.377142857119463*demand_2 +
                    0.760000000028682*demand_3 + 0.548571428547721*demand_4 +
                    0.308571428571783*demand_5 + 1.94036380988528E-011*op_cost_1 -
                    1.35530083027858E-011*op_cost_2 + 1.06033814986285E-011*op_cost_3
        plant#3#2 = 34.2857142858298 + 0.571428571443967*demand_1 + 0.514285714303974*demand_2 +
                    0.399999999938192*demand_3 + 0.457142857153961*demand_4 +
                    0.25714285713509*demand_5 - 1.07743817897251E-011*op_cost_1 +
                    1.50396077497604E-011*op_cost_2 - 1.17343989514811E-011*op_cost_3
< Stage />
< Optimal Decisions />
```

Figure 7.9: Interpreted optimal decisions rules for the *progressive* approximation of the electricity capacity expansion model.

7.2.4 Discussion of Results and Loss of Optimality

What is perhaps quite surprising from the results is that the operational costs offer an insignificant contribution to the decision as to whether to expand the capacities of the power plants and electricity transmission lines. From figs. 7.7 and 7.9, we can infer that the decision rules are dominated by the demand for supply from the regional customers.

With regards to losses in optimality, we can conclude that the true optimal objective lies within the range [24529.9855379761, 25772.3478372735], which yields a relatively small percentage gap of 5.0645%. Thus, decision-maker may be indifferent to the upper-bound and the true optimal value.

$A_{1,1} \in \mathbb{R}^{16\times 8}$	$\begin{pmatrix} -\mathbf{I}_5 & \mathbf{0} \in \mathbb{R}^{5\times 3} \\ \mathbf{I}_5 & \mathbf{0} \in \mathbb{R}^{5\times 3} \\ \mathbf{0} \in \mathbb{R}^{3\times 5} & -\mathbf{I}_3 \\ \mathbf{0} \in \mathbb{R}^{3\times 5} & \mathbf{I}_3 \end{pmatrix}$

$A_{2,1} \in \mathbb{R}^{26\times 8}$	$\begin{pmatrix} \mathbf{0} \in \mathbb{R}^{3\times 5} & \mathbf{0} \in \mathbb{R}^{3\times 3} \\ \mathbf{0} \in \mathbb{R}^{3\times 5} & -350\mathbf{I}_3 \\ -350\mathbf{I}_5 & \mathbf{0} \in \mathbb{R}^{5\times 3} \\ -350\mathbf{I}_5 & \mathbf{0} \in \mathbb{R}^{5\times 3} \\ \mathbf{0} \in \mathbb{R}^{10\times 5} & \mathbf{0} \in \mathbb{R}^{10\times 3} \end{pmatrix}$

$A_{2,2} \in \mathbb{R}^{26\times 8}$	$\begin{pmatrix} \multicolumn{2}{c}{\mathbf{0} \in \mathbb{R}^{3\times 5}} & \multicolumn{2}{c}{-\mathbf{I}_3} \\ \multicolumn{2}{c}{\mathbf{0} \in \mathbb{R}^{3\times 5}} & \multicolumn{2}{c}{\mathbf{I}_3} \\ \multicolumn{2}{c}{-\mathbf{I}_5} & \multicolumn{2}{c}{\mathbf{0} \in \mathbb{R}^{5\times 3}} \\ \multicolumn{2}{c}{\mathbf{I}_5} & \multicolumn{2}{c}{\mathbf{0} \in \mathbb{R}^{5\times 3}} \\ \begin{pmatrix} -1.0 & -1.0 \\ -1.0 & 1.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{2\times 3} & \mathbf{0} \in \mathbb{R}^{2\times 2} & (1\ -1)^{\mathsf{T}} \\ \begin{pmatrix} 0.0 & 1.0 \\ 0.0 & -1.0 \end{pmatrix} & \begin{pmatrix} 0.0 & 1.0 & 0.0 \\ 0.0 & -1.0 & 0.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{2\times 2} & \mathbf{0} \in \mathbb{R}^{2\times 1} \\ \begin{pmatrix} -1.0 & 0.0 \\ 1.0 & 0.0 \end{pmatrix} & \begin{pmatrix} -1.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 \end{pmatrix} & \begin{pmatrix} 0.0 & 1.0 \\ 0.0 & -1.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{2\times 1} \\ \mathbf{0} \in \mathbb{R}^{2\times 2} & \begin{pmatrix} -1.0 & -1.0 & -1.0 \\ -1.0 & -1.0 & -1.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{2\times 2} & \mathbf{0} \in \mathbb{R}^{2\times 1} \\ \mathbf{0} \in \mathbb{R}^{2\times 2} & \begin{pmatrix} 0.0 & 0.0 & -1.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix} & \begin{pmatrix} -1.0 & 1.0 \\ 1.0 & -1.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{2\times 1} \end{pmatrix}$

$B_1 \in \mathbb{R}^{16\times 1}$	$\begin{pmatrix} \begin{pmatrix} -1.0 & -1.0 & -1.0 & -1.0 & -1.0 \end{pmatrix}^{\mathsf{T}} \\ \begin{pmatrix} 2.0 & 2.0 & 2.0 & 2.0 & 2.0 \end{pmatrix}^{\mathsf{T}} \\ \begin{pmatrix} -1.0 & -1.0 & -1.0 \end{pmatrix}^{\mathsf{T}} \\ \begin{pmatrix} 2.0 & 2.0 & 2.0 \end{pmatrix}^{\mathsf{T}} \end{pmatrix}$

$B_2 \in \mathbb{R}^{26\times 9}$	$\begin{pmatrix} \mathbf{0} \in \mathbb{R}^{16\times 1} & \mathbf{0} \in \mathbb{R}^{16\times 2} & \mathbf{0} \in \mathbb{R}^{16\times 3} & \mathbf{0} \in \mathbb{R}^{16\times 3} \\ \begin{pmatrix} 0.0 \\ 0.0 \\ 30.0 \\ -30.0 \end{pmatrix} & \begin{pmatrix} 1.0 & 0.0 \\ -1.0 & 0.0 \\ 0.0 & 1.2 \\ 0.0 & -1.2 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{4\times 3} & \mathbf{0} \in \mathbb{R}^{4\times 3} \\ \begin{pmatrix} 30.0 \\ -30.0 \\ 30.0 \\ -30.0 \\ 30.0 \\ -30.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{6\times 2} & \begin{pmatrix} 1.4 & 0.0 & 0.0 \\ -1.4 & 0.0 & 0.0 \\ 0.0 & 1.6 & 0.0 \\ 0.0 & 0.0 & 1.8 \\ 0.0 & 0.0 & -1.8 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{6\times 3} \end{pmatrix}$

Table 7.9: Generated matrices for the recourse constraints to be utilised by the *conservative* and *progressive* approximations.

$W \in \mathbb{R}^{18\times 9}$	$\begin{pmatrix} \begin{pmatrix} 1.0 \\ -1.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{2\times 2} & \mathbf{0} \in \mathbb{R}^{2\times 2} & \mathbf{0} \in \mathbb{R}^{2\times 2} & \mathbf{0} \in \mathbb{R}^{2\times 2} \\ \begin{pmatrix} 120.0 \\ 0.0 \\ 120.0 \\ 0.0 \end{pmatrix} & \begin{pmatrix} -1.0 & 0.0 \\ 1.0 & 0.0 \\ 0.0 & -1.0 \\ 0.0 & 1.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{4\times 2} & \mathbf{0} \in \mathbb{R}^{4\times 2} & \mathbf{0} \in \mathbb{R}^{4\times 2} \\ \begin{pmatrix} 120.0 \\ 0.0 \\ 120.0 \\ 0.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{4\times 2} & \begin{pmatrix} -1.0 & 0.0 \\ 1.0 & 0.0 \\ 0.0 & -1.0 \\ 0.0 & 1.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{4\times 2} & \mathbf{0} \in \mathbb{R}^{4\times 2} \\ \begin{pmatrix} 120.0 \\ 0.0 \\ 80.0 \\ 0.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{4\times 2} & \mathbf{0} \in \mathbb{R}^{4\times 2} & \begin{pmatrix} -1.0 & 0.0 \\ 1.0 & 0.0 \\ 0.0 & -1.0 \\ 0.0 & 1.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{4\times 2} \\ \begin{pmatrix} 80.0 \\ 0.0 \\ 100.0 \\ 0.0 \end{pmatrix} & \mathbf{0} \in \mathbb{R}^{4\times 2} & \mathbf{0} \in \mathbb{R}^{4\times 2} & \mathbf{0} \in \mathbb{R}^{4\times 2} & \begin{pmatrix} -1.0 & 0.0 \\ 1.0 & 0.0 \\ 0.0 & -1.0 \\ 0.0 & 1.0 \end{pmatrix} \end{pmatrix}$
$h \in \mathbb{R}^{18\times 1}$	$\begin{pmatrix} 1.0 & -1.0 & \mathbf{0} \in \mathbb{R}^{1\times 16} \end{pmatrix}^{\mathsf{T}}$

Table 7.10: Generated support matrices for the *conservative* and *progressive* approximations.

8 Conclusion

The overarching goal of this project is to allow industrial modellers to describe decision problems that are subject to some degree of uncertainty. The format for this description is what we refer to as our algebraic modelling language (AML). Our standardised input format has been implemented in such a way that the syntactical design allows for descriptions of system-specific knowledge to be supplied in a manner that is natural, expressive and compact. This is to ensure that the final delivery successfully helps to alleviate the burden placed on the modeller to formulate highly complex decision-making problems.

8.1 Contributions

For emphasis, we state our primary contributions thus far.

1. We have specified and designed an algebraic modelling language for stochastic programming, whereby the modeller can intuitively specify an optimisation model for decision-making under uncertainty (see section 4.3).

2. We have implemented a parsing routine that can read stochastic models that have been specified in our standardised input format (see section 5.3). As a final result, the parser constructs a highly condensed representation of the input file which can be efficiently queried and manipulated by the client modules (see section 5.3.3).

3. To derive instances of tractable conic programming problems, we have designed and implemented algorithms to generate the matrix components utilised in eq. (Cons-MSP$_{fixed}$) and eq. (Prog-MSP$_{fixed}$) (see section 5.4).

4. Using the derived matrix components, we are able to compute the objective function and constraints to automate the generation of the *conservative* and *progressive* linear programs (see section 5.5).

5. Furthermore, we have introduced a layer of abstraction to interface with a variety of popular external solvers using the YALMIP framework (see section 5.5.3).

6. We have also interpreted the computed solutions to specify the optimal decision rules, and present the results to the user (see section 5.6).

7. As an extension, we have implemented notation for stochastic processes, which allows the modeller increased flexibility in specifying very compact models (see chapter 6).

8.2 Qualitative Evaluation

In this section, we aim to evaluate the final product. In addition to verifying correctness of the computed results, we also focus on the effectiveness of the algebraic modelling language and we discuss our concerns for the performance of the automated model processing.

8.2.1 Verification of Correctness

For our implementation, the basics for establishing code correctness begins with an automated test-harness to unit test the parser using the JUnit framework, and custom MATLAB test classes. Currently, the JUnit classes soley establish the correctness of the parsing routine, and focused on verifying correct construction of the internal model. Our justification for this prioritisation is that the computations, which are derived from the internal model, can only be correct if the intermediate representation of the input has itself been formed accurately.

For black-box testing, we have specified and manually solved a simplified adaptation of the *newsvendor problem* using the linear decision rule approximation (see chapter 3)[2]. In section 3.6, we illustrate the structure of the matrix components for the conservative and progressive linear programs. From inspection, we are able to validate that the routines we have written to automate the generation of these matrices appear to be correct.

In section 4.2 we discussed an existing system that has been prototyped to address the same objectives of this project. While our intention is to replace the legacy system, we also utilise the original prototype as an oracle for checking the accuracy and validity of our computations. Listing 4.1 illustrates the equivalent specification of the newsvendor problem using the initial prototype. In addition to specifying the same numerical parameters for the both models, we have also supplied both formulations with the same sample data. The results generated by the legacy system for the newsvendor model are given in figs. 8.1 and 8.2.

