
# Implementing and Comparing Stochastic and Robust Programming

Université de Liège – Faculté des sciences appliquées

Année académique 2014 - 2015

Mémoire de fin d'études réalisé en vue de l'obtention du grade de master ingénieur civil en informatique par Thibaut Cuvelier.

Traditional optimisation tools focus on deterministic problems: scheduling airline flight crews (with as few employees as possible while still meeting legal constraints, such as maximum working time), finding the shortest path in a graph (used by navigation systems to give directions, usually based on GPS signals), etc.

However, this deterministic hypothesis sometimes yields useless solutions: actual parameters cannot always be known to full precision, one reason being their randomness. For example, when scheduling trucks for freight transportation, if there is unexpected congestion on the roads, the deadlines might not be met, the company might be required to financially compensate for this delay, but also for the following deliveries that could not be made on schedule.

Two main approaches are developed in the literature to take into account this uncertainty: take decision based on probability distributions of the uncertain parameters (stochastic programming) or considering they lie in some set (robust programming). In general, the first one leads to a large increase in the size of the problems to solve (and thus requires algorithms to work around this dimensionality curse), while the second is more conservative but tends to change the nature of the programs (which can impose a new solver technology).

Some authors [2] claim that those two mindsets are equivalent, meaning that the solutions they provide are equivalent when faced with the same uncertainty. The goal of this thesis is to explore this question: for various problems, implement those two approaches, and compare them.

- Is one solution more secluded from variations due to the uncertain parameters?
- Does it bring benefits over a deterministic approach?
- Is one cheaper than the other to compute?

# Contents


**1 Why Optimising Under Uncertainty** **1**
- 1.1 Deterministic optimisation . . . 2
  - 1.1.1 Main structures . . . 2
  - 1.1.2 Mixed integer programs . . . 3
- 1.2 Optimisation under uncertainty . . . 4
- 1.3 Stochastic optimisation . . . 5
  - 1.3.1 One stage . . . 6
  - 1.3.2 Two stages . . . 6
  - 1.3.3 Chance constraint . . . 7
- 1.4 Robust optimisation . . . 7
  - 1.4.1 Soyster uncertainty . . . 8
  - 1.4.2 Ellipsoidal uncertainty . . . 9
  - 1.4.3 Deriving an uncertainty set . . . 11
  - 1.4.4 Other uncertainty models . . . 11
- 1.5 Comparing results under uncertainty . . . 12

**2 Modelling Under Uncertainty** **13**
- 2.1 Facility location . . . 13
  - 2.1.1 Deterministic model . . . 14
  - 2.1.2 Stochastic model . . . 15
  - 2.1.3 Robust model . . . 16
- 2.2 Unit-commitment . . . 16
  - 2.2.1 Deterministic model . . . 17
  - 2.2.2 Stochastic two-stage model . . . 20
  - 2.2.3 Robust model . . . 20
  - 2.2.4 Failures . . . 21

**3 Solving Deterministic Problems** **23**
- 3.1 Simplex algorithm . . . 23
- 3.2 Branch-and-bound . . . 24
- 3.3 Cutting planes . . . 24
- 3.4 Duality theory . . . 25
- 3.5 Software . . . 26

**4 Solving Stochastic Problems** **27**
4.1 Deterministic equivalent . . . 27
4.1.1 One-stage stochastic program . . . 28
4.1.2 Two-stage stochastic program . . . 28
4.2 Benders' decomposition . . . 29
4.2.1 Formal derivation . . . 31
4.2.2 Multicut Benders' decomposition . . . 32
4.2.3 Discussion . . . 32
4.3 Progressive hedging . . . 33
4.3.1 Algorithm statement . . . 33
4.3.2 Possible improvements . . . 34
4.3.2.1 Choice of $\rho$ . . . 34
4.3.2.2 Cyclic behaviour . . . 35
4.3.2.3 Convergence acceleration . . . 35
4.3.3 Discussion . . . 36
4.4 Software . . . 36

**5 Solving Robust Problems** **37**
5.1 Cutting plane . . . 37
5.2 Reformulation . . . 38
5.3 Software . . . 39

**6 Comparing Performance: Facility Location** **41**
6.1 Implementation details . . . 41
6.1.1 Stochastic programming . . . 41
6.1.2 Robust programming . . . 42
6.1.3 Implementation languages . . . 42
6.1.4 Instance generation . . . 44
6.1.5 Performance comparison . . . 44
6.2 Comparison between algorithms . . . 45
6.2.1 Number of solved instances . . . 45
6.2.2 Time to convergence . . . 46
6.2.3 Iterations count . . . 47
6.2.4 Conclusions . . . 48
6.3 Comparison between languages . . . 48

**7 Comparing Robustness: Unit-Commitment** **51**
7.1 Implementation details . . . 51
7.1.1 Instance generation . . . 51
7.1.2 Models implementation . . . 52
7.1.3 Instance use . . . 52
7.2 Need for flexibility . . . 53
7.3 Change in context . . . 54
7.3.1 Demand increase . . . 54
7.3.2 Failures . . . 57
7.3.3 Sooner recourse action . . . 59

7.4 Monte Carlo evaluation . . . . . 62
7.5 Summary . . . . . 64

**8 Drawing Conclusions** **65**
8.1 Stochastic vs. robust . . . . . 66
8.1.1 Modelling issues . . . . . 66
8.1.2 Implementation issues . . . . . 67
8.1.3 Applicability . . . . . 67
8.1.4 Personal opinion . . . . . 67
8.2 Going further . . . . . 68

# List of Figures


1.1 MIP solver benchmark (shifted geometric mean of results). Taken from SCIP's website [`http://scip.zib.de/`]; based on [22]. . . 4
1.2 Joint probability density function $\mathrm{pdf}_{\mathcal{X},\mathcal{Y}}$ of two random variables with Gaussian distribution. . . . . 9
1.3 Ellipsoid obtained as $\mathrm{pdf}_{\mathcal{X},\mathcal{Y}} \leq \rho$, where $\rho$ controls its size. . . . 9
1.4 Comparison of different uncertainty sets centred on a point. . . 10

2.1 Facility location. . . . . 14

3.1 Feasible area in branch-and-bound after two branching steps. . . 24
3.2 Branch-and-bound search tree. . . . . 24

4.1 Structure of a two-stage stochastic program: variables do not depend on scenario until $t = t_r$, then decisions may diverge (dark). With nonanticipativity constraints, all variables are allowed to depend on the scenario: the bifurcation happens at $t = 0$ rather than $t = t_r$ (light). . . . . 29
4.2 Structure of the constraint matrix in (4.4). . . . . 30

6.1 Facility location: number of instances solved depending on algorithm (Java implementation). . . . . 46
6.2 Facility location: time to convergence depending on algorithm (Java implementation). . . . . 47
6.3 Facility location: number of iterations depending on algorithm (Java implementation). . . . . 48
6.4 Facility location: mean iteration time, logarithmic scale (Java implementation). . . . . 48
6.5 Facility location: number of instances solved depending on algorithm (Julia implementation). . . . . 49
6.6 Facility location: time to convergence depending on algorithm (Julia implementation). . . . . 49

7.1 Unit-commitment: scenarios, average scenario, and $L_\infty$ ellipsoid visualisation. These follow an average day on the Belgian grid. . 53

7.2 Unit-commitment: comparison of solutions to the less flexible instance. . . . . 54
7.3 Unit-commitment: cost distribution depending on the model (more flexible instance). . . . . 54
7.4 Unit-commitment: comparison of feasibility when the data is worsened by five percent. . . . . 55
7.5 Unit-commitment: maximum $L_\infty$ ellipsoid size for the given solutions to be feasible when the demand is worsened by five percent. . . . . 56
7.6 Unit-commitment: cost distribution (stochastic evaluation through scenarios) when demand is worsened by five percent. . . . . 56
7.7 Unit-commitment: first-stage solution without failures on average scenario, showing which machines are used. . . . . 57
7.8 Unit-commitment: feasibility count of the various solutions against the other models when failures are added. . . . . 58
7.9 Unit-commitment: maximum $L_\infty$ ellipsoid size for the given solutions to be feasible when failures are added. . . . . 59
7.10 Unit-commitment: cost distribution (stochastic evaluation through scenarios) when failures are added. . . . . 59
7.11 Unit-commitment: feasibility count of the various solutions against the other models when the recourse action happens sooner. . . . 60
7.12 Unit-commitment: maximum $L_\infty$ ellipsoid size for the given solutions to be feasible when the recourse action happens sooner. . . 61
7.13 Unit-commitment: cost distribution (stochastic evaluation through scenarios) when the recourse action happens sooner. . . . . 61
7.14 Unit-commitment: cost distribution (stochastic evaluation through scenarios) when the recourse action happens sooner; data is worsened by ten percent when evaluating the cost. . . . . 62
7.15 Unit-commitment: feasibility count on 1,000 scenarios. . . . . 63
7.16 Unit-commitment: cost distribution (stochastic evaluation through scenarios) averaged on 1,000 scenarios. . . . . 64

# Chapter 1

# Why Optimising Under Uncertainty

Uncertainty is obviously present in many situations. Measures cannot be done to the last digit, due to the sensor sensitivity; computations based on those measures can then use imprecise constants ($\pi$, speed of light, acid dissociation constant, etc. are usually known to some precision), which adds to the uncertainty. Also, forecasts are by essence approximate, and their error grows as the predicted instant is further away from the last known point: the electric production due to wind heavily depends on the weather, which cannot be forecast months in advance with acceptable precision.

Likewise, industrial processes have inherent variability:

- the input ore for smelting might have variable minerals concentration (both exploited and uninteresting–the gang); the tailings can still contain valuable mineral but hardly exploitable (such as for crushing: particulate matter of too low diameter is useless), but also erroneously rejected fragments [31];

- in Hanford (Washington, USA), nuclear waste stored in tanks may have co-contaminants, but with no certainty due to the poor records during the Cold War and the Manhattan Project, with other physicochemical transformations happening to form a nonhomogeneous mixture, which impairs disposal by blending into glass ([12]'s Section 5.6).

Moreover, the decisions might not be implementable to full precision: if the optimal decision is to have a molar concentration of $1.001 \cdot 10^{-18}\,\mathrm{mol/m^3}$ of tin chloride ($SnCl_2$), exactly $6.02214 \cdot 10^{23} \times 1.001 \cdot 10^{-18} = 602,816$ molecules must be present for a volume of one litre, for a weight of approximately $189.8 \cdot 10^{-18}$ grams (as indicated by Wolfram Alpha [`http://www.wolframalpha.com/input/?i=weight+of+1.001*10^-18+mole+of+SnCl2`]), which can be very difficult to ensure in an actual plant.

In unit-commitment problems (scheduling the electricity production), a standard constraint is to satisfy the power needs at each time step: the power produced by all power plants must be at least the (forecast) demand for electricity.

$$\sum_{i=1}^{N} P_i(t) \geq \text{demand}(t)\,. \tag{1.1}$$

The coefficients in the left-hand side of the inequality are certain: they characterise the structure of the problem. There is no difference in the quality of the power delivered by each plant: the left-hand-size coefficients are exact. However, the demand can only be forecast, and is therefore uncertain. If the planning produces exactly demand($t$) with no left capacity, but the needs are slightly larger, this scheduling becomes infeasible: there is not enough power available on the network to meet the demand.

Depending on the problem, these slight modifications in data may have dramatic impact on the infeasibility of the problem: in some pathological cases, a modification of 0.01% of the data may lead to an infeasibility of 210,000% of the perturbation, i.e. seven orders of magnitude larger than the perturbation (see [3]'s preface). This is clearly unacceptable in any real world application: slight perturbations are quite common, but such an infeasibility cannot be worked around.

The goal of optimising under uncertainty is to avoid such solutions: for "acceptable" deviance from the nominal values, the solution should be at least feasible, ideally close to optimality. Of course, this will have some repercussion on the cost of the solution, which should remain rather low. The precise meaning of these sentences depends on the way to take into account the uncertainty.

## 1.1 Deterministic optimisation

Usual optimisation theory focuses on problems of the form

$$\begin{array}{lllll} \min & f(\mathbf{x}) & & & \\ \text{s.t.} & h_i(\mathbf{x}) & = & 0, & \forall i, \\ & g_j(\mathbf{x}) & \leq & 0, & \forall j. \end{array} \tag{1.2}$$

$f(\mathbf{x})$ is the *objective function*, the criterion to decide whether a solution $\mathbf{x}$ is better than another (such as its cost). $h_i(\mathbf{x})$ and $g_j(\mathbf{x})$ indicate the *constraints*, which determine the solutions that are feasible (unit-commitment's power generation (1.1), for example).

### 1.1.1 Main structures

Depending on the structure of the problem and on the properties of the functions $f$, $h_i$, and $g_j$, those programs can be easy to solve. Few such structures will be used in this thesis; they mostly rely on the concept of convexity.

The major one is *linear programming* (LP), which imposes all functions to be linear.

$$\begin{array}{ll} \min & \mathbf{c^T}\,\mathbf{x} \\ \text{s.t.} & \mathbf{A}\,\mathbf{x} = \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}. \end{array} \tag{1.3}$$

LPs always have a convex objective function (which is also always concave). Inequalities can be turned into equalities by adding slack variables:

$$\mathbf{a^T x} \leq b \qquad \Longleftrightarrow \qquad \mathbf{a^T x} + s = b, \quad s \geq 0. \tag{1.4}$$

In some cases, *quadratic programming* (QP) will be useful: the objective is allowed to have quadratic terms, indicated by a matrix $\mathbf{Q}$. The constraints remain linear.

$$\begin{array}{ll} \min & \mathbf{c^T}\,\mathbf{x} + \frac{1}{2}\,\mathbf{x^T}\,\mathbf{Q}\,\mathbf{x} \\ \text{s.t.} & \mathbf{A}\,\mathbf{x} = \mathbf{b}. \end{array} \tag{1.5}$$

The most interesting case of QPs is when the matrix $\mathbf{Q}$ is positive definite: in this case, the objective is a convex function.

Those two structures allow efficient algorithms, which can solve such programs (when they are convex) to optimality in polynomial time. The simplex algorithm and interior-point methods are common algorithms for this. The former has exponential worst-case complexity, but is polynomial in most practical cases; depending on the problem to solve, interior-point methods can be more efficient.

A third structure uses quadratic constraints (instead of objective), and is thus called *quadratically constrained programming* (QCP) :

$$\begin{array}{ll} \min & \mathbf{c^T}\,\mathbf{x} \\ \text{s.t.} & \mathbf{a_i^T}\,\mathbf{x} + \mathbf{x^T}\,\mathbf{R_i}\,\mathbf{x} \leq b_i, \quad \forall i. \end{array} \tag{1.6}$$

The problem is convex if the matrices $\mathbf{R_i}$ are positive definite, in which case it can be solved efficiently by interior-point methods. If the objective is also quadratic (as in QPs), those problems are called *quadratically constrained quadratic programs* (QCQP).

### 1.1.2 Mixed integer programs

Discrete optimisation includes another kind of constraint, which imposes some variables $x_k$ to have integer values. It is often written as $\mathbf{x} \in X$, where $X$ is the domain for the integer variables. Those problems are called *mixed integer programs* (MIP), when both continuous and discrete variables are allowed. The resulting program is no more convex, due to the discreteness of some variables.

$$\begin{array}{lrcll} \min & f(\mathbf{x}) & & & \\ \text{s.t.} & h_i(\mathbf{x}) & = & 0, & \forall i, \\ & g_j(\mathbf{x}) & \leq & 0, & \forall j, \\ & \mathbf{x} & \in & X. & \end{array} \tag{1.7}$$

Adding those integrality constraints makes the problem $\mathcal{NP}$-hard: there is no *theoretically* efficient algorithm to solve this kind of problems. All algorithms have exponential worst-case complexity (as opposed to convex programs), but provide solutions to most practical problems in reasonable time (which is not necessarily true for convex programs). Common algorithms include *branch-and-bound* (B&B) and *branch-and-cut* (B&C).

Solving such problems are usually solved using existing libraries: state-of-the-art solvers are based on B&B and B&C, but have carefully thought implementations and heuristics to speed up the process. Benchmarks show that, for both LPs [22] and QPs [21], commercial solvers perform much better than open-source software: namely, CPLEX, Gurobi, and Xpress perform much better than would any (naïve or not so naïve) implementation of those algorithms. Figure 1.1 shows such a benchmark: in this setting, commercial software solver more than five times faster than the best open source libraries.

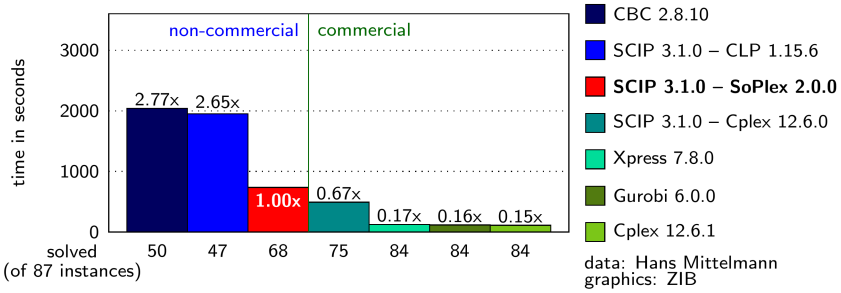


**Figure 1.1** – MIP solver benchmark (shifted geometric mean of results). Taken from SCIP's website [`http://scip.zib.de/`]; based on [22].

## 1.2 Optimisation under uncertainty

This section is based on [25], a tentative retrofitting of a general theory on both stochastic and robust programming.

Previous canonical programs have one common property: they do not take uncertainty into account. The solution to the optimisation process is implemented *before* any uncertainty can be observed. The general uncertain model mimics this procedure: first, decisions are taken; then, they are evaluated when the uncertainty is revealed. The optimisation is done by simulating the uncertainty. This formulation supposes some time discretisation, as the decisions are taken *before* being evaluated.

More formally, the state $\mathbf{S_t}$ can be defined as the information known at time step $t$: it contains both observable, physical state (state of machines, previous demands, previous failures, etc.) and the belief about unobservable parameters (such as the forthcoming demands). $\mathbf{W_t}$ models the information that is learnt between $t-1$ and $t$ (such as the realisation of actual demand): this part is stochastic. Based on that information, the goal of the optimisation problem

is to define the decisions $\mathbf{x_t}$, knowing the cost $C[\mathbf{S_t}, \mathbf{x_t}(\mathbf{S_t})]$ of these decisions: these decisions will influence the state at the next time step $\mathbf{S_{t+1}}$, including the scope of possible decisions. The objective function can thus be written as

$$\min \mathbb{E}_{\boldsymbol{\mathcal{W}}} \left\{ \sum_{t=0}^{T} C[\mathbf{S_t}, \mathbf{x_t}(\mathbf{S_t})] \right\}. \tag{1.8}$$

*Remark.* The expectation operator $\mathbb{E}$ can be replaced by any risk measure, such as the maximum value, some quantile, etc. Such operators could however lead to practical solving issues.

**Example.** The unit-commitment problem consists in deciding what means of production to use to meet some (unknown) electrical demand. These decisions to turn on or off some machine depend on time. This problem can be cast into this generic formalism: the state $\mathbf{S_t}$ indicates which machines are turned on and what the demand was at the previous time steps (before $t$); during each time step, the actual demand gets known, and thus forms $\mathbf{W_t}$. The decisions are about starting or stopping machines: the $\mathbf{x_t}$ directly influence the next state $\mathbf{S_{t+1}}$. The cost function $C$ is composed of starting, fixed, and variable costs, depending on the decisions (starting machines) and state (letting machines on, using them to produce power).

However, the actual probability density function is rarely useable as such: it would require to compute analytically the objective function through the definition of the expectation function. It must thus be approximated.
Stochastic programming is the most natural way to do it: scenarios are drawn from the probability distribution, associated with some probability; thus the "curse of dimensionality," as the deterministic model is written once per scenario. On the other hand, robust programming optimises over the worst case in a given uncertainty set: this does not cause a large increase in the size of the problem, but encoding the uncertainty set might change the structure (the ellipsoid is a common choice, and leads to quadratic constraints).

This approximation step implies that the result of either stochastic or robust programming or performance evaluation of a policy by either is not exact: it is "only" a simulation in a limited number of cases, not using the true (and inaccessible) probability density. Assessing the performance on a much larger number of scenarios than practical for stochastic programming would give a better approximation thereof; with this study, it can then be decided whether or not taking into account uncertainty when optimising is important.

