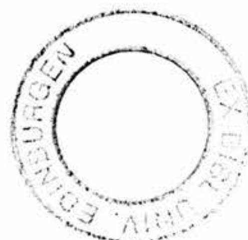
Benders Decomposition Method In Reservoir Management

Crawford S Buchanan

Doctor of Philosophy
University of Edinburgh
1999



Abstract

Multi-stage stochastic linear programming provides a framework in which to model and solve decision making problems that contain uncertain data. In this thesis the main stages in the process of modelling and solving a large-scale multi-stage stochastic linear programme (MSLP) are examined. The principal motivation for this research is the study of the electricity generation network of Southern Brazil. This network contains a high proportion of hydro-electric generation plants, and so the stochasticity of the future inflows has a large influence on decisions. The formulation of MSLPs is difficult within existing algebraic modelling languages. Many MSLPs can be formulated as a set of recurrences. We present a new algebraic modelling language, sMAGIC, that uses the recursive definition of sub-models to aid in the specification of MSLPs. The Benders Decomposition algorithm exploits the sparse structure of MSLPs, achieving a considerable reduction in the time taken to solve MSLPs over direct solution methods, such as the simplex method. In addition, the basic Benders Decomposition algorithm can be extended and is well suited to parallelisation. We present results that show that some of the extensions to the basic algorithm improve the performance of the solver in all cases, while others provide improvements only for particular test problems. The results from our parallel implementation on a network of workstations give near linear speedups. Sampling techniques can be incorporated within the Benders Decomposition method. This allows an approximation to the solution of MSLPs that are too large to solve using Benders Decomposition to be obtained. A Benders Decomposition algorithm that incorporates Monte Carlo sampling is guaranteed to converge asymptotically to the actual solution. To improve the speed of this algorithm, an additive approximation to the cost function is used to guide an importance sampling technique. We compare the use of multiple linear regression and re-calibration to update this additive approximation. In addition to the sparse structure exploited by the Benders Decomposition method, the stochastic parameters in many MSLPs exhibit Markovity between stages. This extra structure is lost when the deterministic equivalent problem is solved using Benders Decomposition. We present a modified form of the Benders Decomposition algorithm that solves such problems as model link graphs rather than as the standard "unfolded" event tree of the deterministic equivalent problem. This allows the automatic sharing of cuts during the solution of a folded problem, and so allows extremely large-scale problems to be solved using Benders Decomposition and importance sampling. In collaboration with National Power plc we have developed a family of problems with up to 120 stages and a deterministic equivalent linear programme with 2.5×10^{59} constraints and 1.4×10^{60} variables. The results presented in this thesis are from the solution to these problems.

Acknowledgements

As with any sizable piece of research, the final results in this thesis are due to the efforts of more than one person. In this respect I am greatly indebted to a great many people who have stimulated, helped, encouraged and advised me in numerous ways during the many months of work that have gone into this thesis.

Much of the work in this thesis has been done in collaboration with National Power plc. Were it not for the help of John Putney, Colin Tan and Ian Fletcher the theory in this thesis would have had very little chance of being applied to such an interesting and complex problem. Working with them on this large case study has provided me with a greater understanding and deeper insight into the pitfalls and profits of applying optimisation theory in the real world.

The significant improvement in my programming skills during my studies owe a great deal to the patience and guidance of Julian Hall. My fellow research students Erich, Theresia, Ann and Corrie have made my studies fun as well as interesting. From outside my immediate colleagues my family are owed a great deal for not doubting I would someday cease to be a student, while Ellen-Raïssa, Elaine and Stuart have been there for me through thick and thin.

Most of all I would like to thank my supervisors, Lyn Thomas and Ken McKinnon, and our colleague Tom Archibald with whom I have studied many different aspects of reservoir management problems. Their combined enthusiasm and insight into this field has provided me with a very stimulating and encouraging environment in which to work. Additionally I owe Ken McKinnon a great deal for initially firing my interest in stochastic programming, and for giving me the opportunity to study for a PhD in this field.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Crawford S Buchanan*)

Table of Contents

List of Figures v

List of Tables vi

Chapter 1 Introduction 1

Chapter 2 Towards Stochastic AMLs 7

2.1 Introduction 7

2.2 The Role Of Algebraic Modelling Languages 8

2.3 Modelling Stochastic Linear Programmes 11

2.3.1 SLPs and their Deterministic Equivalents 12

2.3.2 Proposed Stochastic Extensions To AMLs 16

2.3.3 Hybrid Systems 21

2.3.4 SMPS: As It Is And How It Might Develop 25

2.4 sMAGIC: A Recursive AML 30

2.4.1 Motivation & Development 30

2.4.2 Details Of Language 33

2.5 Conclusions 41

Chapter 3 Benders Decomposition Method 42

3.1 Introduction 42

3.1.1 History 43

3.1.2 Simplex Algorithm 44

3.2 Derivation of BDM 45
3.2.1 Two-stage Benders Decomposition Method 45
3.2.2 Uncertain data 51
3.2.3 Multi-stage Problems 54
3.3 Algorithmic Extensions 58
3.3.1 Multi-cuts 58
3.3.2 Cut Sharing 59
3.3.3 Hot-starts 61
3.3.4 Preliminary Cuts 61
3.3.5 Tree Traversal 63
3.3.6 Results for Extensions 64
3.4 Parallelisation of Benders Decomposition 68
3.4.1 Parallel Algorithms 68
3.4.2 Parallel Benders Decomposition Algorithms 70
3.4.3 Results for Parallel Algorithm 74
3.5 Cut sharing & Graph Notation 80
3.5.1 Motivation & Implementation 80
3.5.2 Results 85
3.6 Conclusions 88

Chapter 4 Sampling Techniques & Benders Decomposition 90

4.1 Introduction 90
4.2 Using Sampling Techniques 92
4.2.1 External Sampling Techniques 93
4.2.2 Internal Sampling Techniques 94
4.2.3 Issues introduced by Sampling 99
4.3 Monte Carlo Sampling & Its Extensions 100
4.3.1 Standard Monte Carlo Sampling 100

4.3.2 Variance Reduction Techniques 101

4.4 Implementation of Importance Sampling 106

4.4.1 Additive Model 106

4.4.2 Updating the Additive Approximation Coefficients 109

4.4.3 Multiple Linear Regression 111

4.4.4 Results 115

4.5 Sample Size and Stopping Rules 119

4.5.1 Results 125

4.6 Conclusions 130

Chapter 5 The Brazilian Electricity Network 132

5.1 Introduction 132

5.2 Generation Network Model 133

5.2.1 Transmission Network 133

5.2.2 The Hydro Network 136

5.3 Inflow Model 141

5.4 sMAGIC implementation of model 144

5.5 Conclusions 149

Chapter 6 Conclusions & Further Research 151

Appendix A Full Hydro Model 157

Appendix B sMAGIC Hydro Model 162

References 165

List of Figures

2.1 An Event Tree 15
2.2 Multi-factory Example 34
2.3 SMAGIC Production and Inventory Problem 37
2.4 A Model Link Graph 38
2.5 Unfolded Tree Representation 39

3.1 The Recourse Function α 47
3.2 Four Iterations of Benders Decomposition 51
3.3 An Event Tree 55
3.4 Allocation of Sub-Problems to Processors 71
3.5 Unfolded Event Tree for Reservoir Problem 81
3.6 Folded Event Tree for Reservoir Problem 82

5.1 A Transmission Network 134
5.2 A Load Duration Curve 135
5.3 A Hydro Network 137
5.4 sMAGIC:Model Declaration 144
5.5 sMAGIC:Constraint Declaration 145
5.6 sMAGIC:Objective Declaration 146
5.7 A Folded Reservoir Problem 147

List of Tables

3.1 4 & 6 Stage Problems, Multi-Cuts 65

3.2 4 & 6 Stage Problems, Aggregated Cuts 66

3.3 Expected Rewards and Percentage Errors 76

3.4 CPU Time(s) on a 70MHz SUN Sparc 5 77

3.5 Speedups on a Network of 70MHz SUN Sparc 5 78

3.6 Details of Serial Solutions 78

3.7 Details of Parallel Solutions 78

3.8 Unfolded vs Folded 4 Stage Problem 86

3.9 Unfolded vs Folded New 4 Stage Problem 87

4.1 Using Multiple Linear Regression and Re-Calibration to Update the Additive Approximation 117

4.2 4 Stage, Morton's $\mathcal{O}(\ln^2 k)$ Sample Formula 126

4.3 6 Stage, Morton's $\mathcal{O}(\ln^2 k)$ Sample Formula 126

4.4 12 Stage, Morton's $\mathcal{O}(\ln^2 k)$ Sample Formula 126

4.5 4 Stage, Fixed Sample Sizes 128

4.6 6 Stage, Fixed Sample Sizes 128

4.7 12 Stage, Fixed Sample Sizes 129

Chapter 1

Introduction

Some of the data in decision making problems are inherently uncertain. Planning models that try to represent these uncertain parameters by deterministic approximations often lead to unsatisfactory results. However, if the distribution of the uncertain parameters is known, then stochastic programming allows these uncertain parameters to be modelled as random variables with given distributions.

Many decision making problems require decisions to be made more than once. Normally the decisions have to be made at regular intervals in the future. As most planning models concern the allocation of scarce resources between competing demands, the allocation of resources in one stage affects the availability of these resources in the future.

Multi-stage stochastic linear programmes provide a framework within which to model problems of this form. They capture both the dynamic and the uncertain nature of the planning problem. Their suitability in modelling many varied types of planning problems has seen their increased use in the last decade. Multi-stage stochastic linear programmes have been applied to problems in many varied fields. E.g. forestry management, the scheduling of hydro-electricity generation, asset liability management, vehicle routing and maintenance scheduling for electricity generation networks.

There are several stages to the process of modelling and solving a problem using a multi-stage stochastic linear programme. The first stage is the formulation of the model. This is a mathematical model that captures the form of the problem. It is generic as the dimensions of a particular problem are parameters of the model, as are the data that specify one particular instance of the model. The next stage is the collection and transformation of data. The data may come from historical

observations, or the decision maker can generate it. The data are then combined with the model and an instance of the problem is generated.

The third stage of this process is the solution of the problem. Once the problem has been solved, the analysis of the results will suggest whether or not the model or the data have to be re-formulated. If alterations need to be made, then the process is repeated. This will require either the model or the data, or both, to be altered and amended.

The aims of this thesis are twofold. The first requirement is to solve a real-world problem for National Power plc. As co-sponsors of this work, in conjunction with EPSRC as part of a CASE award, National Power plc want to solve the problem of modelling the generation of electricity in Southern Brazil over a medium to long term planning horizon. In addition to solving this particular problem, they are keen to gain a broader understanding of hydro-electricity planning problems. At present this is an area of electricity production about which they have very little experience.

The second aim of this thesis is to improve the modelling and solution approaches used to generate and solve more general multi-stage stochastic linear programmes. It looks at several of the stages involved in formulating and solving these problems in turn. For each it adds to the current methods used for that stage. The first of these is the formulation of multi-stage stochastic linear programmes. The initial formulation of a model is as a mathematical problem. This is a symbolic representation of the problem that uses mathematical notation to represent the relevant elements of the problem. Once this version of the model has been specified, it is entered into a computer.

Algebraic modelling languages allow mathematical models to be input into a computer in a format that is close to the mathematical specification of the model. This specification uses summation, index notation and variable names that are associated with data files to generate a compact and generic formulation of the mathematical model. When the algebraic modelling language is presented with particular data for the parameters of the model an instance of the problem is generated.

At present there exist algebraic modelling languages that allow the specification of mixed integer linear programmes and linear programmes. In addition to allowing mathematical problems to be specified in a natural manner, algebraic modelling languages can also perform consistency checks on the model and the data that are presented to it. However, none of the currently available algebraic

modelling languages have any features that allow the specific formulation of multi-stage stochastic linear programmes. This has resulted in the use of multi-stage stochastic linear programmes to be confined mainly to the research community. The formulation of multi-stage stochastic linear programmes in existing algebraic modelling languages is a complicated and cumbersome task.

To overcome this drawback we have developed sMAGIC, a novel algebraic modelling language that is designed specifically to allow the formulation of multi-stage stochastic linear programmes. It does this through sub-models that are defined recursively. In addition to being suitable for the formulation of multi-stage stochastic linear programmes, sMAGIC is a fully functional algebraic modelling language with which mixed integer linear programmes and linear programmes can be formulated. The use of recursively defined sub-models adds extra modularity and flexibility to these models, as well as facilitating the specification of multi-stage stochastic linear programmes.

Once a problem is formulated as a multi-stage stochastic linear programme and combined with appropriate data, the algebraic modelling language generates the instance of the problem that is to be solved. Multi-stage stochastic linear programmes of real-world problems are very large linear programmes, and are too large to be solved by direct algorithms, such as the simplex or interior point methods. However, multi-stage stochastic linear programmes have a significant amount of exploitable structure.

Decomposition methods exploit the structure of linear programmes. Benders Decomposition is a solution algorithm that decomposes multi-stage stochastic linear programmes into much smaller sub-problems. These small linear programmes are solved iteratively. The algorithm provides upper and lower bounds on the optimal solution that converge monotonically. At each stage within a multi-stage stochastic linear programme, the decisions taken affect the possible decisions in future stages. This can be summarised by considering the future costs as a function of the decisions made in the current stage. Benders Decomposition builds an approximation to the future costs at each stage of the problem. It does this by proposing policies to future stages. The answers to these decisions are used to update the approximation to the future costs. However, the number of sub-problems in a problem increases exponentially in the number of stages. Therefore many multi-stage stochastic linear programmes resulting from real-world problems are too large, or would take too long, to solve even with decomposition algorithms.

Sampling techniques allow very large-scale programmes to be solved to an accuracy that is only limited by the time allocated to their solution. Rather than solve the entire problem, sampling techniques select a small set of possible future scenarios. These are solved and an estimate of the actual answer is obtained. The statistical properties of this estimate are used to guide whether to re-sample, having increased the number of samples used to estimate the answer, or whether to stop. The alternative to using sampling techniques with very large-scale programmes is to reduce the detail of the model so that the resulting programmes can be solved accurately. However, the drawback in doing this is that the optimal solution is only ever the exact solution to an approximation of the actual problem.

When combining sampling techniques within Benders Decomposition the likelihood of a sub-problem being solved, and so contributing to the approximation of the future costs of the previous stage is small. Therefore it is necessary to share information about future costs across stages so that some information about the future costs is known at all sub-problems in the problem. An alternative to sharing information across stages is available when an extra level of structure is present within the original multi-stage stochastic linear programme.

The range of possible values for the uncertain parameters in one stage often does not depend upon the range of values in the previous stage. However, the likelihood that the stochastic parameters take particular values from this range may depend upon what values they took in the previous stage, but the range of possible values does not. When problems have this Markovian property, they also contain a more exploitable structure than is present in an arbitrary multi-stage stochastic linear programme. Essentially the future looks the same regardless of which state the problem is in at a particular stage.

It is possible to exploit this structure by "folding" such problems. When this is not done each sub-problem is repeated across a stage with one instance for each of the sub-problems in the previous stage. As these are the same sub-problem, there is a large amount of repetition in both the sub-problems used to define the problem within the solver, and also in the information that is required to approximate the future costs. By folding multi-stage stochastic linear programmes, the number of sub-problems is reduced by a factor equivalent to the number of sub-problems in the previous stage. Hence at later stages, the reduction in the number of sub-problems within the stage is very significant.

This thesis consists of the following chapters. In Chapter 2 we consider the possible ways that the modelling of multi-stage stochastic linear programmes

can be made easier. We review various proposals for the extension of algebraic modelling languages. These extensions have been suggested as ways of allowing multi-stage stochastic linear programmes to be formulated easily. As well as these various extensions to existing algebraic modelling languages, we also review two systems that are polished implementations of the type of ad hoc methods researchers currently use to specify multi-stage stochastic linear programmes.

After considering the extensions to existing algebraic modelling languages and hybrid systems, we outline the use of recursive sub-models within sMAGIC, and how these provide a natural way in which to formulate multi-stage stochastic linear programmes. We also consider the standard format for multi-stage stochastic linear programmes, SMPS, and consider how it could be extended to allow the specification of folded multi-stage stochastic linear programmes.

Our implementation of Benders Decomposition is discussed in Chapter 3. We present the general Benders Decomposition algorithm and how it can be extended to enhance its performance. Results are presented to determine which of these extensions to the basic algorithm are the most beneficial. We also discuss the parallel implementation of Benders Decomposition based methods. Our implementation has been implemented on a local network of workstations, and results are presented which show that the structure of multi-stage stochastic linear programmes means that they are well suited to solution in parallel. Finally in this chapter we discuss the folding of multi- stage stochastic linear programmes whose stochastic parameters are independent between stages. We highlight how this folding means that cuts can be shared across stages, and show that they can be solved by a slightly modified version of Benders Decomposition. This solver is at the heart of our implementation of Benders Decomposition using sampling techniques.

In Chapter 4 we discuss the various ways in which sampling techniques can be combined with Benders Decomposition to make very large-scale multi-stage stochastic linear programmes tractable. After discussing alternative ways of incorporating sampling techniques, we look at the method of Monte Carlo sampling and its extensions. We discuss our implementation of importance sampling, a way of reducing the variance of sample mean estimates generated through Monte Carlo sampling. Two methods for updating the additive approximation that is used to guide the importance sampling are compared. One performs the re-calibration of the additive approximation at regular intervals during the sampling algorithm. The other uses multiple linear regression to update the additive approximation continuously. Finally in this chapter, we

look at stopping rules for sampling based methods. These rules govern when a sampling algorithm should stop, and after iterations in which the algorithm does not stop, then by how much to increase the sample size in the next iteration.

Many of the results in this thesis have been motivated by work carried out in collaboration with National Power plc. In Chapter 5 we present our model of the electricity generation network of Southern Brazil. This generation network has a high proportion of hydro-electricity plants, about 85% of the total capacity, and so multi-stage stochastic linear programming provides an ideal framework in which to model such a problem. The network has a high dependency upon the future rainfall, which is highly uncertain.

We discuss the parts of the model and also our method for generating future inflow patterns, the extra water available in the future which has come from out with the hydro-network. The model splits into two distinct but interconnected parts; the hydro-network and the transmission network.

The first consists of reservoirs with hydro-electricity generation plants connected. These are connected along valleys, with one feeding into another further downstream. Our model has several cascades of reservoirs that are interconnected. The transmission network consists of a collection of electricity demand nodes and inter-connectors between them. In addition to a local demand for electricity, there are also thermal and hydro-electricity generation plants attached to each node. If the supply of electricity from local generation plants is not enough to meet the demand, then electricity can be transferred from other nodes in the transmission network along the interconnecting lines. We also highlight where the methods developed in the previous chapters are useful in the formulation and solution of this problem.

In Chapter 6 we present the conclusions to this thesis. These highlight how the results presented in this thesis have come primarily from the investigation of one particular problem for National Power plc. However the results are applicable to the various stage involved in modelling and solving more general multi-stage linear programmes.

Chapter 2

Towards Stochastic AMLs

2.1 Introduction

Algebraic Modelling Languages (AMLs) allow people to model problems in a language which is somewhere between the way it is written mathematically and the way in which a solver requires the problem. They also translate a model from the former format to the latter.

In the next section we discuss the role of an AML in the mathematical modelling of real-world problems, and consider how this motivates the capabilities of the AML. At present there is no AML available that has the generic capacity to model stochastic programming problems. After introducing multi-stage stochastic linear programmes, we review some proposed ways of extending current AMLs to allow the modelling of this increasingly popular type of large-scale problems. We also look at the current standard format for specifying MSLPs, and how it might be extended. Finally, we describe our modelling language, sMAGIC, and demonstrate how its distinct and novel methods are useful in the modelling of multi-stage stochastic linear programmes.

2.2 The Role Of Algebraic Modelling Languages

The process of modelling and solving a real-world problem can be broken down into several stages. These stages are iterative in nature, and as the process may be repeated several times, often require a large amount of effort. An algebraic modelling language that allows the easy manipulation of a model will greatly speed up this process. It allows the modeller to write the model in an algebraic format that is similar in spirit to the way in which she would write it using mathematical notation.

The process can be divided into three main parts. The first stage is to construct an accurate mathematical model for the problem under consideration. The coefficients of the model are determined from historical or collected data, usually after some form of manipulation. The second stage is to optimise the resulting problem. If the solver gives unexpected results, then this suggests that either the coefficients or the model, or both, are incorrect. Therefore in the third stage the model is re-formulated. If the problem is thought to lie in the coefficients, then either the raw data or the manipulations of it are re-considered. Otherwise the form of the model is considered. Once a new version of the model has been formulated, the problem is generated and re-solved. This cycle is repeated until the modeller feels that the results are sensible.

The first formulation of a model is as a mathematical problem. This is what a modeller wants to input into a computer. Hence the first role of an AML is to provide a framework in which a mathematical model can be formulated and manipulated. This mathematical model is a symbolic formulation, consisting of variables, constraints, bounds and an objective function. It is a generic statement of the model as no specific data is included in it. Therefore different problems, that share the same form but have different data, can be generated from the same model.

This is called the *modeller's form* of the problem in Fourer (1983) [29]. It is written in a way that is similar to the original mathematical formulation of the problems, and so is easily read by any modeller. The use of comments, summation and set notation help to make compact, generic models. Alteration of the parameters that define the sets in the problem results in problems of different sizes. While the model remains the same, the problems generated can be of widely varying dimensions. Once the user is finished with the specification of the model, the second role of an AML is to combine the model with user-specified data to generate an instance of the model that is ready to be solved.

This is known as the *algorithm's form* of the model. It is a compact representation of the problem that is tailored to the requirements of the solver. As a solution method has no concept of how the model came to be formulated in a particular way, or what the various parts of the model represent, the algorithm's form of the model consists only of the numerical data that define the problem, in a format recognisable to a solver. It is just one instance of the model represented by the symbolic modeller's form. As the algorithm's form explicitly contains the coefficients of the problem, it grows with the size of the original problem, and so for real world problems it is very large. The hydro-electricity generation problems discussed in Chapter 5 contain hundreds of thousands of rows and columns.

Working with such a form of the model is necessary for a solution method but not for a modeller. The algorithm's form is neither easy to read nor is it easy to alter. It is the role of AMLs to convert from the mathematical, compact and readable modeller's form to the algorithm's form. Prior to the introduction of AMLs, this task was done by matrix generators (MGs). These were dedicated pieces of code which generated the algorithm's form of particular models, given particular numerical data. Since they were written to generate problems of a specific form, to change the form of the model, or to alter some of the numerical data required the modeller to actually re-write parts of the code. This was a very cumbersome process, both error-prone and time consuming.

The introduction of AMLs in the 1980s, such as AMPL [28], GAMS [14] and XPRESS-MP [3], was prompted by the difficulty of using MGs. All of these AMLs can be used to specify linear programmes, and some also have the capacity to model mixed integer programmes. The growing use of mathematical programming in the last twenty years can be linked to the increasing sophistication and power of AMLs as well as the increase in affordable computing power. In addition to allowing mathematical models to be specified in the modeller's form, algebraic modelling languages have many additional benefits over matrix generators.

These are mainly due to the fact that an AML provides a framework in which to develop a model. In addition to being able to specify models in a symbolic fashion, models can also readily be re-edited and checked for consistency. This includes checking that the definitions of all parts of a model are consistent with their use in the model, that there are no syntactical inconsistencies and that the data supplied by the user is of the correct format and dimensions for the model. In order to make the modeller's form of the model readable, full names are used for variables, constraints and the arrays of coefficients of the model. In addition,

comments can be included in the model to highlight specific features. All of these features are absent from matrix generators.

There is no standard algorithm's form for problems. While a large number of solvers expect problems in a standard format, there are often other less standard formats in use. A notable example of a *de facto* standard is the MPS format that is used for linear programmes. Typically the different formats have their origins in the diverse fields of research where mathematical programmes are used. Therefore, AMLs can normally generate problems in one of several formats. Often the best solver to use for a particular problem is not known in advance. Hence, the ability to generate problems in one of several formats makes the job of switching between solvers much easier. If a problem was generated by a matrix generator, then the output would be fixed to one particular type, unless the user re-wrote parts of the code.

The use of stochastic linear programmes as a tool for decision-making has increased over the last decade. These problems are linear programmes that contain uncertain data. A distribution of possible values model the uncertain data. Their use, however, was mainly confined to the research community. One of the main impediments to their increased development and use, both within and outside the research field, is the lack of suitable AMLs with which to develop these models. Just as the tools and facilities that AMLs bring to the formulation of linear and mixed integer programmes have increased their use, so the development of AMLs capable of specifying stochastic linear programmes might bring to this burgeoning field of study.

The lack of generic facilities in currently available AMLs for modelling stochastic linear programmes (SLPs) forces modellers to generate SLPs in one of two ad hoc manners. Either the entire SLP is generated within an extant AML, or the problem is decomposed and an AML is used to generate the smaller parts of the model. The first of these two options leads to very complicated models, as the modeller has to introduce and manage a large indexing system. The second method usually requires less intricate AML models, but the same model is generated repeatedly using different sets of data. This requires careful marshalling of the data, and the careful construction of the original problem from the constituent parts. For these reasons neither of these methods is satisfactory.

2.3 Modelling Stochastic Linear Programmes

The data used in a linear programme is often uncertain. In the past modellers have overcome this by instead using approximations to the values of the data. However, the modeller usually has an idea about the probability distribution which the data values can take. This information can be included in the model. The resulting problem is called a Stochastic Linear Programme (SLP). In the last decade interest in the formulation and solution of SLPs has increased. Prior to this, the main ways of dealing with such uncertain data were either to look at the problems generated by setting the data to particular values, or to look at a problem that used the mean values of the uncertain coefficients.

An SLP that has all of its uncertain parameters replaced by their means is known as the *expected value problem*. The solution of the expected value problem is normally a conservative estimate of the solution to the actual stochastic problem. If the original problem is very stochastic, then the difference between the stochastic solution and the solution of the expected value problem is large, see Birge (1995) [9]. Another way of dealing with the uncertain data in SLPs is to look at "What-if" scenarios. Here the uncertain parameters are replaced by values from their distributions giving possible scenarios that might occur. The modeller's ideas about the problem guide the choice of parameter values. A typical scenario is to set the availability of a resource at its lowest level, while the demand for the products they are used to make is at its highest. This is an example of a *worst case* scenario.

Both of these methods result in a collection of solutions, whether it is just the mean case, or various scenarios that the modeller hopes capture the spread of possible situations. From these she has to make an informed guess at what are the best decisions to make, given the uncertain future. Both of these methods have their drawbacks, the expected value problem gives consistently poor results for problems which are genuinely stochastic, while the other method requires a large amount of expert knowledge and experience to assist in the selection of good scenarios to analyse. These disadvantages are both caused by the loss of information due to the selection of only part of the probability distribution. Stochastic linear programmes are preferable as they offer a way to retain this information.

In the following part of this section we discuss the general form of SLPs and their deterministic equivalents. We then look at proposed extensions to the currently available algebraic modelling languages, as well as two hybrid systems that may

be of use to modellers in the interim. In the final part of this section, we look at the SMPS format for generating the algorithm's form of SLPs, and consider some possible extensions to it.

2.3.1 SLPs and their Deterministic Equivalents

There are many different ways in which random data can be incorporated into a linear programming problem. Bounds, constraints, costs, or coefficients may all be random parameters. However, it is constructive to impose a classification system on these different types of SLP problems; so as to see the issues involved in modelling the different forms.

A comprehensive and detailed taxonomy of stochastic linear programming problems is given in Gasmann & Ireland (1996) [34]. Their taxonomy is partly inspired by, and so mirrors, the different solution methods that have been developed for SLP problems. The three main divisions that they propose are: chance constrained problems, recourse problems, which they further sub-divide, and hybrid problems. The final class is composed of problems that do not fall squarely into either of the first two classes, but instead can be expressed as combinations of the other classes.

Chance constrained problems are mathematical programming problem where one or more constraints need only be satisfied with a certain probability. For instance, in an inventory management problem, the demand for a particular product might need to be met at least 90% of the time. This is a simple probabilistic constraint. The stochastic nature of chance constrained problems is made more complicated with the introduction of joint chance constraints. In addition to probabilistic constraints, coefficients may also be stochastic. The reader is directed to Kall & Wallace (1994) [48] for a wider introduction to chance constrained programming.

The second class of problems, *recourse* problems, are the main type of SLPs that we discuss in this thesis. There are two reasons for this. The first is that our solution method, which is discussed in Chapter 3, is applicable to this type of stochastic linear programme, and the second is that this is the largest and most common type of SP problem.

One way of writing a multi-stage stochastic linear programme (MSLP), a recourse problem, is as follows.

$$\begin{aligned} \text{minimise} \quad & c_1 x_1 + I\!E_{\omega_2} Q_2(x_1) \\ \text{subject to} \quad & A_1 x_1 = b_1 \\ & l_1 \leq x_1 \leq u_1, \; x_1 \in I\!R^{n_1} \end{aligned} \tag{2.1}$$

with the functions Q_t, $t = 4, \ldots, T-1$, being defined as

$$\begin{aligned} Q_t(x_{t-1}) = \min \quad & (\mathbf{c_t} x_t + I\!E_{\omega_{t+1}} Q_{t+1}(x_t)) \\ \text{subject to} \quad & \mathbf{E_t} x_{t-1} + \mathbf{A_t} x_t = \mathbf{b_t} \\ & \mathbf{l_t} \leq x_t \leq \mathbf{u_t}, \; x_t \in I\!R^{n_t}; \end{aligned} \tag{2.2}$$

and Q_T being defined by

$$\begin{aligned} Q_T(x_{T-1}) = \min \quad & (\mathbf{c_T} x_T) \\ \text{subject to} \quad & \mathbf{E_T} x_{T-1} + \mathbf{A_T} x_T = \mathbf{b_T} \\ & \mathbf{l_T} \leq x_T \leq \mathbf{u_T}, \; x_T \in I\!R^{n_T}. \end{aligned} \tag{2.3}$$

The operator $I\!E_{\omega_t}$ denotes the expectation with respect to the tth-stage random variables, and any of the **bold-face** entries may be random. The state of the system in the tth stage, ω_t, is a vector consisting of the random variables in this stage, and can be regarded as a random variable defined in a canonical probability space $(\Omega, \mathcal{F}, P)$.

This formulation highlights the important aspect of multi-stage problems, namely that decisions made in one time stage affect those in the following time stages. The functions Q_t, $t = 2, \ldots, T$ are *recourse* functions. They quantify the costs incurred in the next stage and are conditional on the decisions made in the current stage. The recurrences given above can be re-written as the following single multi-stage stochastic linear programme.

$$
\begin{array}{ll}
\text{minimise} & c_1x_1 \quad +\mathbb{E}_2Q_2(\mathbf{c_2}x_2 \quad +\mathbb{E}_3Q_3(\mathbf{c_3}x_3 \quad +\ldots \quad +\mathbb{E}_TQ_T(\mathbf{c_T}x_T)\ldots)) \\
\text{subject to} & \\
& \begin{array}{lllllll}
A_1x_1 & & & & & = & b_1 \\
\mathbf{E_2}x_1 & + \; \mathbf{A_2}x_2 & & & & = & \mathbf{b_2} \\
& \mathbf{E_3}x_2 & + \; \mathbf{A_3}x_3 & & & = & \mathbf{b_3} \\
& \ddots & \ddots & & & & \vdots \\
& & & \mathbf{E_T}x_{T-1} & + \; \mathbf{F_T}x_T & = & \mathbf{b_T}
\end{array} \\
& l_1 \leq x_1 \leq u_1, \; \mathbf{l_t} \leq x_t \leq \mathbf{u_t}, \;\; t = 2, \ldots, T
\end{array} \tag{2.4}
$$

Any of the data presented above in **bold-face** can be random, but for many real-world problems the actual number of random parameters is limited. When the distributions of the stochastic elements are discrete, or have been discretised from continuous distributions, the *deterministic equivalent* problem can be formed, see Wets (1974) [62]. The deterministic equivalent problem is the form of the problem used by most solutions methods.

In example (2.4), we assume that the number of possible values for the random variable ω_t in the tth-stage, Ω_t is finite, i.e. $|\Omega| < \infty$. Each realisation of the random parameters in one stage constitutes the coefficients in a sub-problem. The *ancestor function*, $\theta(\omega_t)$, assigns the immediate successor, a sub-problem in the previous stage, to a sub-problem in the current stage. The deterministic equivalent of (2.4) can be formulated as

$$
\begin{array}{ll}
\text{minimise} & c_1x_1 + \sum\limits_{\omega_2\in\Omega_2} p_2^{\omega_2}c_2^{\omega_2}x_2^{\omega_2} + \sum\limits_{\omega_3\in\Omega_3} p_3^{\omega_3}c_3^{\omega_3}x_3^{\omega_3} + \ldots + \sum\limits_{\omega_T\in\Omega_T} p_T^{\omega_T}c_T^{\omega_T}x_T^{\omega_T} \\
\text{subject to} & \begin{array}{lll}
A_1x_1 & = & b_1, \\
E_2^{\omega_2}x_1 + A_2^{\omega_2}x_2^{\omega_2} & = & b_2^{\omega_2}, \; \omega_2 \in \Omega_2 \\
\quad E_3^{\omega_3}x_2^{\theta(\omega_3)} + A_3^{\omega_3}x_3^{\omega_3} & = & b_3^{\omega_3}, \; \omega_3 \in \Omega_3, \\
\quad\quad \ddots \quad\quad \ddots & & \vdots \\
\quad\quad\quad E_T^{\omega_T}x_{T-1}^{\theta(\omega_T)} + A_T^{\omega_T}x_T^{\omega_T} & = & b_T^{\omega_T}, \; \omega_T \in \Omega_T,
\end{array} \\
& l_1 \leq x_1 \leq u_1, \;\; l_t^{\omega_t} \leq x_t^{\omega_t} \leq u_t^{\omega_t}, \;\; \forall\omega_t \in \Omega_t, \; t = 1, \ldots, T,
\end{array} \tag{2.5}
$$

The formulation given in (2.5) is difficult both to understand and to specify. It is possible to represent such MSLP problems figuratively as an *event tree*. As each sub-problem has a unique ancestor, or *parent*, sub-problem, a direct lineage can be traced from a sub-problem back to the first stage of the problem. Sub-problems in all but the first two stages have ancestors in common, and so the deterministic equivalent of a multi-stage stochastic linear programme can be represented by a tree.

An event tree explicitly demonstrates both the periodic and the stochastic nature of a multi-stage stochastic linear programme. The tree consists of nodes, each of which represent a sub-problem, that lie in one of the stages of the problem. Each node is a linear programme that is connected to other nodes in the stages above and below its own stage. The obvious exceptions are the nodes in the first and last stages. The node in the first stage is called the *root* node, and those in the last stage are *leaves*. The tree structure allows each node to have one parent, the unique node in the stage above of which the node is one possible child. In the tree in Figure 2.1, node 7 is the parent of node 8, while nodes 9 and 10 are the children of node 8. This event tree has four stages, with the root node, 1 in the figure, in stage 1. If the deterministic equivalent given in (2.4) were to have come from the event tree in Figure 2.1, then $\Omega_2 = \{\ 2,\ 7,\ 11\}$, $\Omega_3 = \{\ 3,\ 5,\ 8,\ 12,\ 14\}$ and $\Omega_4 = \{\ 4,\ 6,\ 9,\ 10,\ 13,\ 15\ \}$, and $\theta(8\ ) = 7$.

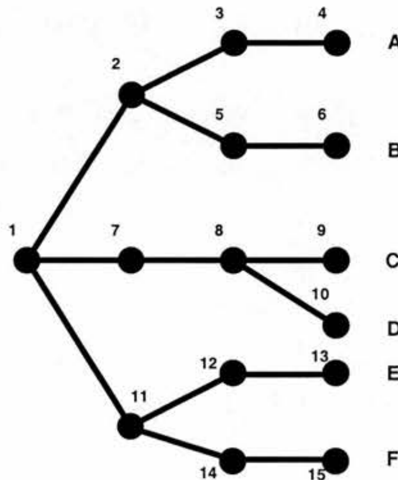


Figure 2.1: An Event Tree

The branching in the event tree represents the range of (discrete) values that the stochastic parameters in the model can take. Given a particular non-leaf node in a tree, the value of the stochastic parameters in the next stage takes one of several possible values. This range of values, conditional on being at a particular node in a stage, induces the branches of the tree from this node to its children.

Associated with each branch is a weight. The weight is the probability of the child occurring relative to its parent having occurred.

The indexing of the elements of a MSLP problem needs to be very precise. Within any stage there are linear programmes, i.e. nodes in the event tree, that are only different in one or a few of their stochastic elements. Therefore, the indexing of the model elements must be able to ensure that there is no ambiguity about which node an element belongs.

In Gasmann & Ireland (1996) [34] the classification of recourse problems is divided into two types; scenario based problems and distribution based problems. The first of these are problems where the event tree structure is specified explicitly within the model. In Figure 2.1, the path that consists of nodes 1, 2, 5 and 6 is one scenario, i.e. a path from the first stage to the last stage that included one specific realisation of the random parameters in each stage. This scenario is labelled B in the figure.

The distributions of the stochastic parameters, and how they are inter-related are given in the other classification. This implicitly induces an event tree that can either be generated by an algebraic modelling language when it creates an instance of the model, or that can be left for the solver to construct.

2.3.2 Proposed Stochastic Extensions To AMLs

In addition to providing a detailed taxonomy of stochastic linear programming, Gasmann & Ireland present extensions to algebraic modelling languages that allow the specification of MSLPs either be constructing scenarios or by the specification of the distributions of random parameters, Gasmann & Ireland (1996) [34]. An alternative set of extensions to current AMLs is presented by Fourer & Gay (1997) [30]. Their extensions are limited to scenario based models, using a more general definition of scenario to that given in the previous section.

Fourer & Gay (1997) [30] propose extending AMPL by defining a scenario as a collection of data which is associated with a model. A user can declare and maintain numerous scenarios. The different scenarios can be generated independently, and so numerous instances of the models can be dealt with concurrently, effectively defining separate problems. When new scenarios are defined, they are allowed to inherit all of their data from an all ready existing scenario, except for the first scenario, which they call the *root scenario*. The

modeller can then move between different scenarios, and, through the use of extended element names, can also reference data in one scenario, from within another. Scenarios can have differing structures as well as different data from other scenarios. These scenarios are not necessarily part of a stochastic model, and so can be generated or solved individually within AMPL.

In order to model stochastic problems, in particular recourse problems, the above definition of scenario is used together with a stochastic framework. This framework captures the periodic nature of the problem. It is specified as a set of disjoint stages, over which model elements are indexed, and is called a *time set*. When modelling a recourse problem, elements within a scenario are indexed according to the time set. Scenarios can be defined as inheriting their structure and data from an existing scenario when they are created. Despite changes made to scenarios, there is still some overlap between different scenarios, and this coupled with the time set induces the event tree.

The indexing of variables and constraints within different scenarios over the time set, in combination with the different scenarios, induces the event tree of the model. There are certain rules that must be defined, so that no ambiguities arise when defining a model. Any element which has no index from the time set is assumed to belong to the initial stage. All elements with one such index are attached to the stage associated with that index. In the case where an element has two or more such indices there needs to be a rule as to how the AML should discern the stage with which an element is associated. Otherwise it would be possible to associate the element with either of the time stages. Such a rule might be to use the first index to define the time stage.

Finally, in Fourer & Gay (1997) [30] it is proposed that the output from a recourse problem defined by scenarios should be generated in SMPS format, see Section 2.3.4, or in their own proposed format, `.nl`. In addition to specifying a multi-stage stochastic linear programmes by scenarios, they also touch upon the issue of modelling MSLPs using the distributions of the stochastic parameters. These points are dealt with in more detail in Gasmann & Ireland (1996) [34], and are discussed below.

The notion of a time set, as an ordered set, each element of which can be associated with one stage of the problem is also introduced in Gasmann & Ireland (1996) [34]. However, their concept of scenarios corresponds to the more regular idea of a path from the root node of an event tree to a leaf node. In this respect, their scenarios are only defined as part of a multi-stage stochastic linear programme,

and cannot be used as individual problems. In addition to the time set, they propose that any parameter or variable which is stochastic should be defined as having an attributed called `random`, similar to the way in which a variable can have the attribute of being integer. This informs the AML to expect distributions for stochastic data. For multi-stage problems, this random attribute is expected to include a time set index, allowing parameters and variables to be random within particular stages.

The distinction between the data and the model of a problem can become less clearly defined when modelling problems using multi-stage stochastic linear programmes. This can be seen with a common example from the formulation of a MSLP. Often a two-stage model of a problem is formulated initially. This is then solved and depending upon the results the modeller might wish to change the model to a multi-stage version. The question can be asked if this is a change in the model or in the data.