```
Decisions_upper.txt:
    x_1_1_1 = +5.0
    x_1_1_2 = -5.0
    x_2_2_2 = 0.0

Objective.txt:
    -25
```

Figure 8.1: Contents of the generated results files upon solving the newsvendor problem's *conservative* LP using the legacy system.

```
Solutions_lower.txt:
    x_1_1_1 = +6.61801327791999
    x_1_1_2 = -6.61801327791999
    x_2_2_2 = 0.0

Objective.txt:
    33.0900663895999
```

Figure 8.2: Contents of the generated results files upon solving the newsvendor problem's *progressive* LP using the legacy system.

Figure (8.3) explains that, while our conservative solutions coincide with those computed by the legacy system, the progressive results are consistently different. Having conducted several investigations involving manual comparisons of the generated matrices, we are yet to pinpoint

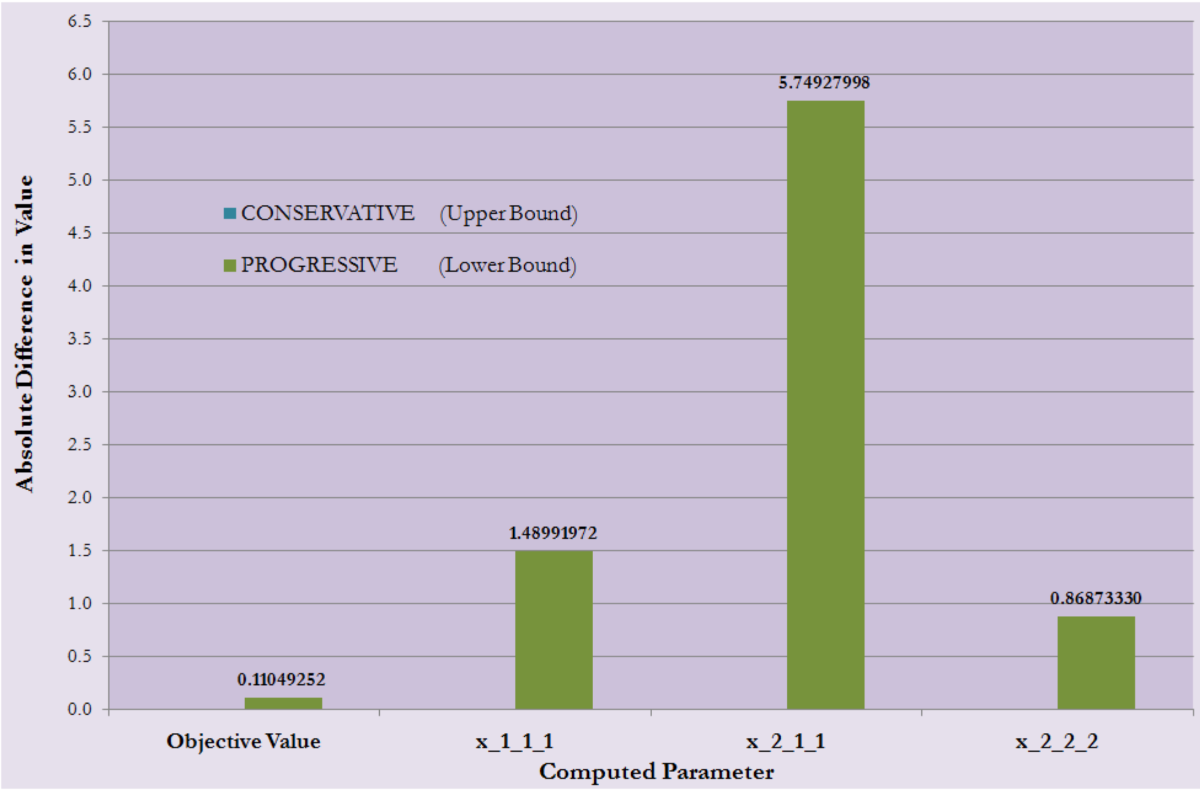


Figure 8.3: Graph illustrating the differences in the values computed by JADA and the legacy system.

the exact cause for the disagreement in values. However, we do report that the difference in the objective value (0.33%) can be considered insignificant.

For purposes of evaluating the expressibility and robustness of our system, we specified a decision-making problem centered on the capacity expansion of an electric power plant. This model is considerably more complex, which is reflected in the size of the generated linear programs. The conservative approximation has 420 linear constraints, while the progressive approximation has 964 constraints[1]. In figure (8.4) we present the conservative solutions as computed by the original prototype, and we note that the optimal objective value of the progressive LP is 12191.5436214909.

In section 7.2.4 we mentioned our surprise regarding JADA's computed optimal decision rules being insignificantly influenced by the uncertainty of the operational costs. Comparisons with the legacy system suggest that our results are incorrect. Rather than being dominated by the stochastic regional demand, the decision rules are also functions of the operational costs, which logically seems appropriate. Therefore, we can conclude that there are still some lingering issues regarding the correctness of our implementation.

[1]The stated values are exclusive of the inequalities for the bounds of the decision variables

```
Decisions_upper.txt:
    at period 1    u1 =  1.085810
    at period 1    u2 =  2.000000
    at period 1    u3 =  1.000000
    at period 1    v1 =  1.000000
    at period 1    v2 =  1.000000
    at period 1    v3 =  1.000000
    at period 1    v4 =  1.000000
    at period 1    v5 =  1.000000
    at period 2    f1 =  0.633140*demand_1 + 1.200000*demand_2
    at period 2    f2 =  -0.366860*demand_1 + 1.200000*demand_2
    at period 2    f3 =  90.000000 +0.366860*demand_1 - 0.566580*demand_3 +
                         1.799720*demand_5 - 1.500000*var_cost_2
    at period 2    f4 =  30.000000 +0.366860*demand_1
    at period 2    f5 =  -30.000000 +0.566580*demand_3 + 1.600000*demand_4 -
                         1.799720*demand_5 + 1.500000*var_cost_2
    at period 2    g1 =  0.566580*demand_3 + 1.600000*demand_4 + 0.000280*demand_5 +
                         1.500000*var_cost_2
    at period 2    g2 =  120.000000 + 1.000000*demand_1 + 1.200000*demand_2 +
                         0.833420*demand_3 + 1.799720*demand_5 - 1.500000*var_cost_2
    at period 2    g3 =  0.0

