INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY
DIVISIÓN DE INGENIERIA Y ARQUITECTURA
PROGRAMA DE GRADUADOS EN INGENIERIA

TECNOLÓGICO DE MONTERREY®

"Deterministic and Stochastic Profit Maximization Versions of the Economic Lot Scheduling Problem with pricing considerations"

TESIS
PRESENTADA COMO REQUISITO PARCIAL
PARA OBTENER EL GRADO ACADEMICO DE

MAESTRO EN CIENCIAS
CON ESPECIALIDAD EN CALIDAD Y PRODUCTIVIDAD

POR:
ING. LUCIANO SALVIETTI CIGNETTI

MONTERREY, N.L. **DICIEMBRE 2006**

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

DIVISIÓN DE INGENIERIA Y ARQUITECTURA
PROGRAMA DE GRADUADOS EN INGENIERIA

Los miembros del Comité de tesis recomendamos que la presente tesis del Ing. Luciano Salvietti Cignetti sea aceptada como requisito parcial para obtener el grado académico de Maestro en Ciencias con especialidad en:

Sistemas de Calidad y Productividad

Comité de Tesis

Dr. Neale Ricardo Smith Cornejo
Asesor

Dr. José Luis González Velarde
Sinodal

Dr. Francisco Ángel-Bello
Sinodal

Aprobado:

Dr. Francisco Ángel-Bello
Director del Programa de Graduados en Ingeniería
Mayo 2006

Dedicated to

To God, for giving me the opportunity to be at Monterrey doing this Master’s degree and also for being my guidance and my cornerstone through life.

To my family, for their everlasting support and help through all these years.

Acknowledgments

To my thesis advisor, Dr. Neale Smith, for supporting and guiding me on the completion of this thesis.

To the doctors at the ITESM Extended Enterprise for Mass Customization Research Chair in Industrial Engineering, for their help and guidance through the research seminars during my student years.

This research thesis was supported by the ITESM Extended Enterprise for Mass Customization Research Chair in Industrial Engineering (Fund Number CAT025)

To all the people at the Quality and Manufacturing center on ITESM Monterrey Campus, thanks for their friendship and support.

Table of Contents

List of Tables and Figures

Figures

Abstract

The Economic Lot Scheduling Problem (ELSP) is a well known problem that focuses on scheduling the production of multiple items on a single machine such that inventory and setup costs are minimized. In this thesis the ELSP is extended to include price optimization with the objective to maximize profits. Two variants are proposed: a deterministic version which will be known as the PELSP and the stochastic version which is called the SPLSP. A solution methodology based on column generation and integer programming is proposed and is shown to produce very close to optimal results with short solution times. Computational testing is performed to evaluate the methodology. The results are discussed and recommendations for further research are provided.

Chapter 1. General Context

1.1 Introduction

The Economic Lot Scheduling Problem (ELSP) has been studied since about 50 years ago. The ELSP deals with the problem of scheduling the production of a multi product system. The problem itself includes a capacity restriction for the whole system to remain feasible, so there are restrictions on the time needed to produce each product as well as the periods in which they will be scheduled to be produced. In a few words, the purpose of the ELSP is to solve the problem of how to maintain cyclic production patterns based on lot scheduling while minimizing costs.

There are numerous of methods that seek to produce an approximation to cyclic coordination. The Basic Period approach (BP) is one of the most known and assumes that the production runs of all products must be realized in a production period that is sufficiently big enough for that purpose. This constraint is the one that creates some problems because some solutions can be become suboptimal. There is also the Extended Basic Period approach (EBP) which removes the basic constraint of the BP approach and admits the possibility that on a certain period of time, some products and not necessarily all can be produced. This relaxation is an important variant that influences the possible solutions to the problem. It is important to notice that for this thesis, the Basic Period approach will be used.

Regarding the optimization of the ELSP, it is important to notice that all the prior authors have used a wide variety of optimization methods to propose feasible and optimal solutions: some have used heuristics, branch and bound algorithms, integer programming and genetic algorithms, but none of them have worked with the column generation technique with mixed integer programming. This technique has been used on different problems and areas.

The purpose of this thesis is to use the column generation technique as an optimization tool that helps to achieve feasible solutions to the ELSP problem with price optimization. In chapter 2, a literature review is presented to place this thesis within the proper context.

Then, chapter 3 will be focused on the development of the mentioned problem but with two variants: the deterministic model of the ELSP with pricing (the so called "deterministic lot scheduling problem with price optimization" or just "PELSP"); and the stochastic version of the ELSP with pricing (the "stochastic lot scheduling problem with price optimization", or just "SPLSP").Basically, the PELSP presents the basic formulation of the ELSP where both the price and the production schedule are found in order to maximize profit. In the case of the SPLSP, the production times and cyclic periods of each product are the decision variables and price is a recourse variables which is not directly optimized. Chapter 3 will also be focused on describing the solution methodology used for the models developed while Chapter 4 will present and discuss the experiments designed to test the robustness of the optimization method. Chapter 5 will provide general conclusions.

1.2 Objectives

As it was said in the introduction, the main purpose of the research on this thesis is to use the Column Generation Technique as an optimization method for the two models presented. In other words, the objectives are:

- Formulate a complete optimization algorithm that finds the solution to the lot scheduling problem with price optimization, for both the deterministic and the stochastic versions, for a manufacturing system with multi products that includes prices, setup times and capacity restrictions
- Use the Column Generation Technique to optimize the deterministic lot scheduling problem with price optimization (PELSP)
- Use the Column Generation Technique to optimize the stochastic lot scheduling problem with price optimization (SPLSP)

1.3 Hypothesis

Based on the objectives described above, it is clearly set that the general purpose of the thesis is to prove that the Column Generation Technique is a good and feasible method to optimize the lot scheduling problem with pricing. Accordingly, two basic hypotheses have been developed, which are the following:

- The Column Generation technique produces results that are GOOD FEASIBLE SOLUTIONS near the optimal solution (on a 5% range), for the deterministic lot scheduling problem with price optimization (PELSP)

- The Column Generation technique produces results that are GOOD FEASIBLE SOLUTIONS near the optimal solution (on a 5% range), for the stochastic lot scheduling problem with price optimization (SPLSP)

1.4 Justification

This thesis can be considered very important for the Operations Research community interested in the optimization of multi product production systems with Prices. The essential value of the models analyzed in this thesis is that the Column Generation technique works well for joint production and pricing problems. In fact, the solution methodology presented on the thesis is on certain way exploratory because up to date there haven’t been similar works which apply this technique to joint pricing and production problems. Finally, it is important to conclude saying that the results of the thesis will help promote more research on lot scheduling production systems with pricing , as well as to set the bases for new projects that derivate from the thesis.

1.5 Scope

The research topic proposed for this thesis deals with an area of production systems optimization that has not been fully studied up to date. The whole topic is innovative and offers an opening in an area that is important to people on production and operations research. The experimentation that was conducted on Chapter 4 will help show the results of my proposal and indicate the following future directions of this optimization area.

Chapter 2. Literature Review

2.1 The Economic Lot Scheduling Problem

The Economic Lot Scheduling Problem (ELSP) is a well known problem, based on the single item Economic Production Quantity (EPQ) model, which focuses on scheduling the production of multiple items on a single machine such that inventory and setup costs are minimized. A great number of approaches to solving the ELSP have been proposed. Since the work on this thesis is directly related to two variants of the ELSP, it is important to understand where these variants fit and how they interact with other models in the existing literature. So, in this section, a review of the deterministic economic lot scheduling model is presented.

One of the earliest models for economic lot programming is proposed by Manne [42], who focuses on the production of several items in a workshop. Rogers [46] continues that approach but looks further and realizes that in practice, when several items have to be manufactured, the situation becomes an economic lot scheduling problem. The author then provides the first models of the ELSP, developed to fit the production schedules of various items into a defined planning period. Later, other methods are developed to solve the ELSP. Bomberger [8] uses dynamic programming to solve the problem. Bomberger [8] is also the first author to propose that the cycle time for each product should be an integer multiple of a Basic Period (*B*) which has to be long enough to allow all items to be produced in that period.

Madigan [41] develops heuristic methods for the ELSP. Doll and Whybark [16] propose additional heuristics but focus on an iterative procedure that directly determines the production cycle for each product as an integer number. Haessler and Hoghe [30], basing their work on the Doll and Whybark [16] model, present an improved iterative procedure that guarantees a feasible solution, unlike previous heuristics.

Elmaghraby [17] provides an extensive review of ELSP research and proposes what is known as the Extended Basic Period (EB) approach model. He also proposes an integer model that systematically achieves feasibility and facilitates determining the production multipliers. Haessler [29] continues the approach but proposes restricting the number of periods between production runs to integer powers of two, very quickly finding good solutions. Graves [27] advances on the mathematical programming front by developing a non linear mixed integer model that works strictly with integer multipliers and a non-integer period. Dobson [15] continues the development of heuristics and develops a method that allows cycle time and lot size variations assuming that the setup costs vary with the number of setups and include only costs that would result in actual cash losses. The approach allows feasible solutions to be found more easily and the resulting solutions smooth the utilization of the facility.

Bretthauer et al. [10] proposes a general model and solution method that can be adapted to the ELSP model developed by Elmaghraby [17]. Their branch and bound algorithm produces better solutions than those produced using previous approaches. Another notable achievement, by Gallego and Shaw [24], is showing that the ELSP is NP hard. Their approach focuses on cyclic scheduling since non cyclic scheduling is a very hard to explain and evaluate. Regarding the structure of the problem, Yao and Elmaghraby [61] study an ELSP model without capacity restrictions and consider the model's behavior under a power-of-two policy originally proposed by Roundy [48] in another context. Their study concludes that the objective function is piece-wise convex.

More recently, Wagner and Davis [57] propose a search heuristic for an ELSP with sequence dependent setups. Yao et al. [62] presents two improved linear models based on a heuristic that guaranteed feasibility for the EB ELSP. Cooke et al. [14] proposes an improved non linear mixed integer model that also considers production sequences. They also compare dynamic programming and genetic algorithm approaches. The most recent studies seem to be those by Yao and Huang [63], who propose a hybrid algorithm for deteriorating products using a power-of-two policy; Lin et al. [40], who also work with deteriorating products but use a common cycle approach; and Chang et al. [11], who develop a specialized model for fuzzy demand.

2.2 Stochastic lot scheduling problems

The second of the ELSP variants proposed on this thesis is a stochastic version of the ELSP, which it is known as the “Stochastic Lot Scheduling Problem with Price optimization” (SPLSP). Following there is a review of work done on the area of stochastic lot scheduling problems.

Sox et al. [53] proposed a complete review on stochastic lot scheduling problems. On their paper, they mention that the general SLSP model is the problem of scheduling production of multiple products, each with random demand, on a single facility with limited production capacity and changeover between products. They also mention that the SLSP does have two basic variants: the continuous time model which is known as the SELSP (Stochastic Economic Lot Scheduling Problem) and the discrete time version which is known as the SCLSP (Stochastic Capacited Lot Scheduling Problem).

Following the prior classification, it can be said that the SPLSP cannot be particularly defined as a member of one of the two variants but it can be portrayed as a hybrid model because it shares some assumptions and characteristics of both models: regarding the SELSP, the model relates with it on the assumptions that the demand must be stationary and that the time horizon is infinite; regarding the SCLSP model, the model shares the assumption of discrete scheduling times.

Having seen where the SPLSP fits on the current classification for stochastic lot scheduling models, it is time to explore some of the work done on the SELSP as well as the SCLSP, taking into account that the focus is to depict the types of solution and general details about the models.

As it was said before, the Stochastic Economic Lot Scheduling Problem (SELSP) is a developed extension of the deterministic Economic Lot Scheduling Problem (ELSP). The focus of the SELSP model is scheduling the production on a single machine multi product system, taking into account random demand and setup times while minimizing the total costs incurred. Winands et al. [59] propose a very complete an up to date review of the work done on the SELSP area. They analyze the different variants that arise from the basic SELSP model, including a classification based on the types of solution: production sequence (fixed and cyclical) and lot sizing policies (global or local). Finally, they present some lines of research for the future.

Some of the papers that we are going to review can fit on the different categories: Qiu and Loulou[45] and Sox and Muckstadt[52] presented models that fit on the dynamic production sequence and global lot sizing category. Meanwhile, Altiok and Shiue[3][4] and Zipkin[64] fall into the dynamic production sequence and local lot sizing category.

On the fixed production sequence and dynamic cycle length category with global lot sizing strategies we can see the works of Bourland and Yano[9], Gallego[22] and Markovitz[43] among others. Also, on the fixed production sequence and dynamic cycle length with local lot sizing strategies we can observe the papers of Anupindi and Tayur[5], Federgruen and Katalan[20] and Vaughan[56]. Finally, it can be observed the work of Erkip et al.[19] on the fixed production sequence and dynamic cycle length category with local lot sizing strategies.

Currently, the SPLSP fits on the “Fixed production sequence + fixed cycle length and Local Lot sizing strategy” category. Regarding this category, Winand et al. [59] tell us that the research work done on it has been insufficient and is a good line of work for future papers, which somehow validates the efforts towards contributing with new work on stochastic scheduling problems. As it will be seen later on the description of the SPLSP model, the fixed production sequence and fixed cycle length are going to be known as the design variables for the SPSLP.

Also, it is important to notice that the SPLSP control system involves two policies: pricing and purchasing. These two policies characterize the singularity of the model and define how the production system is kept on balance while trying to maximize the expected profit.

Another interesting aspect of the SPLSP, which is worth mentioning, is the fact that it works with a service level policy to control the occurrence of backorders. What it has been noticed is that the authors on stochastic lot scheduling problems use different ways to achieve a desired service level for the customer. Following is a comparison table regarding the inclusion of backorder/shortage costs and the type of service level strategies used by the authors reviewed:

Author	Backorder / Shortage costs	Service Level
Altiok and Shiue [3]	Backorder	Average backorder level , probability of backorders due to average inventory levels
Anupindi and Taylor [5]	Backorder	Expected fraction of orders filled immediately, response time for an order and backlog costs
Bourland and Yano [9]	No	Reorder point on passing the safety stock levels
Federgruen and Katalan [20]	Backorder	Lower bounds on the item’s fill rate
Gallego [22]	Backorder	Service levels using backorder and holding costs
Goyal [26]	Shortage	Probability of shortage occurring equal to a service level
Leachman and Gascon [38]	No	Customer service policies (that avoid the happening of shortages)
Markovitz et al. [43]	Backorder	Average server utilization
Qui and Loulou [45]	Backorder	Probability of backorders (equal to constant value)
Sox and Muckstadt [52]	Backorder	Fraction of total capacity required to meet the expected demand for all items through period t
Vaughan [56]	No	Safety stock for a defined cycle service level
Zipkin [64]	Backorder	Service rate (average number of batches processed per unit while the server is busy), Constant Lower Bound for average backorder per item

Table 1. Author Comparison for backorder/shortage costs and types of service level

Goyal[26] started the approach on stochastic lot scheduling models by presenting a model for a single machine production system where he relaxed the assumption of deterministic demand , including also shortage costs. Graves [28] was one of the first authors to work with the SELSP. His approach regards a Markov Decision Model for a one product problem, and using this method he develops a heuristic to solve the SELSP but with a periodic review policy. Gallego[22] proposed an interesting cyclic schedule model based on the original ELSP but including random demands with constant expected rates and backorders. He formulated his model as a control problem and established a linear recovery policy to help achieve feasible and near optimal results.

Bourland and Yano [9] establish a different approach to stochastic lot scheduling problems through the design of a model with capacity slack as a form of flexibility. Their model is a very complete variety of the SELSP that considers capacity slack, safety stocks and overtime while minimizing the expected cost per unit time of inventory, overtime and if applicable, on setup costs. The model is based on a two level hierarchical solution approach with two variations: fixed capacity slack and variable capacity slack. The results of the paper have drawn some conclusions regarding the importance of carefully selecting the amount of capacity slack and the importance of optimizing the idle time on a complex model like the ELSP.

Qiu and Loulou [45] formulated a model of the SELSP through a semi-Markov decision process, for a two product system. The functioning of the model seems to be almost equal to the one in Gallego[22] but the stochastic behavior of demand is recreated through a Poisson process and the unit processing time is known. This model includes setup time and setup costs. The objective of the model was minimizing the total expected cost with discounting over an infinite horizon. The approach given by this model presented very good results such that the error bound of the authors model is much tighter than other error bounds they compared from normal SELSP models.

Federgruen and Katalan [20] analyze the SELSP problem from the modeling perspective of Bourland and Yano [9] but propose a set of production and inventory strategies to minimize holding, backlogging and setup costs. The multi production system works with on a rotational cycle that includes to possibility of idle times. They also proposed the establishment of lower bounds for the optimal cost values and compare their results with the application on some deterministic ELSP problems. Their results show that the base-stock policies proposed as strategies are efficient and rich in solutions and above all, are far better than the solutions given on the deterministic approaches.

Markovitz et al. [43] consider two queuing problems, with a structure similar to the basic model proposed in Qiu and Loulou [45]: one that includes random setup costs and other that includes random setup times. In reality these structures are control problems that can be modeled like versions of the ELSP, but with backorders. Their approach is made through the heavy-traffic analysis method mentioned by Sox et al.[53]. Basically, their approach consists on approximating the scheduling problem through diffusion control. Their analysis gives as a result a dynamic lot sizing policy that gave new insights on the optimal solution for the SELSP: basically they concluded that the lot sizing policy depends if setup costs or setup times are incurred and the cost structure form for all the items.

Regarding the SCLSP model, Sox and Muckstadt [52] describe a finite horizon stochastic optimization model for a single stage multi product system, a dynamic model that bases on the SELSP. Their solution procedures were meant to find near optimal solutions. They work with a master problem solved through Lagrange decomposition algorithm and make the optimization using the Branch & Bound procedure.

2.3 Pricing

The literature on joint pricing and inventory optimization is also clearly related to the work on this thesis. One of the earliest papers on pricing and inventory is by Whitin [58], who proposes a link between pricing and inventory control. Kotler [34] shows that there exists an interaction between marketing policies and the economic order quantity. He first determines the optimal selling price that provides a maximal revenue for a given demand curve and then determines the EOQ considering the selling price and demand as fixed parameters. Kunreuther and Richard [35] develop two inventory models based on economic order quantity (EOQ) and economic production quantity (EPQ) models for determining the pricing and lot sizing policies for the case of the linear demand function. Later, a general inventory model in which demand is a function of a sales price that depends on pricing policies and the unit cost is presented by Ladany and Sternlieb [37]. Urban [55] develops an inventory model based on the EPQ model that considers learning effects and the possibility of defective goods in the fabrication process. It is used to determine simultaneously the lot size, price mark-up, and advertisement expenditure when the demand for the goods is a deterministic function of the selling price.

Several researchers have worked on joint pricing and inventory problems from a supplier's point of view. For example, Monahan [44] analyzes the effect of a price discount offered by a supplier to its unique buyer with the main objective of increasing the supplier's profit. A model for determining simultaneously the price and lot size for the supplier-buyer problem from the supplier's point of view considering a scheme of quantity discounts is presented by Rosenblatt and Lee [47]. They show that the optimal order quantity for a supplier is an integer multiple of the buyer's lot size. Abad [1,2] solves the problem of joint price and lot size determination faced by a retailer when purchasing an item for which the supplier offers an incremental and an all-unit quantity discount scheme considering the linear and constant elasticity demand functions.

More recently, Lee [39] presents a geometric programming approach to determine a profit-maximizing price and order quantity for a retailer. He considers the demand as a nonlinear function of selling price with a constant elasticity. Also, Kim and Lee [33] present a study that focuses on fixed and variable capacity models for the joint setting of a product's selling price and lot size for a profit maximizing organization considering constant but sales price-dependent demands over a planning horizon. Most recently, an EPQ inventory model that determines the production lot size, marketing expenditure and product's selling price is developed by Sadjadi et al. [49] and an algorithm for simultaneous determination of the sales price and lot size in a make-to-order contract production environment from the supplier’s perspective is developed by Banerjee [6]. A very recent contribution is the work by Haugen et al. [31], who develop a model for the Capacited Lot Sizing Problem (CLSP) that includes price optimization. The author’s name their model the PCLSP and their formulation and solution approach set the stage for further research on the topic. It is important to note that the PELSP variant proposed on the thesis, a model and solution method for an economic lot scheduling problem with pricing that uses the basic period approach is proposed, thus extending the work of Bomberger [8] along the lines advanced by Haugen et al. [31].

Following, a review on some stochastic lot scheduling models that involve some kind of pricing technique will be made. Continuing with the classification proposed by Sox et al. [53] and adding the price optimization technique, we could basically encounter two stochastic lot scheduling problems with pricing: the SELSP with pricing for 1 product or multiproduct production systems, or the SCLSP with pricing for 1 product or a multiproduct production system.

One of the earliest approaches of Pricing on production systems can be found on the work by Gallego and van Ryzin[23], who formulated a intensity control problem with dynamic pricing where demand is price sensitive and stochastic, and the main objective is to maximize expected revenues. The problem they present characterizes by two properties: the lack of short term control over inventory stock and the presence of due times for selling goods. As it was said before, the demand is a price sensitive stochastic point process that is a decreasing function of price and no backlogging is allowed. So basically, they propose a multi product production system where the time horizon for planning and scheduling is finite.

One example of a paper that resembles the SELSP with Pricing for a multi-product production system is the work done by Fransoo et al. [21]. They introduced a two level hierarchical model with an infinite time horizon and discrete scheduling times, whose objective was planning and scheduling multi products for a single machine production system that works with stationary stochastic demand, has capacity constraints and does not allow backorders. Their approach specialized on situations where demand levels are high compared to the available production capacity. They developed a heuristic to help produce target production cycles and target service levels for each product. The interesting part of this paper is that the focus is on allocating capacity to individual products in order to maximize the expected profit while keeping a desired service level target for each product. So, the objective function is to maximize profit at a service level constraint.

Gallego and Van Ryzin[25] continued the line of pricing on stochastic lot scheduling models by proposing a finite horizon model for maximizing the expected profit , similar to their model on Gallego and Van Ryzin[23]. The difference is that this new approach includes two heuristics for solving the stochastic problem that are shown to be optimal as the expected sales volume tends to infinite. They also propose the establishment of an upper bound to compare the solutions using a deterministic version of the problem.

Most of the pricing stochastic production system models we have reviewed are multiproduct but following we can see two papers where the focus is the analysis of a system with one product.

Chen and Simchi-Levi[12] proposed an infinite horizon, single product, periodic review model where pricing is made as a joint decision along with the scheduling of production, at the beginning of each working period. The model analyzed involves identically distributed independent random demands, and is also important to notice that demand distributions vary according to the product price. Backlogs are allowed and ordering costs include a fixed and variable cost. They conclude showing that their stationary (s,S,p) policy is optimal to maximize the expected discounted or expected average profit over the infinite planning horizon. The same authors extended their work by proposing a model, which can be seen at Chen and Simchi-Levi[13], where they analyze a finite horizon version. The assumptions regarding the demand and the joint decisions remain the same but now the objective is to find an inventory policy and pricing strategy that maximizes the expected profit over that finite horizon of time. As part of their conclusions on the model they propose, they say that if demand has an additive distribution, the profit functions are k-concave and the stationary review policy is optimal, but for other general demand distributions the profit functions are neither necessarily k-concave nor optimal.

Chapter 3. Modeling and Solution Methodology

3.1 Description of Models

In the present chapter, the two ELSP variants proposed on the thesis will be described and also their solution methodologies will be depicted.

3.1.1 PELSP

Before presenting model formulations for the PELSP, the following basic notation is defined. Additional notation will be introduced and defined as needed.

Parameters:

B: length of the basic period, in days

S_i : setup cost for product *i*, in dollars per setup

c_i : unit production cost for product *i*, in dollars per unit

m_i : production rate for product *i*, in units per day

T_i : setup time for product *i*, in days

h_i : inventory carrying charge for product *i*, in % per day

M: number to products *i*

τ : available capacity in the basic period , in days

Variables:

d_i : demand for product *i*, in units per day

k_i : basic period multiplier for product *i*, a positive integer

Notice that the inventory carrying charge (h_i) is allowed differ for different products. This is because inventory carrying charges include both financial opportunity cost and also operational (cash) costs. These operational costs can vary between products. For example, one product may require special handling or storage due to being more fragile or simply of a different size than another.