If this is thought of as a change in the data, then the definition of certain variables as being random should be present within the data file of the original model. Hence, the original model would not need to be altered, just the data used to generate the problem. However, we feel that when modelling problems that can be expressed as MSLPs, determining the stochastic structure, effectively the shape of the event tree, is an integral part of this process. As the model is developed and changed, this structure is likely to undergo several revisions. This is as much a part of the model as determining the correct form for such elements as constraints and the objective function. One of the main advantages of this is that the model is easier to read. If the data includes much of the stochastic structure of the problem, then it is very difficult to understand the model without knowing what form the data takes.

In addition to modelling recourse problems using scenarios, they also propose extensions to allow the modelling of distribution based problems. To formulate such problems, stochastic parameters need to be defined as being random, and a distribution of values associated with them. When the distributions are discrete, summation is used to include the contribution from each value in the distribution. For continuous distributions, this is extended to the use of an expectation operator, which is another of their proposed extensions.

In the taxonomy in Gasmann & Ireland (1996) [34] distribution based recourse problems are ranked into six types. These are given in order of increasing complexity. The simplest type of distribution problems is the case where all

of the random parameters are independent. For these problems the model specifies which parameters are random. The algebraic modelling language expects distributions for each of the stochastic parameters. The distributions can be discrete or continuous. These describe how a problem that contains this random parameter branches from its parent problem. When generating the problem, the AML uses these distributions to create an event tree.

There are two possible ways of dealing with continuous distributions. Either the algebraic modelling language can sample from the distribution and generate the problems for the resulting discrete approximation, or the information necessary to specify the continuous distribution can be included in the problem that is generated. All solution methods require the distribution to be discrete. Hence a discretisation of a continuous distribution must be made before a problem is solved. The first method requires some way of specifying the coarseness of the discretisation within the AML. Once this has been done, this approximation to the continuous distribution is all that a solution method receives.

The second option is preferable, as it allows the solver to decide how to sample from the distribution and so all of the information is retained until the problem is solved. However, the likelihood of a real-world problem having continuous distributions for any of its random data is extremely unlikely. Hence the ability to model continuous distributions is not a high priority. This is especially the case when the coefficients for a model are based on historical observations.

Within two stage problems, the above methods of specifying distribution based problems are straightforward. When multi-stage problems are modelled, an extra level of complexity is introduced. As when recourse problems are specified by scenario, a periodic structure has to be used. This allows a random parameter to be associated with a particular stage. Again this is done by introducing a time set, an ordered set, each element of which is identified as one stage of the multi-stage problem. All random parameters are assigned to a member of this time set. As discussed above, the specification of the time stages to which random parameters belong is part of the model, and not part of the data.

The next level of complexity is found in problems where the random data are dependent within stages, but are independent between different stages. Each set of dependent variables is specified by a multivariate distribution. The extra detail required to specify these distributions can become prohibitive, and so only low orders of distributions are practical.

Problems where the data depends upon the data in a previous stage, or where the dimensions of the sub-problems are random are the final two classes of problems that are identified in Gasmann & Ireland (1996) [34]. An example of the first type of problem is when a parameter follows a random walk, but where the size of the drift term depends upon the data in a previous stage. An example of the second type of problem is one where the number of power plants in an electricity expansion model depends upon which scenario has been followed up to a certain stage. A similar type of model is often used for financial problems. Here the event tree has many branches in the initial stage, but fewer in the latter stages. This mirrors the fact that short term forecasting methods tend to be accurate, whereas longer term forecasts are usually less reliable. The last type of problem that they specify, but which they have not yet seen in use, are problems where the distributions of parameters depend upon the decision in previous stages.

In Gasmann & Ireland (1995) [33] examples are given of an attempt to model multi-stage stochastic linear programmes using an AML that has no stochastic extensions. In these examples it was necessary for the modeller to define and maintain the event tree structure of the problem. In addition, the modeller had to remember how the indices referred to the time stage structure. These examples highlight how cumbersome models for multi-stage stochastic linear programmes are when they are formulated in an AML that has not been extended. In addition to being cumbersome and difficult to specify, these models are difficult to read and to debug.

Both Fourer & Gay (1997) [30] and Gasmann & Ireland (1996) [34], conclude are that a few extensions to existing AMLs should make it possible to model MSLP problems. In both cases possible extensions to AMPL [28] are presented. An extended language needs to identify elements as being random, and associate them with stages of an order time set. This allows the specification of the many types of chance constrained and recourse problems that are classified in Gasmann & Ireland (1996) [34]. In addition to allowing the easier specification of such problems, it should be possible for the extended AMLs to perform various consistency checks on the models, and generate the problems in an appropriate output format, e.g. SMPS format.

A similar type of consistency checking is a feature of current algebraic modelling languages that are used to specify mathematical programmes. For multi-stage stochastic linear programmes the checks could ensure that the data specified matched the data expected for coefficients and that constraints with random parameters must contain variables that are specified as being random. Further

sophistication of the AML would allow the language to classify a problem as belonging to one of the types given in the taxonomy, and so suggest a relevant type of solver. At present most AMLs are interfaced with linear programming solvers. This allows the solver to be called from within the AML. If an extended AML was able to classify models, it could then call the relevant type of solver.

The proposed extensions discussed above help to make the modelling of multi-stage stochastic linear programming problems easier, quicker and less error-prone than the ad hoc and problem-specific methods that modellers must use today. However, none of the extensions in this section have been implemented yet. Instead, there seems to be some reticence about doing so. One possible reason for this is that there are still no obvious standards for several of the problem types identified in Gasmann & Ireland (1996) [34]. Indeed, the diversity of the many types of stochastic linear programmes suggests that no AML will extend to cover all of the types in the above taxonomy. Hence, it might be that none of the AML developers are willing to take the plunge and extend their language for fear that they choose the wrong standard.

In the meantime, some researchers have concluded that this range and diversity of models means that trying to encompass all of them will never be practical within one AML. Instead, they have extended and improved hybrid methods that try to bring some functionality and rigour to the ad hoc methods that people, and including themselves, use at present. These are presented in the next section.

2.3.3 Hybrid Systems

While there are no algebraic modelling languages that include intrinsic features for modelling stochastic programmes, two research groups have developed systems that are robust versions of the ad hoc methods that many researchers use at present. These tools are aimed primarily at other researchers in the stochastic programming field. The systems that have been implemented, and which are tools for manipulating the output of AMLs, rather than being actual extensions to AMLs, are SETSTOCH [31, 17] and EMOSL [57].

The first of these tools is designed to help exploit decomposable linear programmes. In recent years there has been an increased interest in algorithms that decompose a linear programme into a block structure. Fragnière *et al.* (1997) [31] highlights that many real-world linear programmes contain large amounts of internal structure that can be exploited by such decomposition based solvers. Multi-stage stochastic linear programmes are one type of problem that contains

such an exploitable structure. However, this internal block structure is lost when problems are generated by current AMLs. The problem is generated in the algorithm's form, loosing the underlying structure, and so decomposition based solvers cannot be used to exploit the original structure.

Most AMLs use a fixed naming convention for constraint and variable names when generating linear programmes. This means that a modeller could try to re-discover the block structure of a problem after it has been generated. However, this would require a large amount of effort. Standards for the algorithm's form, such as MPS, often use fixed field lengths, and this can make the process very difficult. SETSTOCH is a tool that aims to make the process of extracting the lost block structure of a linear programme more efficient and less ad hoc.

When an algebraic modelling language generates a linear programme it creates a dictionary. This is a list of the row and column names in the generated problem and how they relate to the constraints and variables of the original problem. SETSTOCH uses this dictionary as an index of how the two forms of the model relate to each other. This allows SETSTOCH to identify any structure that might be hidden within the algorithm's form of the model. Once the structure is identified, the problem can be decomposed and solved using a suitable decomposition based solver.

There are two main disadvantages that limit the use of SETSTOCH to researchers within the large-scale linear programming field. The first of these is trying to identify into which block structure to re-arrange the linear programme. The main reason for re-arranging a linear programme into one structure in preference to another is that most decomposition based algorithms are suited to one particular type of structure. Therefore, the modeller needs to have an insight into the form of the original problem. If the modeller knows that the original problem was a multi-stage stochastic linear programme, then the model contains a primal block angular structure, i.e. the block angular structure for a problem that can be represented by an event tree. The second limitation of SETSTOCH is that it needs to be modified to interface it with different AMLs. As the dictionary produced by each AML is in a different format, SETSTOCH needs to be modified to interface it with different AMLs.

The second of the hybrid systems that are currently available, EMOSL, discussed in Tebboth & Daniels (1998) [57], is an adjunct to an optimisation subroutine library. This system aims to help researchers to create dedicated solvers that are able to exploit the structure of problems. Whereas the SETSTOCH package

discussed above attempts to regain the structure from a linear programme once it has been generated, EMOSL provides a framework in which both the modeller's and the algorithm's form of a problem are held simultaneously. EMOSL is primarily intended for developing decomposition based algorithms.

Currently there are two frameworks in which linear programming algorithms are developed. The first framework is the use of optimisation subroutine libraries. These help a programmer implement a particular algorithm by providing routines that allow the easy manipulation of linear programmes. This is done within a general purpose programming language, such as C or Fortran, and the code is compiled to produce an executable. The resulting optimiser is normally very fast. If well implemented, this provides an algorithm that is faster on problems with a particular exploitable structure than a commercially available solver that is suited to general problems.

The second framework in which algorithms and models are constructed is algebraic modelling languages that have been interfaced with solvers. So it is possible to invoke the solver from within the AML. Thus, after a model has been specified, the AML is told to solve it. First the algorithm's form of the model is generated, and this is then passed to the solver. Often an AML is also able to extract information about the solution of the problem and present it in a manner that reflects the original specification of the model.

The advantages and disadvantages of these two frameworks reciprocate each other. The first system produces very fast and efficient code, but the solver has no way of identifying any structure that can be exploited. This seriously limits the usefulness of such optimisers, as problems that they could readily solve need to undergo a cumbersome manipulation before they can be solved. The second system allows the modeller to refer to particular structures within the model, but the need to re-generate the model each time it is to be passed to the solver, and the communication overhead involved in switching between the AML and the solver makes the algorithms produced slow and impractical for large real-world problems. EMOSL hopes to be a combination of these two distinct methods for model and algorithm development, and as such gain advantages from both of them.

In Hürlimann (1998) [42] another way of further integrating solution methods within an algebraic modelling language is discussed. The motivation is similar to that of EMOSL; to create a system in which it is possible to both declare models and to programme algorithms. This modelling language, LPL presented

in Hürlimann (1998) [43], is an initial attempt to combine these two parts of the modelling process. Rather than allow the modeller to call a solver from within a modelling language, LPL allows the modeller to call individual optimisation subroutines. Hence the model and the algorithm are both specified within the one framework. Essentially LPL is trying to obtain the same functionality as EMOSL, described below, but is an extended modelling language, whereas EMOSL is an extended optimisation subroutine library.

EMOSL contains optimisations subroutine libraries that allow a developer to implement optimisation algorithms. In addition, EMOSL can also read in a model as specified using an algebraic modelling language. Within EMOSL, an instance of the problem is generated from the model. This is what the algorithm goes on to solve. Both the modeller's and the algorithm's form of the model are kept active within the solver. As EMOSL knows how the two representations of the model refer to each other, the algorithm has immediate access to parts of the problem by referring to the original element names. E.g. a collection of constraints called `demand(time)` can be accessed by referring to `demand` and a range of values for the index `time`, say `May` and `June`.

The algorithms generated in EMOSL have comparable speeds to those using just ordinary optimisation subroutine libraries, but they allow the developer greater flexibility in access partings of the model. The only possible drawback of this system is that the two versions of the problem need to be kept in memory at the same time. This increases the memory requirements of an algorithm, but it is likely to be less of an issue as computer memory is not particularly expensive.

The main drawbacks of EMOSL are the limits of its functionality. The first of these is that there is no connection between elements of the model that are indexed over the same set. Consider a multi-stage problem where all elements are indexed over the set of time stages. Within EMOSL it is not possible to access all elements that occur in one particular stage. Instead each element in turn has to be accessed with the index set to that stage. The second limit is when elements are being accessed. In the example of the constraint `demand(time)` given above, the range of the index `time` has to be contiguous. So it is not possible to access constraints for alternate months with one subroutine call. Instead each constraint needs to be accessed individually.

The third limitation of EMOSL is more significant. At present the type of changes that can be made to a model once it is inside an algorithm are limited. Elements can effectively be removed, by setting their coefficients to zero. If an element is

added to the problem after the algorithm starts, then this change only occurs in the algorithm's form of the problem. An algorithm may add cuts to a problem, as in Benders Decomposition discussed in Chapter 3. However, there is no way of accessing this additional constraint in the same manner as the original constraints can be accessed; the modeller's form of the problem has no information about this extra cut. This means that elements can be added and used to alter the solution of the problem, but they cannot be accessed as parts of the model. If such additions were accessible in the same manner as the original elements of the model, then the entire problem could be built from scratch within EMOSL.

Despite their limitations, both of these systems are of benefit to any researcher in the field of stochastic linear programming. Neither system was designed to be an alternative to an algebraic modelling language that has generic capacity for specifying multi-stage stochastic linear programmes. Both of the systems are of use to modellers and algorithm developers as robust tools for speeding up the process of generating MSLPs while such algebraic modelling languages do not exist.

2.3.4 SMPS: As It Is And How It Might Develop

The standard algorithm's form for specifying linear programming problems is the MPS format [18]. This standard is recognised by all linear programme solvers, and so it was used as the basis for a standard algorithm's form for multi-stage stochastic linear programmes. This is called SMPS format, see Birge *et al.* (1987) [7], and it has become the current standard for specifying multi-stage stochastic linear programming problems. In addition to gaining the universality of the MPS format, the SMPS format has also inherited the limitations of the former.

The SMPS format uses three files to specify a multi-stage stochastic linear programme; the *core file*, the *time file* and the *stoch file*. The core file is an MPS file which contains one scenario from the event tree, i.e. it presents the nodes of the tree down one path from the root node to a leaf node of the tree as a single LP problem. The time file specifies how the problem in the core file is divided into time stages. This is done by specifying the names of the first row and column for each time period. Therefore the ordering of the rows and columns has to be consistent with the time structure of the stochastic problem, i.e. all the rows and columns of stage t should come before those stage $t + 1$ in the core file. For this reason the column and row names given in the core file must be in stage-order.

The stoch file specifies the realisations of the random parameters in the model. This can be done in one of three ways. The most simplistic of these is to specify independent random variables by enumerating their distributions, if they are discrete, or by specifying the parameters for standard continuous distributions, such as normal or uniform. It is then left to the solver to reconstruct the full MSLP problem from these distributions. Any random parameter must have a non-zero entry in the core file, even if this value does not match any of its possible realisations. This value acts as a place holder, and is over-written by the values of the stochastic parameters, as specified in the stoch file.

The second format for specifying the stoch file has groups of random elements specified as vectors, rather than as independent values. This format can be used for discretely distributed dependent random variables as well as a limited range of random variables with standard continuous distributions. The final way of presenting the stochastic information is to define the tree by scenarios. Here an initial scenario is defined as a path through the tree, whose structure can be inferred from the stoch and time files. Subsequent scenarios are specified by their differences from previously specified scenarios. A branching stage is given, and the stages prior to this are the same as the scenario upon which it is based. After this stage the differences between the two scenarios are given, and the probability of the new scenario is given.

The SMPS format has become the *de facto* standard for MSLP problems. However, most implementations of MSLP problem solvers have been designed to accept problems in their own input format, as well as being able to solve problems in SMPS format. This results from the SMPS format having several limitations and omissions. A review of the limitations of SMPS is presented in Gasmann & Schweitzer (1996) [35], together with suggestions of some possible extensions to the format.

They see the main limitations in the SMPS format as being the omission of chance constraints and problems with variable sub-problem dimensions, the way in which random entries replace the entries in the core file, the deficient provision for univariate, multivariate and general distributions, the inability to specify a problem node-by-node from its event tree and the fixed field format inherited from the MPS format. As well as suggesting some solutions for the limitations of the SMPS format, they also provide extensions which overcome its shortcomings in modelling network problems.

A large number of the limitations with the original specification of the SMPS format are inherited from the fixed field format used in the MPS format. This fixed format divides each line of the core, time and stoch files into six columns, each of fixed width. Two of these columns contain numerical data, with their fixed widths limiting the accuracy of the numerical data. The remaining columns contain element names, or identify parts of the SMPS format. Hence the names of rows and columns are of fixed length, and so tend to be coded versions of their equivalents in the modeller's form of the problem.

One way to overcome this limitation is to use white space to delimit variable width fields. This would allow numerical data to be given to the available accuracy and element names to be as long as necessary. The limitations inherited from the MPS format are present in the time and stoch files as well as the core file. While the core file is essentially a linear programme in the MPS format, there was never any need for the time and stoch files to adopt the fixed field format of the MPS format. Allowing there to be more than a variable number of fields per line and using white space to delimit the variable length fields would greatly reduce the limitations of the SMPS format.

In Gasmann & Schweitzer (1996) [35] several ways in which to increase the flexibility in the format of the stoch file are discussed. These include the ability to specify more complicated continuous distributions, univariate and multivariate distributions and also that the values of the random parameters need not overwrite their corresponding values in the core file. At present standard continuous distributions, such as uniform and normal distributions, are specified using two numerical fields. However, it would be possible to define more complicated continuous distributions by extending the number of fields per line. Random variables whose values are supplied by a user-specified subroutine could be extended similarly. In addition to specifying the name of the subroutine, a series of parameters are supplied that are passed to the solver.

When specifying multivariate distributions, the present SMPS format runs into difficulties. This is because two element names are required to identify a parameter of a continuous distribution, thus restricting the types of distributions that it is possible to specify using the fixed field format. This could either be extended by using more fields, or by a method of aliasing previously defined parameters. The second method would also avoid the burden of notation inherent in the present system. It is also suggested in Gasmann & Schweitzer (1996) [35] that instead of the random values specified in the stoch file replacing their counterparts in the core file, they be available to be used for other types of

transformations of the original values. Being able to add or multiply additional terms to the original values in the core matrix would make the specification of the stochastic parameters more flexible.

In addition to these extensions, two new ways to specify MSLPs are also considered in Gasmann & Schweitzer (1996) [35]. The first is the introduction of chance constraints, which were not part of the original SMPS format. Individual chance constraints are specified as ordinary constraints in the core file. In the stoch file their reliability levels and the direction of the inequality is given. Joint chance constraints can be specified by combining individual chance constraints with the method for specifying random vectors. Their other suggestion is the ability to specify a problem by the nodes in its event tree.

This is achieved in a similar manner to the specification of problems by scenarios. Each node is either specified in full in the stoch file, or it inherits the details of a previously defined node, and the differences between the two nodes are given. Each node is marked as being an immediate descendant of another node in the tree, and is given a probability, conditional on its parent, of occurring. The obvious exception to this is the root node which has probability one, and can either be specified in full in the stoch file, or can be read from the core file. When defining a node in relation to another node, the previous node need not be related to the new node in any way, i.e. it does not have to be the parent node, or a sibling.

This method for specifying multi-stage stochastic linear programmes is similar to the method used by our algebraic modelling language, sMAGIC, which is presented in the following section. The main difference is that the output of sMAGIC currently does not take advantage of the similarity of nodes in the tree. However, to allow sMAGIC to produce problems using the extended version of SMPS proposed in Gasmann & Schweitzer (1996) [35] it would be necessary to extend the above method for specifying node of an event tree. As sMAGIC is able to generate folded versions or MSLPs whose stochastic parameters exhibit inter-stage independence, the concept of an event tree needs to be extended to include graphs. This means that nodes can have more than one parent, with a different transition probability from each of these parents.

This requirement falls within the final change to the SMPS format that is proposed in Gasmann & Schweitzer (1996) [35]; that the format should be self-extending. The types of problems that will be studied in the future are unlikely to be adequately covered by the current or the extended SMPS format. Whenever

new test problems are created that cannot be expressed in the current version of the standard, they propose that the creator of the problems be allowed to extend the standard. In this way, the researchers working in a particular field of stochastic programming will be able to extend the standard using their particular knowledge. Of course, this could lead to confusion if a large number of different extensions are introduced. To avoid this they recommend that any problems using a new extension come with a full specification of the new extensions.

2.4 sMAGIC: A Recursive AML

Since there is no algebraic modelling language that had the ability to model multi-stage stochastic linear programmes, we have developed sMAGIC. This AML has the facilities to model MSLPs using modular sub-models and recursive definitions. As well as having specific constructs to allow it to model MSLPs, sMAGIC also has similar functionality to that of currently available AMLs.

In addition, the use of recursive definitions to model MSLPs makes sMAGIC suitable for the modelling of MSLPs that can be "folded". These are problems where the stochastic parameters are independent between stages, and so are Markovian. In this section we first discuss the motivations behind sMAGIC, and the criteria that were used to guide its development. Following this we discuss details of sMAGIC with reference to the specification of linear and multi-stage stochastic linear programmes. Some of the material in this section appears in an earlier draft in Buchanan *et al.* (1998) [15].

2.4.1 Motivation & Development

The main motivation behind developing sMAGIC was to create an algebraic modelling language in which it was possible to specify multi-stage stochastic linear programmes. In addition to this, we wanted to be able to model linear and mixed integer programmes using sMAGIC, as these are currently the main types of mathematical programmes that current AMLs can model. Our third aim was that sMAGIC would be modular, and so allow problems to be specified as smaller sub-models that could be combined to form larger models.

Since its earliest version, sMAGIC has been used by researchers working with various different types of optimisation problem. This has lead to the language being developed as a fully functioning AML that can be used to model linear programmes and mixed integer linear programmes as well as MSLPs. sMAGIC has been implemented as a software package by G Skondras, while I am mainly responsible for developing the functional specification of the language. A detailed explanation of how sMAGIC was implemented can be found in the Skondras (1998) [56]. In the remainder of this section we discuss the functional specification of the language, and specifically those aspects that allow it to be used to formulate multi-stage stochastic linear programmes. The development of the ability of sMAGIC to model MSLPs was primarily motivated by our model of the hydro-electricity generation network of Southern Brazil.

Our hydro-electricity model combines a network of thermal generation plants and transmission lines with a network of hydro-electricity networks. This model was developed through various stages. In the first version of the model we considered only the problem in one time period. In this version of the model the two networks must produce enough electricity to meet demand, while conserving water at each reservoir. The thermal and hydro plants are connected to transmission nodes which have a local demand for electricity. If enough electricity is not available locally, then the demand can be met by passing electricity between nodes in the transmission network. Any water coming into a reservoir, either from nature or from upstream reservoirs, must balance the amount leaving the reservoir, either through hydro-generation or due to spillage, less the difference in the initial and the terminal amounts of water in the reservoirs. The model also has a further structure due to the electricity market being modelled by a load duration curve. This divides the production of electricity into time blocks within each stage, during which the level of electricity demanded is constant.

We extended this version of the model into a multi-stage model with a finite time-horizon. Instead of extending the model into a multi-stage linear programme directly, we considered using our initial model as part of a Dynamic Programming recurrence. Dynamic programmes are common models in the hydro-electricity industry, Terry *et al.* (1986) [58]. Here the time period is a state variable, and the amounts of water in the reservoirs are the decision variables. The majority of models for this type of problem in the stochastic programming community are extended linear programmes. However, these models loose most of the structure that is inherent in these problems, especially when they are extended to include stochastic parameters.

So the second version of our model consisted of a series of DP recurrences. Each one of these was a linear programme, with the variables of the reservoir levels being the complicating variables. The final extension to our model was to model the rainfall as being stochastic. So instead of modelling the rainfall as having one value in a stage, we generated a model where it could take one of several values per stage. Using historical data we generated a range of possible rainfalls for each stage. The rainfall is a second state variable; any recurrence can be specified uniquely using the rainfall and the time stage. From the historical data we also calculated the probabilities of moving from one point in the state space to another.

Having developed our model in such a way, we then used this to motivate how sMAGIC allows MSLPs to be specified. The recursive definition of models and

sub-models is the main feature that allows this. Models that are defined within sMAGIC can be called from other models. They can either be declared as having no parameters, in which case when called from within another model they are simply slotted into the larger mathematical programme. In this way sMAGIC allows large models to be constructed from smaller sub-models. Additionally, models can be declared as having one or more parameters. These allow models to be called with specific information that tells them how to behave as well as providing a method of linking the variables of the called model to those of the model in which the call is made. In addition, models are able to call themselves, and it is this feature that allows our MSLP to be modelled using sMAGIC.

Recursive modelling languages exist within other research fields. The chemical engineering world makes particular use of modular modelling languages. These are used to model large systems as separate parts. For examples, a boiler might be modelled as having parameters concerning its fixed attributes, such as dimensions and availability, as well as other operating parameters, such as the temperature of its contents, or the rate of inflow. This can then be called from within a model of a heat exchanger, where the temperature is a variable in the larger model.

Modular modelling languages have several benefits. Primarily, they allow models to be built up in logical units. This makes the verification and de-bugging of sub-models quicker and more structured. Sub-models, such as components of a system of pipes, heat exchangers or filtration towers, can be re-used within other models, helping to speed up the process of future modelling, and capitalising on the benefits of previous modelling efforts. This way of building a model from smaller parts extends naturally to the stochastic programming field. This is one of the aims of sMAGIC.

The form of our multi-stage stochastic linear programme for the Brazilian electricity generation network given above is not covered by the proposed extensions to AMLs that were discussed in the previous section. These problems have stochastic parameters that are independent between stages. This structure provides an opportunity to share information across stages as the problems can be "folded" to produce graphs, rather than the usual event tree structure. Modelling these problems using the extensions to current AMLs proposed above results in the loss of this structure when the problems are generated. In the past decade there has been increased interest in combining MSLPs with sampling techniques to allow the solution of very large-scale problems. These methods are particularly suited to MSLPs with the same form as our hydro example. If sampling based algorithms are used on very large problems that cannot be folded, then the

likelihood that a particular sub-problem in the tree is sampled is very small. This means that very few of the sub-problems in the tree contribute to the estimated bounds on the problem, and so a large sample size is required to provide reasonable estimates of the bounds. When solving folded problems, more than one path through the problem passes through each sub-problem, and so there is a much higher chance that a sub-problem is sampled, and so contribute to the estimated solution. For this reason, it is preferable to generate such problems in a folded form, so that their structure can be exploited. sMAGIC provides a framework in which this form of MSLP can be generated in an intuitive manner.

sMAGIC generates folded problems as a collection of linear programmes in MPS format and a map of the graph that links them. The current standard for the algorithm's form of multi-stage stochastic linear programmes, SMPS format, does not cover folded problems. It is necessary to exploit this type of structure when solving very large multi-stage stochastic linear programmes. The extensions to SMPS given in Gasmann & Schweitzer (1996) [35] do not cover this type of problem. However, the extension that allows MSLPs to be specified node by node could be extended to include problems specified as graphs. Each node could be allowed to have more than one parent. At present the nodes in the extension are only allowed to have one parent, and so an event tree is formed. By allowing a node to have more than one parent, the graph structure of folded problems could be specified.

2.4.2 Details Of Language

Initially we look at the way in which sMAGIC defines sub-models, and how these can be combined to build larger problems. After this we demonstrate how these sub-models can also be used to specify MSLPs. Finally we look at the files that sMAGIC generates when it is generating a problem.

The main feature of sMAGIC that allows the modelling of MSLP problems is its use of recursive sub-models. These sub-models can be used out with stochastic framework to build large linear programmes from smaller parts. Formulating models in this way has several benefits. The first is that the smaller LP models are easier to formulate and verify due to their manageable size. Another benefit is that sMAGIC provides an easy way of using sub-models within other models, as demonstrated by the following example.

In Williams (1985) [63], Williams considers a company that has several factories producing the same products. The different factories have different production

capacities and production lines of varying types. Hence the amount of resources need in each factory to produce the same product is different. A planner has to schedule the production of products at the various factories so as to minimise the total overall costs.

This problem can be broken into the smaller problems of deciding what to produce at each factory. In sMAGIC this can be done by formulating a model for each factory, and using these to build the overall problem. The overall problem is what the planner uses to make decisions. However, the sub-models could be produced by experts at each of the factories, and only combined by the planner. Hence the use of models can be used to facilitate a more modular form of modelling. An sMAGIC formulation of this problem is given in Figure 2.2.

```
MODEL WHOLE_PROBLEM
!MODEL ROOT
use module sizes_and_sets;
DATA
raw_cap<<''raw'';
VARS
raw[n_factories];
OBJECTIVE
minimize dummy = FACT_A(raw[1]) + FACT_B(raw[2]) + .......;
CONSTRAINTS
materials: sum{r IN R} raw[r] <=raw_cap,
FOR {r IN R}
{
bnds{r}: 0 <=raw[r]
};
END MODEL

MODEL FACT_A(raw)
use module sizes_and_sets;
parameter vars raw;
DATA
profit[n_products] <<''profit'',
  grind_time[n_products] <<''grind',
........;
VARS
products_A[n_products];
OBJECTIVE
minimize dummy =sum{p IN P} profit[p]*products_A[p];
CONSTRAINTS
raw_materials: sum{p IN P} products_A[p] <= raw,
grinding: grind_time[p]*products_A[p] <= grind_cap,
polishing: polish_time[p]*products_A[p] <= polish_cap,
FOR {p IN P}
{
bnds{p}: 0 <= products_A[p] <= max_p[p]
};
END MODEL
```

```
MODEL FACT_B(raw)
use module sizes_and_sets;
parameter vars(?) raw;
DATA
profit[n_products]<<''profit'',
time[n_constraints,n_products]<<''time'',
cap[n_constraints]<<''cap'';
VARS
products_B[n_products];
OBJECTIVE
minimize dummy =sum{p IN P} profit[p]*products_B[p];
CONSTRAINTS
FOR {c IN C}
{
constr{c}: time[c,p]*products_b[p] <= cap[c]
}
FOR {p IN P}
{
bnds{p}: 0 <= products_B[p] <= max_p[p]
};
END MODEL
```

Figure 2.2: Multi-factory Example

Models written in sMAGIC can contain several types of object: data, variables, sets and models. The data can be of type real, integer or strings. Data, variables and sets can consist of single elements, or can be made up of arrays of elements. In the example given in Figure 2.2, there are three models, `ROOT`, `FACT_A` and `FACT_B`. The first of these is the main model and it calls both of the sub-models that model the two factories. In addition to the local elements of each model, they also all us a module `sizes_and_sets` where the fixed parameters of the problem have been declared. The use of modules allows the definitions of common dimensions of the problem to be set once and used by any of the models.

In the model `FACT_B`, `products_B` is an array of real valued variables, indexed over `n_products`, which is defined in the module `sizes_and_sets`. These variables are local to the model `FACT_B`, and can take different values in each distinct instance of this model, when it is called as a sub-model of another model. The sub-models can contain parameters in their declaration, as do models `FACT_A` and `FACT_B` in the above example. These parameters can be of type data, set or variable, as either scalars or arrays. In the case above, the parameter `raw` is a scalar variable.

Each call to a sub-model is uniquely defined by the value of the data parameters used to call it. Hence, when sMAGIC generates a problem any call to a model with the same values for its data parameters results in identical models being generated. Therefore sMAGIC does not generate this model again, instead it makes a link from the current model to the existing instance of model that has been called. In this way a graph is generated that has model instances as nodes and model calls as links. We refer to this as the *model link graph* (MLG). For a more detailed explanation of the specification of objects so as to ensure sMAGIC models are pure and without side effects, the reader is referred to Skondras (1998) [56].

In real-world problems the individual LP problems are larger and more complex than the example shown in Figure 2.2. The modularity provided by sMAGIC makes it easy for such models to be formulated by different people at different times and locations. Such individual models might be used in many different large-scale models, and so the modelling effort for subsequent models can be reduced by the use of libraries of sub-models.

As well as facilitating the formulation of modular LP models, the use of sub-models in sMAGIC can be used to model multi-stage stochastic linear programmes. As an example of the formulation of a multi-stage stochastic linear programme using sMAGIC, consider an production and inventory problem, see Gasmann & Ireland (1996) [34]. Sufficient amounts of several products have to be produced in order to meet demand over a finite number of time periods. Any surplus products can be held over and used in the following period. Any stock remaining at the end of the final time period has no value, and the amount held over from period to period is limited, as is the amount of production in each time period.

Such a problem can be formulated as

$$\begin{array}{rcr} \min\limits_{m_t, h_t : t=1,\ldots,T} & \sum\limits_{t=1}^{T} \beta^{t-1}\left(c^m m_t + c^h h_{t-1}\right) & \\ s.t. & h_t = h_{t-1} + m_t - d_t, & (2.6) \\ & 0 \le m_t \le \overline{m},\ 0 \le h_t \le \overline{h},\ t = 1, \ldots, T, & \end{array}$$

where m_t and h_t are the amounts of products produced and stored in time period t, with the associated costs c^m and c^h. The upper bounds for production and storage are $\overline{m}$ and $\overline{h}$, and the demand for products in time period t is d_t. The future costs are discounted using a factor of β. The objective is to minimise the total discounted cost through the T time periods, given that the initial inventory of products is h_0.

The problem given in (2.6) can be specified in sMAGIC as one linear programme, using summation and indexed constraints. However, an alternative way of formulating this form of multi-stage linear programmes is as a set of dynamic programming recurrences. An example for the production and inventory problem given above is the recurrence

$$\begin{array}{rlr} C(t, h) = & \begin{cases} \min\limits_{m,\tilde{h}} c^m m + c^h h + \beta C(t, \tilde{h}), & t < T \\ \min\limits_{m,\tilde{h}} c^m m + c^h h, & t = T \end{cases} & \\ \text{s.t.} & \tilde{h} = h + m - d_t, & (2.7) \\ & 0 \le m_t \le \overline{m},\ 0 \le h_t \le \overline{h}, & \end{array}$$

where the notation is the same as in the previous formulation of the inventory problem. The solution to this problem is to find $C(1, h_0)$, and the corresponding optimal actions in this model and all of its sub-models.

As discussed previously in this chapter, in real-world problems some of the coefficients in a model are uncertain. In the problem above, the demand in future time periods might be uncertain at the start of the planning horizon. However, the modeller usually has an idea of a range of possible future demands, and the probability of each occurring. We assume that the decision maker has to decide how much of each product to make in a time period before the demand for that period is known. If we define the set of possible demand scenarios as Ω_t for period t, and elements of this set as $\tilde{\omega} \in \Omega_t$, then we can represent the probability of

there being demand $d_{\tilde{\omega}}$ in period t, given that there was demand d_{ω} in period $t-1$, as $\pi_{t,\omega,\tilde{\omega}}$.

Extending the recurrence given in (2.7) to include this stochastic parameter results in the following recurrence

$$C(t,\omega,h) = \begin{cases} \min\limits_{m,\bar{h},\tilde{\omega}\in\Omega_t} c^m m + c^h h + \beta \sum\limits_{\tilde{\omega}\in\Omega_t} \pi_{t,\omega,\tilde{\omega}} C(t,\tilde{\omega},h_{\tilde{\omega}}), & t < T \\ \min\limits_{m,\bar{h},\tilde{\omega}\in\Omega_t} c^m m + c^h h, & t = T \end{cases}$$

$$\text{s.t.} \quad h_{\tilde{\omega}} = h + m - d_{\tilde{\omega}},\ 0 \le h_{\tilde{\omega}} \le \overline{h},\ \forall \tilde{\omega} \in \Omega_t,\ \pi_{t,\omega,\tilde{\omega}} > 0 \qquad (2.8)$$
$$0 \le m_t \le \overline{m},\ 0 \le h_t \le \overline{h}.$$

The solution to this problem is to find $C(1,\omega_0,h_0)$, where h_0 is the initial stock and ω_0 is a prior demand level, and all of the corresponding optimal actions in this model and all of its sub-models. Here $C(t,\omega,h)$ is the minimum cost of producing products, subject to having h stock held over from the previous stage, and the previous demand level being ω.

```
model C(t, w, h)
  use module COMMON_COST;
  parameter data t, w;
  parameter variable h[no_products];

  data beta = 0.9;
  variable
    m[n_products], new_h[max_no_demand,n_products];
  objective
    if (t = T)
      minimise C = sum(p IN P) (c_m[p]*m[p] + c_h[p]*h[p])
    else
      minimise C = sum(p IN P) (c_m[p]*m[p] + c_h[p]*h[p])
                 + beta*sum(nw_w IN W[t])
                         pi[t,nw_w,w] * C(t+1,nw_w,nw_h[nw_w,*]);
  constraint
    for (nw_w IN W[t], pi[t, nw_w,w] > 0, p IN P)
    {
      nw_h[nw_w,p] = h[p] + m[p] - d[nw_w,p];
      nw_h[nw_w,p] <= h_mx[p]
    },
    for (p in P)
      0 <= m[p] <= m_mx[p];
end model
```

Figure 2.3: SMAGIC Production and Inventory Problem

The listing given in Figure 2.3 is an sMAGIC formulation of recurrence (2.8). As in the example shown in Figure 2.2, this model uses a module to group the common declarations of the problem together. The production and inventory problem model is called with three parameters. The first two parameters, `t` and `w`, are fixed data that specify the state of the problem. The third parameter,

h, is a variable within the problem. If the model C is called more than once with the same values for the fixed value parameters, then the same model is required. In the production and inventory problem discussed above, this might be the case if there are several ways that the problem can arrive at the same state in a later stage. I.e. it might be possible to arrive at the problem $C(3, \omega_2, h)$ by following various paths from the initial state. As it is called with the same data parameters, this problem is the same regardless of the historical path taken to get to it, so to generate it more than once within the generation of a problem leads to repetition. The problem that results from exploiting such repetition within a multi-stage stochastic programme is represented by a graph, rather than the tree structure discussed earlier in Section 2.3.1.

Imagine a version of the problem, given by recurrence (2.8), that has three stages, with three realisations of the random demand in the second stage, and two in the final stage. Such a problem is represented by sMAGIC as the model link graph given in Figure 2.4.

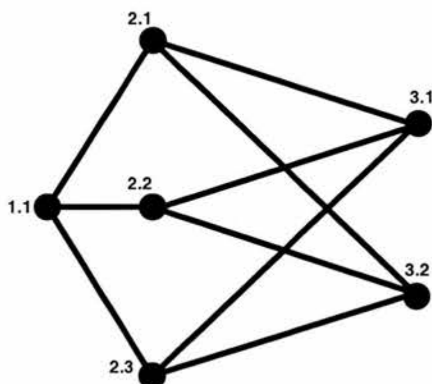


Figure 2.4: A Model Link Graph

Similarly to the event tree of a multi-stage stochastic programme, the nodes in Figure 2.4 relate to individual LP problems. Each of the arcs in the graph relates to a realisation of the stochastic elements of a model call, and has a probability attached to it. In the graph the nodes show all of the distinct states of the problem. However, the nodes in the final stage have more than one parent. There is more than one history of stochastic variable realisations that result in the problem being in any of the states in the final stage. There are two problems in the final stage of this problem, representing the two values for ω in the third stage. However, each of these problems can be called from any of the three previous stage problems, i.e. they can be called with three distinct vectors h.

An alternative way to view this problem is as an event tree. Such a representation is given in Figure 2.5. Here the paths through the tree are represented distinctly,

the graph has been *unfolded* to make explicit all of the possible realisations of the random elements of the problem. In the unfolded problem there is repetition; nodes 3.1a, 3.1b and 3.1c represent the same sub-problem as node 3.1 in the model link graph shown in Figure 2.4. This is the result of the Markovity of the stochastic parameters; they are independent between stages.

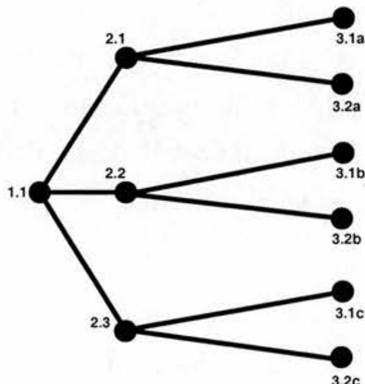


Figure 2.5: Unfolded Tree Representation

Ignoring the structure when MSLPs have it results in a larger number of sub-problems being generated to represent the same problem. For large real-world problems, with more stages and more sub-problems per stage, the difference between the two representations becomes far more extreme. The unfolded tree representation grows exponentially in the number of stages and the number of sub-problems per stage, whereas the folded problem grows linearly. Some of the problems considered in Chapters 4 and 5 consist of 12 stages with three sub-problems per stage. This results in there being $3^{12} \approx 500,000$ nodes in the final stage of the tree, and over 700,000 nodes in the whole tree. Folding this problem to give the equivalent model link graph results in a problem with 3 nodes in the final stage and 37 nodes in total. For this reason such large real-world problems are only tractable if their Markov structure is exploited. sMAGIC provides a natural way to do this.

sMAGIC processes a model in two passes of the language processor that is at the core of sMAGIC. During the first pass the modules and models of the problem are parsed. Storage space and an internal representation of each module and model is created. Syntactical errors are detected in this first pass through the model. During the second pass a depth first search of the graph is performed, starting at the root node. As each model call is encountered, sMAGIC checks whether this instance of the model has been called before. If so, then a link to the existing instance of sub-model from the current model is created. If the model has not been generated before, then a new linear programme is created, and a link is

made to this from the model from which it is called. In this way the problem is generated as either a graph, if there are multiple calls to the same model, or as a tree if all models are only called by at most one other model.