Objective.txt:
    32911.4789892197

Optimality2.txt:
    optimal
```

Figure 8.4: Contents of the generated results files upon solving the newsvendor problem's *conservative* LP using the legacy system.

8.2.2 Irreducible Algebraic Modelling Language Requirements

We have adapted the specifications for programming languages[47] to determine a set of desirable properties for evaluating the effectiveness of our algebraic modelling languages. In particular, we note the need for adequacy, high learnability, and productivity.

The Adequacy Requirement

To evaluate the practicality of our algebraic modelling language, we conducted several experiments, two of which have been methodically detailed in chapter 7. To ensure we satisfied the minimum specification for this project, we initially set ourselves the target of ensuring our input format could allow for relatively simple multi-stage stochastic programs with fixed recourse to be described.

In reality, the stochastic programming problems we expect to be specified in our grammar are more complicated than the newsvendor problem (see section 3.6 and section 7.1). For this reason, we also considered a more complex, real-life decision-making problem concerned with the capacity expansion and investment planning. Such problems typically involve more decisions and are characterised by many stochastic parameters. In section 7.2, we have shown that JADA is more than capable of allowing the decision-maker to formulate a description for such a problem.

While our algebraic modelling language is adequate for those models studied in chapter 7, its vocabulary is inadequate to describe the inventory model considered by Kuhn et al in [2](see appendix E). In particular, we are required to augment the syntax to allow for highly complex expressions involving trigonometric functions and standard mathematical constants.

The Learnability Requirement

We assume that the end-users of JADA will be industrial modellers who are experts of their respective systems but lack a thorough understanding of optimisation theory or algorithm design. For this reason, our grammar has been designed to incorporate literal, common vocabulary that can translate to any decision-making problem, as opposed to purely mathematical notation.

In addition, we have recognised that learnability can be severely impeded by ambiguous syntax, which is we have revised several of our designs to avoid re-defining conventional uses of notation and aim for conformation where possible. For example, two constructive critiques of an initial design addressed the use of square parentheses to denote the multiplication of the decision and random variables in the objective, and our choice to refer to *expectation* by the reserved keyword 'exp'. In the former case, we persisted with the use of square brackets to encourage the modeller to factorise the cost coefficients of the decision variables. Although, this decreased the complexity in extracting the cost vector, it was admittedly not as intuitive as the conventional asterisk symbol '*' to denote multiplication. In the latter case, it had been suggested that denoting *expectation* by the keyword 'exp' could be confused by the exponential function. Consequently, we now denote the expectation measure by the reserved keyword 'expectation'.

Furthermore, wedeliberately decomposed the standard format for describing a model into six distinct sections. This inherently serves to provide categories for declaring temporal attributes, variables, distribution data, costs and constraints. Not only does this achieve a structure that can be easily learned, but it allows the model to be incrementally built. Moreover, we made a choice earlier on in the design process to not impose a strict order for the way these sections were defined. Although this does complicate the parsing logic, it permits the end-user some flexibility of declaring the sections in an order that is personally easier to learn and thus remember.

Unfortunately, we were unable to carry out any usability testing during the completion of this project to objectively assess the degree of learnability. If such tests were to be carried out in the near future, we would need to collate opinions on the our language's degree of brevity, simplicity and the effectiveness of the mechanisms in place for error prevention.

The Productivity Requirement

This requirement aims to evaluate an AML that is already characterised by high learnability and adequate expressibility. We refer the reader to our additional implementation of the stochastic process notation to evaluate how well our language encourages the modeller's productivity.

Upon reviewing the minimal implementation, it had been pointed out that while the language could, to some extent, adequately permit specification of stochastic programs, it did not support capabilities for rapid prototyping and high-level expressibility. This flaw needed to be addressed to successfully avoid burdening the modeller with excessive low level processing. We considered two high-level constructs, which were iterated addition and universal quantification. Not only did this drastically reduce the time spent formulating the program, but it also significantly decreased the size of the program in the user's storage space.

8.2.3 Performance

Beyond parallel for-loops[40], MATLAB does not provide capabilities for concurrency. This is quite problematic since the current performance of the classes aimed at automating the model processing do require a considerable amount of processing time. We are able to identify two causes for the degradation in performance, which include

- overloading MATLAB built-in functions with our custom MATLAB data classes from the `expressions` utility package, and
- the inability to preallocate arrays for instances where the maximum size is not known.

Unfortunately there are not many efficient options available to us to resolve the second cause, but a tactical solution has been to provide an initial size and resize the array accordingly if more or less storage is required. The first cause arises from our need to symbolically perform the matrix multiplications to present the generated linear programs to the user. By overloading MATLAB's built-in functions, we have negatively affected the performance. While this performance issue was not severe for simple models like the newsvendor problem or smaller versions of the capacity expansion models, when we experimented with stochastic programs involving a greater number of variables and constraints, the negative impact of overloading methods, such as the `plus` function[2], was immediately apparent.

[2] For example, if the `plus` function is overloaded to handle any of the integer classes differently, then some of the MATLAB's internal optimisations for the plus function may be disabled[38].

We do ask the reader to note that we have performed some optimisations to the MATLAB code, which include using vectorising algorithms to convert `for` and `while` loops to equivalent vector or matrix operations. Additionally, we have used the `repmat` function to initialise arrays or matrices by replication[41] where appropriate. However, even the frequency of usage for the latter approach has had to be restricted since the `repmat` function is memory intensive[42].

8.3 Further Work

In this section we suggest directions for future development. In addition to addressing the correctness and performance issues, we can also consider extending JADA with regards to the syntax of the algebraic modelling language, the scope of the stochastic programming framework, and the software design.

8.3.1 Extending the Language

In section 8.2.2 we highlighted some prevailing inadequacies of the JADA grammar. In addition to not providing support for mathematical functions, we have not considered binary decision variables or logical expressions for defining the constraints, such as set membership, which can be easily implemented. In addition, we could also incorporate language and functionality for multi-stage stochastic mean-variance portfolio optimisation [59][62], which requires quadratic objective functions and ellipsoidal uncertainty sets[64].

8.3.2 Extending the Problem Scope

The expected overall deliverable focuses on multi-stage stochastic programming problems with fixed recourse, therefore we can extend the functional capabilities of the modelling language and solver sub-system to handle multi-stage programs with random recourse. Widening the problem scope would also allow for the decision-maker to model worst-case optimisation problems.

Random Recourse-Constrained MSPs

The approximation models eqs. (Cons-MSP$_{fixed}$) and (Prog-MSP$_{fixed}$) assume that the recourse matrices are deterministic. By investigating into the cases where these recourse matrices are in fact dependent on the uncertain parameters, we can extend the AML to also model random recourse-constrained problems. The formulations eqs. (Cons-SP$_{random}$) and (Prog-SP$_{random}$) formulate the conservative and progressive programs for approximating one-stage programs with random recourse. We can incorporate a temporal structure into the core model to capture multi-stage stochastic programs with random recourse. To facilitate the computations of eqs. (Cons-SP$_{random}$) and (Prog-SP$_{random}$) for multiple time periods, we have to further sample the distribution of the random variables to derive $\mathbb{Q}(S_\mu)$ the tensor of all moments of its associated probability measure up to the fourth order. We can implement eq. (2.1.2.14) to achieve this.

Worst-case Optimisation for MSPs with Fixed and Random Recourse

We remind the reader that worst-case optimisation considers the situation where the decision-maker has insufficient knowledege about the probability distributions of the uncertain parameters, and thus there are a plausible family of distributions for which the problem's random variables could follow. The AML must now allow the random outcomes to be modelled as a tuple $\xi = (\eta, \zeta)$. The probability measure $\mathbb{P}$ is not fully known, and thus we might need to introduce new language constructs to allow the modeller to specify the marginal distribution of η and the conditional support for ζ. At a higher-level we consider worst-case optimisation for multi-stage problems, with fixed recourse and random recourse respectively by extending the minimal implementation and the previous extension.

8.3.3 Improving the Design and Implementation

The ultimate goal of the project is a fully-integrated environment for decision-making under uncertainty. We would like to extend the functional behavior of the system to allow the user to specify models and interpret the results in a user-friendly manner with maximal automation. Examples of these improvements include:

- Extending the system for supporting multiple data formats and sources for supplying the sampling data such as XML tables, spreadsheets and database tables.
- Sophisticated reporting and analytical engines for user-friendly visualisations of the results. We can improve the presentation of the results by generating output in a PDF format where mathematical notation can be correctly typesetted for readability, and where tables rather than pseudo-XML can be used to display the solutions computed by YALMIP.
- A graphical user-interface to further assist and automate the input of the model description. In the end, we envision a custom integrated development environment (IDE) with syntax highlighting, code completion, context-sensitive content assistance, save and load functionality, and basic template generation for specifying and editing models in the AML format.
- For time-scalable models such as the inventory management system (see appendix E), it is possible to automate the generation of the results over many stages. Thus, we can further enrich the reporting engine with MATLAB charting functions to graphically plot the percentage gaps in the linear decision rule bounds over the many finite time horizons. In addition, we can introduce functionality to enable such simulations to be paused and resumed for user convenience.

A Parser Implementation using Regular Expressions

A.1 Tokeniser.m

Listing A.1: Code listing showing the `generateJADAModel(...)` function defined in `Tokeniser.m`.

```

% + Function Description: extracts the tokens from the contents of a JADA
    file
% + Function Input:        string representing file contents of a JADA file
% + Function Output:       a complete JADAModel
jadaModel = function generateJADAModel(self, fileContents)

  %prepare contents - remove white spaces and newlines
  fileContents = regexprep(fileContents{:}, '\s', '');
  fileContents = cat(2,fileContents{:});

  %check file contains the 'Model' language construct
  index = ismember(fileContents, LanguageConstructs.MODEL)==1;

  %create an empty JADAModel to be filled by the construct tokenisers
  jadaModel = JADAModel();

  %build a cell-array of the construct tokenisers
  constructs = {self.general, self.variables, self.objective, self.
      constraints,
                self.samples, self.support};

  %iterate through language constructs to extract and process tokens
  cellfun(@(construct) Tokeniser.tokeniseLanguageConstruct(fileContents,
                                                          jadaModel,
                                                          construct),
                         constructs);

end %generateJADAModel
```

Listing A.2: Code listing showing the auxiliary function `tokeniseLanguageConstruct(...)` for `generateJADAModel(...)` as was defined in `Tokeniser.m`.