## 1.3 Stochastic optimisation

The stochastic optimisation point of view over uncertainty is to use stochastic models, using the solid mathematical foundations behind probability. Historically, it is the first approach to uncertainty, developed by Dantzig in 1955 [11], only a few years after he first described the simplex algorithm (in 1947 [10]).

This section is inspired by [26].

### 1.3.1 One stage

A *one-stage stochastic program* conceptually imposes to take decisions $\mathbf{x}$ before they are implemented and evaluated on a scenario $\mathbf{W_s}$. All decisions must be implemented before any data from the future ($\mathbf{W_s}$) is known; then, all this information is revealed.

More precisely, a linear one-stage stochastic program has the general form

$$\min_{\mathbf{x}} \quad \mathbf{c^T}\,\mathbf{x} + \mathbb{E}_{\boldsymbol{\mathcal{W}},\boldsymbol{\mathcal{A}},\boldsymbol{\mathcal{B}}} \left\{ \begin{array}{cc} \min\limits_{\mathbf{x}} & \mathbf{w^T}\,\mathbf{x} \\ \text{s.t.} & \mathbf{A}\,\mathbf{x} \gtreqless \mathbf{b} \end{array} \right\} \quad \text{s.t.} \quad \mathbf{x} \geq \mathbf{0}. \tag{1.9}$$

The cost is due to two parts: one that is certain ($\mathbf{c^T}\,\mathbf{x}$), most often the implementation cost (such as starting a machine); the other depends on the uncertain part. The constraints may have uncertain elements, such as a demand.

This formulation has two properties: first of all, the solution is feasible for all possible values of the uncertain parameters; then, it is guaranteed to be optimal on average (when the optimisation is done without approximation on the probability density).

### 1.3.2 Two stages

A more complex family of stochastic programs has *two stages*: decisions can be taken at two instants. The first ones are implemented before any observation can be made for $\mathbf{W_s}$; then, it is observed until $t = t_r$, and the implementer can take some *recourse action* for the next time steps ($t > t_r$), based on these observations. Finally, the complete policy is evaluated based on the realisation of the unknown $\mathbf{W_s}$.

The main difference between one-stage and two-stage stochastic programs is that the decisions partly depend on observations: the implementer has the opportunity to adapt to what they observe during the first stage. As a consequence, a two-stage stochastic program may be less conservative than a one-stage formulation, thanks to the recourse.

A linear two-stage stochastic program has the general form

$$\begin{array}{ll} \min\limits_{\mathbf{x}} & \mathbf{c^T}\,\mathbf{x} + \mathbb{E}_{\boldsymbol{\mathcal{W}}}\{Q(\mathbf{x},\mathbf{W})\} \\ \text{s.t.} & \mathbf{A}\,\mathbf{x} = \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \end{array} \tag{1.10}$$

where $Q(\mathbf{x},\mathbf{W})$ is the optimal value of the second stage for the first-stage decisions $\mathbf{x}$:

$$\begin{array}{rcll} Q(\mathbf{x},\mathbf{W}) & = & \min\limits_{\mathbf{y}} & \mathbf{q^T}\,\mathbf{y} \\ & & \text{s.t.} & \mathbf{T}\,\mathbf{x} + \mathbf{U}\,\mathbf{y} = \mathbf{h}, \\ & & & \mathbf{y} \geq \mathbf{0}. \end{array} \tag{1.11}$$

All matrices in $Q(\mathbf{x},\mathbf{W})$ save $\mathbf{x}$ and $\mathbf{y}$ are random data contained in $\mathbf{W}$ (namely, $\mathbf{q}$, $\mathbf{T}$, $\mathbf{U}$, and $\mathbf{h}$).

This setting can be generalised to more than two stages to make *multistage stochastic programs*: some decision is taken, then some data is observed from a probability distribution, then some decision is taken, etc.

### 1.3.3 Chance constraint

Using the hypothesis of a known probability distribution, another kind of stochastic modelling can be done: *chance constraints*, i.e. constraints on the probability of some event (sometimes said *probabilistic constraints*). This can be used to impose less tight constraints than previously, in case of high uncertainty, but also when no recourse action is available to implement the decisions.

For example, instead of imposing a constraint $\mathbf{a}^\mathbf{T}\,\mathbf{x} \leq b$, where $\mathbf{a}$ and/or $b$ is a random variable, it could be more interesting to have the constraint

$$\mathbb{P}\big(\mathbf{a}^\mathbf{T}\,\mathbf{x} \leq b\big) \geq \eta. \tag{1.12}$$

This is a relaxation of the first constraint: instead of imposing the feasibility for all possible values of the random variables, the solution must be feasible "only" for their most likely values.

The implementation of this kind of constraints often relies on the analytic computation of the constraint, to give the *integrated chance constraint*, a deterministic equivalent of the constraint. For example, a linear constraint $\mathbf{a}^\mathbf{T}\,\mathbf{x} \geq b$ whose coefficients $\mathbf{a}$ are random variables (following a normal distribution whose mean is $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$) can be relaxed as a chance constraint like (1.12). According to [7]'s Theorem 3.18, its integrated form is

$$\overline{\mathbf{a}}^\mathbf{T}\,\mathbf{x} + \Phi^{-1}(\eta)\ \sqrt{\mathbf{x}^\mathbf{T}\,\boldsymbol{\Sigma}\,\mathbf{x}} \leq b, \tag{1.13}$$

where $\Phi$ is the cumulative density function of the standard normal distribution

$$\Phi(x) = \frac{1}{2}\left(1 + \operatorname{erf}\frac{x}{\sqrt{2}}\right). \tag{1.14}$$

Such integrated forms cannot always be obtained easily, and not all closed forms are convex. The normal distribution is special in that the constraint is actually a Lorentz cone. For other distributions, implementation rather uses a convex approximation of the constraint or sampling from the probability distribution.

## 1.4 Robust optimisation

This section is inspired by [3, 5, 16].

On the other hand, robust optimisation does not use probability distribution to account for uncertainty: it optimises in the worst case, when parameters belong to some *uncertainty set*. Often, this set is defined based on a probabilistic model and gathered data, but none of them is present when optimising.

The paradigm is to minimise the worst case (i.e. the maximum value of the objective when minimising), with uncertainties $\mathbf{u_c}$ and $\mathbf{u_i}$ contained in the uncertainty set $\mathcal{U}$:

$$\min_{\mathbf{x}} \left\{ \max_{(\mathbf{u_c},\mathbf{u_i})\in\mathcal{U}} \left\{ f(\mathbf{x},\mathbf{u_c}) \;\middle|\; f_i(\mathbf{x},\mathbf{u_i}) \leq 0 \right\} \right\}. \tag{1.15}$$

This program thus ensures feasibility for all uncertainties in the set $\mathcal{U}$, and optimality for the worst case. It does not explicitly take into account an average over possible values (unlike stochastic programming). The nested min-max problem can be easily rewritten as a standard minimisation by introducing an objective variable $t$:

$$\begin{array}{ll} \min\limits_{\mathbf{x}} & t \\ \text{s.t.} & f(\mathbf{x},\mathbf{u_c}) \leq t, \\ & f_i(\mathbf{x},\mathbf{u_i}) \leq 0, \\ & \forall\, (\mathbf{u_c},\mathbf{u_i}) \in \mathcal{U}. \end{array} \tag{1.16}$$

However, this program cannot be solved as is: it features infinitely many constraints.

Imposing some structure on the uncertainty set, it is possible to rewrite those constraints as a *robust counterpart* of the original problem: the maximisation step (sometimes called the *robust sub-problem*) is included inside the main problem. This rewriting chooses to impose the constraint in the worst case only.

*Remark.* Relaxing the worst case hypothesis in robust programming does not give good results: the uncertain variables $\mathbf{u_c}$ and $\mathbf{u_i}$ would be let free in the uncertainty set, so the solver freely chooses the value it gives them. In other words, forgetting the $\forall$ symbol in (1.15) or the maximisation in (1.15) optimises for the *best case* in the uncertainty set (which might be advantageous compared to the nominal problem).

This mistake is rather easy to make: it consists of implementing the uncertainty set directly into the program, forgetting to solve the robust sub-problem (i.e. rewriting parts of the constraints to take the uncertainty set into account); actually, the uncertain variables $\mathbf{u_c}$ and $\mathbf{u_i}$ should never appear in the robust models, unless there is a specific mechanism to ensure the worst case approach. For example, if $\mathbf{u_i}$ models demand uncertainty in a problem, the most likely value for $\mathbf{u_i}$ is to lower the actual demand as much as possible when this counterpart step is forgotten.

### 1.4.1 Soyster uncertainty

The first kind of uncertainty pursues the worst-case goal: *all* uncertain parameters take their most unfavourable value. This makes a very tight problem, as all sources of uncertainty are supposed to work together towards the worst case. Denoting $K_j$ the set of possible values for constraint vector $\mathbf{a_j}$, the resulting

(linear) program is

$$\begin{array}{ll} \min & \mathbf{c^T x} \\ \text{s.t.} & \overline{\mathbf{A}}\,\mathbf{x} \leq \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \\ & \overline{a}_{ij} = \sup\limits_{\mathbf{a_j} \in K_j} a_{ij}. \end{array} \tag{1.17}$$

However, this formulation can be solved as-is: it does not change the structure of the program, nor does it increases its dimensionality.

### 1.4.2 Ellipsoidal uncertainty

To have a more realistic and less conservative approach to robust programming, the uncertainty set can have another form: an ellipsoid. This shape was motivated by a normal distribution (as shown in Figure 1.2), as the set of points described by $\text{pdf}(\mathbf{x}) \geq \rho$ is an ellipsoid, as in Figure 1.3; the parameter $\rho$ defines the size of the ellipsoid.

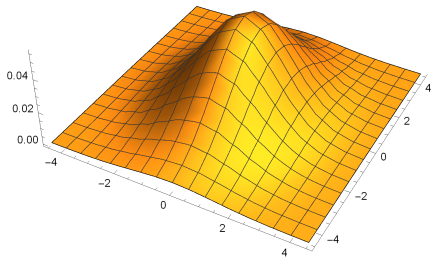


**Figure 1.2** – Joint probability density function $\text{pdf}_{\mathcal{X},\mathcal{Y}}$ of two random variables with Gaussian distribution.

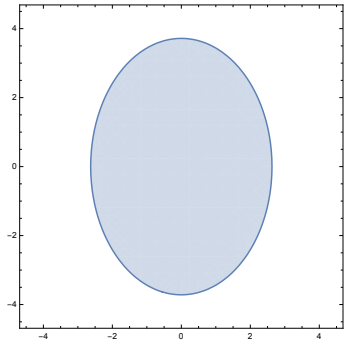


**Figure 1.3** – Ellipsoid obtained as $\text{pdf}_{\mathcal{X},\mathcal{Y}} \leq \rho$, where $\rho$ controls its size.

Formally, an ellipsoidal uncertainty set is defined as

$$\mathcal{U} = \left\{ (a_1, a_2 \ldots a_m) \;\middle|\; a_i = a_i^0 + \Delta_i\, u_i, \quad \|\mathbf{u}\|_2 \leq \rho \right\}. \tag{1.18}$$

$\mathbf{a_i^0}$ denotes the nominal value for the uncertain parameters $\mathbf{a_i}$. The parameters $\Delta_i$ allows weighting the uncertainty of the various components in the total uncertainty budget $\rho$: the higher the $\Delta_i$, the less uncertainty is allowed. The nested min-max formulation is:

$$\begin{array}{lll} \min\limits_{\mathbf{x}} & \max\limits_{\mathbf{u}} & \mathbf{c^T x} \\ & \text{s.t.} & \sum_j \left(a_{ij}^0 + \Delta_j\, u_j\right)^T x_j \leq b_i, \\ & & \|\mathbf{u}\|_2 \leq \rho, \\ & & \mathbf{x} \geq \mathbf{0}. \end{array} \tag{1.19}$$

This kind of uncertainty sets has one computational disadvantage when compared to the previous approach: when the nominal problem is linear, the equivalent program is no more linear, as encoding the uncertainty set requires a quadratic constraint (QCP). (Proof below.)

$$\begin{array}{ll} \min & \mathbf{c^T\, x} \\ \text{s.t.} & \mathbf{a_i^{0T}\, x} \leq b_i - \rho \left\| \Delta_i \, \mathbf{x} \right\|_2 , \\ & \mathbf{x} \geq \mathbf{0} \end{array} \tag{1.20}$$

The norm used in the ellipsoid ($L_2$) can vary: $L_\infty$ corresponds to Soyster uncertainty, and $L_1$ can yield linear programs. Those have very different shapes, as shown in Figure 1.4. As a consequence, the ellipsoid must be carefully chosen to fit the situation at hand.

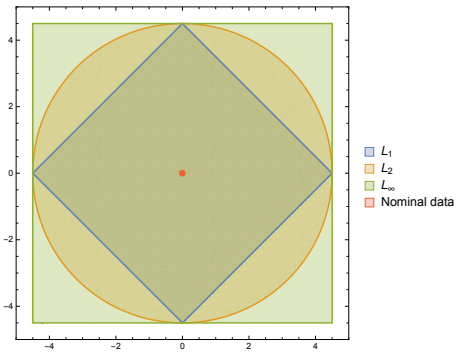


**Figure 1.4** – Comparison of different uncertainty sets centred on a point.

*Proof.* The robust program can be written as

$$\begin{array}{ll} \min & \mathbf{c^T\, x} \\ \text{s.t.} & \mathbf{a_i^T\, x} \leq b_i, \quad \forall \mathbf{a_i} \in \left\{ \mathbf{a_i^0} + \mathbf{Q^T\, u} \,\middle|\, \left\| \mathbf{u} \right\|_2 \leq \rho \right\}. \end{array} \tag{1.21}$$

where the matrix $\mathbf{Q^T}$ encodes the $\Delta_i$ coefficients. The expression of $\mathbf{a_i}$ can be injected into the constraint:

$$\begin{array}{ll} \min & \mathbf{c^T\, x} \\ \text{s.t.} & \left( \mathbf{a_i^0} + \mathbf{Q^T\, u} \right)^{\mathbf{T}} \mathbf{x} \leq b_i, \quad \forall \mathbf{u} \in \left\{ \mathbf{u} \,\middle|\, \left\| \mathbf{u} \right\|_2 \leq \rho \right\}. \end{array} \tag{1.22}$$

Not all values of $\mathbf{u}$ need to be considered: if the constraint is satisfied in the worst case, then it is satisfied for all values of $\mathbf{u}$. This transformation is equivalent to solving the robust sub-problem once and for all.

$$\max_{\left\| \mathbf{u} \right\|_2 \leq \rho} \left[ \mathbf{a_i^{0T}\, x} + \mathbf{u^T\, Q\, x} \right] = \left( \mathbf{a_i^0} \right)^{\mathbf{T}} \mathbf{x} + \max_{\left\| \mathbf{u} \right\|_2 \leq \rho} \mathbf{u^T\, Q\, x} \leq b_i. \tag{1.23}$$

No knowledge can be assumed on $\mathbf{x}$ or its relationships to $\mathbf{u}$: only standard algebra tools can be used to make progress. In particular, the Cauchy–Schwarz

inequality states that $\mathbf{u^T Q\, x} \leq \|\mathbf{u}\|\ \|\mathbf{Q\, x}\|$, for any vector norm $\|.\|$. The norm of $\mathbf{u}$ is bounded by $\rho$, so that the constraint becomes

$$\mathbf{a_i^{0T}\, x} + \rho\ \|\mathbf{Q\, x}\|_2 \leq b_i. \tag{1.24}$$

□

### 1.4.3 Deriving an uncertainty set

The general form of uncertainty sets is constrained by the parameters of the model that must be made uncertain. However, it leaves open the question of the size of the ellipsoid (the parameter $\rho$). An interesting approach uses statistics: exploit historical knowledge of the random process to determine it.

Using the concept of typical sets developed by Shannon in information theory, [2] showed that, assuming the $n$ random variables follow a Gaussian distribution of zero mean and correlation matrix $\mathbf{\Sigma}$, an uncertainty set can be given by

$$\mathcal{U} = \left\{ \mathbf{u} \,\middle|\, -\rho \leq \left\|\mathbf{\Sigma^{-1}\, u}\right\|_2^2 - n \leq \rho \right\}, \tag{1.25}$$

where $\rho$ is determined such that the probability of having a point inside this set (the "typical set probability") is $1 - \varepsilon$, chosen depending on the application and the needed level of robustness against uncertainty.

Simpler techniques can be used for $L_1$ or $L_\infty$ ellipsoids: based on the raw data, determine the smallest ellipsoid so that all points belong to it (or only some percentage). The algorithm is quite simple: first, determine the nominal value (usually, the mean); then, compute the needed size for each point to be in the set; the size of the ellipsoid is the maximum size.

### 1.4.4 Other uncertainty models

Other models of uncertainty are possible, with a full typology of models. Soyster's model (Section 1.4.1) can be called *column-wise uncertainty*: both the cost vector and the right-hand-side of the constraints are considered certain, each column of the constraint matrix (one column contains the coefficients of one variable for all constraints) evolves in an uncertainty set.

On the other hand, *row-wise uncertainty* (such as the ellipsoids of Section 1.4.2, sometimes said Ben-Tal and Nemirovski's model) considers the coefficients of variables inside a constraint belong to an uncertainty set. Another model of this family proposes to encode the uncertainty budget as a number of parameters allowed to take different values from their nominal value within some range (Bertsimas and Sim's model). Thus, the constraint $\mathbf{a_i^T\, x} \leq b_i$ becomes, with nominal values $\mathbf{a_i}$, maximum deviation $\mathbf{\hat{a}_i}$, uncertain variables set $S_i$, and uncertainty budget $\Gamma_i$:

$$\mathbf{a_i^T\, x} + \max_{\{S_i \in J_i : |S_i| = \Gamma_i\}} \sum_j \hat{a}_{ij}\, y_j \leq b_i, \qquad -1 \leq y_j \leq 1. \tag{1.26}$$

Another kind of robust programming can be useful: right-hand-side uncertainty, where the right-hand sides of the constraints are constrained in an uncertainty set (thus multiple constraints are linked together) [20].

Robust programming can also be used to give approximations of chance constraints.

## 1.5 Comparing results under uncertainty

One goal of this thesis is to actually compare those two approaches to take into account uncertainty when optimising decisions. The computer scientist point of view will first analyse the computational issues of those techniques: Which one is the fastest? How does it scale with problem size? How to improve time to solution?

On the other hand, the practitioner will prefer to analyse the behaviour of the solutions: Which one gives the most robust solution, is feasible in most situations, has the less variance on a given set of scenarios? How is the cost split up: the implementation cost and the operational cost? This evaluation must be done on a number of scenarios, using a simulation-like approach: for the answers to be accurate, it would be better to have more scenarios than the optimisation process had access to.

# Chapter 2

# Modelling Under Uncertainty

One major issue with optimisation under uncertainty is modelling the sources of uncertainty. Below are examples of such models, which will be further studied in Chapters 6 and 7.

## 2.1 Facility location

The *facility location* problem (FL) is concerned with the optimal placement of facilities (such as warehouses, stores, or hospitals) to minimise the cost of transportation to the clients, such as in Figure 2.1. (It is sometimes called *plant problem*.)

This problem has many variations: facilities can be opened in discrete and predetermined places or as points in a given zone; multiple goods can be delivered to different clients; the transportation can happen on a complete graph or in more complex settings; competitors can be taken into account by being far from them; hazardous materials can be stored away from houses; facilities can have a maximum capacity they can deliver. The variations are countless; some of them are $\mathcal{NP}$-complete.

This thesis will use a rather simple version: facility location with discrete capacitated facilities and a complete network, which can be turned into a MILP. The main source of uncertainty is the demand of the clients, which can be forecast but not exactly predicted. Failures of facilities could also be considered.

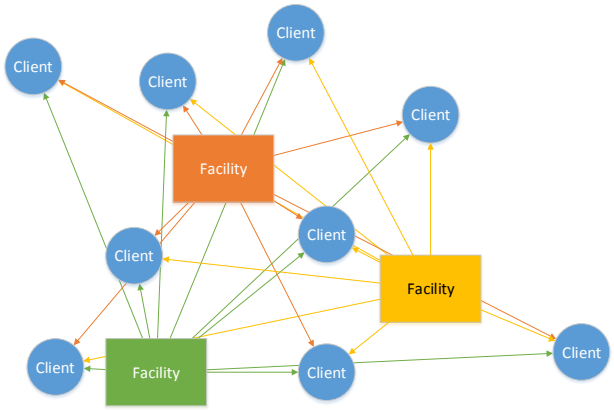


**Figure 2.1** – Facility location.

### 2.1.1 Deterministic model

The problem has one main set of decision variables $x_i$: whether the facility $i$ is open. The flow from facility $i$ to client $j$ will be denoted $y_{i\to j}$. Each client has some demand to satisfy ($d_j$), but each facility has a maximum capacity ($c_i$). Moving a product from facility $i$ to client $j$ has a cost $c_{i\to j}$ while opening a facility $i$ has a cost $f_i$; the variable production cost will be named $v_i$.