There are two items attached to the links between models, represented by arcs in the graph or tree of a problem. The first of these is the probability of the sub-model occurring, conditional of having been at the parent model. This is the coefficient of the sub-model in the objective function where the sub-model is called. Models can be called with parameters that are variables. But the variables need not have the same names in the different models. Therefore, a mapping between variables names in one model and another is attached to each link.

The output from sMAGIC consists of a collection of files. The first is an overview of the graph representation of the problem. It consists of the number of nodes in the graph, a list of file names that contain the LP problems associated with each node, and finally a list of the arcs the link these nodes. For each arc in the graph, there is a corresponding probability of it occurring, and the file name of the mapping between variables that are passed along the arc.

In Chapter 3 we discuss how the solution of multi-stage stochastic programmes generated as graphs can lead to previously intractable problems being solved due the exploitation of inter-stage independencies. In Chapter 4 we look at how sampling techniques can be combined within a Benders Decomposition based algorithm to allow very large-scale multi-stage stochastic linear programmes generated from real-world problems to be solved.

2.5 Conclusions

Algebraic modelling languages provide a powerful tool for the generation of mathematical programmes. As yet no algebraic modelling language has specific features that allow the formulation of multi-stage stochastic linear programmes. This has resulted in the growing interest in multi-stage stochastic linear programmes being confined mainly to the research community. Extensions to existing algebraic modelling languages have been proposed, but none have been implemented yet.

sMAGIC is a powerful algebraic modelling language that has been designed to provide a natural way to model multi-stage stochastic linear programmes in an intuitive manner. It does this through the use of recursive definitions and sub-models. In addition to facilitating the formulation of multi-stage stochastic linear programmes, these features increase the modularity of the language and its ability to formulate linear and mixed integer programmes.

The stochastic parameters of many multi-stage stochastic linear programmes exhibit inter-stage independence. These can be formulated compactly as a set of recurrences. The use of recursive sub-models in sMAGIC is a natural way to model problems with this structure. sMAGIC generates problems with this structure as model link graphs, rather than as their event trees. Problems can theoretically be converted between graph and tree format. As problems with this Markovian structure grow in size the tree representation becomes too large to generate or solve, while the model link graph remains much smaller. Therefore, much larger problems can be generated and solved if they exhibit inter-stage independence between their random parameters.

The other benefit of representing multi-stage stochastic linear programmes with an underlying Markovian structure as model link graphs is that their structure can be exploited by Benders Decomposition based algorithms that incorporate sampling techniques. If these problems are generated as trees then their structure can not be exploited so readily.

Chapter 3

Benders Decomposition Method

3.1 Introduction

Most planning problems require decisions to be made at various intervals over a particular time horizon using uncertain information. Multi-stage stochastic linear programmes (MSLPs) are natural models for such problems. The MSLPs of real-world problems are very large, but have a particular structure. Hence to solve them efficiently it is necessary to exploit this structure.

Benders Decomposition is an iterative method for solving multi-stage stochastic linear programmes. It decomposes a problem into its natural sub-problems. These are solved individually using a linear programme solver, e.g. the simplex method. As they are relatively small this is not computationally expensive. The benefits of Benders Decomposition are that it allows otherwise intractable problems to be solved in a reasonable time.

After an outline of the history and motivation of Benders Decomposition based algorithms, we present the simplex algorithm which is direct solution approach that can be used to solve MSLPs. In the following section we derive Benders Decomposition for increasingly more complicated types of MSLPs. Once a final algorithm has been derived, we look at extensions to this basic algorithm and provide results that show how they affect the algorithm's performance. The following section describes possible ways in which Benders Decomposition can be implemented for parallel computers. We present our parallel Benders Decomposition algorithm, and compare this with serial and parallel implementations of two Dynamic Programming algorithms. Finally we discuss a variant of Benders Decomposition for problems that exhibit inter-stage independence between their stochastic parameters.

3.1.1 History

When linear programming problems were first formulated in the 1940s and 50s, the need to consider the uncertainty in the data was noted, Dantzig (1955) [21] and Beale (1955) [5]. Multi-stage stochastic linear programmes provide a framework in which to formulate real-world problem planning problems where decisions are made at specific points in time using uncertain data. Such problems can be very large and are often too large to solve by direct methods such as the simplex algorithm. This is due to either the limitations of the computer's memory, or because these problems take too long to solve on even relatively fast computers. However, these problems have a definite structure which can be taken advantage of, significantly reducing the computational effort required to solve them.

Benders Decomposition was originally proposed as a method for solving mixed integer programmes, Benders (1962) [6]. The principle behind the method was to partition the variables so as to find and exploit a structure within the mixed integer programme. The method was first applied to 2-stage linear programmes by van Slyke & Wets (1969) [61]. They called the algorithm the L-shaped method, after the block-angular form of the problem. Birge (1985) [8] showed that the L-shaped method in van Slyke & Wets (1969) [61] could be extended to solve multi-stage problems.

The algorithm decomposes a large linear programme, the deterministic equivalent of the original multi-stage stochastic linear programme, into many smaller sub-problems. These are then solved iteratively, with information being passed between sub-problems. The algorithm provides monotonically improving upper and lower bounds on the solution of the problem. Hence the algorithm can be terminated once the gap between the bounds is within a prescribed tolerance. It is also guaranteed to converge for any feasible problem, and to recognise infeasible or unbounded problems.

Since the late 1980s the interest in Benders Decomposition has grown. MSLiP was the first academic version of a Benders Decomposition implementation to be made available, Gassmann (1990) [36]. Since then several other research groups have developed their own implementations of Benders Decomposition , see Thompson (1997) [60], Birge *et al.* (1996) [10] and Morton (1996) [53]. SP/OSL, the first commercially available implementation of Benders Decomposition, was released in 1998. This is an extension to the Optimization Subroutine Library (OSL) routines of IBM, King (1994) [49]. SP/OSL can be used as either a stand-alone solver for multi-stage stochastic linear programmes, or as an extension to the OSL

routines. Two other implementations of Benders Decomposition are commercially available; DECIS Infanger (1994) [46] and SDDP Pereira & Pinto (1991) [55], both of which are designed to incorporate sampling techniques, see Chapter 4.

3.1.2 Simplex Algorithm

The alternative to decomposing the deterministic equivalents of MSLPs, which are very large, highly structured LPs, is to solve them using direction solution methods. As the whole very large problem has to be solved at one time, direct solution methods are limited by the memory of the computer used. In addition as direct methods are unlikely to exploit the structure of the problem, they can take an impractical amount of time to solve real-world problems. The most common direct method for the solution of linear programmes is the simplex algorithm.

The simplex algorithm was first developed in the 1940s. It solves linear programmes by traversing the vertices and edges of the feasible region defined by the constraints of the problem. After determining an initial basis, which defines a vertex of the feasible region, the algorithm moves to a new vertex according to which of the adjacent vertices results in the greatest decrease in the objective function. This assumes that the problem is one of minimisation. The algorithm terminates when it reaches a vertex from which there is no benefit in moving to an adjacent vertex. For a fuller description of the simplex algorithm, how it deals with infeasible or unbounded problems, and how to implement it in practice, see Bazarra *et al.* (1990) [4].

3.2 Derivation of BDM

Multi-stage linear programmes are characterised by a series of interdependent linear programmes. Each of these represents a decision making problem in one time stage. The separate sub-problems are interdependent because the decision made in one stage affect the availability of resources in future stages. Hence the decisions that are available in one time stage depend upon what has happened in previous stages. The objective is to minimise the total costs across all of the time periods.

The main idea underlying Benders Decomposition is to decompose large multi-stage stochastic linear programmes into smaller sub-problems. Each sub-problem has an objective function consisting of two parts. The immediate costs incurred in the current stage and an approximation of the costs in future stages. The future cost function is known as the *recourse function*. The approximation to the recourse function is built up in a sub-problem by determining the future costs for particular decision in the current sub-problem. When the future costs for each of these decision are calculated a linear approximation to the true recourse function is generated and this is added to the approximation.

The main principles of the Benders Decomposition Method are derived in the next subsection where it is applied to a two-stage deterministic problem. In the following subsections, this is extended to the solution of multi-stage stochastic linear programmes, and finally extensions to the basic Benders Decomposition algorithm are presented.

3.2.1 Two-stage Benders Decomposition Method

The following linear programme is one formulation of a two-stage planning problem.

$$
\begin{array}{rlll}
\text{minimse} & c_1x_1 + c_2x_2 & & \\
\text{subject to} & A_1x_1 & = & b_1, \\
& E_2x_1 \; + \; A_2x_2 & = & b_2, \\
& l_1 \leq x_1 \leq u_1, & & \\
& l_2 \leq x_2 \leq u_2, & &
\end{array}
\tag{3.1}
$$

where x_1 and x_2 are the first and second stage decision variables and the costs related to each set of decision variables are c_1 and c_2. The stages are separate points in time when a decision has to be made.

An example of this type of problem is the capacity expansion planning of an electricity generating system. A decision maker has to decide by how much to increase the capacity of various thermal generation plants. This is the decision in the first stage, x_1. Increasing the various plants incurs immediate costs, due to the purchase of new generators. The second stage of the problem is the running of the plants with their new capacities. The operating schedule for some period in the future is the second stage decision, x_2. The amount of capacity available in the second stage depends on the decision made in the first stage. This is given by the second equation in problem (3.1). Stages need not be of the same length. It could be that the first stage in this example would be a few months long, while the second stage could last several years.

The problem given in (3.1) can be decomposed into two coupled sub-problems. A first stage sub-problem given by

$$\begin{aligned} \text{minimise} \quad & c_1 x_1 + \alpha(x_1) \\ \text{subject to} \quad & Ax = b, \\ & l_1 \leq x_1 \leq u_1, \end{aligned} \tag{3.2}$$

and a second stage sub-problem

$$\begin{aligned} \alpha(x_1) = \text{minimise} \quad & c_2 x_2 \\ \text{subject to} \quad & A_2 x_2 = b_2 - E_2 x_1, \\ & l_2 \leq x_2 \leq u_2. \end{aligned} \tag{3.3}$$

The value function, $\alpha(x_1)$, of the second stage sub-problem (3.3) is a convex, piecewise linear function. An example where x_1 is one dimensional is given in Figure 3.1. For any first stage decision $\tilde{x}_1$, $\alpha(\tilde{x}_1)$ represents the resulting second-stage costs, i.e. the recourse of the first stage decision. An approximation to the recourse function is built up by generating linear approximations around first stage decisions. In Figure 3.1 the dotted line $\tilde{C}_1$ is a linear approximation to the recourse function around the decision $\tilde{x}_1$.

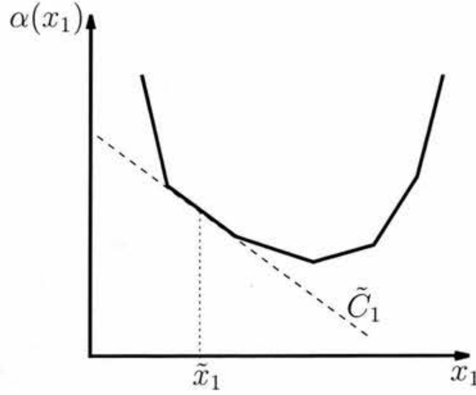


Figure 3.1: The Recourse Function α

Benders Decomposition stores and solves the first and second stage sub-problems as two separate linear programmes. It iteratively builds up an approximation to the recourse function within the first stage sub-problem. This is done by solving the first stage sub-problem, with its current approximation, obtaining a decision $\tilde{x}_1$. Then the second stage sub-problem is solved for this decision. The solution of second stage sub-problem provides dual information from which a linear approximation to the recourse function around the point $\tilde{x}_1$ is constructed. This constraint is used to improve the approximation to the recourse function in the first stage sub-problem.

For any first stage decision, $\tilde{x}_1$, the corresponding second stage sub-problem (3.3) can be re-formulated as its dual,

$$\begin{aligned} \alpha(\tilde{x}_1) = \text{ maximise } \quad & \pi_2(b_2 - E_2\tilde{x}_1) + \lambda_2 l_2 - \mu_2 u_2 \\ \text{subject to } \quad & A_2\pi_2 + \lambda_2 - \mu_2 = c_2, \\ & \lambda_2, \mu_2 \geq 0. \end{aligned} \tag{3.4}$$

The second stage sub-problem in (3.3) may be infeasible for a particular first stage decision, $\tilde{x}_1$. At present we assume that for decision $\tilde{x}_1$ sub-problem (3.3) is feasible, and so its dual problem (3.4) is bounded. Therefore the optimal solution of the dual second stage sub-problem occurs at a vertex of its dual feasible region. For a discussion of duality theory and linear programming in general see Fletcher (1987) [26].

Inspection of the dual sub-problem (3.4) shows that its feasible region does not depend upon the first stage decision, $\tilde{x}_1$. Therefore, it is possible to re-write the dual sub-problem in the following form.

$$\begin{aligned} \alpha(\tilde{x}_1) = \text{maximise} \quad & \pi_2(b_2 - E_2\tilde{x}_1) + \lambda_2 l_2 - \mu_2 u_2 \\ \text{subject to} \quad & \pi_2 \in \Pi_2,\ \Pi_2 = \{\pi_2 \mid A_2\pi_2 + \lambda_2 - \mu_2 = c_2\}, \\ & \lambda_2, \mu_2 \geq 0, \end{aligned} \tag{3.5}$$

where Π_2 is the set of the vertices of the feasible region of the dual of the second stage sub-problem, and is independent of $\tilde{x}_1$. As the solution of the dual sub-problem does not depend upon the first stage decision, $\tilde{x}_1$, it is possible to enumerate all of the feasible second stage vertices, and evaluate the objective function at each of these vertices, finding the value of π_2 which maximises the function $\pi_2(b_2 - E_2\tilde{x}_1) + \lambda_2 l_2 - \mu_2 u_2$. This results in the following problem,

$$\begin{aligned} \text{minimise} \quad & \alpha \\ \text{subject to} \quad & \alpha \geq \pi_2(b_2 - E\tilde{x}_1) + \lambda_2 l_2 - \mu_2 u_2,\ \forall \pi_2 \in \Pi_2. \\ & \lambda_2,\ \mu_2 \geq 0. \end{aligned} \tag{3.6}$$

When solving the original problem (3.1), all of these constraints are used to represent the recourse function of the first stage sub-problem. Benders Decomposition solves a relaxed version of the original problem (3.1). This relaxation is formed by adding constraints from problem (3.6) to the decomposed first stage sub-problem.

Each of the dual feasible vertices of problem (3.6) corresponds to a constraint of the same form as $\tilde{C}_1$ shown in Figure 3.1. Each constraint of the form $\alpha \geq \pi_2(b_2 - E_2\tilde{x}_1) + \lambda_2 l_2 - \mu_2 u_2$ is a supporting hyperplane of the recourse function, $\alpha(x_1)$, with equality at the point $x_1 = \tilde{x}_1$. I.e. they are local linear approximations to the recourse function

For any first stage decision $\tilde{x}_1$, the optimal solution to the second stage sub-problem (3.3), $\tilde{x}_2$, will have a corresponding dual optimal solution, $(\tilde{\pi}, \tilde{\lambda}, \tilde{\mu})$, assuming that the second stage sub-problem is feasible and non-degenerate. This dual optimal solution is used to construct a supporting hyperplane which is exact at $\tilde{x}_1$. Such a constraint is called an *optimality cut.* It corresponds to one of the elements of Π_2, and can be added to the first stage sub-problem in order to improve its approximation of the recourse function.

The algorithm replaces the recourse function $\alpha(x_1)$ in the first stage sub-problem (3.2) by a scalar variable, α. Initially, before any optimality or feasibility cuts have been passed from the second stage sub-problem up to the first stage sub-problem, a large, negative lower bound needs to be given for α. This prevents the initial first stage sub-problem from being unbounded. In each iteration of the algorithm, the first stage sub-problem is solved, and its solution, $\tilde{x}_1$, is passed to the second stage sub-problem. As the first stage sub-problem is a relaxation of the original problem, its objective value is a lower bound on the objective value for the original problem (3.1).

The second stage sub-problem is solved, using the solution to the first stage sub-problem, $\tilde{x}_1$, generating a second stage decision $\tilde{x}_2$. This results in a feasible solution, $(\tilde{x}_1, \tilde{x}_2)$, to the original problem. Hence the objective value associated with this solution, $\tilde{x}_1 c_1 + \tilde{x}_2 c_2$, is an upper bound on the optimal value of the original problem. These two bounds are used as a test for optimality. If the difference between them is within a prescribed tolerance, then the original problem is solved, and the algorithm terminates.

If the bounds have not converged, then the dual solution to the second stage sub-problem is used to create an optimality cut. This is added to the first stage sub-problem, improving its approximation of the recourse function. Another iteration is then performed, using the updated first stage sub-problem. Since the number of dual feasible bases of the second stage sub-problem is finite, i.e. $|\Pi_2| \leq \inf$, there are only a finite number of optimality cuts that can be added to the first stage sub-problem. This is due to each optimality cut being generated from one element of Π_2.

The crux of Benders Decomposition method is that we can approximate the recourse function, $\alpha(x_1)$, by a collection of supporting hyperplanes, i.e. optimality cuts. This collection is normally far smaller than $|\Pi_2|$ for real-world problems, and so the size of the first stage sub-problem when the algorithm terminates will still be much smaller than the original problem.

In the discussion above, we assumed that the second stage sub-problem was feasible for any first stage suggestion. When this is the case the problem is said to have *complete recourse*. In many problems this is not the case, and these problems are said to have *incomplete recourse*. At the start of the algorithm, it is likely that some of the initial first stage decisions will result in the second stage sub-problem being infeasible. This is because the initial approximation to the recourse functions are poor. An infeasible second stage sub-problem has an

unbounded dual, i.e. there exists a direction of unboundedness for the dual second stage sub-problem, $(\tilde{\pi}, \tilde{\lambda}, \tilde{\mu})$, such that $0 \leq \tilde{\pi}(b_2 - E_2\tilde{x}_1) + \tilde{\lambda}l_2 - \tilde{\mu}u_2$, for all possible x_1. Hence adding the constraint

$$\alpha \geq \tilde{\pi}(b_2 - E_2x_1) + \tilde{\lambda}l_2 - \tilde{\mu}u_2 \tag{3.7}$$

to the first stage sub-problem reduces the feasible region of the first stage sub-problem and so eliminates the decision $\tilde{x}_1$ which resulted in the infeasible second stage sub-problem. Such a cut is known as an *infeasibility cut.*

The Benders Decomposition algorithm can be stated as follows:

Step 0 Let $k := 0, \alpha = -M$, some large M,

Step 1 $k := k + 1$, solve the first stage sub-problem, generating $\tilde{x}_1^k$,

Step 2 Solve the second stage sub-problem for $\tilde{x}_1^k$, generating $\tilde{x}_2^k$,

Step 3 If the second stage sub-problem is infeasible, then generate a feasibility cut, add to the first stage sub-problem and goto Step 2,

Step 4 If $(\tilde{x}_1^k c_1 + \alpha^k) - (\tilde{x}_1^k c_1 + \tilde{x}_2^k c_2) \leq$ tol then optimal, otherwise add an optimality cut to the first stage sub-problem,

Step 5 Go to step 1.

Four iterations of this algorithm on a simple problem are shown in Figure 3.2. The graphs show the recourse function and its approximation for a problem which has a one dimensional first stage sub-problem. The bold, piecewise linear convex function is the recourse function, $\alpha(x)$, and the dashed lines represent the cuts which make up the approximation to this recourse function. In the first iteration, the approximation consists of two cuts. The solution in this iteration is x_1. This is passed to the second stage sub-problem. Using the dual information from the solution of the second stage sub-problem, an optimality cut, C_1, is generated as a linear approximation to the recourse function about the point x_1. This cut is added to the approximation. The first stage solution in the second iteration is x_2. This is passed to the second stage sub-problem and its dual solution is used to make cut C_2. This is repeated in the third iteration, with cut C_3 being added to the approximation in the first stage sub-problem. In the fourth iteration, the first stage solution x_4 is optimal.

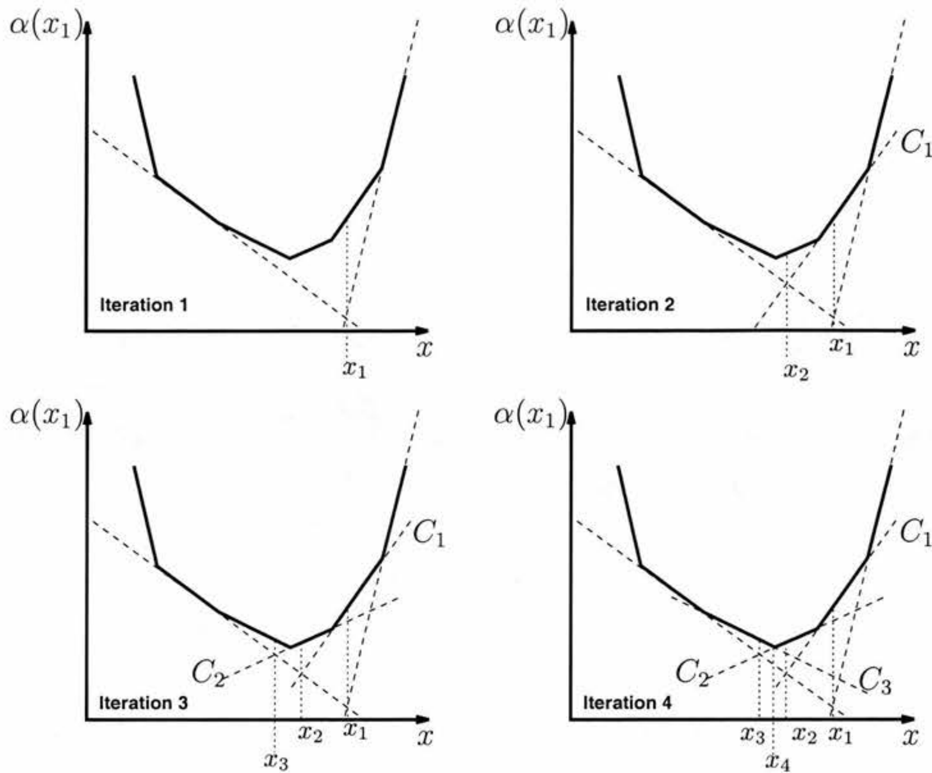


Figure 3.2: Four Iterations of Benders Decomposition

The only reasons for the algorithm to stop are either because the first stage sub-problem is infeasible, or because the bounds are within a specified tolerance. As the number of bases for each sub-problem is finite, the number of optimality cuts that can be passed backwards to a parent is finite. Hence the algorithm is guaranteed to converge in a finite number of iterations, or detect an infeasible problem, Birge (1985) [8].

3.2.2 Uncertain data

The two-stage problem in the previous subsection assumed that all of its data were known with certainty. This is unlikely to be the case when modelling real-world problems, and instead these problems are modelled as stochastic linear programmes.

A two-stage stochastic linear programme can be written as follows

EDINBURGH UNIV. BIBL. EX

$$
\begin{array}{ll}
\text{minimse} & c_1x_1 + I\!E_2(\mathbf{c}_2x_2) \\
\text{subject to} & A_1x_1 \qquad\qquad = \; b_1, \\
 & \mathbf{E}_2x_1 \; + \; \mathbf{A}_2x_2 \; = \; \mathbf{b}_2, \\
 & l_1 \le x_1 \le u_1, \;\; l_2 \le x_2 \le u_2,
\end{array}
\tag{3.8}
$$

where the bold face entries may be stochastic, and $I\!E_2$ denotes the expected value operator with respect to the random variables in the second stage. We assume that these stochastic elements are defined over a discrete probability space, i.e. they can take one of a range of values with a known probability. If the stochastic elements are originally specified as continuous distributions, then it is necessary to discretise this distribution; Benders Decomposition can only solve problems with discretely distributed stochastic parameters. As shown in Wets (1974) [62], the deterministic equivalent problem can be formulated as

$$
\begin{array}{ll}
\text{minimise} & c_1x_1 + \sum_{\omega_2 \in \Omega_2} p_2^{\omega_2} c_2^{\omega_2} x_2^{\omega_2} \\
\text{subject to} & A_1x_1 \qquad\qquad\quad = \; b_1, \\
 & E_2^{\omega_2}x_1 \; + \; A_2^{\omega_2}x_2^{\omega_2} \; = \; b_2^{\omega_2}, \; \omega_2 \in \Omega_2, \\
 & l_1 \le x_1 \le u_1, \\
 & l_2^{\omega_2} \le x_2^{\omega_2} \le u_2^{\omega_2}, \; \forall \omega_2 \in \Omega_2,
\end{array}
\tag{3.9}
$$

where a particular second stage realisation of the stochastic elements of the original problem are denoted by ω_2, $(c_2^{\omega_2}, E_2^{\omega_2}, A_2^{\omega_2}, b_2^{\omega_2})$, and the probability of this realisation occurring is $P(\omega = \omega_2,\ \omega_2 \in \Omega_2) = p^{\omega_2}$.

The problem in (3.9) is of a similar form to problem (3.1) considered in the previous subsection. The main difference is that problem (3.9) contains $|\Omega_2|$ second stage sub-problems that are loosely coupled by the summation in the objective function. Therefore it is possible to decompose this problem into a similar first stage sub-problem as before and $|\Omega_2|$ second stage sub-problems, one for each realisation of the random variable, ω_2. Each of these second stage sub-problems has the same form as the sub-problem in the deterministic case, (3.3).

The recourse function for the first-stage decisions is a convex combination of the piecewise linear objective functions of each of the second stage sub-problems associated with the scenarios $\omega_2 \in \Omega_2$, and so is itself a piecewise linear function. This leads to the following first stage sub-problem.

$$\begin{aligned} \text{minimise} \quad & c_1 x_1 + \sum_{\omega_2 \in \Omega_2} \alpha_{\omega_2}(x_1) \\ \text{subject to} \quad & Ax = b, \\ & l_1 \leq x_1 \leq u_1, \end{aligned} \tag{3.10}$$

where $\alpha_{\omega_2}(.)$ is the value function of the second stage sub-problem for scenario ω_2. The algorithm presented in the previous subsection extends to solve problems such as (3.9).

The algorithm decouples the deterministic equivalent problem into one first stage sub-problem and $|\Omega_2|$ second stage sub-problems. The first stage sub-problem is solved, and for its decision x_1, each of the second stage sub-problems is solved. The upper and lower bounds are computed. If their difference lies within a desired tolerance, then the algorithm terminates. If the algorithm has not found an optimal solution then the dual solutions of the second stage sub-problems are used to construct an optimality cut of the form

$$\alpha(x_1) \geq \sum_{\omega_2 \in \Omega_2} p_2^{\omega_2} \pi_2^{\omega_2} b_2^{\omega_2} - \sum_{\omega_2 \in \Omega_2} p_2^{\omega_2} \pi_2^{\omega_2} E_2^{\omega_2} x_1 + \sum_{\omega_2 \in \Omega_2} p_2^{\omega_2} \lambda_2^{\omega_2} l_2^{\omega_2} - \sum_{\omega_2 \in \Omega_2} p_2^{\omega_2} \mu_2^{\omega_2} u_2^{\omega_2}$$

As in the algorithm presented earlier, this cut is added to the first stage sub-problem and another iteration of the algorithm is performed.

If a first stage decision results in an infeasible second stage sub-problem, then a feasibility cut of the form given in (3.7) in the previous section is added to the first stage sub-problem and the algorithm passes to the next iteration,

3.2.3 Multi-stage Problems

The multi-stage extension of problem (3.8) is the following.

$$\begin{array}{ll}
\text{minimise} & c_1x_1 \; + \mathbb{E}_2 Q_2(\mathbf{c_2}x_2 \; + \mathbb{E}_3 Q_3(\mathbf{c_3}x_3 \; + \ldots \; + \mathbb{E}_T Q_T(\mathbf{c_T}x_T)\ldots)) \\
\text{subject to} & \\
& \begin{array}{lllllll}
A_1x_1 & & & & & = & b_1, \\
\mathbf{E_2}x_1 & + & \mathbf{A_2}x_2 & & & = & \mathbf{b_2} \\
& & \mathbf{E_3}x_2 & + & \mathbf{A_3}x_3 & = & \mathbf{b_3}, \\
& & \ddots & & \ddots & & \vdots \\
& & & \mathbf{E_T}x_{T-1} & + \; \mathbf{A_T}x_T & = & \mathbf{b_T},
\end{array} \\
& l_1 \le x_1 \le u_1, \;\; \mathbf{l_t} \le x_t \le \mathbf{u_t}, \;\; t = 1, \ldots, T,
\end{array}$$

where the bold face entries represent possibly stochastic elements, and $\mathbb{E}_t$ is the expectation operator with respect to random variables in stage t. As for the two stage case, we assume that the random variables are defined over a discrete probability space, if necessary discretising any continuous distributions. The deterministic equivalent of this problem is

$$\begin{array}{ll}
\text{minimise} & c_1x_1 + \sum\limits_{\omega_2 \in \Omega_2} p_2^{\omega_2} c_2^{\omega_2} x_2^{\omega_2} + \sum\limits_{\omega_3 \in \Omega_3} p_3^{\omega_3} c_3^{\omega_3} x_3^{\omega_3} + \ldots + \sum\limits_{\omega_T \in \Omega_T} p_T^{\omega_T} c_T^{\omega_T} x_T^{\omega_T} \\
\text{subject to} & \begin{array}{lll}
A_1x_1 & = & b_1, \\
E_2^{\omega_2}x_1 + A_2^{\omega_2}x_2^{\omega_2} & = & b_2^{\omega_2}, \; \omega_2 \in \Omega_2 \\
\quad E_3^{\omega_3}x_2^{\theta(\omega_3)} + A_3^{\omega_3}x_3^{\omega_3} & = & b_3^{\omega_3}, \; \omega_3 \in \Omega_3, \\
\quad\quad \ddots \quad\quad \ddots & & \vdots \\
\quad\quad\quad E_T^{\omega_T}x_{T-1}^{\theta(\omega_T)} + A_T^{\omega_T}x_T^{\omega_T} & = & b_T^{\omega_T}, \; \omega_T \in \Omega_T,
\end{array} \\
& l_1 \le x_1 \le u_1, \;\; l_t^{\omega_t} \le x_t^{\omega_t} \le u_t^{\omega_t}, \;\; \forall \omega_t \in \Omega_t, \; t = 1, \ldots, T,
\end{array} \qquad (3.11)$$

where p^{ω_t} is the probability of scenario ω_t occurring in stage t, $\theta(\omega_t)$ is the ancestor for scenario ω_t in stage t, and the other elements are defined as follows from

problem (3.9). Each realisation of the stochastic parameters in stage $t > 2$ can depend on the value that the stochastic parameters took in stage $t - 1$. The ancestor function associates the current value of the stochastic variables with a realisation in the previous stage.

In the multi-stage problem given above, it is possible to systematically decompose the original problem into a sequence of smaller two stage problems. The initial problem can be decomposed into a first stage sub-problem consisting of stage $t = 1$, and a sub-problem corresponding to each realisation of the second stage random variables, $\omega_2 \in \Omega_2$, where each sub-problem contains stages 2–T. This decomposed problem can be solved using Benders Decomposition method, as it has the form of the stochastic two stage sub-problem discussed in the previous subsection.

The sub-problems of the decomposed problem can be further decomposed. Each of the $|\Omega_2|$ sub-problems can be split into a new "first stage" sub-problem, that contains the second stage problem for a particular realisation of the second stage stochastic variables, and a collection of sub-problems that contain stages 3 – T. The number of sub-problems depends upon the dependencies between the realisation of second stage random variables and the possible realisations of the third stage random variables. This process can be continued until the final decomposition has stage $T - 1$ problems as the "first stage" sub-problem, and stage T problems as the "second stage" sub-problems.

The dependencies between the stochastic parameters combined with the periodic structure of a multi-stage stochastic programme result in the representation of multi-stage stochastic programmes as event trees, an example of which is given in Figure 3.3

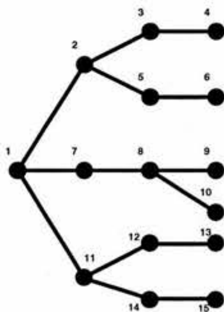


Figure 3.3: An Event Tree

This event tree has four stages, $T = 4$, and the following realisations of the stochastic parameters: $\Omega_2 = \{$ 2, 7, 11$\}$, $\Omega_3 = \{$3, 5, 8, 12, 14$\}$, $\Omega_4 = \{$4, 6, 9, 10, 13, 15$\}$. The nodes of the event tree represent the individual linear programmes

that are the result of repeatedly decomposing the original multi-stage stochastic linear programme. The arcs in the tree represent the possible realisations of the stochastic parameters in each stage. These realisations depend upon from where in a stage a node branches. This leads to the concept of a *parent*, or an *ancestor*, of a node. This is the sub-problem in the previous stage which precedes a sub-problem. In Figure 3.3 the parent of node 10 is node 8, or in the notation of problem (3.11), $\theta(10) = 8$.

It is possible for there to be no dependencies between the stochastic parameters in adjacent stages. In Figure 3.3 this might be the case if nodes 2 and 11 are different, but their descendents are not, i.e. nodes 3 and 4 are the same as nodes 12 and 13, and nodes 5 and 6 are the same as nodes 14 and 15. In this case, it is possible to avoid this duplication of sub-trees by "folding" the event tree into a graph. Now node 11 has nodes 3 and 5 as children, and nodes 12, 13, 14 and 15 are no longer needed. Modelling multi-stage stochastic linear programmes that have this type of structure is discussed in Chapter 2 and their solution using Benders Decomposition is discussed in Section 3.5 in the current chapter.

The solution of a multi-stage problem using Benders Decomposition can be derived from the two-stage algorithm presented earlier by applying it in a similar recursive manner to the way we decomposed the problem into an event tree of sub-problems. After solving the root node, the first stage decisions are passed down to the second stage sub-problems. These sub-problems are solved, and they pass their decisions down to the third stage sub-problems. This process is continued until all of the sub-problems have been solved. This forms the *forwards pass* of the algorithm.

In the *backwards pass* the dual solutions for each of the sub-problems are used to generate cuts and these are passed backwards to the parents of the sub-problems. First the dual solutions of the leaf nodes are used to generate cuts that are passed up to the stage $T-1$ sub-problems. These problems use their dual solutions to generate cuts that are passed backwards to their ancestors. This is continued until a cut is placed in the first stage sub-problem. The first-stage problem is re-solved, and its bounds checked. If the solution is not optimal then another iteration of Benders Decomposition is performed.

This algorithm is one interpretation of how the two-stage algorithm can be implemented for multi-stage problems. The main flaw in the interpretation given above is that the cuts generated during the backwards pass are not used until the next iteration. This means that new information has been sitting idle. Instead,

after a cut is placed in a sub-problem, this sub-problem can be re-solved. The dual solution associated with this solution can then be used to generate a cut that is passed up to its parent. In this way the new information has been passed up through the tree during the current iteration, and so is used when deciding whether the problem is optimal at the end of the current iteration.

A further observation is that, once a non-leaf or non-root sub-problem of a multi-stage stochastic linear programme has been solved, there are two ways in which the new information can be passed. Either the primal solution is passed down to the children of the current node, or the dual solution is used to generate a cut, that can then be passed to the parent of the current node. This raises the issue of when is it in the best interests of the algorithm to pass information forwards or backwards through the tree. In the next section one of the extensions to the basic Benders Decomposition algorithm that we look at is the possibility of using different tree traversing strategies.

3.3 Algorithmic Extensions

The basic Benders Decomposition algorithm presented in the previous subsection can be extended in several ways. Some of these are applicable only to multi-stage problems, tree traversal strategies, while others are applicable to both two-stage and multi-stage problems, preliminary cuts, hot-starts, cut sharing and multi-cuts. At the end of this section we present result that show what the effects of these extensions are on the basic algorithm.

3.3.1 Multi-cuts

The multi-cut algorithm was first applied to the L-shaped method by Birge & Louveaux (1988) [12]. Whereas the Benders Decomposition method given above builds one approximation to the recourse function of a sub-problem, the multi-cut algorithm builds a separate approximation to the recourse function for every child of the sub-problem. In the original algorithm, all of the cuts generated by the children of a sub-problem are aggregated into one cut that is added to the ancestor sub-problem. When a two stage problem is decomposed, one scalar variable is introduced into the first stage sub-problem to represent the future costs. In the multi-cut algorithm, this is replaced by $|\Omega_2|$ scalar variables, one for each possible second stage scenario. Cuts of the following form are then added to the first stage sub-problem at each non-optimal iteration,

$$\alpha_{\omega_2} \geq p_2^{\omega_2}\pi_2^{\omega_2}b_2^{\omega_2} - p_2^{\omega_2}E_2^{\omega_2}x_1 + p_2^{\omega_2}\lambda_2^{\omega_2}l_2^{\omega_2} - p_2^{\omega_2}\mu_2^{\omega_2}u_2^{\omega_2},$$

where α_{ω_2} is the scalar variable in the objective function of the first stage sub-problem that represents the future costs incurred in second stage scenario ω_2.

The benefit of using multi-cuts results from more detail being added to a recourse function than if the cuts from the subsequent stage are aggregated. This is likely to reduce the overall number of iterations that the Benders Decomposition algorithm needs to solve a problem. The cost of this increased detail is that the number of cuts added to each sub-problem is increased, by a factor equal to the number of children associated with the sub-problem, $|\Omega_{t+1}|$ for a stage t sub-problem, in addition to the increased number of decision variables that are used to approximate the future costs; one for each child, as opposed to just one for the aggregated future costs. Care also has to be taken not to accumulate redundant cuts at a sub-problem when one of its children is infeasible.

In Birge & Louveaux (1988) [12], a counter example is given where the number of iterations need to solve the problem using multi-cuts is greater than with the aggregated cut algorithm. They conclude that in general the multi-cut algorithm results in less iterations than the aggregate cut method for two-stage problems. Two more recent studies have given conflicting results about the benefits of the multi-cut algorithm.

3.3.2 Cut Sharing

If there is no dependency between the stochastic parameters in different stages of a multi-stage stochastic linear programme, then the recourse functions of the sub-problems across each stage do not depend upon the realisation of the stochastic elements. Hence it is possible to share cuts generated for one sub-problem in a stage of the problem with other sub-problems in the same stage.