```
% + Function Description: auxiliiary method to extract and process tokens
% + Function Input:        #1) string representing name of construct
%                          #2) intermediate representation of the
%                              parsed file contents
%                             (partially complete JADAModel)
%                          #3) Implementation of an IConstructTokeniser
% + Function Output:       none
function tokeniseLanguageConstruct(fileContents, jadaModel, construct)

  %get and apply language construct's regular expression to extract its
     tokens
  tokens = (regexp(fileContents, construct.getRegex(), 'tokens');
  tokens = GenericStructures.flattenCellArray(tokens, false);

  %delegate processing of extracted tokens to language construct
  construct.processTokens(tokens, jadaModel);

end %tokeniseLanguageConstruct
```

B

Parser Implementation using ANTLR Version 3.0

B.1 Alternative Context-Free Grammars

Table B.1: Comparison of Investigated Context-free Grammars.

Name	Parsing Algorithm	Output Languages	Grammar/ Code	Lexer	Development Platform	IDE	License
ANTLR	LL(*)	C C++ C# Java Python	Mixed	Generated	Java Virtual Machine	Yes	BSD
Spirit	Recursive Descent	C++	Mixed	Internal	All	No	Boost
YACC++	LR(1) LALR(1)	C++ C#	Mixed	Generated/ External	All	No	proprietary

B.2 ParserEngine.java

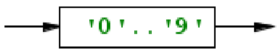
Listing B.1: Code listing showing the co-ordination of the auto-generated Java classes for the parsing logic.

```java
/**
 *   @Class    This class co-orindates reading in a model, ANTLR tokenisation
 *             and generation of an internal representation
 *             (ImmutableJADAModel) of the JADA file
 **/
public class ParserEngine
{

  //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  // METHODS
  //***********************************************************************

  /**
   * Default Constructor
   */
  public ParserEngine() {}

  /**
   * Parses a JADA file to generate an internal representation
   * (ImmutableJADAModel)
   *
   * @param String representing an absolute filepath reference to a JADA file
   *
   * @return   An ImmutableJADAModel instance which represents the
   *           underlying data of the model specified in the file
   */
  public static ImmutableJADAModel parseFile(String filepath)
                throws IOException, RecognitionException
  {
    //Construct a file reader to read JADA file at given filepath
    FileReader reader = new FileReader(filepath);

    //Construct a JADA Lexer
    JADALexer lexer = new JADALexer(new ANTLRReaderStream(reader));

    //Close file reader
    reader.close();

    //Tokenise contents of file and return generated internal representation
    return parse(lexer);
  }

  /**
   * Parses a string representing the file contents of a JADA file
   * to generate an internal representation (ImmutableJADAModel)
   *
   * @param String representing an absolute filepath reference to a JADA file
   *
   * @return   An ImmutableJADAModel instance which represents the
   *           underlying data of the model specified in the file
```

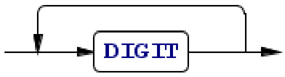
```
   */
  public static ImmutableJADAModel parseContents(String fileContents)
              throws IOException, RecognitionException
  {
    //Return an immutable equivalent model
    return parse(new JADALexer(new ANTLRStringStream(fileContents)));
  }

  /**
   * Helper method for parsing a model specified in JADA syntax
   *
   * @param a JADALexer object
   *
   * @return  An ImmutableJADAModel instance which represents the
   *          underlying data of the specified model
   */
  private static ImmutableJADAModel parse(JADALexer lexer)
              throws IOException, RecognitionException
  {
    //Construct a JADA Parser
    JADAParser parser = new JADAParser(new CommonTokenStream(lexer));

    //Tokenise JADA file by invoking start rule to get a parser result
    JADAParser.model_return result = parser.model();

    //Get the AST
    CommonTree ast = (CommonTree)result.getTree();

    //Process the AST to generate an ImmutableJADAModel to return
    JADATree treeParser = new JADATree(new CommonTreeNodeStream(ast));

    //Return an immutable equivalent model
    return treeParser.model().getImmutableJADAModel();
  }