The objective may thus be written as

$$z = \underbrace{\sum_i f_i\, x_i}_{\text{opening}} + \underbrace{\sum_i \sum_j c_{i\to j}\, y_{i\to j}}_{\text{transporting}} + \underbrace{\sum_i v_i \left( \sum_j y_{i\to j} \right)}_{\text{producing}}. \tag{2.1}$$

The first constraint is that any facility cannot send products to a client if it is not open:

$$y_{i\to j} \leq M\, x_j, \qquad \forall i, \forall j. \tag{2.2}$$

The constant $M$ must be chosen large enough for uncapacitated models; in this case, the flow cannot be higher than the capacity of the facility, $c_i$, in the case it has only one client. The implemented model is uncapacitated, and the constant $M$ corresponds to the maximum a facility might produce, which is the total demand $\sum_j d_j$.
The second constraint is that each client must receive their demand.

$$\sum_i y_{i\to j} \geq d_j, \qquad \forall j. \tag{2.3}$$

Those two constraints are enough to formally define the facility location problem; other constraints can be added to strengthen the formulation.

The flow from any plant cannot be higher than the demand of the client $d_j$.

$$y_{i\to j} \leq d_j, \qquad \forall i, \forall j. \tag{2.4}$$

The total production of all opened facilities must be greater than the demand. (This constraint is very useful for the Benders' decomposition in Section 4.2: if the $y_{i\to j}$ variables are omitted from the model, the solution in the $\mathbf{x}$ variables is still feasible.)

$$\sum_i c_i\, x_i \geq \sum_j d_j. \tag{2.5}$$

The complete model is thus the following:

$$\begin{array}{lrcll} \min & \sum_i f_i\, x_i + \sum_i \sum_j c_{i\to j}\, y_{i\to j} + \sum_i v_i \left(\sum_j y_{i\to j}\right) & & & \\ \text{s.t.} & y_{i\to j} & \leq & M\, x_j, & \forall i, \forall j, \\ & \sum_i y_{i\to j} & \geq & d_j, & \forall j, \\ & y_{i\to j} & \leq & d_j, & \forall i, \forall j, \\ & \sum_i c_i\, x_i & \geq & \sum_j d_j, & \\ & x_i & \in & \mathcal{B}, & \forall i, \\ & y_{i\to j} & \in & \mathbb{R}, & \forall i, \forall j. \end{array} \tag{2.6}$$

### 2.1.2 Stochastic model

The stochastic approach to modelling the demand uncertainty is to have a probability density function of demands. It can be sampled to get scenarios associated with some probabilities. Whatever the scenario is, the facilities to open (i.e. to build and maintain) must remain the same. However, the actual flows from facilities to clients are decided by the clients themselves, once the facilities are built ("once the solution is implemented").

The deterministic equivalent thus has variables shared across scenarios (whether a facility must be built), and others which must depend on the scenario (the actual flows). Constraints having at least one scenario-dependent variable will have to be duplicated once per scenario; the others can be kept.

Denoting by a $s$ subscript the variables and constants that will depend on the scenario, the stochastic model thus looks like

$$\begin{array}{lrcll} \min & \sum_i f_i\, x_i + \sum_s \sum_i \sum_j c_{i\to j}\, y^{(s)}_{i\to j} + \sum_s \sum_i v_i \left(\sum_j y^{(s)}_{i\to j}\right) & & & \\ \text{s.t.} & y^{(s)}_{i\to j} & \leq & M\, x_j, & \forall i, \forall j, \forall s, \\ & \sum_i y^{(s)}_{i\to j} & \geq & d^{(s)}_j, & \forall i, \forall j, \forall s, \\ & y^{(s)}_{i\to j} & \leq & d^{(s)}_j, & \forall i, \forall j, \forall s, \\ & \sum_i c_i\, x_i & \geq & \max_s \left\{\sum_j d^{(s)}_j\right\}, & \\ & x_i & \in & \mathcal{B}, & \forall i, \\ & y^{(s)}_{i\to j} & \in & \mathbb{R}, & \forall i, \forall j, \forall s. \end{array} \tag{2.7}$$

If there are $n$ facilities and $m$ clients, the deterministic model has $n$ binary variables, $n \times m$ continuous variables, and $3 \times m \times n + 1$ constraints. For $s$

scenarios, this stochastic model will still have $n$ binary variables, but $s \times n \times m$ continuous variables, and $3 \times m \times n \times s + 1$ constraints. This indicates the stochastic problems will be of higher dimension, and will probably take longer to solve (but only for the LP relaxations, as the number of binary variables does not increase).

### 2.1.3 Robust model

On the other hand, the robust model will decide the demands belong to some uncertainty set $\mathcal{U}$. The abstract version of the problem thus looks like this (without the strengthening constraints):

$$\begin{array}{lccll} \min & \sum_i f_i\, x_i + \sum_i \sum_j c_{i\to j}\, y_{i\to j} + \sum_i v_i \left(\sum_j y_{i\to j}\right) & & & \\ \text{s.t.} & y_{i\to j} & \leq & M\, x_j, & \forall i, \forall j, \\ & \sum_i y_{i\to j} & \geq & d_j, & \forall i, \forall j, \quad \forall \mathbf{d} \in \mathcal{U}, \\ & x_i & \in & \mathcal{B}, & \forall i, \\ & y_{i\to j} & \in & \mathbb{R}, & \forall i, \forall j. \end{array} \tag{2.8}$$

This uncertainty set is centred around a nominal demand, with some uncertainty bounded by an ellipsoid whose norm is $L_n$ and size $\rho$:

$$\mathcal{U} = \left\{ \mathbf{d} + \mathbf{u} \;\middle|\; \|\mathbf{u}\|_n \leq \rho \right\}. \tag{2.9}$$

It is worth noticing that increasing the demand always makes the problem more difficult.

The equivalent model for a $L_\infty$-norm only uses the worst case: the actual demand is increased by $\rho$ at each time step with respect to the nominal data, as the norm constraint becomes $\|\mathbf{u}\|_\infty = \max_i |u_i| \leq \rho$.

$$\begin{array}{lccll} \min & \sum_i f_i\, x_i + \sum_i \sum_j c_{i\to j}\, y_{i\to j} + \sum_i v_i \left(\sum_j y_{i\to j}\right) & & & \\ \text{s.t.} & y_{i\to j} & \leq & M\, x_j, & \forall i, \forall j, \\ & \sum_i y_{i\to j} & \geq & d_j + \rho, & \forall i, \forall j, \\ & x_i & \in & \mathcal{B}, & \forall i, \\ & y_{i\to j} & \in & \mathbb{R}, & \forall i, \forall j. \end{array} \tag{2.10}$$

As the uncertain coefficients are present in the right-hand side of the constraints, and they must evolve simultaneously, writing explicitly the robust counterpart for other norms is not as easy as the reasoning of Section 1.4.2.

## 2.2 Unit-commitment

The unit-commitment is much more involved than the facility location: it is concerned with the control of a set of power plants in order to meet some electricity demand. More precisely, it contains two parts: the *unit-commitment* itself, which decides what units to use; then the *economic dispatch*, which computes the optimal use of the started units [33].

Many constraints on those parts can be implemented, mainly for modelling the mechanics of the generators:

- a power plant has a maximum power (its nominal value), but may also have a minimum power to ensure its stability;
- for thermal units, it may require some time to properly start or shutdown;
- it may have ramping constraints (slow-start machine): a plant cannot produce its maximum power just after being started (it may need some time to reach the right temperature, e.g.);
- it may be forced to remain on or off for some time, which avoids the solver to choose solutions where a unit is used only for a few time steps before being shut down (this would cause excessive wear and tear for the machine);
- etc.

Other types of plants can be introduced in the model, such as hydroelectric with water level management and wildlife preservation constraints. Other types of constraints can be implemented to model an electric network, with power flow equations modelling the fact that a line cannot carry an infinite current.

Also, some legal constraints can be imposed onto the producer, such as a spinning reserve: the choice of machines must be such that it can produce a given percentage more than the demand expectation, mainly to deal with uncertainty in the demand and with machine failures.

These constraints imply that the unit-commitment and economic dispatch have different behaviours when facing uncertainty: if the demand suddenly and unforeseeably increases, it is not possible to quickly start another generator (in general: some gas turbines can start in 30 minutes [1]); however, the actual power they can produce can be adapted more easily in real time.

This thesis will focus on thermal units, requiring some time to start or shutdown, forced minimum uptime and downtime, and a range of admitted values, which makes sense for an industrial plant. The uncertainty will be first set on the demand, then power plant failures will be considered.

*Remark.* If a machine takes no time to start or to shutdown, studying the effect of uncertainty is much less interesting: a machine could then be started instantaneously, exactly when it is needed, with no actual forecast (the same holds for minimum uptime and downtime). As a consequence, initial conditions must exist: knowing what was the state of a machine before the optimisation takes place is needed for those constraints to be correctly imposed.

### 2.2.1 Deterministic model

The main decision variables of the problem are to decide whether a machine is on, $\text{on}_{\text{machine}}(t)$. This thus requires some time discretisation. The other

important variables are the produced power by each plant at each time step, $\text{power}_{\text{machine}}(t)$.

To ease formulation, other derived variables will be useful: $\text{off}_{\text{machine}}(t)$ is the opposite of $\text{on}_{\text{machine}}(t)$, $\text{start}_{\text{machine}}(t)$ will indicate whether the machine is started at time $t$ (goes from off to on state), $\text{stop}_{\text{machine}}(t)$ if it stops (goes from on to off).

The objective is to minimise the total cost, which is due to starting the machines, to keeping them on, and to the actual production.

$$\begin{array}{lll} \min & \sum_m \sum_t \text{startCost}_m \, \text{start}_m(t) & \\ + & \sum_m \sum_t \text{fixedCost}_m \, \text{on}_m(t) & \\ + & \sum_m \sum_t \text{linearCost}_m \, \text{power}_m(t) \, . & \end{array} \tag{2.11}$$

The first constraint imposes that the power production at each time steps is larger than the demand.

$$\sum_m \text{power}_m(t) \geq \text{load}(t) \, , \qquad \forall t. \tag{2.12}$$

A plant can only produce some power if it is on. In that case, its contribution will be bounded by a minimum and a maximum.

$$\text{power}_m(t) \leq \text{maxPower}_m(t) \; \text{on}_m(t) \, , \qquad \forall m, \forall t. \tag{2.13}$$

$$\text{power}_m(t) \geq \text{minPower}_m(t) \; \text{on}_m(t) \, , \qquad \forall m, \forall t. \tag{2.14}$$

$\text{off}_m(t)$ is defined as the complement of $\text{on}_m(t)$.

$$\text{off}_m(t) = 1 - \text{on}_m(t) \, , \qquad \forall m, \forall t. \tag{2.15}$$

A machine starts if it was off at the previous time step and then becomes on (at the first time step it is fully operational). It thus corresponds to a logical AND: $\text{start}(t) = \text{off}(t-1) \times \text{on}(t + \text{timeStart})$. This expression can be linearised, as a product of two binary variables:

$$\text{start}_m(t) \leq \text{off}_m(t-1) \, , \qquad \forall m, \forall t. \tag{2.16}$$

$$\text{start}_m(t) \leq \text{on}_m(t + \text{timeStart}_m) \, , \qquad \forall m, \forall t. \tag{2.17}$$

$$\text{start}_m(t) \geq \text{off}_m(t-1) + \text{on}_m(t + \text{timeStart}_m) - 1, \qquad \forall m, \forall t. \tag{2.18}$$

The same reasoning holds for stopping a plant.

$$\text{stop}_m(t) \leq \text{on}_m(t-1) \, , \qquad \forall m, \forall t. \tag{2.19}$$

$$\text{stop}_m(t) \leq \text{off}_m(t + \text{timeStop}_m) \, , \qquad \forall m, \forall t. \tag{2.20}$$

$$\text{stop}_m(t) \geq \text{on}_m(t-1) + \text{off}_m(t + \text{timeStop}_m) - 1, \qquad \forall m, \forall t. \tag{2.21}$$

If a machine is started in $t$, then it must be on (producing electricity) as soon as it is completely operational (which may happen a few time steps after the order

to start it), then remain on for the minimum uptime. This constraint can be written as a logic formula: $\text{start}(t) \Rightarrow \bigwedge_{i=0}^{\text{minUp}-1} \text{on}(t + \text{timeStart} + i)$. In linear inequalities, it might become

$$\sum_{i=0}^{\text{minUp}_m - 1} \text{on}(t + \text{timeStart}_m + i) \geq \text{minUp}_m \, \text{start}_m(t)\,, \qquad \forall t, \forall m. \tag{2.22}$$

Also, it cannot produce any power between the moment it is started and the moment it is fully operational, meaning that $\text{start}(t) \Rightarrow \bigwedge_{i=0}^{\text{timeStart}-1} \text{off}(t + i)$.

$$\text{off}_m(t + i) \geq \text{start}_m(t)\,, \qquad \forall m, \forall t, \forall i \in [0, \text{timeStart}_m)\,. \tag{2.23}$$

These two constraints are also valid when stopping a machine, *mutatis mutandis*.

$$\sum_{i=0}^{\text{minDown}_m - 1} \text{off}(t + \text{timeStop}_m + i) \geq \text{minDown}_m \, \text{stop}_m(t)\,, \qquad \forall t, \forall m. \tag{2.24}$$

$$\text{on}_m(t + i) \geq \text{stop}_m(t)\,, \qquad \forall m, \forall t, \forall i \in [0, \text{timeStop}_m)\,. \tag{2.25}$$

Other constraints can be added to strengthen the formulation; the ones that were added are based on [23], whose goal was to reach the convex hull for the minimum uptime and downtime; more constraints are imposed in this model, meaning that the complete formulation is not ensured to be a convex hull.
A machine cannot start and stop at the same time.

$$\text{start}_m(t) + \text{stop}_m(t) \leq 1, \qquad \forall m, \forall t. \tag{2.26}$$

If there is a difference in the variable on, it must be caused by the machine starting or stopping some time steps before.

$$\begin{aligned} \text{on}_m(t) - \text{on}_m(t-1) &= \text{start}_m(t - \text{timeStart}_m) \\ &- \text{stop}_m(t - \text{timeStop}_m)\,, \qquad \forall m, \forall t. \end{aligned}$$

A machine must be on if it was started if it was started timeStart time steps before, or minUp before that instant.

$$\sum_{t'=t-\text{minUp}_m-\text{timeStart}_m}^{t-\text{timeStart}_m} \text{start}_m(t') \leq \text{on}_m(t)\,, \qquad \forall m, \forall t. \tag{2.27}$$

The same holds for the opposite state with the necessary modifications.

$$\sum_{t'=t-\text{minDown}_m-\text{timeStop}_m}^{t-\text{timeStop}_m} \text{stop}_m(t') \leq \text{off}_m(t)\,, \qquad \forall m, \forall t. \tag{2.28}$$

### 2.2.2 Stochastic two-stage model

To take into account the demand uncertainty (only in constraint (2.12)), stochastic programming proposes to use scenarios. In practice, the unit-commitment would be solved at time $t$ to get a planning for the next few time steps. Ideally, the problem is solved very frequently, at a frequency matching the discretisation (usually, fifteen minutes). As a consequence, the operator has some recourse action every time step: they may decide what to do next (start a machine, stop another one, etc.).

This indicates a recourse action: the program will optimise with very little uncertainty for one time step (the forecast the electricity demand fifteen minutes ahead can be considered exact), then there will be a recourse action to decide what actions to take afterwards. To take decisions without hindering optimality of future ones, the model must consider the uncertainty after the recourse action, using a set of scenarios. After this recourse, the actions will depend on the demand scenario: those decisions will not be implemented as such, they allow the solution before the recourse action to plan for future load. For example, this will avoid shutting down a power plant with a high starting cost right now if it is very likely to be used afterwards, even if it is useless right now.

The actual model may be written as a series of deterministic models, each indexed with the scenario, for all time steps. Then, the decisions before the recourse $t_r$ are imposed to be equal among all scenarios (*nonanticipativity constraints*).

$$\mathrm{on}_m^{(s)}(t) = \mathrm{on}_m^{(1)}(t)\,, \qquad \forall m, \forall t < t_r, \forall s \neq 1. \tag{2.29}$$

$$\mathrm{power}_m^{(s)}(t) = \mathrm{power}_m^{(1)}(t)\,, \qquad \forall m, \forall t < t_r, \forall s \neq 1. \tag{2.30}$$

More complex implementations avoid creating all those variables, and have no scenario index before the recourse. However, these are much harder to write.

### 2.2.3 Robust model

The other view on demand uncertainty is robust programming. There will be no recourse action as such, as no scenarios are used to model the uncertainty: these are replaced by ellipsoids. Before $t_r$, the demand is known exactly; after this moment, this hypothesis no more holds, the demand is now inside an uncertainty set. Denoting $\mathrm{load}^{(0)}(t)$ the nominal value for the demand, the right-hand side of (2.12) must take this uncertainty into account:

$$\sum_m \mathrm{power}_m(t) \geq \mathrm{load}^{(0)}(t)\,, \qquad \forall t < t_r, \tag{2.31}$$

$$\sum_m \mathrm{power}_m(t) \geq \mathrm{load}^{(0)}(t) + u(t)\,, \qquad \forall t \geq t_r, \tag{2.32}$$

These uncertainty "variables" are constrained into an uncertainty set of some norm ($L_1$, $L_2$, or $L_\infty$) and of size $\rho$:

$$\|\mathbf{u}\|_n \leq \rho. \tag{2.33}$$

The objective (2.11) takes the worst case inside this uncertainty set:

$$\begin{array}{lll} \min\limits_{\mathbf{on},\mathbf{start}\ldots} & \max\limits_{\|\mathbf{u}\|_n \leq \rho} & \sum_m \sum_t \text{startCost}_m \, \text{start}_m(t) \\ & + & \sum_m \sum_t \text{fixedCost}_m \, \text{on}_m(t) \\ & + & \sum_m \sum_t \text{linearCost}_m \, \text{power}_m(t) \, . \end{array} \tag{2.34}$$

As before, the robust counterpart can only be written explicitly for $L_\infty$: in that case, all components of $\mathbf{u}$ are equal to the size of the ellipsoid $\rho$ (as the worst case in this set is to have a demand increased by the maximum amount at every time step).

### 2.2.4 Failures

The above developments only take into account uncertainty in the demand. However, actual unit-commitment is plagued by machine failures: they may break unexpectedly during operation, or be down due to planned maintenance. In both cases, the machines cannot be used for electricity production for a certain number of time periods.

Modelling a planned maintenance is rather easy: it is sufficient to set its produced power to zero, i.e. $\text{power}(t) = 0$. (A more realistic approach would be to set $\text{on}(t) = 0$; however, this kind of modelling is too tight, and leads to high-cost solutions or to infeasibilities in the present model.)

On the other hand, unplanned failures can be modelled in different ways. The most intuitive is probably stochastic: failures are considered as (low-probability) scenarios, where the produced power is sometimes set to zero for some machines.

Instead of setting the power to zero, another solution is to consider some equivalent demand [19], which is akin to robust programming: the nominal load $\text{load}^{(0)}(t)$ is increased by the maximum power the power plant that is failing (or, to have a less conservative approach, its actual production). If scenarios are not used for implementation, the exact size and structure of the program are kept.

Both approaches (scenarios with zero power; no scenarios but equivalent demand) are not necessarily tied to either stochastic or robust modelling for demand uncertainty: all combinations are possible. Mixing both mindsets to model different sources of uncertainty gives more flexibility to the modelling process, to balance conservativeness and model size for solving performance.

# Chapter 3

# Solving Deterministic Problems

Deterministic problems are well-studied: efficient algorithms have been developed, and state-of-the-art implementations can solve very large instances. The precise value of "very large" depends on the structure of the problem, but is usually quite high for linear problems (millions of variables and constraints).

*Remark.* This section only provides brief explanations about how to solve (mixed integer) linear programs. More details can be found in classical textbooks (both general and dedicated), such as [17] for linear programming and [32] for mixed integer linear programming.

## 3.1 Simplex algorithm

The simplex algorithm is used to solve linear programs in *standard form*, which means minimising a linear objective subject to equality constraints.

$$\begin{array}{ll} \min & \mathbf{c^T x} \\ \text{s.t.} & \mathbf{A x} = \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}. \end{array} \tag{3.1}$$

Transforming inequalities into such equalities imposes to add slack variables: as a consequence, there are many more variables than constraints in most linear programs.