Assume that the stochastic parameters of a multi-stage stochastic linear programme have no inter-stage dependency. Then the dual solution of one sub-problem can be used to generate a dual feasible solution of any other sub-problem in the same stage. A sub-problem associated with scenario ω_t in stage t of a multi-stage stochastic linear programme, which we assume has complete recourse, has the following form

$$\begin{aligned} \text{minimise} \quad & c_t^{\omega_t} x_t^{\omega_t} + \alpha_t^{\omega_t} \\ \text{subject to} \quad & A_t^{\omega_t} x_t^{\omega_t} = b_t^{\omega_t} - E_t^{\omega_t} x_{t-1}^{\theta(\omega_t)}, \\ & D_t^{\omega_t} x_t^{\omega_t} + e\alpha_t^{\omega_t} \geq d_t^{\omega_t}, \\ & l_t^{\omega_t} \leq x_t^{\omega_t} \leq u_t^{\omega_t}, \end{aligned} \tag{3.12}$$

where the second set of constraints represents the optimality cuts that have been added to the sub-problem. $D_t^{\omega_t}$ represents the gradients of the optimality cuts, $d_t^{\omega_t}$ the associated cut intercepts, and e is a vector of 1's. The dual of this problem is the following,

$$\begin{aligned} \text{maximise} \quad & \pi_t^{\omega_t}(b_t^{\omega_t} - E_t^{\omega_t} x_{t-1}^{\theta(\omega_t)}) + \lambda_t^{\omega_t} l_t^{\omega_t} - \mu_t^{\omega_t} u_t^{\omega_t} + \rho_t^{\omega_t} d_t^{\omega_t} \\ \text{subject to} \quad & A_t \pi_t^{\omega_t} + \lambda_t^{\omega_t} - \mu_t^{\omega_t} - D_t^{\omega_t} \rho_t^{\omega_t} \leq c_t^{\omega_t}, \\ & e\rho_t^{\omega_t} = 1, \\ & \lambda_t^{\omega_t}, \mu_t^{\omega_t}, \rho_t^{\omega_t} \geq 0, \end{aligned} \tag{3.13}$$

where the dual vectors $\pi_t^{\omega_t}, \lambda_t^{\omega_t}, \mu_t^{\omega_t}$ and $\rho_t^{\omega_t}$ correspond to the original constraints in (3.12), the lower and upper bounds on the primal variables, and the optimality cuts in the primal sub-problem respectively. An optimal solution to the dual sub-problem (3.13), $(\tilde{\pi}_t, \tilde{\lambda}_t, \tilde{\mu}_t, \tilde{\rho}_t)$, can be used to generate a feasible dual solution for any other stage t sub-problem, say $\hat{\omega}_t$. The vector $(\hat{\pi}_t, \hat{\lambda}_t, \hat{\mu}_t, \hat{\rho}_t) = \left(\tilde{\pi}_t, \tilde{\lambda}_t, \left[A_t^{\hat{\omega}_t}\tilde{\pi}_t + \tilde{\lambda}_t - D_t^{\hat{\omega}_t}\tilde{\rho}_t - \hat{c}_t\right]^+, \hat{\rho}_t\right)$, is a feasible solution to sub-problem $\hat{\omega}_t$, where $[v]^+$ takes the component-wise positive entries of the vector v, Morton (1993) [52].

The dual feasible solution derived for sub-problem $\hat{\omega}_t$ from the dual solution of sub-problem (3.13) is unlikely to be optimal. However, it is a feasible solution, and so can be used to generate a valid cut about the decision, $x_{t-1}^{\theta(\omega_t)}$, from the previous stage. A cut is said to be valid if it lies below the recourse function for all possible decision in the previous stage. It need not be a supporting hyperplane for any one decision from the previous stage. Hence adding a cut that has been generated from the dual solution of another sub-problem may not refine the approximation to a recourse function. This is particularly true when the algorithm is close to terminating. However, during the initial iterations of the algorithm, the approximation to a recourse function is likely to be poor, and so any valid cut is of use in helping to define the broad shape of the recourse function. If cut sharing is used through out the algorithm, there is likely to be a build up of valid but inexact cuts at sub-problems. This can slow down the algorithm, as the sub-problems grow due to the ineffective cuts, and take longer to solve.

The sharing of cuts is vital when using Benders Decomposition with sampling techniques to solve very large multi-stage linear programmes. As discussed in Chapter 4, only a small proportion of the possible paths through a multi-stage stochastic linear programme are selected when using a sampling based algorithm. Hence, the likelihood of ever solving a particular sub-problem in the final stages of the problem is very small. Sharing cuts helps to generate valid, though inexact, cuts from these "un-sampled" sub-problems. Therefore more scenarios are taken into account when the recourse functions are approximated. Without the use of cut sharing, their contributions would not be considered at all.

The cut sharing result is extended to problems that show particular types of dependency amongst random elements between stages in Infanger & Morton (1996) [47]. They look at problems where the right-hand side follows a linear lag-one model, and extend this to higher order linear lag models. As was discussed in Chapter 2, when there is no inter-stage dependency between the stochastic

parameters in different stages of a multi-stage stochastic linear programme, the problem can be formulated as a model link graph, rather than as an event tree. In Section 3.5 we show that this results in a type of cut sharing where the cuts are always supporting hyperplanes, rather than valid cuts.

3.3.3 Hot-starts

The level of similarity between sub-problems in the same stage of a multi-stage stochastic programme is high. Often it is only a few elements that vary between sub-problems, e.g. the right-hand sides or the cost coefficients. This similarity between sub-problems can be exploited to provide possibly feasible bases with which to start the simplex method on sub-problems, rather than solving them from scratch.

During the initial iteration of the Benders Decomposition algorithm, each sub-problem is solved for the first time. This requires a large number of simplex iterations as there are no previous solutions and their bases from which to start. The standard solution to this situation when solving a single linear programme is to use a crash basis. This is a basis that is constructed so that it is feasible. However, using crash bases in Benders Decomposition would require a separate crash basis for each sub-problem. Instead a crash basis is used for the first sub-problem in a stage to be solved. The optimal basis for this sub-problem is then used as an initial basis for the other sub-problems in the stage. This basis may not be feasible for the other sub-problems, but when the difference is small between sub-problems in the same stage it will provide a good approximation to the optimal basis.

After the initial pass of the algorithm, the sub-problems change relatively little between their solution in one iteration and the next. The most that can happen is that one or a few cuts, if the multi-cut algorithm is being used, are added to a sub-problem. Therefore it is beneficial to use the optimal basis from the previous solution of a sub-problem as the initial basis when next solving the same sub-problem. Results given later show that this use of *hot-starts* generally leads to reductions in the solution times for the whole algorithm.

3.3.4 Preliminary Cuts

During the initial iteration of the Benders Decomposition algorithm, the approximations to the recourse functions in the sub-problems are vague; only a

large negative lower bound that ensures that the recourse function is bounded below. Therefore all of the decisions made in the first forward pass of the algorithm are myopic, as the approximations to the recourse functions have no information about the future.

The variables that represent the decisions in a multi-stage stochastic linear programme are usually bounded below and often have upper bounds as well. This means that a sub-problem in the final stages can estimate the range of values the decisions from the previous stage can take. For example, in the hydro-electricity problems discussed in Chapter 5 the amount of water in the reservoirs is bounded; a minimum amount of water must be left in each reservoir at the end of each stage and the reservoirs only have finite capacities. In addition, the reservoirs cannot be run dry at the end of the planning horizon, as they will most probably continue to operate after this point in time. To account for this, bounds are placed on the final reservoir levels to ensure that there is a reasonable amount of water remaining.

When the algorithm begins, the bound on the final reservoir levels is only present in the final stage sub-problems. Hence the decisions made in the first forward pass have no information about these bounds, and tend to run the reservoirs at their lowest levels. This is because there is no incentive to conserve water to meet the final bounds.

Preliminary cuts help to avoid this inefficient first pass. They are generated by performing a backwards pass through the problem using trial decisions. First the final two stages of the problem are considered. A possible trial decision from the stage $t-1$ sub-problems is the worst case when the decision is to leave all resources at their lowest amounts. If the decisions are bounded above, then another trial decision is to leave everything over to the final stage. Any feasible decision can be used.

The decision is passed to the final stage sub-problems, and they generate and pass cuts back. These cuts are added to the stage $t-1$ sub-problems. Because the decision that generated the cuts is feasible, the cuts are valid. Trial decisions from the stage $t-2$ sub-problems are then passed forwards and the sub-problems solved. These pass the cuts that they generate to their parents and so the backwards passes progresses to the root node. All of the cuts added contain information about the future costs and are valid. Hence they are better than the approximations to the recourse functions when the algorithm starts from scratch.

We have implemented preliminary cuts in our Benders Decomposition algorithm. As there is no obvious pattern to the type and number of trial decisions that should be used, we have implemented a simple heuristic. As there are a large number of reservoirs in the problems we were investigating, we assumed that all of the reservoirs were equally full, relative to their capacity. This heuristic is particularly useful when solving large problems using sampling within the Benders Decomposition as it results in the initial samples providing reasonable results.

3.3.5 Tree Traversal

Benders Decomposition produces two types of information; primal solutions that are passed forwards to the children of a sub-problem and dual solutions that are passed backwards, in the form of optimality or feasibility cuts, to the parent of a sub-problem. For two-stage stochastic linear programmes the algorithm performs forwards and backwards passes one after another. When solving a multi-stage stochastic linear programme this need not be the case. At any point during the algorithm, after a sub-problem has been solved, the resulting solutions can either be passed forwards or backwards, except for the root and leaf sub-problems. The protocol that guides the order in which to solve the sub-problems of a multi-stage stochastic programme is called a *tree traversing strategy.* Three different strategies have been considered in the literature: *fast-forward-fast-back* (FFFB), *fast-forward* (FF), and *back-first* (BF).

The first of these, FFFB, is the method suggested in the previous section and was proposed in Wittrock (1985) [64]. In each iteration a complete forwards pass is performed, followed by a complete backwards pass. If an infeasible sub-problem is found during the forwards pass, then this pass is terminated and the algorithm changes direction, taking the feasibility cut backwards to the first stage. The strategy FF was proposed in Birge (1985) [8], in an implementation of nested Benders Decomposition. This strategy only passes a cut backwards from the current sub-problem when the sub-tree rooted at this sub-problem is optimal. The third approach, BF, is a more "cautious" strategy than the other two. Here a decision is only passed forwards to the next stage of the problem if there have been no new cuts generated by solving the sub-problems in the current stage.

The three strategies behave as follows on the multi-stage stochastic programme represented by the event tree in Figure 3.3. FFFB solves node 1 and passes the decision to nodes 2, 7 and 11 which are then solved. Their solutions are passed to nodes 3 and 5, 8, 12 and 14. These in turn are solved and their solutions passed

to the leaves. The dual solutions from the leaves are used to generate optimality cuts that are placed in nodes 3, 5, 8, 12 and 14, and these nodes are re-solved. Their dual solutions are used to generate cuts that are placed in nodes 2, 7 and 11. Finally these nodes are re-solved and their dual solutions used to generate a cut to place in node 1. If the bounds at node 1 are not tight enough then another iteration is performed, otherwise the algorithm terminates.

The FF strategy behaves similarly to FFFB until the leaves have been solved and cuts passed to their parents. These sub-problems are then re-solved and their bounds are checked for optimality. If they are not optimal, then their new primal solutions are passed forwards and their children re-solved. Only when a third stage node is optimal is its current dual solution used to generate a cut that is passed back to its parent. This process is then repeated for the sub-trees rooted in the second stage, and once all of these are optimal a cut is placed in the root node.

The third strategy, BF, is the reverse of the FF strategy. It does not pass primal solutions forwards until the sub-tree consisting of the sub-problems from the root problem to the stage of the current sub-problem is solved to optimality. For instance, only when the sub-tree consisting of nodes 1, 2, 7 and 11 in Figure 3.3 is optimal will the primal solutions of nodes 2, 7 and 11 be passed down to the third stage nodes. The three stage sub-tree is then solved to optimality before the final stage is included.

All three of these strategies have been tested in the literature, by Gassmann (1990) [36] and Morton (1993) [52], and their agreed conclusion is that FFFB is the preferred strategy for problems with many stages, while FF is better for smaller problems, with less stages.

3.3.6 Results for Extensions

The following results come from the solution of a four and a six stage multi-stage stochastic linear programme. These problems are hydro-electricity reservoir management problems that use as their base the model discussed in Chapter 5. The networks consist of thirty three thermal plants and thirty three reservoirs connected in a series of inter-connected valleys. Each time period is divided into three load duration blocks and the transmission network has one demand node. The non-leaf nodes each have three children.

		Hot-Starts		No Hot-Starts	
		FF	FFFB	FF	FFFB
4 Stage Problem					
	CPU (s)	60.69	66.18	71.88	72.96
	Passes	29	37	36	38
	Splx Its	50771	52735	71999	76456
	First Splx Its	6263	6165	24521	24521
	After Splx Its	44508	46570	47478	51935
	Cuts	4092(1364)	4440(1480)	5088(1696)	4560(1520)
6 Stage Problem					
	CPU (s)	268.70	253.44	302.78	423.46
	Passes	37	35	43	67
	Splx Its	86632	86253	291405	303440
	First Splx Its	21122	21122	215818	215818
	After Splx Its	65510	65131	75587	87622
	Cuts	40791(13597)	38220(12740)	47472(15824)	73164(24388)

Table 3.1: 4 & 6 Stage Problems, Multi-Cuts

Our implementation of Benders Decomposition is written in Fortran 77, and uses the simplex solver in the IBM Optimization Subroutine Library (OSL) Version 2 [45] to solve the sub-problems. The following experiments were run on a 333MHz Sparc Ultra-5 with 64MB of memory. Each run was solved using a relative tolerance of 10^{-6} in the objective value as the criterion for optimality.

Three different extensions to the basic algorithm are considered: the use of aggregated cuts or multi-cuts, hot-starts and the choice of tree traversal strategy, FFFB or FF. Both of the test problems were solved using every combination of these solver options. These computational results are presented in Tables 3.1–3.2. Table 3.1 contains the results for multi-cut algorithm for the four and six stage problems, while Table 3.2 contains the results for the aggregated cut algorithm for the four stage and the six stage problems.

Each of the four tables is further divided into the following columns. The first two columns contain the results for hot-starts, the first of these for the FF protocol and the second for the FFFB protocol. The next two columns present the results when hot-starts are not used, with one column for the FF protocol and the other for FFFB.

The rows of each table contain the following information for the different runs of the solver. The CPU time, in seconds, the number of iterations of the algorithm, the total number of simplex iterations, this is then divided into the number of iterations on the first pass of the algorithm and the total number in the remaining iterations, and finally the total number of cuts added. In Table 3.1 the figures in brackets are the total number of cuts divided by the number of children for non-leaf nodes.

		Hot-Starts		No Hot-Starts	
		FF	FFFB	FF	FFFB
4 Stage Problem					
	CPU (s)	136.56	108.80	144.60	105.79
	Passes	11	59	12	55
	Splx Its	78346	83238	100662	104294
	First Splx Its	5424	5259	24143	24143
	After Splx Its	72922	77979	76519	80151
	Cuts	2436	1794	2648	1753
6 Stage Problem					
	CPU (s)	322.48	356.11	453.85	388.69
	Passes	7	57	8	69
	Splx Its	126118	139743	371231	327998
	First Splx Its	18319	18390	214724	214724
	After Splx Its	107799	121353	156507	113274
	Cuts	5286	4925	8395	5060

Table 3.2: 4 & 6 Stage Problems, Aggregated Cuts

The first point to note from these results is that in all but one instance multi-cuts out performs aggregated cuts. The one exception is when solving the six stage problem using FFFB and no hot-starts. This agrees with the results stated in Gassmann (1990) [36], that multi-cuts generally give a slight improvement in solution times over aggregated cuts. However, in Morton (1996) [53], the improvements in the CPU times using multi-cuts were 35% on average for the four tests problems in the study. For our four stage problem the average improvement is 44.0%, and for the six stage problem the average improvement is 18%, and the average for all of the results is 31%. The difference between the performance for the four and the six stage problems can partially be explained when the increase in the total number of cuts is compared for the two problems. Using multi-cuts on

the four stage problem results in an average of 2.2 times as many cuts being added to sub-problems. This compares with a factor of 8.9 for the six stage problem. In summary, the reductions in solution time gained by using multi-cuts is greater for the smaller problem, although on average multi-cuts reduces the solution time by a third.

The second point to draw from these results is that the use of hot-starts results in shorter solution times in all but one of the cases, where multi-cuts and FFFB are used to solve the four stage problem. This is the only case where the number of cuts added when hot-starts is used is greater than the number added when not using hot-starts. For the four stage problem the average improvement through using hot-starts is 7% and for the six stage problem the improvement is 22%.

There appears to be less of a clear result to draw about the differences in performance for the two tree traversal strategies. For the four stage problem, the FF strategy is the best when solving using multi-cuts and for aggregated cuts the FFFB strategy is better. This distinction cannot be made for the six stage problem. In most cases, the strategy that takes the least time to solve also generates the least number of cuts.

3.4 Parallelisation of Benders Decomposition

Solving a multi-stage stochastic linear programme by Benders Decomposition requires the solution of many sub-problems. The majority of these are independent of each other because they are in different parts of the event tree. Hence these sub-problems can be solved independently of each other. This property makes Benders Decomposition suitable for implementation on a parallel computer. Sub-problems are distributed amongst several processors and solved concurrently when possible. In this section we discuss the issues involved when implementing parallel algorithms, present our parallel implementation of Benders Decomposition method and finally present results for this implementation.

3.4.1 Parallel Algorithms

One definition of a parallel computer is any computing environment that consists of independent processors or memory, and a means of communicating between them. This ranges from a collection of networked workstations to purpose-built multi-processor supercomputers, such as a Cray T3D. All parallel computers have the ability to either perform the same task simultaneously on many different pieces of data, or to perform many different tasks simultaneously on different data. This leads to the following classification of computers.

Traditional computers are known as SISD, single instruction, single data machines. These perform an algorithm sequentially on one item from a collection of data at any one time. The two most common types of parallel computers in use are SIMD and MIMD. The first of these, single instruction, multiple data machines, process only one instruction at any one moment, but this task is carried out on a collection of processors, all of which have their own data. The second type, multiple instruction, multiple data machines, is the most common type of parallel architecture. Included in this category are both multiprocessor and multicomputer machines.

The distinction between multiprocessors and multicomputers is that the former consists of several processors which share a common pool of memory. This memory is either concentrated in one central location, or it is distributed amongst the processors. Whichever is the case, each processor has access to any item of data, regardless of where it is actually situated. Each processor in such a parallel machine is independent and runs its own program. However, an algorithm written for such a multiprocessor environment should take into account the distribution

of the shared memory; while processors are unaware of the physical location of the shared memory, the time taken to access the memory may depend upon its distance from a processor.

Multicomputers consist of processors that each have their own memory, upon which separate programmes act. These processors interact by passing messages between themselves. While the network that connects the processors does not usually consist of a direct link between each pair of processors, the messages being passed through intermediate processors do not stop these processors, but can reduce their processing speed. We have implemented our parallel Benders Decomposition algorithm on a collection of Sun workstations, linked by a local area network; a multicomputer

In addition to re-writing an algorithm to work over several processors, when implementing a parallel version of an algorithm the communication of information between processors and the amount of memory at each processor have to be considered. Most parallel algorithms that run on multicomputers have one processor that controls the overall execution of the algorithm. This *master process* initiates the multicomputer, identifying which processors are being used and spawns processes on the other *slave processors*. The process running on the different processors may be identical, or each process may be dedicated to performing particular tasks. The difference between the master and slave processes is that the master process initiates and terminates the overall algorithm. Once the algorithm has terminated it is the master process that terminates the slave processes and collects the final results.

The data of the problem is divided between processors so that independent parts of the algorithm work concurrently. The distribution of work between the different processors effects the performance of the algorithm. If the work load varies considerably between processors, then the available processing power is used inefficiently. As an example of poor load balancing, imagine a two processor multicomputer being used to solve a two-stage stochastic linear programme. If one processor is allocated the first stage sub-problem and the other is allocated all of the second stage sub-problems, then the first processor will sit idle for a significant proportion of the running time.

When a multicomputer consists of a heterogeneous collection of processors, the question of load balancing is even more important; some processors might be many orders of magnitude faster that others and so the amount of tasks that they can do is also greater. A similar question arises when the amount of memory at each

processor varies. Processors with small memories take longer to perform tasks that deal with large amounts of data due to paging. This is when a processor only has enough memory to work on one portion of the data it needs at any one time. Once a particular portion of data has been dealt with it is temporarily written to disk and a new portion read in. This extra input and output can drastically slow down a task.

A further consideration is the amount and speed of communications that will be necessary between processors when implementing a parallel algorithm for a multicomputer. If the speed of passing a message is slow relative to the speed of the processors, then the algorithm should try to minimise the number of messages sent. Otherwise the processors often sit idle, waiting for a message to arrive.

The sequencing of communications must be considered in addition to their speed. Regardless of how quickly messages are passed between processors, these messages are processed once they arrive at a processor. The receiving processor is not able to do any of its allotted work while it is processing messages, nor can it deal with more than one message at a time. Therefore, it is necessary to design parallel algorithms that avoid bottle necks in communications. This is when a queue of messages are waiting to be processed by one processor. This processor is prevented from continuing with its own tasks. In addition, other processors are likely to be waiting for replies before continuing with their tasks.

The primary aim of implementing an algorithm on a parallel machine is to reduce the overall time required to solve problems. There are two criteria used to judge how effectively an algorithm has achieved this goal. The first measures the *speedup*. This is the ratio of the time taken to solve a problem by a sequential implementation of the algorithm, and the time taken to solve the same problem by the parallel implementation. The *efficiency* measures how evenly balanced the processors are, and is defined as the ratio of speedup to number of processors used. Inevitably, due to the need for communications between processors, most parallel algorithms are less than 100% efficient. However, careful attention to the load balancing on processors and to the passing of messages so as to avoid bottlenecks leads to highly efficient parallel algorithms.

3.4.2 Parallel Benders Decomposition Algorithms

Our parallel implementation of Benders Decomposition is guided by both the architecture of our parallel computer, a network of workstations, and by the shape and form of the multi-stage stochastic programmes that we were solving;

symmetrical trees where the sub-problems have consistent dimensions. The first of these requires that the amount of communications is kept to a minimum. One way to achieve this is to divide the sub-problems of the original problem between processors such that the sub-problems on each processor are as self-contained as possible. An example of this is given in Figure 3.4.

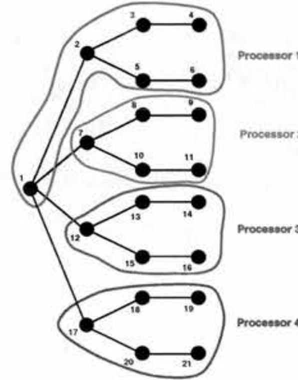


Figure 3.4: Allocation of Sub-Problems to Processors

In the figure, the event tree of the original multi-stage stochastic programme is divided into four sub-trees, each allocated to a separate processor. Each sub-tree is self-contained in that once a decision is passed to the root node of any sub-tree, that tree does not need to communicate to any other tree, except when its root node has been solved and dual information is ready to be passed back to the parent of the root node. Hence the processors can continue computing concurrently until the dual information needs to be passed to the parent of the root node.

In the example given in the figure, the division of the original problem between four processors is evident, as the tree naturally branches into four second stage sub-problems. For different numbers of processors and shapes of tree, the general rule of thumb is to divide the tree by splitting it as close as possible to the first stage, so that the resulting parts are sub-trees of the original tree. There should be the same number of sub-trees as processors. Inevitably, very few MSLPs generated for real-world problems have a branching structure that is identically to the number of processors available. In this case, the load balancing between the processors will be poorer than for the problem shown in the example.

The form that our parallel implementation of Benders Decomposition takes is almost the same as the serial algorithm. However, the choice of tree traversing strategy is more complicated then in the serial algorithm. While the majority of primal and dual solutions are passed to other sub-problems on the same processor, it is necessary to coordinate the passing of solutions between processors. Our

implementation uses the FFFB protocol both within the sub-trees on individual processors and for the passing of primal and dual solutions between sub-problems on different processors.

It is also possible to use different tree traversing strategies when solving stages whose sub-problems are split across processors. The FF strategy for these stages would be preferable for multicomputers where the speed of communications between processors is slow. This ensures that the cuts that are passed between processors are from optimal sub-trees. The aim of this is to try to reduce the number of overall iterations of the algorithm, as each of these requires inter-processors communications. As our multicomputer was a collection of workstations with a relatively quick local area network linking them, our parallel implementation of Benders Decomposition uses the FFFB protocol for stages that are spread across different processors.

The way in which infeasibility cuts are dealt with is the final difference between our parallel Benders Decomposition algorithm and the serial version. These cuts are passed backwards through the problem as in the serial implementation. In addition, the other processors are sent messages informing them to stop processing any decisions that are related to the infeasible decision. Once all of the processors have stopped processing these decisions an infeasibility cut is placed in the parent sub-problem, and the algorithm proceeds.

In the next section we discuss results from our parallel implementation of Benders Decomposition. These results also appear in Archibald *et al.* (1999) [1], where our parallel implementation of Benders Decomposition is compared against a standard dynamic programming algorithm, a modified dynamic programming algorithm and the revised simplex method on a collection of hydro-electric reservoir management problems.

Other parallel implementations of the Benders Decomposition algorithm consist of a similar sub-tree based approach, Birge *et al.* (1996) [10], and two farming approaches, Thompson (1997) [60] and Infanger (1994) [46].

The parallel implementation in Birge *et al.* [10] is similar to the one discussed above. Their implementation is developed for a network of workstations, and tries to achieve good load balancing between the different processors by dividing the original problem as close to the root node as possible. The study looks at several implementational issues as applied to a broad range of problem instance from the POSTS [39] collection of multi-stage stochastic linear programmes. They conclude that their implementation achieves a good level of load balancing, greater

efficiencies when fewer processors are used, and that the use of the FFFB protocol in all stages of the divided problem is the preferred tree traversal strategy.

The algorithms presented in Thompson (1997) and Infanger (1994) [60, 46] differ in one key respect to the sub-tree algorithm described above. They allocate sub-problems to processors dynamically, as the algorithm proceeds, rather than statically at the start as is the other implementations. The reasons for dynamically allocating sub-problems is different for both of these implementations. The algorithm presented in Thompson (1997) [60] is implemented on a collection of platforms. The best results were from its implementation on a transputer array, where the communication overhead is very small. The algorithm in Infanger (1994) [46] incorporates importance sampling, see Chapter 4, but it is only applied to two-stage problems.

The parallel implementation of Benders Decomposition in Thompson allocates sub-problems to processors dynamically as the algorithm proceeds. The aim of this is to try to achieve good load balancing. This approach is called *farming*, as sub-problems are "farmed" out to slave processors as and when these processors become idle. Two queues are maintained at the master processor: one containing available slave processors and one consisting of sub-problems that need to be solved. The waiting tasks are sent out to the slave processors when they indicate that they are idle, while any new sub-problems that have to be solved are added to the queue. This algorithm is implemented for several parallel architectures and platforms, the best results being achieved with a transputer array.

One of the results in Thompson (1997) [60] is that any implementation is likely to be highly problem dependent. This was also found by Birge *et al.* (1996) [10]. The algorithm in Thompson (1997) [60] is ideally suited to multi-stage stochastic linear programmes that have large sub-problems, with more than a hundred rows and columns, and where the ratio of communication speed to processor speed is high. As there is a large number of messages being passed between the master and slave processors during this algorithm, the need for very fast communications is paramount. The main drawback of this algorithm is that any processor can be asked to solve any sub-problem from the original MSLP. This requires the entire data of the original problem to be stored on each slave processor. Hence, there are severe restrictions on the size of the multi-stage stochastic programmes that can be solved using such an algorithm.

In Infanger (1994) [46], a farming approach is used in the solution of two-stage stochastic linear programmes using Benders Decomposition and importance

sampling. As in the algorithm of Thompson, the master processor maintains two queues, one of sub-problems waiting to be solved, and another with idle slave processors. This algorithm is designed to solve two-stage stochastic linear programmes with a large number of second stage sub-problems, e.g. some problems have up to 10^6 scenarios. As the standard Benders Decomposition algorithm would require each of the sub-problems to be solved, it is necessary to incorporate sampling techniques within the Benders Decomposition algorithm. The algorithm solves a subset of the second stage sub-problems and uses their dual solutions to make statistical conjectures about the solution of the whole problem. More details of this algorithm, and the use of sampling techniques in conjunction with Benders Decomposition are examined in Chapter 4.

In this two-stage algorithm, a standard sub-problem is kept at each of the slave processors. When a processor is asked to solve a particular sub-problem, the realisation of the stochastic parameters for the scenario are passed to the slave processor, along with the solution to the first-stage sub-problem. This does not need to be re-sent to that processor until the next iteration of the algorithm. This sub-problem is solved and its dual solution is passed back to the master processor. The slave processor is added to the idle processor list, and the dual solution stored. Once all of the sample of second-stage sub-problems have been solved, the master process calculates the required statistics, and decides whether to terminate or to perform another iteration of the algorithm.

This two-stage parallel Benders Decomposition algorithm was tested on a capacity expansion problem WRPM. This problem has 10^6 second-stage sub-problems. Infanger presents results using different numbers of processors in the multicomputer and different sample sizes. He concludes that the larger the sample sizes, then the greater the efficiency obtained. As a large sample size is comparable to solving larger multi-stage stochastic linear programmes, this results tallies with those for Thompson's algorithm. In addition, the efficiency was found to reduce as the number of processors used increased, again corroborating Thompson's findings.

3.4.3 Results for Parallel Algorithm

The following results also appear in Archibald *et al.* (1999) [1]. This article compares the performance of four methods for solving multi-stage stochastic linear programmes: Revised Simplex, Benders Decomposition, Dynamic Programming and Aggregated Dynamic Programming. The principal aim of the article was

to investigate how Benders Decomposition performed compared to the other methods: whether it was ever inferior to the Revised Simplex method, if it suffered from the "curse of dimensionality" that usually limits DP based methods to small and medium scale problems, and how well it could be implemented on a parallel computer.

The solvers were tested on a collection of multi-stage stochastic linear programmes that model the management of electricity generation networks where a large proportion of the generation capacity is from hydro-electric reservoirs. The demand for electricity is met by energy produced from either thermal or hydro-electricity generation. The cost of the former depends upon the price of the fuel used, whereas the latter has no actual cost. Instead its value is due to its ability to offset the cost involved in using a thermal plant. The amount of hydro-electric power available in one time period depends on the use of water in previous time periods and the natural inflows of water to the reservoirs. In addition, the hydro-turbines have limited capacities. The amount of thermal power is limited by the plant capacity and availability. The decision in each time period is how much water to release through the turbines, and the related decision of how much water to retain in the reservoirs. Any demand for electricity unmet by hydro-generation is provided by the thermal plants in order of increasing unit cost.

The revised simplex method (RS) used was the implementation in OSL v1.2 [44], using the identity matrix as the initial basis. The simplex implementation in OSL v1.2 was also used as the sub-problem solver in our implementation of Benders Decomposition (BD). For the results in Tables 3.3 – 3.7, the only extensions to the basic Benders Decomposition algorithm used were hot-starts and the FFFB tree traversal strategy.

The dynamic programming (DP) implementation used both discretised water and action spaces. Only actions that resulted in the water state ending in one of the defined discrete water levels were considered. Therefore no interpolation between discrete levels was required. Further details of the DP implementation can be found in Archibald *et al.* (1999) [1]. The aggregate dynamic programming (ADP) algorithm reduces the computational burden of traditional DP methods. Hence ADP suffers less from the "curse of dimensionality" that limits the use of standard DP methods to problems with smaller state and action spaces. When using multi-stage stochastic linear programmes to model hydro-electric generation, this limits the size of networks, in terms of the number of reservoirs, that can be considered. ADP goes some way to overcoming this "curse of dimensionality" by considering each reservoir as one reservoir in a three reservoir network. This reduced network

consists of the reservoir under consideration, one aggregated reservoir consisting of all of the upstream reservoirs, i.e. those reservoirs whose released water can eventually flow into the reservoir under consideration, and one reservoir that is the aggregation of all of the remaining reservoirs in the system.

The ADP algorithm systematically solves aggregated problems of this form for all of the reservoirs in the original network. The results from these are then decoupled to provide policies for each reservoir. The benefits of ADP are that the computational burden, and hence the solution time, increases linearly with the number of reservoirs, so allowing large systems to be tackled. As the method effectively divides the problem into independent problems, the algorithm is suitable for implementation on a parallel computer. Further details of the aggregated dynamic programming method can be found in Archibald *et al.* (1997) [2].

All of the solution methods were implemented on a local area network of eight 70MHz Sun Sparc 5 workstations. One of the workstations had 48MB of memory, and the rest had 32MB. The serial results were obtained on the machine with the larger memory. The parallel implementations of BD, DP and ADP used Parallel Virtual Machine (PVM) version 3.1 [37]. By facilitating the communication of messages between processors, PVM allows a network of heterogeneous Unix workstations to be used as a parallel machine.

Problem	Sub-Problem Size	Det Equivalent Size	Expected Reward	RS & BD	DP	ADP
P3R	7x17, 7x23	10885x34211	19796383	0.00%	0.94%	1.25%
P4R	9x22, 9x30	13995x44578	19620562	0.00%	0.52%	0.74%
P8R	17x42, 17x58	26435x86046	18721904	0.00%	—	2.21%
P17R	35x87, 35x121	54425x179349	18691236	0.00%	—	3.13%

Table 3.3: Expected Rewards and Percentage Errors

The reservoir management model consists of three possible inflows patterns in each stage and has two underlying weather states. This gives six scenarios per stage, over a planning horizon of 4 stages. Therefore each problem has an event tree with 6 branches per stage and four stages. There is a total of 1555 sub-problems in the overall multi-stage stochastic linear programme. Table 3.3 contains the dimensions of the problems, their actual solutions and the accuracy of the four solution methods. There are four separate problems: P3R, P4R, P8R and P17R, which model networks with 3, 4, 8 and 17 reservoirs respectively. The first column details the size of the sub-problems in each problem. The first size is for the non-leaf sub-problems, and the second for the leaf sub-problems. The

final stage sub-problems are larger as the model has a piecewise linear function to model the terminal value of any water remaining in the reservoirs. The size of the deterministic equivalent linear programme is given in the next column. This is the problem that the RS method solves. The final four columns give the expected reward for each problem and the percentage error in the solutions provided by the four methods. The RS and BD methods give exact solutions, to within numerical rounding, where the BD algorithm was solved using a relative tolerance of 10^{-6} in the objective value.

The DP method used a discretisation of the state and action spaces, and hence an exact solution to the original problem cannot be obtained from this relaxation. To obtain a solution that is closer to the actual solution, it is necessary to make the discretisations of the state spaces finer, and so make the solution more computationally complex. The solutions to the ADP method are obtained by taking its proposed policies and using a simulator to perform top-down corrections to produce feasible solutions. The final column of Table 3.3 shows that, in general, the accuracy of the ADP method decreases as the number of reservoirs increases, i.e. as problem size increases, but it is able to tackle all of the problems, unlike the standard DP method, which could only solve the two smaller problems, P3R and P4R.

Example	RS	BD	DP	ADP
P3R	7214.63	94.98	173.24	15.67
P4R	11505.23	106.97	15214.62	524.13
P8R	38709.65	764.93	—	1986.30
P17R	181509.60	2254.46	—	11177.98

Table 3.4: CPU Time(s) on a 70MHz SUN Sparc 5

Solutions using the serial implementations of the four methods on the workstation with 48MB of memory are given in Table 3.4. These show that BD is the fastest of the four methods for all but the smallest test problem, with ADP being the second fastest method for all but this smallest problem. The results show that only methods that take advantage of the structure of the problem are practical options for large problems, as shown by the performance of RS and DP.

All but the RS method were implemented in parallel, and the results from their execution on a network of up to eight workstations are shown in Table 3.5. The results show that the parallel implementation of BD is efficient; the speedups being near-linear. For all but one of the problems the speedup was super-linear. This is primarily due to the amount of paging that affected the serial implementation of BD. In the parallel implementation of BD the multi-

Example	RS	BD	DP	ADP
P3R	—	6.08× on 6	2.33× on 3	2.15× on 3
P4R	—	6.01× on 6	2.54× on 3	2.68× on 4
P8R	—	5.94× on 6	—	3.29× on 8
P17R	—	7.47× on 6	—	5.55× on 8

Table 3.5: Speedups on a Network of 70MHz SUN Sparc 5

stage stochastic linear programme is split almost equally between the available processors. Hence the size of the problems at each of the processors is roughly one sixth of the size of the problem that the serial implementation solves. Therefore the amount of paging during the parallel solution is minimal. This is evident in the results for problem P17R, the largest problem. However, for all of the problems the speedup was close to linear, and the amount of paging for the smallest problem was small during its serial solution. In general, both of the DP based methods parallelise well, with efficiencies being over 67% for all but one the problems.

Example	ELP	CPU	Cuts	Splx Its	BD Its	Sub-problems
P3R	109.05	94.98	828	20231	5	8599
P4R	108.24	106.97	867	11309	6	10192
P8R	851.32	764.93	5180	99819	35	59571
P17R	3286.40	2254.46	10578	315165	66	113143

Table 3.6: Details of Serial Solutions

Example	ELP	CPU	Cuts	Splx Its	BD Its	Sub-problems
P3R	17.95	89.14	874	15696	5	8645
P4R	18.01	90.39	790	11117	5	8561
P8R	143.29	700.93	4736	84313	33	56019
P17R	439.88	2042.15	10673	322630	62	107022

Table 3.7: Details of Parallel Solutions

Tables 3.6 and 3.7 give more detailed accounts of the serial and parallel BD implementation respectively. The first two columns record the elapsed and CPU times for the solves, the next two columns contain the total number of cuts added and the total number of simplex iterations performed, the next column records the number of iterations of the BD algorithm, and the final columns show the number of sub-problems that were solved in total. There are two points to note from the comparison of the figures in these two tables. Firstly, the CPU time for both methods differs. This is due to the parallel algorithm being able to follow a different solution path from the serial algorithm. Some of the sub-problems are degenerate, i.e. they have more than one optimal solution, and the

solution obtained depends on the starting basis, which is different in the parallel case. There are also large differences in the solution times for the three largest problems, P4R, P8R and P17R. This is mainly due to the difference in the number of passes required rather than in the time needed to solve the sub-problems.

3.5 Cut sharing & Graph Notation

In this section we discuss the motivation behind the folding of multi-stage stochastic linear programmes and how our implementation of Benders Decomposition is able to solve folded problems. Finally, we discuss the unpromising results for the test problems derived from our hydro-electricity management model.

3.5.1 Motivation & Implementation

Models of real-world planning problems can lead to very large multi-stage stochastic linear programmes. This is because these models grow exponentially in the number of stages. Such problems can push extended Benders Decomposition solvers to their limits. However, they often fail to exploit any further structure due to the form of their stochastic parameters.

Stochastic parameters that depend upon only the previous stage are common to many real-world problems that can be modelled as multi-stage stochastic linear programmes. This property is readily exploited either through the use of cut sharing, as discussed in Section 3.3.2, or by modelling the problem as a set of recurrences. This formulation results in a model that is represented by a graph structure, rather than the event tree structure of general multi-stage stochastic linear programmes.

The benefits of formulating a model in this way are that the Benders Decomposition method can easily be adapted to solve problems of this form. In such folded models, cut sharing, as discussed above, does not need to re-calculate or repeat cuts that are shared across a stage. There are two reasons why this might be useful. Cuts are shared automatically across stages, rather than being repeated. This results in a smaller number of cuts being added to the problem in general. In addition, as the nodes themselves are not repeated across stages, the problems are significantly smaller than their "unfolded" equivalents. The second benefit is that very large problems that can be folded are able to be solved using Benders Decomposition with sampling techniques. This is because Benders Decomposition with sampling techniques relies upon the ability to share future cost information across stages.

An example of a problem which exhibits Markovity is the management of hydro-electric reservoirs. The stochastic parameters in this problem are the inflows to the reservoirs in each time period. A range of these inflows is modelled and used to

enumerate the possible scenarios in each time period. The problem is Markovian because the range of inflows in one period does not depend upon which scenario occurred in the previous period. Only the transition probability will depend upon the scenario in the previous stage. Consider a four stage problem, with three scenarios per stage, that has history independent stochastic parameters. This problem is represented by the event tree given in Figure 3.5.