  //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  // END METHODS
  //**********************************************************************
}
```

B.3 Lexer Syntax Diagrams

B.3.1 Numbers



(a) Digit

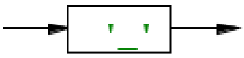


(b) Integer (INT)

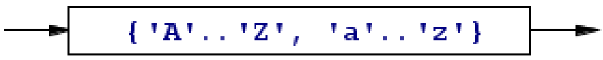
(c) Float

Figure B.1: Syntax diagrams representing the context-free grammar specified in the lexer source files for *numbers*.

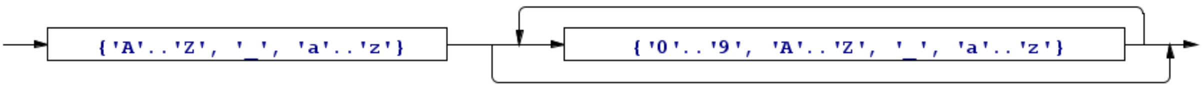
B.3.2 Variable Identifiers



(a) Underscore (USCR)



(b) Alphabetical Character (ALPHA)



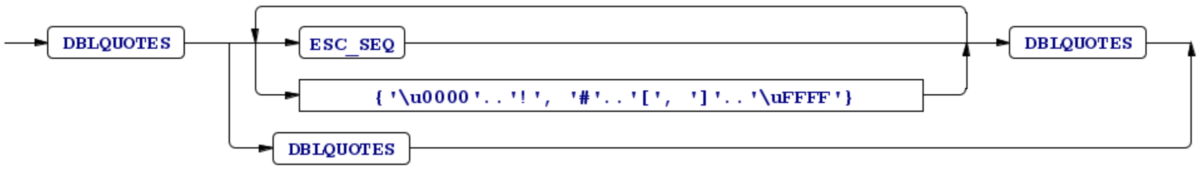
(c) Identifier (IDENT)

Figure B.2: Syntax diagrams representing the context-free grammar specified in the lexer source files for *variable identifiers*.

B.3.3 Strings and Whitespace

(a) Double Quotation Marks (DBLQUOTES)

(b) Backslash (BKSLASH)



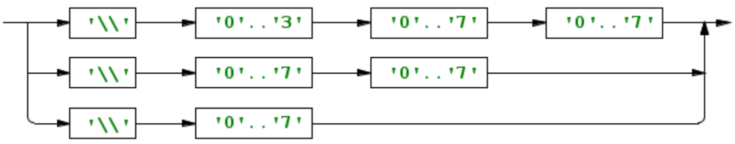
(c) String

(d) Whitespace

Figure B.3: Syntax diagrams representing the context-free grammar specified in the lexer source files for *strings* and *whitespace*.

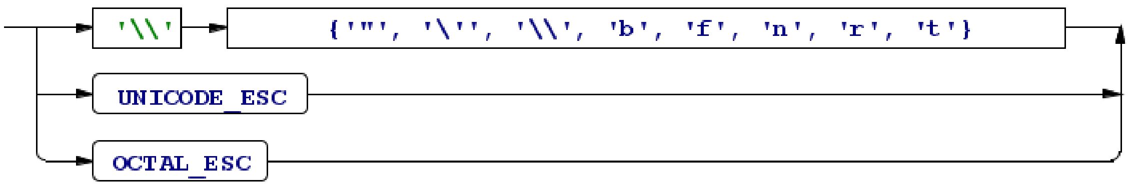
B.3.4 Escape Sequences

(a) Hexadecimal Digit (HEX_DIGIT)



(b) Octal Escape Sequence (OCTAL_ESC)

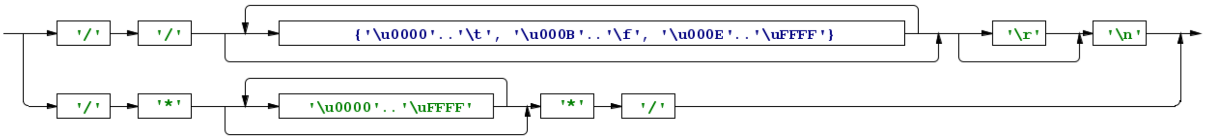
(c) Unicode Escape Sequence (UNICODE_ESC)



(d) Escape Sequence (ESC_SEQ)

Figure B.4: Syntax diagrams representing the context-free grammar specified in the lexer source files for *escape sequences*.

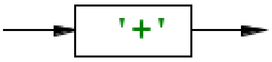
B.3.5 In-lined and Block Comments



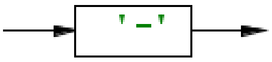
(a) Comment

Figure B.5: Syntax diagrams representing the context-free grammar specified in the lexer source files for *comments*.

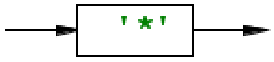
B.3.6 Mathematical Operators



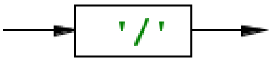
(a) Addition (PLUS)



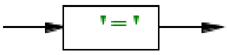
(b) Subtraction (MINUS)



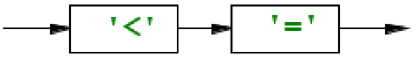
(c) Multiplication (TIMES)



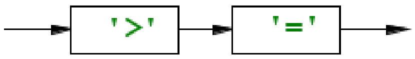
(d) Division (DIV)



(e) Equals



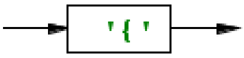
(f) Less-Than-Or-Equal-To (LTE)



(g) Greater-Than-Or-Equal-To (GTE)

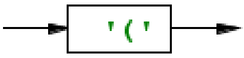
Figure B.6: Syntax diagrams representing the context-free grammar specified in the `JADALexer` source file for *mathematical operators*.

B.3.7 Symbols (Delimiters and Terminals)

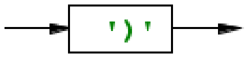


(a) Left Bracket (LBRACKET)

(b) Right Bracket (RBRACKET)



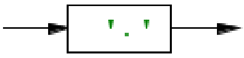
(c) Left Parenthesis (LPAREN)



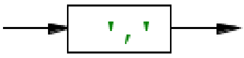
(d) Right Parenthesis (RPAREN)

(e) Left Square Parenthesis (LSQPAREN)

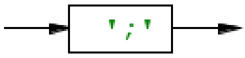
(f) Right Square Parenthesis (RSQPAREN)



(g) Dot



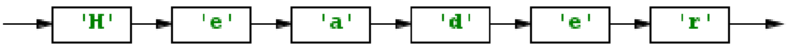
(h) Comma



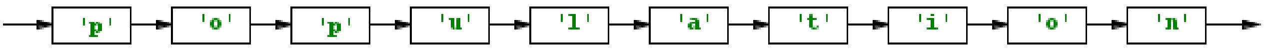
(i) Semi-Colon (SEMI)

Figure B.7: Syntax diagrams representing the context-free grammar specified in the lexer source files for *symbols*.

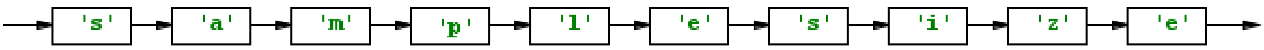
B.3.8 Reserved Keywords



(a) Header

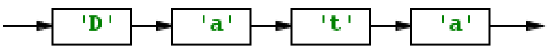


(b) Population



(c) Samplesize

(d) Variables

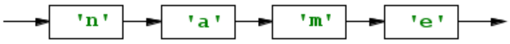


(e) Data

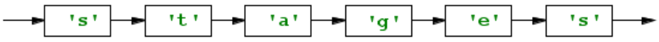
Figure B.8: Syntax diagrams representing the context-free grammar specified in the `SampleDataLexer` source file for the *reserved keywords*.

(a) Model

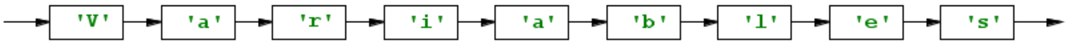
(b) General



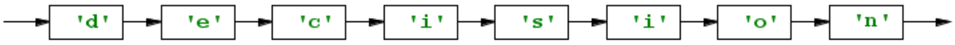
(c) Name



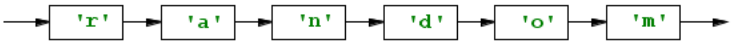
(d) Stages