In order to provide continuity with the non-pricing ELSP model, the ELSP model that applies the basic period approach is re-stated here. The objective is to minimize total cost per unit of time. The model is:

$$\min Z = \sum_{i=1}^{M} \left[\left(\frac{S_i}{k_i\, B} \right) + \left(\frac{c_i\, h_i\, (m_i - d_i)\, d_i\, k_i\, B}{2\, m_i} \right) \right] \tag{1}$$

s.t.

$$\sum_{i=1}^{M} \left(T_i + \frac{d_i\, k_i\, B}{m_i} \right) \le B\,, \tag{2}$$

Where k_i is a strictly positive integer, $B > 0$ and $0 \le d_i \le m_i$. In the ELSP, *B* is a decision variable and d_i is a constant. In contrast, in the PELSP formulation, *B* is a given constant and d_i becomes a decision variable. The demand rates d_i become variables because the demand varies with a settable price. A monopoly assumption is being made, which although not always realistic has been made in nearly all of the existing pricing and inventory literature. The benefits of pricing optimization and the factors driving developments in this area are documented in [18]. We assume that *B* is a constant for two reasons: 1) as a simplifying assumption, and 2) because in practice it is unlikely that a real firm will be willing to produce on a schedule that does not follow a pattern that fits commonly used units of time such as hours, days, or weeks. For this last reason it seems more realistic to solve the problem for values of *B* that are deemed acceptable by the firm rather than let the length of *B* be determined by the solution method.

In the PELSP, the firm can set sales prices and demand is modeled as a function of price. In the thesis, it is assumed that the demand function is of the exponential type given by:

$$d_i\,(P_i) = R_i \exp\left(-P_i/\alpha_i\right), \tag{3}$$

where R_i is a market size parameter for product *i* and α_i is a price scaling constant. These parameters are usually obtained using statistical estimation techniques in order to best fit the demand behavior of a given product. For examples of the use of the exponential demand function in previous pricing research see [36] and [51]. Since (3) is an invertible function, the price can be expressed as a function of demand given by:

$$P_i\,(d_i) = \alpha_i \ln\left(R_i / d_i\right). \tag{4}$$

The objective is to maximize profit. The objective function is:

$$\max U = \sum_{i=1}^{M} P_i\,(d_i)\cdot d_i - \sum_{i=1}^{M}\left[\left(\frac{S_i}{k_i\,B}\right) + \left(\frac{c_i\,h_i\,(m_i - d_i)\,d_i\,k_i\,B}{2\,m_i}\right) + c_i\,d_i\right]. \tag{5}$$

After substituting (4) into (5) we obtain the following formulation of the problem:

$$\max U = \sum_{i=1}^{M}\left[\alpha_i \ln\left(R_i / d_i\right)\right] d_i - \sum_{i=1}^{M}\left[\left(\frac{S_i}{k_i\,B}\right) + \left(\frac{c_i\,h_i\,(m_i - d_i)\,d_i\,k_i\,B}{2\,m_i}\right) + c_i\,d_i\right] \tag{6}$$

s.t.

$$\sum_{i=1}^{M} \frac{d_i\,k_i\,B}{m_i} \le \tau - \sum_{i=1}^{M} T_i\,, \tag{7}$$

with $d_i > 0$, and $k_i > 0$ and integer. The decision variables are the d_i and k_i. Notice that the right hand side of (7) is no longer *B*. This new right hand side is introduced because in general the time available for setups and production need not equal the time span of the basic period. From (6), it can be seen that *B* may be interpreted as the real time span (clock time) over which demand is realized. In general, τ may be less than the clock time. For example, in a 24 hour period, the firm might operate for only one 8 hour shift.

3.1.2 SPLSP

Before presenting model formulations for the SPLSP, the following basic notation is defined. Additional notation will be introduced and defined as needed. The design variables are G_i and k_i, which will be depicted below.

Parameters:

B: length of the basic period, in days

τ: Time available for production in a basic period, in days

S_i : Setup cost for product *i*, in dollars per setup

c_i : Unit production cost for product *i*, in dollars per unit

m_i : Production rate for product *i*, in units per day

T_i : setup time for product *i*, in days

h_i : Holding cost for product *i*, in % per day

M: number to products *i*

$D_i(\cdot)$: Demand function for product *i*, in units per day

e_t : Random error for *t* periods

V_i : Vendor Price, for product *i*, in dollars per unit

Variables:

G_i : Production time for product *i*, in days, a real > 0

k_i : Basic period multiplier for product *i*, an integer > 0

In order to provide continuity with the non-pricing ELSP model, the ELSP model that applies the basic period approach is re-stated here. The objective is to minimize total cost per unit of time. The model is:

$$\min Z = \sum_{i=1}^{M} \left[\left(\frac{S_i}{k_i\ B} \right) + \left(\frac{c_i\ h_i\ (m_i - d_i)}{2\, m_i} \right) \right] \tag{8}$$

s.t.

$$\sum_{i=1}^{M} \left(T_i + \frac{d_i\ k_i\ B}{m_i} \right) \leq B\,, \tag{9}$$

where k_i is a strictly positive integer, $B > 0$ and $0 \leq d_i < m_i$. In the ELSP, B is a decision variable. In the PELSP formulation showed on the prior section, B is a given constant and d_i becomes a decision variable. The demand rates d_i become variables because the demand varies with a settable price.

In the SPLSP formulation, *B* remains constant. It is assumed that *B* is a constant for two reasons: 1) as a simplifying assumption, and 2) because in practice it is unlikely that a real firm will be willing to produce on a schedule that does not follow a pattern that fits commonly used units of time such as hours, days, or weeks. For this last reason it seems more realistic to solve the problem for values of *B* that are deemed acceptable by the firm rather than let the length of *B* be determined by the solution method.

The SPLSP is a model which tries to solve the problem of determining the optimal values of G_i and k_i to maximize the Total Profit. This model works with a policy that requires the maintenance of a service level to the costumers, through two mechanisms:

1. Control of Prices
2. Buying when there is a backlog (The purchase is first done to cover the backlog and then to reach a certain inventory level)

Note: The Price and the amount of products bought can be considered the recourse variables.

In the SPLSP, the firm can set sales prices and demand is modeled as a function of price. So, it is assumed that the demand function is of the linear type given by:

$$D_i(P_i) = (b_i - a_i P_i) \tag{10}$$

At first , an initial inventory of 0 units was supposed and a price was determined so that it could guarantee a certain probability of not having stock outs. Using (10), the price can now be expressed as a function of demand given by:

$$P_i = \frac{b_i - D_i(P_i)}{a_i} \tag{11}$$

Where a_i and b_i are market size parameters defined for each product.

To control the desired price, a formula to produce for a given value of G_i (obtained through the optimization that will be explained on section 3.2.2) is used. This formula, known as the Required Production for product *i* (RP_i), is given by:

$$RP_i = \frac{I_i + m_i G_i}{B\left(k_i + \left(z\sigma\sqrt{k_i}\right)\right)} \qquad (12)$$

It's important to notice that the Required Production for each product is aimed to control the inventory at a service level of 95%, given by the expression:

$$\Pr\left(I_i \geq 0\right) \geq 0.95 \qquad (13)$$

After the required production is done, we need to observe the Final Inventory, where there are two possible events:

1. The Final Inventory is Positive, thus the control of the price is continued for the next period of evaluation.
2. The Final Inventory is Negative, thus buying product has to be done in order to solve the stock out. Regarding this situation, there are two policies for determining the quantity that has to be bought:
    a. The amount bought is the difference between the Final Inventory of the actual period (IF_i) and the Beginning Inventory of the prior period (I_{i-1}). The purpose of this policy is to ensure that a positive initial inventory

level will be reached, which is equal to the same initial inventory level on the prior period ; So, the buy is then given by:

$$\gamma = I_{i-1} - IF_i \tag{14}$$

b. The amount bought is calculated as the difference between the Final Inventory of the actual Period (IF_i) and an optimum level quantity ($Q_i *$). This optimum level quantity represents the maximum amount of units that should be bought to maximize the total profit ; the quantity is given by:

$$Q_i * = \left(\frac{B\left(z\sigma\sqrt{k_i} + k_i\right)}{2} \right) \cdot \left(b_i - \frac{a_i \ V_i \left(z\sigma\sqrt{k_i} + k_i\right)}{k_i} - \frac{2 \ m_i \ G_i}{B\left(z\sigma\sqrt{k_i} + k_i\right)} \right)$$

(15)

Thus, the quantity bought is,

$$\gamma = Q_i * - IF_i \tag{16}$$

Is important to notice that because of the shape of $Q_i *$, there are certain values of the design variable G_i that produce negative values on Q_i*, which means that is too expensive to buy that much quantity and is it better just to buy the amount that was not delivered to the client on the prior period. Thus, If $Q_i * < 0$, then $Q_i * = 0$, and (16) changes like this:

$$\gamma = - IF_i \tag{17}$$

3.1.2.1 Estimating the Expected Profit

As it was said before, the objective is to maximize the Expected value of the Total Profit, given by:

$$Max = EV\left[\text{Profit}\left(\mathbf{G},\mathbf{k},\mathbf{e}\right)\right] \tag{18}$$

Since the stochastic modeling is a bit more complicated than deterministic modeling, it was chose to estimate the expected profit using Simulation. The Simulation method consisted on recreating a certain number of working periods, each with the same amount of time and resources but with a probabilistic behavior given by an Error Value, and calculating the Total Profit Value for each Period.

It is important to notice that the working mechanisms of the SPLSP, allow a basic model with two cases:

a. Called “Case 1” implies that there is a surplus of finishing inventory, which means that the finish inventory was higher than expected. This is the most common case.
b. Called “Case 2”, implies that there is a shortage of finishing inventory on the analyzed period, which means that the level of finished inventory was lower than expected resulting in negative numbers. If this happened, buying a certain amount of a product is allowed so to reach the beginning inventory level of the prior period, in order to maintain the same service level. This case happens just 5% of

the time. Since two policies for determining the amount to be bought have already been determined, there are also two versions of case 2:

i. Case 2a. Corresponds to the policy where it is just needed to buy an amount in order to reach the same initial inventory level on the prior period

ii. Case 2b. Corresponds to the policy where it is required to buy an amount in order to fulfill the lost order from the clients , the amount bought depends on the value of Q_i* (it can either be restored the initial inventory level to 0 , if Q_i* is negative ; or to some positive inventory level , if Q_i* is positive)

Next is a graph that shows the two basic cases on a 4 period display (the dotted lines represent the inventory goals for each period):

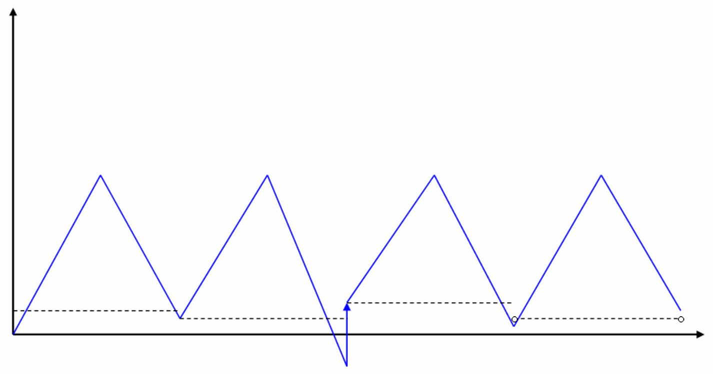

Figure 1. SPLSP cases graph

The objective functions (Expected Profit) will now be defined for cases 1 and 2 (the two versions):

Case 1

$$U = \sum_{i=1}^{M} P_i\, D_i\, B_i \cdot \sum_{t=1}^{k} e_t \; - \sum_{i=1}^{M} \left[S_i \; + c_i\, m_i\, G_i \; + c_i\, h_i\, k_i \left(m_i\, G_i \left(1 - \frac{G_i}{2 k_i\, B} \right) + I_i \; + \frac{D_i \sum_{t=1}^{k} e_t\, B}{2} \right) \right]$$

(19)

Case 2.a

$$U = \sum_{i=1}^{M} P_i\, D_i\, B_i \sum_{t=1}^{k} e_t \; - \sum_{i=1}^{M} \left[S_i \; + c_i\, m_i\, G_i + \; V_i \left(I_{i-1} - IF_i \right) \; + \; c_i\, h_i\, k_i \left(\frac{m_i\, G_i^{\,2}}{2\,B} \left(\frac{m_i}{2 \sum_{t=1}^{k} e_t} - 1 \right) + \frac{I_i}{D_i\, B_i \sum_{t=1}^{k} e_t} \left(m_i\, G_i \; + \frac{I_i}{2} \right) \right) \right]$$

(20)

Case 2.b

$$U = \sum_{i=1}^{M} P_i\, D_i\, B_i \sum_{t=1}^{k} e_t \; - \sum_{i=1}^{M} \left[S_i \; + c_i\, m_i\, G_i + \; V_i \left(Q_i * - IF_i \right) \; + \; c_i\, h_i\, k_i \left(\frac{m_i G_i^{\,2}}{2\,B} \left(\frac{m_i}{2 \sum_{t=1}^{k} e_t} - 1 \right) + \frac{I_i}{D_i\, B_i \sum_{t=1}^{k} e_t} \left(m_i\, G_i \; + \frac{I_i}{2} \right) \right) \right]$$

(21)

Since each optimization is based on the evaluation of *t* periods, the Total Profit is equal to the sum of all Profit Formulations for each period:

$$\text{Profit}\left(G_i, k_i, e_t\right) = \sum_{t=1}^{M} U_t \tag{22}$$

And constrained by the following:

$$\sum_{i=1}^{M} G_i \leq \tau - \sum_{i=1}^{M} T_i \tag{23}$$

The decision variable is G_i. Notice that the right hand side of (23) is no longer *B*. This new right hand side is introduced because in general the time available for setups and production need not equal the time span of the basic period. From (19), (20) and (21), it can be seen that *B* may be interpreted as the real time span (clock time) over which demand is realized. In general, τ may be less than the clock time. For example, in a 24 hour period, the firm might operate for only one 8 hour shift.

3.1.2.2 Limitations

The stochastic version of the ELSP has some limitations due to its more complicated structure. Following some important issued will be explained:

A. <u>Concavity:</u>

The deterministic model (PELSP) had a Profit Formulation with a Concave shape and the second derivative test was done at the moment so to validate this assumption. The test was done without further issues and *strict concavity* was a certain fact on the model. For the Stochastic model, it was assumed that the Profit Formula still had a Concavity shape, but a derivative test had to be done. In the model described on this section, the SPLSP´s second derivatives showed that the Hessian was negative semidefinite for certain values of the variables *G* and *k*, so it is not possible to guarantee that the Profit Formula has a concave shape..

At first, this result had a bit of impact because some manual experiments on computer worksheets (Microsoft Excel) were done and the results showed us that the Profit formula behaved most of the time as Concave function. Besides, since the formula expresses Maximization it should have a concave shape.

B. <u>Optimization for the subproblem :</u>

This issue has to do with the fact that there isn't an analytic expression for the total profit for all the periods because the sequence of cases change due to the error instances and the variations on the quantities of production, which in other words means that it's difficult to find the exact analytical expression for the total profit and the first partial derivatives. For the prior reason, a numerical method with lineal search was chosen to optimize the subproblem.

The method does a sequential line search to find a near optimal solution. Since an optimal solution is not obtained, the upper bound is not exact, just a good approximation to the real optimal value.

C. Number of Periods

This issue has to do with the establishment of a certain number of periods to evaluate the Profit formulations. There are many options that could be used as viable alternatives for the optimization:

- 1 replica with a large number of periods (> 100)
- A group of replicas with a medium number of periods (from 20 to 60)
- A very large number of replicas with small number of periods (10 to 20)

D. Bias

This limitation has to do with a possible bias due to the way the simulation begins. It was decided to start the simulation with a beginning inventory of 0 units for each product. This assumption was made to simplify the start of the simulation but it is not a realistic one.

Other possibilities are:

- Setting the Beginning Inventory on an Initial Fixed value, determined by a service level. This would mean that the beginning inventory would have to be equal to a desired goal level (desired finishing inventory)
- Setting the Beginning inventory to any value above zero, just to allow it to be positive and also to allow a more random behavior.

E. Alternative Policies

The actual way that the problem is done consists on the following: Whenever the finishing inventory of a product dips below zero (negative inventory), emerges the decision of making a buy to at least buy the amount not sold to the clients and depending on the chosen policy the next period begins with 0 units or another positive value.

There are some other evaluation policies that can be used to emulate a more real behavior, such as:

- *Lost Sales*: This alternative would work almost equally as the actual policy but the difference would be that if a finishing inventory dips below zero (negative inventory) then a penalty cost would be assumed for the quantity of product that was not sorted to the client on the moment he needed it.

3.2 Solution Methodology

The prior section presented the description of the two variants of the ELSP model. On this following section, the solution methodology chose for each variant will be depicted and explained.

3.2.1 PELSP

In order to solve the PELSP, the problem is decomposed, formulating a master problem and subproblems. The master problem is a linear programming model to which columns, generated by solving the subproblems, are added at each iteration *v*. The master problem is solved to determine the values of the dual variables (shadow prices) which are, in turn, used to define the subproblems for the next iteration.

The master problem for iteration *v* is:

$$\max U = \sum_{i=1}^{M} \sum_{j=1}^{n(v,i)} U_{ij} X_{ij} \tag{24}$$

s.t.

$$\sum_{i=1}^{M} \sum_{j=1}^{n(v,i)} A_{ij} X_{ij} \le \tau - \sum_{i=1}^{M} T_i \tag{25}$$

$$\sum_{j=1}^{n(v,i)} X_{ij} = 1 \qquad \forall i\ , \tag{26}$$

with $0 \le X_{ij} \le 1,\ \forall i, j$.

Where:

U_{ij} : total profit for plan *j* of product *i*, in dollars,

A_{ij} : total production time available for plan *j* of product *i*, in days,

X_{ij} : proportions of use for plan *j* of product *i*,

$n(v,i)$: number of columns for product *i* in the *v* th linear program.

The values of U_{ij} and A_{ij} are found using the following expressions:

$$U_{ij} = \alpha_i \ln\left(R_i / \hat{d}_{ij}\right) \hat{d}_{ij} - \frac{S_i}{\hat{k}_{ij} B} - \frac{c_i\, h_i\, (m_i - \hat{d}_{ij})\, d_{ij}\, \hat{k}_{ij}\, B}{2 m_i} - c_i\, \hat{d}_{ij}, \tag{27}$$

and

$$A_{ij} = \frac{\hat{d}_{ij}\, \hat{k}_{ij}\, B}{m_i}, \tag{28}$$

where the $\hat{d}_{ij}$ and $\hat{k}_{ij}$ values are determined by solving a single item subproblem with costs augmented by the shadow prices found by the linear programming master problem.

The formulation of the unconstrained subproblem is:

$$\begin{aligned} \max\ U = & \sum_{i=1}^{M} \left[\alpha_i \ln\left(R_i / d_i\right)\right] d_i \\ & - \sum_{i=1}^{M} \left[\left(\frac{S_i}{k_i\, B} \right) + \left(\frac{c_i\, h_i\, (m_i - d_i)\, d_i\, k_i\, B}{2 m_i} \right) + c_i\, d_i + \lambda \left(\frac{d_i\, k_i\, B}{m_i} \right) \right], \end{aligned} \tag{29}$$

where λ is the shadow price of (25). The solution procedure will be described in terms of a *Main* routine and the procedures performed at various steps. A flow chart of the *Main* routine is shown in figure 2. The *Main* routine proceeds as follows:

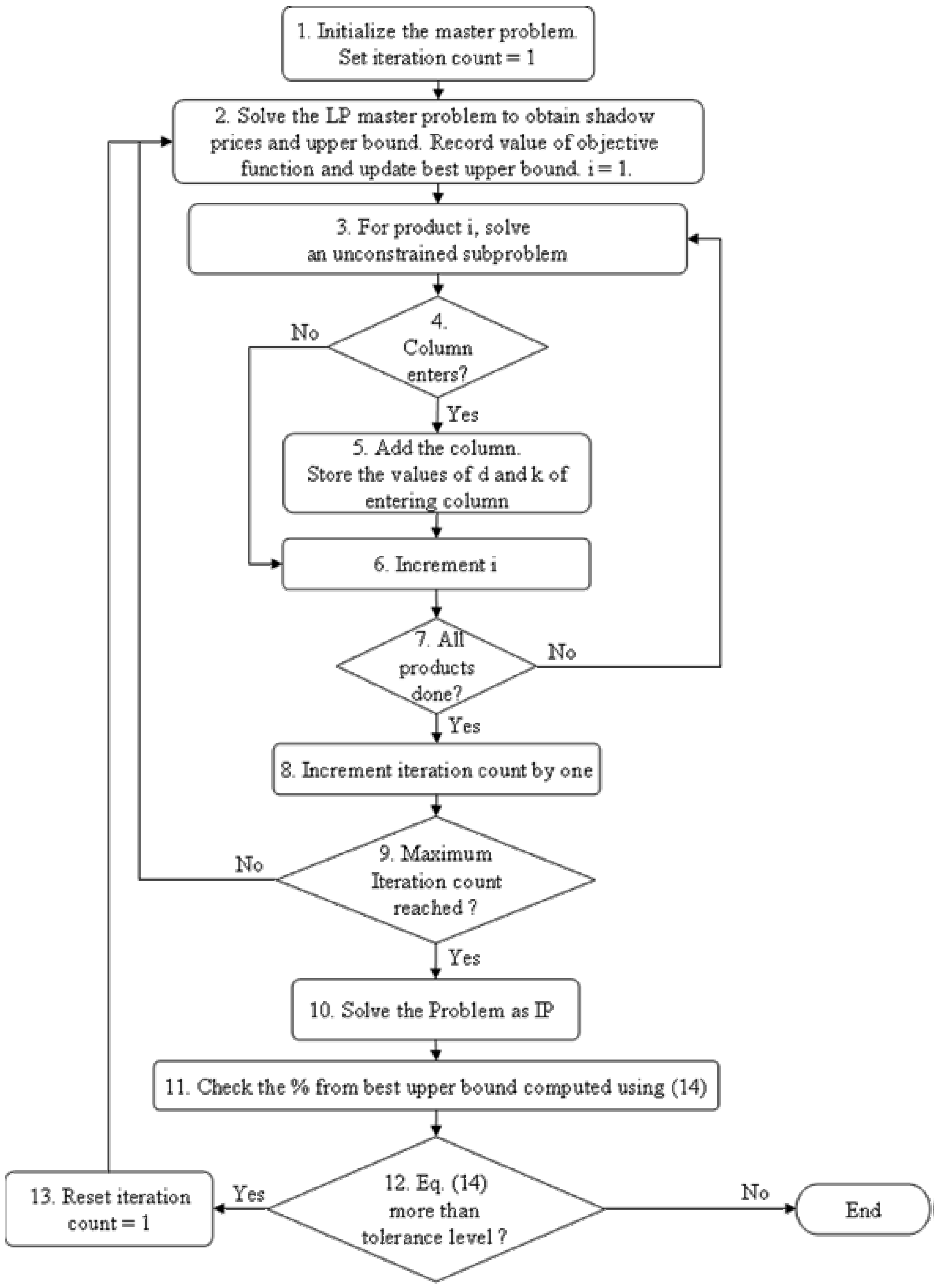
1. Initialize the master problem. Set iteration count = 1
2. Solve the LP master problem to obtain shadow prices and upper bound. Record value of objective function and update best upper bound. i = 1.
3. For product i, solve an unconstrained subproblem
4. Column enters?
No
Yes
5. Add the column. Store the values of d and k of entering column
6. Increment i
7. All products done?
No
Yes
8. Increment iteration count by one
9. Maximum Iteration count reached ?
No
Yes
10. Solve the Problem as IP
11. Check the % from best upper bound computed using (14)
12. Eq. (14) more than tolerance level ?
Yes
No
13. Reset iteration count = 1
End

Figure 2. Flowchart of Routine *Main*.