Its main idea is to impose the value of some variables to zero so that finding the solution boils down to solving a linear system. Such zero variables are said to be *outside the basis*, while those obtained by solving the linear system are *basic*. To find the best solution to the program, the algorithm has to decide what variables must be zero.

An important property of the algorithm is that it always finds the optimal solution (up to machine precision); also, when it has solved a problem, it can

often solve a slightly modified version of this problem very efficiently (in a few iterations), i.e. by adding new variables or new constraints, or by changing some coefficients in $\mathbf{A}$, $\mathbf{b}$, or $\mathbf{c}$, through the use of duality. However, this algorithm cannot take into account integrality constraints.

## 3.2 Branch-and-bound

Branch-and-bound is a technique that builds upon the simplex algorithm to support integrality constraint. The main idea is to *relax* integrality constraints: this gives a standard linear program, which can then be solved by the simplex algorithm.

The solution is unlikely to have integer values for all integer variables: it *branches* on a variable $x_i$, i.e. creates a tree like Figure 3.2 which explores $x_i \leq \left\lfloor x_i^{(j)} \right\rfloor$ on the one hand, and $x_i \geq \left\lceil x_i^{(j)} \right\rceil$ on the other; this process reduces the feasible area of the relaxation, as shown in Figure 3.1. For a binary variable, it is equivalent to having a branch $x_i = 0$ and another $x_i = 1$. By this process, it tries to increase the number of variables that have integer values.

To prove optimality, it uses the notion of *gap*, by maintaining both a lower bound and an upper bound on the objective value: when those two bounds are equal, the problem is solved to optimality. The lower bound is obtained by the relaxation: this step removes constraints, so the objective value can only decrease (for a minimisation problem); the upper bound is given by feasible integer solutions.

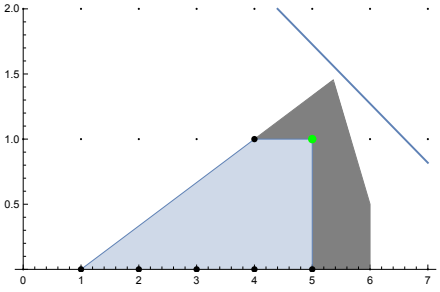


**Figure 3.1** – Feasible area in branch-and-bound after two branching steps.

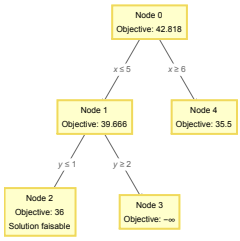


**Figure 3.2** – Branch-and-bound search tree.

## 3.3 Cutting planes

Another technique to solve discrete programs also uses the simplex method in an iterative process: when the solution provided by the simplex algorithm is not integer, *cutting planes* are added to the problem. Their goal is to cut out those fractional points, and thus approaching the convex hull of all feasible solutions.

The basic idea is to rewrite constraints such as $x \leq c$, where $x$ should be integer, as

$$x \leq \lfloor c \rfloor . \tag{3.2}$$

This will cut out non-integer-feasible values in $(\lfloor c \rfloor , c]$. More generally, the Chvátal-Gomory procedure generates from a constraint $\mathbf{a_i^T}\,\mathbf{x} \leq b_i$ inequalities of the form

$$\sum_{j=1}^{n} \lfloor u\, a_{i,j} \rfloor \; x_j \leq \lfloor u\, b_i \rfloor , \tag{3.3}$$

where $u$ is a constant in $\mathbb{R}_0$ and $\mathbf{x}$ a vector of integer variables.

This inequality can be generalised into the *mixed-integer rounding inequalities* (MIR) when some variables are continuous. Other kinds of cuts can be generated through dedicated algorithms: the Chvátal-Gomory will generate all valid inequalities, and thus give the exact convex hull, but after a large amount of time; when the cuts use the structure of the constraints, they can be more efficient.

## 3.4 Duality theory

In mathematical programming, *duality* is a tool used to view a problem from two perspectives, called the *primal* and *dual* problems. Going from one to the other is a rewriting of the problem using some principles. The usual program is the primal; its dual provides a lower bound to the objective (weak duality) or has the same objective value (strong duality).

The most common duality is Lagrangian: the constraints are added into the objective using some *dual multipliers* to penalise them. When maximising the Lagrangian along the dual variables and minimising it along the primal variables, the obtained solution is optimal: the constraints have a large (detrimental) contribution to the objective, but it is still minimum, meaning that all constraints are satisfied (to lower the penalisation). The generic problem (1.2) is associated to the dual problem

$$\min_{\mathbf{x}} \max_{\substack{\boldsymbol{\lambda} \geq \mathbf{0} \\ \boldsymbol{\mu} \in \mathbb{R}^n}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = f(\mathbf{x}) + \sum_j \mu_j\, g_j(\mathbf{x}) + \sum_i \lambda_i\, h_i(\mathbf{x}) . \tag{3.4}$$

This can be solved iteratively: first choose values for $\boldsymbol{\lambda}$ and $\boldsymbol{\mu}$, then minimise $\mathbf{x}$ with those dual variables, then compute the new dual variables with that $\mathbf{x}$, etc., until convergence.

When only equality constraints are present in the problem, the *augmented Lagrangian* adds a quadratic term to penalise the constraints:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \nu) = f(\mathbf{x}) + \sum_i \lambda_i\, h_i(\mathbf{x}) + \nu \sum_i h_i^2(\mathbf{x}) . \tag{3.5}$$

With respect to the standard Lagrangian, when solving with an iterative process, this formulation will tend to make fewer iterations.

In the linear case, to avoid unboundedness of the dual, constraints must be added; the dual problem can then be written as a transposition of the primal problem: the objective coefficients become the right-hand side of the constraints, and the constraint matrix is transposed. Also, the objective sense is reversed.

$$\begin{array}{lll} \min & \mathbf{c^T x} & \\ \text{s.t.} & A_i^T x = b & i \in M_3, \\ & x_j \geq 0 & j \in N_1. \end{array} \qquad \begin{array}{lll} \max & \mathbf{p^T b} & \\ \text{s.t.} & p_i \in \mathbb{R} & i \in M_3, \\ & p^T A_j \leq c_j & j \in N_1. \end{array} \tag{3.6}$$

## 3.5 Software

Actual implementations of MILP software uses a combination of branch-and-bound and cutting planes (which is sometimes called *branch-and-cut*) with heuristics. They can also be based on other LP solvers than the simplex algorithm, either primal or dual (mainly, interior point methods). With good pruning strategies for the branch-and-bound tree, it can be very efficient.

State-of-the-art (commercial) solvers are much faster than their free counterparts [22]. Examples of such codes are CPLEX, Gurobi, and XpressMP. They often support more than MILP, such as MIQP.

Those solvers usually have a low-level API, which imposes building the constraint matrix. Many practitioners prefer using a modelling tool, that allows them to think in terms of variables and constraints rather than lines and columns of a matrix. Many solvers include a higher-level API (CPLEX Concert, Gurobi, XpressMP, Mosek Fusion, etc.), but domain-specific languages are also common (AMPL, Xpress Mosel, AIMMS, ZIMPL, etc.), with solver-agnostic APIs (Google or-tools, OscaR, Coin-OR Rehearse, Coopr, etc.).

# Chapter 4

# Solving Stochastic Problems

The algorithms outlined in Chapter 3 only work on deterministic and finite problems: they cannot have expectation or probability operators (stochastic programming), or an infinite number of constraints (robust programming).

In order to solve one- or two-staged stochastic programs, the only approach is forming the so-called *deterministic equivalent*: the expectation operator is discretised using a number of scenarios (a set of pairs of a probability and values for the uncertain parameters approximating the whole probability distribution), then write the whole stochastic program *in extenso* for those scenarios, thus getting rid of the stochasticity of the formulation.

This technique works very well until the problems are too large or too complex: it is equivalent to writing multiple times the complete nominal (deterministic) problem, once per scenario, and solving the whole program at once. It may take very large amounts of CPU time and memory, this is why standard decomposition techniques (used previously to solve very large deterministic problems) can be used; otherwise, heuristics can be used to approximate the deterministic equivalent.

This thesis will explore one method of each kind: the Benders' decomposition and the progressive hedging heuristic. Their main objective is not always to solve problems faster: it is first and foremost to find a solution with reasonable resources. The first one, Benders' decomposition, essentially works with a main problem containing the first stage, with sub-problems solved to retrieve information about the second-stage behaviour of the first-stage solution; progressive hedging tries to bring together the solution to each scenario by penalisation.

## 4.1 Deterministic equivalent

The first solution technique is the most natural: the *deterministic equivalent* writes the full set of scenarios in one large (deterministic) program.

### 4.1.1 One-stage stochastic program

The generic one-stage stochastic program is given by (1.9):

$$\min_{\mathbf{x}} \quad \mathbf{c^T}\,\mathbf{x} + \mathbb{E}_{\boldsymbol{\mathcal{W}},\boldsymbol{\mathcal{A}},\boldsymbol{\mathcal{B}}}\left\{ \begin{array}{cc} \min\limits_{\mathbf{x}} & \mathbf{w^T}\,\mathbf{x} \\ \text{s.t.} & \mathbf{A}\,\mathbf{x} \gtreqless \mathbf{b} \end{array} \right\} \quad \text{s.t.} \quad \mathbf{x} \geq \mathbf{0}. \tag{4.1}$$

The parameters $\mathbf{w}$ and $\mathbf{A}$ of the sub-problem are given by a joint probability distribution $\text{pdf}_{\boldsymbol{\mathcal{W}},\boldsymbol{\mathcal{A}}}$. Sampling it gives a series of scenarios, each of them being a triple of a probability $p_s$, and values for the parameters: $\mathbf{W_s}$, $\mathbf{w_s}$, and $\mathbf{b_s}$. As the variables $\mathbf{x}$ are shared by all scenarios (there is no recourse: these decisions must be taken before any uncertainty is known), the deterministic equivalent becomes

$$\begin{array}{cc} \min\limits_{\mathbf{x}} & \mathbf{c^T}\,\mathbf{x} + \sum_s p_s\,\mathbf{w_s^T}\,\mathbf{x} \\ \text{s.t.} & \mathbf{A_s}\,\mathbf{x} \gtreqless \mathbf{b_s}, \forall s, \\ & \mathbf{x} \geq \mathbf{0}. \end{array} \tag{4.2}$$

### 4.1.2 Two-stage stochastic program

A more interesting case is the two-stage stochastic program ((1.10) and (1.11)):

$$\begin{array}{cc} \min\limits_{\mathbf{x}} & \mathbf{c^T}\,\mathbf{x} + \mathbb{E}_{\boldsymbol{\mathcal{W}}}\{Q(\mathbf{x},\mathbf{W})\} \\ \text{s.t.} & \mathbf{A}\,\mathbf{x} = \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \end{array} \quad \text{where} \quad Q(\mathbf{x},\mathbf{W}) = \begin{array}{cc} \min\limits_{\mathbf{y}} & \mathbf{q^T}\,\mathbf{y} \\ \text{s.t.} & \mathbf{T}\,\mathbf{x} + \mathbf{U}\,\mathbf{y} = \mathbf{h}, \\ & \mathbf{y} \geq \mathbf{0}. \end{array} \tag{4.3}$$

Here, the decisions of the sub-problem (1.11) can depend on the actual realisation of the random variable $\boldsymbol{\mathcal{W}}$: this is called a *recourse action*. When forming the deterministic equivalent, the variables $\mathbf{y}$ thus have to depend on the scenario. The decisions for a scenario $s$ are denoted $\mathbf{y^{(s)}}$.

The deterministic equivalent is then obtained by discretising the expectation operator, and by letting the second-stage decisions depend on the scenario:

$$\begin{array}{cc} \min\limits_{\mathbf{x}} & \mathbf{c^T}\,\mathbf{x} + \sum_s p_s\,\mathbf{q_s^T}\,\mathbf{y^{(s)}} \\ \text{s.t.} & \mathbf{A}\,\mathbf{x} = \mathbf{b}, \\ & \mathbf{T_s}\,\mathbf{x} + \mathbf{U_s}\,\mathbf{y^{(s)}} = \mathbf{h_s}, \\ & \mathbf{x} \geq \mathbf{0}, \\ & \mathbf{y^{(s)}} \geq \mathbf{0}, \forall s. \end{array} \tag{4.4}$$

The two-stage stochastic program can also be seen as a series of realisation, where *all* decisions depend on the scenario (both $\mathbf{x}$ and $\mathbf{y}$); then, noticing that the first-stage decisions *cannot* depend on data that will be revealed later on, at $t = t_r$, *nonanticipativity constraints* must be imposed: all first-stage decisions

$\mathbf{x^{(s)}}$ must be equal. The situation is depicted in Figure 4.1.

$$\begin{array}{ll} \min\limits_{\mathbf{x}} & \mathbf{c^T}\,\mathbf{x^{(s)}} + \sum_s p_s\,\mathbf{q_s^T}\,\mathbf{y^{(s)}} \\ \text{s.t.} & \mathbf{A}\,\mathbf{x^{(s)}} = \mathbf{b}, \\ & \mathbf{T_s}\,\mathbf{x^{(s)}} + \mathbf{U_s}\,\mathbf{y^{(s)}} = \mathbf{h_s}, \\ & \boxed{\mathbf{x^{(s)}} = \mathbf{x^{(1)}}, \forall s \neq 1,} \\ & \mathbf{x^{(s)}} \geq \mathbf{0}, \forall s, \\ & \mathbf{y^{(s)}} \geq \mathbf{0}, \forall s. \end{array} \tag{4.5}$$

This formulation has the advantage that the only coupling constraint between scenarios is the nonanticipativity one (the previous formulation always mixes both).

*Remark.* Those formulations can accommodate more than two stages: the variables for the third stage will depend on the realisation of the uncertainty at $t = t_{r_1}$ *and* at $t = t_{r_2}$, which implies two indices for the scenario. A two-stage stochastic program already tends to be quite large; with more stages, covering the same time span but with more recourse (thus more flexibility, less conservativeness), the equivalent program will be even larger.

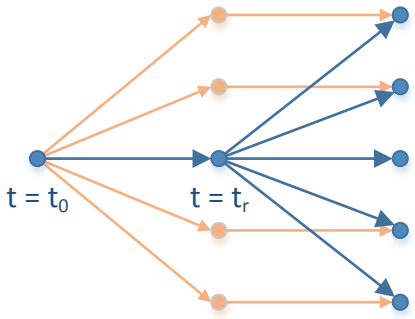


**Figure 4.1** – Structure of a two-stage stochastic program: variables do not depend on scenario until $t = t_r$, then decisions may diverge (dark). With nonanticipativity constraints, all variables are allowed to depend on the scenario: the bifurcation happens at $t = 0$ rather than $t = t_r$ (light).

## 4.2 Benders' decomposition

*Benders' decomposition* is a decomposition technique that was developed to solve large linear programs, in particular mixed-integer programs. It can use the block structure of the constraints of the deterministic equivalent (4.4) (as shown in Figure 4.2): for each scenario in the second stage, the constraints link

the first-stage variables $\mathbf{x}$ and the second-stage variables of the corresponding scenarios (thus only a subset of all $\mathbf{y}$ variables).

This algorithm separates the problem in two: the *master problem* only contains the first stage of the stochastic program, while the *sub-problem* deals with the second stage. Each iteration first solves the master problem to get the first-stage decisions $\mathbf{x}$; then, these are tested in the sub-problem; information from this sub-problem is carried to the master problem until convergence via cutting planes. Three cases can then happen at the sub-problem: either the first-stage is infeasible (generate a feasibility cut), or this solution is not optimal (generate an optimality cut), or it is optimal (stop).

*Remark.* In the case of stochastic programming, Benders' decomposition is also called *L-shaped method*, due to the shape of the constraint matrix looking like a L (Figure 4.2). It can also be generalised to more than two stages (nested Benders' decomposition, applying iteratively standard Benders' decomposition) and to nonlinear programs (generalised Benders' decomposition). Benders' decomposition is sometimes used to solve hard integer problems, as sub-problems can be solved using tools based on logic, more suitable to some kinds of integer problems (logic-based Benders' decomposition).

Benders' decomposition generates new constraints, and can thus be called a *row-generation algorithm*. Its dual, the Dantzig-Wolfe decomposition, works by adding new variables (column-generation). However, for stochastic programs, Benders' decomposition is much more frequently encountered.

Solving repeatedly very similar problems (here, variations are only new constraints) can be done efficiently for linear programs, as the simplex algorithm may be hot started (as hinted in Section 3.1). This decomposition may then be implemented rather efficiently for those problems.

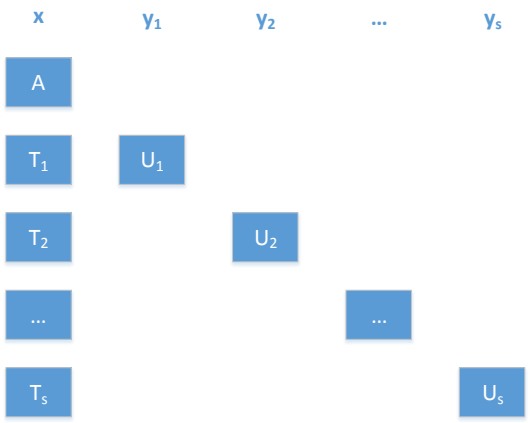


**Figure 4.2** – Structure of the constraint matrix in (4.4).

### 4.2.1 Formal derivation

This section is inspired by [27, 24].

To ease the derivation of Benders' cuts, it is easier to rewrite the full deterministic equivalent (4.4) as

$$\begin{array}{ll} \min & \mathbf{c^T}\,\mathbf{x} + \mathbf{q^T}\,\mathbf{y} \\ \text{s.t.} & \mathbf{A}\,\mathbf{x} + \mathbf{B}\,\mathbf{y} = \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \quad \mathbf{y} > \mathbf{0}, \end{array} \tag{4.6}$$

aggregating all second-stage decisions as $\mathbf{y}$, and all constraints as the single $\mathbf{A}\,\mathbf{x} + \mathbf{B}\,\mathbf{y} = \mathbf{b}$. This problem can be subdivided in two parts: one having only $\mathbf{x}$ as variables, the other taking $\mathbf{x}$ as constant.

$$\begin{array}{ll} \min_{\mathbf{x}} & \mathbf{c^T}\,\mathbf{x} + v(\mathbf{x}) \\ \text{s.t.} & \mathbf{x} \geq \mathbf{0}, \end{array} \qquad v(\mathbf{x}) = \begin{array}{ll} \min_{\mathbf{y}} & \mathbf{q^T}\,\mathbf{y} \\ \text{s.t.} & \mathbf{B}\,\mathbf{y} = \mathbf{b} - \mathbf{A}\,\mathbf{x}, \\ & \mathbf{y} \geq \mathbf{0}. \end{array} \tag{4.7}$$

The sub-problem $v(\mathbf{x})$ is linear; the right-hand side of its constraints, $\mathbf{b} - \mathbf{A}\,\mathbf{x}$, is constant.

This sub-problem's solution has implications on the master: if $v(\mathbf{x})$ is unbounded for any valid $\mathbf{x}$, the master problem is unbounded too, as for (4.4). Its dual is:

$$v(\mathbf{x}) = \begin{array}{ll} \max_{\mathbf{p}} & \mathbf{p^T}\,(\mathbf{b} - \mathbf{A}\,\mathbf{x}) \\ \text{s.t.} & \mathbf{B^T}\,\mathbf{p} \leq \mathbf{q}. \end{array} \tag{4.8}$$

Thanks to the dualisation, another important property appears: the feasible region of $v(\mathbf{x})$ does not depend on $\mathbf{x}$ (only its optimal solution). As a consequence, it is possible to reformulate it using only its extreme points $\left(\mathbf{p^1_{ep}}, \mathbf{p^2_{ep}} \dots \mathbf{p^P_{ep}}\right)$ and extreme rays $\left(\mathbf{p^1_{er}}, \mathbf{p^2_{er}} \dots \mathbf{p^R_{er}}\right)$, with one variable $v$, by imposing boundedness and optimality (using respectively the extreme rays and the extreme points).