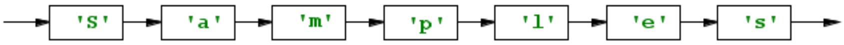
(e) Variables



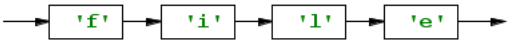
(f) Decision



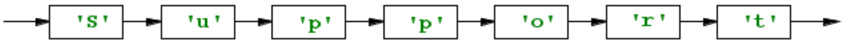
(g) Random



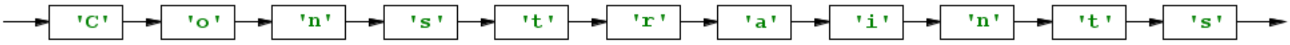
(h) Samples



(i) File

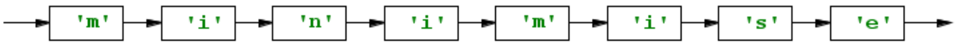


(j) Support

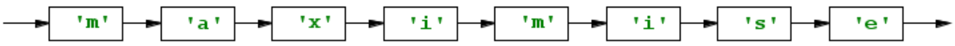


(k) Constraints

(l) Objective



(m) Minimise



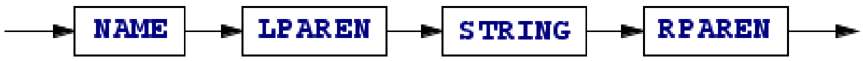
(n) Maximise

(o) Expectation

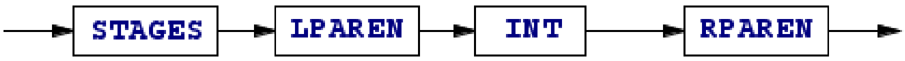
Figure B.8: Syntax diagrams representing the context-free grammar specified in the `JADALexer` source file for the *reserved keywords*.

B.4 Parser Syntax Diagrams

B.4.1 General Language Construct



(a) Name

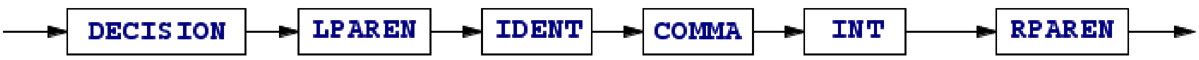


(b) Stages

(c) General

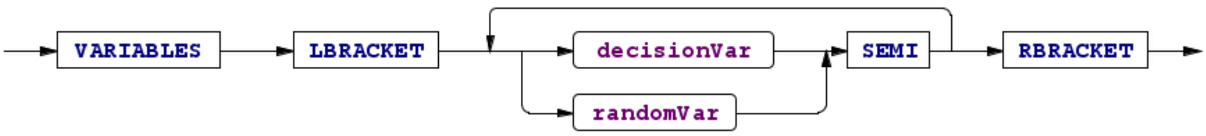
Figure B.9: Syntax diagrams representing the context-free grammar specified in the `JADAParser` source file for the *General* language construct.

B.4.2 Variables Language Construct



(a) Decision Variable (decisionVar)

(b) Random Variable (randomVar)



(c) Variables

Figure B.10: Syntax diagrams representing the context-free grammar specified in the `JADAParser` source file for the *Variables* language construct.

B.4.3 Constraints Language Construct

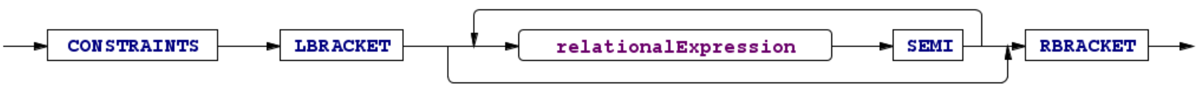


Figure B.11: Syntax diagrams representing the context-free grammar specified in the `JADAParser` source file for the *Constraints* language construct.

B.4.4 Support Language Construct

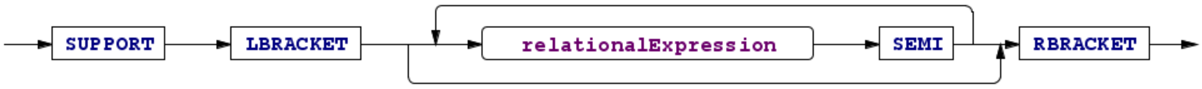
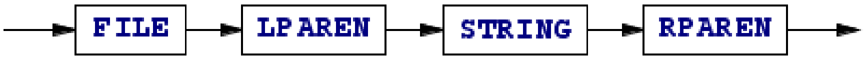
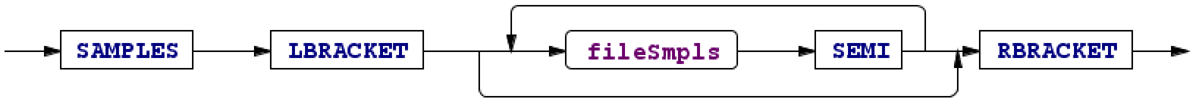


Figure B.12: Syntax diagrams representing the context-free grammar specified in the `JADAParser` source file for the *Support* language construct.

B.4.5 Samples Language Construct



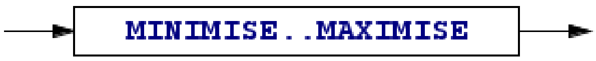
(a) File Samples (fileSmpls)



(b) Samples

Figure B.13: Syntax diagrams representing the context-free grammar specified in the `JADAParser` source file for the *Samples* language construct.

B.4.6 Objective Language Construct



(a) Maximisation/Minimisation (goal)

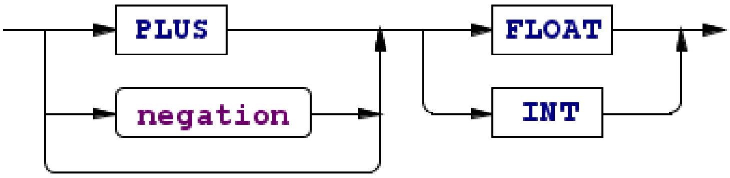
(b) Statistic

(c) Decision Cost (costTerm)

(d) Objective

Figure B.14: Syntax diagrams representing the context-free grammar specified in the `JADAParser` source file for the *Objective* language construct.

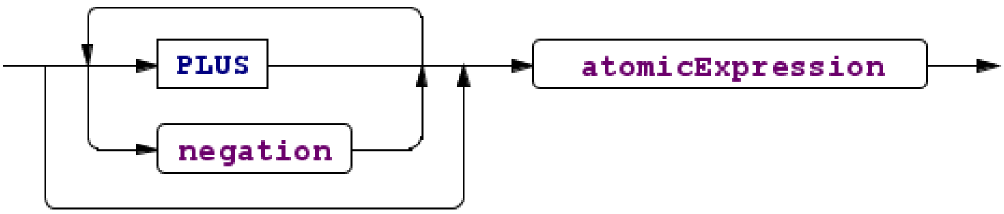
B.4.7 Arithmetic Expressions Constructs



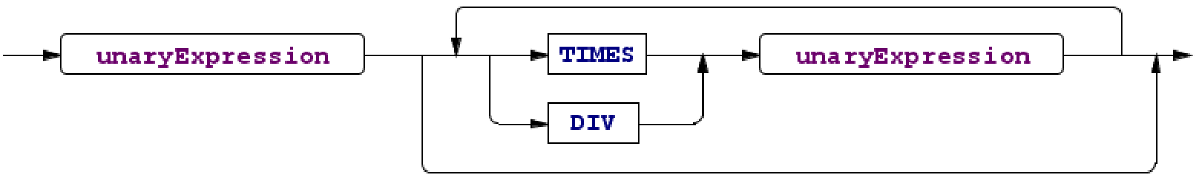
(a) Number

(b) Atomic Expression

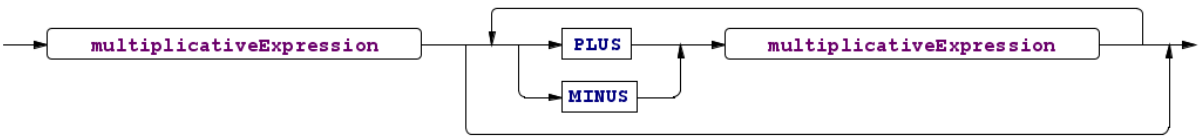
(c) Negation



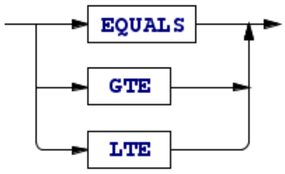
(d) Unary Expression



(e) Multiplicative Expression Expression



(f) Additive Expression Expression

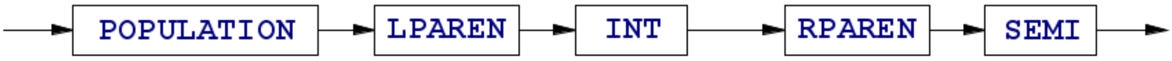


(g) Relational Operators Expression

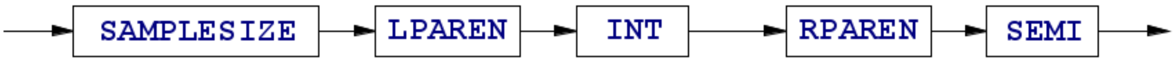
(h) Relational Expression Expression

Figure B.14: Syntax diagrams representing the context-free grammar specified in the `JADAParser` source file for the *arithmetic expressions*.

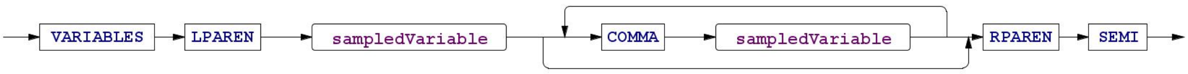
B.4.8 Sample Data File Constructs



(a) Population

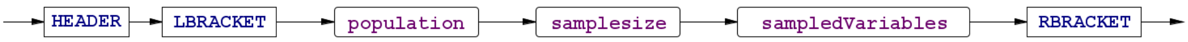


(b) Sample Size

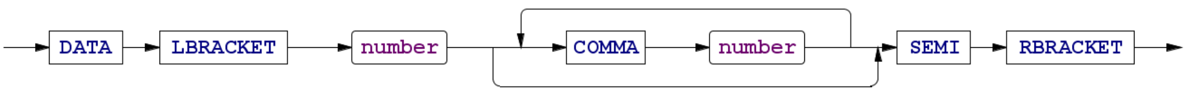


(c) Sampled Variables

(d) Sampled Variable



(e) Header



(f) Data

(g) Sample Data

Figure B.15: Syntax diagrams representing the context-free grammar specified in the `SampleDataParser` source file for the *sample data*.

B.5 Validation Logic

Listing B.2: Code listing demonstrating the validation logic for checking multiple declarations of a language construct.