Block 1 of *Main*: Initialize the master problem and set iteration count = 1. In order to initialize the master problem, the subproblem (29) for each product is solved with $\lambda = 0$. Solving these subproblems yields initial values of d_i and k_i for each product. Leaving the k_i values unchanged, the solution for each of the products is modified to obtain two sets of columns whose coefficients are found using (27) and (28). To generate the first set of columns, the demand and production is scaled down in the solutions for each product so that if the master problem contained only this set of columns, (25) would be satisfied with a strict inequality.

The second set of columns is generated by scaling up the demand and production in the solutions for each product so that if the master problem contained only this set of columns, (25) would be violated. The reason for generating these two sets of columns is to guarantee that the master problem will be feasible and so that (25) will be tight in the solution of the master problem (24) – (26). This introduces inefficiency in the case that the unconstrained solutions of the subproblems for each product constitute a feasible (and therefore optimal) solution to the complete problem but it worked well in practice. The scaling used in the implementation was to scale up by a factor of 1.75 and down by a factor of 0.25 with further scaling by 1.1 and 0.1, respectively performed if needed to obtain the desired initial columns.

Block 2 of *Main*: Solve the LP master problem to obtain shadow prices and an upper bound. Also record the objective value and update the value of the best upper bound

(BUB) if the current bound is the tightest so far. Each time the master problem is solved, an upper bound is computed using the following expression:

$$UB = OV - \sum_{i=1}^{n} \theta_i \ , \tag{30}$$

where, *OV* is the objective value given by (24) and θ_i is given by:

$$\theta_i = [A_i \ln(R_i / d_i)] d_i - \left[\left(\frac{S_i}{k_i B} \right) + \left(\frac{c_i h_i (m_i - d_i) d_i k_i B}{2 m_i} \right) + c_i d_i + \lambda \left(\frac{d_i k_i B}{m_i} \right) \right] - \lambda_i , \tag{31}$$

where λ_i is the shadow price of (26) for product *i*. Each time an upper bound is computed, the value is compared with the BUB found so far. If the latest upper bound is an improvement (lower), it is stored as the new BUB. At the end of the solution procedure, the BUB is used to determine a percentage difference from BUB measure of the quality of the final integer solution.

Block 3 of *Main*: For each product, solve an unconstrained subproblem (29) using the value of λ found in Block 2 of *Main*. The subproblem involves finding values of demand (*d*) and the basic period multiplier (*k*) for a single product. The solution procedure is shown as a flowchart in figure 3. The procedure *SolveSub* proceeds as follows:

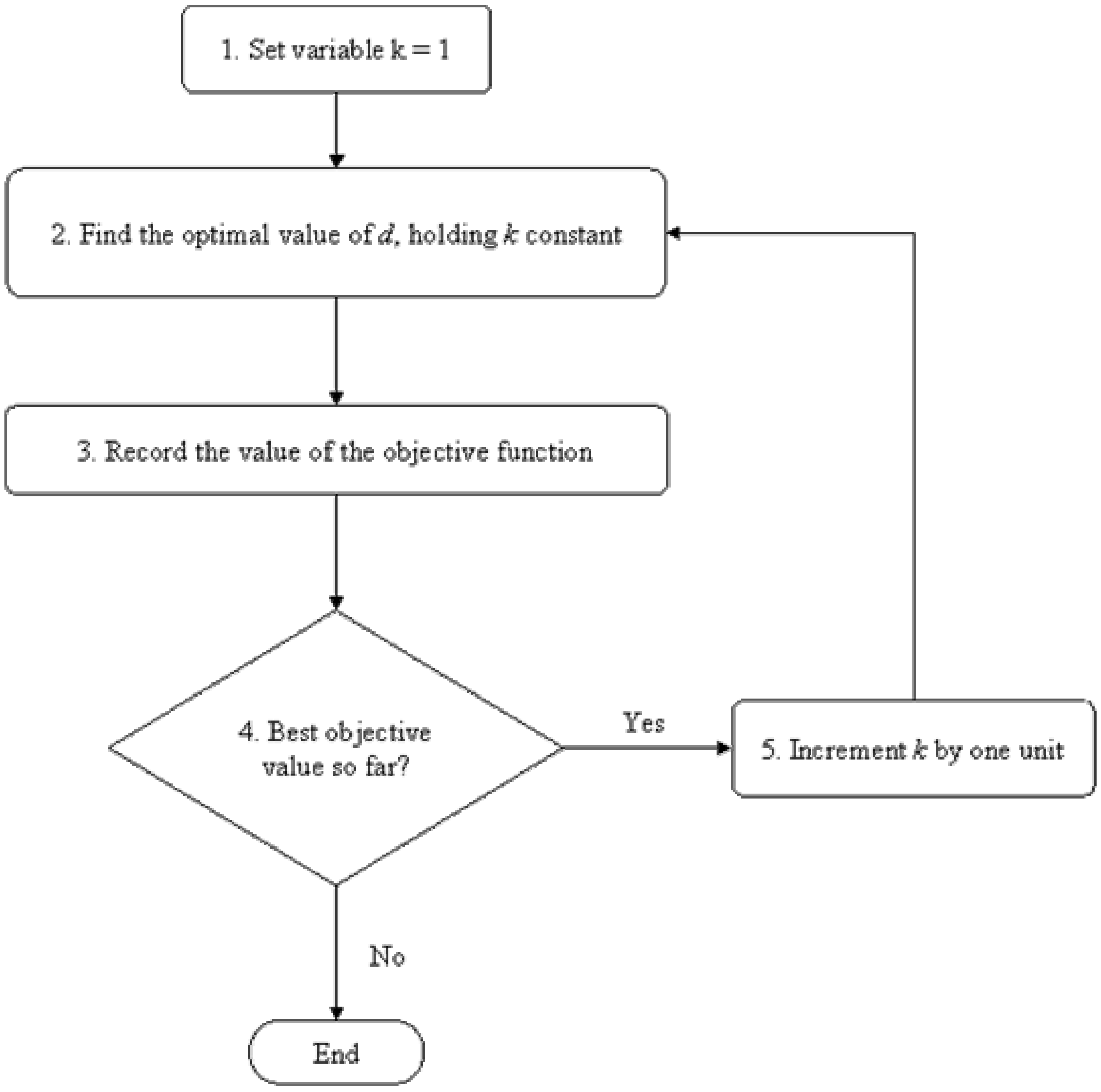
1. Set variable k = 1
2. Find the optimal value of *d*, holding *k* constant
3. Record the value of the objective function
4. Best objective value so far?
Yes
5. Increment *k* by one unit
No
End

Figure 3. Flowchart for Procedure *SolveSub*

Block 1 of *SolveSub*: Set $k = 1$.

Block 2 of *SolveSub*: A modified bisection method is used to find the optimal value of d, holding k constant.

For information on the basic bisection method, see [32] for a simple explanation or [60] for an explanation and a computer program. The search range for the routine needs to be specified with two values: a Lower Limit (LL) and an Upper Limit (UL). For the implementation of this routine the values set were: LL = 0.001 and UL = value of m, the production capacity. The value of LL was chosen to avoid division by zero in (29). The value of UL is a natural limit that guarantees that the demand rate does not exceed the production rate. This is a standard assumption of the EPQ, on which the ELSP and PELSP are based.

The objective function is unimodal for a constant value of k so it can be optimized using standard methods. The routine we programmed is based on the one proposed in [60]. It assumes that the optimal objective value is to be found between the LL and UL and that the first derivative of the objective function has opposite sign at LL vs. UL. In the PELSP, this may not always be the case.

Figures 4, 5, and 6 show examples of the possible form of the objective function between the LL and UL, LL being the leftmost value of d and UL the rightmost value. The figures are depicted on the following pages:

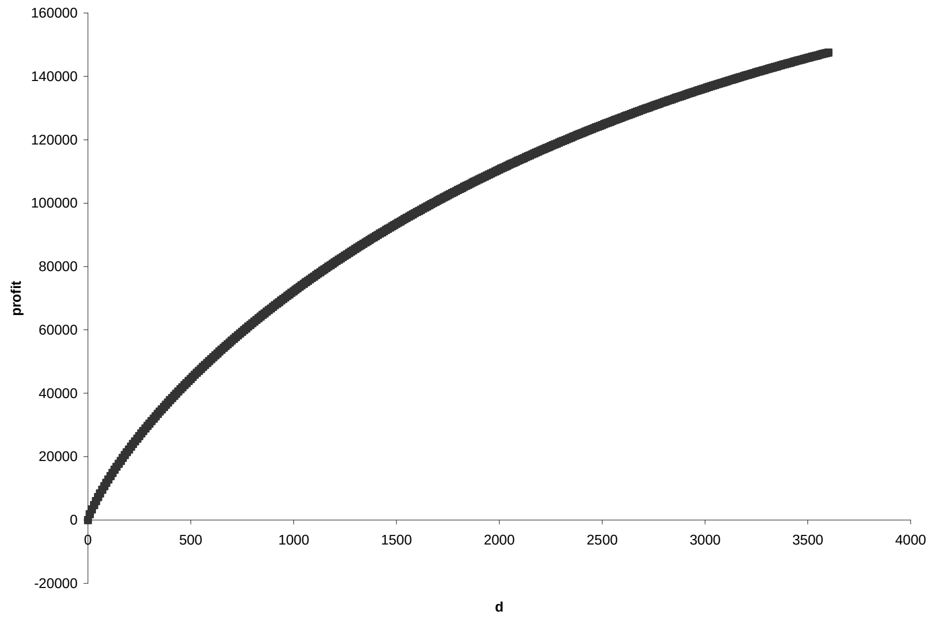

Figure 4. Case 1, B = 1, S = 33.5932, c = 20.6260, m = 3474.7896,

T = 0.0683, h = .000295

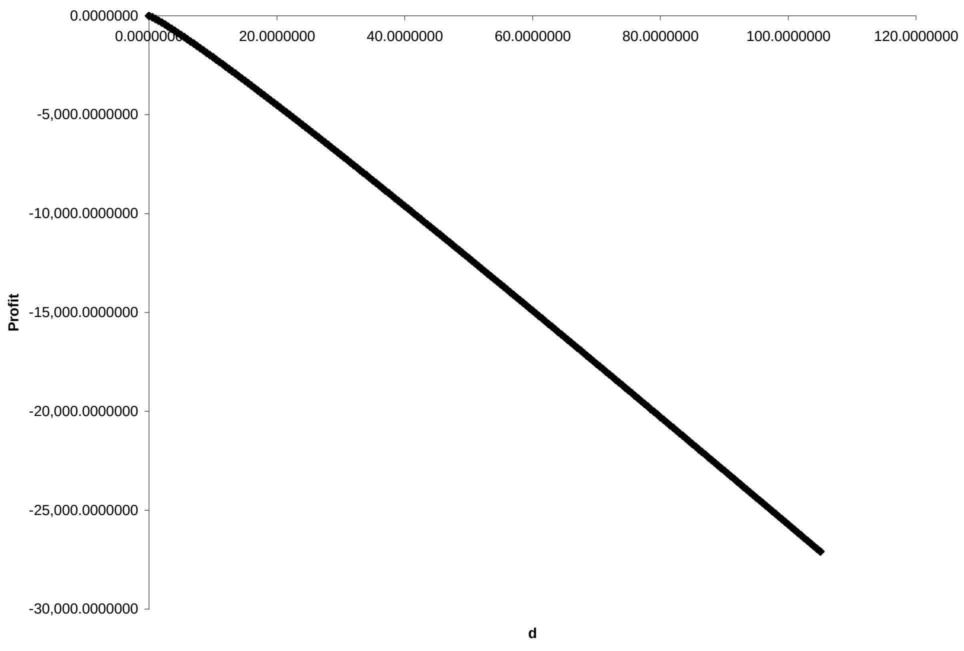

Figure 5. Case 2, B = 1, S = 500, c = 1000, m = 1000,

T = 0.0683, h = .0000833

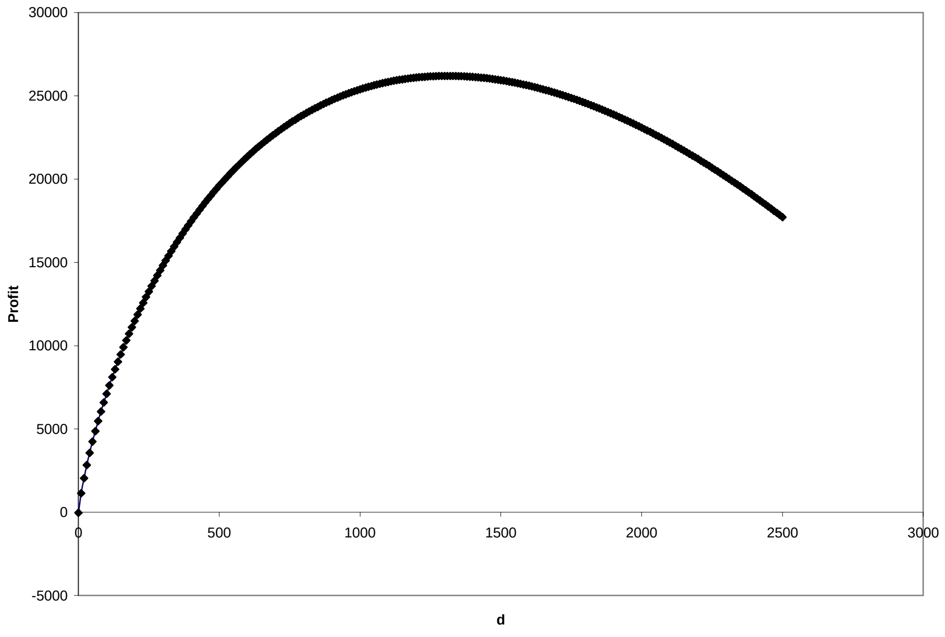

Figure 6. Case 3, B = 1, S = 33.5932, c = 20.6260, m = 2500,

T = 0.0683, h = .000295

The three cases that need to be considered are:

1. The first derivative of the objective function with respect to d is positive at both d = LL and d = UL. In this case, the optimal value of d is UL. See figure 4.
2. The first derivative of the objective function with respect to d is negative at both d = LL and d = UL. In this case, the optimal value of d is LL. See figure 5.
3. Otherwise, use the normal procedure [60]. See figure 6.

By considering the above cases explicitly, (29) could be solved correctly regardless of the positions of LL and UL relative to the objective function.

Block 3 of *SolveSub*: Record the value of the objective function.

Block 4 of *SolveSub*: If the latest solution has the best objective function so far, continue to Block 5 of *SolveSub*, otherwise end.

Block 5 of *SolveSub*: Increment *k* by one and return to Block 2 of *SolveSub*.

Block 4 of *Main*: Decide if the solution provides a column that may be added to the master problem. To determine if a column is to be added to the master problem it is necessary to evaluate expression (31) for each product *i*. If $\theta_i > 0$ and the value of (27) is positive, the new column may be allowed to enter the master problem. In this implementation, adding columns with values of θ_i below a threshold value of 0.1 was disallowed since it was found empirically that adding columns that did not significantly improve the objective value was inefficient.

Block 5 of *Main*: Add the entering column. Store the *d* and *k* values of the entering column.

Block 6 of *Main*: Increment the product index, i, by one.

Block 7 of *Main*: If subproblems for all products have been solved, proceed to Block 8 of *Main*, otherwise return to Block 3 of *Main* to solve another subproblem.

Block 8 of *Main*: Increment iteration count by one.

Block 9 of *Main*: If the maximum iteration count has been reached continue to Block 10 of *Main*, otherwise return to Block 2 of *Main*. This value corresponds to the number of iterations allowed for the problem to obtain the best solution available. A maximum count of 20 iterations was used in this implementation of the procedure.

Block 10 of *Main*: Solve the problem as an IP, adding the integrality constraints: $X_{ij} \in \{0,1\} \forall i, j$.

Block 11 of *Main*: Check the % from best upper bound using the following expression:

$$\frac{\text{BUB} - \text{Current Solution}}{\text{BUB}} \times 100\% . \tag{32}$$

Block 12 of *Main*: If expression (33) evaluates to more then a specified tolerance (5% in our implementation), continue to Block 13 of *Main*, otherwise end.

Block 13 of *Main*: Reset the iteration count to 1 and return to step 2. Else, end.

3.2.2 SPLSP

In order to solve the SPLSP, the problem is first decomposed, formulating a master problem and subproblems. The master problem is a linear programming model to which columns, generated by solving the subproblems, are added at each iteration *v*. The master problem is solved to determine the values of the dual variables (shadow prices) which are, in turn, used to define the subproblems for the next iteration.

The master problem for iteration *v* is:

$$\max U = \sum_{i=1}^{M} \sum_{j=1}^{n(v,i)} U_{ij} \, X_{ij} \tag{33}$$

s.t.

$$\sum_{i=1}^{M} \sum_{j=1}^{n(v,i)} A_{ij} \, X_{ij} \le \tau - \sum_{i=1}^{M} T_i \tag{34}$$

$$\sum_{j=1}^{n(v,i)} X_{ij} = 1 \qquad \forall i\,, \tag{35}$$

with $0 \le X_{ij} \le 1,\ \forall i, j$.

Where:

U_{ij} : total profit for plan *j* of product *i*, in dollars,

A_{ij} : total production time available for plan *j* of product *i*, in days,

X_{ij} : proportions of use for plan *j* of product *i*,

$n(v,i)$: number of columns for product *i* in the *v* th linear program.

The values of U_{ij} and A_{ij} are found using the following expressions:

$U_{ij} =$ Sum of Profit Formulas (19), (20) and/or (21), for all *t* periods (36)

And

$$A_{ij} = G_{ij}\,, \tag{37}$$

Where the value of G_{ij} is determined by solving a single item subproblem with costs augmented by the shadow prices found by the linear programming master problem. The formulation of the unconstrained subproblem is shown:

$$\max U_{..} = U_{ij} - \lambda \cdot G_{ij} \tag{38}$$

where λ is the shadow price of (33) and G_{ij} is the use of capacity on the base period, for product *i* on production plan *j*

The solution procedure will be described in terms of a *Main* routine and the procedures performed at various steps.

3.2.2.1 Routine Main

The flow of the main routine is as follows:

1. Initialize the master problem (procedure explained below in subsection 3.2.2.2).Set iteration count = 1.
2. Solve the LP master problem to obtain shadow prices and an upper bound (procedure explained below in subsection 3.2.2.4). Record best upper bound.
3. For each product:
    a. Solve an unconstrained subproblem (procedure explained below in subsection 3.2.2.5).
    b. Decide if the solution provides a column that may be added to the master problem (procedure explained blow in subsection 3.2.2.3).
4. Add the entering columns, if any. Store the *G* and *k* values of the entering columns.
5. Increment the iteration count by one.
6. Return to step 2 unless a maximum iteration count is reached. This value corresponds to the number of iterations allowed for the problem to obtain the best solution available. We used a maximum count of 20 iterations in our implementation of the procedure.
7. Solve the problem as an IP, adding the integrality constraints $X_{ij} \in \{0,1\} \forall i, j$.
8. Check the % from best upper bound computed using the following expression:

$$\frac{\text{Best Upper Bound} - \text{Current Solution}}{\text{Best Upper Bound}} \times 100\% \tag{39}$$

9. If expression (39) evaluates to more then a specified tolerance (5% in our implementation), reset the iteration count to 1 and return to step 2. Else, end.

3.2.2.2 Initialization

In order to initialize the master problem, the subproblem for each product is solved with $\lambda = 0$. The solutions determine the periods in which setups are made. Respecting the setup schedule, the solution for each of the products is modified to obtain two sets of columns whose coefficients are found using (36) and (37). To generate the first set of columns, the demand and production is scaled down in the solutions of all products so that if the master problem contained only this set of columns, (34) would be satisfied with a strict inequality. The second set of columns is generated by scaling up the demand and production in the solution of all products so that if the master problem contained only this set of columns, (34) would be violated. The reason for generating these two sets of columns is to guarantee that the master problem will be feasible and so that (34) will be tight in the solution. This introduces inefficiency in the case that the unconstrained solutions of the subproblems for each product constitute a feasible (and therefore optimal) solution to the complete problem but it worked well in practice. The scaling used in the implementation was to scale up by a factor of 1.25 and down by a factor of 0.25 with further scaling by 1.1 if needed to obtain the desired initial columns.

3.2.2.3 Column entry

To determine if a column is to be added to the master problem we evaluate the following expression for a product *i*:

$$\theta_{ij} = U_{ij} - \lambda G^* - \lambda_i \ , \tag{40}$$

Where λ_i is the shadow price of (33) for product *i*. If $\theta_{ij} > 0$ and the value of (34) is positive, the new column may be allowed to enter the master problem. In this implementation, adding columns with values of θ_i below the threshold value of one was disallowed since it was found empirically that adding columns that did not significantly improve the objective value was inefficient.

3.3.4 Computing upper bound

Each time the master problem is solved, an upper bound is computed using the following expression:

$$UB = OV - \sum_{i=1}^{n} \sum_{j=1}^{m} \theta_{ij} \ , \tag{41}$$

Where, *OV* is the objective value given by (33). Each time an upper bound is computed, the value is compared with the best upper bound (BUB) found so far. If the latest upper bound is an improvement (lower), it is stored as the new BUB. At the end of the solution procedure, the BUB is used to determine a % difference from BUB measure of the quality of the final integer solution using expression (39).

3.3.5 Solving the subproblem

The subproblem involves finding values of Production Time (*G*) and the basic period multiplier (*k*) for a single product. The solution procedure is the following:

1. Set $k = 1$.
2. Use a Manual Optimization method (explained on section 3.2.2.6) to find the optimal value of *G*, holding *k* constant.
3. Record the value of the objective function.
4. If the last solution has the best objective function so far, continue to step 5; else, stop.
5. Increment *k* by one and return to step 2.
6. Once the Optimal values of *G* and *k* are determined, another intern optimization is done
7. Record the new optimal values of *G* and *k*

3.2.2.6 Search Optimization Method

This method consists on evaluating a certain amount of periods beginning with a given value of the first variable, k. It is started by iterating with successive values of the G variable (using a Step Size of 0.1 units), in order to obtain the Expected Value of the Profit. The program stops when the Expected Profit of an *n* period has a lower value than the Expected Profit on the *n-1* period.

When the Method finds the best solution (values of k* and G*) another optimization routine is performed to further improve the solution. The routine consists on fixing the value of k and make a new optimization beginning on the (G* - 0.1) value and iterating with a more refined step size (0.01 units) until the program finds the Maximum Value of the Expected Profit. The limit for this iteration is the (G*+ 0.1) value.

Chapter 4. Experimentation

4.1 PELSP computational experience

In order to test the solution methodology proposed for the PELSP, on section 3.2.1, a full factorial experimental design was employed. The factors varied systematically are number of products, length of *B*, and capacity restrictedness. The number of products was set at 2, 10, or 30. The length of the basic period was set at 1 day, 7 days, or 30 days. Capacity restrictedness was set as follows. In the ELSP problem, the values of *B* and τ are equal. Here, however, the two values don't have to be the same. In order to include problems that differ in how constrained they are by capacity, problems with τ equal to 60%, 80%, or 100% of the *B* were generated. For example, if *B* = 7 days, then the three levels of τ are 4.2 days, 5.6 days, and 7 days, respectively. For each combination of three factor levels described above, five random instances were generated to obtain a total of 135 problem instances.

The random problem instances were constructed by generating values of setup cost (*S*), unit cost (*C*), production rate (*m*), holding cost (*h*), elasticity (α), market size (*R*) and setup time (*t*) for each product in the instance from uniform distributions. The upper and lower limits of the uniform distribution used to generate values of the first six parameters are shown in table 2.