A solution $\mathbf{x}$ gives an unbounded value if there is an extreme ray $j$ such that $\left[\mathbf{p^j_{er}}\right]^{\mathbf{T}}\,(\mathbf{b} - \mathbf{A}\,\mathbf{x}) > 0$; each extreme point gives an approximation $\left[\mathbf{p^i_{ep}}\right]^{\mathbf{T}}\,(\mathbf{b} - \mathbf{A}\,\mathbf{x})$ of the objective value, the maximum of all these values is the actual maximum. As a consequence, $v(\mathbf{x})$ can be written as the minimisation of the variable $v$, the bound on the objective:

$$v(\mathbf{x}) = \begin{array}{ll} \min_{v} & v \\ \text{s.t.} & \left[\mathbf{p^j_{er}}\right]^{\mathbf{T}}\,(\mathbf{b} - \mathbf{A}\,\mathbf{x}) \leq 0, \quad \forall j, \\ & \left[\mathbf{p^i_{ep}}\right]^{\mathbf{T}}\,(\mathbf{b} - \mathbf{A}\,\mathbf{x}) \leq v, \quad \forall i. \end{array} \tag{4.9}$$

As such, this reformulation is of no use: it requires to compute *all* extreme points and rays, i.e. all intersections of two constraints (there are exponentially many extreme points)—which the simplex algorithm only visits in the worst case. Also, it has a very high number of constraints: it is not clear solving this problem could be any fast. However, this problem can be directly added into

the master problem.

$$\begin{array}{ll} \min_{\mathbf{x}} & \mathbf{c^T}\,\mathbf{x} + v \\ \text{s.t.} & \mathbf{x} \geq \mathbf{0}, \\ & \left[\mathbf{p^j_{er}}\right]^{\mathbf{T}} (\mathbf{b} - \mathbf{A}\,\mathbf{x}) \leq 0, \quad \forall j, \\ & \left[\mathbf{p^i_{ep}}\right]^{\mathbf{T}} (\mathbf{b} - \mathbf{A}\,\mathbf{x}) \leq v, \quad \forall i. \end{array} \tag{4.10}$$

The last trick for Benders' decomposition is that those many constraints can be added iteratively: it starts from a small subset of those (to ensure boundedness of $v$), solving a relaxed master problem. The solution $(\mathbf{x}^*, v^*)$ is then used to continue:

- if $v^* = v(\mathbf{x}^*)$, then adding constraints will not bring any improvement, the process stops (the algorithm has converged);
- if $v(\mathbf{x}^*)$ is unbounded, $\mathbf{x}^*$ is infeasible for the second-stage, a *feasibility cut* $\left[\mathbf{p^j_{er}}\right]^{\mathbf{T}} (\mathbf{b} - \mathbf{A}\,\mathbf{x}^*) \leq 0$ is added to the master;
- if $v^* < v(\mathbf{x}^*)$, an *optimality cut* $\left[\mathbf{p^i_{ep}}\right]^{\mathbf{T}} (\mathbf{b} - \mathbf{A}\,\mathbf{x}) \leq v$ is added.

The solution to $v(\mathbf{x})$ can be computed using the dual form (4.8).

Since there is a finite number of extreme points and extreme rays, as each cutting plane cuts out the current solution, this procedure will at most explore a finite number of solutions. However, its worst-case complexity is exponential in the number of constraints of (4.4).

### 4.2.2 Multicut Benders' decomposition

The previous derivation was done by aggregating all second-stage variables and constraints: it cannot take full advantage of the nature of two-stage stochastic programs, as the scenarios are completely decoupled. *Multicut Benders' decomposition* proposes to solve one such sub-problem per scenario, as they do not depend on each other.

The main difference is that, at each iteration, *multiple* cutting planes are derived from the second-stage, one per scenario, with one variable $v$ per scenario. As a consequence, the master problem grows faster, but hoping that this will bring convergence faster. This algorithm was presented in [8], showing improvement over single-cut Benders' decomposition.

### 4.2.3 Discussion

Benders' decomposition is quite commonly implemented to solve stochastic programs, and has quite a lot of literature.

Its efficiency depends a lot on the structure of the problem to solve: if there are many constraints, the dual sub-problem will have many variables, and will often be slower to solve than a less constrained version of the program; it may

be beneficial to remove strengthening constraints in this case. Experiments are performed in Section 6.2.

Also, the generated cuts are not always strong: they may bring very little information to the master problem, leading to a very large number of added constraints before reaching an optimal solution. There has been some research on strengthening Benders cuts, as in [30] for the facility location problem.

Other solutions to improve running times include removing previous Benders constraints when they are dominated by new ones (the new planes can be close to parallel to existing ones, which increases the condition number of the constraint matrix, thus the numerical errors), and specifically for multicut Benders decomposition, having multiple scenarios per sub-problem (bundling); both were implemented in [8] to get good results. Regularisation terms can also be added ([7], Section 5.2). The effectiveness of these methods strongly depends on the application.

Performance-wise, in multicut Benders' decomposition, the scenarios can then be solved in parallel (e.g., on different machines, using much less memory on each machine than the full deterministic equivalent); however, the master problem may become a bottleneck—even more if it takes longer and longer to solve.

## 4.3 Progressive hedging

This section is inspired by [29].

On the other hand, *progressive hedging* is a heuristic that solves the different scenarios separately, then tries to bring them to a common solution using penalisation to impose the nonanticipativity constraints. This thus decomposes the stochastic program so that all scenarios are completely independent, with some synchronisation after each iteration. Albeit just a heuristic, it is able to provide high-quality solution ([9] reports objective with 1-2% delta to optimal).

The penalisation terms are similar to augmented Lagrangian techniques (Section 3.4) to impose nonanticipativity constraints. They are written using scenario-dependent weights (one per variable and per scenario), updated at each iteration, with a step length $\rho$.

While Benders' decomposition works by decomposing along the stages (the main problem contains only the first stage, then the sub-problems the second stage), and can thus be called *vertical*, progressive hedging does the decomposition along the other axis, the scenarios (*horizontal* strategy).

### 4.3.1 Algorithm statement

The basic principles of this algorithm are the following.

1. Solve each scenario $s$ independently, as a deterministic problem.

$$\mathbf{x_0^{(s)}} = \arg\min_{\mathbf{x}\in X_s} f_s(\mathbf{x})\,. \tag{4.11}$$

$$\mathbf{w_0^{(s)}} = 0. \tag{4.12}$$

2. Average the solutions to each scenario to yield an estimate of the solution.

$$\overline{\mathbf{x}}_{\mathbf{k-1}} = \sum_s P_s \, \mathbf{x_{k-1}^{(s)}}. \tag{4.13}$$

3. Solve each scenario $s$, as in 1, but with a penalisation term to impose the nonanticipativity constraints with respect to this average.

$$\mathbf{w_k^{(s)}} = \mathbf{w_{k-1}^{(s)}} + \rho \left[ \mathbf{x_{k-1}^{(s)}} - \overline{\mathbf{x}}_{\mathbf{k-1}} \right]. \tag{4.14}$$

$$\mathbf{x_k^{(s)}} = \arg\min_{\mathbf{x} \in X_s} f_s(\mathbf{x}) + \mathbf{w_k^{(s)}} \cdot \mathbf{x} + \frac{\rho}{2} \left\| \mathbf{x} - \overline{\mathbf{x}}_{\mathbf{k-1}} \right\|_2. \tag{4.15}$$

4. If the new solutions are close enough to each other (the *heterogeneity* is low enough e.g.), the algorithm has converged; otherwise, it performs one new iteration starting at 2.

$$\text{heterogeneity} = \sum_s P_s \left\| \mathbf{x_k^{(s)}} - \overline{\mathbf{x}}_{\mathbf{k}} \right\|_2. \tag{4.16}$$

As a heuristic, progressive hedging does not guarantee convergence, nor does it guarantee to give integer solutions for mixed-integer problems: actually, it might have an oscillatory behaviour between a few integer solutions. It is thus important to stop even if the heuristic has not yet converged.

*Remark.* Progressive hedging provably converges for convex problems. However, integer programs cannot be convex by definition, due to their discrete nature.

### 4.3.2 Possible improvements

This section is inspired by [28].

#### 4.3.2.1 Choice of $\rho$

As such, the algorithm may show very poor performance: it may take a long time before converging, and thus be impractical, especially for mixed-integer programs. One very important parameter to tune is the value of $\rho$: the heuristic may converge fast to a less-than-optimal solution or slowly to a better solution... or anything in between.

Heuristics have been proposed to make a better choice of $\rho$: some studies showed that values between 50 and 100 were needed for good convergence, while others chose values below 1. Actually, this parameter should be chosen so that the penalisation term has a real impact on the total objective function: it must be of the same order of magnitude than the deterministic part of the objective. One heuristic for this has been introduced in [28], and proposes to use one value

of $\rho$ per variable. For the integer variable $x$, the corresponding value $\rho_x$ is given by

$$\rho_x = \frac{c_x}{x_{\min} - x_{\max} + 1}, \tag{4.17}$$

where $c_x$ is its coefficient in the linear objective, $x_{\min}$ its minimum value in the current solutions, and $x_{\max}$ the maximum:

$$x_{\min} = \min_s x_k^{(s)}, \qquad x_{\max} = \max_s x_k^{(s)}. \tag{4.18}$$

This heuristic is parameter-less (and thus does not require parameter tuning for good behaviour), but does not depend on the problem: more effective techniques may be devised using knowledge about the situation.

#### 4.3.2.2 Cyclic behaviour

Another point of interest is the cyclic behaviour of the algorithm for some mixed-integer instances: the heuristic follows a cycle of solutions while trying to improve the homogeneity, but fails to escape this cycle as the internal status of the algorithm does not evolve. In these cases, the best solution is to stop the algorithm. The most simple technique is to check whether the averaged solution does not take a previously seen value; however, the weights might still evolve: it is thus best to check whether the weights are not too close to previous values. When only integer variables are considered, checking the solution was not seen before is simply an equality check in a set of solutions; however, as soon as continuous quantities are considered (which is the case for the weights), more complex data structures are needed.

#### 4.3.2.3 Convergence acceleration

In some cases, the averaged value for a variable sees only subtle changes from iteration to iteration: in this case, its value can be (heuristically!) imposed, so that it is no more considered for progressive hedging. This decreases the solution time for the sub-problems, as they have fewer variables to consider, but may decrease solution quality.

If only a few variables remain to be brought together, the algorithm may spend many iterations to ensure convergence (usually, the first iterations dramatically decrease the heterogeneity, while the following ones are fine-tuning the solution): the deterministic equivalent can then be used for those few variables. As many variables were removed from the problem, this solution technique becomes interesting, even for very large problems.

Another solution to bring convergence faster is to bundle multiple scenarios in one sub-problem, creating a small deterministic equivalent. The choice of scenarios to bundle is important to performance, as they may create more difficult sub-problems.

### 4.3.3 Discussion

Each iteration brings the solution to the different scenarios together: this does not imply all scenarios will have the same solution when the algorithm has converged; performing one more iteration only brings more confidence that the solution respects all constraints and is close to optimality, but with no guarantee.

While Benders' decomposition is an exact algorithm and requires heavy problem reformulation, the progressive hedging heuristic is much simpler to implement: it is a sequence of the deterministic problems with some penalisation. It can even be simpler to implement than a deterministic equivalent of the stochastic program.

Albeit being a heuristic, exact information can be retrieved from its execution: at any iteration, [14] proposes a method to compute lower bounds on the objective function, even in the mixed-integer case. Doing so "*requires approximately the same effort as executing a standard iteration of* [the progressive hedging algorithm]." This allows the users to have a better idea of the quality of the solution the heuristic has, exactly like standard branch-and-bound's gap. This technique generalises in the case of scenario bundling and for multistage programs, and it also depends on the choice of $\rho$.

The progressive hedging algorithm can be implemented in parallel, as the different sub-problems are completely independent from each other. It cannot spread the computations on more machines than scenarios by itself; more parallelism can only be achieved by adapting the underlying solver code. With sufficient parallelism, this heuristic is tractable: it only requires the deterministic problems to be tractable in order to have tractable PH iterations; the only problem is the number of iterations, on which no bound can be computed. This is especially useful when advanced machinery has been developed to solve deterministic instances (like unit-commitment).

## 4.4 Software

Some optimisation software packages include stochastic extensions, either as solvers or modelling environments using deterministic solvers. Most often, they work using the deterministic equivalent or Benders' decomposition, such as AIMMS, DECIS, FortSP, LINDO, MSLiP, Quantego QUASAR. Others include progressive hedging, such as Coin-OR PySP.

Some of them also propose other solution techniques, such as scenario reduction: instead of solving the deterministic equivalent for all scenarios, write it only for a subset of scenarios, chosen to be representative of the whole set of scenarios.

# Chapter 5

# Solving Robust Problems

Robust programming is usually more difficult to implement than stochastic programming: except in easy cases like the one presented in the introduction (Section 1.4.2), writing the deterministic equivalent of a robust constraint is not straightforward; moreover, this is tedious work for the modeller.

To work around this issue, two approaches are developed in the literature: one of them is reformulation, i.e. automatically rewriting the constraints using the uncertainty set; the other works iteratively, adding new cutting planes to ensure the worst case is achieved. Both of them are successful, depending on the structure of the problem at hand; hybrid algorithms can be even more powerful.

This chapter is inspired by [6].

## 5.1 Cutting plane

The idea behind cutting-planes implementation of robust programming is very similar to that of Benders' decomposition in Section 4.2: a master problem is solved with no uncertainty, then sub-problems are solved to take this uncertainty into account, by adding new constraints to the master problem. Those new constraints ensure the worst case is achieved, by solving the robust sub-problem (defined in Section 1.4).

For a robust problem with a given number of uncertain constraints $\mathbf{a_i^T x} \geq b_i$, the uncertain coefficients being constrained in an ellipsoid $\mathcal{U}_i$ centred on $\bar{\mathbf{a}}_\mathbf{i}$, the algorithm is as follows:

1. The master problem is solved, with the uncertain coefficients $\mathbf{a_i}$ set to their nominal value $\bar{\mathbf{a}}_\mathbf{i}$ (i.e. the constraint $\mathbf{a_i^T x} \geq b_i$ becomes $\bar{\mathbf{a}}_\mathbf{i}^\mathbf{T} \mathbf{x} \geq b_i$). The optimal solution is called $\mathbf{x}^*$.

2. For each uncertain row $i$, solve the robust sub-problem to find the worst-case values of the coefficients $\tilde{\mathbf{a}}_\mathbf{i}$ for this specific solution $\mathbf{x}^*$:

$$\tilde{\mathbf{a}}_\mathbf{i} = \underset{\tilde{\mathbf{a}}_\mathbf{i} \in \mathcal{U}_i}{\arg\max}\ \mathbf{a_i^T x}^*. \tag{5.1}$$

If the solution violates the uncertain constraint with these parameters (i.e. if $\tilde{\mathbf{a}}_\mathbf{i}^\mathbf{T}\,\mathbf{x} < b_i$, with appropriate tolerance), it is added to the master problem.

3. If no constraint was added at step 2, $\mathbf{x}^*$ is the optimal solution to the robust problem; otherwise, the new master problem is solved, starting again at step 1.

As a consequence, this algorithm adds at most one constraint per uncertainty set per iteration: it only does so if the constraint cuts the current solution. It ensures the worst case not based on the objective value, but on individual constraints violation.

The structure of the master problem is the same as that of the deterministic program (without uncertainty being taken into account): it only gets larger, with new constraints of the same structure as before. However, the robust subproblem depends on the structure of the uncertainty set. As a consequence, solving a robust linear program with usual $L_2$ ellipsoid accounts to many linear programs (larger and larger), but also many QCPs (with fewer variables).

Also, convergence guarantees depend on the uncertainty sets: if it is polyhedral, the simplex algorithm can only end up in an extreme point, so that a finite number of solutions of constraints can be added for each uncertainty set; as a consequence, the algorithm must converge (even though the master problem may become intractably large). For $L_2$ ellipsoids, there is no such theoretical property: convergence is not guaranteed; this does not, however, impede practical use.

*Remark.* This algorithm only works for continuous problems; it needs some refinements to work inside a branch-and-bound tree, as explained in [6]. It can also be implemented to remove the hypothesis of row-wise ellipsoids.

## 5.2 Reformulation

On the other hand, reformulation works before the optimisation process: it does not necessarily involve solving a sequence of programs, as it rewrites the robust program as a deterministic one. These techniques thus depend heavily on the structure of the uncertainty set.

A general result can be obtained if the uncertainty set can be represented as a cone (such as the positive orthant, the Lorentz cone, or the semi-definite cone) $K$. Denoting by $K_*$ its dual cone, a generic robust constraint can be written

$$\left(\mathbf{A}\,\mathbf{w} + \mathbf{b}\right)^\mathbf{T}\,\mathbf{x} + \left(\mathbf{c}^\mathbf{T}\,\mathbf{w} + d\right) \leq 0, \qquad \forall\,\mathbf{R}\,\mathbf{w} - \mathbf{r} \in K_*. \tag{5.2}$$

Equivalently, the constraint can be written as a sum of the uncertain part and a term independent from $\mathbf{w}$: $\left(\mathbf{A}^\mathbf{T}\,\mathbf{x} + \mathbf{c}\right)^\mathbf{T}\,\mathbf{w} + \left(\mathbf{b}^\mathbf{T}\,\mathbf{x} + d\right)$. This constraint will hold if its maximum is still below zero; hence, seeing it as an optimisation problem, duality theory proposes to add the dual variables $\mathbf{z}$ such that imposing

(5.2) is equivalent to the following set of constraints:

$$\mathbf{z} \in K, \qquad \mathbf{R}^\mathbf{T}\,\mathbf{z} = \mathbf{A}^\mathbf{T}\,\mathbf{x} + \mathbf{c}, \qquad \mathbf{r}^\mathbf{T}\,\mathbf{z} + \left(\mathbf{b}^\mathbf{T}\,\mathbf{x} + d\right) \leq 0. \tag{5.3}$$

(The proof details are in [4], Section 2.1.)

Other kinds of reformulations are possible, and are presented in [18]. They have different complexity results, in terms of generated variables (such as $\mathbf{z}$ above) and constraints (listed in (5.3)).

## 5.3 Software

Some mathematical programming software have support for robust programming. For example, YALMIP implements reformulation techniques (AIMMS seems to do the same), while JuMPeR hybridises both. ROME, on the other hand, is based on (probabilistic) distributional reformulation, using statistical moments for modelling (see [15] for details).

Overall, there is less software developed for robust programming than for stochastic programming. For example, the commercial, dedicated stochastic solver FortSP was announced in 2006, while the first public beta of the research project ROME was out in 2009. What is more, little software currently supports both stochastic programming and robust programming, such as the black-box commercial tools AIMMS and FrontLine Risk Solver.

# Chapter 6

# Comparing Performance: Facility Location

The facility location introduced in Section 2.1 has the advantage of being a very simple model, so that implementation of models and algorithms is rather easy. On the other hand, it is harder to find meaningful instances to have a more realistic comparison of the actual solutions. As a consequence, comparison focuses on performance.

## 6.1 Implementation details

Two main models are implemented: a stochastic model with demand scenarios (of which the deterministic case is a special case with only one scenario), and a robust model with $L_\infty$ ellipsoid. Multiple algorithms are implemented to solve the stochastic model: deterministic equivalent, progressive hedging, and Benders' decomposition. No algorithm is parallelised; only the underlying solver could use multiple threads.

A comparison point is given by a deterministic model on the average scenario.

### 6.1.1 Stochastic programming

The progressive hedging implementation has not only the basic algorithm, but also a cycle detection (Section 4.3.2.2), as it often exhibited such behaviours in early tests. The value of the $\rho$ parameter is fixed to fifty, as this value gave the best quality solutions and very reasonable solution times: the objective value is within a few percent compared to the deterministic equivalent solution's objective, while remaining tractable to compute.
Higher values of $\rho$ brought faster convergence (the algorithm took less time), but worse solutions; lower values tended to spend many iterations to give better solutions, but after a much longer time.

Two versions of Benders' decomposition were implemented: the standard decomposition, with all scenarios aggregated in the sub-problem (Section 4.2); the multicut decomposition, with one scenario per sub-problem (Section 4.2.2). The code is able to use the strengthened version of the model (including constraints (2.4) and (2.5)); however, they are not used for the benchmarks, as they performed poorly compared to the standard model: this is easily explained by the dualisation step of Benders' decomposition, where more variables are added for each primal constraint, and thus increase the solution time.

### 6.1.2 Robust programming

Only one robust model was implemented: an $L_\infty$ ellipsoid. Its centre is given by the average scenario. The size is computed as the minimum value such that all scenarios fit inside the ellipsoid, as explained in Section (1.4.3).