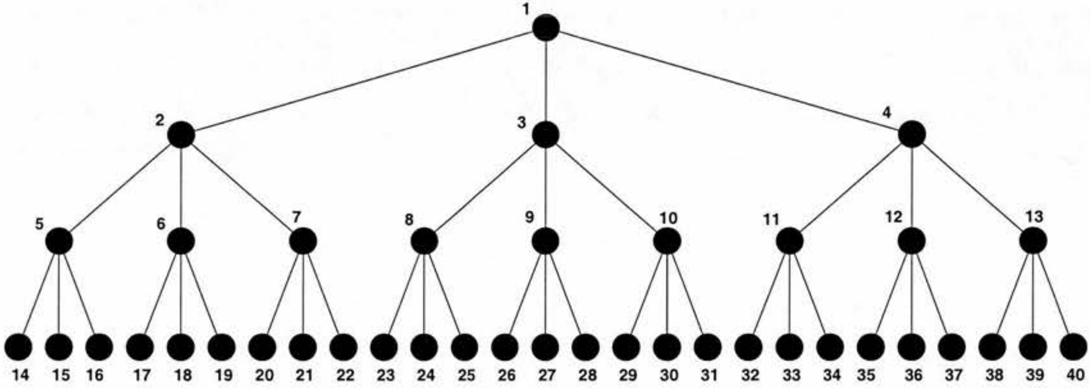


Figure 3.5: Unfolded Event Tree for Reservoir Problem

Here the sub-problems represented by nodes 5, 8 & 11 are the same linear programme, corresponding to one of the three inflow patterns that can occur in the third stage. The same can be said for the sets of nodes 6, 9 & 12, and 7, 10 & 13. This duplication can be eliminated by formulating the problem as a set of recurrences such as those given in equations (3.14–3.16).

$$\begin{aligned} \text{minimise} \quad & c_1 x_1 + \mathbb{E}_2 Q_2(x_1) \\ \text{subject to} \quad & A_1 x_1 = b_1 \\ & l_1 \leq x_1 \leq u_1, \; x_1 \in \mathbb{R}^{n_1} \end{aligned} \tag{3.14}$$

with the functions Q_t, $t = 4, \ldots, T-1$, being defined as

$$\begin{aligned} Q_t(x_{t-1}) = \min \quad & (\mathbf{c_t} x_t + \mathbb{E}_{t+1} Q_{t+1}(x_t)) \\ \text{subject to} \quad & \mathbf{E_t} x_{t-1} + \mathbf{A_t} x_t = \mathbf{b_t} \\ & \mathbf{l_t} \leq x_t \leq \mathbf{u_t}, \; x_t \in \mathbb{R}^{n_t} \end{aligned} \tag{3.15}$$

and Q_T being defined by

$$\begin{aligned} Q_T(x_{T-1}) = \min \quad & (\mathbf{c_T} x_T) \\ \text{subject to} \quad & \mathbf{E_T} x_{T-1} + \mathbf{A_T} x_T = \mathbf{b_T} \\ & \mathbf{l_T} \leq x_T \leq \mathbf{u_T}, \; x_T \in I\!R^{n_T}. \end{aligned} \tag{3.16}$$

Here the operator $I\!E_t$ denotes the expectation with respect to the tth-stage random variables and any of the **bold-face** entries may be random. The functions Q_t, $t = 2, \ldots, T$ are the recourse functions for stages $t = 1, \ldots, T-1$. They represent the costs incurred in the following stages, which depend upon the decisions made in the current stage. For the reservoir example represented by Figure 3.5, the distribution of the stochastic parameters is discrete, with three scenarios per stage, and only the right hand sides, $\mathbf{b_t}$, are random. In this case the expectation operator can be replaced with the weighted summation of the recourse functions.

The problem described by the recurrences (3.14–3.16) generates a problem in which the range of scenarios in one period does not depend upon those in the previous stage. Hence it is possible to have arrived at a particular scenario in a stage having come from any of the scenarios in the previous stage. The only difference between these transitions is the probability of them occurring. Problems of this form do not have an underlying event tree structure, but instead that of a graph. The graph corresponding to the event tree in Figure 3.5 is given in Figure 3.6.

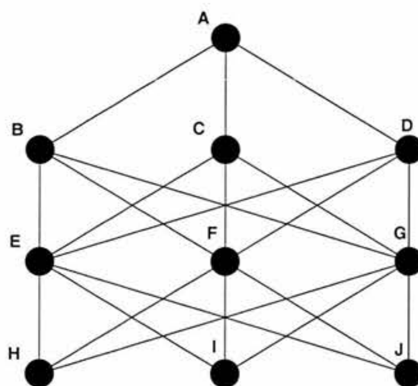


Figure 3.6: Folded Event Tree for Reservoir Problem

In the graph, node E represents the three third stage nodes 5, 8 & 11 from the unfolded tree of the same problem given in Figure 3.5. The three scenarios E, F & G in the graph are repeated three times when the problem is "unfolded" to give the event tree formulation in Figure 3.5. The arcs from nodes B, C & D to

node E correspond to the three transitions that can arrive at node E in the third stage. In the event tree they correspond to the transitions between nodes 2 & 5, 3 & 8 and 4 & 11. In this way a sub-problem no longer has a unique parent, but instead can have several parents in the previous stage. For this reason, it is necessary to label a decision with the parent from which it came, so that the cuts that are formed by propagating this decision forwards through the problem can be returned to this parent.

By "folding" the event tree of the original problem to generate the graph results in the number of nodes growing linearly in the number of stages, rather than exponentially. This allows problems that are ordinarily too large to tackle to be solved if they exhibit Markovity in their stochastic parameters. In the example above, as the number of nodes in the folded problem is a quarter of the number in the unfolded tree, there is a four fold decrease in the memory requirements for the four stage problem. As the number of stages increases, this saving in memory requirements grows exponentially. For example, for a thirteen stage problem with three scenarios per non-root stage, the decrease in size is 21,000 fold.

While the size of folded problems is significantly smaller than their unfolded equivalents, there is unlikely to be less work involved in solving the folded problem than is required to solve the unfolded problem. The folded problem represents the same multi-stage stochastic linear programme as the unfolded problem. Hence there are the same number of paths through both the graph and the tree representations of the problem. Here a path represents a unique sequence of realisations of the stochastic parameters from the root node to a leaf node.

The same number of sub-problems are solved during the first pass of the algorithm when solving either representation of the problem. This may not be the case in later iterations as the cuts that have been added to sub-problems will differ and so the need to explore different parts of the problem will differ. Therefore the significant savings in the amount of sub-problems, and so memory, required to represent the problem as a folded graph does not necessarily translate to savings of the same order in the effort, in the number of sub-problems solved, required by the algorithm.

Solving the folded representation of a problem introduces two additional factors which affect the work and time involved in the solution of the problem. The first of these is the benefit due to the sharing of cuts. After several iterations of the algorithm, sub-problems in the later stages of the graph will contain cuts

that where generated by questions from more than one of the parents of this sub-problem. As the questions from the parents may have been very different, the cuts generated by these questions may approximate very different parts of the recourse function of this sub-problem. Hence when the sub-problem is asked further questions by sub-problems in the previous stage, the approximation to the recourse function contains more information than if the problem had not been folded.

For example, in Figure 3.6 cuts added to node F can result from questions asked by nodes B, C or D. In future iterations all of the previous cuts are available regardless of which parent sends a decision to node F. When solving the unfolded problem, as given in Figure 3.5, the cuts generated by decisions from nodes 2, 3 & 4 are added to nodes 6, 9 & 12 respectively. In future iterations, the solution of node 12 only takes into account the cuts previously generated by decisions from node 4, despite node 4 being a repetition of nodes 6 & 9. In this way folding problems leads to a natural sharing of information across stages.

The second factor is also due to decisions from different parents causing cuts to be generated at different parts of the value surface. When a sub-problem is solved, apart from the first time, the previous optimal basis is used as an initial basis. When an unfolded problem is solved, the only difference between a sub-problem before it is re-solved and after it was last solved will be at most the addition of a cut, or perhaps a few cuts if multi-cuts is used. Hence the previous optimal basis is likely to be a good initial basis for the solution of the slightly modified sub-problem. In addition, the decision that requires the sub-problem to be re-solved has originated from the same parent as the previous decisions passed to the sub-problem.

When a folded problem is solved the use of the previous optimal basis may not result in decreases in the number of simplex iterations taken to solve a sub-problem. As the previous solution of the sub-problem may have been due to a decision from another parent to the one currently passing a decisions to the sub-problem, the previous optimal basis need not be a good candidate as an initial basis for the current decision. This is because decisions from different parents might require a sub-problem to consider different regions of its recourse function. Instead extra iterations may be necessary to travel from an area of the recourse function to another. This affect will be more pronounced at later stages in the problem as the paths from the root node have more opportunity to diverge with each additional stage.

In order to allow folded problems to be solved by Benders Decomposition method, it is only necessary to alter the way in which decisions are passed forwards and cuts backwards through the problem. When solving a folded problem a sub-problem need not have a unique parent and so a decision must be identified as having come from a particular parent.

As part of our parallel implementation of Benders Decomposition it was necessary to identify from which node any decision had come, as the root nodes of the sub-trees on each processor receive decisions from nodes on other processors. This method of labelling a decision as having come from a particular node can be extended to the serial case where folded problems are under consideration. As well as labelling each decision, the cuts generated by these decisions are also labelled, and so can be identified with the parent which generated them.

Beyond the third stage of a folded problem it is possible for a sub-problem to have more than one ancestor in stages before just the previous one. Hence it is necessary to tag decisions and cuts by their entire ancestry back to the first stage. This could result in the full enumeration of the paths through the equivalent unfolded problem. However, it is possible to enumerate decisions and cuts using only the same number of tags as there are nodes in the folded problem. This is especially necessary when sampling is incorporated with Benders Decomposition to allow the solution of very large multi-stage stochastic linear programmes as these can be large enough to make the unfolded tree too large to enumerate.

3.5.2 Results

Our initial tests with the solution of the four and six stage MSLPs described in Section 3.3.6 gave much longer solution times for the folded versions of these problems. In order to investigate whether this was due to having cuts from decisions sent from different parents at sub-problems, we ran two sets of experiments.

Firstly we looked at how the solutions of the equivalent folded and unfolded four stage problem compared when the problems were solved to different levels of optimality. The results for these runs are given in Table 3.8 where the two versions of the problem were solved for stopping tolerances of 10^{-2} - 10^{-6}.

The results show that only when the problems were solved to the two largest, i.e. widest, stopping tolerances was the folded problem easier to solve. For all of the other, tighter tolerances, the solution to the unfolded problem is significantly

		Stopping Tolerance				
		10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
Unfolded						
	Passes	31	48	74	94	118
	Cuts	1253	1830	2606	3062	3458
	Splx Its	163387	204332	252501	287014	311135
	CPU (s)	96.57	127.60	176.55	206.27	248.30
Folded						
	Passes	14	33	53	80	99
	Cuts	468	1125	1834	2552	2853
	Splx Its	80503	172278	273880	406950	498756
	CPU (s)	48.29	127.25	248.74	451.13	593.65

Table 3.8: Unfolded vs Folded 4 Stage Problem

quicker than the solution of the folded problem. The number of simplex iterations performed during the solution of the sub-problems is similarly smaller for the solution of the folded problem compared to the unfolded problem using the two weaker stopping tolerances and larger for the other stopping tolerances.

In comparison, the total number of cuts added to the folded problem is smaller than the number added to the unfolded problem for all of the tolerances. These results in Table 3.8 suggest that both of the affects described previously are occurring. The number of cuts required during the solution of the folded problem is less than for the solution of the unfolded problem because the folded problem is able to share information across stages. However, for the tighter stopping tolerances the number of simplex iterations performed during the solution of the problem is greater for the folded problem, suggesting that many extra iterations are required due to the use of previous optimal bases when solving the folded problem.

In order to corroborate these conclusions we compared the unfolded and folded results for a modified version of the four stage problem. The new problem was constructed to have the one sub-problem repeated across each stage. This would provide a way of investigating what happens if the affect of using the previous optimal basis to solve a sub-problem afresh is removed. As each sub-problem across a stage is the same, the previous optimal basis will be a good candidate when any sub-problem is solved the next time. The results for these runs are given in Table 3.9

		Stopping Tolerance				
		10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
Unfolded						
	Passes	23	40	41	52	55
	Cuts	943	1439	1492	1694	1718
	Splx Its	126047	162893	166952	186535	189018
	CPU (s)	72.55	101.31	106.43	120.15	124.83
Folded						
	Passes	10	18	22	20	21
	Cuts	361	569	630	559	560
	Its	41522	57839	69101	62780	64552
	CPU (s)	28.92	46.16	55.82	50.00	51.75

Table 3.9: Unfolded vs Folded New 4 Stage Problem

The results in Table 3.9 show that the solution of the folded version of the modified four stage problem is quicker than the solution of the unfolded version for all stopping tolerances. In addition to the solution of the folded version folded of the new four stage problem generating fewer cuts, as was the case for the previous four stage problem, the folded version of the new four stage problem also requires far fewer simplex iterations.

This agrees with the conclusion above that the original four stage problem takes longer to solve for smaller stopping tolerances because the previous optimal bases are not good candidates for initial bases when solving a sub-problem for a decision from a different parent to the one which resulted in the previous solution of the sub-problem.

One way to overcome this problem would be to store the optimal basis for each sub-problem and tag it in a similar manner to the method used to monitor the ancestry of decisions and cuts when solving folded multi-stage stochastic linear programmes. However, this would require one basis to be stored for every node in the equivalent unfolded tree. This would increase the amount of storage required, but might reduce the number of simplex iterations taken to solve sub-problems which had previously been solved.

3.6 Conclusions

Multi-stage stochastic linear programmes are linear programmes whose constraint matrices are very sparse and which have a primal block angular structure. Benders Decomposition is an iterative solution method that exploits this primal block angular structure. The method decomposes the multi-stage stochastic linear programme into much smaller sub-problems. These sub-problems are solved iteratively, and the primal and dual solutions are passed between the sub-problems.

The method was originally formulated for the solution of mixed integer programmes, but has been extended to allow the solution of multi-stage stochastic linear programmes. By exploiting the block angular structure of the deterministic equivalent to a MSLP, Benders Decomposition is able to solve much larger problems than direct methods, such as the simplex algorithm, are able to solve and does so is far less time.

The performance of the basic Benders Decomposition algorithm can be enhanced by the use of various extensions. These include the re-use of previous optimal bases, the use of multi-cuts, cut sharing, different tree traversal strategies and preliminary cuts. In general the use of multi-cuts and previous optimal bases leads to improvements in solution times. The benefits of using the other extensions are more problem dependent.

As a large number of independent sub-problems are solved during on iteration of the Benders Decomposition method, the algorithm can be implemented very effectively on a parallel computer. Our implementation on a network of workstations achieves near linear speedups. It divided the multi-stage stochastic linear programme into large sub-trees, and solves these on separate processors. This way of implementing the algorithm is suited to a network of workstations as it does not generate a large amount of communications between processors.

In addition to the block angular structure of multi-stage stochastic linear programmes, many problems also have history independent stochastic parameters. This allows the event tree that represents the multi-stage stochastic linear programme to be folded to give a graph of sub-problems. The folded version of a MSLP grows linearly in the number of stages, whereas the unfolded version grows exponentially. Consequently the folded version is a far more compact representation of the original problem.

The reduction in size by folding multi-stage stochastic linear programmes which have stochastic parameters that are independent between stages allows much larger problems to be solved using Benders Decomposition. When solving folded problems, far fewer cuts are generated as this representation of a MSLP leads to the automatic sharing of cuts across each stage. However, the solution times required to solve folded problems to within tight stopping tolerances are much longer than when solving the same problems in their unfolded representation. This is due the conflict of using previous optimal bases. But as will be seen in the following chapter, combining folded problems and a Benders Decomposition algorithm which incorporates importance sampling allows extremely large multi-stage stochastic linear programmes to be be solved.

Chapter 4

Sampling Techniques & Benders Decomposition

4.1 Introduction

The mathematical models of real-world problems can become very large. Despite increasing performance of desktop computers, such large-scale problems remain intractable by direct and decomposition based algorithms. To tackle this problem a decision maker can decide to do one of two things. Either re-model the problem using a smaller, less detailed model. This would be more tractable, and so an exact solution to this model of the problem could be obtained by using the currently available solvers. However, this exact solution would be of an inaccurate model, and so need not be an accurate solution to the original problem. The second option open to a decision maker is to try to approximate the answer to the large-scale problem, thus obtaining an approximate answer to a detailed model of the original real-world problem.

One of the most common ways to find an approximate answer to a mathematical model is to use sampling. The main characteristics of multi-stage stochastic linear programmes that make them amenable to sampling is that they can be thought of as a large collection of future scenarios that occur through time, with each scenario having a cost and a probability attached to it. Sampling from such a problem involves selecting a sub-set of the possible scenarios and solving for these scenarios. An estimate of the actual solution to the original problem can be found by taking the mean of the solutions obtained for each of the samples taken.

The introduction of sampling techniques into Benders Decomposition based algorithms can be done in several ways. Regardless of which method is used, further considerations need to be taken into account. These include questions

about how many samples should be taken and what faith can be given to the resulting estimates of the original problem's solution. In the following section we consider the two broad ways in which sampling can be introduced into Benders Decomposition based algorithms. After this we consider the use of Monte Carlo sampling and various variance reduction techniques. Finally we discuss some suggested stopping rules for sampling based algorithms.

4.2 Using Sampling Techniques

The idea of incorporating sampling methods when solving multi-stage stochastic linear programmes has been around almost as long as the notion of multi-stage stochastic linear programmes themselves, see Dantzig & Madansky (1961) [20]. The multi-stage stochastic linear programmes formulated to model real-world problems soon become intractable as their size grows exponentially in the number of stages. Sampling techniques used in conjunction with existing solution algorithms hope to obtain a reasonable estimate to the actual solution of an accurate model by sampling a small selection of the possible future scenarios. If sampling techniques are not used, and instead a decision maker is limited to existing algorithms, then the alternatives are to use bounding techniques or to formulate a smaller and less accurate model.

Neither of these alternatives is particularly appealing. The first attempts to obtain bounds on the solution to a large-scale model. It does this, for minimisation problems, by discretising the support of the stochastic parameters and uses the Jensen and Edmunson-Madansky inequalities to obtain lower and upper bounds on the actual solution to the problem. The lower bound is obtained from Jensen's inequality which states that the expectation of a convex function is always greater than or equal to the value of the convex function at its expectation, i.e. the solution of the expected value problem. Here the function is being minimised is the objective function of the original problem. The upper bound of a convex function can be obtained by replacing the distribution by the end points of its support and taking a linear combination of the function value at these end points. This is the Edmunson-Madansky inequality. Both of these bounds can be improved by dividing the support of the convex function into disjoint parts, and calculating the bounds over these discrete parts of the distribution. How to decompose the support is a difficult question to address in itself.

The other option open to a decision maker is to model the real-world problem in less detail, and so generate a smaller problem. This can be solved using a standard algorithm, without the need to incorporate sampling techniques. The main disadvantage of this option is that while accurate results for this model can be obtained, they are only the exact solution of an inaccurate model.

There are two distinct ways of incorporating sampling techniques into solution algorithms. The first, called *external sampling*, selects a sample of possible futures before the algorithm is used to solve any programmes. The sample is used to generate a multi-stage stochastic linear programme that is an approximation of

the original problem, but which is smaller and hence easier and quicker to solve. The approximation is solved using a suitable algorithm, and the solution used to form an estimate of the original problem's solution. This process may be repeated with a fresh sample and any new solutions can be used to improve the estimate to the solution to the original programme.

The second way of using sampling techniques in conjunction with solution algorithms for multi-stage stochastic linear programmes is called *internal sampling*. Here the sampling is performed within the algorithm. At each iteration of the algorithm a new sample is selected. Instead of obtaining exact bounds in each iteration, estimates of these bounds are obtained. These are then used to select the samples in the next iteration.

4.2.1 External Sampling Techniques

Two different examples of external sampling techniques can be found in Morton *et al.* (1999) [51], where samples from the realisations of the stochastic parameters of the original problem are used to generate approximating programmes, and in Dempster & Thompson (1996) [23], where a relaxed version of the original problem is solved and the EVPI at the sub-problems is used to redefine the relaxation.

The method of Morton *et al.* is to generate a sequence of approximations to a two-stage stochastic linear programme by sampling over the possible realisations of the stochastic parameters. The solutions of these approximating programmes are used to form probabilistic lower bounds on the original problem's solution. The method is independent of the algorithm used to solve the approximating programmes. This method has not yet been extended to the solution of multi-stage stochastic linear programmes. The theory of epi-convergence, see Dupačová & Wets (1988) [24], under-pins this method. The main result from this theory is that a sequence of solutions to sampled approximations of stochastic linear programmes contain an accumulation point that is the solution to the original problem, with probability one. In addition to the basic method, the authors also discuss various techniques for variance reduction. These are discussed in the following section.

The EVPI-sampling algorithms of Corvera-Poiré and Dempster [19, 23, 16] use the EVPI of sub-problems within an approximation to a multi-stage stochastic linear programming to iteratively refine the approximation, and so to obtain an improved estimate of the original programme's solution. EVPI is the *expected*

value of perfect information, and is a measure of how much a decision maker would be willing to pay to know which future scenario occurred, rather than trying to hedge against all possible futures. At each iteration of the algorithm, an approximation of the original programme that is more representative of the original programme is solved using Benders Decomposition and the EVPI calculated at each sub-problem of the approximation.

Consider the original multi-stage stochastic linear programme solved to optimality by Benders Decomposition. If the EVPI is calculated at each sub-problem, then this can be used as a guide to how "stochastic" each sub-tree rooted at any non-leaf sub-problem is. If a sub-problem has a very low EVPI, then the sub-tree rooted at this sub-problem does not vary considerably from the expected value problem for the sub-tree. The *expected value problem* is the problem where each stage consists of a sub-problem with the expected value of the stochastic parameters across that stage in the sub-tree. Therefore it is possible to collapse a sub-tree with low EVPI at its root down to the expected value problem and not loose much accuracy in the solution. If the converse is true, and a sub-problem has a high EVPI, then it is desirable to consider the full sub-tree rooted at this sub-problem. The solution to this sub-tree differs greatly from the solution to the expected value problem for the same sub-tree, and so is necessary to obtain the true solution to the original programme.

EVPI-sampling uses the EVPI to decide how to update the approximation to the original programme. If a sub-problem has a high EVPI, then the sub-tree rooted at that node is represented in greater detail than if the EVPI is low, when the sub-tree can be replaced by either the expected value problem, or by one path through the sub-tree, sampled at random. The benefits of solving a large-scale multi-stage stochastic linear programme by EVPI-sampling are likely to be hampered by the need to calculate the EVPI for each sub-problem of the current approximation. These calculations are time consuming, especially if the approximation is similar in size to the original programme. This occurs if the original problem has a large EVPI, i.e. it is particularly stochastic.

4.2.2 Internal Sampling Techniques

The difference between the examples of external sampling discussed above, and internal sampling techniques is that internal sampling techniques use sampling within an algorithm that can be used to solve multi-stage stochastic linear programmes. Because all of the original programme is not solved by the algorithm

at any one time, much larger problems can be tackled than if the original algorithm had been used.

Three types of internal sampling techniques are stochastic quasi-gradient methods, Gaivoronski (1988) [32], stochastic decomposition, Higle & Sen (1996) [38] and Monte Carlo extensions to Benders Decomposition method, Infanger (1994) [46] and Pereira & Pinto (1991) [55].

Quasi-gradient methods solve mathematical programmes by performing a sequence of descent steps. These involve moving a certain distance from the current solution along a line of descent calculated at the current solution. A subgradient is used as the direction of descent at each step. By definition, a subgradient is a non-zero vector that can be used to generate a supporting hyperplane at a particular point. If the function being minimised is convex, then the subgradient is unique, and is the gradient of the function at that point, otherwise the subgradients form a convex set. For particularly large problems, calculating the gradient at a sequence of points is a cumbersome task, and so a partial solution is used to generate a subgradient. Depending upon the step length, this ensures that the updated solution is an improvement. In the stochastic case, a sample of realisations of the stochastic parameters is used to generate subgradients. The problems generated by the sample of the stochastic parameters generate linear programmes, from which the subgradients are calculated. These are then averaged across the sample, and used as the descent direction. The convergence rates of stochastic quasi-gradient methods are slow. In addition the choice of step sizes and the subgradients are very problem dependent.

Stochastic Decomposition is a method of incorporating sampling and partial optimisation techniques within Benders Decomposition developed by Higle & Sen (1996) [38]. It can be seen, as can the following two internal sampling methods, as being an incorporation of sampling techniques within a cutting plane method. The algorithm has only been developed for two-stage stochastic linear programmes that have their stochasticity confined to the right-hand sides.

At each iteration of the Stochastic Decomposition algorithm a new second-stage scenario is randomly generated and the sub-problem associated with that scenario is solved to optimality. This optimal dual solution of the sub-problem is added to a collection of second-stage dual solutions. Each of the previously sampled second-stage sub-problems is then solved using the collection of dual solutions as a dual feasible region. As this is a restricted dual feasible region, most of these

solutions are not necessarily optimal, but as the collection of second-stage dual solutions increases, the solution tends towards their optimal solutions.

The solution of each of the sampled second-stage sub-problems is used to generate an estimated cut that is added to the first-stage problem. Existing cuts in the first-stage problem are updated. This is done by re-weighting the gradients as the sample mean of the now increased sample size, using the current second-stage sub-problem that has just been solved to optimality. The right-hand sides are re-weighted with an extreme lower bound. This results in older cuts, that are likely to be less accurate, being less effective as the algorithm proceeds.

The feasible dual solutions of the second-stage sub-problems are the same for each sub-problem, as the stochastic parameters are restricted to the right-hand sides for Stochastic Decomposition. As this set is finite, the collection of feasible dual solutions grow towards the actual dual feasible region. Hence, sub-problems that are solved by the partial optimisation at later iterations of the algorithm are likely to be closer to their actual solutions. Optimising over the set of dual feasible solutions is a relatively cheap operation, as compared to solving a second-stage sub-problem. This saving in effort is magnified as the algorithm proceeds and it is necessary to re-solve all of the sub-problems sampled in previous iterations to update the cuts that they generated.

There are two source of error in the cuts. Firstly they are estimates generated from the sampled second-stage sub-problems. Hence it is possible that the cuts are not necessarily lower bounds on the recourse function at the point about which they are linear approximations, and instead might cut off part of the feasible region. The second source of error is that when the cuts are updated, the re-solved sub-problems are only partially optimised, using the set of feasible dual solutions. While this results in more approximate cuts, any cut generated in this manner is still an approximation to a supporting hyperplane. As the algorithm proceeds the cuts become more stable, as larger sample sizes are used, and also as the collection of feasible second-stage dual solutions increases. For this reason, when the cuts from the previous iterations are updated, they are weighted with an extreme lower bound, and so become less important as they drop down to becoming strict under approximations of the recourse function.

The final internal sampling technique involves the inclusion of Monte Carlo based sampling methods within Benders Decomposition. Two distinct ways of doing this have been proposed, one for two-stage stochastic linear programmes that have very many second stage sub-problems, and one that is applicable to multi-

stage stochastic linear programmes with a "manageable" number of scenarios across each stage.

The two-stage method developed by Infanger solves a sample of second-stage sub-problems in each iteration, and uses estimated cuts to update the first-stage sub-problem. In each iteration, the master problem is solved, and a sample of second-stage sub-problems are solved given this first stage decision. The sample mean of the solutions to the sampled second-stage sub-problems provides an estimate of the upper bound on the second-stage costs, and hence provides an upper bound on the whole problem, with attached variance.

The dual solutions to the second-stage sub-problems are used to generate cuts that are passed backwards to the root problem. Instead of weighting these cuts according to the probabilities of the sub-problems, their sample mean is taken, and so an estimated cut is passed back. This cut is an estimate of the supporting hyperplane about the point defined by the first stage decision that has generated this cut. As the cut is an estimate it does not have to be a supporting hyperplane, and can actually over-estimate the recourse function at the point about which it is generated, hence cutting off part of the feasible region. Infanger shows that the error in the height of the cut at this point can be taken to have zero mean and variance equal to that of the second-stage costs. This result is extended to show that it is a fair approximation to assume that the error in the height at any point along the cut is approximately the same as at the point around which it is an estimated linear approximation.

Once the second-stage sub-problems have been sampled, and upper and lower bounds computed, these bounds can be used to test whether or not to stop. As the two bounds are not exact, but instead are sample means with associated variances, their distributions must be compared using the Student t-test. A confidence interval can be generated using the left side from the distribution of the lower bound, and on the right side from that of the upper bound. The algorithm stops if the size of this interval, as a ratio of the lower bound, is small enough. Otherwise the sample size is increased and the next iteration performed.

Infanger notes that the distributions used in a Student t-test need to be independent. This need not be the case for the upper and lower bound estimates, as the upper bound might be associated with one of the binding cuts that gave the lower bound. Therefore it is necessary to re-sample for one of the distributions. Instead of doing this after every sample has been taken, Infanger suggests that the estimated upper and lower bounds are tested for convergence. If they are

close enough, then the upper bound is re-sampled, and checked again, with the algorithm stopping if this second test is successful. If the optimality test does not stop the algorithm, a new sample is taken and the algorithm performs a new iteration. The binding cuts from the previous iteration can be re-used in the next iteration. For each binding cut, the first-stage decision that originally generated the cut is used in each of the new sampled sub-problems. These are then solved, and the binding cuts re-computed. The new iteration is then performed. The first-stage problem is solved, and the newly sampled sub-problems are solved to generate new cuts and bound estimates.

Infanger also outlines how this method can be extended to the multi-stage case. Instead of sampling second stage sub-problems, the sample at each iteration now consists of paths through the entire problem, each path consisting of a node in each stage of the overall problem. The costs associated with each path are averaged to provide an estimate of an upper bound on the original problem. Once the estimated cuts have been added to the non-leaf nodes, the solution of the first stage sub-problem is used as an estimate of the lower bound. As in the two-stage case, the binding cuts added need to be re-sampled at subsequent iterations, so as to ensure that they do not restrict the feasible region.

In the multi-stage case there are now two sources of error in the lower bound estimates. The first is due to the use of estimated cuts, as in the two-stage case. The second source of error is due to the fact that the cuts added to all but the nodes in the penultimate stage are generated by sub-problems that contain estimated cuts themselves. Hence the error due to the use of estimated cuts accumulates as cuts are passed backwards through the problem.

An alternative internal sampling method was proposed by Pereira & Pinto (1991) [55] for multi-stage stochastic linear programmes that have many stages and a limited number of sub-problems per stage. At each iteration a number of paths through the problem are sampled. These are then solved in two passes. During a forward pass the nodes on each path are solved in turn, their decisions being passed forwards down the path. On the backwards pass, the siblings of the node on the path in each stage are solved using the solution of their parent. The cuts generated are passed backwards to the node on the path in the previous stage.

In this way, the cuts added to the non-leaf nodes are valid lower approximations to the recourse function of these nodes. Because all but one of a node's children have been solved using only the current approximations to their recourse functions, the cuts generated are unlikely to be supporting hyperplanes. However, because all of

the cuts added are valid, the solution of the first stage sub-problem is a genuine lower bound on the solution of the multi-stage stochastic linear programme. The sample mean of the cost associated with each path provides an estimate of an upper bound on the problem. Pereira & Pinto propose that the variance in this estimate is used to construct a 95% confidence interval around the upper bound, and a Student t-test used to check for optimality.

4.2.3 Issues introduced by Sampling

When using sampling techniques extra considerations are required to ensure the successful estimation of the solution to the problem under consideration. These can be posed as the questions: How many samples should be taken? and, What can one say about the accuracy of the final answer obtained?

During the execution of Benders Decomposition based algorithms that do not incorporate sampling techniques, upper and lower bounds on the optimal solution are known at each iteration. In addition, these bounds are also guaranteed to improve monotonically as the algorithm progresses. This means that a multi-stage stochastic linear programme may be solved to any tolerance a user specifies. When sampling techniques are incorporated into Benders Decomposition based algorithms this is not the case. Instead sample means are used to estimate one or both of the bounds on the optimal solution.

As estimates are being used, it is not the case that the bounds improve monotonically. The estimates are the means of a randomly sampled set of observations. Hence the estimates are unbiased, as they converge to the true values of the bounds when enough samples are taken, regardless of which set of randomly chosen samples is used. The variances of the estimates can be used to provide a gauge on how much faith we can place in the current estimates. The lower the variance, the more confident a decision maker can be about the value of the estimate as a bound on the actual solution.

Hence the answer to the two addition questions are intertwined. The number of samples that must be taken depends upon how much confidence one wants to have in the final solution. Investigations into answering these questions in the multi-stage stochastic linear programming field can be found in Morton (1993) [52]. Later in this chapter, we consider these findings in addition to our own rules for deciding when to stop our implementation of Pereira & Pinto's sampling algorithm as applied to folded multi-stage stochastic linear programmes.

4.3 Monte Carlo Sampling & Its Extensions

The type of sampling that we combine within our implementation of Benders Decomposition is *Monte Carlo* sampling. This method samples at random from the distribution of possible scenarios, and uses the mean of the sampled scenarios as an estimate of the true expectation. After outlining the theory behind Monte Carlo sampling, we look at various variance reduction techniques. These aim to reduce the computational effort required to obtain as accurate results as basic Monte Carlo sampling.

4.3.1 Standard Monte Carlo Sampling

The problem of calculating the optimal solution of a multi-stage stochastic linear programme can be expressed as the following simplified expectation.

$$\mathbb{E}(C) = \sum_{\omega \in \Omega} p(\omega) C(\omega), \tag{4.1}$$

where ω is one particular scenario from the set Ω that contains all of the possible scenarios that can occur, $p(\omega)$ is the probability of scenario ω occurring and $C(\omega)$ is the cost incurred by this scenario. The term scenario is used to mean one particular path in a multi-stage stochastic linear programme. Where a path is one route from the first stage sub-problem to a final stage sub-problem consisting of one sub-problem from each stage in between.

In real-world problems the number of possible future scenarios can be large, of the order of 10^{60}, and so evaluating each one is impractical. Monte Carlo sampling gives an estimate of the expectation in equation (4.1). The average cost of a few selected scenarios is calculated, rather than the weighted sum of the costs for all of the scenarios. This gives an unbiased estimate of $E(C)$, provided the samples are drawn according to the probability distribution $p(\omega)$. This estimate is

$$\overline{\mathbb{E}}(C) = \frac{1}{N} \sum_{i=1}^{N} C(\omega^i), \tag{4.2}$$

where N is the number of samples used to obtain the estimate. The variance of this estimate is given by

$$var(\overline{\mathbb{E}}(C)) = \frac{var(C)}{N}, \tag{4.3}$$

where $var(C)$ is the variance of C the cost function.

This gives a measure of the uncertainty surrounding the estimate in equation (4.2). However, it requires that the variance of C is known, which is not likely to be the case, as this would involve knowing $\mathbb{E}(C)$, which is what we are trying to estimate.

$$\overline{var}(C) = \frac{1}{N-1}\sum_{i=1}^{N}\left[C(\omega_i) - \overline{\mathbb{E}}(C)\right]^2 \tag{4.4}$$

Instead, the standard sample variance given in equation (4.4) can be used to estimate $var(C)$, and this can be used to estimate the variance of our estimate of the expectation of the original function.

$$\begin{aligned} \overline{var}(\overline{\mathbb{E}}(C)) &= \frac{\overline{var}(C)}{N} \\ &= \frac{1}{N}\left(\frac{1}{N-1}\sum_{i=1}^{N}\left[C(\omega^i) - \overline{\mathbb{E}}(C)\right]^2\right) \end{aligned} \tag{4.5}$$

The theoretic variance of the estimate given by Monte Carlo sampling, (4.3), varies inversely with the sample size N. Hence, to improve the certainty of estimates, the sample size has to be increased. This results in an increase in computational effort.

Monte Carlo sampling makes use of the information given in the original probability distribution of the different possible future scenarios. However, no use is made of other information that can readily be gleaned from the problem. Stratified sampling, antithetic variables and importance sampling are three methods that try to reduce the variance associated with an estimate obtained from Monte Carlo sampling.

4.3.2 Variance Reduction Techniques

When sampling from a large distribution, the variance of the sample mean is a measure of how certain one can be of this estimate. Being able to reduce

the variance of a sample mean without increasing the sample size increases the certainty of an estimate without increasing the computational effort involved. Alternatively, with the same amount of computational effort we can obtain a more certain sample mean. Any method that tries to improve an estimate by minimising this uncertainty is called a *variance reduction* technique. There are several ways of reducing the variance of standard Monte Carlo sampling. These include stratified sampling, antithetic variables and importance sampling.

Stratified sampling attempts to reduce the variance of a sample mean estimate by splitting the support of the random variable into disjoint sections, or strata, such that the values that the random variable takes within any one stratum are similar. Samples are drawn from these strata, with a mean and its variance being estimated for each disjoint stratum. If the strata are selected such that the values that the random variable takes within each strata are similar, then the variance of these estimates are very low. The individual sample means and variance estimates for the separate strata are combined to provide an estimate of the sample mean and its variance for the original problem.

The main problem in applying stratified sampling when using Monte Carlo sampling within Benders Decomposition based algorithms is that it is very difficult to know *a priori* how best to divide up the support of the random variable. The main questions that arise when considering this type of variance reduction method are how many strata to use, how to define the strata and how many samples to allocate to each strata. As with all three variance reduction techniques discussed here, if the methods are not used with some insight and forethought, then they can in fact cause the variances to be greater than those given by basic Monte Carlo sampling. Indeed, the "reduction" in the title of this section might better be changed to "altering".

The principle underlying *antithetic variables* is that if two samples have a negative correlation, then their combined contribution to the variance is lower than if they were independent. In the notation of the previous section, let ω^1 and ω^2 be two samples. If they were independent, then the variance of the sample mean is given as $var(C)/2$. However, if there is a negative correlation between the samples, then the variance of the sample mean is given by $(1+r)var(C)/2$, where $r(<0)$ is the correlation between the two samples.

One way to introduce such a correlation between two samples is to use the inverse distribution of the original random variable. Thus, if $\omega^1 = F^{-1}(U)$, then $\omega^2 = F^{-1}(1-U)$, where F is the distribution function of C and U is the uniform

distribution on (0,1). The magnitude of the correlation between two samples generated in this manner depends upon how symmetrical distributed the random variable C is; the more skew this distribution, the less negative the correlation. Again, it is possible to use this method of variance reduction and obtain a result that is less accurate than if it had not been used. This is often the case if the underlying distribution is very skew.

In the original calculation of the expectation of the cost function, equation (4.1), two factors determine the contribution of a scenario to the expectation; the probability of the scenario occurring and the magnitude of the cost for this scenario. If a scenario has a very small probability but an extremely high cost, then its contribution to the expectation of the cost function will be larger than its probability suggests. Basic Monte Carlo sampling is unlikely to select such a scenario, due to its small probability, despite the contribution from this scenario being significant. A technique that attempts to capture this effect is *importance sampling.*

Importance sampling tries to exploit this by sampling according to a new distribution which takes the cost of a scenario into account. As calculating the cost of a scenario is computationally expensive, requiring the solution of a series of linear programmes, importance sampling uses a simple analytic function to approximate the cost function. This approximation is used to select of important scenarios.

There are two ways of applying such an approximating function. The first is to consider the difference between its value and that of the original function, and estimate the mean and variance of this difference. This is the method of *control variates*, of which a basic analysis in an electricity transmission setting can be found in Pereira *et al.* (1992) [54]. In the importance sampling that we incorporate into our Benders Decomposition, the ratio of the approximating function to the original function is estimated, along with its variance.

Returning to the expectation given in (4.1), this can be re-written as

$$\begin{aligned} \mathbb{E}(C) &= \sum_{\omega \in \Omega} p(\omega) C(\omega) \\ &= \sum_{\omega \in \Omega} q(\omega) \left(\frac{p(\omega) C(\omega)}{q(\omega)} \right) \end{aligned} \tag{4.6}$$

where $q(\omega)$ is an arbitrary probability distribution on Ω. Substituting $\widehat{C}(\omega) = (p(\omega)C(\omega))/q(\omega)$ into equation (4.6) gives

$$
\begin{aligned}
\mathbb{E}(C) &= \sum_{\omega \in \Omega} q(\omega) \widehat{C}(\omega) \\
&= \mathbb{E}(\widehat{C})
\end{aligned}
\tag{4.7}
$$

The variance of $E(\widehat{C})$ is given by

$$
\begin{aligned}
var(\mathbb{E}(\widehat{C})) &= \sum_{\omega \in \Omega} \left[\widehat{C}(\omega) - \mathbb{E}(\widehat{C})\right]^2 q(\omega) \\
&= \sum_{\omega \in \Omega} \left[\widehat{C}(\omega) - \mathbb{E}(C)\right]^2 q(\omega)
\end{aligned}
\tag{4.8}
$$

The choice of $\widehat{C}$ that minimises this variance is

$$
\widehat{C}(\omega) = \mathbb{E}(C), \quad \forall \omega \in \Omega
$$

This is equivalent to

$$
q(\omega) = \frac{p(\omega)C(\omega)}{\mathbb{E}(C)}, \quad \forall \omega \in \Omega \tag{4.9}
$$

The probability distribution $q(\omega)$ has the ideal property that any sample taken using this probability distribution gives an estimate with zero variance, i.e. sampling one path according to this optimal distribution gives the correct expectation as the estimate. This distribution fulfills the criterion given above, that scenarios are sampled according to how much they contribute to the expectation, and so more "important" scenarios have a greater chance of being sampled.

The probability distribution introduced in (4.6) is arbitrary, so any probability distribution can be used, with the resulting variance being given by (4.8). However, some choices of probability distribution lead to an increase in the variance of the associated $\widehat{C}$, and so increase the uncertainty in the estimate of its expectation — defeating the purpose of changing probability distribution.

The importance distribution given in (4.9) relies upon the expected cost being known. As this is the quantity that is being calculated, this is not a practical option. However, it can be used to guide the construction of a good distribution

to sample from. An approximation to this distribution is derived by substituting another function, $Z(\omega)$, for $C(\omega)$ in (4.9).