	Upper value	**Lower value**	**Units**
S	50	30	$ / setup
c	40	20	$ / unit
m	4000	2000	units/day
h	0.1	0.05	% / day
α	40	20	--
R	100000	10000	units/day

Table 2. Value ranges for product parameters

Table 2 shows the upper and lower limits on setup times used in the experiments. The distribution of the setup times was varied depending on the number of products and the length of the *B* in order to obtain a reasonable percentage of feasible problems. Using long setup times in problems with many products or short basic periods caused many of the generated problems to violate constraint (7) .A day was assumed to equal 8 hours.

As an example, consider a problem instance with 30 products and a *B* of 1 day. When the lower limit of the distribution was set to 0.05 days and the upper limit of the distribution was set at 0.3, practically all the generated problem instances were infeasible with respect to (7). With the lower limit at 0.04 and the upper limit at 0.07, 27% of the generated problems were feasible. The values actually used are those shown in table 3. These values provided a reasonable percentage of feasible problems.

	***B* (days)**		
# products	**1**	**7**	**30**
2	[0.3, 0.05]	[0.3, 0.05]	[0.3, 0.05]
10	[0.07, 0.04]	[0.1, 0.05]	[0.1, 0.05]
30	[0.025, 0.01]	[0.1, 0.05]	[0.1, 0.05]

Table 3. Setup time ranges for different sets of experiments (in days)

The results of the experiments are summarized in tables 4, 5, and 6, corresponding to problem instances of size 2, 10, and 30, respectively.

***B* (days)**		**1**			**7**			**30**	
Capacity (days)	**0.6**	**0.8**	**1**	**4.2**	**5.6**	**7**	**18**	**24**	**30**
Average % from BUB	0.227265	0.11310	0.076959	0.088093	0.06359	0.057011	0.098762	0.13289	0.097532
Maximum % from BUB		0.40659			0.20201			0.24944	

Table 4. Results for problems with 2 products.

***B* (days)**		**1**			**7**			**30**	
Capacity (days)	**0.6**	**0.8**	**1**	**4.2**	**5.6**	**7**	**18**	**24**	**30**
Average % from BUB	0.032717	0.00483	0.004402	0.003307	0.00310	0.002137	0.003682	0.00230	0.002423
Maximum % from BUB		0.092204			0.00641211			0.00583	

Table 5. Results for problems with 10 products.

***B* (days)**		**1**			**7**			**30**	
Capacity (days)	**0.6**	**0.8**	**1**	**4.2**	**5.6**	**7**	**18**	**24**	**30**
Average % from BUB	0.010905	0.00415	0.002818	0.003344	0.00220	0.003347	0.001747	0.01464	0.001871
Maximum % from BUB		0.015521			0.008921			0.06559	

Table 6. Results for problems with 30 products.

As can be seen in tables 4, 5, and 6, the proposed solution method finds solutions that are provably very close to optimal. The performance appears to be generally better for problems with a larger number of products, with the maximum % from upper bound for problems of 2, 10, and 30 products being 0.4066, 0.0922, and 0.0656, respectively. The corresponding average % from upper bound values are 0.1061, 0.0065, and 0.0050. This trend agrees with the observations of Trigeiro et al. [54] in regard to the Capacitated Lot Sizing Problem (CLSP), a discrete-time, finite horizon lot sizing problem. Also, Haugen et al. [31] observe a similar behavior with a CLSP problem with pricing (PCLSP). In contrast to the non-pricing ELSP, the PELSP model seems relatively easy to solve. A similar observation is made by Haugen et al. [31] in regard to the PCLSP.

Also, in the PELSP, The LP basis is of size n + 1 where *n* is the number of products; this implies that in the LP solution each product but one is represented by only one column and the last product is represented by a combination of two columns. In other words, the LP solution draws a result very near to the IP solution, which is why the solutions are provably very close to optimal.

The sensitivity to the capacity restrictedness of the problems is difficult to characterize. For problems with 2 products it can be seen in table 4 that the behavior differs depending on the value of *B*. For *B* = 1, the average % from upper bound decreases with increasing capacity. This is also the case for *B* = 7. However, for *B* = 30, the average % from upper bound first increases and then decreases with increasing capacity. A similar behavior can be seen for problems with 10 products in table 5. For problems with 30 products yet another behavior is observed. For *B* =1, the average % from upper bound decreases with increasing capacity. However, for *B* = 7 and *B* = 30 the average % from upper bound first decreases and then increases with increasing capacity. The lack of a clear trend may be due to the fact that in the PELSP, demand can always be reduced by increasing the price. Hence, it is just as easy to find a near optimal solution when capacity is more restricted as when it is more abundant. The general conclusion is that the quality of the solution remains very good over a broad range of problem sizes and degrees of capacity restrictedness. The average solution computation times for problems of 2, 10, and 30 products was 3.07, 6.35, and 13.36 seconds, respectively.

The effect of varying the maximum number of iterations (see section 3.2.1, Block 9 of *Main*) and the tolerance (see section 3.2.1, Block 12 of *Main*) was studied by re-solving a subset of the test problems with these parameters set to different values. In order to study the effect of reducing the number of iterations, five of the original test problems for each problem size where resolved with the number of iterations set to 5 and 30. The original number of iterations was 20. The tolerance was kept at the original 5% while solving these problems. The results are summarized in table 7. For all problem sizes, the number of iterations was found to directly affect solution time. The effect of number of iterations was found to be that increasing the number of iterations to 30 did not improve the solution quality but reducing it to 5 did reduce the solution quality noticeably. The original value of 20 iterations seems to be a good compromise between solution speed and quality.

Problem size		**2**			**10**			**30**	
# iterations	**5**	**20**	**30**	**5**	**20**	**30**	**5**	**20**	**30**
Avg. % from UB	0.9221	0.0769	0.0769	0.0308	0.0044	0.0044	0.0456	0.0028	0.0028
Solution time	1.12	3.07	4.20	2.86	5.94	7.43	10.25	13.36	18.45

Table 7. Effect of number of iterations.

The effect of varying the tolerance was studied by re-solving five randomly selected problems of each size with the tolerance set at 2% and 8% as compared to the original 5%. The number of iterations was kept at the original 20 while solving these problems. Varying this parameter was found to have no effect on the solution quality on any of the test problems. The reason for this is that the procedure is finding solutions that are well below 2% from the upper bound. The purpose of this parameter is to force the procedure to continue searching for better columns in the rare case that the first 20 iterations do not produce a within tolerance solution. In our testing, we found that 20 iterations were sufficient for all the test problems.

Also, in order to assess the performance of the methodology when the demand function is of a different type, a small set of problems were solved using a linear demand function instead of an exponential demand function. The linear demand function is the following:

$$d_i\,(P_i) = R_i - a_i\,P_i\,, \tag{42}$$

where R_i is a market size parameter for product *i* and a_i is a price scaling constant. The single-item subproblems are slightly harder to solve with this function because the price has an upper bound of R_i / a_i, beyond which the demand becomes negative. Care must be taken not to exceed this maximum price while solving the subproblem. The testing consisted of solving 15 problems with 2 products and 15 problems with 10 products.

The problem instances were constructed by randomly generating values of setup cost (*S*), unit cost (*C*), production rate (*m*), holding cost (*h*), price scaling factor (*a*), market size (*R*) and setup time (*t*) for each product in the instance from uniform distributions. The upper and lower limits of the uniform distribution used to generate values of the first six parameters are shown in table 8. The setup times for the instances with 2 products and 10 products were generated from uniform distributions with ranges (.05, .3) and (.04, .07), respectively. The average percentages from upper bound were 0.0156 and 0.197 for problems with 2 and 10 products, respectively.

The average CPU times were 4.33 and 18.05 seconds, respectively. These limited tests show that the solution quality in terms of percentages from upper bound remains similar while the solution times increased somewhat while remaining short.

	Upper value	Lower value	Units
S	50	30	$ / setup
c	40	20	$ / unit
m	4000	2000	units / day
h	0.1	0.05	% / day
R	5000	4000	units / $ --
a	150	100	units / day

Table 8. Value ranges for product parameters

4.1.1 Managerial Insights

In order to gain managerial insights, the mathematical form of the model was examined and a series of computational experiments were performed. It is important to emphasize that these observations are relevant to the base period solution form and may not apply to more general solution forms.

The general behavior of the solutions and objective value was studied by solving two-product problems. A base problem was defined with all parameter values set at the midpoints of the ranges shown in table 2. Modified problems where then solved to observe the changes in the objective value and the d_i values. The modified problems where identical to the base problem with the exception of a change in exactly one parameter, set to either the lower or upper limit of the range shown in table 2.

The general behavior of the objective value is to increase (decrease) when $R, \alpha,$ or m are increased (decreased) and to decrease (increase) when S, c, T, or h are increased (decreased). This is not unexpected since a larger R increases the size of the market, a larger α allows a higher price to be charged for the same demand, and a larger m effectively increases the production capacity. Likewise, an increase in the costs (S, c, h) would be expected to reduce the profit. An increase in T effectively reduces available production capacity so a decrease in profit is also unsurprising. In terms of production rates, increasing $R, \alpha,$ or m resulted in greater demand (d) for the changed product and reducing T resulted in greater demand for both products. Reducing c resulted in greater demand for the changed product as well. The effect of changing *S* and *h* was very slight. Changes in the value of *h* tend to have little effect on demand because inventory carrying costs are relatively small compared to the gross profit (sales minus production costs) in an optimized solution. The small effect of changing *S* can be understood by observing that the partial derivative of the objective function with respect to demand (*d*) does not include *S*. Thus, demand is not affected by *S* unless the change in *S* prompts a change in *k*. Also, notice that a decrease in *S* may prompt an increase in *k*, which will tend to further reduce the value of the setup cost term. On the other hand, an increase in *S* may prompt a decrease in *k*, which will tend to amplify the increase in the setup cost term. However, if the original value of *k* is low (which is common in reasonably capacity constrained problems), the possible decrease in *k* is limited, thus limiting the effect of the change in *S* on demand (*d*).

Additional investigations were focused on better understanding the effect of reducing setup times and setup costs. Both types of reductions were studied separately. Setup time reductions will be addressed first. By inspecting constraint (7) it is evident that decreasing setup times increases the available capacity in the base period. Since this capacity increase results in a relaxed problem as compared to the problem before setup time reduction, the objective value will increase. It is interesting to notice that the increase is completely independent of which product's setup time is reduced. For a manager using a base period schedule, this implies that setup time reduction projects should be selected based on cost efficiency (most reduction for the money) rather than on other criteria such as longest setup times first, most frequent setups first, etc.

In regard to setup cost reduction, a similarly interesting observation can be made by examining the objective function (6). When all products have a *k* value equal to one, the increase in objective value resulting from setup cost reductions will be independent of which product's setup cost is reduced. This is due to two factors. The first is that each product's setup cost term in the objective function has the same denominator when *k* is equal to one for all products. The second is that reducing setup costs will never change a *k* value in this case. This can be understood by observing that with *k* equal to one, the setups are already occurring as often as possible and a setup time reduction can only encourage more frequent setups. The solution form with all *k* values equal to one is of particular importance because the optimal solution is often of this form when the problem becomes more capacity constrained. This is to be expected because producing in every period minimizes the base period capacity requirements that are constrained by (7). For a manager using a base period schedule with all values of *k* equal to one, this implies that setup cost reduction projects should be selected based on cost efficiency (most reduction for the money) rather than on other criteria such as largest setup costs first, most frequent setups first, etc.

In order to better understand the effect of setup cost reduction when not all *k* values are equal to one, a number of two-product problem instances were found with optimal solutions with *k* not equal to one. Each problem instance was then solved with different percentages of setup cost reduction applied first to one product and then the other (same percent reduction applied to each product). No simple pattern was discovered. The following numerical example will illustrate the complexity.

Product 1 parameters α, R, S, c, m, T, and h are 29.25, 42599.92, 67.0, 55.0, 35000.0, 0.324, and 0.0501, respectively. Product 2 parameters α, R, S, c, m, T, and h are 37.90, 27719.42, 31.98, 35.91, 35000.0, 0.311, and 0.0839, respectively. The values of B and τ are both equal to 1.0. In this example, the solution of the original problem has k = 2 for product 1 and k = 1 for product 2. Product 2 has a setup cost that is less than half that of product 1. With a 14.77% setup cost reduction, it was found that reducing the setup cost for product 2 was better. The same preference for product 2 was observed for setup cost reductions of 29.55, 44.32, 59.09, 73.87, 88.64%. When a reduction of 96.03% was applied, however, it was more advantageous to apply the reduction to product 1. The value of k for product 1 changed from 2 to 1 when the 73.87% reduction was applied to product 1. The following two points are of interest:

1) Even though the setup cost for product 2 was less than half that for product 1, it was still the preferred choice over a wide range of percent reductions. This was despite the fact that the value of k for product 1 did not change until a reduction of 73.87% was applied. This is hard to anticipate intuitively.

2) At some point between a reduction of 88.64 and 96.03% it becomes more advantageous to apply the reduction to product 1. This was despite the fact

that the value of *k* for product 1 changed to one when a reduction of 73.87% was applied. This switching of preference is hard to anticipate intuitively.

This example illustrates that intuition can lead to incorrect decisions for this type of problem due to complex interactions. This may have been anticipated based on the nonlinearity of the problem. When the current solution contains values of *k* greater than one, it is recommended to determine the effect of a proposed setup cost reduction by finding the (near) optimal solution using the method proposed in this paper rather than relying on intuition.

4.2 SPLSP Computational Experience

In order to test the solution method proposed for the SPLSP model, it was decided to perform a number of different experiments in order to validate the model and also to test its robustness. The SPLSP was tested using three different size problems: 2, 10 and 30 products. The random problem instances were constructed by generating values of setup cost (*S*), unit cost (*C*), production rate (*m*), setup time (*T*), holding cost (*h*), market size parameters (*a* and *b*) and vendor price (*V*) for each product in the instance from uniform distributions. Different values of the instances for the three size problems were used so that the production schedules would be feasible. The upper and lower limits of the uniform distribution used to generate values of the first six parameters are shown in tables 9, 10 and 11.

	Upper value	**Lower value**	**Units**
S	60	40	\$ / setup
c	8	4	\$ / unit
m	250	125	units / day
G	0.1	0.05	days
h	0.25	0.2	% / year
V	9	5	\$ / units
a	20	12	--
b	200	160	--

Table 9. Value ranges for parameters (2 products)

	Upper value	**Lower value**	**Units**
S	60	40	\$ / setup
c	8	4	\$ / unit
m	500	400	units / day
G	0.01	0.005	days
h	0.25	0.2	% / year
V	9	5	\$ / units
a	20	12	--
b	200	160	--

Table 10. Value ranges for parameters (10 products)

	Upper value	**Lower value**	**Units**
S	80	40	$ / setup
c	8	4	$ / unit
m	2000	1500	units / day
G	0.006	0.003	days
h	0.25	0.2	% / year
V	9	5	$ / units
a	20	12	--
b	200	160	--

Table 11. Value ranges for parameters (30 products)

4.2.1 Algorithm Parameters

The objective of this set of experiments was to observe the sensibility of the model due to changes on two algorithm parameters: the *maximum number of iterations* and the *% deviation from UB threshold*. The tests were done through the testing of 2 problems for each problem size (2, 10 and 30 products). Two other parameters had fixed values: the number of base periods (48) and the value of the standard deviation of the error instances (0.05). Initially, to proceed with the experiments, four levels for each parameter were tested. In the case of the *maximum number of iterations* these were the values used: 5, 10, 20 and 30 iterations. Regarding the *% deviation from UB threshold,* the values used were: 1%, 2%, 5% and 7.5%

Following (next page), in Tables 12, 13 and 14 it can be seen the results of the testing for the *maximum number of iterations*, for each problem size:

Number of Iterations		**2 products**	
		1st set	***2nd set***
5	**UB**	1,215.43	1,229.03
	IP	1,207.71	1,215.91
	% deviation from UB	*0.6352%*	*1.0675%*
	CPU time (seconds)	*1.44*	*0.98*
10	**UB**	1,215.43	1,229.03
	IP	1,207.71	1,215.91
	% deviation from UB	*0.6352%*	*1.0675%*
	CPU time (seconds)	*2.21*	*2.06*
20	**UB**	1,215.43	1,229.03
	IP	1,207.71	1,215.91
	% deviation from UB	*0.6352%*	*1.0675%*
	CPU time (seconds)	*2.43*	*2.45*
30	**UB**	1,215.43	1,229.03
	IP	1,207.71	1,215.91
	% deviation from UB	*0.6352%*	*1.0675%*
	CPU time (seconds)	*3.58*	*3.71*

Table 12. Maximum number of iterations testing results for 2 product problems

Number of Iterations		**10 products**	
		1st set	***2nd set***
5	**UB**	6,447.35	6,449.01
	IP	6,444.93	6,448.86
	% deviation from UB	*0.0375%*	*0.0023%*
	CPU time (seconds)	*4.36*	*4.49*
10	**UB**	6,447.35	6,449.01
	IP	6,444.93	6,448.86
	% deviation from UB	*0.0375%*	*0.0023%*
	CPU time (seconds)	*5.94*	*5.83*
20	**UB**	6,447.35	6,449.01
	IP	6,444.93	6,448.86
	% deviation from UB	*0.0375%*	*0.0023%*
	CPU time (seconds)	*8.22*	*8.15*
30	**UB**	6,447.35	6,449.01
	IP	6,444.93	6,448.86
	% deviation from UB	*0.0375%*	*0.0023%*
	CPU time (seconds)	*10.65*	*10.43*

Table 13. Maximum number of iterations testing results for 10 product problems

Number of Iterations		**30 products**	
		1st set	***2nd set***
5	**UB**	22,859.80	22,207.50
	IP	22,818.70	22,192.00
	% deviation from UB	*0.1798%*	*0.0698%*
	CPU time (seconds)	*9.98*	*12.34*
10	**UB**	22,859.80	22,207.50
	IP	22,818.70	22,192.00
	% deviation from UB	*0.1798%*	*0.0698%*
	CPU time (seconds)	*12.28*	*15.14*
20	**UB**	22,859.80	22,207.50
	IP	22,818.70	22,192.00
	% deviation from UB	*0.1798%*	*0.0698%*
	CPU time (seconds)	*16.38*	*18.54*
30	**UB**	22,859.80	22,207.50
	IP	22,818.70	22,192.00
	% deviation from UB	*0.1798%*	*0.0698%*
	CPU time (seconds)	*20.9*	*22.34*

Table 14. Maximum number of iterations testing results for 30 product problems

In general, the results observed on the prior tables allowed to conclude that the *maximum number of iterations* should be set to a value of 10 iterations since in all the problems tested there was no need of more than 10 iterations. In fact, the test problems show that the solutions obtained with 5 iterations were similar to the ones with 10 iterations. Since, the experiments with 10 iterations were solved almost in times similar to the 5 iterations experiments; it was decided to select 10 iterations to assure a more robust behavior.

Tables 15, 16 and 17 show the results of the testing for the *% deviation from UB threshold*, for each problem size. (Note: for the realization of these experiments the number of iterations was fixed on a value of 10). The results – presented on the next pages - are:

% deviation from UB		**2 products**	
		1st set	***2nd set***
1%	**UB**	1,215.43	1,229.03
	IP	1,207.71	1,215.91
	% deviation from UB	*0.6352%*	*1.0675%*
	CPU time (seconds)	*2.2*	*30.54 (**)*
2%	**UB**	1,215.43	1,229.03
	IP	1,207.71	1,215.91
	% deviation from UB	*0.6352%*	*1.0675%*
	CPU time (seconds)	*2.21*	*2.05*
5%	**UB**	1,215.43	1,229.03
	IP	1,207.71	1,215.91
	% deviation from UB	*0.6352%*	*1.0675%*
	CPU time (seconds)	*2.22*	*2.05*
7.5%	**UB**	1,215.43	1,229.03
	IP	1,207.71	1,215.91
	% deviation from UB	*0.6352%*	*1.0675%*
	CPU time (seconds)	*2.2*	*2.05*

Table 15. % deviation from UB testing results for 2 product problems

% deviation from UB		**10 products**	
		1st set	***2nd set***
1%	**UB**	6,447.35	6,449.01
	IP	6,444.93	6,448.86
	% deviation from UB	*0.0375%*	*0.0023%*
	CPU time (seconds)	*5.91*	*5.83*
2%	**UB**	6,447.35	6,449.01
	IP	6,444.93	6,448.86
	% deviation from UB	*0.0375%*	*0.0023%*
	CPU time (seconds)	*5.92*	*5.82*
5%	**UB**	6,447.35	6,449.01
	IP	6,444.93	6,448.86
	% deviation from UB	*0.0375%*	*0.0023%*
	CPU time (seconds)	*5.9*	*5.84*
7.5%	**UB**	6,447.35	6,449.01
	IP	6,444.93	6,448.86
	% deviation from UB	*0.0375%*	*0.0023%*
	CPU time (seconds)	*5.94*	*5.82*

Table 16. % deviation from UB testing results for 10 product problem

% deviation from UB		**30 products**	
		1st set	***2nd set***
1%	**UB**	22,859.80	22,207.50
	IP	22,818.70	22,192.00
	% deviation from UB	*0.1798%*	*0.0698%*
	CPU time (seconds)	*12.27*	*15.14*
2%	**UB**	22,859.80	22,207.50
	IP	22,818.70	22,192.00
	% deviation from UB	*0.1798%*	*0.0698%*
	CPU time (seconds)	*12.28*	*15.13*
5%	**UB**	22,859.80	22,207.50
	IP	22,818.70	22,192.00
	% deviation from UB	*0.1798%*	*0.0698%*
	CPU time (seconds)	*12.28*	*15.14*
7.5%	**UB**	22,859.80	22,207.50
	IP	22,818.70	22,192.00
	% deviation from UB	*0.1798%*	*0.0698%*
	CPU time (seconds)	*12.29*	*15.14*

Table 17. % deviation from UB testing results for 30 product problems

Regarding the *% deviation from UB threshold*, it was found that the best value that should be used as an algorithm fixed parameter is a 2% error threshold due to the fact that mostly all the solutions never went beyond the 1% threshold and just one of the random instance sets, corresponding to the second set of the 2 product problem size, had a % deviation from UB that was greater than 1% (1.0675%)

4.2.2 Performance testing

Finally, some experiments were done to test the performance of the method on a set of randomly generated problems. Two standard deviation values were used: 0.05 and 0.2. For these experiments, we generated 10 random instance problems for each standard deviation value within each problem size (2, 10 and 30 products) and recorded the *% deviation from UB* as well as the *CPU times*. A total of 60 random instance problems were generated. Also, some model's parameters had fixed values: number of base periods (48) as well as number of iterations (10) and % deviation from UB threshold (2%).

Regarding the experiments with standard deviation equal to 0.05, some interesting results were obtained. Table 18 contains a summary of the results. Basically, it can be seen that the mean % deviations from the UB are not greater than 1% (the greatest mean % value is 0.855% corresponding to a 2 product system) which means that the solutions obtained are very close to the optimal values.

σ = 0.05	**Products**		
	2	***10***	***30***
maximum CPU time (seconds)	2.320	6.150	15.140
minimum CPU time (seconds)	2.100	5.770	11.690
Mean CPU time (seconds)	*2.233*	*5.938*	*13.106*
maximum % deviation from UB	1.1807	0.0897	0.4391
minimum % deviation from UB	0.4371	0.0011	0.0073
Mean % deviation from UB	*0.8554*	*0.0270*	*0.1240*

Table 18. Results with σ = 0.05

In the case of the experiments with standard deviation equal to 0.2, it is shown that the results are very similar to those found on the experiments with a standard deviation equal to 0.05. Table 19 shows the results. At first, it can be observed that the mean % deviations from the UB does not have very large values. In fact, the greatest value corresponds to a 1.0586% deviation from the UB from the 2 product system, and the lowest value corresponds to a 0.1151% deviation from the UB for the 30 product system.