The actual robust model is then equivalent to imposing the worst case in this ellipsoid: for demands, it corresponds to the highest demand in the ellipsoid. This model is then equivalent to solving a deterministic problem

### 6.1.3 Implementation languages

Those algorithms were implemented twice. The first implementation is done using directly CPLEX through its Java API, which represents a rather standard way of working with industrial mathematical programs. The second one is done in Julia, a quite recent scientific programming language similar to MATLAB, with JuMP, the standard modelling domain-specific language for mathematical programming in Julia. The performance between those languages can then be compared.

The Julia community is fairly active in the domain of optimisation under JuliaOpt's umbrella, featuring many wrappers to standard solvers using a standard interface MathProgBase: at the beginning of this thesis, this list included Clp, Cbc, CPLEX, GLPK, Gurobi, and Mosek; it now also includes Bonmin, Couenne, ECOS, Ipopt, KNITRO, NLopt, and SCS. Those solvers are supported for many problem structures: (MI)LP, (MI)SOCP, SDP, (MI)NLP, depending on the solvers' capabilities.

The main advantage of Julia is that it provides performance similar to that of compiled languages like C, C++, or Fortran, while still being a dynamic language comparable to Python or MATLAB. In particular, JuMP exploits Julia's metaprogramming to provide performance on par with dedicated modelling languages like AMPL or with direct calls to the solver—usually, modelling layers build a full internal representation of the model before sending it to the solver, which JuMP avoids, like AMPL.

Models expressed with JuMP are also very easy to read, they are close to what would be done in AMPL or to their mathematical formulation:

```
m = Model()
# Add variables.
```

```
@defVar(m, build[1:factories], Bin)
@defVar(m, shipment[1:factories, 1:clients] >= 0)
# Add constraints.
for factory in 1:factories
  @addConstraint(m,
    sum(shipment[factory, :]) <= supply[factory] * build[factory])
end
for client in 1:clients
  @addConstraint(m, sum(shipment[:, client]) == demand[client])
end
```

The same model using directly the solver's API is much more cumbersome to create, for example in Java:

```
IloCplex model = new IloCplex();
IloIntVar[] build;
IloNumVar[][] shipment;
// Add variables.
IloIntVar[] build = model.boolVarArray(factories);
IloNumVar[][] shipment = new IloNumVar[factories][];
for (int factory = 0; factory < factories; ++ factory) {
  shipment[factory] = model.numVarArray(clients, 0, Double.MAX_VALUE);
}
// Add constraints.
for (int factory = 0; factory < factories; ++ factory) {
  model.addLe(model.sum(shipment[factory]),
              model.prod(supply[factory], build[factory]));
}
for (int client = 0; client < clients; ++client) {
  IloLinearNumExpr lhs = model.linearNumExpr();
  for (int factory = 0; factory < factories; ++ factory) {
    lhs.addTerm(1.0, shipment[factory][client]);
  }
  model.addEq(lhs, demand[client]);
}
```

However, this language and its ecosystem are still young: Julia first appeared in 2012, and is still under heavy development. Julia 0.3 is the first almost stable branch, but compatibility will not be fully preserved with versions 0.4.x. There is currently no debugger, nor a full-featured IDE (Juno is still burgeoning); as a consequence, ensuring Julia code is properly working is not as easy as for Java code: in particular, Benders' decomposition code brings suboptimal solutions, and is thus omitted from the results.

*Remark.* JuMPeR is a robust extension to JuMP. It provides easy access to all uncertainty sets that can be represented using linear constraints and $L_2$ ellipsoids. However, it has not been relatively stable and well documented until 2015, after this part of the thesis was completed.

### 6.1.4 Instance generation

Facility location instances are randomly generated, using a custom-built file format. It contains three important vectors: the supply (capacity for facilities), the clients' demand, and the fix cost of opening each facility; the other important part is a matrix: the variable cost of transporting one unit from one facility to one client. It has a stochastic variant, with multiple demand scenarios.

Each number is randomly generated as a sequence of random numbers drawn from a uniform distribution around an average value, and a variable standard deviation. The average and standard deviation for each vector or matrix is arbitrarily defined such that the instances are ensured to have at least a feasible solution. Those values are otherwise arbitrarily chosen and hard-coded.

Every instance has as many clients as factories, and scenarios if any. As a consequence, actual problem sizes (number of variables and constraints) evolves as a *cubic* function of the size factor: if the indicated size is $n$, the model will have $n$ binary variables, $n^3$ continuous variables ($n^2$ for deterministic and robust), $2\,n^2 + n^3 + 1$ constraints ($2\,n + n^2 + 1$ for deterministic and robust).

### 6.1.5 Performance comparison

To evaluate the performance of those algorithms, even when focusing on solution time, it is hard to provide an apple-to-apple comparison:

- there is no explicit control of the progress of the solution to the deterministic equivalent, said "direct" (beyond the usual MILP solver parameters, such as gap tolerance);
- at each iteration of the Benders' decomposition, due to constraint (2.5), the solution is guaranteed to be feasible: new iterations will bring optimality, using the set of scenarios;
- progressive hedging may never converge; there is never a guarantee that the solution is feasible, but it is expected that more iterations will give a good solution.

As a consequence, the algorithms were compared on the time to convergence: while Benders' decomposition is ensured to be optimal, progressive hedging is not.

**Methodology.** Each algorithm is run on instances of increasing size, with as many clients as factories or scenarios. Those instances are generated beforehand as text files, and are reused to compare the various algorithms and languages: both Java and Julia are compared on the same data sets; all algorithms are compared on the same data sets.

The total time is stored, with three components: the time to read the instance file, the time to create the model (i.e. variables, constraints, and objective, for the main problem but also the sub-problems if any), and the time to solve it (when the solver works, plus the iterative process if any).

Size increases by steps of 10 from 10 to 250. When an algorithm is no more able to solve problems of a given size (not enough memory on the machine, time budget exceeded), subsequent instances of larger size in the same data set are not tested. As a consequence, time averages use fewer values for larger problem sizes.

To avoid intrinsic variability between runs (some instances are more difficult than others, some bring implementations to perform many iterations; the operating system's scheduler is not deterministic; etc.), ten data sets are generated, and the arithmetic average of those values are computed. Outliers are not removed.

Those benchmark are performed on an Intel i7 975 CPU with 12 GB of RAM, using CPLEX 12.6.0 on Linux. The Java virtual machine is limited to 10.5GB (no explicit limit for Julia), runs are limited to four hours.

## 6.2 Comparison between algorithms

Two metrics are used to compare between algorithms: the number of instances it is able to solve at a given size, and the average time it takes to solve those instances.

### 6.2.1 Number of solved instances

Figure 6.1 shows the number of instances all algorithms could solve. Using a deterministic model on the average case or a robust $L_\infty$ model gives excellent results in this regard: they are able to solve all given instances.
The only difference between those two methods is the choice of values for the uncertain parameters: the first one takes the average over all scenarios, the latter takes the worst value in the $L_\infty$ ellipsoid.
Unlike other algorithms, they are not prematurely stopped during the benchmark process due to resource exhaustion; as they do not have scenarios, the size of the problems they solve increases only quadratically with the size factor, as opposed to the other algorithms.

Progressive hedging is the second best method to solve stochastic facility location problems, as it can cope with rather large instances: up to 200 scenarios with 200 clients and 200 factories; this means a sequence of 200 MIQPs with 200 binary variables each, $200 \cdot 200 = 40,000$ continuous variables, $200 + 200 + 200 \cdot 200 + 1 = 40,401$ constraints, and a quadratic objective.
However, larger instances cannot be solved due to the time it takes (more than four hours overall); these could be worked around using parallelism (either on the same but larger shared-memory machine or using a cluster).

The third rank is occupied by the deterministic equivalent, which solves instances of size up to 150, which are 75% of the size of what progressive hedging can manage. It then solves a single MILP model with 150 binary variables, $150 \cdot 150 \cdot 150 = 3,375,000$ continuous variables, $150 \cdot 150 + 150 \cdot 150 + 150 \cdot 150 \cdot 150 + 1 = 3,420,001$ constraints, and a linear objective.

Larger instances fail due to large memory requirements; buying larger machines to solve those instances is the main solution to this problem, but this is obviously limited (either by technical details or by budget).

Far behind lies Benders' decomposition, which already struggles to solve problems less than 100 in size, less than half what progressive hedging can do. Using multiple cuts even lowers the performance. This decomposition method failed due to large time requirements (more than four hours).

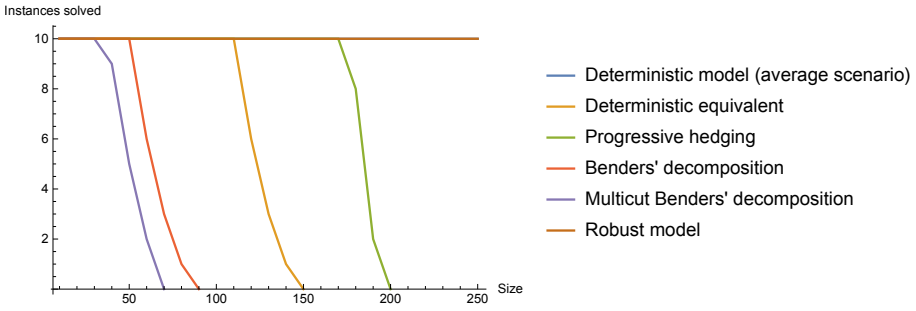


**Figure 6.1** – Facility location: number of instances solved depending on algorithm (Java implementation).

### 6.2.2 Time to convergence

The two winners of the previous paragraph are barely visible in Figure 6.2 for the smallest instances: they take very little time to solution, no more than a few seconds for any instance size, which is in accordance with previous results.

For rather small instances, Benders' decomposition already takes much more time than progressive hedging or the deterministic equivalent. The time they take is also hardly predictable. Using a stronger formulation of facility location considerably slows down the iterative process (of a factor of five, according to limited experiments).

Deterministic equivalent and progressive hedging are almost on par. Around 150, the deterministic equivalent starts to be clearly faster than progressive hedging. It also seems to follow closely a quadratic curve; progressive hedging also seems to follow a quadratic curve, but with much larger variations.

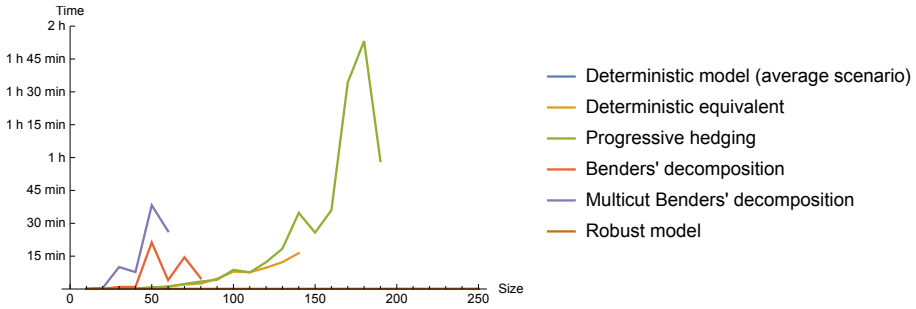


**Figure 6.2** – Facility location: time to convergence depending on algorithm (Java implementation).

### 6.2.3 Iterations count

Only Benders' decomposition and progressive hedging are iterative processes. The number of iterations they perform is not similar, as shown in Figure 6.3: while Benders' decomposition has an average number of iterations of 159, its multicut variant performs a little fewer iterations (148); progressive hedging does only 18 iterations.

When compared to the total time needed for these algorithms, Benders' iterations are similar in speed to progressive hedging's (as shown in Figure 6.4). This is expected by their formulation: Benders' sub-problems are linear (MILPs), while progressive hedging uses $L_2$ regularisation (MIQPs); however, one progressive hedging iterations brings a lot more information than a Benders' iteration. Some experiments show that the use of $L_1$ regularisation in progressive hedging only does not bring improvements in terms of convergence (number of iterations), time (in total or per iteration), or solution quality: it is sometimes better, sometimes not.

The two families of algorithms are very different on one point: Benders' decomposition have relatively large number of iterations, with large increases when problems get larger (standard deviation: 91 for standard Benders' decomposition, 92 for its multicut variant); on the other hand, progressive hedging performs fewer iterations, which very little sensitivity to problem size (standard deviation: 8). It seems progressive hedging is more influenced by the intrinsic difficulty of the problem, i.e. the amount of effort required to push solutions together, the differences between the natural optimal solution to all scenarios.

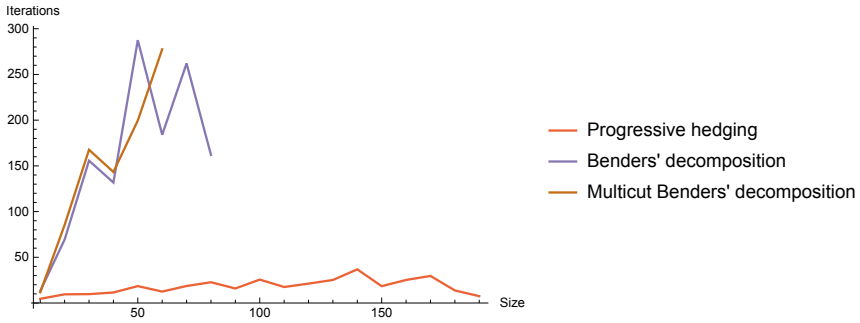


**Figure 6.3** – Facility location: number of iterations depending on algorithm (Java implementation).

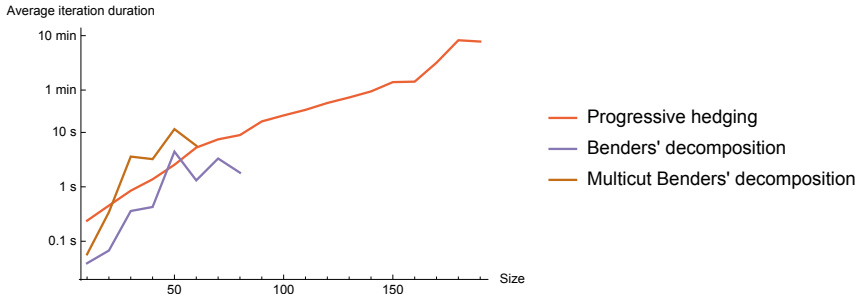


**Figure 6.4** – Facility location: mean iteration time, logarithmic scale (Java implementation).

### 6.2.4 Conclusions

From these results, it seems that Benders' decomposition is not effective at all for solving stochastic facility location with these constraints. As long as computational resources are plentiful compared to the size of the problems to solve, the deterministic equivalent is the most effective way of getting an optimal solution. Once the resources are scarce in front of the size of the problem, the progressive hedging heuristic is the only way of getting a solution.

These conclusions obviously only hold for this particular facility location problem and the generated instances, and could be very different for other problems.

## 6.3 Comparison between languages

The same tests are then performed with the Julia implementation. Due to debugging difficulties, Benders' decomposition is not included in the benchmarks (the code is working, but does not give optimal solutions, indicating bugs).

The number of solved instances shows no clear difference between Julia (Figure 6.5) and Java (6.1): the deterministic equivalent can solve up to a little less than 150 in size factor, and progressive hedging to 180 (instead of 200).

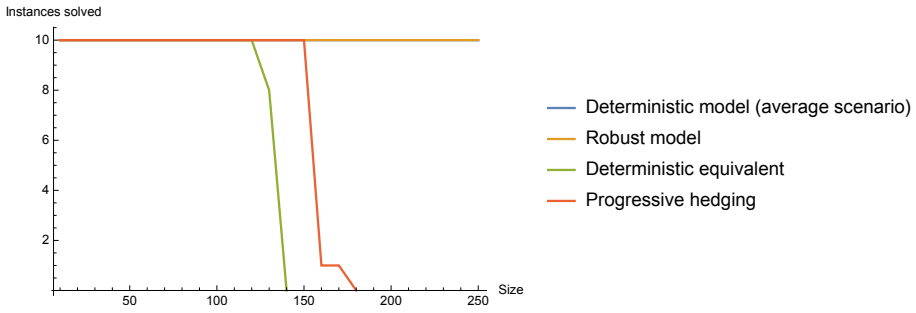


**Figure 6.5** – Facility location: number of instances solved depending on algorithm (Julia implementation).

However, comparing the time is more interesting: Julia (Figure 6.6) has less noisy curves than Java (Figure 6.2), while also being slower. Julia's performance is then much easier to predict than Java's.

This might be due to minor implementation details of the algorithms, but also to the memory management of the virtual machines: Java is tailored for business applications, while Julia is written with high performance computing on clusters in mind. Both virtual machines have garbage collection; as much memory as possible is marked as reclaimable in both codes (calling CPLEX' end() function in Java, using local variables rewritten before next iteration in Julia); in Java, the garbage collector is called before any time measurement is done. Despite this care, the differences remain.

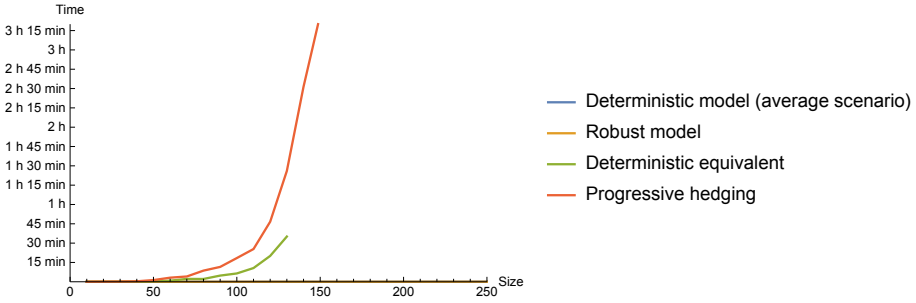


**Figure 6.6** – Facility location: time to convergence depending on algorithm (Julia implementation).

In conclusion, both Java and Julia are very capable of handling those tasks: the noticed differences are rather negligible. Practitioners should not rely on performance beliefs to choose the implementation language for this kind of tasks: the vast majority of the time is taken by the underlying solver. More

computation-heavy code might benefit from Julia's integration of BLAS implementations and other standard libraries, however.

# Chapter 7

# Comparing Robustness: Unit-Commitment

Compared to facility location, unit-commitment as introduced in Section 2.2 is more complex, and has more direct applications. Meaningful data can be obtained from TSOs (transmission system operator, managing the electrical grid): Elia offers data about its generating facilities (`http://www.elia.be/en/grid-data/power-generation/generating-facilities`) and the total load on the Belgian grid (`http://www.elia.be/en/grid-data/Load-and-Load-Forecasts/total-load`). As a consequence, comparisons focuses on resilience against uncertainty.

## 7.1 Implementation details

As for facility location, the main goal is to compare the stochastic and robust approaches. The focus was on modelling rather than solving these problems: the code proposes a two-stage stochastic model, plus various robust $L_\infty$ models, all described in Section 2.2. It also provides a comparison point with a deterministic optimisation on the average scenario.

The robust models only differ in the way they take into account the failures; the three approaches seemed to be very close in the solutions they provide: only the scenario-based method will be evaluated, the most intuitive one.

### 7.1.1 Instance generation

The majority of the instances are generated using UCIG (`http://www.poderico.it/ucig/`). It generates instances of arbitrary size, mixing thermal and hydroelectric units; only thermal units are used. It also includes a spinning reserve; as this constraint is not implemented, the demand is increased by this percentage.

The demand is generated according to a given curve; it is recomputed based on recent historical data from Elia (total load on Belgian grid from January 1

to April 4, 2015), with an average over days. Elia's data provide one datum per fifteen minutes; these are averaged to get hourly load.

Another instance is hand-crafted, purely based on Elia's data, both for the generation units and the load (left untouched, except for—arbitrarily—filling missing loads). All units are included, with their nominal power as per Elia's indications. Other parameters (minimum power, time to start/stop, minimum up/down time, start/fixed/variable costs) are arbitrarily determined based on their type: a nuclear power plant takes much longer to start than a gas turbine. All power plants are modelled as thermal units, including hydraulic reservoirs and wind turbines, although they are obviously of very different nature and constrained differently in reality.

However, all the tests presented in this thesis are performed on either one UCIG instance of ten power plants, or a modified version thereof: each thermal unit is divided in four units, with a fourth of the nominal power; the cost constants are kept identical. This modified version has thus much more flexibility to withstand the uncertainty, while not modifying the total production capacity or the demand.
The hand-crafted instance is not used due to its size: it is too large to be solved decently fast with the current code. Likewise, only a handful of scenarios are used afterwards: at most fifteen. To have a better uncertainty coverage, more scenarios would be useful.