The criteria that $Z(\omega)$ must satisfy are that it is a good approximation to $C(\omega)$, it is computationally easy to calculate for each ω and it is practical to calculate its expectation. A sub-optimal distribution, $\overline{q}(\omega)$, that uses Z instead of C, can be substituted into equation (4.7), to give

$$\mathbb{E}(C) = \sum_{\omega \in \Omega} \overline{q}(\omega)\overline{C}(\omega) \tag{4.10}$$

where $\overline{C}(\omega) = C(\omega)\mathbb{E}(Z)/Z(\omega)$, and $\overline{q}(\omega)$ is given by

$$\overline{q}(\omega) = \frac{p(\omega)Z(\omega)}{\mathbb{E}(Z)} \tag{4.11}$$

The better an approximation Z is of C, the lower the variance of the expectation given in (4.10). As $\overline{q}(\omega)$ is a probability distribution, it is necessary that $Z(\omega)$ has the same sign as $\mathbb{E}(Z)$ for all $\omega \in \Omega$.

4.4 Implementing Importance Sampling within Benders Decomposition

When implementing importance sampling within Benders Decomposition based algorithms the main task is to find a function that is a good approximation of the cost function. The characteristics that a function has to have for it to be considered to be a good approximation are that it is relatively close in value to the function being approximated over the range of scenarios under consideration, it is easy to evaluate for every scenario, it is easy to sample from the associated density function and it is computationally easy to calculate its expectation.

Two approximation schemes were suggested in Dantzig & Glynn (1990) [22]; a multiplicative model and an additive. The additive model has been analysed in more detail in Infanger (1994) [46] as well as being used by Morton (1993) [52] and being extended to piece-wise linear additive approximations in Krishna (1993) [50].

The multiplicative model assumes that the original function is multiplicative in its arguments. While being easy to use in calculations, this model has been found to be a fairly poor approximation, Dantzig & Glyn (1990) [22].

4.4.1 Additive Model

The additive model assumes that the cost function being estimated is additive in its arguments. Assuming that there are r stochastic parameters in the problem, this model assumes the following

$$C(\omega) \approx \sum_{i=1}^{r} C_i(\omega_i) \tag{4.12}$$

where each scenario ω can be written as a vector of the stochastic parameters $(\omega_1, \omega_2, \ldots, \omega_r)$, and its probability occurring, $p(\omega)$, can be written as $\prod_{i=1}^{r} p_i(\omega_i)$. The stochastic parameters in the model are assumed to be independent random variables. The cost function is separated into two parts, with the aim of isolating the difference in the costs of the various scenarios. The first is the cost of a base case scenario, and the second the difference between the cost associated with a scenario and the cost of the base case scenario, as shown by the following

$$\begin{aligned} C(\omega) &= C(\tau) + (C(\omega) - C(\tau)) \\ &\approx C(\tau) + Z(\omega) \end{aligned} \tag{4.13}$$

where τ is a base case scenario. The task is now to find expectation of the second term in (4.13).

Possible candidates for the base case are either the lowest cost scenario or the highest cost scenario. Using an extreme case scenario ensures that the sign of the function $Z(\omega)$ is always either positive or negative. This is required to ensure that the sign of $\mathbb{E}(Z)$ is the same as the sign of $Z(\omega)$ for all $\omega \in \Omega$. The base case scenario need not actually occur in Ω, e.g. the scenario with the lowest cost might not be a possibility for a given multi-stage stochastic linear programme, but as its value is used as an offset, it can still be used as a base case. In the hydro-electric planning problem with no cost for spilling, the scenario where there is always heavy rainfall provides the lowest cost scenario.

Since we are assuming that $C(\omega)$ is additive, we approximate the difference term in (4.13) with the following marginal cost model

$$\begin{aligned} Z(\omega) &= \sum_{i=1}^{r} [C(\tau_1, \tau_2, \ldots, \tau_{i-1}, \omega_i, \tau_{i+1}, \ldots, \tau_r) - C(\tau)] \qquad (4.14) \\ &= \sum_{i=1}^{r} Z_i(\omega_i) \end{aligned}$$

where the marginal cost of scenario ω with respect to τ is approximated by $Z(\omega)$.

To use $Z(\omega)$ we need to know its expectation. As we are assuming that the original cost function, and so also our approximation function, is additive, we can calculate this as the sum of r one-dimensional sums, instead of the one r-dimensional sum in the original expectation. This gives

$$\begin{aligned} \mathbb{E}(Z) &= \sum_{\omega \in \Omega} p(\omega) Z(\omega) \\ &= \sum_{\omega \in \Omega} p(\omega) \sum_{i=1}^{r} Z_i(\omega_i) \\ &= \sum_{i=1}^{r} \sum_{\omega \in \Omega} p(\omega) Z_i(\omega_i) \\ &= \sum_{i=1}^{r} \mathbb{E}(Z_i), \end{aligned}$$

where the distribution is assumed to be discrete and finite, $|\Omega| < \infty$.

The desired expectation, (4.1), can be re-written in the same form as in equation (4.10),

$$\begin{aligned} E(C) &= C(\tau) + \sum_{\omega \in \Omega} p(\omega)\left(C(\omega) - C(\tau)\right) \\ &= C(\tau) + \sum_{\omega \in \Omega} \overline{q}(\omega) \frac{(C(\omega) - C(\tau))}{Z(\omega)} \mathbb{E}(Z) \\ &= C(\tau) + \sum_{i=1}^{r} \sum_{\omega \in \Omega} \overline{q}(\omega) \frac{(C(\omega) - C(\tau))}{Z(\omega)} \mathbb{E}(Z_i), \end{aligned} \tag{4.15}$$

where $\overline{q} = \left(p(\omega)Z(\omega)\right)/\mathbb{E}(Z)$. This equation shows that the expectation we are after can be written as a sum of expectations. Importance sampling gives an estimate of each of these sums individually. Hence our estimate to the original expectation is given by

$$\overline{\mathbb{E}}(C) = C(\tau) + \sum_{i=1}^{r} \frac{\mathbb{E}(Z_i)}{n_i} \sum_{\omega \in S_i} \frac{C(\omega) - C(\tau)}{Z(\omega)} \tag{4.16}$$

as an estimate for the original expectation. Each of the r expectations that have to be calculated in (4.15) is approximated using importance sampling with a sample S_i, where $n_i = |S_i|$. When sampling to find a scenario ω for expectation i, the importance distribution is used to find the ω_i component, and the original distribution is used to find the other components.

The variance of this estimate is given by

$$var(\overline{\mathbb{E}}(C)) = \sum_{i=1}^{r} \frac{\mathbb{E}(Z_i)^2 \sigma_i^2}{n_i} \tag{4.17}$$

where $\sigma_i^2 = var(\frac{C(\omega)}{Z(\omega)})$, the variance being taken with respect to the ith importance distribution. Hence, how the sample sizes, n_i, are chosen effects the variance in the estimate. The individual variances, σ_i^2 are not known, but can be approximated in a similar manner to the approximation used in equation (4.5).

The additive approximation given above for a two-stage stochastic linear programme can be extended to the multi-stage case. In this setting, it is necessary to construct an additive approximation, Z_t, for each stage t. As in the two-stage

case, a base case scenario is selected, and an approximation function constructed for the difference between any scenario and this base case scenario. When using importance sampling with this multi-stage additive approximation, it is necessary to sample stage by stage. Having chosen a sample S_t for stage t, a sample of the descendents of each of the scenarios sampled in stage t is made. This is repeated forwards through the problem.

4.4.2 Updating the Additive Approximation Coefficients

When constructing an additive model, the first step is to select a base case scenario, τ in the notation of equation (4.13). This could be a scenario in which a series of extreme values for the stochastic parameters occur. In the reservoir problems discussed in Chapter 5, the scenario where the inflows in each stage are either all at their highest, a scenario with downpours in every stage, or all at their lowest, drought in every stage are two candidates. However, any scenario can be used as a base case scenario, as it is the difference between this scenario and the other scenarios that is used to form the additive approximation.

Having selected a base case scenario, the sub-problems along this scenario are solved and the costs associated with each stage of the scenario are stored and used as a reference for the other scenarios. The additive coefficients for sub-problems off the base case scenario are obtained by solving scenarios that systematically deviate from the base case scenario in one sub-problem. These are calculated for all of the values that each stochastic parameter in each stage can take.

Working within a particular stage, each stochastic parameter is changed from its base case value to each of the possible values that it can take. For each of these variations, a path is solved that is the same as the base case scenario in all other respects, except for the deviation of this stochastic parameter. The difference between the cost of this path and that of the base case scenario is added to the additive model. This is repeated for each value that the stochastic parameter can take. The process is repeated for each of the stochastic parameters in that stage, and the whole process is repeated for every stage of the problem.

In Infanger (1994) [46], the discussion of the additive approximation model for use with importance sampling is mainly limited to two-stage stochastic linear programmes. The description of the multi-stage problems is only dealt with as an addendum. Due to this it fails to highlight that there is a significant difference between the two cases. When solving a non-leaf sub-problem of a multi-stage stochastic linear programme, the solution depends upon the current

approximation to the future costs of the sub-problem. When an additive approximation is first generated, the problem may only contain extreme bounds for the recourse functions, and so result in a very poor approximation.

As any Benders Decomposition based algorithm proceeds, the solution of the same sub-problem becomes more accurate than when it was solved at the start of the algorithm. This is not the case with a two-stage problem, as the solution of any sub-problem except the root node, is guaranteed to provide an exact solution. An additive approximation that is fitted to the initial form of a multi-stage stochastic linear programme is likely to decrease in accuracy as the algorithm continues. To overcome this problem, some use must be made of the new information that has been generated, in the form of cuts passed backwards through the problem, by the algorithm.

One way in which this can be achieved is to re-calibrate the additive approximation at regular intervals as the algorithm proceeds. This requires the additive approximation to be calibrated from scratch, and so incurs a significant overhead due to the number of sub-problems that have to be solved. New information is passed backwards through the problem with every sample that is taken. However, this is only being used in the additive approximation after the next re-calibration. Therefore there is a need to balance the gain in the accuracy of the approximation due to the re-calibration against the computational effort required to update the approximation.

An alternative way to update the additive approximation as the sample paths are taken is to use multiple linear regression. The additive model can be thought of as a linear function with as many variables as there are values that the stochastic parameters in every stage can take. I.e. a problem consisting of three stages with six nodes across each stage has eighteen variables in the additive model. Having sampled a path through the problem, the variables take values zero or one, signifying whether that node was on the path, and the overall cost of the sample path is the value of the linear function for these inputs.

Using multiple linear regression to update the additive model has the advantage that it allows the information obtained with every new sample to be used immediately. It is possible to do this because it is relatively easy to update the estimated coefficients of a multiple linear regression when new observations become available. As there is no need to solve any sub-problems, this updating is also computationally cheap.

4.4.3 Multiple Linear Regression

The aim of multiple linear regression is to fit a multi-dimensional linear model to a collection of observed data. The general form of such a model is

$$\begin{aligned} y &= \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p, \\ &= \beta_0 + \sum_{k=1}^{p} \beta_k x_k. \end{aligned} \tag{4.18}$$

Here the variables $x_1, \ldots, x_p$ are p predictor variables. In the additive model given above, these variables take values zero or one, depending on whether node $i \in [1, p]$ was on the sample path. The fitted response variable, y, is the predicted cost for the path defined by $(x_1, x_2, \ldots, x_p)$. While the actual observed responses are denoted by $\hat{y}$. As we are using multiple observations to fit our model, each resulting from one sample through the problem, it is convenient to re-write equation (4.18), as

$$Y = X\beta, \tag{4.19}$$

where Y is an $n \times 1$ vector of responses, X is an $n \times (p+1)$ matrix of predictors, and its columns can be written as $(p+1)$ $n \times 1$ vectors, $\mathbf{x}_i, i = 1, \ldots, p$. β is a $(p+1) \times 1$ vector of regression coefficients. If β_0 is non-zero, i.e. the model has a constant term, then the initial column of the matrix, $\mathbf{x}_1$, is a column of one's,

The aim of multiple linear regression is to find the best unbiased estimate for the regression coefficients. This results in $\hat{\beta}$, the predictions for the regression coefficients given by the fitted model, minimising the difference between the actual response and the observed response for each of the samples used to fit the model. This can be expressed as

$$\begin{aligned} \hat{\beta} &= \underset{\beta}{\operatorname{argmin}} \|\hat{Y} - X\beta\|_2^2, \\ &= \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^{n} \left(\hat{y}_i - \beta_{i0} - \sum_{k=1}^{p} \beta_k x_{ik} \right)^2, \end{aligned} \tag{4.20}$$

using the form of the model given in equation (4.18). By considering this minimisation, the best estimate of $\hat{\beta}$ is the solution to the following normal equations.

$$X^T X\hat{\beta} = X^T\hat{Y}, \tag{4.21}$$

which can be re-written as

$$\hat{\beta} = (X^T X)^{-1} X^T \hat{Y}. \tag{4.22}$$

The form of the solution given in equation (4.22) suggests that the best estimate of the regression coefficients can be found by calculating $X^T X$, inverting this matrix, and then post-multiplying it by the product $X^T Y$. However, there are two reason why this is not the ideal way to perform this calculation. The main disadvantage of this method is that it requires the computation of a large number of inner products that are prone to numerical errors. Another source of numerical instability is in the inversion of the matrix product; it would be preferable to obtain this by first factorising the matrix product and using this with substitutions instead of directly applying the matrix inversion. The second disadvantage is that the number of rows in X increases in number with each new sample. Therefore, it would be advantageous to be able to update our estimate of $\hat{\beta}$, rather than having to re-calculate our estimate from scratch after each new sample.

One way to overcome both of these problems is to decompose the matrix X into the following matrix product, where Q is an $n \times n$ orthogonal matrix, and R is upper triangular,

$$X = QR.$$

Additionally, if

$$Q^T\hat{Y} = \begin{bmatrix} c \\ d \end{bmatrix} \begin{matrix} p+1 \\ n-(p+1) \end{matrix}$$

and

$$Q^T X = R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \begin{matrix} p+1 \\ n-(p+1) \end{matrix}$$

then the original minimisation used to find $\hat{\beta}$, given in equation (4.20), can be re-written as

$$\begin{aligned} \|\hat{Y} - X\beta\|_2^2 &= \|Q^T(\hat{Y} - X\beta)\|_2^2, \text{ as } Q \text{ is orthogonal,} \\ &= \|Q^T\hat{Y} - Q^TX\beta\|_2^2, \\ &= \|\begin{bmatrix} c \\ d \end{bmatrix} - \begin{bmatrix} R_1 \\ 0 \end{bmatrix}\beta\|_2^2, \\ &= \|c - R_1\beta\|_2^2 + \|d\|_2^2, \end{aligned} \tag{4.23}$$

As $\|d\|_2^2$ is constant, the solution of

$$R_1\beta = c, \tag{4.24}$$

is equal to the solution of the original problem, (4.20). Since R_1 is upper triangular, the solution to equation (4.24) can be obtained by back substitution. This is a reasonably cheap computation that is not prone to numerical instability. Hence the benefit of using such a QR-factorisation depends upon the ease and stability of decomposing the original matrix, X. One method of constructing this decomposition is to build Q from a series of orthogonal elementary matrices.

These elementary matrices are called *Householder transformations* and a fuller discussion of their properties can be found in Householder (1964) [40], while specific advice on their implementation as computer codes can be found in Thisted (1988) [59]. Householder transformations, H_i, are constructed so that the ith transformation deletes the $n-i$ sub-diagonal entries in the ith column of a matrix when the matrix is pre-multiplied by H_i. In addition to this, the transformations are generated so that they are length preserving, i.e. they are orthogonal. Matrices $H_1, H_2, \ldots, H_p$ are constructed iteratively, such that H_{i+1} is used used to pre-multiply the partially transformed matrix, $X_i = H_iH_{i-1}\ldots H_1X$, to generate X_{i+1}. After the p transformations have been applied the resulting matrix X_p is upper triangular, and so has the desired form of R. Also, since the individual transformations are orthogonal, the matrix Q can be formed explicitly as $Q = H_pH_{p-1}\ldots H_2H_1$.

The general form of the Householder transformations is

$$H_i = I_n - 2\frac{\tilde{\mathbf{x}}_i\tilde{\mathbf{x}}_i^T}{||\tilde{\mathbf{x}}_i||^2},$$

where I_n is the $n \times n$ identity matrix and the vector $\tilde{\mathbf{x}}_i$ is derived from the last $n - i + 1$ components of vector $\mathbf{x}_i$ using the following construction

$$\tilde{\mathbf{x}}_i = \begin{pmatrix} 0 \\ x_{ii} \pm s \\ \mathbf{x}_i^L \end{pmatrix},$$

where x_{ji} is the j component of the vector $\mathbf{x}_i$, $\mathbf{x}_i^L$ are the last $n - i$ components of $\mathbf{x}_i$, and $s^2 = \sum_{j=i}^{n} x_{ji}^2 = ||\mathbf{x}_i^L||_2^2 + x_{ii}^2$.

As new observations become available, after solving another sample with the additive approximation model, it is possible to either update the factorisation of X or to re-compute it from scratch. This might be done after the oldest observations have been discarded. By adding an extra row to X, i.e. to include an extra observation, the previous QR-factorisation can be updated to a QR-factorisation for the new, appended X by applying a sequence of Givens transformations. These are orthogonal elementary matrices that have the property that they are constructed in order to zero one specific element of a matrix. If $\mathbf{w}$ is the $p \times 1$ vector from the new sample, then the appended matrix, $\bar{X}$, can be defined as

$$\bar{X} = \begin{bmatrix} \mathbf{w}^T \\ X \end{bmatrix},$$

and let

$$H = \begin{bmatrix} \mathbf{w}^T \\ R \end{bmatrix},$$

where R is the $n \times p$ upper triangular matrix from the original QR-factorisation. The matrix H is upper Hessenberg, and so can be reduced to an upper triangular matrix by removing at most p entries. This can be achieved by using a sequence of p Givens transformations, $G_1, G_2, \ldots, G_p$, each removing one entry, giving the new upper triangular matrix $\bar{R}$.

$$\bar{R} = G_p^T G_{p-1}^T \dots G_2^T G_1^T H,$$

where $\bar{R}$ is an upper triangular, and is the new, updated R matrix for the decomposition. The updated orthogonal matrix, Q, of the decomposition can be defined as

$$\bar{Q} = \text{diag}(1, Q) G_1 G_2 \dots G_{p-1} G_p,$$

where Q is the orthogonal matrix form the previous QR-factorisation. Hence, the only additional work required to update X, given a new observation $\mathbf{w}$, is to calculate the Givens transformations and perform the matrix multiplications. As the Givens transformations are orthogonal elementary matrices, they require very little work to construct, and in calculations are numerically stable. However, as the computational effort involved in calculating $\hat{\beta}$ is not high, particularly since it does not involve the solution of any sub-problems, the extra gain in updating the estimated regression coefficients, over factorising the matrix $\bar{X}$ from scratch, is unlikely to be a significant compared to the running time of an algorithm, especially when compared to re-calibrating the additive approximation from scratch.

4.4.4 Results

An additive approximation generated before any cuts have been placed in the multi-stage stochastic linear programme will be a poorer guide to an importance sampling scheme than one which is generated after cuts have been passed backwards through the problem. This is because more information about future costs is available at the initial stages of the problem. The aim of any updating procedure is to capitalise upon this extra information, and hence maintain a good additive approximation, i.e. an additive approximation that is close to the original cost function, and so reduces the variance of the estimated expectation when it is used to guide the selection of samples within an importance sampling scheme.

Two ways to update an additive approximation were given in the previous section. One uses multiple linear regression to update the coefficients of the additive model after each sample is taken, while the other re-calibrates the additive model after a fixed number of samples. The computational effort involved in the two

methods for updating the additive approximation is spread during the solution process in different ways. The multiple linear regression method performs a small amount of extra work after each sample, while the re-calibration method performs a significant amount of work a limited number of times. Every time the additive model is re-calibrated several sub-problems have to be solved, whereas the multiple linear regression method requires only the updating of a matrix and a backwards transformation after every sample.

The multiple linear regression method updates the coefficients of the additive model after every sample. As the initial samples are taken when very few cuts have been added to the sub-problems, the samples taken later are more likely to be accurate, i.e. closer to the actual cost of the particular path sampled. The additive approximation generated by the multiple linear regression method can be improved by gradually removing the earlier, less accurate samples. This is achieved by setting a limit to the number of observations used in the multiple linear regression.

The methods for updating the additive approximation presented in the previous section were tested on three instances of the hydro-electricity generation model described in Chapter 5. The three problems have four, six and twelve stages, each with three scenarios per stage. This results in their additive approximations consisting of eight, twelve and twenty four coefficients respectively. Both methods for updating the additive approximation were tested using 1000 random samples drawn according to an importance distribution that uses the additive model described in the previous section. This was repeated 100 times, and the average results are shown in Table 4.1.

The results in Table 4.1 show that the times taken by the re-calibration method increase nearly monotonically with the number of re-calibrations. In general the multiple linear regression method is faster than the re-calibration method, with the difference being greater for the larger problems. Of the figures given in Table 4.1, the first three columns contain the results for the multiple linear regression method using the most recent 25, 100 and 1000 observations. The other four columns contain the results when the additive model is re-calibrated regularly once, twice, five or ten times during the 1000 samples. Each experiment was repeated 100 times with a different seed for the random number generator. For each experiment, the results contain the standard deviation of the estimate of the expected costs, the total time for the run and the total time spent updating the coefficients of the additive model, both in seconds. The results in **bold-face**

		MLR			Re-Calibrating			
		No Observations			No Calibrations			
		25	100	1000	1	2	5	10
4 Stage Problem								
	Std Dev	**2178.83**	2632.41	2872.47	4391.86	4403.4391	2437.52	**1985.27**
	Total Time (s)	**264.65**	259.80	261.51	259.68	264.24	267.22	**272.02**
	Update Time (s)	**0.22**	0.20	0.18	1.36	1.96	3.85	**6.98**
6 Stage Problem								
	Std Dev	409.89	**404.55**	407.00	**412.84**	424.98	430.38	435.35
	Total Time (s)	334.91	**328.08**	328.14	**333.83**	334.53	339.80	342.87
	Update Time (s)	0.33	**0.30**	0.29	**1.85**	2.69	5.42	9.68
12 Stage Problem								
	Std Dev	7146.40	7128.34	**6183.13**	7384.06	6794.76	7717.63	**6209.34**
	Total Time (s)	1224.71	1215.09	**1207.49**	1259.02	1258.79	1292.76	**1305.90**
	Update Time (s)	0.84	0.77	**0.74**	7.36	10.73	22.43	**40.48**

Table 4.1: Using Multiple Linear Regression and Re-Calibration to Update the Additive Approximation

have the lowest standard deviation for each of the methods for a particular test problem.

The first conclusion to draw from Table 4.1 is that the number of observations required by the MLR method to obtain the best reduction in the standard deviation increases with the problem size. The standard deviation of the smallest problem is smallest when the least number of observations are used, while the converse is true for the largest problem. This concurs with the number of coefficients in the additive approximation also increasing with problem size.

There does not appear to be the same pattern in the re-calibration results. For the two larger test problems one of the re-calibration runs gives the lowest standard deviation, but for the medium sized problem the difference in the standard deviation between any of the runs is fairly low. The most dramatic differences between standard deviations across runs is for the four stage problem, where two of the re-calibration runs are nearly double the remainder.

However, there is a large difference between the two methods in the times taken. For the largest problem, the time taken by the runs using the multiple linear regression is always less than that taken by the re-calibration method, while for the smaller problems, this effect is similar but less extreme.

Neither method has a high computational overhead, but both result in a reduction in the standard deviation of the sample mean obtained using importance sampling for the smallest and the largest problems. For the medium sized problem the re-

calibration does not result in a reduction in the standard deviation, however the standard deviations for all of the tests on this problem are relatively close to one another. This may be due to the variance of the value function for the medium sized problem being an order of magnitude smaller than that of the two other problems.

In all of the tests shown, the time spent updating the additive approximation is never more than 3% of the total time taken to perform the 1000 samples in the runs presented in Table 4.1. The times taken to perform the updating using the multiple linear regression method decrease when more observations are used, while the times taken to perform the updates using the re-calibration method increase when more re-calibrations are used.

4.5 Sample Size and Stopping Rules

When sampling is used within Benders Decomposition, one or both of the upper and lower bounds on the optimal solution are replaced by estimates generated from sample means. This introduces several questions that are specific to sampling based algorithms, as compared to the results that can be drawn about the deterministic Benders Decomposition algorithm.

Here the term deterministic refers to the Benders Decomposition algorithm that does not incorporate any type of sampling and which gives the same results when applied to a multi-stage stochastic linear programme on different occasions. The randomness of the sampling methods discussed in this chapter are random within the limits of computer generated pseudo-random number streams.

The first question is what is now meant by a solution being optimal. The deterministic Benders Decomposition algorithm generates monotonically improving upper and lower bounds. An optimal solution is defined as one that has bounds that are within a pre-specified tolerance of each other. This result does not necessarily hold when either of these bounds is replaced by an estimate. Instead of trying to stop optimally in this instance, it is more important to ensure that a sampling based algorithm stops correctly, i.e. that the difference between the solution proposed by the sampling based algorithm and the optimal solution is less than the width of a confidence interval.

As the bounds are estimated by sample means, it is possible to use the variance of each estimate to generate confidence intervals around the bounds. In constructing confidence intervals we try to measure how sure we can be of a result. The lower the variance of an estimate, the more confidence we can have that the sample mean is close to the actual value of the bound it estimates.

The second question that arises when using sampling based Benders Decomposition algorithms is how many samples to take. We want the number of samples to ensure that the final answer lies within a particular confidence interval. In addition, if the algorithm does not terminate after taking this number of samples, how many more samples should be used next time? As was stated in the previous section, the variance of a sample mean is proportional to the reciprocal of the number of samples. Hence taking a larger numbers of samples results, on average, in smaller variances in the estimated bounds, and hence generates tighter confidence intervals. Therefore the question of how many samples can

now be replaced by the more pertinent question of how should the number of samples increase so as to ensure stopping correctly.

The issues of stopping rules and sample sizes is discussed briefly in Infanger (1994) [46], when implementing a Benders Decomposition based algorithm incorporating importance sampling for two-stage stochastic linear programmes. As the cuts that are added to the first-stage problem are estimated in this algorithm, both the upper and lower bounds are estimates. It is also assumed that the upper and lower bounds are approximately normal in their distributions. Based on this assumption, the Student t-test is employed to determine whether the gap between the bounds is within some specified tolerance. If this is the case, then a final check is made to ensure that a 95% confidence interval is tight enough, i.e. it is small enough relative to the lower bound estimate, and if this is the case then the algorithm stops. Otherwise a new sample is taken and the process is repeated.

Infanger's implementation of Benders Decomposition with importance sampling, DECIS, appears to use only fixed sample sizes. Although the notion that the sample size should be increased as iterations of a sampling based algorithm proceeds is discussed, in the results presented only fixed sample sizes are used. The results show that the larger the sample size used, the greater the coverage of the stopping rule. It is assumed that the upper and lower bound estimates are normal random variables.

The questions about stopping rules and sampling sizes are given a detailed examination in Morton (1993) [52]. Here the more general problem of sampling in conjunction with optimisation is discussed. The problem is formulated as

$$\begin{aligned} \text{minimise} \quad & z(x) \\ \text{subject to} \quad & x \in X \end{aligned} \tag{4.25}$$

For any sample of size n_k, in iteration k, a corresponding decision, x_k, is generated along with associated upper and lower bound estimates, $U_k(n_k)$ and $L_k(n_k)$. These estimated bounds depend both on the iteration and the number of samples used in this iteration. They can be combined to define a difference random variable, $D_k(n_k) = U_k(n_k) - L_k(n_k)$, that Morton initially assumes satisfies the central limit theorem, (CLT), i.e.

$$\sqrt{n_k}(D_k(n_k) - \mu_k) \Rightarrow N(0, \sigma^2) \text{ as } n_k \to \infty, \text{ where } \sigma > 0, \quad (4.26)$$
$$D_k(n_k) \text{ are independent for } k = 1, 2, \ldots,$$

where $\Rightarrow$ denotes convergence in distribution. The true upper and lower bounds at iteration k are u_k and l_k, where $U_k(n_k) \Rightarrow u_k$ and $L_k(n_k) \Rightarrow l_k$ as $n_k \to \infty$. The CLT in (4.26) can be re-stated as saying that at any iteration k of an algorithm, as the number of samples taken form the current state of the optimisation problem increases, then the distribution of the difference random variable, i.e. the gap between the upper and lower bound estimates, converges to a normal distribution. Very little is assumed about the convergence of the optimisation algorithm, only that there exists a subsequence $\{\mu_{k_j}\}_{j=1}^{\infty}$ such that $\mu_{k_j} \to 0$.

In addition to these assumptions, a positive confidence interval, denoted by ϵ, is used to define when the algorithm stops correctly, i.e. if $z(x_T) \leq z^* + \epsilon$, where T denotes the *stopping iteration* and is defined as

$$T = \inf_{k \geq 1}\{k : D_k \leq \epsilon'\}, \quad (4.27)$$

for a stopping tolerance of ϵ'. Using the definitions given above, Morton derives rates of increase for the number of samples taken that guarantee that the algorithm stops correctly with a certain probability. Different results are derived based on different assumptions about the properties of the difference random variable. The key point underlying all of the results is that the sample size should increase if we want to have any guarantee that the algorithm will stop correctly.

This result is derived at first under the assumption that the difference random variables are independent at each iteration and normally distributed, with their variance proportional to the reciprocal of the samples size. This leads to a sample size that is $\mathcal{O}(\ln k)$. Further results are then derived in a similar manner based upon different assumptions about the properties of the difference random variable. One of these results is the case where the difference random variable has no property other than it depends upon its history. This result is shown to be applicable to the Benders Decomposition algorithm that incorporates sampling with exact cuts being passed backwards through the problem. It has a sample size $\mathcal{O}(\ln^2 k)$.

The first of the results, for difference random variables that are assumed to be normally distributed, is given in Theorem 4.1

Theorem 4.1 (Normal Difference r.v) *Assume that* D_k *is drawn from* $N(\mu_k, \frac{\sigma^2}{n_k})$, *that* T *is defined as in (4.27), and define*

$$n_k = \left\lceil \left(\frac{\sigma_k}{\epsilon - \epsilon'}\right)^2 (\beta + 2p \ln k) \right\rceil, \tag{4.28}$$

where $0 \leq \epsilon' < \epsilon$, $\beta = max\{2ln[\zeta(p)/\sqrt{2\pi}\alpha], 1\}$, $0 < \alpha < 1$, *and* $\zeta(p) = \sum_{k=1}^{\infty} k^{-p}$, $p > 1$.

Then

$$P\{\mu_T \leq \epsilon\} \geq 1 - \alpha. \text{ (coverage)} \tag{4.29}$$

If, in addition, have independent D_k*'s, there exists a convergent subsequence of* $\{\mu_k\}$ *and* $\epsilon' > 0$ *then*

$$P\{T < \infty\} = 1. \text{ (finite stopping time)} \tag{4.30}$$

Proof

See Morton (1993) Chapter 4, Theorem 4.2 [52].□

This theorem gives two results. The first is that the sampling algorithm is guaranteed to stop in a finite number of iterations, T. However, no gauge is given as to the size of T. The second result is that if the sample size in iteration k is defined as in equation (4.28), then $[0, \epsilon]$ is a $(1-\alpha) \times 100\%$ confidence interval for μ_T.

As was suggested in Infanger (1994) [46], the term iteration may be interpreted loosely. Instead of increasing the sample size at every iteration, batches of the current sample size are taken, and after each is completed, a pre-test is used to see if the sample bounds are sufficiently close. If this is the case, then a re-sampling is done with the same sample size. If the bounds of this sample are not close enough, then the sample size is increased using the formula given in (4.28). If the bounds derived from the re-sampled batch are sufficiently close, then

the algorithm terminates. Otherwise the algorithm returns to solving batches of samples using the new sample size.

This result gives an *a priori* control on the quality of the proposed solution, x_T. However, because we are assuming that the difference random variable is normal, it is possible to take any sample size, n', and construct a $(1-\beta)\times 100$ confidence interval of the form $[z^*, z^* + D_{T'} + \frac{d_\beta \sigma_{T'}}{\sqrt{n}}]$, where $P(N(0,1) \leq d_\beta) = 1-\beta$. The difference is that the the width of the confidence interval is known beforehand when the sample size formula (4.28) is used. For the example above, we cannot control the width of the confidence interval, $D_{T'} + \frac{d_\beta \sigma_{T'}}{\sqrt{n}}$.

In addition to the assumption that the difference random variable is normally distributed, the above result also assumes that the sequence $\{\mu_k, \sigma_K\}_{K=1}^{\infty}$ is a sequence of constants. When the sampling is done within the optimisation algorithm, i.e. an internal sampling technique, the mean and variance are random variables that depend on the previous iterations, $k-1,\ldots,1$. Once in iteration k, the values for the mean and variance of the estimate depend directly upon the particular cuts that are sitting in each sub-problem. These were generated because of the samples that were chosen in the previous iterations, and if different scenarios had been chosen in prior iterations, then the current cuts would be different. This would be the case if a different stream of pseudo-random numbers had been used. Therefore the mean and variance depend directly on the history of the algorithm, and can be said to be *history-dependent.*

In deriving a formula for the sample-size when the mean and variance are constant when conditioned on the history, the difference random variables are assumed to satisfy the following,

$$P\left\{\frac{D_k-\mu_k}{\sigma_k/\sqrt{n_k}}\middle|\mathcal{H}_{k-1}\right\} \to \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{y}\exp(-u^2/2)du, \text{ as } n_k\to\infty. \tag{4.31}$$

This is a conditional form of the CLT, and implies that given a particular history, $\mathcal{H}_{k-1}$, the distribution of the difference random variable tends towards a normal distribution, with mean μ_k and variance $\sigma_k/\sqrt{n_k}$. The following theorem, that also assumes that the difference random variable is bounded, gives a sample-size formula for the history-dependent case.

Theorem 4.2 (Historical Difference r.v.) *Assume that T is defined as in (4.27), that the difference random variable is history-dependent and satisfies (4.31) and the following holds*

$$\sup_{k\geq 1} \mathbb{E} \exp\left[\gamma\left(\frac{D_K - \mu_k}{\sigma_k/\sqrt{n_k}}\right)\right] \text{ is bounded for } |\gamma| \leq \gamma_o. \tag{4.32}$$

Define

$$n_k = \left\lceil \left(\frac{\sigma_k}{\epsilon - \epsilon'}\right)^2 \left(\beta' + 2p\ln^2 k\right) \right\rceil, \tag{4.33}$$

where $0 \leq \epsilon' < \epsilon, \beta' = \max\{2\ln[\phi(p)/(\sqrt{2\pi}\alpha)], 1\}$, *where* $\phi(p) = \sum_{k=1}^{\infty} k^{-p\ln k}, p > (2\gamma_o^2)^{-1}$ *and* $0 < \alpha < 1$.

Then

$$\lim_{\epsilon\downarrow\epsilon'} P\{\mu_{T(\epsilon)} \leq \epsilon\} \geq 1 - \alpha. \text{ (asymptotic validity)} \tag{4.34}$$

Proof

See Morton (1993) Chapter 4, Theorem 4.9 [52].□

Whereas Theorem 4.1 gives a result that depends on the difference random variable being normally distributed, Theorem 4.2 only assumes the weaker property given in (4.31). Our implementation of Benders Decomposition, which uses importance sampling to solve multi-stage stochastic linear programmes, satisfies these weaker properties. As noted above, the decision at any iteration of our implementation of Benders Decomposition is a random variable, dependent upon the samples taken in previous iterations, and so the random variables of our problem are history-dependent. The probability space for the scenarios is discrete, and hence finite, so the upper bound estimates satisfy the historical CLT given in (4.31).

As exact cuts are used, the lower bound is deterministic, as conditioned on the history, and so the difference random variable also satisfies the historical CLT. In addition, since there are a finite number of scenarios, the cost as a function of scenario is bounded, and so the difference random variable also satisfies (4.32), and hence the asymptotic valid result of Theorem 4.2 follows. Hence the sample size given in (4.33) is guaranteed to give us an asymptotically valid confidence interval that we can choose the length of in advance.

Pre-selecting the values for ϵ and ϵ' used in formula (4.33) in Theorem 4.2 gives an absolute precision confidence interval. However, when implementing the sample size formula, the variance term, σ_k^2, is not likely to be known, and so the required sample size is also unknown. One way to way to overcome this is to select the width of the confidence interval, ϵ, and the stopping tolerance, ϵ', proportional to the population variance. However, this still suffers from the variance being uncertain. Morton shows that a sample variance estimate can be used in place of the actual population variance.

After stopping, the difference random variable can be re-sampled, and assumed to be normally distributed. Hence an approximate confidence interval can be constructed. A $(1 - \delta) \times 100\%$ confidence interval can be constructed, where $1 - \delta = P\{N(0, 1) \leq \omega_\delta\}$, with width $D_T^+ + \frac{\omega_\delta s_t}{\sqrt{n}}$, where n is the size of the re-sample. Using the variance information from previous iterations, we can now update the sample size so as to control the width of the confidence interval.

4.5.1 Results

In this section we discuss the results from using two stopping rules on three test problems. One rule uses batches with a fixed sample size, and terminates when the upper bound estimate lies within a constructed confidence interval, provided the confidence interval is not too large. The second stopping rule uses a sample size that increases according to Morton's $\mathcal{O}(\ln^2 k)$ sample size formula given in (4.33).

The fixed stopping rule consists of the following. Firstly a confidence interval is constructed. This has a width of either 1 or 2 standard deviations around the sample mean estimate of the upper bound, where a sample variance estimator from the current iteration is used in place of the standard deviation. The rule stops if the lower bound, which is not an estimate, lies within the confidence interval, and if the width of the confidence interval, as a percentage of the estimated upper bound, is within a pre-specified stopping tolerance.

The following results were obtained by applying this fixed sample size formula and Morton's $\mathcal{O}(\ln^2 k)$ to the solution of three multi-stage stochastic linear programmes. They are hydro-electricity reservoir management problems, based on our model of Southern Brazil which is discussed in Chapter 5. The three problems have four, six and twelve stages, and each has three scenarios per stage.

Stop Tol (%)	10	5
Stopping Its	1.13	1.36
Non-Stopping Its	2.86	3.13
Final Batch Size	45.84	50.31
Coverage (%)	100	100
Time (s)	36.65	41.64

Table 4.2: 4 Stage, Morton's $\mathcal{O}(\ln^2 k)$ Sample Formula

Stop Tol (%)	10	5
Stopping Its	1.00	1.02
Non-Stopping Its	2.24	2.26
Final Batch Size	40.00	36.8
Coverage (%)	100	100
Time (s)	44.39	43.81

Table 4.3: 6 Stage, Morton's $\mathcal{O}(\ln^2 k)$ Sample Formula

Tables 4.2–4.4 contain results from repeated experiments using the $\mathcal{O}(\ln^2 k)$ sample size formula (4.33). Each table presents the average results from 100 replications of the algorithm with a stopping tolerance of either 5% or 10%, relative to the sample estimate of the standard deviation. The rows of the tables contain the following summary statistics: number of stopping cycles performed; number of non-stopping batches sampled; size of the final batch; coverage and the running time in seconds. All of these results are the average taken over the 100 runs of the algorithm, except the coverage result. The initial sample size for each run was thirty. Each of the runs was performed with a different seed for the pseudo-random number stream.

Stop Tol (%)	10	5
Stopping Its	1.64	3.18
Non-Stopping Its	9.35	13.77
Final Batch Size	68.42	115.72
Coverage (%)	96	97
Time (s)	450.76	1355.76

Table 4.4: 12 Stage, Morton's $\mathcal{O}(\ln^2 k)$ Sample Formula

The first observation from Tables 4.2–4.4 is that the two smaller problems require far less work to solve compared with the larger problem using either stopping rule. This is reflected both in the average time taken to stop and the coverage of both methods. The times taken for the two smaller problems is significantly smaller than that required for the larger problem. The coverage for the smaller problems is also better than that of the larger problem for both stopping rules. However, this improvement in coverage is relative given that the coverage of both methods for all three problems is very high.

For the $\mathcal{O}(\ln^2 k)$ sample rule, the times taken to solve the two smaller problems does not vary much between the two different stopping tolerances. There is, however, a large increase in the amount of work required to solve the larger problem for the smaller stopping tolerance. This is a reflection of the fact that the twelve stage problem has many more sub-problems, and the unfolded equivalent problem is extremely large.