σ = 0.2	**Products**		
	2	***10***	***30***
maximum CPU time (seconds)	2.540	6.980	14.670
Minimum CPU time (seconds)	2.070	5.850	11.330
Mean CPU time (seconds)	*2.304*	*6.071*	*12.913*
Maximum % deviation from UB	1.8883	0.5199	0.2010
Minimum % deviation from UB	0.0001	0.0036	0.0305
Mean % deviation from UB	*1.0586*	*0.2013*	*0.1151*

Table 19. Results with σ = 0.2

As a general conclusion, it can be said that since the % deviations from the UB of each random instance sets are not very large (all bellow 2%) it can be stated that the method has a solid overall performance. Also, it can be seen in Tables 18 and 19 that the CPU times obtained show that the method is very fast since it only took less than 15 seconds of CPU time to solve a problem with 30 products. These fast CPU times demonstrate that the method works very well on the tested range of problem sizes.

4.2.3 Robustness

After the prior experiments were completed, it was decided to make some experiments to test the robustness of the solutions found with the proposed method. What continued was to generate 2 problems for each problem size (2, 10 and 30 products) along with 8 sets of random instance errors for each. The sets were designed to recreate 4 possible test scenarios involving two levels of each of the following parameters: the maximum number of base periods (48 or 60) and the standard deviation of the error instances (0.05 or 0.2). The 4 scenarios were:

1. Two random instance sets with 48 base periods each and a σ of the error instances equal to 5%
2. Two random instance sets with 48 base periods each and a standard deviation of the error instances equal to 20%
3. Two random instance sets with 60 base periods each with a σ of the error instances equal to 5%

4. Two random instance sets with 60 base periods each and a standard deviation of the error instances equal to 20%.

Then, solutions for each set (OV and UB) were obtained, and they were kept as comparison data. Next, 4 random problem instances (called "test problems") were generated, one for each of the test scenarios explained above, in order to obtain the solution values of the design variables G_i and k_i for each of the products. Consequently, in order to test robustness, the solution values of each "test problem" were put on their corresponding scenario sets, and using the same original random errors it was then proceeded to record the new solution given for those specific values of the variables. The objective of the insertion of these values was to see how the solution changed and what how much it deviated respect to the solutions found on the "test problem"

The following example illustrates the process. Let's take into account the case of 2 products and the first scenario: 48 base periods and a standard deviation of the error instances equal to 5%. The results for the "test problem" were: UB = 1211.66, OV = 1200.93, % deviation from UB = 0.8856%, CPU time = 4.44 seconds; and the values of their design variables were: $k_1 = 2$, $G_1 = 2.40$, $k_2 = 1$, $G_2 = 2.37$.

Then, two sets of random error instances were generated and their optimization solutions were obtained: $UB_1 = 1235.45$, $OV_1 = 1230.41$, % deviation from $UB_1 = 0.4079$; $UB_2 = 1222.70$, $OV_2 = 1210.99$, % deviation from $UB_2 = 0.9577$. Each of the two sets had their own design variable solutions but in order to test robustness, the values of the design variables (k_1, G_1, k_2, G_2) were tested on the two sets to see how the OV would change.

The results obtained were the following: new Objective Value for the 1st set = 1224.17, and new Objective Value for the 2nd set = 1200.68

Now, comparing the OV of the "test problem" with the new Objective Values of the two random error sets, it can be seen that on the case of the first random error instance set, the new Objective Value solution deviated 0.51% from the original OV; meanwhile, on the second random error instance set, the new Objective Value solution deviated 0.85% from the original solution.

Now, taking a more general look into the general results, in the case of the 2 product system it was found that the instance sets with the inserted solutions produced results very close to the solution found on the proof set. The biggest mean deviation from the original solution (0.91%) was found on the case were 60 base periods were used and the standard deviation of the errors was 0.2. Following, in Table 20, the complete set of results can be seen:

		Number of base periods	
	2 products	**48**	**60**
σ	**0.05**	0.68%	0.55%
	0.2	0.22%	0.91%

Table 20. Mean deviations from original solution (2 products)

Regarding the 10 product system, the results showed some improvement respect to the results on the 2 product system. The maximum deviation from the original solution was found to be a value of 0.33% (corresponding to the sets with 48 base periods and with error's standard deviation of 0.2). The complete results can be seen on Table 21, which follows:

		Number of base periods	
	10 products	**48**	**60**
σ	**0.05**	0.11%	0.03%
	0.2	0.33%	0.25%

Table 21. Mean deviations from original solution (10 products)

Finally, regarding the 30 product system, the results varied more than the 2 or 10 product instances. In this case, the biggest mean deviation was a 1.81% value and corresponded to the sets with 60 base periods and with the error's standard deviation equal to 0.2. The complete set of results can be seen next on Table 22, as follows:

		Number of base periods	
	30 products	**48**	**60**
σ	**0.05**	0.82%	1.74%
	0.2	0.14%	1.81%

Table 22. Mean deviations from original solution (30 products)

As a conclusion, it can be seen that the solutions obtained with the proposed method show robustness since the % deviations from the original solutions are not very large, and particularly aren't greater than 1% (in the case of the systems with 2 and 10 products) and 2% (in the case of the 30 product system). These results suggest that the column generation technique mixed with integer programming can produce robust solutions over a range of problem sizes and levels of variability.

General Note: All the problem instances for the PELSP and the SPLSP were solved on a Pentium IV 133Mhz processor. The programming was done in C++ and the mixed integer programming solver **lp_solve** **5.5.0.5** [7] was utilized to solve the master problem.

Chapter 5. Conclusions

5.1 Conclusions

The present thesis addresses two versions of the lot scheduling model with price optimization. Their solutions methods were based on the Column Generation technique mixed with Integer Programming and it was shown that the proposed methodology produces solutions very close to the optimal with fast computation times.

It is also important to notice that computational experiments indicate that the deterministic lot scheduling problem with price optimization is relatively easy to solve in contrast to the standard economic lot scheduling problem. Also, the experiments conducted showed the effect of the parameters of the procedure on solution quality and computation time. Managerial insights were also provided regarding deciding for which products to invest in setup time and setup cost reduction. In general, the effect of setup cost reduction for individual product is hard to predict intuitively and it is recommended finding a (near) optimal solution using our proposed method

Regarding the stochastic lot scheduling problem with price optimization it can also be concluded that the proposed methodology approach has been found to be very suitable and helped achieve robust results. The solution method was tested using a range of factorial experiments which produced very close to optimal solutions approximations.

5.2 Recommendations for further research

There are some future lines of study regarding lot scheduling problems with price optimization and it is important to note that the present thesis can be extended in several directions that complement and make the formulation suitable for specific situations.

One direct line of study relates to the modeling of the lot scheduling problem with price optimization including a lost sales policy, which is an alternative to the policies proposed on the current thesis. It is also interesting to extend the modeling of the same deterministic and stochastic models to consider lead times as part of the formulation. Also, in the specific case of the stochastic lot scheduling model, an important extension would be towards the inclusion of sequence dependent setups.

Finally, one very interesting approach would be to extend the current work into game-theoretic formulations that consider the effect of competition on demand and pricing. This type of modeling could help address a more complete view of scheduling on real environments where competition can lead to important policy changes.

References

[1] P.L. Abad, Determining optimal selling price and lot size when the supplier offers all-unit quantity discounts, Decision Sciences 19 (3) (1988) 622-634.

[2] P.L. Abad, Joint price and lot-size determination when supplier offers incremental quantity discounts, Journal of the Operational Research Society 39 (6) (1988) 603-607.

[3] T. Altiok, G.A. Shiue. Single-stage, multi-product production/inventory systems with backorders, IIE Transactions 26 (2) (1994) 52-61

[4] T. Altiok, G.A. Shiue. Single-stage, multi-product production/inventory systems with lost sales, Naval Research Logistics 42 (6) (1995) 889-913

[5] R. Anupindi, S. Tayur. Managing stochastic Multiproduct systems: model, measures, and analysis, Operations Research 46 (3S) (1998) 98-111

[6] A. Banerjee, Concurrent pricing and lot sizing for make-to-order contract production, International Journal of Production Economics 93-94 (1) (2005) 189-195.

[7] M. Berkelaar, Introduction to lp_solve 5.5.0.5, http://lpsolve.sourceforge.net/5.5/, Accessed on November 2, 2005.

[8] E.E. Bomberger, A Dynamic Programming Approach to a Lot Size Scheduling Problem, Management Science 12 (11) (1966) 778-784.

[9] K. Bourland, C. Yano. The strategic use of capacity Slack in the economic lot scheduling problem with random demand, Management Science 40 (12) (1994) 1690-1704

[10] K. Bretthauer, B. Shetty, S. Syam, S. White, A model for resource constrained production and inventory management, Decision Sciences 25 (4) (1994) 561-580.

[11] P. Chang, M. Yao, S. Huang, C. Chen, A genetic algorithm for solving a fuzzy Economic Lot-Size Scheduling problem, International Journal of Production Economics in press (2005).

[12] X. Chen, D. Simchi-Levi. Coordinating Inventory Control and Pricing strategies with random demand and fixed ordering cost: the infinite horizon case, Mathematics of Operations Reseach 29 (3) (2004) 698-723

[13] X. Chen, D. Simchi-Levi. Coordinating Inventory Control and Pricing strategies with random demand and fixed ordering cost: the finite horizon case, Operations Reseach 52 (6) (2004) 887-896

[14] D.L. Cooke, T.R. Rohleder, E.A. Silver, Finding effective schedules for the economic lot scheduling problem: a simple mixed integer programming approach, International Journal of Production Research 42 (1) (2004) 21-36.

[15] G. Dobson, The economic lot-scheduling problem: achieving feasibility using time-varying lot sizes, Operations Research 35 (5) (1987) 764-771.

[16] L.C. Doll, C.D. Whybark, An iterative procedure for the single-machine multi-product lot scheduling problem, Management Science 20 (1) (1973) 50-55.

[17] S.E. Elmaghraby, The economic lot scheduling problem (ELSP): review and extensions, Management Science 24 (6) (1978) 587-598.

[18] W. Elmaghraby, P. Keskinocak, Dynamic pricing in the presence of inventory considerations: research overview, current practices, and future directions, Management Science 49 (10) (2003) 1287-1309.

[19] N. Erkip, R. Gullu, A. Kocabiyikoglu. A quasi-birth-and-death model to evaluate fixed cycle time policies for stochastic multi-item production/inventory problem, Proceedings of MSOM Conference, Ann Harbor, Michigan. (2000)

[20] A. Federgruen, Z. Katalan. The ELSP: Cyclical base stock policies with idle times, Management Science 42 (6) (1996) 783-796

[21] J.C. Fransoo, V. Sridharan, J.W.M. Bertrand. A hierarchical approach for capacity coordination in multiple products single-machine production systems with stationary stochastic demands, European Journal of Operational Research 86 (1) (1995) 57-72

[22] G. Gallego. Scheduling the production of several items with random demand in one facility, Management Science 36 (12) (1990) 1579-1592.

[23] G. Gallego, G. van Ryzin. Optimal Dynamic Pricing of inventories with stochastic demand over finite horizons, Management Science 40 (8) (1994) 999-1020

[24] G. Gallego, D.X. Shaw, Complexity of the ELSP with general cyclic schedules, IIE Transactions 29 (2) (1997) 109-113.

[25] G. Gallego, G. van Ryzin. A Multiproduct dynamic pricing problem and its applications to network yield management, Operations Research 45 (1) (1997) 24-41

[26] S.K. Goyal. Lot size scheduling on a single machine for stochastic demand, Management Science 19 (11) (1973) 1322-1325

[27] S.C. Graves, On the deterministic demand multi-product single-machine lot scheduling problem, Management Science 25 (3) (1979) 276-280.

[28] S.C. Graves. The multi-product production cycling problem, AIIE Transactions 12 (3) (1980) 233-240

[29] R.W. Haessler, An improved extended basic period procedure for solving the economic lot scheduling problem, AIIE Transactions 11 (4) (1979) 336-340.

[30] R.W. Haessler, S.L. Hoghe, A note on the single-machine multi-product lot scheduling problem, Management Science 22 (8) (1976) 909-912.

[31] K.K. Haugen, A. Obstad, B.I. Pettersen, The profit maximizing capacitated lot-size (PCLSP) problem, European Journal of Operational Research in press (2006).

[32] F.S. Hillier, G.J. Lieberman, Introduction to operations research, 7th ed., Mc Graw Hill, New York, 2001.

[33] D. Kim, W.J. Lee, Optimal joint pricing and lot sizing with fixed and variable capacity, European Journal of Operational Research 109 (1) (1998) 212-227.

[34] P. Kotler, Marketing decision making: a model building approach, Holt, Rinehart and Winston, New York, 1971.

[35] H. Kunreuther, J.F. Richard, Optimal pricing and inventory decisions for non-seasonal items, Econometrica 39 (1) (1971) 173-175.

[36] S. Ladany, Optimal market segmentation of hotel rooms-the non-linear case, Omega 24 (1) (1996) 29-36.

[37] S. Ladany, A. Sternlieb, The interaction of economic ordering quantities and marketing policies, AIIE Trans 6 (1) (1974) 35-40.

[38] R.C. Leachman, A. Gascon. A heuristic policy for multi-item, single-machine production systems with time-varying stochastic demands, Management Science 34 (3) (1988) 377-390

[39] W.J. Lee, Determining order quantity and selling price by geometric programming: optimal solution, bounds and sensitivity, Decision Sciences 24 (1) (1993) 76-87.

[40] G.C. Lin, D.E. Kroll, C.J. Lin, Determining a common production cycle time for an economic lot scheduling problem with deteriorating items, European Journal of Operational Research in press (2005).

[41] J.G. Madigan, Scheduling a multi-product single machine system for an infinite planning period, Management Science 14 (11) (1968) 713-719.

[42] A.S. Manne, Programming of economic lot sizes, Management Science 4 (2) (1958) 115-135.

[43] D. Markovitz, M. Reiman , L. Wein. The Stochastic Economic Lot Scheduling Problem: Heavy Traffic Analysis of dynamic cyclic policies, Operations Research 48 (1) (2000) 136-154

[44] J.P. Monahan, A quantity discount pricing model to increase vendor profits, Management Science 30 (6) (1984) 720-726.

[45] J. Qiu, R. Loulou. Multiproduct production/inventory control under random demands, IEEE Transactions, Automatic Control. 40 (2) (1995) 350-356

[46] J. Rogers, A computational approach to the economic lot scheduling problem, Management Science 4 (3) (1958) 264-291.

[47] M.J. Rosenblatt, H.L. Lee, Improving profitability with quantity discounts under fixed demand, IIE Transactions 17 (4) (1985) 388-395.

[48] R. Roundy, Rounding off to powers of two in continuous relaxations of capacited lot sizing problems, Management Science 35 (12) (1989) 1433-1442.

[49] S. Sadjadi, M. Oroujee, M.B. Aryanezhad, Optimal production and marketing planning, Computer Optimization and Applications 30 (1) (2005) 1-9.

[50] Salvietti, L. and N. R. Smith. A profit-maximizing economic lot scheduling problem with price optimization. To appear in the European Journal of Operational Research.

[51] S.A. Smith, D. Achabal, Clearance pricing and inventory policies for retail chains, Management Science 44 (3) (1998) 285-300.

[52] C. Sox, J. Muckstadt. Optimization based planning for the Stochastic Lot Scheduling Problem, IIE Transactions 29 (5) (1997). 349-357

[53] C. Sox, P. Jackson, A. Bowman, J. Muckstadt . A review of the stochastic lot scheduling problem, International Journal of Production Economics 62 (3) (1999) 181-200

[54] W.W. Trigeiro, L.J. Thomas, J.O. McClain, Capacitated lot sizing with setup times, Management Science 35 (3) (1989) 353-366.

[55] T.L. Urban, Deterministic inventory models incorporating marketing decisions, Computers and Industrial Engineering 22 (1) (1992) 85-93.

[56] T.S. Vaughan. The effect of correlated demand on the cyclical scheduling system, International Journal of Production Research 41 (9) (2003) 2091-2106

[57] B. Wagner, D.J. Davis, A search heuristic for the sequence-dependent economic lot scheduling problem, European Journal of Operational Research 141 (1) (2002) 133- 146.

[58] T.M. Whitin, Inventory control and price theory, Management Science 2 (1) (1955) 61-68.

[59] E. Winands, I. Adan, G. van Houtun. The Stochastic Economic Lot Scheduling Problem: a survey, Technical University Eidenhoven. (2005)

[60] S. Yakowitz, F. Szidarovszky, An introduction to numerical computations, Macmillan Publishing Company, New York, 1986.

[61] M.J. Yao, S.E. Elmaghraby, On the economic lot scheduling problem under power-of-two policy, Computers and Mathematics with Applications 41 (2001) 1379-1393.

[62] M.J. Yao, S.E. Elmaghraby, I. Chen, On the feasibility testing of the economic lot scheduling problem using the extended basic period approach, Journal of the Chinese Institute of Industrial Engineers 20 (5) (2003) 435-448.

[63] M.J. Yao, J. Huang, Solving the economic lot scheduling problem with deteriorating items using genetic algorithms, Journal of Food Engineering 70 (3) (2005) 309-322

[64] P.H. Zipkin. Models for design and control of stochastic multi-item batch production systems, Operations Research. 34 (1) (1986) 91-104

Appendix 1. C ++ program for the PELSP (article 1)

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include <time.h>

#include "lp_lib.h"


// declaracion de subrutina a utilizar en MAIN //
double solve(double, double, double, double, double, double, double, double, double,
double, double*, double*);



// ********************** Programa Principal *********************** //


int main()
{


// --- DECLARACION DE VARIABLES --- //

bool condition;

int productos;
int i,xint,conteo,maxcount;

//arrays//
double *aS, *aC, *aM, *aA, *aR, *aT, *ah, *ahday;
double *aMIN, *aMAX, *aAmin, *aAmax,*CoefLambda,*aNuevoCoef,*aNuevoCoefA;
double *row0, *row1, *row2,*row3,*duals;
double x,diferencia,Period,B,Capacidad,tolerancia,porcdif,obj,objentero,UB,bestUB;

double d,k; //variables optimizadas de subrutina solve
double dmin,dmax,dmin2; //variables de demanda , para coeficientes minimo y maximo

double *columna1, *Comparacion;
double value,suma,E,Suma,SumaT;
double duration,cputime;

time_t start, finish;

// --- FIN declaracion de variables --- //


// comienza el reloj de C
start = clock();
//time (&start);


// -- INICIO -- Valores iniciales de algunos parametros //
SumaT=0;
maxcount=20; // conteo maximo del ciclo while
tolerancia = 0.05; // tolerancia porcentual para error bound
porcdif=1000000; // valor M para comenzar ciclo while externo
value=1; // valor para la comparacion que ayuda a la generacion de columnas

// -- FIN -- //




/* Leer el archivo "datos.txt" para obtener los inputs requeridos */

ifstream ff("datos.txt");

ff >> x;
xint = int(x);
cout << " **************************************** " << endl;
cout << " *          ELSP with Prices             * " << endl;
cout << " **************************************** " << endl;
cout << " " << endl;
//cout << "El Sistema es de " << xint << " productos " << endl;


// Definicion de algunos arreglos para almacenar valores //
aA = (double *)calloc( xint ,sizeof( double ) );
aR = (double *)calloc( xint ,sizeof( double ) );
aS = (double *)calloc( xint ,sizeof( double ) );
aC = (double *)calloc( xint ,sizeof( double ) );
aM = (double *)calloc( xint ,sizeof( double ) );
aT = (double *)calloc( xint ,sizeof( double ) );
ah = (double *)calloc( xint ,sizeof( double ) );
```

```
ahday = (double *)calloc( xint ,sizeof( double ) );

aMIN = (double *)calloc( xint ,sizeof( double ) );
aMAX = (double *)calloc( xint ,sizeof( double ) );
aAmin = (double *)calloc( xint ,sizeof( double ) );
aAmax = (double *)calloc( xint ,sizeof( double ) );
aNuevoCoef = (double *)calloc( xint ,sizeof( double ) );
aNuevoCoefA = (double *)calloc( xint ,sizeof( double ) );

CoefLambda = (double *)calloc( xint ,sizeof( double ) );

duals = (double *)calloc( 2 + xint, sizeof( double ) );
row0 = (double *)calloc( 1 + xint*2, sizeof( double ) );
row1 = (double *)calloc( 1 + xint*2, sizeof( double ) );
row2 = (double *)calloc( 1 + xint*2, sizeof( double ) );
row3 = (double *)calloc( 1 + xint*2, sizeof( double ) );
columna1 = (double *)calloc( xint*2+1, sizeof( double ) );


ff >> x;
Period = int(x);
//cout << " " << endl;
//cout << " El tiempo anual disponible para la produccion es: " << Period << " dias " <<
endl;

ff >> x;
B = x;
//cout << " " << endl;
//cout << " El Periodo basico es de: " << B << " dias " << endl;
//cout << " " << endl;
//cout << " " << endl;

ff >> x;
Capacidad = x;
//cout << " " << endl;
//cout << " La Capacidad es de: " << Capacidad << " dias " << endl;
//cout << " " << endl;


//cout << " " << endl;


  for (productos=0; productos<xint; productos++)
  {

            /* Leer y guardar los valores */
```

```
        ff >> x;  aA[productos] = x;
        //cout << " " << endl;
        //cout << " La constante multiplicadora A para demanda exponencial: " <<
aA[productos] << "\n" << endl;

        ff >> x;  aR[productos] = x;
        //cout << " " << endl;
        //cout << " Parametro R para la formula de demanda exponencial: " <<
aR[productos] << "\n" << endl;

        ff >> x;  aS[productos] = x;
        //cout << " " << endl;
        //cout << " Costo de Setup ($), S es: " << aS[productos] << "\n" << endl;

        ff >> x;  aC[productos] = x;
        //cout << " " << endl;
        //cout << " Costo Unitario ($/unidad), C es: " << aC[productos] << "\n" << endl;

        ff >> x;  aM[productos] = x;
        //cout << " " << endl;
        //cout << " Tasa Diaria de Produccion (unidades/dia), m es: " << aM[productos]
<< "\n" << endl;

        ff >> x;  aT[productos] = x;
        //
        cout << " " << endl;
        cout << " Tiempo de Setup (dias), T es: " << aT[productos] << "\n" << endl;

        ff >> x;  ah[productos] = x;
        //cout << " " << endl;
        //cout << " Costo Anual de Holding , h es: " << ah[productos] << "\n" << endl;
        //cout << " " << endl;

        // Calculo de Holding Cost por dia //
        ahday[productos] = ah[productos]/Period;
        //cout << " El Holding Cost por dia ($/unidad*dia), es: " << ahday[productos]
<< "\n" << endl;


        E=0; // valor inicial para la variable de Lambda, es cero porque la 1era rutina de
solve no la utiliza



   // ******************************************* //
```

```
            // -- llamada a subrutina "solve" que me da los valores optimos de K y d para
cada producto -- //
            solve(E, B,
aS[productos],aC[productos],aM[productos],aA[productos],aR[productos],0,ah[productos
],ahday[productos],&d,&k);

            //cout << " " << endl;
            //cout << " **************************************** " << endl;
            //cout << " *    Valores exportados de SUBRUTINA    * " << endl;
            cout << " **************************************** " << endl;
            printf("D:\t%.20f",d); // Imprime el valor de la demanda con 20 decimales para
mas precision
            cout << " " << endl;
            cout << "K: " << k << endl;
            cout << " " << endl;

     // -- Calculo de demandas para los dos planes iniciales -- //
            //cout << " " << endl;
            dmin=(1)*(d); // La demanda para el plan minimo es un 25% de la demanda
optima
            dmax=(1.75)*(d); // La demanda para el plan maximo es un 75% mas grande que
la demanda optima
            //cout << " Demanda bajo de capacidad: " << dmin << endl;
            //cout << " Demanda por encima de capacidad: " << dmax << endl;
            //cout << " " << endl;
            //cout << " " << endl;
            //cout << " ---------------------------------------------------------- " << endl;



            // -*- Calcular Coeficientes para los planes iniciales del LP -*- //

            // -- Coeficiente Minimo -- //
            aMIN[productos]=(aA[productos]*log(aR[productos]/dmin)*dmin)-
(aS[productos]/(k*B))-((aC[productos]*ahday[productos]*(dmin)*(aM[productos]-
dmin)*k*B)/(2*aM[productos]))-(aC[productos]*dmin);
            //cout << " El Coeficiente Minimo es: " << aMIN[productos] << "\n" << endl;