### 7.1.2 Models implementation

A simple progressive hedging implementation is included. It shows how easy it is to implement stochastic programming on top of an existing deterministic implementation: the latter only needs support for penalisation terms, the former is a loop around this model with weight update and heterogeneity computation. A few improvements are implemented: cycle detection (Section 4.3.2.2) and $\rho$ computed for each variable (Section 4.3.2.2).
However, they are not sufficient to make this implementation viable: while the deterministic equivalent is enough to solve realistic instances in a few minutes, progressive hedging gets stuck for hours without showing signs of convergence.

As before, the models are implemented both in Java and Julia. The main difference between those two codes is that the Julia code also has other ellipsoids than $L_\infty$, thanks to JuMPeR implementation of algorithms in Chapter (5).

### 7.1.3 Instance use

Instances have long demands, which span over multiple days, instead of a series of scenarios. This lengthy sequence is then cut into equally-sized chunks (for example, if the instance has data for ten days, every chunk may have five days worth of load). To make two-stage data, the first chunk is used as the first stage (with no uncertainty), and the others are the second-stage scenarios, with equal probability. For example, if the instance has ten days of demand, with a

five-fold chunking, the first stage will have two days of deterministic data, then four second-stage scenarios of two days.

The $L_\infty$ ellipsoid is computed as before: the centre is the average scenario, while its size is computed so that all scenarios are contained inside the ellipsoid. The scenarios and the corresponding ellipsoid are represented in Figure 7.1, using the UCIG-generated demand. This uncertainty set is only used for the "second-stage" variables: the uncertainty is spread only after the recourse happens in stochastic programming.

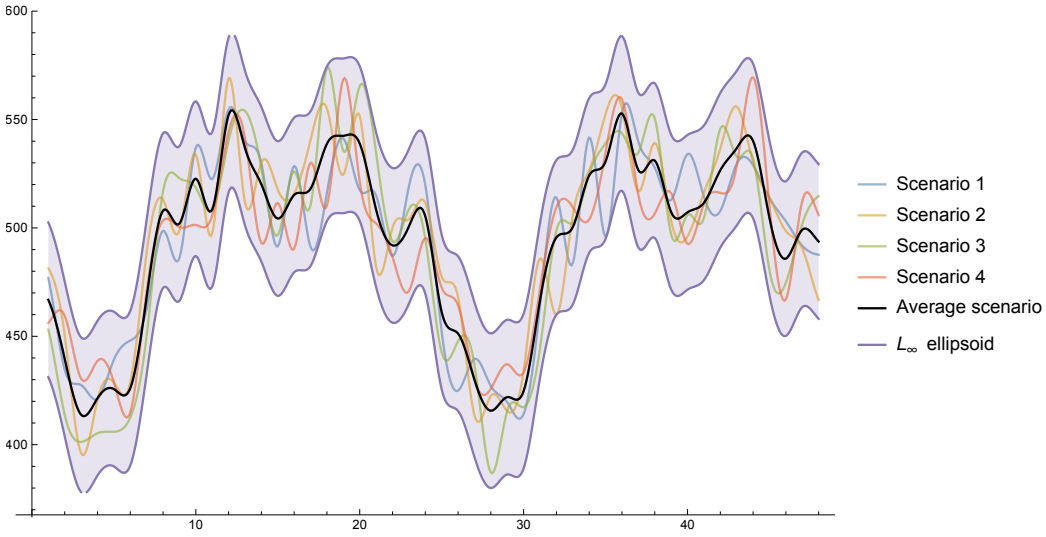


**Figure 7.1** – Unit-commitment: scenarios, average scenario, and $L_\infty$ ellipsoid visualisation. These follow an average day on the Belgian grid.

## 7.2 Need for flexibility

A first robustness test compares the solutions obtained from the three models for two instances: the basic 10-plants instance generated by UCIG, and its modified versions with more power plants. The main result is that the less flexible instance sees equal solutions for both stochastic programming and the deterministic model with the average scenario; the robust solution is never equal to them (as indicated in Figure 7.2).

For the more flexible instance, however, all solutions are different, even though the deterministic average and the stochastic program behave very similarly: they have identical costs (Figure 7.3). However, the robust approach gives a worse-performing solution when tested in the same bench.

It should be noted that changing other parameters have little influence on this conclusion: lowering starting costs, lowering the minimum up/down time, lowering the starting/stopping time are not enough.

The models need some flexibility to withstand the uncertainty; otherwise, its solution space is too restricted: too few solutions with optimal objective value exist. From now on, only this more flexible instance will be used.

| | Deterministic (average) | Stochastic (deterministic equivalent) | Robust (failures as scenarios, Linfinity) |
|---|---|---|---|
| Deterministic (average) | | Identical | Different |
| Stochastic (deterministic equivalent) | Identical | | Different |
| Robust (failures as scenarios, Linfinity) | Different | Different | |

**Figure 7.2** – Unit-commitment: comparison of solutions to the less flexible instance.

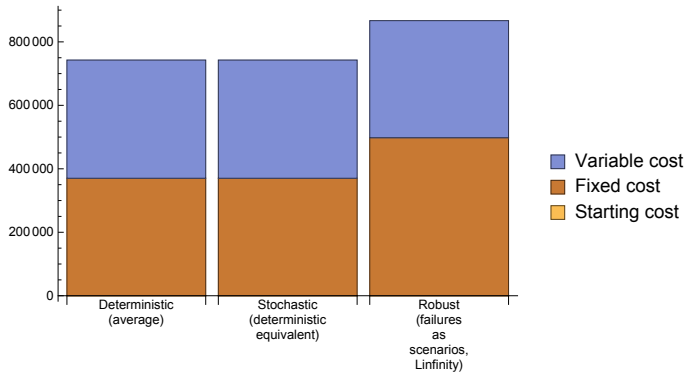


**Figure 7.3** – Unit-commitment: cost distribution depending on the model (more flexible instance).

## 7.3 Change in context

Other robustness tests can be performed based on this instance by modifying it a little: either add more demand scenarios or failures scenarios, move the recourse time sooner (so the uncertainty is spread over more time periods).

### 7.3.1 Demand increase

The current demand scenarios are left untouched; new scenarios are created by multiplying them by $1 \pm \alpha$, where $\alpha$ is the added uncertainty (this process will be called "worsening"). Hence, the number of scenarios is tripled by this operation. Each copy of a scenario have the same likelihood; after this generation process, those numbers are normalised to be probabilities (summing to one).

**Feasibility analysis.** With $\alpha = 5\%$, the behaviour of all three solutions start to clearly diverge. Figure 7.4 shows that the solution to the average scenario is of poor quality: it is not feasible on the two other models, this model is too liberal. This does not indicate that, in practice, this average solution would not fit: it would if the actual demand is close to what was expected; otherwise, it will not even be feasible. The robust model is very conservative, and thus rejects the stochastic solution.

Spending time on the uncertainty is thus useful if the uncertainty is enough present in the problem; if it is barely a characteristic of the reality to model, there are surely other ways to improve the model.

| Solution \ Model | Deterministic (average) | Stochastic (deterministic equivalent) | Robust (failures as scenarios, Linfinity) |
|---|---|---|---|
| Deterministic (average) | | Infeasible | Infeasible |
| Stochastic (deterministic equivalent) | Feasible | | Infeasible |
| Robust (failures as scenarios, Linfinity) | Feasible | Feasible | |

**Figure 7.4** – Unit-commitment: comparison of feasibility when the data is worsened by five percent.

**Robust analysis.** This feasibility analysis does not provide deep robustness information. Another indicator is the maximum size of the $L_\infty$ ellipsoid the solution can withstand. This number is obtained by letting the size of the ellipsoid in the robust program free (instead of setting it to a given value), imposing the solution, then optimising over the ellipsoid size (maximising it).

In this regard, the average scenario brings a solution that can support much less uncertainty: 25.182, as opposed to 62.354 for the stochastic program, and 106.522 for the robust program (Figure 7.5); the ellipsoid has a size 63.555.

This indicates a factor of four between the average and the robust model for this robustness metric; the situation is less terrible with the stochastic program, where the difference is of a factor of roughly two. The stochastic program has a value which is close to the computed ellipsoid size: in this case, speaking in robust terms, the stochastic model is slightly less conservative than the robust approach, but this little difference causes large robustness differences.

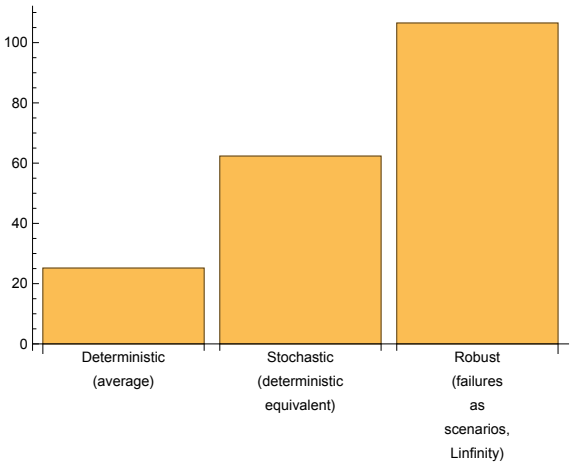


**Figure 7.5** – Unit-commitment: maximum $L_\infty$ ellipsoid size for the given solutions to be feasible when the demand is worsened by five percent.

**Stochastic analysis.** The previous experiment only cares about feasibility of the solution: it does not tell what the objective function is. This is more akin to the stochastic programming approach: see what happens in the scenarios. As a consequence, it requires feasibility for the stochastic model: the deterministic model cannot be evaluated.

The two remaining solutions—stochastic and robust— are very similar, as shown in Figure 7.6. None of them start any machine. The stochastic model spends less in fixed cost (406,933 vs 412,876 for the robust model) but a little more in variable cost (371,590 vs 370,060). Overall, the difference between the two models is 4,413 in favour of the stochastic model.

This issue is similar to overfitting in machine learning: the stochastic model has access to all scenarios, and can fine-tune its solution to this specific situation; on the other hand, the robust model has only a digested version of the scenarios, and tries to optimise the worst case of a set containing all scenarios.

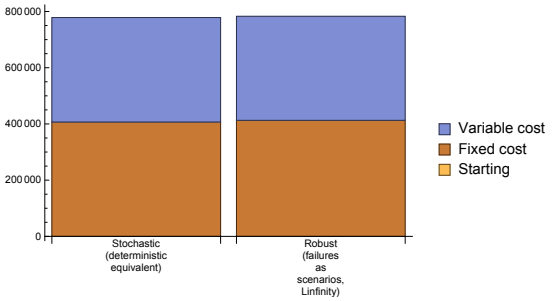


**Figure 7.6** – Unit-commitment: cost distribution (stochastic evaluation through scenarios) when demand is worsened by five percent.

**In brief.** In this case, robust programming thus gives much more risk-averse solutions with a very similar cost in the optimisation scenarios.

### 7.3.2 Failures

A second test case does not change the demand, but adds machine failures as new scenarios. Three machines are considered, based on results of Figure 7.7: one that is used through out the first stage (number 1), one that is used a lot during the first stage (number 5), and one that is never used (number 6). The scenarios see two machines failing throughout the second stage (1 and 6), while the other fails only for the first ten time steps (5). As a consequence, only the decisions taken under uncertainty (the second stage) are directly impacted by the failures; the first stage has to be prepared to withstand these possible failures.

Failures are added as one scenario each, with a rather low likelihood (five percent, which translates into a probability of 4.34% with the four equiprobable demand scenarios). The associated demand is the average of the demand scenarios.

The deterministic optimisation is done two-folds: on the one hand, the demands are averaged, and the failures are aggregated; on the other hand, the failures are forgotten. This model has no notion of scenarios, including for failures: the first model is more constrained, as the failed machines can never be used, whatever the scenario is; the second one is equivalent to the raw data, with no failures.

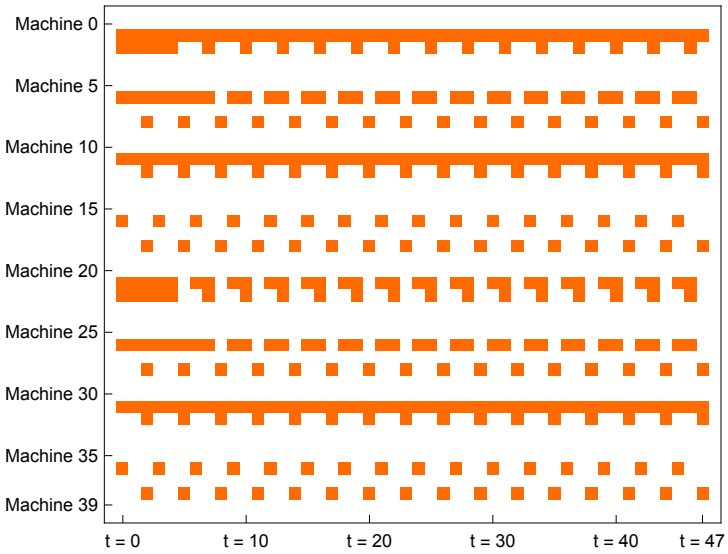


**Figure 7.7** – Unit-commitment: first-stage solution without failures on average scenario, showing which machines are used.

**Feasibility analysis.** The feasibility analysis in Figure (7.8) shows the number of *other* models a solution is feasible against; for example, the solution to the deterministic average is feasible on two other models (out of three). The results are expected: the deterministic average (albeit with all failures) seems less robust than the stochastic solution—the stochastic solution is feasible on two models, while the deterministic average solution is only feasible in one model.

The deterministic average with no failures is the worst approach here: it completely ignores the failures, which are important for the feasibility.

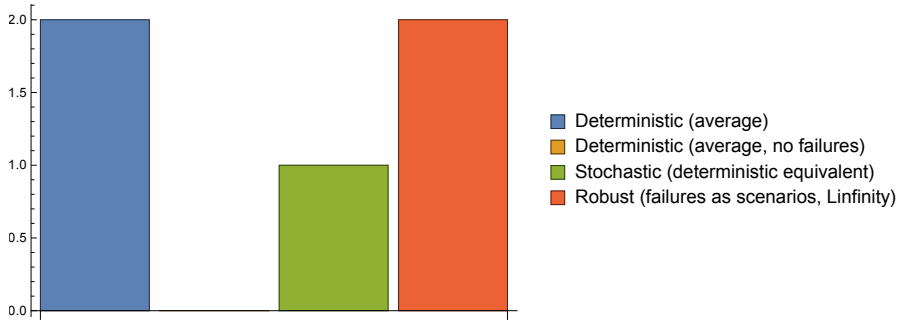


**Figure 7.8** – Unit-commitment: feasibility count of the various solutions against the other models when failures are added.

**Robust analysis.** The robust analysis shown in Figure 7.9 confirms the previous remarks: not caring at all for failures is problematic (the demand must be much lower than the average scenario for this solution to be feasible counting in the failures); optimising with scenarios brings more robustness than using a deterministic model, by a rather large margin; the most robust solution is provided by the robust model.

*Nihil noui sub sole*, except the absolute values: the maximum ellipsoid size is much larger than in previous runs. The robust model can now cope with uncertainty almost five times larger than previously, as for the stochastic solution. Being immune to some failures seems to force the solvers to require more robust solutions.

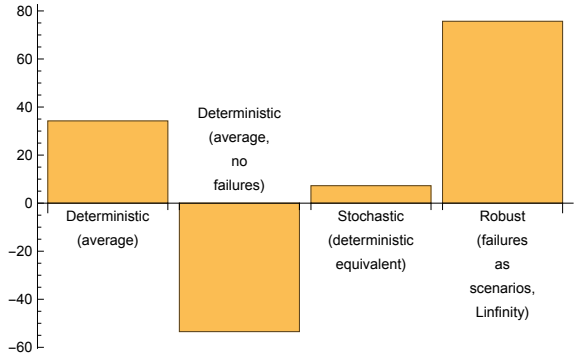


**Figure 7.9** – Unit-commitment: maximum $L_\infty$ ellipsoid size for the given solutions to be feasible when failures are added.

**Stochastic analysis.** Another point of view brings explanations to this difference in robustness: Figure 7.10 shows that the stochastic solution has a lower cost than the two others. This is clearly a trade-off between robustness and cost: a cheaper solution has worse robustness, but not by large amounts.

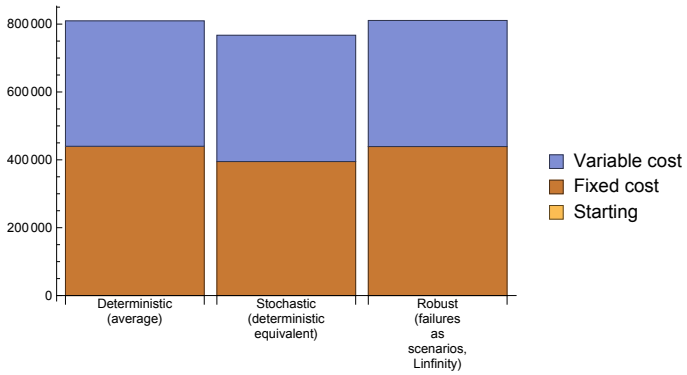


**Figure 7.10** – Unit-commitment: cost distribution (stochastic evaluation through scenarios) when failures are added.

**In brief.** The robust model can withstand much larger uncertainty than the stochastic model; the latter is also overtaken by deterministic average. The large differences between those models are probably due to the very low number of scenarios used for optimisation (only four). This does not allow for a good coverage of the real uncertainty: the solution is indeed optimal for those scenarios, but falls short if the uncertainty is larger.

### 7.3.3 Sooner recourse action

The last test in this section changes the time the recourse action happens: currently, the instance has a total length of four days, with a recourse after two

days; this test focuses on changing this recourse so it happens after six hours (i.e. bring it forty-two hours sooner). Doing so thus reduces the first-stage, and needs lengthier scenarios for the second stage.

This setting should be closer to what real users such as Elia would do: optimise their decisions for a few time steps, but take into account the uncertainty about the future, to avoid being taken by surprise; keep flexibility in the schedule in the case of forthcoming events, foreseeable or unforeseeable.

**Implementation details.** The missing parts are filled using a random number generator: the average is the corresponding value in what was the first stage; this $[-1, 1]$-random number is then multiplied by the standard deviation of previous second-stage scenarios. As a consequence, the filled time steps should be close to the actual scenarios, with similar fluctuations over time.

This has no impact on the size of the ellipsoid; however, it is used for almost all time steps in the program, compared to only a half previously.

**Feasibility analysis.** Figure 7.11 clearly shows that this is the only case where the stochastic solution is feasible against the robust scenario, which does not happen for the other test cases. This could be a hint that the robust model is less conservative in this case, or that the scenarios span over the same uncertainty set as the robust model.

The deterministic average's solution is also feasible on the stochastic model—which is not necessarily true for the other test cases.

| Solution \ Model | Deterministic (average) | Stochastic (deterministic equivalent) | Robust (failures as scenarios, Linfinity) |
|---|---|---|---|
| Deterministic (average) | | Feasible | Infeasible |
| Stochastic (deterministic equivalent) | Feasible | | Feasible |
| Robust (failures as scenarios, Linfinity) | Feasible | Feasible | |

**Figure 7.11** – Unit-commitment: feasibility count of the various solutions against the other models when the recourse action happens sooner.

**Robust analysis.** The maximum ellipsoid sizes of Figure 7.12 have the same shape as previously: the deterministic average can withstand much lower uncertainty than the stochastic solution, itself much less than the robust model.

As with failures, the maximum allowable uncertainty is larger, for all three models. This can be explained by the larger flexibility let to the solver when evaluating the solutions: the models only impose theirs for the first stage (six hours), while evaluating this robustness score lets the solver optimise for the rest of the four days.

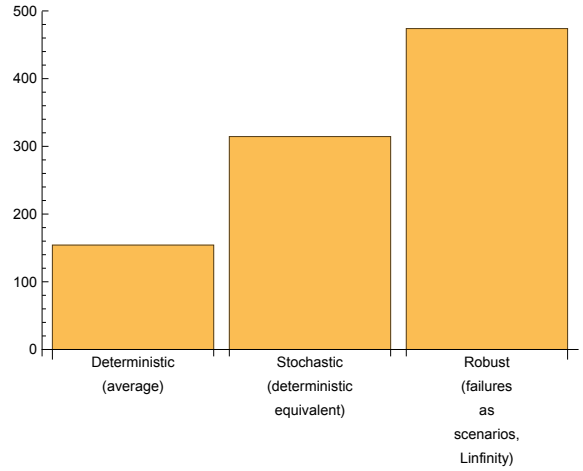


**Figure 7.12** – Unit-commitment: maximum $L_\infty$ ellipsoid size for the given solutions to be feasible when the recourse action happens sooner.