The algorithm performs batches of a fixed sample size until the difference random variable satisfies a stopping pre-test. In the runs presented above this test was whether the difference random variable was 5% or less of the current lower bound. Before performing a stopping cycle, the sample size is increased according to the sample size formula given in (4.33). After this re-sample is performed, the difference random variable is checked for termination. If the difference random variable is less than the stopping tolerance, as a fraction of the lower bound, then the algorithm terminates. Otherwise iterations of using the new sample size are repeated until the pre-test is satisfied

The pre-test need not be of the form given above. Any test that indicates that the upper bound estimate is close to the lower bounds is sufficient. The sample size is not increased with every new sample so as to limit the computational burden. However once the pre-test is satisfied, the sample size is increased before the stopping cycle is performed.

As a comparison with Morton's stopping rule, we ran our algorithm with a stopping rule that uses a fixed sample size. The results from these tests are given in Tables 4.5–4.7. This stopping rule uses fixed sample sizes of either 150 or 200 samples. The algorithm was run for a confidence interval with a width of one or two standard deviations, and for stopping tolerances of either 5% or 10%. As can be seen from the results in Tables 4.5–4.7, the solution times for all three problems is much higher than for the equivalent runs that use an increasing sample size.

	Stop Tol (%)	10		5	
	History	150	200	150	200
± 2 Std Dev					
	Samples	329	426	442	530
	Coverage (%)	100	100	100	99
	Time (s)	68.32	85.64	89.85	102.87
± 1 Std Dev					
	Samples	437	664	453	688
	Coverage (%)	98	100	98	100
	Time (s)	87.31	126.07	90.33	126.64

Table 4.5: 4 Stage, Fixed Sample Sizes

The coverage is better than for the other stopping rules when the confidence interval is 2 standard deviations wide, and as good when it is only one standard deviation wide. However, the amount of work performed is on average much higher using the fixed sample size. This is because there is no flexibility in either the sample size, nor in the width of the confidence interval. This is countered when the sample size formula is used as the "correct" sample size is found for the desired stopping tolerance. When selecting a sample size for a fixed sample size rule, there is very little to judge what the correct size to use is. The size of the confidence interval is also a parameter that is difficult to judge.

	Stop Tol (%)	10		5	
	History	150	200	150	200
± 2 Std Dev					
	Samples	171	234	171	234
	Coverage(%)	100	100	100	100
	Time (s)	50.85	66.07	50.65	65.91
± 1 Std Dev					
	Samples	251	336	251	336
	Coverage (%)	100	100	100	100
	Time (s)	69.98	89.95	70.43	89.66

Table 4.6: 6 Stage, Fixed Sample Sizes

Originally we performed these runs with sample sizes of thirty, but several of the more difficult tests failed to stop correctly. This is because the width of the confidence interval depends directly upon the standard deviation of the cost

	Stop Tol (%)	10		5	
	History	150	200	150	200
± 2 Std Dev					
	Samples	632	798	1171	1354
	Coverage (%)	100	100	100	100
	Time (s)	784.35	1009.85	1474.46	1720.05
± 1 Std Dev					
	Samples	1325	2009	1344	2059
	Coverage (%)	97	97	97	97
	Time (s)	1680.57	2603.79	1702.48	2673.78

Table 4.7: 12 Stage, Fixed Sample Sizes

function of the problem being solved. As the sample size is fixed, the average width of any confidence interval is also effectively fixed. For this reason we selected larger sample sizes than the initial sample size in the other runs.

As found in Morton (1993) [52], using a $\mathcal{O}(\ln^2 k)$ sample size leads to a far better stopping rule than if a fixed sample size is used. The performance, in terms of both coverage and the amount of work performed, of the stopping rule with a fixed sample size greatly depends upon the selected sample size. However the only way to estimate what a "good" samples size is relies upon knowing features of the cost function under investigation, such as its standard deviation. As in the case of importance sampling, with hindsight it is possible to sample perfectly.

This problem is avoided with the $\mathcal{O}(\ln^2 k)$ stopping rule because of its growing sample size. This results in the "correct" sample size being determined by the algorithm itself. Therefore avoiding the use of a sample size that is far larger than might be sufficient, or of using one that is too small to give any meaningful coverage.

4.6 Conclusions

The multi-stage stochastic linear programmes that are formulated as models of real-world problems are often very large. They are often too large to be solved by either direct methods or decomposition algorithms. This can be overcome by the use of sampling techniques in conjunction with another solution method.

There are two ways in which to incorporate sampling techniques into solution methods. One way is to perform the sampling around the solution method. This involves using the solution method to solve an approximation to the original problem. The approximation is determined by sampling to find a particular instance of the stochastic parameters of the problem. The resulting problem is solved exactly by the solution method of choice and the results used to derive a statistical estimate of the solution to the overall problem.

The other way of incorporating sampling techniques and a solution method is to perform the sampling within the solution algorithm. One example of this is the use of Monte Carlo sampling within Benders Decomposition. In each iteration of the algorithm, instead of solving all of the sub-problems in a multi-stage stochastic linear programme, a sample of possible paths through the event tree is selected and the sub-problems along these paths are solved. The cuts and solutions from these sub-problems are used to approximate the results that would have been obtained had a full iteration of the Benders Decomposition algorithm been performed.

Importance sampling is a way of improving the estimates obtained using Monte Carlo Sampling. When selecting samples, importance sampling uses both the distribution of the paths through an event tree and also the costs associated with the paths. The latter is not used when using basic Monte Carlo sampling. In this way a path that has a very small likelihood of occurring but which has a very large cost if it does occur will be sampled more frequently by an importance sampling algorithm than when basic Monte Carlo sampling is used.

The costs of the paths are part of the solution to the problem we are actually trying to solve using importance sampling. Hence they are not available to guide our importance sampling scheme. Instead we construct an additive approximation to the cost function, and use this to guide the importance sampling scheme. As the algorithm proceeds, more information is available which can be used to update the additive approximation. We have looked at two possible ways of using this newly available information.

One method uses multiple linear regression to update the approximation after each new sample, while the other method re-calibrates the approximation from scratch after batches of samples. The number of observations should be increased for problems with more stages when using the multiple linear regression method. For the re-calibration method, the main influence on the benefit obtained by updating more frequently depends upon the variance of the cost function being approximated, with more frequent re-calibrations required for problems with a larger variance. However, as this is what one is trying to estimate, it is difficult to predict how large the variance of the cost function will be before performing the sampling. Due to the difficulty in predicting the magnitude of the variance of a cost function, the multiple linear regression method would appear to be the preferred method for updating the additive approximation.

When Monte Carlo sampling is used within Benders Decomposition the notion of what is an optimal solution changes. This is because either one or both of the bounds on the optimal solution are estimated by sample means. In this case we are more interested in what confidence we can have in the solution and whether we have stopped correctly. We looked at two possible types of stopping rule. One used a fixed sample size, while the sample size in the other depended upon a formula, and increased with the number of samples taken. Our results confirmed those of Morton, that the fixed sample size rule is very poor. While the rule based on the sample size formula does not give us any idea of how many iterations will be required before starting the algorithm, the results from our tests show that the time is much less than that needed for the fixed sample size rule.

Chapter 5

The Brazilian Electricity Network

5.1 Introduction

The electricity generation networks of several countries around the world have a high proportion of hydro-electric generation capacity. This type of generation network is more difficult to schedule compared to systems that consist predominantly of thermal generating plants. The difference is due to three factors: the uncertainty surrounding future rainfall patterns, the fact that hydro-electric power effectively has a marginal cost of nothing and the ability to store water in reservoirs for use in the future. Such a problem is ideally suited for formulation as a multi-stage stochastic linear programme, as this allows both the dynamic and uncertain nature of the problem to be captured.

In collaboration with National Power plc, we have developed a model for the electricity generation network of South/ Southeast Brazil. This model has been formulated using our algebraic modelling language sMAGIC, and the resulting multi-stage stochastic linear programmes have been solved using our implementation of Benders Decomposition which incorporates sampling techniques to allow the solution of very large-scale problems and the solution of "folded" trees.

The following section describes our model of the electricity generation network, discussing the thermal plants, the transmission network and the hydro-electric model. Section 5.3 discusses the inflow model and how it uses historical data to generate the scenario tree or scenario graph for folded problems. The combination of all parts of the model are presented in Section 5.4 as implemented using our algebraic modelling language sMAGIC.

5.2 Generation Network Model

Within any electricity generation network there are several types of generation plants. These include thermal plants that use fossils fuels, i.e. coal, gas and oil, nuclear plants and hydro-electric plants. Wind and solar power are also used, but in the majority of national or regional electricity generation networks these provide a negligible amount of energy. Electricity is relatively difficult to store, the main way available being pump-storage which can be very inefficient, and so the main problems facing a decision maker are how much electricity to produce at any given moment, and when to generate it so as to minimise costs.

The objective in the model of the electricity generation network of Southern Brazil is to minimise the total cost of generation while meeting demand for electricity throughout the planning horizon, subject to meeting any water restrictions imposed on the reservoir network. The constraints of the model divide naturally into two networks; the transmission network and the hydro network.

5.2.1 Transmission Network

The transmission network models the delivery of electricity throughout the region such that the demands at local hubs are met. The demand at each hub can be met by supply coming from attached thermal or hydro generation plants, from other hubs within this regional network through the transmission network, or from the inter-connected generation networks of other regions. This results in a network consisting of nodes, hubs, with arcs joining some or all of them, the transmission lines and generation plants attached to hubs.

Each generation plant supplies its output to the local hub. If this electricity does not meet the local demand at a hub, or if it is not the most cost effective source of electricity, then it is possible to pass surplus electricity at one hub to another hub by means of the transmission lines between hubs. These lines are constrained by their physical capacity. In addition to electricity available from within the regional transmission network, it may be possible to "buy-in" electricity from an inter-connected neighbouring generation network. If this is the case, then the inter-connector can be represented by an extra thermal plant in the model. A typical transmission network is given in Figure 5.1.

In this example there are four separate hubs, Hi, i=1,. . . , 4, each with a demand Di. Hub H1 has three hydro-electricity plants, R1, R2, R3, attached to it and two thermal plants, T1, T2. Hub H1 is also connected to hubs H3 and H4, and can

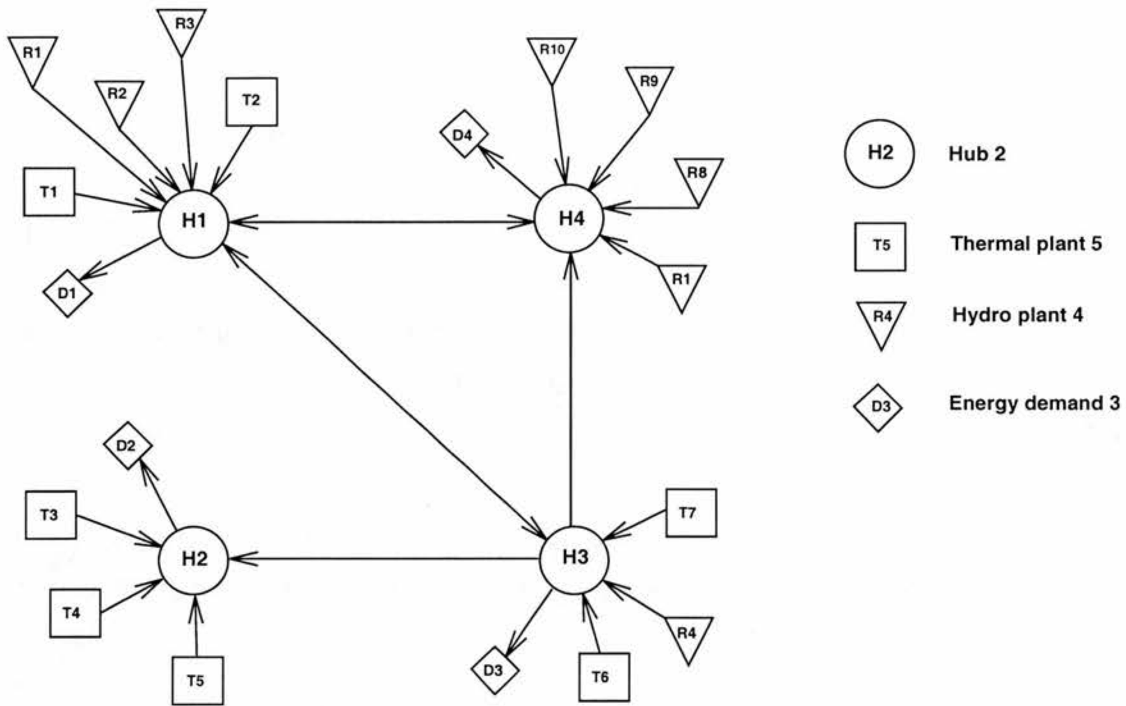


Figure 5.1: A Transmission Network

supply or draw extra electricity to or from these hubs. Similarly, there are three thermal plants connected to the network via hub H2. However, this hub cannot supply electricity to any other hub, but can draw electricity from hub H3 should it need extra power. Hubs H3 and H4 have local demands and generation plants associated with them and are connected so as to allow the exchange of electricity between them and the other hubs.

Individual thermal power plants are modelled as having both fixed and variable costs, both of which can vary in between periods. During the running of any type of generation equipment it is necessary to service existing turbines or replace faulty ones. This results in plants being unable to generate for periods. As our models have a period of at least one month, which may be far longer than the necessary maintenance period, we model this reduction in generation capacity by allowing the maximum capacity of any plant to be reduced from the actual maximum capacity. If the problem we were modelling were to be a short-term operating problem, the start-up and shut-down of the thermal plants would need to be taken into consideration. Before a thermal plants can become fully operational it must heat up. This results in there being a delay between when the plant is "switched on" and when it can generate at its current maximum rate. In the short-term this is a significant factor in whether a plant should ever be shut-down, rather than being left operating at reduced capacity so as to avoid having to start it from cold at some point in the near future. However, as our

model covers a period of at least one month, these extra considerations need not be explicitly included in the model.

In order to guarantee that the linear programmes generated from the model that incorporates this thermal transmission network are never infeasible, an additional "ghost" plant is included in the model. This plant is able to supply an infinite amount of electricity to any of the demand hubs within the transmission network. However, it is modelled as having an extremely high cost per MWh, and so it would only ever be used if an instance of the model had too little capacity to meet the demand for power. As such it can be thought of as being a penalty term for failing to meet demand.

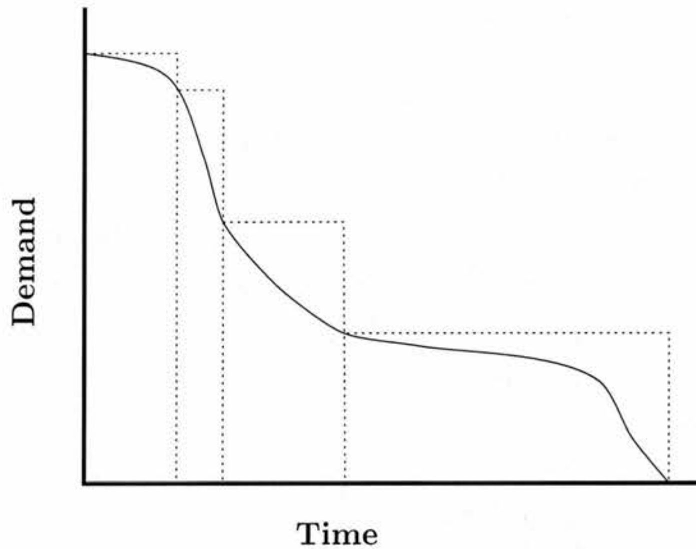


Figure 5.2: A Load Duration Curve

The demand for electricity is not constant throughout a day. Instead, there are peaks and troughs, mirroring how the various uses of electricity changes over each day. As our model is primarily designed for medium to long-term planning, it is not possible to capture these fluctuations in demand in chronological order. However, as the demand at peak times can be very high relative to low demand, it is necessary to account for these extrema. One way to model this is by the use of a *load duration curve*, an example of which is shown in Figure 5.2. This charts the amount of energy required in order of decreasing level of demand, so that non-contiguous periods with the same level of demand are next to each other. In our model we use a stepped approximation, as shown by the dotted line in Figure 5.2, to give a discrete load duration curve, which consists of four load blocks in the example given. Each of these blocks is specified by the level of demand within them and by their duration. The generation of electricity from either thermal or hydro-electricity plants is associated with one particular load block, and so the

variables denoting the generation and water discharges need to be indexed by load block, as well by thermal plant or reservoir.

5.2.2 The Hydro Network

The hydro network consists of the hydro-electric plants, the reservoirs that are attached to some of these plants and the various water courses that connect the plants and reservoirs. The hydro-electricity plants are connected to one another in series, usually following rivers down valleys. These different series can feed into each other and so networks of hydro-electricity plants are formed. Water that is released through or around one hydro-electricity plant is available to be used by the hydro-electricity plant immediately downstream from it.

Water sitting in a reservoir at the top of a valley can be used not only to generate electricity when it is passed through the turbines of that hydro-electricity plant attached to the reservoir, but also when it is passed through the hydro-electricity plants downstream. In addition the water can be stored in reservoirs and used to generate electricity in future time periods. This makes the decision of when to release water through a turbine far more complicated than the decision about when to switch on a thermal plant.

There are two main types of hydro-electricity plant. Those that are connected to a reservoir, and so have the ability to store water for future use, and those that have no reservoir attached to them. The latter are know as *run of river* hydro-electricity plants, and they have no option but to use the available water, whether from upstream discharges or from natural inflows, immediately, either to generate electricity or to spill it downstream. A third type of plant is possible, a *pump-storage reservoir.*

The role of pump-storage plants is to move water from one location to a reservoir upstream. This allows energy to be stored, in the form of water, which can be used to generate in the future. In a short-term operating model it might be the case that it is preferable not to switch off a thermal plant with a high start-up cost; the plant may provide a cheap supply of electricity once it is up and running. Instead the thermal plant can be kept running at a reduced level. This output is surplus to demand and so it is used to operate a pump-storage plant, hence allowing this cheap thermal energy to be stored for use in the future. This is likely to be the case in short-term operating models, but does not play an important part in medium to long-term planning models. The hydro network of Southern

Brazil does not have any significant pump-storage facilities, and this is reflected by their absence from our model.

Connecting the reservoirs and run of river plants are various types of water course. These may be natural rivers or man-made pipes. In either case there are limits on the amounts that can be passed along these in any given time period. For rivers there is a limit to how much water can be discharged into it before the extra water causes flood damage, whereas for pipes there is the physical limit of the bore of the pipe.

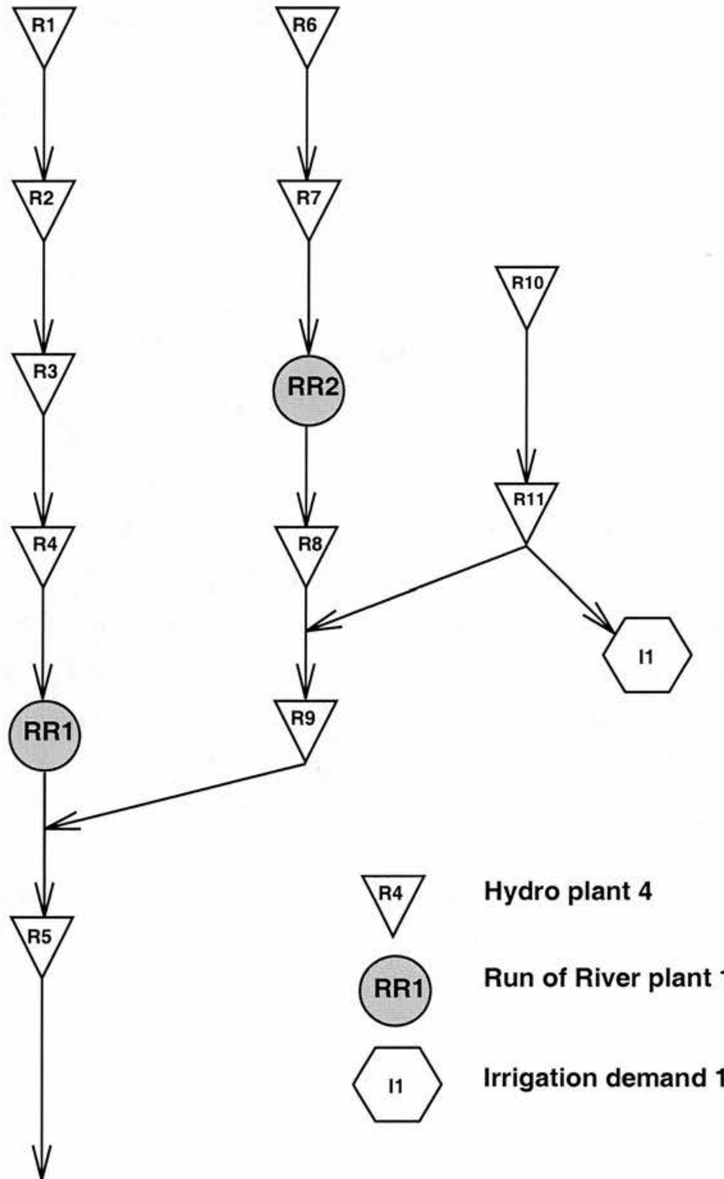


Figure 5.3: A Hydro Network

The model provides three ways of discharging water from a hydro-electricity plant. The first is water that is passed through the turbines. This is the water that is used to generate electricity. The other two channels are used to discharge water

downstream without passing it through the turbines. The first of these are called *spill channels*, and they allow a limited amount of water to be released without any generation. This might be necessary when a reservoir is nearly full and a wet weather scenario occurs. If the amount that needs to be spilled is in excess of the capacity of a spill channel then the extra water can be released through a *flood channel*, but these incur a penalty cost. An example of a hydro-network that consists of 11 reservoirs with hydro-electricity plants attached, 2 run of river plants and various water courses is shown in Figure 5.3. In addition, this example has a demand on water for agricultural use, which is modelled as a reservoir with the same capacity as the amount needed that must be filled in each time period.

Our model assumes that any water discharged from an upstream hydro-electricity plant is available immediately. In a short-term operating model such an assumption may not generally hold; the distance between hydro-electricity plants might be too long for any discharges upstream to reach a plant or reservoir downstream within a time period of half an hour. However, given the geographical dimensions of the region that we were modelling and the medium to long-term planning horizons that we were considering, this assumption is valid.

As with thermal plants, all of the turbines of a hydro-electricity plant may not be available all of the time. This is modelled by allowing the upper bound of the capacity of a plant to be lowered. In addition to the ability to store water for future use, the main difference between a thermal plant and a hydro-electricity plant is that the efficiency of a thermal plant is fairly constant with respect to the rate at which it is generating. This is not necessarily the case for a hydro-electricity plant. The reason for this is a property of hydro generation known as the *head effect*.

The head is the relative height of the water above the actual turbine. If a reservoir is nearly empty, then the water passed through the turbines does not fall through a great height, and so passes through the turbines with less pressure than discharges from the reservoir when it is nearly full. The head effect means that the efficiency of a turbine is the product of the head and the amount of water passed through it, and so is a non-linear term. The difference in efficiency can be significant when a reservoir's volume varies between being nearly full to nearly empty with any regularity. There is no variation in the head of the water passing through a run of river plant.

The head effect can either be modelled exactly, leading to a non-linear model, or by using an approximation. As we require a multi-stage stochastic linear

programme in order to use our implementation of Benders Decomposition, we have used the latter in our Brazilian model. The approximation is the mean of a range of efficiencies that a turbine exhibits operating with different head. A comparison between this model, that uses a linear approximation of the non-linear term, and its solution using Benders Decomposition, and a formulation of the same model that incorporated the non-linear term explicitly is given in Archibald *et al.* (1996) [1], where the non-linear model was solved using a variant of dynamic programming.

In this study the model was also part of the Brazilian electricity generation network. Even although the variations observed in the levels in the reservoirs were not negligibly small, the solutions obtained by the two solvers for the two different models were very close, to within 4% of each other. On this evidence we assumed that our linear approximation was reasonable for this model and the particular climate of Southern Brazil.

As the planning horizon of our model is finite, the final amounts of water in the reservoirs need to be taken into account. Ignoring the possibility that the water will be used after the end of the planning horizon of the model results in a model that places no value in keeping water beyond the final time period. However, most reservoirs are not de-commissioned at the end of the planning horizon, and so it is not desirable to run the reservoirs dry at the end of the planning horizon.

One method for valuing the remaining water is to directly generate a value function for this water. This can be achieved by solving a collection of multi-stage problems for a range of initial water levels in the reservoirs. The solutions to these problems can then be used to determine a value function for any terminal water in each reservoir. As the marginal value of water in a reservoir decreases as the reservoir's volume increases, assuming linear head effect, these terminal value functions are concave. Hence they need to be modelled by a piece-wise linear function, in order to keep the model linear. However, the work involved in generating terminal value functions for an electricity generation network with a significant number of reservoirs or time periods is significant. For this reason we model the terminal value of the water in the reservoirs by stipulating that the reservoirs are left with a certain percentage of their volume at the end of the planning horizon. Typically the reservoirs were left 50% full at the end. Instead of modelling this as an exact constraint, we allowed water to be "bought" at the end, but at an extremely high price. This prevents the sub-problems from becoming infeasible regardless of the decisions in the previous stages.

When solving various versions of the model for National Power plc, the main output in which they were interested was the system marginal costs. At any of the hubs of the transmission network, within any block of the load duration curve, the system marginal price is the most that the system would pay to meet any extra demand. These prices are determined by the unit cost of the most expensive thermal unit in operation that is attached to that hub, for that load block.

The deregulated electricity market in Brazil is to be a *spot-price pool* market. This means that individual generation companies bid in advance to supply electricity to their local transmission network. The price that they are paid for any accepted bid is the system marginal price of the network. Hence any investor, such as National Power plc, is interested in the future system marginal prices when they are trying to value generation assets in an electricity generation network.

Our model is primarily intended for medium to long-term planning horizons, and at present has been implemented with a time period of one month. This can be readily altered to allow longer time periods to be considered, but should only be reduced to shortener time periods if the assumptions discussed above are taken into consideration. An instance of the model is used to generate the sub-problems that make up a multi-stage stochastic linear programme. Different time periods in the model need not be the same length, e.g. a model might have a one month first period, three months in the second period and a year in the final period.

5.3 Inflow Model

The uncertainty in an electricity generation network with a large proportion of hydro-electricity plants comes from the future inflows. In addition to the transmission and hydro networks given above, the other major part of our electricity generation network model is the stochastic structure that models the future inflows. Historical data is used to generate a model that captures the full range of possible inflow patterns.

The term inflow is used instead of rainfall as the latter term is misleading in the context of hydro-electric generation. Within any given time period, the extra water that is available for generation can come from one of two sources; water passed from upstream and water from nature. The water from nature comes from two sources. Any rain or snow fall that enters the reservoirs directly, and any indirect sources of water. This might be from snow-melt in the spring or summer, or from rainfall further up a valley permeating through the ground to end up in the reservoir at a later date. In this respect we use inflows to mean any extra water from nature during a time period. Records of past inflows are used to generate the stochastic parameters that shape the scenario tree or graph for the multi-stage stochastic linear programmes.

This problem has been investigated in the more general setting of constructing a discrete scenario tree using historical data to generate the stochastic parameters. One method is to fit a time series to the data and use this time series to generate many future scenarios. From these, a tree of future scenarios is sampled. However, this method does not readily generalise to the generation of multi-stage stochastic linear programmes as graphs. A second method is to construct a tree or graph that matches certain statistical properties of the historical data. These might include various moments of the data, or one or more particular scenarios, such as an extreme event. These two methods are discussed in more detail below, followed by a discussion of the method that we use.

Sampling from the set of all possible futures provides one way of generating the scenario tree for a multi-stage stochastic linear programme. First a model, typically a vector auto-regressive time series, is fitted to the historical data. Using this model, large numbers of future scenarios are generated as they are required. The correct form for the vector auto-regressive model is selected after interpreting the auto-correlation functions of the historical data. For inflow data it is likely that the best model is auto-regressive with a short immediate memory

and additionally has a strong seasonal correlation. Using the fitted model, large numbers of forecasts can be generated, and a scenario tree sampled from these.

An example of this type of model, for the generation of a portfolio management problem, is presented in Chen *et al.* (1997) [16]. They formulate their model in two stages. Firstly they construct a vector auto-regressive data process that is fitted so as to capture specific underlying economic indicators, using historical economic data. This is a continuous state space, discrete time model that must have a time step that is at least as short as the time stages of any multi-stage stochastic linear programme generated from it. On top of this data process, a coefficient process is defined. This transforms these economic indicators into returns on stocks and shares for any time frame that is the same or coarser than that of the data process, e.g. the data process might be defined on a daily basis, whereas for a multi-stage stochastic linear programme that has monthly time stages. In this case the coefficient process transforms an instance of the data process into a series of monthly coefficients.

To generate a scenario tree, firstly a shape for the tree is specified, and then the data process model is run for each path through the tree. Starting with supplied initial conditions, a one step forecast is made. Its output is used as initial conditions for the next step. This is done recursively along a path as many times as there are data process time steps in the length of the path. For each path of the scenario tree, the transformation from data process to coefficient process is applied, resulting in the coefficients that are used to generate the individual linear programmes of the sub-problems in the scenario tree.

In Høyland (1998) [41], Høyland & Wallace present a method based on non-linear programming that generates a limited number of discrete outcomes which attempt to capture the key statistical properties of the underlying stochastic parameters of a multi-stage stochastic programme. This method first calculates summary statistics from the historical data of the random variables. A discrete distribution is then fitted that tries to capture these statistical properties. In order to fit such a model, the method solves a non-linear optimisation problem that tries to minimise the sum of the squares of the deviations between the actual statistical properties and those of the constructed discrete distribution.

Typical properties that the constructed distribution will capture include the expectations, the standard deviations, the skewness and the kurtosis. Particular events, such as a worst case scenario, can be included in the specification of the optimisation problem, and so the discrete distribution that is generated will

include these events explicitly. As the resulting optimisation problem is non-linear and probably non-convex, it is unlikely that a global solution will be found. While the optimal value of the problem is zero, the value at a local optimiser provides a gauge on how close the solution is to being optimal. Starting a solution method from different initial solutions, and comparing the solutions obtained should give some indication of the stability of a particular solution. In [41], this method is used to obtain a scenario tree for an asset liability management problem with four asset classes, and six branches per stage.

The method used in our model attempts to reduce the scale of the data analysis problem. Our models for Southern Brazil typically consist of 30–50 reservoirs. Monthly historical data is available for 64 years for each reservoir. However, there is a strong correlation between the reservoirs. This is probably due to their close geographical proximity to each other. Therefore we model the problem as having one state variable that records the total rainfall over all the reservoirs for each month of the year.

The distribution of this variable is divided into three parts, the middle and two tails. The method tries to construct a model with three scenarios per stage which capture this information. The reasons for this three point discrete distribution are twofold. Firstly, as we our implementation of Benders Decomposition uses importance sampling, we want to generate multi-stage stochastic linear programmes that have very many stages and a relatively small number of scenarios per stage. The second reason is that the historical data showed that the bulk of the distribution of the inflows was fairly constant with occasional outliers. These suggested the type of discrete distribution that we used.

Having constructed a total inflow variable from the historical data, its distribution was divided so that the tails consist of the top and bottom 10% of the historical data. A count is then taken of the number of transitions from one of the three parts of the distribution in one month to any of the three parts of the distribution in the following that occur in the historical data. Normalising these tallies gave us the transition probabilities for our model. To calculate the inflow patterns for a particular scenario in any month, the rainfall for a particular reservoir when the total variable fell within this scenario for the given month is tallied. These totals were averaged to give the inflow patterns for a particular scenario in any given month.

5.4 sMAGIC implementation of model

The two parts of the model for the electricity generation network of Southern Brazil are brought together in a formulation of the problem that uses the algebraic modelling language sMAGIC. Using this formulation of the model, instances of the problem of various sizes have been generated for solution and analysis by National Power plc. sMAGIC provides a natural framework in which to formulate our model of the problem as it has been designed specifically to facilitate the creation of multi-stage stochastic linear programmes.

The model was used to generate "folded" multi-stage stochastic linear programmes. Our implementation of Benders Decomposition can take advantage of this folded structure. A full version of the mathematical model can be found in Appendix A, and a listing of the sMAGIC formulation of this model can be found in Appendix B. The folding of multi-stage stochastic linear programmes that exhibit Markovity is discussed in more detail in Chapter 3.

Recursive definitions allow linear programmes to be formulated so that they may depend upon parameters of various types. Thus parts of a model that are similar can be formulated once and by calling this model with different parameters, different versions of the model are generated. If a model does not depend upon any parameters, it will give the same model whenever it is called. This has the advantage of allowing various parts of a large model to be formulated separately and only brought together when the final problem is generated.

The part of the sMAGIC formulation given in Figure 5.4 shows the start of the definition of a recursive model called `COST`. This is the general model for one sub-problem of our Brazilian model. It combines the reservoir network and the transmission network, and is called using parameters to define which sub-problem is being called.

```
MODEL COST(time, scenario, old_level)
parameter vars old_level[n_rvr];
parameter data scenario, time;
.
.
.
END MODEL COST
```

Figure 5.4: sMAGIC:Model Declaration

The model `COST` is defined as having three parameters; `time`, `scenario` and `old_level`. These parameters fall into two types. The first two are called *data*

parameters and are used to define a particular instance of a model. Whenever the model COST is called from within another model using different values for these data parameters, a new instance of the model that uses these parameters is generated. Whenever the model COST is invoked with the same values for these two parameters anywhere else within another model, a new instance of the model is not generated, but instead a link is made from the model instance that called the model to the existing instance of the model COST.

The third parameter in the definition of COST, old_level, is different from the previous parameters. Whereas data parameters are used to define a unique instance of a model, *variable parameters*, such as old_level, define which variables are common to both the model COST and the model from which the model COST has been called. When a call to an instance of an existing model is made, the variables that are passed in as parameters may have different names within the two models where they occur. This requires sMAGIC to generate a mapping from the variable names in the calling model to those in the model that is called.

The use of models means that sMAGIC can build up formulations of linear programmes from smaller parts. This allows similar parts of a large problem to be specified once as a model, and then re-generated with slight variations by calling the model with different data parameter values. The variations between different instances of a model are achieved by the use of definitions within a model being dependent upon the values that data parameters take when the model is called. The extract from the sMAGIC specification of the Southern Brazil model shown in Figure 5.5 demonstrates how constraints can be defined conditionally.

```
CONSTRAINTS
 FOR {r IN R}
 {if (level_max[r] <> level_min[r])
   {if  (time = n_stages)
    {rvr_cap{r}:(level_min[r]+level_max[r])/2 <= level[r] <=level_max[r]}
   else
    {rvr_cap{r}:level_min[r] <= level[r] <=level_max[r]}
   }
 };
```

Figure 5.5: sMAGIC:Constraint Declaration

This piece of code defines the bounds (constraints) on the amount of water that can be kept in a reservoir r, one member of the set of all reservoirs R. The bounds are conditional on two things. Firstly, if the reservoir has no capacity, then no bound is generated, as there is no variable level[r] to model the amount of water

in this reservoir. This is the case if the maximum equals the minimum level of a reservoir, as is the case in a run of river hydro-electricity plant. If the reservoir does have storage capacity, then different bounds are generated depending on the value of the data parameter `time`. If the model is called in the final time stage, then the reservoir must be at least half full, as in this example, whereas in other time stages all of the water in the reservoir is available for generation.

While models can be used repeatedly to generate similar parts of the problem, the use of parameters allows them to do so flexibly. The use of conditional elements of the model, depending upon either data parameters or the actual data of an instance of the model, allows models to encapsulate more detailed models. Additionally, the extract from the sMAGIC specification of the Southern Brazil model given in Figure 5.6 shows how compact formulations of multi-stage stochastic linear programmes can be also specified using model calls, and in particular how these calls can be used recursively.

```
OBJECTIVE
if (time = n_stages)
 {minimize COST = ((sum{u IN U} (thml_eff[u]*fuel_cost[u] + 2.5)*thml_gen[u]
           + sum{l IN L, b IN B} (ghst_gen[b,l]*ghst_cost[b,l]
           + sum{r IN R} (conv[r]*defct[r]*ghst_cost[b,l] )))
 }
else
 {minimize COST = (sum{u IN U} (thml_eff[u]*fuel_cost[u] + 2.5)*thml_gen[u]
           + sum{l IN L,b IN B} (ghst_gen[b,l]*ghst_cost[b,l]))
           + sum{new_scenario IN W} (
            prob[time,scenario,new_scenario]*COST(time+1,new_scenario,level) )
 };
```

Figure 5.6: sMAGIC:Objective Declaration

Here the objective of the model `COST` is defined as having two forms depending upon whether the model is called in the final time stage of the problem or a prior time stage. In the final stage of the problem, the objective consists of three parts; the costs due to thermal generation, the costs due to the use of the ghost plant and the costs due to meeting the terminal water constraints. In previous stages of the problem, the objective consists of the thermal generation costs, the cost of the ghost plant and also the expectation of the future cost, i.e. the recourse function. This is the cost in future time periods, and it depends directly upon decisions taken in this instance of the model. The recourse cost is formulated by the use of a recursive call to the model `COST` itself. For each of the possible scenarios that might occur in the next time stage, the cost of the sub-problem that represents that scenario is weighted by the probability of it occurring and summed to give the recourse cost in this model.

For each call to the model COST, sMAGIC checks to see if an instance of the model with the same data parameters has been invoked previously. If so, then a link is made between the instance of the current model and the existing instance of the model that has been called as part of its objective function. If the instance of the called model does not yet exist, then it is generated, and a link made to it from the instance of the current model. Attached to the links between instances of models are both a probability and a variable mapping. The first of these is the likelihood of the second model occurring in the future when the first model is current, i.e. a transition probability. When one model invokes another model, whether it is another instance of itself or an instance of a different model, any variable parameters that are part of the definition of the called model may have different names in the instances of the two models. However, despite the different names, these variables are common to both models, and so sMAGIC generates a mapping that relates variables in one model instance to their counterparts in the other model instance.

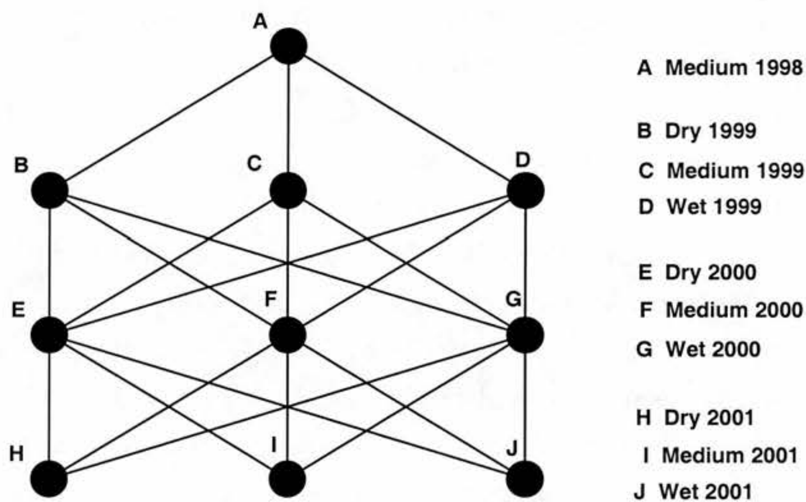


Figure 5.7: A Folded Reservoir Problem

Consider the "folded" multi-stage stochastic linear programme represented by the graph given in Figure 5.7. This problem has three time periods with three scenarios in each period. Assuming that the formulation contains specifications similar to those given in Figure 5.4–5.6, sMAGIC would generate this problem in the following recursive manner. Three passes through the specification of the model are performed. During the first pass the syntax is parsed and in the second pass the individual models are generated as parameterised procedures. The third pass is performed in two phases. During the forwards pass the links between the instances of the models are generated. This gives the graph shown above in

Figure 5.4. During the backwards pass the actual linear programmes associated with each of the nodes in the graph are generated.

If the problem starts with an average inflow pattern and the reservoirs half full, then the first sub-problem, A, would be associated with an instance of the model COST called with `scenario` set to "medium", `time` set to one and the vector `old_level` set to half the storage capacities of the reservoirs. While processing this instance of the model, sMAGIC calls the model COST three times, one for each of the first stage problems, as this model forms part of the objective function of the model being processed. Each of these calls is made with `time` set to two, `scenario` set to one of the three possible first stage scenarios, "dry", "medium" and "wet" and with `old_level` set to the variables for the levels in the reservoirs of the model instance associated with the root sub-problem. These are the amounts of water that are passed on to the future sub-problems. This process is repeated when the instance of the model associated with sub-problem B is processed, resulting in in the generation of model instances associated with sub-problems E, F and G.