            // -- Coeficiente Maximo -- //
            aMAX[productos]=(aA[productos]*log(aR[productos]/dmax)*dmax)-
(aS[productos]/(k*B))-((aC[productos]*ahday[productos]*(dmax)*(aM[productos]-
dmax)*k*B)/(2*aM[productos]))-(aC[productos]*dmax);
            //cout << " El Coeficiente Maximo es: " << aMAX[productos] << "\n" << endl;

            // -- Coeficiente de Restriccion de Tiempo Minimo -- //
            aAmin[productos]= aT[productos] + ((dmin*k*B)/aM[productos]);
            //cout << " Aprima es> " << ((dmin*k*B)/aM[productos]) << endl;
            //cout << " El Coeficiente de tiempo Minimo es: " << aAmin[productos] << "\n"
<< endl;

            // -- Coeficiente de Restriccion de Tiempo Maximo -- //
            aAmax[productos]=((dmax*k*B)/aM[productos]);
     //cout << " El Coeficiente de tiempo Maximo es: " << aAmax[productos] << "\n" <<
endl;


            SumaT=aT[productos]+SumaT;

            cout << "SumaT> " << SumaT << endl;



  } // end         FOR



// --- *** --- Procedimiento para Obtener la D minima "factible" para comenzar el LP ---
*** --- //


                    Suma=0;
                    for (productos=0;productos<xint;productos++){
                              Suma = Suma + aAmin[productos];
                              } //end for
                    //cout << " " << endl;
                    //cout << " --- *** Suma Minima de Capacidades para los productos: " <<
Suma << endl;
                    //cout << " " << endl;


          while ( Suma > Capacidad){

                    dmin2=dmin/10;
                    cout << " Demanda baja modificada: " << dmin2 << endl;

                    Suma=0;

                    for (productos=0;productos<xint;productos++){

                    // -- Coeficiente Minimo -- //
```

```
                        aMIN[productos]=(aA[productos]*log(aR[productos]/dmin2)*dmin2)-
(aS[productos]/(k*B))-((aC[productos]*ahday[productos]*(dmin2)*(aM[productos]-
dmin2)*k*B)/(2*aM[productos]))-(aC[productos]*dmin2);

                        // -- Coeficiente de Restriccion de Tiempo Minimo -- //
                        aAmin[productos]= aT[productos] + ((dmin2*k*B)/aM[productos]) ;
                        //cout << " setup2 : "<< aT[productos] << endl;
                //cout << " A2 es: "<< ((dmin2*k*B)/aM[productos]) << endl;
        //cout << " El Coeficiente de tiempo Minimo es: " << aAmin[productos] << "\n" <<
endl;


                        Suma = Suma + aAmin[productos];

                        } //end for

                        //cout << " " << endl;
                        //cout << " --- *** Suma Minima de Capacidades para los productos: " <<
Suma << endl;
                        //cout << " " << endl;

                        dmin=dmin2;

            }; //endwhile suma vs. capacidad




// -----
************************************************************************
** ----- //

  // lp solve  comienza ... //

  lprec *lp;
  HINSTANCE lpsolve;

  lpsolve = LoadLibrary("lpsolve55.dll");

  if (lpsolve == NULL) {
    printf("Unable to load lpsolve shared library\n");
    return(FALSE);
  }
```

```
 //
**********************************************************************
** //
 // Inicia el desarrollo de la estructura del programa //


 lp=make_lp(0,xint*2); /* prods*2 variables, 0 restricciones */

 // Descripcion del sistema utilizado //
 cout << "
*********************************************************************
" << endl;
 cout << " *                          LP SOLVE                          * " << endl;
 cout << "
*********************************************************************
" << endl;
 cout << " * Description: Open Source (Mixed-Integer) Linear Programming System  * "
<< endl;
 cout << " * Language: Multi-Platform, pure ANSI C / POSIX Source Code         * " <<
endl;
 cout << " * Official Name: lp_solve                                          * " << endl;
 cout << " * Release Data: Version 5.5.0.5. , dated: October 2005             * " << endl;
 cout << " * Co-developers: Michael Berkelaar, Kjell Eikland, Peter Notebaert   * " <<
endl;
 cout << " * Licence Terms: GNU LGPL (Lesser General Public Licence)          * " <<
endl;
 cout << "
*********************************************************************
" << endl;
 cout << " " << endl;
 //cout << " " << endl;

 set_maxim(lp); // Define el problema como MAXIMIZAR

 set_lp_name(lp, "ELSPwP 1.0"); /* Pone el nombre al programa */


 for (productos=0; productos<xint; productos++) // FOR 1
 {

 // Estructuracion de la funcion objetivo //
 row0[productos*2+1] = aMIN[productos];
 row0[productos*2+2] = aMAX[productos];
 set_obj_fn(lp, row0);

 // Estructuracion de las restricciones del problema //
 set_add_rowmode(lp, TRUE);
 row1[productos*2+1] = aAmin[productos] - aT[productos];
 row1[productos*2+2] = aAmax[productos];

 for (i=0; i<xint*2+1; i++) row2[i]=0;
 set_add_rowmode(lp, TRUE);
 row2[productos*2+1] = 1.0;
 row2[productos*2+2] = 1.0;

 set_add_rowmode(lp, FALSE); // concluye la adicion de nuevas restricciones

 add_constraint(lp, row2, EQ , 1.0);

 }  // end FOR 1


 add_constraint(lp, row1, LE, Capacidad - SumaT); // agrega la restriccion de Capacidad



 // Imprime la estructura del problema //
 //cout << " " << endl;
 print_lp(lp);
 //cout << " " << endl;
 //cout << " " << endl;
 //cout << " *** Solucion *** " << endl;
 //cout << " " << endl;


 // Resuelve el problema y lo presenta en pantalla //
 solve(lp);
 //cout << " " << endl;
 //cout << " " << endl;

 obj=get_objective(lp);


 // Obtiene los valores duales de la solucion //
 get_dual_solution(lp,duals);
 //cout << " " << endl;
 //cout << " *** duales *** " << endl;
 for (i=1; i<xint+2; i++) {
 //cout << " Precio Sombra: " << duals[i] << endl;
 }

 // Imprime la solucion del problema . 1 = numero de columnas para poner valores //
```

```
  print_solution(lp,1);
  //cout << " " << endl;
  //cout << " " << endl;

  // Imprime en pantalla los valores duales del problema //
  print_duals(lp);
  //cout << " " << endl;
  //cout << " " << endl;
  //cout << " *** FIN *** " << endl;
  //cout << " " << endl;
  //cout << " " << endl;


  // condiciones para ciclo de comparacion //
  condition = FALSE;
  Comparacion = (double *)calloc( xint, sizeof( double ) );



  ////////////////////////////////////////////////////////////////////////////
  // *** --- Fase 2 > Subproblemas utilizando precios sombra --- *** //
  ////////////////////////////////////////////////////////////////////////////

  for (productos=0; productos<xint; productos++) // FOR 2
  {

  E=duals[xint+1]; // guarda el valor de LAMBDA para la subrutina, bajo el nombre de E;
  //cout << " El valor de lambda (variable E) es:  " << E << endl;

  // llamada a subrutina para obtener los valores de D y K  y volver a calcular coeficientes
//

  // Nuevo Coeficiente con Lambda
  CoefLambda[productos]=solve(E, B,
aS[productos],aC[productos],aM[productos],aA[productos],aR[productos],0,ah[productos
],ahday[productos],&d,&k);
  //cout << " " << endl;
  //cout << " El Coeficiente Modificado con Valores Sombra es: " <<
CoefLambda[productos] << "\n" << endl;
  //cout << " " << endl;

  //printf(" D es:\t%.20f",d);
  //cout << " " << endl;
  //cout << " K es: " << k << endl;
  //cout << " " << endl;
  // Nuevo Coeficiente para plan entrante //
  aNuevoCoef[productos] = (aA[productos]*log(aR[productos]/d)*d)-
((aS[productos])/(k*B))-((aC[productos]*ahday[productos]*(d)*(aM[productos]-
d)*k*B)/(2*aM[productos]))-(aC[productos]*d);
  //cout << " *** El nuevo Coeficiente entrante es: " << aNuevoCoef[productos] << endl;
  //cout << " " << endl;


   if ( ( aNuevoCoef[productos] < 0 ))
            CoefLambda[productos]=duals[productos+1];

  //cout << " " << endl;
  diferencia = CoefLambda[productos] - duals[productos+1];
  //cout << " La diferencia es: " << diferencia << endl;
  //cout << " " << endl;
  //cout << " " << endl;
  Comparacion[productos] = diferencia;



  // Inicia condicion , una columna nueva entra si la diferencia es mayor a cero //
  condition = (Comparacion[productos] > value) || condition;

  if ((Comparacion[productos] > value) ) {

  // Nuevo Coeficiente para plan entrante //
  aNuevoCoef[productos] = (aA[productos]*log(aR[productos]/d)*d)-
((aS[productos])/(k*B))-((aC[productos]*ahday[productos]*(d)*(aM[productos]-
d)*k*B)/(2*aM[productos]))-(aC[productos]*d);
  //cout << " El nuevo Coeficiente entrante es: " << aNuevoCoef[productos] << endl;
  //cout << " " << endl;

  // Nuevo Coeficiente de tiempo para plan entrante //
  aNuevoCoefA[productos] = ((d*k*B)/aM[productos]);
  //cout << " La restriccion de tiempo para el nuevo plan es: " <<
aNuevoCoefA[productos] << endl;
  //cout << " " << endl;

  for (i=0; i<xint+2; i++) columna1[i]=0;

  if ((Comparacion[productos] > 0)) {
            columna1[productos+1]=1.0;
            columna1[0] = aNuevoCoef[productos];
      columna1[xint+1] = aNuevoCoefA[productos];
            add_column(lp, columna1);
```

```
 }
 //cout << " " << endl;
 //cout << " " << endl;

 } // end IF
 //print_lp(lp);

 } // end FOR 2


 // ciclo para obtener el Upper Bound de la funcion //
 suma=0;
 for (i=0;i<xint;i++){
         suma = suma + Comparacion[i];}
 UB = obj + suma ;
 bestUB = UB;

 //cout << " ************************ " << endl;
 cout << " *** *** **** UB : " << bestUB << endl;
 //cout << " ************************ " << endl;

 ////////////////////////////////////////////////////////////////////////////


// comienza el WHILE externo //
while ( porcdif > tolerancia ) {

 conteo=0;


 // comienza el WHILE interno //
 while  (condition && (conteo < maxcount)) {

                     conteo++;

             // Solve LP //

                     solve(lp);

                     //print_solution(lp,1);

                     obj=get_objective(lp);

                     // Get dual values //
                     get_dual_solution(lp,duals);
                     //cout << " " << endl;
```

```
                    //cout << " *** duales *** " << endl;
                    for (i=1; i<xint+2; i++) {
                    //cout << " Precio Sombra: " << duals[i] << endl;
                    }; // end FOR interno


                    for (productos=0; productos<xint; productos++)
                    {


                              E=duals[xint+1]; // guarda el valor de LAMBDA para la
subrutina, bajo el nombre de E;
                              //cout << "
****************************************** " << endl;
                              //cout << " El valor entrante de lambda (variable E) es:  "
<< E << endl;
                              //cout << "
****************************************** " << endl;
                              //cout << " " << endl;

                              // llamada a subrutina para obtener los valores de D y K  y
volver a calcular coeficientes //

                              // Nuevo Coeficiente con Lambda
                              //cout << "
***************************************** " << endl;
                              //cout << "
***************************************** " << endl;
                              //cout << " LLAMADA A SUBRUTINA " << endl;

                              CoefLambda[productos]=solve(E, B,
aS[productos],aC[productos],aM[productos],aA[productos],aR[productos],0,ah[productos
],ahday[productos],&d,&k);
                              //cout << " El Coeficiente Modificado con Valores Sombra
es: " << CoefLambda[productos] << "\n" << endl;
                              //cout << " " << endl;

                              printf(" D es:\t%.20f",d);
                              //cout << " " << endl;
                              cout << " K es: " << k << endl;

                              // Nuevo Coeficiente para plan entrante //
                              aNuevoCoef[productos] =
(aA[productos]*log(aR[productos]/d)*d)-((aS[productos])/(k*B))-
((aC[productos]*ahday[productos]*(d)*(aM[productos]-d)*k*B)/(2*aM[productos]))-
(aC[productos]*d);
```

```
                                        //cout << " *** Evaluacion de Coef Entrante: " <<
aNuevoCoef[productos] << endl;
                                        //cout << " " << endl;

                                        if ( ( aNuevoCoef[productos] < 0 ))
                                                  CoefLambda[productos]=duals[productos+1];


                                        //cout << " " << endl;
                                        diferencia = CoefLambda[productos] - duals[productos+1];
                                        //cout << " --------------------------------------- " << endl;
                                        //cout << " La diferencia es: " << diferencia << endl;
                                        //cout << " " << endl;
                                        //cout << " " << endl;
                                        Comparacion[productos] = diferencia;




                                        // Inicia condicion , una columna nueva entra si la
diferencia es mayor a cero //
                                        condition = (Comparacion[productos] > value) || condition;

                                        if ((Comparacion[productos] > value)) {

                                        // Nuevo Coeficiente para plan entrante //
                                        aNuevoCoef[productos] =
(aA[productos]*log(aR[productos]/d)*d)-((aS[productos])/(k*B))-
((aC[productos]*ahday[productos]*(d)*(aM[productos]-d)*k*B)/(2*aM[productos]))-
(aC[productos]*d);
                                        //cout << " El nuevo Coeficiente entrante es: " <<
aNuevoCoef[productos] << endl;
                                        //cout << " " << endl;

                                        // Nuevo Coeficiente de tiempo para plan entrante //
                                        aNuevoCoefA[productos] = ((d*k*B)/aM[productos]);
                                        //cout << " La restriccion de tiempo para el nuevo plan es: "
<< aNuevoCoefA[productos] << endl;
                                        //cout << " " << endl;

                                        for (i=0; i<xint+2; i++) columna1[i]=0;
                                        if ((Comparacion[productos] > 0)) {
                                        columna1[productos+1]=1.0;
                                        columna1[0] = aNuevoCoef[productos];
                                        columna1[xint+1] = aNuevoCoefA[productos];
                                        add_column(lp, columna1);
```

```
                                } //end if

                                //print_lp(lp);


                                } // end IF condicional


                        } // end FOR


        // ciclo para obtener el Upper Bound del LP //
        suma=0;
        for (i=0;i<xint;i++){
                        suma = suma + Comparacion[i];
                } //end for
        UB = obj + suma ;
                if ( UB < bestUB)
                bestUB = UB;

        cout << " *** *** **** UB : " << bestUB << endl;

        print_lp(lp);
        //cout << " " << endl;

        //cout << " " << endl;
        //cout << "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@ " << endl;
        cout << " @@@@ @@@@ CONTEO DE ITERACIONES: " << conteo << endl;
        //cout << "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@ " << endl;
        cout << " " << endl;


 };// end WHILE interno



 for (i=1;i<=get_Ncolumns(lp);i++)
 set_int(lp, i, TRUE);

 // resuelve el nuevo LP con integer variables //
 solve(lp);
```

```
  print_solution(lp,1);
  cout << " " << endl;
  cout << " " << endl;

  // Obtiene el valor de la funcion objetivo entera = UB //
  objentero=get_objective(lp);
  cout << " El objetivo LP: " << obj << endl;
  cout << " El objetivo entero: " << objentero << endl;

  // Calculo de Upper bound //
  cout << " El Upper Bound (UB) es: " << bestUB << endl;

  // Calculo de diferencia porcentual //
  porcdif = (bestUB-objentero)/(bestUB);
  cout << " " << endl;
  cout << " ****************************************************** " <<
endl ;
  cout << " *                                    * " << endl ;
  cout << " * --------------    RESULTADO FINAL    -------------- * " << endl;
  cout << " *                                    * " << endl;
  cout << " *  El porcentaje de error es:  " << porcdif*100 << endl;
  cout << " *                                    * " << endl ;
  cout << " ****************************************************** " <<
endl ;

  // Cambia las variables a no enteras para que vuelva a iterar sin ellas //
  for (i=1;i<=get_Ncolumns(lp);i++)
  set_int(lp, i, FALSE);

}; // end WHILE externo



// ** CPU TIME //
//time (&finish);
//duration = difftime (finish,start);

finish = clock();
duration = (finish-start);

cputime=duration/CLOCKS_PER_SEC;// se usa CLCOKS_PER_SEC porque cada
unidad del clock dura 1/1000 de segundo.
printf (" CPU time taken is:  %.2lf seconds.\n", cputime);

// **
```

```
delete_lp(lp);

FreeLibrary(lpsolve);

return (0);

}; // end MAIN



//
************************************************************************
*****//
// *   Aca comienza la SUBRUTINA , para resolver el programa de K y d optimos. *
//
************************************************************************
*****//

double solve (double E, double B,double aS ,double aC ,double aM ,double aA ,double
aR ,double aT ,double ah ,double ahday2 ,double *d ,double *k)
{

// declaracion de variables //
int count,j;
double LB,UB,EB,xl,xu,xm;
double F; // F = es la derivada parcial de U respecto a demanda, evaluado en lower bound
double G; // F = es la derivada parcial de U respecto a demanda, evaluado en medium
bound
double U,Uant; // Funcion de Utilidad

double Daa,Da,Dopt;

j=1;


//cout << " ----------------------------------------------------------- " << endl;
//cout << " " << endl;

//cout << " $$$ VALORES EXPORTADOS $$$ " << endl;
//cout << " " << endl;
//cout << " B: " << B << endl;

//printf(" S es:\t%.20f",aS);
//cout << " " << endl;
```

```
//printf(" C es:\t%.20f",aC);
//cout << " " << endl;

//cout << " M es: " << aM << "\n" << endl;
//cout << " A es: " << aA << "\n" << endl;
//cout << " R es: " << aR << "\n" << endl;
//cout << " T es: " << aT << "\n" << endl;

//printf(" H es:\t%.20f",ahday2);
//cout << " " << endl;
//cout << " " << endl;

//cout << " ------------------------------------------------------------ " << endl;

///cout << " " << endl;
///cout << " " << endl;

            count=1;
            U=-1000000000;
     Uant=-1000000000000000000;

            Daa=0;
            Da=0;
            Dopt=0;


            while( U > Uant){

                    // valores iniciales //
                    LB=0.0001; // el lower bound para la demanda es igual a una tasa de 1
unidad por dia
                    UB=aM-1; // el upper bound no puede exceder el tamano de la tasa de
produccion M
                    EB=0.0001; // error bound

                    xl=LB;
                    xu=UB;
                    F= (-B*ahday2*count*(aM-2*xl)*aC)/(2*aM) - aC + (aA*log(aR/xl)) -
aA - ((B*E*count)/aM);
                    //cout << " ********************************* " << endl;
                    //cout << " " << endl;
                    //cout << " " << endl;

                    xm=xu;
                    G= (-B*ahday2*count*(aM-2*xm)*aC)/(2*aM) - aC + (aA*log(aR/xm)) -
aA -((B*E*count)/aM);


                    if ( ( F > 0 ) && ( G > 0 ) ) {   //  if 1 , ( Escenario , todos positivos )

                              xl=UB;

                              Dopt=xl;

                              U=((aA*log(aR/Dopt)*Dopt)-(aS/(count*B))-
((aC*ahday2*(Dopt)*(aM-Dopt)*count*B)/(2*aM))-(aC*Dopt)-(aT*E)-
((Dopt*count*B*E)/aM));
                              *d=Dopt;

                              //cout << " " << endl;
                              //cout << " El valor de U es: " << U << endl;
                              //cout << " ********************************** " << endl;
                              //cout << " " << endl;
                              //cout << " ------------------------------------------------------------- "
<< endl;
                              //cout << " " << endl;

                              count++;

                              *k=count-1;
                              *d=Dopt;

                              //cout << " ****************************************** "
<< endl;
                              //cout << " COMPROBACION DE VALORES (D,K) DE
SUBRUTINA " << endl;
                              //cout << " k es: " << *k << endl;
                              //cout << " " << endl;
                              //cout << " d es: " << *d  << endl;
                              //cout << " " << endl;
                              //cout << " ****************************************** "
<< endl;
                              //cout << " " << endl;

                              return(U);

                    } // end if 1
```

```
        if ( ( F < 0 ) && ( G < 0 ) ) {   //  if 2 , ( Escenario , todos negativos )

                xu=LB;

                Dopt=xu;

                U=((aA*log(aR/Dopt)*Dopt)-(aS/(count*B))-
((aC*ahday2*(Dopt)*(aM-Dopt)*count*B)/(2*aM))-(aC*Dopt)-(aT*E)-
((Dopt*count*B*E)/aM));
                *d=Dopt;

                //cout << " " << endl;
                //cout << " El valor de U es: " << U << endl;
                //cout << " ********************************** " << endl;
                //cout << " " << endl;
                //cout << " ------------------------------------------------------------ "
<< endl;
                //cout << " " << endl;

                count++;

                *k=count-1;
                *d=Dopt;

                //cout << " ***************************************** "
<< endl;
                //cout << " COMPROBACION DE VALORES (D,K) DE
SUBRUTINA " << endl;
                //cout << " k es: " << *k << endl;
                //cout << " " << endl;
                //cout << " d es: " << *d  << endl;
                //cout << " " << endl;
                //cout << " ***************************************** "
<< endl;
                //cout << " " << endl;


                return(U);

        } // end if 2



        while ( (xu-xl) >= EB) {
```

```
                        G= (-B*ahday2*count*(aM-2*xm)*aC)/(2*aM) - aC +
(aA*log(aR/xm)) - aA -((B*E*count)/aM);
                        //cout << " Valor de G: " << G << endl;

                        if (F*G < 0)
                              xu=xm;
                        else
                              xl=xm;

                        xm=(xl+xu)/2;

            }; //end while interno



            Daa=Da;
            Da=Dopt;
            Dopt=xm;

            Uant=U;
            U=((aA*log(aR/Dopt)*Dopt)-(aS/(count*B))-((aC*ahday2*(Dopt)*(aM-
Dopt)*count*B)/(2*aM))-(aC*Dopt)-(aT*E)-((Dopt*count*B*E)/aM));
            *d=Dopt;


            //cout << " Uant " << Uant << endl;

            //cout << " " << endl;
            //cout << " El valor de U es: " << U << endl;
            //cout << " ********************************* " << endl;
            //cout << " " << endl;
            //cout << " ------------------------------------------------------------ " << endl;
            //cout << " " << endl;

            count++;

      }; // end while externo


*k=count-2;
*d=Da;

//cout << " ***************************************** " << endl;
cout << " COMPROBACION DE VALORES (D,K) DE SUBRUTINA " << endl;
cout << " k es: " << *k << endl;
//cout << " " << endl;
cout << " d es: " << *d  << endl;
```

```
//cout << " " << endl;



cout << " ***************************************** " << endl;
//cout << " " << endl;



return(Uant);