```
// PARSER RULES: 'GENERAL' MODEL-SUB-BODY Specification
general
   ...
   //_________________ Post-processing
  @after
  {
    validator.setGeneralAlreadyDeclared(true);
  }

  //_________________ Define rule
  : GENERAL
    {
      validator.checkConstructNotAlreadyDeclared
                (Construct.GENERAL_CONSTRUCT, $GENERAL.getLine());
    }
    ...
  ;
```

B.6 Importing the Parser into the MATLAB Workspace

Listing B.3: Code listing illustrating how the `parseFile(...)` method defined in the loaded class definition `ParserEngine.class` is invoked in the MATLAB class wrapper `Parser.m`.

```
% + Function Description: parses an JADA file
% + Function Input:       absolute filepath to JADA file
% + Function Output:      ImmutableJADAModel (Java object) contains
%                         internal representation of parsed file
function jadaModel = parseFile(filePath)

  jadaModel = system.parser.ParserEngine.parseFile(filePath);

end %parseFile
```

C

Conservative and Progressive Constraints Computations

C.1 Implementation of the Placeholder Methods

C.1.1 Conservative Approximation

getDecisionRulesOuterFactor(t)

return Γ[‘second order moments’]

getStandardisedFeasibilityCondition(expr)

return expr + (Γ[‘slack decision rules’, t])(Γ[‘LHS support matrix’])

getPositiveSlacknessCondition(t)

$h \leftarrow$ Γ[‘RHS support matrix’]

$\Lambda_{symbolic,t} \leftarrow$ Γ[‘slack decision rules’, t]

$\text{constraint}_{LHS} \leftarrow \text{cell}(\Lambda_{symbolic,t} h, \Lambda_{symbolic,t})$

$\text{constraint}_{RHS} \leftarrow$ cell(0.0, *new* ConstantTerm(0.0))

$\text{constraint}_{quantifier} \leftarrow$ `ConstraintQuantifier.GTE`

return [constraint_{LHS}, $\text{constraint}_{quantifier}$, constraint_{RHS}]

C.1.2 Progressive Approximation

getDecisionRulesOuterFactor(t)

$M_{\mathbb{E}[\xi\xi^\intercal]} \leftarrow$ Γ[‘second order moments’]

return $M_{\mathbb{E}[\xi\xi^\intercal]} P_t^\intercal (P_t M_{\mathbb{E}[\xi\xi^\intercal]} P_t^\intercal)^{-1}$

getStandardisedFeasibilityCondition(expr)

return expr + (Γ['slack decision rules', t])P_t

getPositiveSlacknessCondition(t)

$W \leftarrow \Gamma$['LHS support matrix']

$h \leftarrow \Gamma$['RHS support matrix']

$e_1 \leftarrow \Gamma$['basis vector']

constraint$_{LHS} \leftarrow$ `cell`($(W - he_1^\top)M_{\mathbb{E}[\xi\xi^\top]}{P_t}^\top{S_t}^\top$, $S_t P_t M_{\mathbb{E}[\xi\xi^\top]} e_1$)

constraint$_{RHS} \leftarrow$ `cell`(0.0, *new* ConstantTerm(0.0))

constraint$_{quantifier} \leftarrow$ `ConstraintQuantifier.GTE`

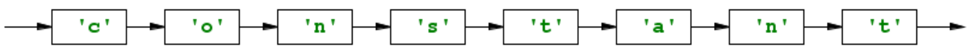
return [constraint$_{LHS}$, constraint$_{quantifier}$, constraint$_{RHS}$]

D

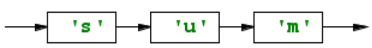
Extended Parser Implementation For Stochastic Processes Notation

D.1 Lexer Syntax Diagrams

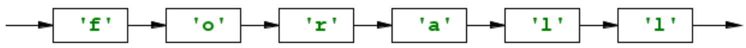
D.1.1 Extended Reserved Keywords



(a) Constant



(b) Summation

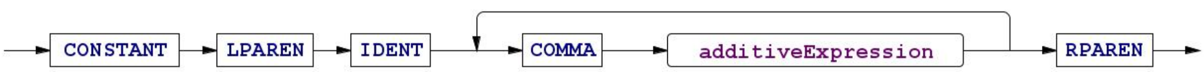


(c) Universal Quantification (forall)

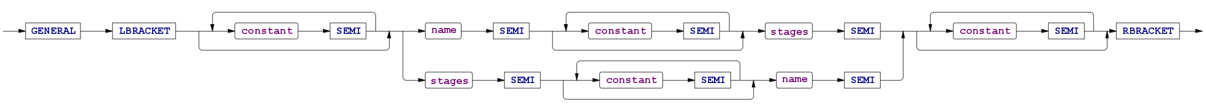
Figure D.1: Syntax diagrams representing the extended context-free grammar specified in the `JADALexer` source file for *reserved keywords*.

D.2 Parser Syntax Diagrams

D.2.1 Extended General Language Construct



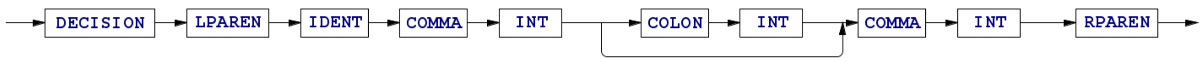
(a) Constants



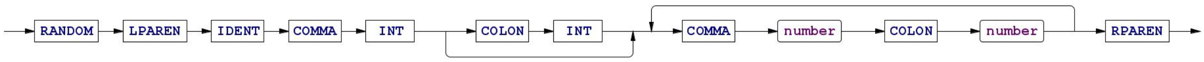
(b) General

Figure D.2: Syntax diagrams representing the extended context-free grammar specified in the `JADAParser` source file for the *General* language construct.

D.2.2 Extended Variables Language Construct



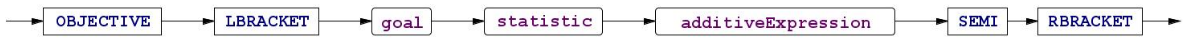
(a) Decision Variable (decisionVar)



(b) Random Variable (randomVar)

Figure D.3: Syntax diagrams representing the extended context-free grammar specified in the `JADAParser` source file for the *Variables* language construct.

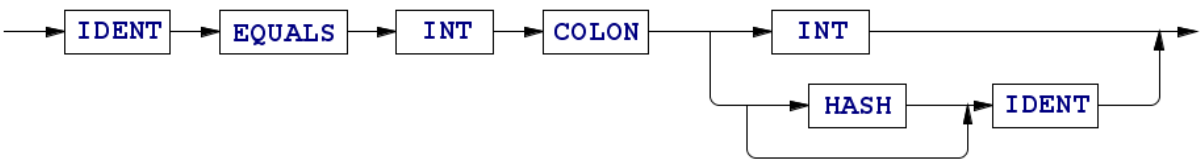
D.2.3 Extended Objective Language Construct



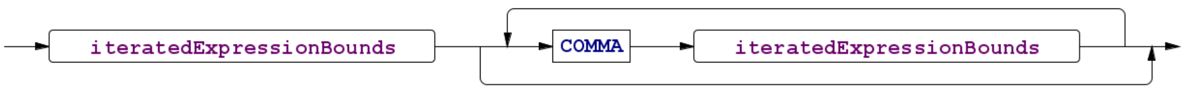
(a) Objective

Figure D.4: Syntax diagrams representing the extended context-free grammar specified in the `JADAParser` source file for the *Objective* language construct.

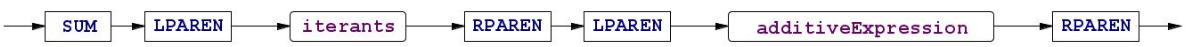
D.2.4 Extended Arithmetic Expressions Constructs



(a) Iterated Expression Bounds

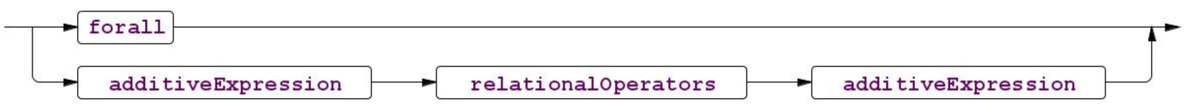


(b) Iterants

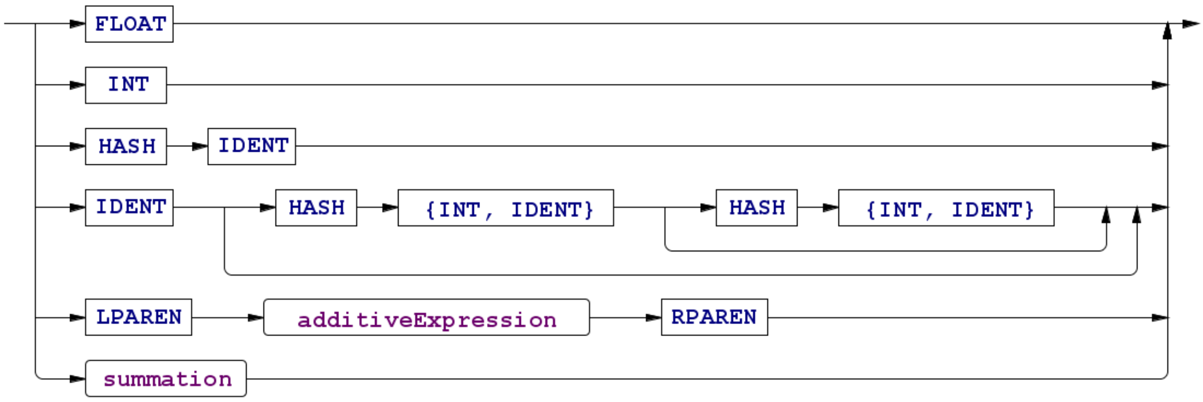


(c) Summation

(d) Universal Quantification (forall)



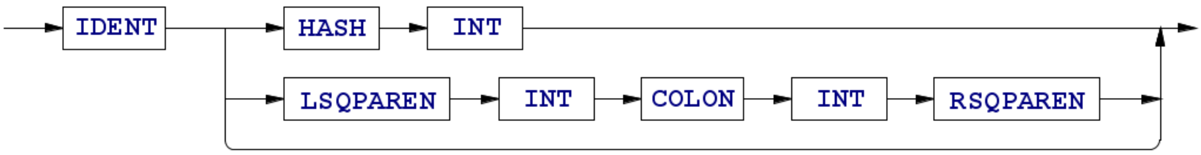
(e) Relational Expression Expression



(f) Atomic Expression

Figure D.4: Syntax diagrams representing the extended context-free grammar specified in the `JADAParser` source file for the *arithmetic expressions*.

D.2.5 Extended Sample Data File Constructs



(a) Sampled Variable

Figure D.5: Syntax diagrams representing the extended context-free grammar specified in the `SampleDataParser` source file for the *sample data*.

E

Additional Case Study: An Inventory Management System

Suppose the existence of an single product inventory system made up I factories, where all the goods produced by the factories are delivered to a warehouse. The decision-maker's objective is to meet a random demand at a minimum expected production cost. Given a planning horizon of T bi-weekly periods, we assume the following model:

- Random variable ξ_t denotes the stochastic demand of the produced good in period.
- The cost of producing one unit of the product at factory i is representable as $c_{i,t}$.
- Quantity $\bar{x}_{i,t}$ is the maximum production capacity of factory i.
- Decision variable $x_{i,t}$ is the amount of goods produced by factory i.
- $\bar{x}_{tot,i}$ determines the cumulative production capacity over the total planning horizon for factory i. We make the assumption that $\bar{x}_{tot,i} < \sum_{t=1}^{T} \bar{x}_{i,t}$.

The other static parameters of the model are:

- the initial inventory level $x_{0,wh}$.
- $\bar{x}_{wh}$ and $\hat{x}_{wh}$, which represent the permitted maximum and minimum inventory levels respectively.

The stochastic demand is modelled as a random vector $\xi = (\xi_1, \cdots, \xi_T)$ that follows a uniform distribution. The support Ξ for its probability measure $\mathbb{P}$ defined as:

$$\Xi = [(1-\theta)\,\xi^*\,\varsigma_t,\ (1+\theta)\,\xi^*\,\varsigma_t]_{t=1}^{T}, \tag{E.0.5.1}$$

where ξ^* denotes the average demand and θ represents the variability in demand. The seasonability factor ς_t captures the expectation of spring having the highest demands through the following definition:

$$\varsigma_t = 1 + \frac{1}{2}\sin\left[\frac{\pi}{12}(t-1)\right]. \tag{E.0.5.2}$$

We can formulate the problem description as the following stochastic program:

$$
\begin{aligned}
&\textit{minimise}\ \mathbb{E}\left[\sum_{t=1}^{T}\sum_{i=1}^{I} c_{i,t}\, x_{i,t}(\xi^{t})\right] \\
&\left.\begin{aligned}
\textit{subject to}\ & x_{i,t} \in \mathcal{L}^2_{t,1} \\
& 0 \le x_{i,t}(\xi^{t}) \le \bar{x}_{i,t}(\xi^{t}) \\
& \sum_{t=1}^{T} x_{i,t}(\xi^{t}) \le \bar{x}_{tot,i} \\
& \hat{x}_{wh} \le x_{0,wh} + \sum_{s=1}^{t}\sum_{i=1}^{I} x_{i,s}(\xi^{s}) - \sum_{s=1}^{T} \xi_s \le \bar{x}_{wh}
\end{aligned}\right\}\quad \forall i \in I, \forall t \in \mathbb{T}
\end{aligned}
\tag{E.0.5.3}
$$

Bibliography

[1] **Dantzig, G.B.** Linear Programming Under Uncertainty. *Management Science*, v.1 n.3 (2004) pp.197-206.

[2] **Kuhn, D., Wiesemann, W., and Georghiou, A.** Primal and Dual Linear Decision Rules in Stochastic and Robust Optimization. *Optimisation Online* (2009), URL `http://www.optimization-online.org/DB_HTML/2009/02/2218.html`.

[3] **Dyer, M., and Stougie, L.** Computational Complexity of Stochastic Programming Problems. *Mathematical Programming*, v.106 n.3 (2006), pp 423-432.

[4] **Ferguson, A.R., and Dantzig, G.B.** The Allocation of Aircraft to Routes: An Example of Linear Programming Under Uncertain Demands. *Management Science*, v.3 n.1 (1956), pp.45-73.

[5] **Carino, D.R., Kent, T., Meyers, D.H., Stacy, C., Sylvanus, M., Turner, A.L., Wantanabe, K., and Ziemba, W.T.** The Russell-Yasuda Kasai Model: An Asset/Liability Model for a Japanese Insurance Company Using Multistage Stochastic Programming. *Interfaces*, v.24 n.1 (1994), pp.29-49.

[6] **Thiele, A.** Robust Stochastic Programming with Uncertain Probabilities. *IMA Journal of Management Mathematics*, v.19 n.3 (2008), pp.289-321.

[7] **National Aeronatics and Space Administration (NASA).** Mach Number. *Glenn Research Centre* (2009), URL `http://www.grc.nasa.gov/WWW/K-12/airplane/mach.html`