Later during the processing of the problem, when the model instance for sub-problem C is processed, it has to generate instances of the model for sub-problems E, F and G. However, since these instances of the model have been created when the model instance of sub-problem B was processed, they need not be generated again. Instead, the current instance of the model for sub-problem C is linked to the existing instances of the model for sub-problems E, F and G, with probabilities and mappings attached to each of the new links. In Figure 5.7 this relates to the formation of the arcs from node C to nodes E, F and G, given that the arcs from B to these nodes have been generated previously. In such a manner the remainder of the problem is generated. When any of the model instances associated with the leaf nodes are processed, no further calls to the model are made, as in this case the alternative objective function is used, as given in Figure 5.6.

In such a way sMAGIC generates a folded version of the multi-stage stochastic linear programme specified. Should it be necessary, this folded representation can be "unfolded" to give the problem in the standard MSLP formulation. However, in its unfolded form of a tree, the underlying Markov structure of the multi-stage stochastic linear programme is lost, and so cannot be taken advantage of. Alternatively, the model can be formulated so that a tree is generated instead of a graph. This is the case if each instance of the model COST is called with a distinguishing data parameter, such as a distinct node number in the event tree. When parsing such a problem, sMAGIC generates a unique instance of the model for each call, as each call is made with different parameters.

5.5 Conclusions

Constructing our model of the electricity generation network of Southern Brazil had two main aims. Firstly to provide a flexible set of test problems with which to investigate our Benders Decomposition code, using folded multi-stage stochastic linear programmes and importance sampling. The second aim was to provide National Power plc with a decision support tool for the valuation of hydro-electricity plants. Specific results relating to the individual features that constitute our first aim can be found in the previous chapters of this thesis.

The problems generated for National Power plc had a ten year planning horizon, as the future prices for electricity in the newly deregulated market are the mechanism that drive the price of generation assets. As the monthly changes in the pool price for electricity was of interest, we constructed several problems with monthly time stages, resulting in problems with up to 120 stages. We generated several instances of these problems, each with different numbers and types of generation plants being built over the planning horizon. This captured the different amounts and types of generation plants that might be built by the many different companies that the newly deregulated electricity market of Brazil is likely to attract. The resulting problems have numbers of hydro-electricity plants that vary form thirty to fifty and numbers of thermal plants that range from thirty to fifty three. The different building schedules also included the construction of different sizes of hydro-electricity plants.

The benefit of constructing a model that contains such short time periods up to the end of the planning horizon can be questioned. A modeller's ability to estimate the distribution of the uncertainties in a stochastic model, the future inflows in this case, is itself uncertain. This calls into question the value of modelling a medium to long-term planning problem with an equal period length for each stage. However, as it is the seasonality of hydro-electricity generation that has the largest affect on the spot-price of electricty, the periods were of one month throughout the planning horizon.

Both the ability to estimate the distributions in the final stages and the use made of the results have doubts attached to them. The results from a multi-stage stochastic linear programme are intended to give a feel for the future costs associated with the model, and to give operational decision in the short-term that hedge against the uncertain future. In addition to this the estimated distributions of the uncertain parameters become less certain the further into the future they are projected.

National Power, plc, one of the major electricity producers in Europe, now have a decision support package that provides them insights into the likely future prices in hydro-electricity dominated generation markets. An area in which they previously had very little experience or understanding.

The model of Southern Brazil demonstrates the usefulness of sMAGIC. It allows a multi-stage stochastic linear programmes to be formulated in a compact form. It can also be used to model problems that exhibit Markovity between the stochastic parameters in different stages. In addition it is capable of modelling many other types of mathematical programmes.

The models generated for National Power plc are examples of very large-scale multi-stage stochastic linear programmes based on real-world problems. The results in the previous chapters show that these problems can be solved in a reasonable time, to a pre-specified accuracy by our implementation of Benders Decomposition. Such detailed problems could not be solved without exploiting their Markov structure by "folding" them. For the very large problems, there is also a need to use sampling techniques. These are ideally suited to the folded problems, as they allow the quicker distribution of future cost information through the sharing of cuts.

Chapter 6

Conclusions & Further Research

This thesis has examined the process of formulating and solving multi-stage stochastic linear programmes. One of the motivations for this has been to study the real problem of generating hydro-electricity in Southern Brazil for National Power plc. While studying this problem in particular, this thesis has also examined the various stages involved in formulating and solving more general multi-stage stochastic linear programmes.

The interest of National Power plc in multi-stage stochastic linear programmes reflects the more general increase in attention which they have received in the last decade. This has happened mainly within the academic community, although industry, especially the utility and financial services, is increasingly investigating the benefits of including uncertainty in planning models.

To this end we have formulated a large-scale model of the electricity generation network in Southern Brazil. This model is a very large multi-stage stochastic linear programme. It includes up to fifty thermal and fifty hydro-electric generation plants and covers a planning horizon of up to ten years. The sub-problems having 93 constraints and 520 variables, and the problems consists of 120 stages with three scenarios branching from the non-leaf nodes. The deterministic equivalent of this problem has 2.5×10^{59} constraints and 1.4×10^{60} variables. Despite its size, such problems are made tractable by exploiting both the basic structure of the multi-stage stochastic linear programme and the Markovity of the underlying stochastic parameters.

National Power plc were primarily interested in gaining an understanding of how the price of electricity is determined in a system with a large proportion of hydro-electricity generation capacity. The predicted spot-price of electricity is one factor that is used to determine the financial value of the generation assets in

the system. Due to the uncertain nature of the future inflows, National Power plc wished to develop a decision tool that was designed to capture this uncertainty in its model. Our algebraic modelling language and solver, which incorporates sampling techniques, along with the option of "folding" multi-stage stochastic linear programmes form the key components of this decision support tool. While they have been developed specifically with the Southern Brazil model in mind, their facilities extend to the formulation and solution of more general multi-stage stochastic linear programmes.

In Chapter 2 we survey the methods currently available to facilitate the formulation of multi-stage stochastic linear programmes. One of the factors that has prevented the broader use of multi-stage stochastic linear programmes has been the difficulty in formulating the problems. While the development of algebraic modelling languages to their current level of functionality has lead to the widespread use of linear programming as a modelling tool for decision making, there is still no commercially available algebraic modelling language that has the specific capability to model multi-stage stochastic linear programmes.

The currently available modelling systems fall into two categories, algebraic modelling languages and ad hoc methods. The second of these are formalised and more rigorous implementations of the methods previously used by some modellers to generate multi-stage stochastic linear programmes. They suffer from the same problems as previous ad hoc methods by being cumbersome and inflexible.

The other category consists of the proposed extensions to existing algebraic modelling languages in order to allow them to be used to specify multi-stage stochastic linear programmes. However, the resulting languages are in some ways restricted because they are extensions of existing algebraic modelling languages. We have developed the novel algebraic modelling language sMAGIC specifically with the aim of formulating multi-stage stochastic linear programmes, rather than being constrained by an existing language. Our model of the electricity generation network of Southern Brazil was developed using the recursive definition of sub-problems available within sMAGIC. In this was the model is compact, modular and generates the final instance of the problem as a model link graph. In this form the Markov structure of the problem can be exploited by our solver.

A possible extension to sMAGIC would be to integrate it directly with our Benders Decomposition software. At present many commercially available algebraic modelling languages are integrated with a solver. This allows a modeller to formulate their model and then solve it. Once this problem has been solved,

it is then possible to analyse the results in an environment that is similar the one in which the model was originally formulated. Another, more minor, extension to sMAGIC would be to have the option of generating problems in different output formats. The most useful of these would be the SMPS format.

In Chapter 3 we examine the Benders Decomposition algorithm that underlies our solver. Our implementation of Benders Decomposition performs considerably better than the most common direct solution method, the simplex method, and it has been implemented in parallel and obtains near linear speedups on a network of workstations. Several of the extensions to the basic algorithm have proved to lead to general improvements in the solution times for a range of problems, while others have proved only to be beneficial for particular problems. In addition we have shown how our implementation required very little modification to allow folded problems to be solved. We use small instances of our model of Southern Brazil to compare the performance of the algorithmic extensions and the parallel version of the solver.

A future development for our solver would be to try to increase its numerical stability. During a run of the Benders Decomposition algorithm large numbers of cuts can be added to a sub-problem. However, after adding very few cuts it is possible for there to be linear dependencies between the constraints. This can hamper the numerical stability of the solver in subsequent iterations. One way to avoid this problem would be to use a pre-solver on the sub-problems prior to starting the algorithm. This helps to remove the problem associated with linear dependencies between constraints. An alternative would be to remove added cuts as they become old, or are seen to never be active.

In Chapter 4 we examine ways of incorporating sampling techniques with the Benders Decomposition algorithm to allow very large-scale multi-stage stochastic linear programmes. The main problems that National Power plc wished to investigate were extremely large instances of the Southern Brazil model with up to 120 stages. This size of problem is beyond the capabilities of both direct solution methods and the basic and extended Benders Decomposition algorithm. However, generating these problems as folded multi-stage stochastic linear programmes makes them tractable using sampling techniques in conjunction with our Benders Decomposition solver.

As the importance sampling algorithm progresses new cuts are added to sub-problems. If the additive approximation to the cost function that is used to estimate the importance distribution is not updated with these new cuts, then

this new and improved information is never capitalised upon. Hence we compared two methods of updating the approximation to exploit this new information. The first updating method uses multiple linear regression after each new sample to update the additive approximation using the most recent observations, while the second uses the additive approximation for a specified number of samples and then recomputes the approximation from scratch. The results show that the performance of the multiple linear regression method depends upon the size of the additive function being constructed, with larger problems requiring more observations to obtain a significant improvement in the additive approximation. However the re-calibration method is more dependent on the variance of the underlying cost function that is being approximated. As this variance is not known before the algorithm starts, this makes the re-calibration method less useful when dealing with a new problem for the first time.

Our sampling algorithm currently uses an additive model to approximate the value function of multi-stage stochastic linear programmes. In the case of our hydro-electricity management problems this has proved to be a good guide for the importance sampling. However, for other forms of MSLPs this might not be the case. For example, if the penalty costs for flooding are included in the hydro-electricity model, then the additive model is unlikely to be a good approximation. It is unlikely that one general approximation will be adequate for all multi-stage stochastic linear programmes.

The computational work involved in constructing and sampling using an approximation is very small compared to the solution of sub-problems. It would therefore be possible for an algorithm to use one approximation, updating its coefficients as samples are taken, while also having one or more other approximations of different forms which are also updated as samples are taken. These unused approximations could be compared with the results from the current approximation after each sample has been performed. The algorithm could then change to the best approximation for the MSLP being solved.

At present the parallel implementation of our Benders Decomposition solver does not incorporate importance sampling. There are two possible ways to do this; one for folded problems, and one for unfolded problems. The case for solving unfolded problems is the more straightforward extension of our current parallel solver. As the MSLP is divided into approximately equally sized sub-trees which are placed on individual processors, instead of sampling one path through the tree at each iteration, the parallel sampling algorithm would sample one path from each sub-tree. This results in an algorithm which is likely to have good load

balancing. Provided the sub-trees are allocated so that the probability of any one occurring is reasonably similar, then there is likely to be very little bias in the sampled statistics.

However, the sampling algorithm presented in Chapter 4 requires the sharing of cuts across stages in order to be able to have any hope of obtaining a reasonable estimate of the actual solution to a very large problem. Cut sharing occurs naturally when solving folded problems, and so it would be beneficial to implement a parallel version of our folded Benders Decomposition algorithm. The main obstacle with this parallel algorithm is the allocation of sub-problems to processors.

The large sub-trees allocated to processors in our current parallel algorithm are folded onto one another when the multi-stage stochastic linear programme is formulated as a folded problem. Therefore it would be counter productive to unfold these sub-trees so as to solve the problem in parallel. Instead a different way of allocating sub-problems to processors is required. Two possible ways of doing this are either to put all sub-problems in one stage on the same processor or to put one sub-problem from each stage on a different processor. Of course, it is unlikely that either way would be able to divide the sub-problems evenly between processors, and this would cause some imbalance.

The more prominent change that either of these methods for allocating sub-problems to processors will cause is that the number of messages passed between processors will be much larger than for the unfolded parallel algorithm. This is mainly due to the fact that when an unfolded problem is solved, each sub-problem is solved once during each pass of the algorithm. When a folded problem is solved each sub-problem is solved several times during one pass of the algorithm. This is because most sub-problems in the folded version of the problem have more than one parent. This added level of connectivity in the graph make it more difficult to allocate sub-problems to processors without increasing the need for decisions and cuts to be passed between processors. When the parallel algorithm is implemented on a network of workstations this is likely to have a large affect on the efficiency of the algorithm.

As folded problems are far more compact, it is not impractical to have an entire copy of the whole problem on each processor. In this case, sampling could be done concurrently on each processor, with cuts communicated to other processors at regular intervals. As the number of cuts being added to sub-problems would be greater than during the serial implementation of the algorithm, as there are cuts

from the other processors in addition to those from samples performed on the same processor, this implementation would require a good method of selecting how to delete old or inactive cuts.

In Chapter 5 we look at both the details of our Brazilian model and how it is implemented using sMAGIC. The method of generating the distribution of stochastic parameters used in this model of the electricity generation network of Southern Brazil provides a flexible way to generate multi-stage stochastic linear programmes with stages containing varying time periods. However, this method does not attempt to create a distribution that has similar moments to those of the historical data. One way to do this is the method described in Fleten, Høyland & Wallace (1997) [27]. This method uses nonlinear programming to fit the summary statistics of either the empirical data to a multi-stage stochastic linear programme of a desired form. However no use has been made of this method to fit data to a folded model.

Our work has been motivated by the need of National Power plc to develop a decision toll with which to gain a broader understanding of electricity generating systems with a large proportion of hydro-plants. However, our investigations have resulted in a decision support toll that is geared towards the formulation and solution of more general multi-stage stochastic linear programmes. The possibilities for future research given above for each of the component parts of the system, the algebraic modelling language, both the serial and parallel solvers, the method for updating the approximation used in the sampling techniques, make it a more robust, flexible and powerful decision support tool for analysing decisions made under uncertainty.

Appendix A

Full Hydro Model

The mathematical formulation of the hydro-electricity generation model is given below. Following an explanation of the various parts of the model there is a list of the notation used.

$$C^t_\omega(\mathbf{h}) \quad = \quad \min_{\tilde{\mathbf{h}},\mathbf{x},\mathbf{s},\mathbf{f},\mathbf{z}} [\Gamma(\mathbf{y}) + P(\mathbf{f}) + G(\mathbf{g})] + \begin{cases} \sum_{\omega' \in \Omega_{t+1}} \pi_{\omega'|\omega,t} C^{t+1}_{\omega'}(\tilde{\mathbf{h}}), & t < T \\ \sum_{r \in R} \mathbf{c}^c \mathbf{z}(r) - \sum_{r \in R} \tilde{\mathbf{h}}(r)\epsilon, & t = T \end{cases} \tag{A.1}$$

The objective function (A.1) is composed of immediate costs and future costs. The immediate costs are composed of the cost of thermal generation, the penalty for flooding and the cost of meeting demand for electricity by using the ghost plant. The future costs are composed of the expectation of the costs in the next stage, in all but the final stage. In the final stage the future costs are the cost of "buying" water to meet constraints on the final levels in the reservoirs, less the value of any scrap water left over.

$$\text{Transmission} \qquad \mathbf{d}^t_{\omega,l} = E(\mathbf{y}_l) + T(\mathbf{x}_l) + \mathbf{g}(l), \ \ \forall l \in L \tag{A.2}$$

There are two sets of constraints. The first models the balance between the demand and supply of electricity. Equation (A.2) consists of a constraint for each load block. This constraint balances the demand for electricity, which depends upon the stage, scenario and load block, with the various sources of generation. Electricity can be supplied by thermal, hydro or ghost generation.

Thermal generation, $E(\mathbf{y}_l)$, uses either nuclear or fossil fuels to provide electricity and the costs for this type of electricity is mainly composed of the price of the

fuel used, $\Gamma(\mathbf{y})$. Electricity can also be generated by the hydro-electric plants, $T(\mathbf{x}_l)$ Water, either from the flow of a river or from a reservoir, is passed through a turbine to generate electricity. This source of electricity is essentially free. The final source of electricity is from the ghost plant. This is a fictional thermal generation unit. It has a very large capacity, but its unit cost is also very high. Its purpose is to ensure that the model can meet all levels of demand for electricity, i.e. that the model is never infeasible.

$$\text{WaterBalance} \quad \mathbf{q}_\omega^t - \sum_{l \in L} M\left[\mathbf{x}_l + \mathbf{s}_l + \mathbf{f}_l\right] - I_\omega^t = \begin{cases} \text{if } \mathbf{h}_\mathrm{L}=\mathbf{h}_\mathrm{U}, & \begin{cases} 0, & t < T \\ -\mathbf{z}, & t = T \end{cases} \\ \text{if } \mathbf{h}_\mathrm{L}\neq\mathbf{h}_\mathrm{U}, & \begin{cases} \tilde{\mathbf{h}} - \mathbf{h}, & t < T, \\ \tilde{\mathbf{h}} - \mathbf{h} - \mathbf{z}, & t = T \end{cases} \end{cases} \tag{A.3}$$

The second set of constraints, equation (A.3), models the movement of water through the reservoir and river network. The water remaining in a reservoir at the end of a stage equals the water in the reservoir at the start of the stage plus any water from nature or the releases from upstream reservoirs less any releases from this reservoir. Water can be released from a reservoir either through the turbines, $\mathbf{x}_l$, as un-penalised spillage, $\mathbf{s}_l$, or if the amount of spillage is too high, as penalised flooding, $\mathbf{f}_l$. As the reservoirs are connected along valleys, which can also feed into other valleys, water released from an upstream reservoir, either as turbine releases or as spilling or flooding, flows into immediately downstream reservoirs.

As some hydro-electric generation plants are not connected to a reservoir, and are termed *run of river* plants, these are modelled as having reservoirs with no storage capacity, $\mathbf{h}_L = \mathbf{h}_U$. In the final stage of the model a certain amount of water must be left in the reservoirs. While this stage marks the end of the planning horizon of the model, the hydro-electricity generation network will exist beyond this period in time. Therefore it is not possible to run the reservoirs dry at the end of the planning horizon. The reservoirs must be left with a specified amount of water in each reservoir. We implement this by allowing water to be "bought" in the final stage, $\mathbf{z}$, but at a very high price, $\mathbf{c}^c$.

$$\text{ghost} \quad \mathbf{g}(l) \leq \mathbf{g}_U Hr_l^t, \quad \forall l \in L \tag{A.4}$$

$$\text{burn} \quad \mathbf{y}_l(u) \leq \mathbf{y}_U(u) Hr_l^t, \quad \forall l \in L, u \in U \tag{A.5}$$

$$\text{turbines} \quad \mathbf{x}_l(r) \leq ef \mathbf{x}_U, \quad \forall r \in R, l \in L \tag{A.6}$$

$$\text{reservoirs} \quad \left\{ \text{if } \mathbf{h}_L \neq \mathbf{h}_U, \quad \left\{ \begin{array}{rl} \mathbf{h}_L \leq \tilde{\mathbf{h}} < \mathbf{h}_U, & t \neq T \\ \mathbf{h}_L + pf(\mathbf{h}_U - \mathbf{h}_L) \leq \tilde{\mathbf{h}} < \mathbf{h}_U, & t = T \end{array} \right. \right. \tag{A.7}$$

There are also bounds on the ghost plant, (A.4), the thermal fuels plants, (A.5), the capacity of the turbines, (A.6), and the volume of reservoirs, (A.7). In the bounds on the ghost, thermal and hydro plants, the bounds exist for each load duration block, while there is only one per stage for those limiting the reservoir capacities. The amount of capacity for the ghost and thermal plants depends upon the duration of each load duration block, Hr_l^t. For hydro-electricity plants that have a reservoir, a percentage, ef, of the volume must remain.

$$\Gamma(\mathbf{y}) = \sum_{u \in U} \left[c_u^y th_u \sum_{l \in L} \mathbf{y}_l(u) \right] \tag{A.8}$$

The cost of thermal electricity, $\Gamma(\mathbf{y})$, is derived from the amount of fuel burnt at each thermal unit, $\mathbf{y}_l(u)$. Each thermal plant has a particular thermal efficiency, th_u, and a cost derived from the fuel it uses, c_u^y.

$$P(\mathbf{f}) = \sum_{l \in L} c_l^f \sum_{r \in R} \mathbf{f}_l(r) \tag{A.9}$$

Each reservoir has a finite capacity. Beyond this, a reservoir can spill a small amount of water, $\mathbf{s}_l$, at no cost, but any additional water released into the local environment, $\mathbf{f}_l$, is penalised, as given in (A.9).

$$G(\mathbf{g}) = \sum_{l \in L} c_l^g \mathbf{g}(l) \tag{A.10}$$

The use of the ghost plant is charged at a high rate, c^g, and summed over each load duration block before being added into the objective function, (A.10).

$$E(\mathbf{y}_l) = \sum_{u \in U} \mathbf{y}_l(u) \tag{A.11}$$

$$T(\mathbf{x}_l) \;=\; Hr_l^t \sum_{r \in R} [\varpi_r \mathbf{x}_l(r)] \tag{A.12}$$

The total amount of electricity supplied by thermal and hydro generation are given in equations (A.11-A.12). The factor ϖ is the conversion of water released through the turbines into electricity.

The model presented above makes use of the following notation:

$\mathbf{x}_l$ is the vector of r turbine releases in load block l,

$\mathbf{y}_l$ is the vector of u thermal burn in load block l,

$\mathbf{g}(l)$ is the ghost burn in load block l,

$\mathbf{s}_l$ is the vector of r turbine spills in load block l,

$\mathbf{f}_l$ is the vector of r reservoir floods in load block l,

$\mathbf{z}$ is the vector of r reservoir deficits,

$\mathbf{h}$ is a vector of r initial reservoir levels,

$\tilde{\mathbf{h}}$ the final reservoir levels,

$\mathbf{d}_{\omega,l}^t$ is the demand for electricity in load block l, of stage t when in scenario ω,

$\mathbf{q}_\omega^t$ is the inflow in scenario ω in stage t, I_ω^t is the demand for water for irrigation in scenario ω in stage t,

$\mathbf{h}_L$ and $\mathbf{h}_U$ are the maximum and minimum reservoirs levels,

$\mathbf{g}_U$ is the maximum capacity of the ghost plant,

$\mathbf{y}_U(u)$ is the maximum capacity of thermal plant u,

$\Gamma(.)$ is the total cost of thermal generation,

$G(.)$ is the total cost of ghost generation,

$P(.)$ is the penalty for flooding,

$T(.)$ is the energy from turbine releases,

$E(.)$ is the energy from thermal plants,

c_l^g is the cost of using the ghost plant in in load block l,

c_u^y is the cost of using thermal unit u,

c_l^f is the penalty for flooding in load block l,

ϵ is the value of water remaining in a reservoir at the end

t is the time stage, ω' and ω are scenarios in stage $t+1$ and

M is the $r \times r$ incidence matrix for releases between upstream

th_u is the thermal efficiency of thermal unit u,

Hr_l^t is the number of hours in load block l, in stage t,

cv_r is the factor for conversion from turbine releases into electrical energy for reservoir r,

ef is the global efficiency of the hydro-turbines,

pf is the percentage of the maximum volume of water that must

Appendix B

sMAGIC Hydro Model

```
! sMAGIC model which produces NP's Brazil model.
MODULE sizes
! contains the problem sizes
DATA
n_rvr = 33, n_spill = 33, n_turb = 33,
n_therm = 23, n_ldc = 3, n_bus = 1,
n_stg = 12, n_scen = 3;
SET
R={1..n_rvr}, G={1..n_turb},
S={1..n_spill},L={1..n_ldc},
B={1..n_bus}, U={1..n_therm},
W={1..n_scen};
END MODULE sizes

MODULE fixed
! contains the problem coefficients
use module sizes;
DATA
days=31,
prob[n_stg,n_scen,n_scen]<< "da/probs",
ghost_co[n_bus,n_ldc]=(#n_bus*n_ldc)540,
ghost_mx=90000,
thef[n_therm] << "da/thef",
inflow[n_stg,n_scen,n_rvr] << "da/inflows",
fuel_co[n_therm] << "da/fuel_co",
conv[n_rvr] << "da/conv",
hrs[n_stg,n_ldc] << "da/hours",
gent_mx[n_therm] << "da/gent_mx",
turb_mx[n_turb] << "da/turb_mx",
lev_mn[n_rvr] << "da/lev_mn", lev_mx[n_rvr] << "da/lev_mx",
dem_factor << "da/dem_factor",
dem[n_bus,n_stg,n_ldc] << "da/demands";
END MODULE fixed

MODULE network
! map of the hydro incidence network and transmission
! network
use module sizes;
DATA
tin[n_turb] << "da/tin", tou[n_turb] << "da/tou",
sin[n_spill] << "da/sin", sou[n_spill] << "da/sou",
hbus[n_rvr] << "da/hbus", bus[n_therm] << "da/bus";
END MODULE network

MODEL COST(t,scenario,old_lev)
! model for a node in the graph.
use module network;
use module fixed;
use model COST;
parameter vars old_lev[n_rvr];
```

```
parameter data scenario, t;
DATA
S_in_H = 60*60, mill=1000000, thou=1000, epsilon=0.00001;
VARS
level[n_rvr], spill[n_spill,n_ldc], turb[n_turb,n_ldc],
burn[n_therm,n_ldc], thml[n_therm], ghost[n_bus,n_ldc],
def[n_rvr];
OBJECTIVE
if (t <> n_stg)
! Objective includes: cost of thermal generation, cost of ghost plant and
! future costs.
{minimize dummy = (sum{u IN U} (thef[u]*fuel_co[u] + 2.5)*thml[u] +
sum{l IN L,b IN B} (ghost[b,l]*ghost_co[b,l]))/thou
+ sum{nw_scenario IN W} prob[t,scenario,nw_scenario]*COST(t+1,nw_scenaerio,level)
}
else
! In final stage, objective also has cost of not leaving enough water in reservoirs
! less scrap value of water
{minimize dummy = ((sum{u IN U} (thef[u]*fuel_co[u] + 2.5)*thml[u] +
sum{l IN L,b IN B} (ghost[b,l]*ghost_co[b,l]
      + sum{r IN R} (mill*conv[r]*def[r]*ghost_co[b,l]/S_in_H)
))/thou - sum{r IN R,lev_mn[r]<lev_mx[r]} level[r]*r*epsilon)
};
CONSTRAINTS
!couple thermal burn across load slices
FOR {u IN U}
{tgen{u}: thml[u] = sum{l IN L} burn[u,l]},

! bound ghost plant
FOR {l IN L}
{ghmx{l}:sum{b IN B} ghost[b,l] <= ghost_mx*days*hrs[t,l]},

! water balance, per reservoir, per load slice.
! allowed to "borrow" water in the final period, to meet 50%
! full constraints.
! model differently for run of river plant.
FOR {r IN R}
{if (lev_mx[r] = lev_mn[r])
{if (t = n_stg)
{wbal{r}:  sum{l IN L}
 (S_in_H*hrs[t,l]/mill)*(
    sum{g IN G,tin[g]=r} turb[g,l]
    + sum{s IN S,sin[s]=r} spill[s,l]
  - sum{g IN G,tou[g]=r} turb[g,l]
  - sum{s IN S,sou[s]=r} spill[s,l]
 )
+ def[r]
= - inflow[t,scen,r]  }
else
{wbal{r}:  sum{l IN L}
 (S_in_H*hrs[t,l]/mill)*(
    sum{g IN G,tin[g]=r} turb[g,l]
      + sum{s IN S,sin[s]=r} spill[s,l]
  - sum{g IN G,tou[g]=r} turb[g,l]
  - sum{s IN S,sou[s]=r} spill[s,l]
 )
= - inflow[t,scen,r]  }
}
else
{if (t = n_stg)
{wbal{r}: old_lev[r] -lev[r] + sum{l IN L}
 (S_in_H*hrs[t,l]/mill)*(
    sum{g IN G,tin[g]=r} turb[g,l]
  + sum{s IN S,sin[s]=r} spill[s,l]
  - sum{g IN G,tou[g]=r} turb[g,l]
  - sum{s IN S,sou[s]=r} spill[s,l]
 )
+ def[r]
= - inflow[t,scen,r] }
else
{wbal{r}: old_lev[r] -lev[r] + sum{l IN L}
 (S_in_H*hrs[t,l]/mill)*(
```

```
    sum{g IN G,tin[g]=r} turb[g,l]
  + sum{s IN S,sin[s]=r} spill[s,l]
  - sum{g IN G,tou[g]=r} turb[g,l]
  - sum{s IN S,sou[s]=r} spill[s,l]
 )
= - inflow[t,scen,r] }
}},
! meet thermal demand per node of the electricity netwrok, per load slice
FOR {b IN B, l IN L}
{tr{b,l}: sum{g IN G,hbus[g]=b} hrs[t,l]*conv[g]*turb[g,l]
   + sum{u IN U,bus[u]=b} burn[u,l]
           - dem[b,t,l]*hrs[t,l]*dem_factor
   = -ghost[b,l]},
! bounds on thermal burn
FOR {l IN L, u IN U}
{brmx{u,l}: burn[u,l] <=gent_mx[u]*hrs[t,l]},
! bounds on turbine capacity, get 75% of capacity
FOR {g IN G, l IN L}
{genmx{g}:turb[g,l] <= turb_mx[g]*0.75},
! bounds on reservoir level, must be 50% full in final period
FOR {r IN R}
{if (lev_mx[r] <> lev_mn[r])
{if (t = n_stg)
{rvrcap{r}:(lev_mn[r]+lev_mx[r])/2 <= lev[r] <=lev_mx[r]}
else
{rvrcap{r}:lev_mn[r] <= lev[r] <=lev_mx[r]}}
};
END MODEL COST

generate COST(1,2,1/2*(lev_mx-lev_mn)) >> "../hydro/mps";
```

References

[1] T W Archibald, C S Buchanan, K I M McKinnon, and L C Thomas. Nested benders decomposition and dynamic programming for reservoir management. *Journal of The Operational Research Society*, 50(5):468–479, 1999.

[2] Thomas Archibald, Kenneth McKinnon, and Lyn Thomas. An aggregate stochastic dynamic programming model of multi-reservoir systems. *Water Resources Research*, 33(2):333–340, 1997.

[3] Dash Associates. *XPRESS-MP User Guide and Reference Manual.* Bliswoth House, Church Lane, Blisworth, Northants, NN7 3BX, UK, 10$^{\text{th}}$ edition, 1997.

[4] Mokhtar S Bazaraa, John J Jarvis, and Hanif D Sherali. *Linear Programming and Network Flows.* John Wiley & Sons, 2nd edition, 1990.

[5] E M L Beale. On minimizing a convex function subject to linear inequalities. *Journal of the Royal Statistical Society*, 17B:173–184, 1955.

[6] J F Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.

[7] J R Birge, M A H Dempster, H I Gassmann, E A Gunn, A J King, and S W Wallace. A standard input format for multiperiod stochastic linear programs. *COAL Newsletter*, 17:1–19, 1987.

[8] John R. Birge. Decomposition and partitioning methods for multistage stochastic linear programs. *Operations Research*, 33(5):989–1007, 1985.

[9] John R Birge. Models and model value in stochastic programming. *Annals of Operations Research*, 59:1–18, 1995. Also in [13] book.

[10] John R. Birge, Christopher J. Donohue, Derek F. Holmes, and Oleg G.Svintsitski. A parallel implementation of the nested decomposition algorithm for multistage stochastic linear programs. *Annals of Operations Research*, 75:327–352, 1996. see entry [11].

[11] John R. Birge, Christopher J. Donohue, Derek F. Holmes, and Oleg G. Svintsitski. A parallel implementation of the nested decomposition algorithm for multistage stochastic linear programs. Sent directly, to appear 95/96, 1995.

[12] John R Birge and François Louveaux. A multicut algorithm for two-stage stochastic linear progams. *European Journal of Operations Research*, 34:131–149, 1988.

[13] John R Birge and François Louveaux. *Introduction to Stochastic Programming*. Springer Series in Operations Research. Springer, 1997.

[14] A Brooke, D Kendrick, and A Meeraus. *GAMS: A User Guide*. The Scientific Press, Redwood City, CA, USA, 1998.

[15] Crawford S Buchanan, Kenneth I M McKinnon, and George K Skondras. The recursive definition of stochastic linear programming problems within an algebraic modelling language. Working Paper MS 98-, University of Edinburgh, Department of Mathematics & Statistics, Edinburgh EH9 3JZ, UK, 1998. Submitted to AOR.

[16] Z Chen, G Consigli, M A H Dempster, and N Hicks-Pedrón. Towards sequential sampling for dynamic portfolio management. Technical Report 01/97, University of Cambridge, Judge Institute Of Management Studies, Trumpington Street, Cambridge, UK, 1997.

[17] Christian Condevaux-Lanloy and Emmanuel Fragnière. Setstoch: A tool for multistage stochastic programming with recourse. Technical report, Logilab, HEC, Section of Management Studies, University of Geneva, 102 Bd. Carl Vogt, CH-1211 Gevève 4, Switzerland, August 1998.

[18] IBM Corp. *MPSX/370 Version 2, Users Guide*, 1988.

[19] Xavier Corvera-Poiré. *Model Generation and Sampling Algorithms for Dynamic Stochastic Programming*. Phd thesis, Department of Mathematics, University of Essex, 1995.

[20] G B Dantzig and M Madansky. On the solution of two-staged lienar programs under uncertainty. In J Neyman, editor, *Proceedings of the 4th Berkeley Symposium on Mathematical Statistics and Probabliity*, pages 165–176, 1961.

[21] George B Dantzig. Linear programming under uncertainty. *Management Science*, 1:197–206, 1955.

[22] George B Dantzig and Peter W Glynn. Parallel processors for planning under uncertainty. *Annals of Operations Research*, 22:1–21, 1990.

[23] M A H Dempster and R T Thompson. Evpi-based importance sampling solution procedures for multistage stochastic linear programmes on parallel mimd architectures. Technical Report 01/96, Univeristy of Cambridge, Judge Institute Of Management Studies, Trumpington Street, Cambridge, UK, 1996.

[24] J Dupačová and R Wets. Asymptotic-behaviour of statistical estimators and of optimal solutions of stochastic optimization problems. *Annals of Statistics*, 16(4):1517–1549, 1988.

[25] Yuri Ermoliev and Roger J-B Wets, editors. *Numerical techniques for stochastic optimization*. Number 10 in Springer series in computational mathematics. Springer-Verlag, 1988.

[26] Roger Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, 1987.

[27] Stein-Erik Fleten, Kjetil Høyland, and Stein Wallace. The performance of stochastic dynamic and fixed mix portfolio models. Also appears in [41], 1997.

[28] R Fourer, D M Gay, and B W Kernigham. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, South San Francisco, CA, USA, 1993.

[29] Robert Fourer. Modeling languages versus matrix generators for linear programming. *ACM Transactions on Mathematical Software*, 9:143–183, 1983. pre-print from GKS.

[30] Robert Fourer and David M Gay. Proposals for stochastic programming in the ampl modeling language. Talk given at ISMP97, Lausanne, 1997.

[31] Emmanuel Fragnière, Jacek Gondzio, Robert Sarkissian, and Jean-Philippe Vial. Structure exploiting tool in algebraic modeling languages. Technical Report 1997.2, Logilab, HEC, Section of Management Studies, University of Geneva, 102 Bd. Carl Vogt, CH-1211 Gevève 4, Switzerland, 1997.

[32] A Gaivoronski. Stochastic quasigradient methods and their implementation. In Ermoliev and Wets [25].

[33] H I Gassmann and A M Ireland. Scenario formulation in an algebraic modelling language. *Annals of Operations Research*, 59:45–75, 1995.

[34] H I Gassmann and A M Ireland. On the formulation of stochastic linear programs using algebraic modelling languages. *Annals of Operations Research*, 64:83–112, 1996.

[35] H I Gassmann and E Schweitzer. Proposed extensions to the smps input format for stochastic linear programs. Working Paper WP-96-1, School of Business Administration, Dalhousie University, Halifax, Canada, 1996.

[36] Horand I. Gassmann. Mslip: A computer code for the multistage stochastic linear programming problem. *Mathematical Programming*, 47:407–423, 1990.

[37] A Geist, A Beguelin, J Dongarra, W Jiang, R Manchek, and V Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Oak Ridge TN, USA, ornl/tm-12187 edition, 1993.

[38] Julia L Higle and Suvrajeet Sen. *Stochastic decomposition: a statistical method for large scale stochastic linear programming.* Kluwer, 1996.

[39] D F Holmes. Posts: Portable stochastic programming test set. Web-site. ftp://freebie.engin.umich.edu/pub/stoprobs/posts/.

[40] Alston Scott Householder. *The theory of matrices in numerical analysis.* Blaisdell, New York, 1964.

[41] Kjetil Høyland. *Asset Liability Management for a Life Insurance Company.* PhD thesis, Department of Economics and Technology Management, Norwegian University of Science and Technology, N-7034 Trondheim, Norway, 1998.

[42] Tony Hürlimann. Modeling: A new paradigm of programming. OR40, Lancaster, September 1998.

[43] Tony Hürlimann. Reference manual for the lpl modeling language (4.30). Technical report, University of Fribourg, 1998. www2-iiuf.unifr.ch/tcs/lpl/.

[44] IBM. *OSL:Optimization Subroutine Library*, release 2 edition, 1991.

[45] IBM. *OSL:Optimization Subroutine Library*, release 5 edition, 1997. See www6.software.ibm.com/es/oslv2/features/welcome.htm.

[46] Gerd Infanger. *Planning under uncertainty: solving large-scale stochastic linear programs.* Boyd & Fraser, 1994.

[47] Gerd Infanger and David P Morton. Cut sharing for multistage stochastic linear programs with interstage dependency. *Mathematical Programming*, 75:241–256, 1996.

[48] Peter Kall and Stein W. Wallace. *Stochastic Programming.* Systems and Optimization. John Wiley & Sons, 1994.

[49] Alan J King. *Stochastic Programming Interface Library User's Guide.* IBM Research Division, Thomas J Watson Research Center P.O. Box 218 Yorktown Heights, NY 10598, 1.0 edition, 1994. kingaj@watson.ibm.com, see also ekk web-site.

[50] Alamuru S Krishna. Enhanced algorithms for stochastic programming. Technical Report 93-8, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California 94305, 1993.

[51] Wai-Kei Mak, David P Morton, and R Kevin Wood. Monte carlo bounding techniques for determining solution quality in stochastic programs. *Operations Research Letters*, 24(1–2):47–56, 1999.

[52] David P Morton. Algorithmic advances in stochastic programming. Technical Report 93-6, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California 94305, 1993.

[53] David P Morton. An enhanced decomposition algorithm for multistage hydrelectric scheduling. *Annals of Operations Research*, 64:211–235, 1996.

[54] M V F Pereira, M E P Maceira, G C Oliveira, and L M V G Pinto. Combining analytic models and monte-carlo techniques in probabilistic system analysis. *IEEE Transactions on Power Systems*, 7(1):265–272, 1992.

[55] M. V. F. Pereira and L. M. V. G. Pinto. Multi-stage stochastic optimization applied to energy planning. *Mathematical Programming*, 52:359–375, 1991.

[56] George K Skondras. Modelling languages in mathematical programming. by research, University of Edinburgh, Department of Mathathematics and Statistics, James Clerk Maxwell Building Edinburgh EH9 3JZ, September 1998.

[57] James Tebboth and Robert Daniels. A tightly integrated modelling and optimisation library: A new framework for rapid algorithm development. to appear in aor, 1998.

[58] L A Terry, M V F Pereira, T A Araripe Neto, L F C A Silva, and P R H Sales. Coordinating the energy generation of the brazilian national hydrothermal electrical generating system. *Interfaces*, 16:16–38, 1986.

[59] Ronald Aaron Thisted. *Elements of Statistical Computing.* Chapman & Hall, 1988.

[60] Robin T Thompson. *Fast Sequential and Parallel Methods for Solving Multistage Stochastic Linear Programmes.* PhD thesis, Department of Mathematics, University of Essex, 1997.

[61] R. van Slyke and R. J.-B. Wets. L-shaped linear programs with applications to optimal control and stochastic linear programs. *SIAM Journal on Applied Mathematics*, 17(4):638–663, 1969.

[62] Roger J.-B. Wets. Stochastic programs with fixed recourse: The equivalent deterministic program. *SIAM Review*, 16(3):309–339, 1974.

[63] Hilary Paul Williams. *Model Building in Mahematical Programming.* John Wiley & Sons, 2nd edition, 1985. get reference for new edition which will use sMAGIC.

[64] Robert J Wittrock. Dual nested decomposition of staircase linear programs. *Mathematical Programming*, 24:65–86, 1985.