};// end subrutina solve
```

Appendix 2. C++ program for the SPLSP (article 2)

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include <time.h>

#include "lp_lib.h"


// INICIO - declaracion de subrutinas a utilizar en MAIN //

/// optimizador de subproblema
double solucion(int,int,int,double
*E,double,double,double,double,double,double,double,double,double,double,double,dou
ble,double,double*,double*,double*);

// evaluador de funcion de Utilidad
double evaluar (int,int,int,double
*E,double,double,double,double,double,double,double,double,double,double,double,dou
ble,double,double);

// - FIN



// ********************** Programa Principal *********************** //

int main()
{


// --- DECLARACION DE VARIABLES --- //

bool condition,agregadas;

int productos;
int i,b,xint,noise,conteo,maxcount,m,w;

//arrays//
double *aS, *aC, *aM, *aA, *aR, *aT, *ah, *ahday,*aV;
double *aMIN, *aMAX, *aAmin,
*aAmax,*CoefLambda,*aNuevoCoef,*aNuevoCoefA,*Neg,*UpperSuma;
double *row0, *row1, *row2,*row3,*duals,*aOPT,*aAopt;
double x,diferencia,Period,B,Capacidad,tolerancia,porcdif,objentero,UB,bestUB,obj;


double *E,*Et; // arreglo que guarda los ruidos del archivo ruidos.txt

double t,k,z,S,Tuno; //variables optimizadas de subrutina solve
double tmin,tmax,tmin2; //variables de demanda , para coeficientes minimo y maximo

double *columna1, *Comparacion;
double value,suma,F,Suma;

double Tp,K;

time_t start, finish;
double duration, cputime;

// --- FIN declaracion de variables --- //


//* comienza el reloj de C
start = clock();
//*

// -- INICIO -- Valores iniciales de algunos parametros //

maxcount=20; // conteo maximo del ciclo while
tolerancia = 0.02; // tolerancia porcentual para error bound
porcdif=100000000; // valor M para comenzar ciclo while externo
value=0.000001; // valor para la comparacion que ayuda a la generacion de columnas

z = 1.96; // correspondiente a un nivel de servicio del 95%

b=1;

// -- FIN -- //




/* Leer el archivo "datos.txt" para obtener los inputs requeridos */

ifstream ff("datos.txt");
```

```
ff >> x;
xint = int(x);
cout << " ***************************************** " << endl;
cout << " *        Stochastic ELSP v 3.1.4        * " << endl;
cout << " ***************************************** " << endl;
cout << " " << endl;
cout << "El Sistema es de " << xint << " productos " << endl;

ff >> x;
noise = int(x);
cout << "El numero de ruidos por producto es: " << noise << endl;


// Definicion de algunos arreglos para almacenar valores //
aA = (double *)calloc( xint ,sizeof( double ) );
aR = (double *)calloc( xint ,sizeof( double ) );
aS = (double *)calloc( xint ,sizeof( double ) );
aC = (double *)calloc( xint ,sizeof( double ) );
aM = (double *)calloc( xint ,sizeof( double ) );
aT = (double *)calloc( xint ,sizeof( double ) );
ah = (double *)calloc( xint ,sizeof( double ) );
ahday = (double *)calloc( xint ,sizeof( double ) );
aV = (double *)calloc( xint ,sizeof( double ) );
//TO = (double *)calloc( xint ,sizeof( double ) );

Et = (double *)calloc( xint*noise ,sizeof( double ) ); // arreglo que almacena ruidos
E = (double *)calloc( xint*noise ,sizeof( double ) ); // arreglo que almacena ruidos

aMIN = (double *)calloc( xint ,sizeof( double ) );
aMAX = (double *)calloc( xint ,sizeof( double ) );
aOPT = (double *)calloc( xint ,sizeof( double ) );
aAopt = (double *)calloc( xint ,sizeof( double ) );
Neg = (double *)calloc( xint ,sizeof( double ) );
aAmin = (double *)calloc( xint ,sizeof( double ) );
aAmax = (double *)calloc( xint ,sizeof( double ) );
aNuevoCoef = (double *)calloc( xint ,sizeof( double ) );
aNuevoCoefA = (double *)calloc( xint ,sizeof( double ) );

CoefLambda = (double *)calloc( xint ,sizeof( double ) );

duals = (double *)calloc( 2 + xint, sizeof( double ) );
row0 = (double *)calloc( 1 + xint*3, sizeof( double ) );
row1 = (double *)calloc( 1 + xint*3, sizeof( double ) );
row2 = (double *)calloc( 1 + xint*3, sizeof( double ) );
row3 = (double *)calloc( 1 + xint*3, sizeof( double ) );
columna1 = (double *)calloc( xint*3+1, sizeof( double ) );
```

```
UpperSuma = (double *)calloc( xint*3+1, sizeof( double ) );


ff >> x;
Period = int(x);
//cout << " " << endl;
//cout << " El tiempo anual disponible para la produccion es: " << Period << " dias " <<
endl;

ff >> x;
B = x;
//cout << " " << endl;
//cout << " El Periodo basico es de: " << B << " dias " << endl;
//cout << " " << endl;
//cout << " " << endl;

ff >> x;
Capacidad = x;
//cout << " " << endl;
//cout << " La Capacidad es de: " << Capacidad << " dias " << endl;
//cout << " " << endl;

// valores iniciales de optimizacion : t , k
Tp = 0.1;
//cout << " El tiempo de produccion es: " << Tp << " dias " << endl;
//cout << " " << endl;

K = 1;
//cout << " El periodo de produccion ,k, es: " << K << " dias " << endl;
//cout << " " << endl;
//


//* Leer el archivo "ruidos.txt" para obtener los ruidos requeridos */

ifstream yy("ruidos.txt"); // archivo de donde lee los ruidos obtenidos

for (m=0;m<(xint*noise);m++)
{
        yy >> x; Et[m] = x;
        //cout << " El ruido , Et { " << m << " } es: " << Et[m] << endl;
}
//* fin lectura de ruidos


                //cout << " ^^^^^^^^^^^^^^^^^^^^^ b es: " << b << endl;
```

```
  // ciclo para cada producto
  for (productos=0; productos<xint; productos++)
  {

          /* Leer y guardar los valores */

          ff >> x;  aA[productos] = x;
          //cout << " " << endl;
          //cout << " Paramatro A para la formula de demanda lineal: " << aA[productos]
<< "\n" << endl;

          ff >> x;  aR[productos] = x;
          //cout << " " << endl;
          //cout << " Parametro B para la formula de demanda lineal: " << aR[productos]
<< "\n" << endl;

          ff >> x;  aS[productos] = x;
          //cout << " " << endl;
          //cout << " Costo de Setup ($), S es: " << aS[productos] << "\n" << endl;

          ff >> x;  aC[productos] = x;
          //cout << " " << endl;
          //cout << " Costo Unitario ($/unidad), C es: " << aC[productos] << "\n" << endl;

          ff >> x;  aM[productos] = x;
          //cout << " " << endl;
          //cout << " Tasa Diaria de Produccion (unidades/dia), m es: " << aM[productos]
<< "\n" << endl;

          ff >> x;  aT[productos] = x;
          //cout << " " << endl;
          //cout << " Tiempo de Setup (dias), T es: " << aT[productos] << "\n" << endl;

          ff >> x;  ah[productos] = x;
          //cout << " " << endl;
          //cout << " Costo Anual de Holding , h es: " << ah[productos] << "\n" << endl;
          //cout << " " << endl;

          ff >> x;  aV[productos] = x;
          //cout << " " << endl;
          //cout << " El vendor Price V,  es: " << aV[productos] << "\n" << endl;
          //cout << " " << endl;

          // Calculo de Holding Cost por dia //
          ahday[productos] = ah[productos]/Period;
```

```
            //cout << " El Holding Cost por dia ($/unidad*dia), es: " << ahday[productos]
<< "\n" << endl;

            //cout << " " << endl;
            //cout << " " << endl;
            //cout << " " << endl;
            //cout << " a " << endl;

            F=0; // valor inicial para la variable de Lambda, es cero porque la 1era rutina de
solve no las subrutinas no la usan en la inicializacion



     // ********************************************** //

            S=0;

            // -- llamada a subrutina "solve" que me da los valores optimos de K y Tp para
cada producto -- //

solucion(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR
[productos],aT[productos],ahday[productos],aV[productos],Tp,B,S,z,&k ,&t,&Tuno);


            //cout << " " << endl;
            //cout << " ***************************************** " << endl;
            //cout << " *    Valores exportados de SUBRUTINA    * " << endl;
            //cout << " ***************************************** " << endl;
            //printf("T:\t%.20f",t); // Imprime el valor de la demanda con 20 decimales para
mas precision
            //cout << " " << endl;
            //cout << "K es: " << k << endl;
            //cout << "Tuno es: " << Tuno << endl;
            //cout << " " << endl;


            //TO[productos]=t;

     // -- Calculo de tiempos para los dos planes iniciales -- //
            //cout << " " << endl;
            tmin=(0.25)*(t); // La demanda para el plan minimo es un 50% de la demanda
optima
            tmax=(1.25)*(t); // La demanda para el plan maximo es un 25% mas grande que
la demanda optima
            //cout << " Tiempo de Prod -- debajo de capacidad: " << tmin << endl;
            //cout << " Tiempo de Prod -- encima de capacidad: " << tmax << endl;
            //cout << " " << endl;
            //cout << " " << endl;
            //cout << " ------------------------------------------------------------ " << endl;


            //Evaluacion Previa con tmax, para no permitir coeficientes negativos

            Tp=tmax;
           // cout << " Tmax es: " << tmax << endl;


            Neg[productos]= evaluar
(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR[product
os],aT[productos],ahday[productos],aV[productos],Tp,k,B,S,z) ;

            ///cout << " === INICIO > CICLO WHILE === " <<  endl;
            //cout << " " << endl;

            while ( Neg[productos] < 0 )
            {
                      Tp=Tp/1.1;
                      //cout << " Tp es: " << Tp << endl;
                      Neg[productos]= evaluar
(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR[product
os],aT[productos],ahday[productos],aV[productos],Tp,k,B,S,z) ;
              // cout << " El Negativo es: " << Neg[productos] << "\n" << endl;
               tmax=Tp;
            }

             //cout << " === FINAL > CICLO WHILE === " <<  endl;
             //cout << " Tmax es: " << tmax << endl;


            // -*- Calcular Coeficientes para los planes iniciales del LP -*- //

            // -- Coeficiente Minimo -- //
            Tp=tmin;
            //cout << " Tmin es: " << Tp << endl;
            aMIN[productos]= evaluar
(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR[product
os],aT[productos],ahday[productos],aV[productos],Tp,k,B,S,z) ;
            //cout << " **++ Error importado: " << *Et << endl;


            //cout << " El Coeficiente Minimo es: " << aMIN[productos] << "\n" << endl;
            // -- Coeficiente Maximo -- //
            Tp=tmax;
            //cout << " Tmax es: " << Tp << endl;
            aMAX[productos]= evaluar
(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR[product
os],aT[productos],ahday[productos],aV[productos],Tp,k,B,S,z) ;
            //cout << " El Coeficiente Maximo es: " << aMAX[productos] << "\n" << endl;

            k=1;
            Tp=1;
            aOPT[productos]= evaluar
(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR[product
os],aT[productos],ahday[productos],aV[productos],Tp,k,B,S,z) ;
            //cout << " El Coeficiente Irrestricto es: " << aOPT[productos] << "\n" << endl;

            aAopt[productos]= aT[productos] + Tuno;
            //cout << " El Coeficiente de tiempo Irrestricto es: " << aAopt[productos] <<
"\n" << endl;


                    if ( aOPT[productos] < 0 )
                            {
                            aOPT[productos] = 0;
                            aAopt[productos]=0;
                            }


            // -- Coeficiente de Restriccion de Tiempo Minimo -- //
            aAmin[productos]= aT[productos] + tmin;
            //cout << " El Coeficiente de tiempo Minimo es: " << aAmin[productos] << "\n"
<< endl;
                    //cout << " aT es: " << aT[productos] <<  "\n" << endl;
                    //cout << " D es: " << DE << endl;
                    //cout << " K es: " << k << endl;
                    //cout << " B es: " << B << endl;
                    //cout << " aM es: " << aM[productos] <<  "\n" << endl;


            // -- Coeficiente de Restriccion de Tiempo Maximo -- //
            aAmax[productos]= aT[productos] + tmax;
    //cout << " El Coeficiente de tiempo Maximo es: " << aAmax[productos] << "\n" <<
endl;
            //cout << " " << endl;
            //cout << " ***** FIN ***** " << endl;
            //cout << " producto:  " << productos << endl;

} // endFOR

b=b+1;

//cout << " b e: " << b << endl;



// --- *** --- Procedimiento para Obtener la T minima "factible" para comenzar el LP ---
*** --- //


                    Suma=0;
                    for (productos=0;productos<xint;productos++){
                              Suma = Suma + aAmin[productos];
                              } //end for
                    //cout << " " << endl;
                    //cout << " --- *** Suma Minima de Capacidades para los productos: " <<
Suma << endl;
                    //cout << " " << endl;


          while ( Suma > Capacidad){

                    tmin2=tmin/2;
                    //cout << " Tiempo modificado: " << tmin2 << endl;

                    Suma=0;

                    for (productos=0;productos<xint;productos++){

                    // -- Coeficiente Minimo -- //
                    Tp=tmin2;
                    aMIN[productos] = evaluar
(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR[product
os],aT[productos],ahday[productos],aV[productos],Tp,k,B,S,z) ;
             //cout << " El Coeficiente Minimo es: " << aMIN[productos] << "\n" << endl;


                    // -- Coeficiente de Restriccion de Tiempo Minimo -- //
                    aAmin[productos]= aT[productos] + tmin2;


                    if ( aMIN[productos] < 0 )
```

```
                        {
                        aMIN[productos] = 0;
                        aAmin[productos]=0;
                        }


                Suma = Suma + aAmin[productos];

                } //end for

                //cout << " " << endl;
                //cout << " --- *** Suma Minima de Capacidades para los productos: " <<
Suma << endl;
                //cout << " " << endl;

                tmin=tmin2;

        }; //endwhile suma vs. capacidad




// -----
**********************************************************************
** ----- //

 // lp solve  comienza ... //

 lprec *lp;
 HINSTANCE lpsolve;

 lpsolve = LoadLibrary("lpsolve55.dll");

 if (lpsolve == NULL) {
   printf("Unable to load lpsolve shared library\n");
   return(FALSE);
 }

 //
**********************************************************************
** //
 // Inicia el desarrollo de la estructura del programa //


 lp=make_lp(0,xint*3); /* prods*3 variables, 0 restricciones */
```

```
  // Descripcion del sistema utilizado //
  cout << "
**********************************************************************
" << endl;
  cout << " *                           LP SOLVE                           * " << endl;
  cout << "
**********************************************************************
" << endl;
  cout << " * Description: Open Source (Mixed-Integer) Linear Programming System  * "
<< endl;
  cout << " * Language: Multi-Platform, pure ANSI C / POSIX Source Code          * " <<
endl;
  cout << " * Official Name: lp_solve                                          * " << endl;
  cout << " * Release Data: Version 5.5.0.5. , dated: October 2005              * " << endl;
  cout << " * Co-developers: Michael Berkelaar, Kjell Eikland, Peter Notebaert    * " <<
endl;
  cout << " * Licence Terms: GNU LGPL (Lesser General Public Licence)            * " <<
endl;
  cout << "
**********************************************************************
" << endl;
  cout << " " << endl;
  cout << " " << endl;

  set_maxim(lp); // Define el problema como MAXIMIZAR

  set_lp_name(lp, "SPELSP 3.1.4"); /* Pone el nombre al programa */


  for (productos=0; productos<xint; productos++) // FOR 1
  {

  // Estructuracion de la funcion objetivo //
  row0[productos*3+1] = aMIN[productos];
  row0[productos*3+2] = aMAX[productos];
  row0[productos*3+3] = aOPT[productos];
  set_obj_fn(lp, row0);

  // Estructuracion de las restricciones del problema //
  set_add_rowmode(lp, TRUE);
  row1[productos*3+1] = aAmin[productos];
  row1[productos*3+2] = aAmax[productos];
  row1[productos*3+3] = aAopt[productos];

  for (i=0; i<xint*3+1; i++) row2[i]=0;
  set_add_rowmode(lp, TRUE);
  row2[productos*3+1] = 1.0;
  row2[productos*3+2] = 1.0;
  row2[productos*3+3] = 1.0;

  set_add_rowmode(lp, FALSE); // concluye la adicion de nuevas restricciones

  add_constraint(lp, row2, EQ , 1.0);

  }  // end FOR 1


  add_constraint(lp, row1, LE, Capacidad); // agrega la restriccion de Capacidad



  // Imprime la estructura del problema //
  cout << " " << endl;

  print_lp(lp);
  //cout << " " << endl;
  //cout << " " << endl;
  //cout << " *** Solucion *** " << endl;
  //cout << " " << endl;


  // Resuelve el problema y lo presenta en pantalla //
  solve(lp);
  //cout << " " << endl;
  cout << " " << endl;

  obj=get_objective(lp);


  // Obtiene los valores duales de la solucion //
  get_dual_solution(lp,duals);
  //cout << " " << endl;
  //cout << " *** duales *** " << endl;
  for (i=1; i<xint+2; i++) {
  //cout << " Precio Sombra: " << duals[i] << endl;
  }

  // Imprime la solucion del problema . 1 = numero de columnas para poner valores //
  print_solution(lp,1);
  cout << " " << endl;
  cout << " " << endl;
```

```
  // Imprime en pantalla los valores duales del problema //
  print_duals(lp);
  //cout << " " << endl;
  //cout << " " << endl;
  //cout << " *** FIN *** " << endl;
  //cout << " " << endl;
  //cout << " " << endl;


  // condiciones para ciclo de comparacion //
  condition = FALSE;
  agregadas = FALSE;
  Comparacion = (double *)calloc( xint, sizeof( double ) );



  ////////////////////////////////////////////////////////////////////////////////
  // *** --- Fase 2 > Subproblemas utilizando precios sombra --- *** //
  ////////////////////////////////////////////////////////////////////////////////

for (productos=0; productos<xint; productos++) // FOR 2
  {

  F=duals[xint+1]; // guarda el valor de LAMBDA para la subrutina, bajo el nombre de E;
  //cout << " El valor de lambda (variable F) es:  " << F << endl;

  // llamada a subrutina para obtener los valores de D y K  y volver a calcular coeficientes
//

  //Tp=TO[productos]; // variable que asigna el valor optimo a Tp, para calcular el Coef
entrante

  // llamada a solve para obtener nuevos valores optimos para T y K

solucion(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR
[productos],aT[productos],ahday[productos],aV[productos],Tp,B,S,z,&k ,&t,&Tuno);
  //solve
(F,aS[productos],aC[productos],aM[productos],aA[productos],aR[productos],aT[product
os],ahday[productos],aV[productos],Tp,B,S,z,&k ,&t,&Tuno);

  //printf(" T es:\t%.20f",t);
  //cout << " " << endl;
  //cout << " K es: " << k << endl;
  //cout << " " << endl;
  // Nuevo Coeficiente con Lambda
  CoefLambda[productos] = evaluar
(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR[product
os],aT[productos],ahday[productos],aV[productos],t,k,B,S,z) ;
                              //evaluar
(F,aS[productos],aC[productos],aM[productos],aA[productos],aR[productos],aT[product
os],ahday[productos],aV[productos],t,k,Period,B,S,z) ;

  //cout << " F(Lambda) es: " << F << endl;
  //cout << " T es: " << t << endl;
  //cout << " K es: " << k << endl;



  //cout << " " << endl;
  //cout << " El Coeficiente Modificado con Valores Sombra es: " <<
CoefLambda[productos] << "\n" << endl;
  //cout << " " << endl;



  F=0;



  // Nuevo Coeficiente para plan entrante //
  aNuevoCoef[productos] = evaluar
(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR[product
os],aT[productos],ahday[productos],aV[productos],t,k,B,S,z) ;
                              //evaluar
(F,aS[productos],aC[productos],aM[productos],aA[productos],aR[productos],aT[product
os],ahday[productos],aV[productos],t,k,Period,B,S,z) ;
  //cout << " *** El nuevo Coeficiente entrante es: " << aNuevoCoef[productos] << endl;
  //cout << "producto: " << productos+1 << endl;
  //cout << " " << endl;



   if ( ( aNuevoCoef[productos] < 0 ))
            CoefLambda[productos]=duals[productos+1];
    //  cout << " 9999 Comprobacion de Coef Lamba es: " << CoefLambda[productos] <<
"\n" << endl;



  //cout << " " << endl;
  diferencia = CoefLambda[productos] - duals[productos+1];
  //cout << " La diferencia es: " << diferencia << endl;
  //cout << " " << endl;
  //cout << " " << endl;
  Comparacion[productos] = diferencia;

  UpperSuma[productos] = Comparacion[productos];



  // Inicia condicion , una columna nueva entra si la diferencia es mayor a cero //
  condition = (Comparacion[productos] > value) || condition;


if (condition = TRUE)
  {
    agregadas = TRUE;
  }



  if ((Comparacion[productos] > value) ) {

  // Nuevo Coeficiente para plan entrante //

  aNuevoCoef[productos] = evaluar
(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR[product
os],aT[productos],ahday[productos],aV[productos],t,k,B,S,z) ;
                              //evaluar
(F,aS[productos],aC[productos],aM[productos],aA[productos],aR[productos],aT[product
os],ahday[productos],aV[productos],t,k,Period,B,S,z) ;
  //cout << " El nuevo Coeficiente entrante es: " << aNuevoCoef[productos] << endl;
  //cout << "producto: " << productos+1 << endl;
  //cout << " " << endl;

  // Nuevo Coeficiente de tiempo para plan entrante //
  aNuevoCoefA[productos] = aT[productos] + (t);
  //cout << " La restriccion de tiempo para el nuevo plan es: " <<
aNuevoCoefA[productos] << endl;
   //cout << "producto: " << productos+1 << endl;
   //cout << " " << endl;

  for (i=0; i<xint+2; i++) columna1[i]=0;

  if ((Comparacion[productos] > 0)) {
            columna1[productos+1]=1.0;
            columna1[0] = aNuevoCoef[productos];
      columna1[xint+1] = aNuevoCoefA[productos];
            add_column(lp, columna1);
  }
  cout << " " << endl;
  cout << " " << endl;

  } // end IF
  //print_lp(lp);

  } // end FOR 2



for (w=0;w<xint;w++)
{
        //cout << "Ub Suma [ " << w << " ] es : " << UpperSuma[w] << endl;
}

if ( UpperSuma[w] < 0){
        UpperSuma[w]=0;
}

  // ciclo para obtener el Upper Bound de la funcion //
  suma=0;
  for (w=0;w<xint;w++)
  {
          suma = suma + UpperSuma[w];
  }

  if (suma < 0 ){
          suma=0;
  }

  //cout << " objetivo es: " << obj << endl;
  UB = obj + suma ;
  bestUB = UB;


  //cout << " *********************** " << endl;
  //cout << " *** *** **** UB : " << bestUB << endl;
  //cout << " *********************** " << endl;

  ///////////////////////////////////////////////////////////////////////////


 // comienza el WHILE externo //
 while ( porcdif > tolerancia ) {

  conteo=0;
 // comienza el WHILE interno //
 while  (agregadas && (conteo < maxcount)) {

                        conteo++;

              // Solve LP //

                        solve(lp);

                        set_timeout(lp, 20);

                        //print_solution(lp,1);

                        obj=get_objective(lp);
                        //cout << " 1111111111111111111 Obj> " << obj << endl;

                        // Get dual values //
                        get_dual_solution(lp,duals);
                        //cout << " " << endl;
                        //cout << " *** duales *** " << endl;
                        for (i=1; i<xint+2; i++) {
                        //cout << " Precio Sombra: " << duals[i] << endl;
                        }; // end FOR interno


                        for (productos=0; productos<xint; productos++)
                        {


                                   F=duals[xint+1]; // guarda el valor de LAMBDA para la
subrutina, bajo el nombre de E;
                                   //cout << "
****************************************** " << endl;
                                   //cout << " El valor entrante de lambda (variable F) es:  "
<< F << endl;
                                   //cout << "
****************************************** " << endl;
                                   //cout << " " << endl;

                                   // llamada a subrutina para obtener los valores de D y K  y
volver a calcular coeficientes //
                                   //Tp=TO[productos];

                                   // Nuevo Coeficiente con Lambda
```

```
                                //cout << "
******************************************* " << endl;
                                //cout << "
******************************************* " << endl;
                                //cout << " LLAMADA A SUBRUTINA " << endl;


        solucion(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[produc
tos],aR[productos],aT[productos],ahday[productos],aV[productos],Tp,B,S,z,&k ,&t,&Tu
no);
                                //solve
(F,aS[productos],aC[productos],aM[productos],aA[productos],aR[productos],aT[product
os],ahday[productos],aV[productos],t,B,S,z,&k ,&t,&Tuno);

                                //printf(" T es:\t%.20f",t);
                                //cout << " " << endl;
                                //cout << " K es: " << k << endl;



                                CoefLambda[productos] = evaluar
(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR[product
os],aT[productos],ahday[productos],aV[productos],t,k,B,S,z) ;
                                                    //evaluar
(F,aS[productos],aC[productos],aM[productos],aA[productos],aR[productos],aT[product
os],ahday[productos],aV[productos],t,k,Period,B,S,z) ;
                                //cout << " El Coeficiente Modificado con Valores Sombra
es: " << CoefLambda[productos] << "\n" << endl;
                                //cout << " " << endl;



                                F=0; // para recomenzar el ciclo de valores optimos de T y
K

                                // Nuevo Coeficiente para plan entrante //
                                aNuevoCoef[productos] = evaluar
(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR[product
os],aT[productos],ahday[productos],aV[productos],t,k,B,S,z) ;
                                                    //evaluar
(F,aS[productos],aC[productos],aM[productos],aA[productos],aR[productos],aT[product
os],ahday[productos],aV[productos],t,k,Period,B,S,z) ;
                                //cout << " *** Evaluacion de Coef Entrante: " <<
aNuevoCoef[productos] << endl;
                                //cout << "producto: " << productos+1 << endl;
                                //cout << " " << endl;

                                if ( ( aNuevoCoef[productos] < 0 ))
```

```
                                        CoefLambda[productos]=duals[productos+1];
                              //  cout << " 9999 Comprobacion de Coef Lamba es: " <<
CoefLambda[productos] << "\n" << endl;



                              //cout << " " << endl;
                              diferencia = CoefLambda[productos] - duals[productos+1];
                              //cout << " ---------------------------------------- " << endl;
                              //cout << " La diferencia es: " << diferencia << endl;
                              //cout << " " << endl;
                              //cout << " " << endl;
                              Comparacion[productos] = diferencia;

                              UpperSuma[productos] = Comparacion[productos];


                              // Inicia condicion , una columna nueva entra si la
diferencia es mayor a cero //
                              condition = (Comparacion[productos] > value) || condition;


                              if (condition = TRUE)
                              {
                                        agregadas = TRUE;
                              }



                              if ((Comparacion[productos] > value)) {

                              // Nuevo Coeficiente para plan entrante //
                              aNuevoCoef[productos] = evaluar
(b,xint,noise,Et,F,aS[productos],aC[productos],aM[productos],aA[productos],aR[product
os],aT[productos],ahday[productos],aV[productos],t,k,B,S,z) ;
                                                  //evaluar
(F,aS[productos],aC[productos],aM[productos],aA[productos],aR[productos],aT[product
os],ahday[productos],aV[productos],t,k,Period,B,S,z) ;
                              //cout << " El nuevo Coeficiente entrante es: " <<
aNuevoCoef[productos] << endl;
                              //cout << " " << endl;

                              // Nuevo Coeficiente de tiempo para plan entrante //
                              aNuevoCoefA[productos] = aT[productos] + (t);
                              //cout << " La restriccion de tiempo para el nuevo plan es: "
<< aNuevoCoefA[productos] << endl;
```

```
                                        //cout << "producto: " << productos+1 << endl;
                                        //cout << " " << endl;

                                        for (i=0; i<xint+2; i++) columna1[i]=0;
                                        if ((Comparacion[productos] > 0)) {
                                        columna1[productos+1]=1.0;
                                        columna1[0] = aNuevoCoef[productos];
                                        columna1[xint+1] = aNuevoCoefA[productos];
                                        add_column(lp, columna1);

                                        } //end if

                                        //print_lp(lp);


                                        } // end IF condicional


                              } // end FOR


for (w=0;w<xint;w++)
{
          //cout << "Ub Suma [ " << w << " ] es : " << UpperSuma[w] << endl;
}

if ( UpperSuma[w] < 0){
          UpperSuma[w]=0;
}

  // ciclo para obtener el Upper Bound de la funcion //
  suma=0;
  for (w=0;w<xint;w++)
  {
            suma = suma + UpperSuma[w];
  }

  if (suma < 0 ){
            suma=0;
  }

  //cout << " ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZz objetivo es: "
<< obj << endl;
  UB = obj + suma ;
  bestUB = UB;
```

```
            //cout << " *** *** **** UB : " << bestUB << endl;

            print_lp(lp);
            //cout << " " << endl;

            //cout << " " << endl;
            cout << "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@ " << endl;
           cout << " @@@@ @@@@ CONTEO DE ITERACIONES: " << conteo << endl;
            cout << "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@ " << endl;
            //cout << " " << endl;
            //cout << " " << endl;


  };// end WHILE interno



  for (i=1;i<=get_Ncolumns(lp);i++)
  set_int(lp, i, TRUE);

  // resuelve el nuevo LP con integer variables //
  solve(lp);
  print_lp(lp);
  print_solution(lp,1);
  cout << " " << endl;
  cout << " " << endl;

  // Obtiene el valor de la funcion objetivo entera = UB //
  objentero=get_objective(lp);
  cout << " El objetivo LP: " << obj / (48) << endl;
  cout << " El objetivo entero: " << objentero / (48) << endl;

  // Calculo de Upper bound //
  cout << " El Upper Bound (UB) es: " << bestUB / (48) << endl;

  // Calculo de diferencia porcentual //
  porcdif = (bestUB-objentero)/(bestUB);
  cout << " " << endl;
  cout << " ****************************************************** " <<
endl ;
  cout << " *                                                    * " << endl ;
  cout << " * --------------     RESULTADO FINAL    -------------- * " << endl;
  cout << " *                                             * " << endl;
  cout << " *  El porcentaje de error es:  " << porcdif*100 << endl;
  cout << " *                                             * " << endl ;
  cout << " ****************************************************** " <<
endl ;

  // Cambia las variables a no enteras para que vuelva a iterar sin ellas //
  for (i=1;i<=get_Ncolumns(lp);i++)
  set_int(lp, i, FALSE);

}; // end WHILE externo



// ** CPU TIME //
finish = clock();
duration = (finish-start);
cputime=duration/CLOCKS_PER_SEC;// se usa CLCOKS_PER_SEC porque cada
unidad del clock dura 1/1000 de segundo.
printf (" CPU time taken is:  %.2lf seconds.\n", cputime);
// **

set_timeout(lp, 30);

delete_lp(lp);

FreeLibrary(lpsolve);

return (0);

}; // end MAIN



////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
//
**********************************************************************
*****////
// *   2. SUBRUTINA para Subproblema , para resolver el asunto  K y d optimos. * ///
```

```
//
***************************************************************************
*****////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////


double solucion (int b, int xint, int noise, double *E, double F,double aSp, double aCp,
double aMp, double aAp,double aRp ,double aTp ,double ahdayP ,double aVp,double Tp,
double B,double S,double z,double *k ,double *t,double *Tuno)
{

        int l,c,a;
        double Utt,lb,ub,Tiempo,tiempo,Time,tempo;
        double Uoptant,Utotal,Uprima,Ug;
        double K,ka;
        double Step,StepPe;


// -- Inizio> algunos valores iniciales para el ciclo while externo
c=1;
l=1;
K=1;
a=1;
Step=0.1;
StepPe=0.01;
Time=0; // variable que almacena Tps

Utt= -1000000000000;
Uoptant = -1000000000000000000;

// -- Fine




while ( Utt > Uoptant ) { ////// while EXTERNO , para optimizar K

//cout << " " << endl;
//cout << " " << endl;
//cout << " " << endl;
//cout << " ** [-----------------------------------] ** " << endl;
//cout << "      Comienza el ciclo de K = " << K << " : " << endl;
//cout << " ** [-----------------------------------] ** " << endl;
//cout << " " << endl;
```

```
//cout << " " << endl;
//cout << " " << endl;

Uoptant=Utt;


// -- Inizio> algunos valores iniciales para el ciclo while interno
Utotal= -1000000000000;
Uprima = -1000000000000000000;
Tp = 0.1; // recomenzar el ciclo de K, usando valores de Tp desde el valor dado en esta
parte
// -- Fine


while ( Utotal > Uprima ) { // while INTERNO

//cout << " ##### ---- Inicio del While Interno" << endl;

Uprima=Utotal; // Uprima asume el valor Utotal del periodo anterior para poder comparar
//cout << "Uprima es : " << Uprima << endl;

// llamada a tercera rutina, para obtener la evaluacion y el valor de Uopt
Utotal = evaluar (b,xint,noise,E,F,aSp,aCp,aMp,aAp,aRp,aTp,ahdayP,aVp,Tp,K,B,S,z) ;

tiempo=Tp-Step;

Tp=Tp + Step ;   // Incrementa el ciclo de Conteo para Tp, usando Step fijo


//cout << " " << endl;
// -- Valores importantes para el ciclo de while interno
//cout << " Valores de> While Interno" << endl;
//cout << "Utotal es : " << Utotal << endl;
//cout << "Uprima es : " << Uprima << endl;
//cout << " " << endl;

Ug = Utotal;



}; // end WHILE interno



//cout << " *  La utilidad optima es:  " << Uprima << endl; // es l-2 porque 2 posiciones
mas se cuentan
```

```
//cout << " *  El tiempo optimo de Produccion es :  " <<  tiempo  << endl;
//cout << " " << endl;

//Topt[c]=tiempo; // almacena el valor optimo del Tiempo de Produccion, para la K
correspondiente
//cout << " *  El Tp  es:  " << tiempo << endl;
//cout << " *  El Tiempo  es:  " << Time << endl;

tempo=Time;

K=K+1; // se agrega K y se recomienza el ciclo
c=c+1;

Utt=Uprima;

Time=tiempo; // para guardar el valor correcto de Tp


}; // end while externo


c=c-1;
K=K-1;

//cout << " *  El tempo  es:  " << tempo << endl;

//cout << " " << endl;
//cout << " " << endl;
//cout << " ****************************************************** " <<
endl ;
//cout << " *                                              * " << endl ;
//cout << " * ------------   RESULTADO OPTIMIZADO    ------------- * " << endl;
//cout << " *                                              * " << endl ;
//cout << " *  La utilidad Optima Global es:  " << Uoptant << endl;
//cout << " *  La K optima es:  " << K-1 << endl;
//cout << " *  El Tp optimo es:  " << tempo << endl;
//cout << " *                                              * " << endl ;
//cout << " ****************************************************** " <<
endl ;

//cout << " " << endl;
//cout << " " << endl;

//////////////////////////////////////////////////////////////////////////////////////////
// ^^^^^ Conjunto de actividades adjuntas a la rutina, para lograr una mejor optimizacion
// ^^^ La idea es calcular limites para volver a realizar una optimizacion mas fina
```

```
lb=tempo-0.1; // lower bound donde comienza la nueva optimizacion
ub=tempo+0.1; // upper bound donde termina la nueva optimizacion

//cout << " " << endl;
//cout << " ^^ " << endl;
//cout << " *^^  El LB optimo es:  " << lb << endl;
//cout << " *^^ El UB optimo es:  " << ub << endl;
//cout << " ^^ " << endl;
//cout << " " << endl;

//^^
Utotal= -1000000000000;
Uprima = -1000000000000000000;
Tp = lb ;
l=1;
//^^

ka=K-1;


while ( Utotal > Uprima ) // WHILE q realiza la optimizacion requerida

{

Uprima=Utotal; // Uprima asume el valor Utotal del periodo anterior para poder comparar
//cout << "Uprima es : " << Uprima << endl;

// llamada a tercera rutina, para obtener la evaluacion y el valor de Uopt
Utotal = evaluar (b,xint,noise,E,F,aSp,aCp,aMp,aAp,aRp,aTp,ahdayP,aVp,Tp,ka,B,S,z) ;
Tp=Tp + StepPe ; // Incrementa el ciclo de Conteo para Tp

//cout << " " << endl;

// Variable Ug[l] , para almancenar los valores de la Utilidad en cada Tp
Ug  = Utotal;


}; // end WHILE



//cout << " *  La utilidad optima es:  " << Uprima << endl; // es l-2 porque 2 posiciones
mas se cuentan
//cout << " *  El tiempo optimo de Produccion es :  " <<  Tp - 0.02  << endl;
```

```
Tiempo = Tp - (StepPe*2); // almacena el valor optimo del Tiempo de Produccion, para
la K correspondiente



// -- Valores que se exportaran a main --

*k=K-1; // asignacion de valor optimo de K al apuntador correspondiente
*t=Tiempo;  // asignacion de valor optimo de Tp al apuntador correspondiente

*Tuno=tempo;

//cout << " **************************************** " << endl;
//cout << " COMPROBACION DE VALORES (K,Tp) DE SUBRUTINA " << endl;
//cout << " " << endl;
cout << " La K optima es: " << *k << endl;
//cout << " " << endl;
cout << " La Tp optima es: " << *t  << endl;
//cout << " " << endl;
//cout << " La D es: " << *Dd << endl;
//cout << " " << endl;
//cout << " **************************************** " << endl;
//cout << " " << endl;

return (0);

} // end SUBPROBLEMA







////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
//
*************************************************************************
*****////
// *   3. SUBRUTINA Evaluadora , para obtener los valores de Utilidad por caso * ///
//
*************************************************************************
*****////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