[8] **Padula, S., Gumbert, C., and Li, W.** Aerospace Applications of Optimization Under Uncertainty. *Optimization and Engineering*, v.7 n.3 (2008), pp.317-328.

[9] **Sokol, J.S.** A Robust Heuristic for Batting Order Optimization Under Uncertainty. *Journal of Heuristics*, v.9 n.4 (2003), pp.353-370.

[10] **Denton, B., Viapiano, J., and Vogl, A.** Optimization of Surgery Sequencing and Scheduling Decisions Under Uncertainty *Health Care Management Science*, v.10 n.1 (2007), pp.13-24.

[11] **Benders, J.F.** Partitioning Procedures for Solving Mixed-variables Programming Problems. *Numerische Mathematik*, v.4 n.1 (1962), pp.238-252.

[12] **Birge, J.R.** Stochastic Programming Computation and Applications. *INFORMS Journal on Computing*, v.9 n.2 (1997), pp.111-133.

[13] **Eppen, G.D., Martin, R.K., and Schrage, L.** A Scenario approach to Capacity Planning. *Operations Research*, v.37 n.4 (1989), pp.517-527.

[14] **Powell, W.B., Marar, A., Gelfansd, J., Bowers, S.** Implementing Real-Time Optimization Models: A Case Application From The Motor Carrier Industry. *Operations Research*, v.50 n.4 (2002), pp.571-581.

[15] **Birge, J.R., and Rosa, C.H.** Modelling Investment Uncertainty in the Costs of Global CO_2 Emission Policy. *European Journal of Operations Research*, v.83 n.3 (1995), pp.466–488.

[16] **LP Solve.** MPS File Format.
URL `http://lpsolve.sourceforge.net/5.5/mps-format.htm`

[17] **Gassmann, H.I.** The SMPS Format for Stochastic Linear Programs.
URL `http://myweb.dal.ca/gassmann/smps2.htm`

[18] **Fourer, R., Gay, D.M., and Kernighan, B.W.** AMPL: A Modeling Language for Mathematical Programming. *Management Science*, v.36 n.5 (1990), pp.519-554.

[19] **NEOS Server for Optimisation** NEOS Solvers.
URL `http://neos.mcs.anl.gov/neos/solvers/index.html`

[20] **OptiRisk System** SPInE.
URL `http://www.optirisk-systems.com/products_spine.asp`

[21] **Dupacova, J., Hurt, J., and Stephan, J.** Stochastic Modelling in Economics and Finance. *Springer, First Edition*, (2002), pp.222.

[22] **Gassmann, H.I., and Schweitzer, E.** A Comprehensive Input Format for Stochastic Linear Programs. *Annals of Operations Research*, v.104 n.1-4 (2001), pp.89-125.

[23] **Ash, R.B., and Doleans-Dade, C.A.** Probability and Measure Theory. *Harcourt Academic Press, Revised Second Edition*, v.104 n.1-4 (2000), Chp. 1,4,5.

[24] **Rao, V.** Lectures in Real Analysis (Measure Theory and Integration). *Centre For Electronics Design and Technology, Indian Institute of Science*, Lecture 5 (2009), pp.1-4.

[25] **Ushakov, N.G.** Conditional Mathematical Expectation, Encyclopaedia of Mathematics, SpringerLink.
URL `http://eom.springer.de/c/c024500.htm#c024500_00mast`

[26] **Higle, J.L.** Stochastic Programming: Optimization When Uncertainty Matters. *INFORMS Tutorials in Operations Research*, (2005), pp.6,8-14.

[27] **Birge, J.R., and Louveaux, F.** Introduction to Stochastic Programming. *Springer Series in Operations Research and Financial Engineering*, (1997), ch.1-3,7.

[28] **Rockafellar, R.T.** Lecture Notes in Optimisation Under Uncertainty. *Department of Mathematics, University of Washington*, (2001), pp.18-22.

[29] **Greene, S.** SVG icon of two standard gaming dice.
URL `http://commons.wikimedia.org/wiki/File:2-Dice-Icon.svg`

[30] **Vandenberghe, L., and Boyd, S.** Semidefinite Programming. *SIAM Review*, v.38 n.1 (1996), pp.49-95.

[31] **Shapiro, A., and Nemirovski, A.** On Complexity of Stochastic Programming Problems. *Applied Optimization*, v.99 n.1 (2005), pp.111-146.

[32] **Shapiro, A., and Nemirovski, A.** On Complexity of Multi-stage Stochastic Programs. *Operations Research Letters*, v.34 n.1 (2006), pp.1-8.

[33] **Chen, X., Sim, M., Sun, P., and Zhang, J.** A Linear Decision-Based Approximation Approach to Stochastic Programming. *Operations Research*, v.56 n.2 (2008), pp.344-357.

[34] **Georghiou, A.** A Simplified Newsvendor Problem. *Department of Computing, Imperial College London* (2008).

[35] **Valente, C., Mitra, G., Sadki, M., and Fourer, R.** Extending Algebraic Modelling Languages for Stochastic Programming. *INFORMS Journal on Computing*, v.21 n.1 (2009), pp.107-122.

[36] **The MathWorks** Object-Oriented Programming in MATLAB.
URL `http://www.mathworks.com/access/helpdesk_r13/help/toolbox/lmi/lmi.html`

[37] **The MathWorks** Using Sun Java Classes in MATLAB Software.
URL `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/f44062.shtml`

[38] **The MathWorks** Overloading Built-In Functions.
URL `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f8-784135.html#f8-790494`

[39] **The MathWorks** Preallocating Arrays.
URL `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f8-784135.html#f8-793781`

[40] **The MathWorks** When Possible, Replace for with parfor (Parallel for).
URL `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f8-784135.html#brdtujr-1`

[41] **The MathWorks** Vectorizing Loops.
URL `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f8-784135.html#br8fs0d-1`

[42] **The MathWorks** Techniques for Improving Performance.
URL `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f8-784135.html`

[43] **Gahinet, P., Nemirovski, A., Laub, A.J., and Chilali, M.** LMI Control Toolbox, For Use with MATLAB.
URL `http://www.mathworks.com/access/helpdesk_r13/help/toolbox/lmi/lmi.html`

[44] **Löfberg, J.** YALMIP.
URL `http://users.isy.liu.se/johanl/yalmip/pmwiki.php`

[45] **Tigris.org** TortoiseSVN.
URL `http://tortoisesvn.tigris.org`

[46] **Frey, J.** Tortoise SVN Wrapper, MATLAB Central.
URL `http://www.mathworks.com/matlabcentral/fileexchange/24307-tortoise-svn-wrapper`

[47] **Schünemann, U.** Requirements Specification and Development Principles for Programming Languages.
URL `http://web.cs.mun.ca/~ulf/pld/princ.html`

[48] **Birge, J.R.** Current Trends in Stochastic Programming Computation and Applications. *Department of Industrial and Operations Engineering, University of Michigan*, pp.25-27.

[49] **Ben-Tal, A., Goryashko, A., Guslitzer, E., and Nemrovski, A.** Adjustable Robust Solutions of Uncertain Linear Programs. *Mathematical Programming*, v.99 n.2 (2004), pp.351-376.

[50] **University of Alaska Fairbanks.** LL Grammars.
URL `http://www.cs.uaf.edu/~cs331/notes/LL.pdf`

[51] **StackOverflow.** Disadvantages of the Spirit Parser-Generator Framework from Boost Libraries.
URL `http://stackoverflow.com/questions/432173`

[52] **Sourceforge.net.** ANTLR Plugin for Eclipse.
URL `http://antlreclipse.sourceforge.net`

[53] **San Jose State University Singular Matrix Database.** SPNRANK Routine.
URL `http://www.math.sjsu.edu/singular/matrices/software/SJsingular/Doc/spnrank.pdf`

[54] **San Jose State University Singular Matrix Database.** SJsingular (SOURCE CODE).
URL `http://www.math.sjsu.edu/singular/matrices/software`

[55] **San Jose State University Singular Matrix Database.** Documentation for SPNRANK Routine.
URL `http://www.math.sjsu.edu/singular/matrices/software/SJsingular/Doc/spnrank.pdf`

[56] **The Math Less Traveled.** Sigma Notation.
URL `http://www.mathlesstraveled.com/?page_id=50`

[57] **Georghiou, A., Wiesemann, W., Kuhn, D.** Generalised Decision Rule Approximations for Stochastic Programming via Liftings. *Working paper, Department of Computing, Imperial College London*, (2010).

[58] **Kuhn, D., Parpas, P., and Rustem, B.** Stochastic Optimization of Investment Planning Problems in the Electric Power Industry. *Process Systems Engineering: Energy Systems Engineering, (*GEORGIADIS, .M, KIKKINIDES, E., AND PISTIKOPOULOS, E., *eds.)*, Wiley-VCH, Weinheim, v.5 (2008), pp215–230.

[59] **Frauendorfer, K., and Siede, H.** Portfolio Selection Using Multistage Stochastic Programming. *Central European Journal of Operations Research*, v.7 n.4 (2000), pp.277-289.

[60] **Perold, A.F.** Large Scale Portfolio Optimization. *Management Science*, v.30 n.10 (1984), pp.1143-1160.

[61] **Markowitz, H.** Portfolio Selection. *Journal of Finance*, v.7 (1952), pp.77-91.

[62] **Young, M.** A Minimax Portfolio Selection Rule with Linear Programming Solution. *Management Science*, v.44 n.5 (1998), pp.673-683.

[63] **Rocha, P., Kuhn, D.** Multistage Stochastic Portfolio Optimisation in Deregulated Electricity Markets Using Linear Decision Rules. *Working paper, Department of Computing, Imperial College London*, (2010).

[64] **Hadjiyiannis, M.J., Goulart, P.J., Kuhn, D.** An Efficient Method to Estimate the Suboptimality of Affine Controllers. *Working paper, Department of Computing, Imperial College London*, (2010).