**Stochastic analysis.** As previously, the cost decomposition of Figure 7.13 shows that the stochastic model is able to provide better solutions for this set of scenarios. This remarks still holds when the set of scenarios used for the cost computation is worsened by ten percent (as in Figure 7.14), but the difference is slightly reduced.

The deterministic average solution is worse than the two others; in particular, it is the only one to start a machine at some point.

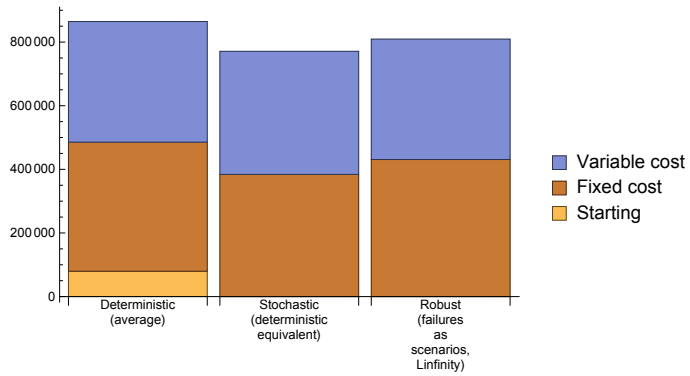


**Figure 7.13** – Unit-commitment: cost distribution (stochastic evaluation through scenarios) when the recourse action happens sooner.

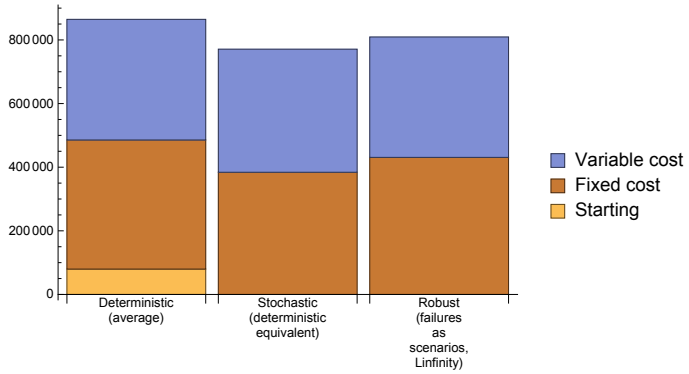


**Figure 7.14** – Unit-commitment: cost distribution (stochastic evaluation through scenarios) when the recourse action happens sooner; data is worsened by ten percent when evaluating the cost.

**In brief.** Bringing the recourse action sooner gives more flexibility to the models to withstand higher uncertainty.

Overall, this set of results has the most natural look throughout the plots: forgetting the uncertainty gives the worst solutions of all three models, but they are feasible (even though they do not remain feasible when the demand increases a lot); with stochastic programming, the cost is reduced to the minimum, but unexpected variations may make the schedule infeasible; robust programming provides the most robust solution, in the sense that it can resist much larger uncertainty. The only point that may look less intuitive is that robust programming's solution has lower costs than the deterministic average's.

## 7.4 Monte Carlo evaluation

The previous tests are more academic than practical: they choose some parameter to change, then test its impact on the solutions behaviour. Real applications do not have this freedom: the actual scenario is not known in advance, it might have unpredictable demand, or many failures at once.

To assess the practical capabilities to withstand the uncertainty, a better solution is to use a Monte Carlo approach. It consists in a generation of a large number of scenarios, then the evaluation of the solutions on these scenarios; the solutions are obtained from running the algorithms on the base flexible instance. Some properties of the solutions are then estimated from all these runs.

The scenarios are generated using a Gaussian random number generator: the average is the average scenario for the second stage, with a standard deviation of twice the $L_\infty$ ellipsoid size. As a consequence, the probability the generated number is inside the ellipsoid is:

$$\mathcal{X} \sim \mathcal{N}\left(\mu, \frac{\sigma}{2}\right), \qquad P(\mu - \sigma \leq \mathcal{X} \leq \mu + \sigma) = \operatorname{erf}\sqrt{2} \approx 95.5\%. \tag{7.1}$$

As a consequence, when drawing one thousand scenarios, about forty-five of them should be outside the $L_\infty$ ellipsoid; they should be close to the stochastic scenarios, due to the chosen standard deviation (smaller than the ellipsoid).

**Feasibility analysis.** The feasibility analysis shows very good results for the robust solution: it is feasible on almost all scenarios, only two out of on thousand are not, which is better than the 95.5% feasible that were expected based on scenario generation. This should be expected, given the previous robustness analyses: the actual maximum uncertainty the solution can withstand is often much larger than the minimum required.

The deterministic solution is no exceptional: its results are rather bad, being feasible only on 14.4% of the scenarios. What is less expected is the results for the stochastic model: it is barely better than the deterministic one (17.4%). This is however consistent with previous results that showed that the stochastic model seems to trade robustness against optimality—which it is allowed and even advised to do, as long as it remains within the boundaries defined by the optimisation scenarios.

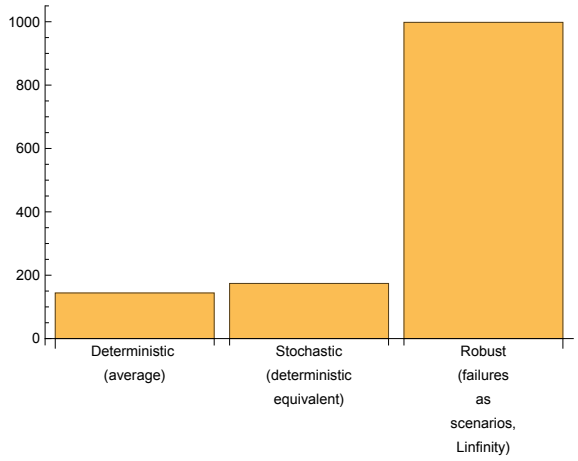


**Figure 7.15** – Unit-commitment: feasibility count on 1,000 scenarios.

**Stochastic analysis.** The other side of the coin is about the cost when implementing the solutions in the generated scenarios. As expected, the robust solution is more costly than the others, but this average was computed on 998 trials, not 144 or 174. It is more surprising to see that the deterministic solution is slightly *better* than the stochastic solution, with about 150 monetary unit of difference.

The stochastic model seem thus to focus on the optimisation scenarios, without being able to generalise to close scenarios. As before, the robust model increases the fixed cost, which allows it to decrease its variable costs.

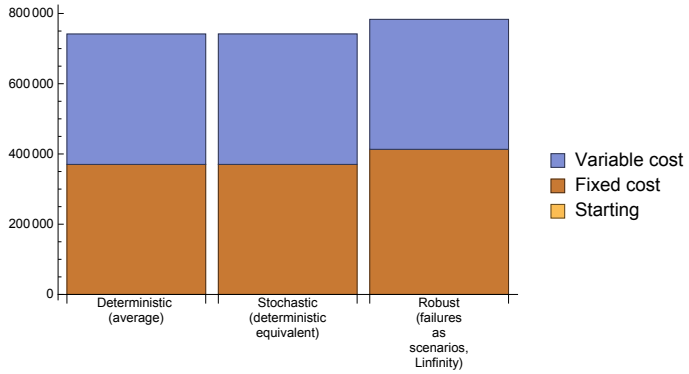


**Figure 7.16** – Unit-commitment: cost distribution (stochastic evaluation through scenarios) averaged on 1,000 scenarios.

## 7.5 Summary

Overall, based on these experiments, the three models can be summarised as this:

- the deterministic model is more than enough when the uncertainty is mild (slight variations between the scenarios);
- the stochastic model shines when the uncertainty is much more present (such as failures) *and* that the scenarios provide a good coverage of the uncertainty;
- the robust model avoids stochastic overfitting due to a small set of scenarios, as it directly uses none of them.

About the contents of each solution, the deterministic model is the only one to sometimes start machines. The stochastic model tends to find a balance between variable and fixed costs (it seems to lower the number of machines let running), while the robust model prefers to keep machine on and to use them less (keeping more flexibility for adverse events).

Other robust models could be explored, e.g. using a $L_1$ or $L_2$ norm instead of the $L_\infty$. They are devised to lower the conservativeness of the robust models: the model is less tight, which prevents the solution from being that immune to uncertainty, but allows reductions in the objective. (Those other models are implemented in the Julia version of the code, which has no integration with the benchmarking programs; time did lack to include those other models into the benchmarks.)

# Chapter 8

# Drawing Conclusions

This master thesis first introduced the need for uncertainty in optimisation, then briefly surveyed some modelling and implementation issues of stochastic and robust programming. At its heart however stands the actual implementation and comparison of those two paradigms.

The first comparison focused on performance issues of taking into account the uncertainty. The model was facility location, a rather standard and common problem in operational research. It was the opportunity to implement a deterministic model, some algorithms to solve its stochastic variant (namely, the deterministic equivalent, Benders' decomposition, and progressive hedging), and a robust model (with no specific algorithm).

The second comparison was about robustness issues of the solution in energy systems, with the unit-commitment problem. Three models were developed: one deterministic, the other stochastic, the latter robust. Those three models have different properties when faced with uncertainty. (Only one dedicated algorithm was implemented: progressive hedging, for stochastic programs.)

Those were implemented in two languages: Java, historically present, for general purposes; Julia, a newcomer in the scientific computing domain with a strong optimisation community. Their main difference is the ease of use for modelling and coding mathematical algorithms.

Currently, the literature hardly explored the comparison between those two paradigms: it focuses on one or the other. This fact is also reflected by the fact that very few tools implement both stochastic and robust programming. There is thus little insight on what paradigm to use in which case: this thesis expects to bring some elements to the debate.

*Remark.* The facility location code counts 3187 lines of Java code (160 KB) and 1341 lines of Julia code (56 KB). The unit-commitment code has 4540 lines of Java code (248 KB), plus 1115 lines (64 KB) for the comparisons; it has 1707 lines of Julia code (80 KB). Overall, this means 11,890 lines of code (608 KB).

## 8.1 Stochastic vs. robust

The stochastic point of view is to use probabilistic and statistic information and to bring it into the optimisation problem: instead of minimising the costs for a particular set of values for the parameters, it proposes to optimise according to a probability distribution of those parameters: mainly by minimising the *expectation* of the objective function, but also by including chance constraints.

On the other hand, robust programming is concerned with uncertainty sets: the solution must be feasible for all points in this set. The new objective function is usually to minimise the worst case inside this set: every sentence about robust programming is very likely to contain the words "worst case."

The stochastic mindset might be more sound to practitioners, used to statistical thinking. On the other hand, robust programming uses concepts not as widely spread. Both of them make sense to compare solutions, as what is done in Section 7.3.

Before thinking about adding uncertainty to an existing model, though, the first question to ask is: could it be useful? The results of Section 7.2 indicate that too mild uncertainty makes this operation costly (in development and evaluation time, mainly) with little gains over existing deterministic model.

### 8.1.1 Modelling issues

Stochastic programming has a great advantage over robust programming: it can naturally welcome recourse actions, i.e. decide that, at some point in the time frame of the optimisation program, the operators will have the opportunity to reconsider its decisions for the following time steps, based on the information they retrieved up to that instant. Those recourse actions may happen at various points. On the other hand, robust programming can have parametrised recourse ("robust adaptable optimisation" in [5]), but this technique is less powerful.

Both paradigms can lead to various choices in the modelling: Should a stochastic model include one or more recourse actions? Should a constraint be loosened using probabilities (chance constraint)? How to model a robust worst case (see failures modelling)? Those various choices may make sense, depending on the application; what is more, they are not incompatible: both may be mixed in a given model; robust programming can also be used to approximate stochastic programming.

Given the results obtained in Section 7.3, more complex mixes can be devised: the stochastic model can give a series of optimal solutions, the one with the largest robustness might be chosen, so that the best of both worlds is kept (notwithstanding implementation issues). Section 7.3.1 shows that the stochastic and robust models have solutions of similar cost, but very different maximum acceptable uncertainty: this hints that mixing stochastic objective and maximum uncertainty, for example through multiobjective optimisation, might make sense in some cases.

### 8.1.2 Implementation issues

To take uncertainty into account in an existing deterministic program, the easiest approach to implement is stochastic programming: using progressive hedging, it is possible to solve stochastic programs by only adding penalisation terms to the existing deterministic code. Thus it does not impose large changes in existing code base: add a few methods to the existing code, then build an iterative process around it; it can then be parallelised easily in case the performance is lower than the requirements.

Robust programming can also be implemented with even less effort for $L_\infty$ ellipsoids, as it is simply replacing the parameters by their worst-case value in the uncertainty set. Other solving algorithms are more cumbersome to implement, except when they are in a modelling layer.

On the other side, more advanced implementations can go up to solving the deterministic equivalent on supercomputers, as the projects PIPS-S and PIPS-IPM try to do: they manage to solve *"12-hour horizon problems with more than 32,000 scenarios in under 1 hour"* (`http://www.mcs.anl.gov/~petra/pips.html`).

The effort to implement uncertain models can thus vary wildly, depending on the needs. A quick robust implementation will give very conservative solutions, but throwing more effort at the implementation could bring a better trade-off.

### 8.1.3 Applicability

Stochastic programming requires access to statistic information, i.e. historical data and forecasts: these may not be available, or at a high price. If a stochastic model is used with too few scenarios (or badly chosen given the uncertainty), its results might be below expectations due to overfitting those scenarios, as Chapter 7 concludes.

On the other hand, robust programming does not need precise information: it is possible to build uncertainty sets from aggregated statistical moments, for example. With little information, it is able to provide solutions that generalise well—is suffers less from overfitting.

These thoughts must be moderated: common forecast methods for time series like ARMA (autoregressive moving-average) provide a prediction, but also confidence intervals about the prediction. This exactly matches the uncertainty sets used for unit-commitment, where the confidence interval is the ellipsoid size.

### 8.1.4 Personal opinion

The results obtained in Chapters 6 and 7 indicate that, when performance really matters, robust programming is the only viable option: its solution time is very close to that of the deterministic model. However, this model brings rather conservative solutions: the two approaches have both advantages and disadvantages.

A solution TSOs like Elia could implement is to have stochastic models to plan decision on a longer term (a few hours in advance), so the number of scenarios can be chosen to have a good representation of the uncertainty; this model could be complemented by a robust program, run more often, on the short-term (fifteen minutes), to adapt the decisions to the current realisation, with lower uncertainty (and thus conservativeness).

## 8.2 Going further

The only robust model thoroughly tested is based on an $L_\infty$ norm, which is known to be very conservative. Other uncertainty sets could be tried to explore new compromises between risk-adversity and optimality.

The unit-commitment instances included in the tests are very small: [9] uses 50 demand scenarios, with 1,000 power plants, and a time-horizon of 48 hours (with a 15-minute discretisation). Their goal is *"to solve moderate-scale stochastic unit commitment problems with reasonable numbers of scenarios in less than 30 minutes of wall clock time on a commodity hardware:"* in its current state, the code is not able of such performance.

More down to earth, the code implementing the various models has very large redundancy: it consists in large chunks of variables and constraints copied and pasted from one model to another, with all the problems this might cause on the long run. With no modelling environment available in general-purpose programming languages, reducing this redundancy can only be done at the expense of a very large increase in code complexity, which may lower the expected maintenance benefits.

The conclusions above only hold for the tested cases: this study is far from exhaustivity, it focused on only two problems. To draw more general conclusions, the study should be generalised to more problems, for example through meta-analysis. However, few such analyses have currently been performed, preventing more global analysis: [13] is one such comparison, and the authors notice that *"research on comparing both approaches is scarce"*.

# Bibliography


[1] Balling, L. (2011). Fast cycling and rapid start-up: new generation of plants achieves impressive results. *Modern Power Systems*.

[2] Bandi, C. and D. Bertsimas (2012). Tractable stochastic analysis in high dimensions via robust optimization. *Mathematical Programming 134* (1), 23–70.

[3] Ben-Tal, A., L. E. Ghaoui, and A. Nemirovski (2009). *Robust Optimization* (First ed.). Princeton Series in Applied Mathematics. Princeton University Press.

[4] Ben-Tal, A. and A. Nemirovski (2002). Robust optimization: methodology and applications. *Mathematical Programming 92* (3), 453–480.

[5] Bertsimas, D., D. B. Brown, and C. Caramanis (2011). Theory and applications of robust optimization. *SIAM Rev. 53*, 464–501.

[6] Bertsimas, D., I. Dunning, and M. Lubin (2014). Reformulations versus cutting planes for robust optimization. *Optimization Online*.

[7] Birge, J. R. and F. Louveaux (2011). *Introduction to Stochastic Programming* (Second ed.). Springer Series in Operations Research and Financial Engineering. Springer Verlag.

[8] Birge, J. R. and F. V. Louveaux (1988). A multicut algorithm for two-stage stochastic linear programs. *European Journal of Operational Research 34* (3), 384–392.

[9] Cheung, K., D. Gade, C. Monroy, S. Ryan, J. Watson, R. Wets, and D. Woodruff (2013). Toward scalable stochastic unit commitment, part 2: assessing solver performance. *IEEE Transactions on Power Systems (submitted)*.

[10] Dantzig, G. B. (2002). Linear programming. *Operations Research 50* (1), 42–47.

[11] Dantzig, G. B. (2011). *Linear Programming Under Uncertainty*, Volume 150 of *International Series in Operations Research and Management Science*, Book section 1, pp. 1–11. Springer New York.

[12] Diwekar, U. (2008). *Introduction to Applied Optimization* (2nd ed.), Volume 22 of *Springer Optimization and Its Applications*. Springer US.

[13] Evers, L., K. Glorie, S. van der Ster, A. Barros, and H. Monsuur (2012). The orienteering problem under uncertainty stochastic programming and robust optimization compared. Report, Erasmus University Rotterdam, Erasmus School of Economics (ESE), Econometric Institute. Econometric Institute Research Papers.

[14] Gade, D., G. Hackebeil, S. Ryan, J. Watson, R. Wets, and D. Woodruff (2013). Obtaining lower bounds from the progressive hedging algorithm for stochastic mixed-integer programs. *Under Review*.

[15] Goh, J. and M. Sim (2011). Robust optimization made easy with rome. *Operations Research 59*(4), 973–985.

[16] Gorissen, B. L., I. Yanikoglu, and D. den Hertog (2015). A practical guide to robust optimization. *Omega 53*(0), 124–137.

[17] Kreyszig, E. (2011). *Advanced Engineering Mathematics*. John Wiley and Sons.

[18] Loefberg, J. (2012). Automatic robust convex programming. *Optimization methods and software 27*(1), 115–129.

[19] Long, Z. and Z. Bo. Robust unit commitment problem with demand response and wind energy. In *Power and Energy Society General Meeting, 2012 IEEE*, pp. 1–8.

[20] Minoux, M. (2009). *Robust linear programming with right-hand-side uncertainty, duality and applications Robust Linear Programming with Right-Hand-Side Uncertainty, Duality and Applications*, Book section 569, pp. 3317–3327. Springer US.

[21] Mittelmann, H. D. (2014). Mixed integer q(c)p benchmark.

[22] Mittelmann, H. D. (2015). Mixed integer linear programming benchmark (miplib2010).

[23] Morales-Espana, G., C. Gentile, and A. Ramos (2014). Tight mip formulations of the power-based unit commitment problem.

[24] Murphy, J. (2013). Benders, nested benders and stochastic programming: An intuitive introduction. *ArXiv e-prints*, 57.

[25] Powell, W. B. (2014). Bridging the fields of stochastic optimization. *Optima* (96).

[26] Shapiro, A., D. Dentcheva, and A. Ruszczynski (2009). *Lectures on Stochastic Programming: Modeling and Theory* (First ed.). MOS-SIAM Series on Optimization.

[27] Taskin, Z. C. (2010). Benders decomposition. *Wiley Encyclopedia of Operations Research and Management Science. John Wiley and Sons, Malden (MA)*.

[28] Watson, J.-P. and D. L. Woodruff (2011). Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems. *Computational Management Science 8*(4), 355–370.

[29] Watson, J.-P., D. L. Woodruff, and W. E. Hart (2012). Pysp: modeling and solving stochastic programs in python. *Mathematical Programming Computation 4*(2), 109–149.

[30] Wentges, P. (1996). Accelerating benders' decomposition for the capacitated facility location problem. *Mathematical Methods of Operations Research 44*(2), 267–290.

[31] Wills, B. A. and T. Napier-Munn (2005). *Wills' Mineral Processing Technology*. Wills' Mineral Processing Technology (Seventh Edition). Oxford: Butterworth-Heinemann.

[32] Wolsey, L. A. (1998). *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. John Wiley and Sons.

[33] Zhang, M. and Y. Guan (2009). Two-stage robust unit commitment problem. *University of Florida, USA*.