```
double evaluar (int b,int xint, int noise, double *E, double F,double aSp, double aCp,
double aMp, double aAp,double aRp ,double aTp ,double ahdayP ,double aVp, double Tp,
double K,double B,double S, double z)
{


//cout << " $$$ --- SUBRUTINA --- $$$ " << endl;
//cout << " " << endl;
//cout << " " << endl;
//cout << " $$$ VALORES EXPORTADOS $$$ " << endl;
//cout << " " << endl;
//cout << " F(Lambda) es: " << F << endl;
//cout << " aSp es: " << aSp << "\n" << endl;
//cout << " aCp es: " << aCp << "\n" << endl;
//cout << " aMp es: " << aMp << "\n" << endl;
//cout << " aSp es: " << aSp << "\n" << endl;
//cout << " aAp es: " << aAp << "\n" << endl;
//cout << " aRp es: " << aRp << "\n" << endl;
//cout << " aTp es: " << aTp << "\n" << endl;
//cout << " ahday2 es: " << ahdayP << "\n" << endl;
//cout << " aVp es: " << aVp << "\n" << endl;
//printf(" H es:\t%.20f",ahday2);
//cout << " " << endl;
//cout << " " << endl;
//cout << " V es: " << aV << "\n" << endl;
//cout << " b es: " << b << "\n" << endl;
//cout << " xint es: " << xint << "\n" << endl;
//cout << " noise es: " << noise << "\n" << endl;



int i,w,a,indice;

double d;
double P;
double Utres,Ucuatro; // Guardan los valores de utilidades para cada tipo de caso
double Prod,meta,Sum;
double Compra,Utotal,Uopt,UP,Usuma,Step;
double q;      // inventario inicial
double ku;     // inventario inicial para ciclo IF
double Q;      // inventario final
double s;      // Sigma para k=1, Sigma para cualquier valor de K

double *EE; //arreglos de ruidos
```

```
/// algunos valores iniciales
Step=0.1;
///

EE = (double *)calloc( xint*noise ,sizeof( double ) );
//


// --- FIN - declaracion de variables --- //


//cout << " " << endl;
//cout << " **************************************** " << endl;
//cout << " *            3. EVALUACION          * " << endl;
//cout << " **************************************** " << endl;
//cout << " " << endl;

for (a= (b-1)*noise ; a < noise + (b-1)*noise ;a++)
{
        //cout << " El ruido exportado de main { " << a << " } es: " << E[a] << endl;
}

//cout << " **************************************** " << endl;

// -- //  ciclo para almacenar los ruidos , segun la K , este es el ciclo importante


        a=0; // inicializar variable que guarda ruidos

        indice = (b-1)*noise;

        ///////////////////////
        for (i=0;i<(noise/K);i++)
        { // inicia FOR , ciclo C

                // inicializar variables en uso, b y sum
                w=0;
                Sum=0;

                while ( w < K )
                { // inicia WHILE, ciclo C

                        Sum = E[indice] + Sum;
                        indice++;
                        w=w+1;

                } // termina WHILE , ciclo C

                //i=i-1; // arreglo para restar un periodo a la suma hecha en el ciclo anterior

                //cout << " i es: " << i << endl;
                //cout << " La suma: " << Sum << endl;
                //cout << " " << endl;

                EE[a]=Sum/K; // Arreglo que guarda el valor de los ruidos, segun la K
                a=a+1;

        } // termina FOR , ciclo C
        ///////////////////////

        //cout << " " << endl;

//
for (a=0;a<(noise/K);a++)
{
        //cout << "El ruido E de k[ " << a << " ] es : " << EE[a] << endl;
}
//

//- Periodo Inicial -//

// *  algunas variables que se deben definir al comenzar cada periodo */

//cout << " " << endl;
//cout << " *** " << endl;
//cout << " " << endl;
//cout << " CONDICIONES DE EVALUACION AL COMENZAR CADA PERIODO"
<< endl;
//cout << " " << endl;

s = 0.05; // desviacion estandar
S = s*sqrt(K); // valor real de sigma , para k mayor a 0

ku=0; // inventario inicial

// == ver algunos valores
//cout << " ku es: " << ku << endl;
//cout << " aMp es: " << aMp << endl;
//cout << " Tp es: " << Tp << endl;
//cout << " B es: " << B << endl;
```

```
//cout << " K es: " << K << endl;
//cout << " z es: " << z << endl;
//cout << " S es: " << S << endl;

// Demanda
d = ( (ku+(aMp*Tp)) ) / ( B *( K + z*S ) ) ;
//cout << " La tasa de demanda , d , para este periodo es: " << d << " unidades por dia "
<< endl;
//cout << " " << endl;

// Precio
P = (aRp - d)/(aAp);
//cout << " El precio, P , para este periodo es: " << P << " unidades por dia " << endl;
//cout << " " << endl;

// Cantidad de produccion
Prod = Tp*aMp;
//cout << " La produccion para el periodo sera de: " << Prod << " unidades " << endl;
//cout << " " << endl;
//cout << " " << endl;
//cout << " " << endl;

//inicializadores
Utres=0;
Ucuatro=0;



//////////////////////////////////////////////////////////////////////

// Inventario Inicial para periodos subsecuentes , donde ku=I=z*s*m*t1


q=0; // inicializador de variables



//--{} Ciclo IF para determinar el valor del inventario Inicial

if (q<0.0){
        ku= (z*S*aMp*Tp)/(K);} // Inventario Inicial puesto para cierto nivel de servicio ,
Se compra
else
{
         ku = q;}  // Inventario Inicial es igual el Inv. Final del periodo Anterior
```

```
//--{} Fin ciclo IF



////////////////////////////////
Usuma=0;
for ( a=0 ; a<(noise/K) ; a++ ) // ciclo FOR para simulaciones de periodos
{

        //cout << " " << endl;

        if ( ku < 0 )
                ku= (z*S*aMp*Tp)/(K);

        d = ((ku+(aMp*Tp))/((B)*(K+(z*S)))); // calculo de demanda para cada periodo


        P = (aRp - d)/(aAp); // calculo de precio , segun demanda


        Q= Tp*aMp + ku - d*K*B*(EE[a]); // inv. final
        //cout << " El ruido Ea es: " << E[a] << endl;

        //cout << " El INVENTARIO INICIAL del periodo es: " << ku << endl;

        //cout << " El INVENTARIO FINAL del periodo es: " << Q << endl;


        meta= Tp*aMp - (d)*(K*B) + ku; // inventario inicial
        //cout << " La META del periodo es: " << meta << " unidades " << endl;



// ---- EVALUACION DE CASOS DEL ELSP ---- //

// ** ciclos IF para los dos casos posibles //

if ( ( ku >= 0.0 ) & ( Q > 0.0 ) )
{

        // Caso 3 //
        Utres = P*(d)*(EE[a]*K)*B - aSp - aCp*aMp*Tp - (aCp*(ahdayP))*ku*K -
(aCp*(ahdayP)*K)*(aMp*Tp)*(1-(Tp/(2*K*B))) + (aCp*(ahdayP)*K)*(EE[a]*B*d*0.5);

        //cout << " price> " << P*(d)*(E[a]*K)*B << endl;
        //cout << " setup> " << - aSp << endl;
```

```
	//cout << " prod> " << - aCp*aMp*Tp << endl;
	//cout << " inv1> " <<- (aCp*(ahdayP/12))*ku*K << endl;
	//cout << " inv2> " << - (aCp*(ahdayP/12)*K)*(aMp*Tp)*(1-(Tp/(2*K*B))) <<
endl;
	//cout << " inv3> " << + (aCp*(ahdayP/12)*K)*(E[a]*B*d*0.5) << endl;


	UP=Utres;
	//cout << " ** Caso TRES ** " << endl;
	//cout << " Utilidad del periodo " << a << " es: " << UP[a] << endl;
	//cout << "  " << endl;

	ku=Q; // inventario inicial del sig. periodo es mi inventario final en este periodo


}

if ( ( ku >= 0.0 ) & ( Q < 0.0 ) )
{

	// Caso 4 //
	Ucuatro = P*(d)*(EE[a]*K)*B - aSp - aCp*aMp*Tp -
aVp*((d*(EE[a]*K)*B)+(aMp*Tp)*(((z*S)/(K)) - 1)) -
aCp*K*(ahdayP)*((Tp*Tp)/(2*B))*(((aMp*aMp)/(d*(((EE[a]))))) - (aMp/K))  -
((aCp*K*(ahdayP)*(ku))/(B*d*((EE[a]))))*((Tp*aMp) + (ku/2)) ;

	//cout << " 1: " << P*(d)*(E[a]*k)*B << endl;
	//cout << " setup: " << -aS << endl;
	//cout << "prod: " << -aC*aM*Tp << endl;
	//cout << " v: " << - V*((d*(E[a]*k)*B) +(aM*Tp)*(((z*S)/(k)) - 1)) << endl;
	//cout << " inv1 " << - aC*k*(aH/12)*((Tp*Tp)/(2*B))*(((aM*aM)/(d*(((E[a])))))
- (aM/k)) << endl;
	//cout << " inv2 " << - ((aC*k*(aH/12)*(ku))/(B*d*((E[a]))))*((Tp*aM) + (ku/2))
<< endl;

	UP=Ucuatro;
	//cout << " ** Caso CUATRO ** " << endl;
	//cout << " Utilidad del periodo " << i << " es: " << UP[a] << endl;

	Compra = ((d)*(EE[a])*(K)*B)+(aMp*Tp)*(((z*S)/(K)) - 1);
	//cout << " Se necesitan comprar: " << Compra << " unidades " << endl;
	//cout << "  " << endl;

	ku=Q; // Cambio de variable para que el Inv. Final del periodo Anterior sea el Inv.
Inicial del periodo siguiente
```

```
}

//Variable que va acumulando el valor de las utilidades
Usuma = UP + Usuma;

} // termina ciclo FOR



//////////////////////////////////////////////////////////////////////////////////////////////
// *** -- e. Resultado Final del PELSP  -- **************************** //
//////////////////////////////////////////////////////////////////////////////////////////////



////////////// Ya no lo uso -- 21 jun06 //////////////////// Ciclo FOR para suma de periodos
//for ( a=1 ; a<(periodos/K) ; a++ ){
//          Suma = UP[a] + Suma;}


Utotal = Usuma - Tp*F - aTp*F;

//cout << " " << endl;
//cout << " " << endl;

//cout << " * --------------     RESULTADO FINAL    -------------- * " << endl;
//cout << " *                                                      * " << endl;
//cout << " *  La utilidad total para la simulacion es:  " << Utotal << endl;
//cout << " *  El tiempo de Produccion es :  " << Tp  << endl;
//cout << " * -------------------------------------------------- * " << endl;


Tp=Tp + Step ; // Incrementa el ciclo de Conteo para Tp

//valores que se transferiran:

Uopt=Utotal;


//cout << " --- Fin Tercera Rutina --- " << endl;
//cout << " " << endl;
//cout << " " << endl;

          return (Uopt);
```