Czech Technical University in Prague
Faculty of Electrical Engineering

Bachelor Thesis

Modelling languages for optimization

Prague, 2010

Author: Michal Podhradský

Supervisor: Petr Havel

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídicí techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Michal Podhradský**

Studijní program: Softwarové technologie a management
Obor: Inteligentní systémy

Název tématu: **Modelovací jazyky pro optimalizaci**

Pokyny pro vypracování:

1. Seznamte se s možnými modelovacími jazyky pro formulaci celočíselných lineárních optimalizačních problémů (např. AMPL, ZIMPL, GAMS, YALMIP apod.)

2. Porovnejte jejich funkce, možnosti, rozhraní, uživatelskou přívětivost, cenu apod.

3. Vybrané jazyky použijte pro implementaci příkladu optimálního plánování provozu teplárny a zhodnoťte jejich použitelnost pro daný typ úlohy

Seznam odborné literatury:

Dodá vedoucí práce

Vedoucí: Ing. Petr Havel

Platnost zadání: do konce zimního semestru 2010/2011

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

L.S.

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 12. 10. 2009

Declaration

I declare that I have created my Bachelor Thesis on my own and I have used only literature cited in the included reference list.

In Prague, 21st May 2010

signature

Acknowledgement

I would like to thank my supervisor Petr Havel for guidance and useful comments and also the people from the Centre for Applied Cybernetics in Prague for consultations kindly provided. And last but not least, I would like to show my appreciation to my family for the support they have given me during my studies and say many thanks to Jo for lending her proofreading skills.

And very, very special thanks belongs to Monika for her support and endless patience.

Abstrakt

Cílem této bakalářské práce je porovnání v současnosti dostupných modelovacích jazyků pro formulaci celočíslených lineárních optimalizačních problémů a doporučení jazyka, který je nejvhodnější použít pro modelování a plánování optimálního provozu tepláren a kogeneračních jednotek. Takový jazyk musí být schopen rychlého převodu modelu do formátu čitelného zvoleným solverem, musí být snadno propojitelný s Java aplikacemi a musí být schopen načtení a ukládání dat z a do MS Excel tabulek.

Dostupné (komerční i open-source) jazyky a jejich základní funkce jsou nejprve porovnány v přehledné tabulce. Následně je devět jazyků (Yalmip, GAMS, OptimJ, Gurobi Java API, LINGO, AIMMS, AMPL, MPL a Zimpl) vybráno pro další testování, sestávající se z implementace zjednodušeného modelu teplárny. Během této implementace je sledována zejména snadnost formulace problému, propojitelnost s Java aplikacemi, přehlednost kódu a možnosti manipulace s MS Excel tabulkami. Na závěr jsou vybrány tři jazyky (Yalmip, OptimJ, Zimpl) a je na nich testována rychlost formulace problému pomocí rozšířeného modelu teplárny.

Jako nejrychlejší se ukázal být jazyk Zimpl, nicméně jako nejvhodnější pro reálné nasazení se jeví jazyk OptimJ (díky svému propojení s Javou a dostačující rychlostí). Jako vhodné se dále jeví jazyky používající algebraickou notaci, například AMPL, GAMS nebo LINGO. Yalmip se ukázal pro praktické nasazení nevhodný zejména kvůli pomalé formulaci problému.

Abstract

The aim of this work is to thoroughly compare various currently available modelling languages for Mixed-Integer Linear Programming (MILP), both commercial and open-source, and eventually choose and recommend the one which is the most suitable for the task of modelling and optimal scheduling of cogeneration systems. A desired modelling language has to have a good performance while extracting the model into a solver-readable format, has to integrate well with a Java environment and has to be capable of reading and writing to MS Excel spreadsheets. However, the comparison is useful for anybody facing similar optimization and scheduling problems.

First, an overall comparison of available languages and their basic features is created. According to this comparison the 9 most promising languages are chosen for further testing (Yalmip, GAMS, OptimJ, Gurobi Java API, LINGO, AIMMS, AMPL, MPL and Zimpl). The second part of testing consists of implementing a model of a simple CHP system and investigating languages model formulating features, possibilities of Java interacting and facilities for MS Excel spreadsheets manipulation. Regarding the results of the second part, three languages (Yalmip, OptimJ, Zimpl) are selected for implementation of a more complicated model and benchmarking.

The fastest language is Zimpl, however the most suitable language for commercial use is OptimJ (thanks to its Java integration and reasonable speed). Algebraic languages such as AMPL, GAMS and LINGO are worth considering. Yalmip is not recommended for commercial use due its slow performance.

Contents

Nomenclature

Variable	Description
$\mathbf{PK}^{state}$	Binary vector representing status of boilers (on/off)
$\mathbf{TG}^{state}$	Binary vector representing status of turbines (on/off)
$\mathbf{Mp}^{PK}$	Vector corresponding to the steam outflow from boilers [t/h]
$\mathbf{Mp}^{TG}$	Vector corresponding to the steam flow through turbines [t/h]
Mp^{VK}	Steam flow to the condenser [t/h]
Mp^{ZO}	Steam flow to the water heater [t/h]
dev	Electric power deviation from the desired value [MW]
$\mathbf{P}^{TG}$	Vector corresponding to the electric power produced by generators [MW]
$\mathbf{Q}_{in}^{PK}$	Vector corresponding to the power input of boilers [MW]
$\mathbf{Q}_{out}^{PK}$	Vector corresponding to the power output of boilers [MW]

Parameter	Description
m	Number of boilers
n	Number of steam turbines
Q_{demand}	Desired heat production [MW]
P_{demand}	Desired electric power production [MW]
$\mathbf{Mp}^{TGmin}$	Vector representing the minimal allowed steam flow through turbines [t/h]
$\mathbf{Mp}^{TGmax}$	Vector representing the maximal allowed steam flow through turbines [t/h]
i_{in}^{PK}	Enthalpy of feeding water at the input of boilers [kJ/kg]
i_{out}^{PK}	Enthalpy of steam at the input of turbines [kJ/kg]
i_{out}^{TG}	Enthalpy of steam at the output of turbines [kJ/kg]
c_{fuel}	fuel cost [CZK/MW]
c_{dev}	contracting penalty for each MWh

Chapter 1

Introduction

The aim of this work is to thoroughly compare various currently available modelling languages for Mixed-Integer Linear Programming (MILP), both commercial and open-source ones, and eventually choose and recommend the one which is the most suitable for the task of modelling and optimal scheduling of cogeneration systems. The cogeneration system is a system that simultaneously generates both electricity and useful heat. Scheduling means both determining the optimal on/off states (*unit commitment*) of the system units and their output (their *economic dispatch*) for each time interval of the planning horizon [27].

Modelling and scheduling of such systems using MILP requires the following steps:

1. Describing a real cogeneration system with a set of mathematical equations. Simplification of the system is often needed.

2. Using a modelling language to create an optimizaion model and formulate the equations of the system from step 1 (i.e. an objective function and a set of constraints).

3. Seting the parameters of the model, such as the desired heat and electric power production, the planning horizon and other restrictions.

4. Using features of the modelling language to transfer the model into an MILP problem format and let this problem be solved by appropriate MILP solver.

5. Showing the results of optimization and the final schedule in a user-friendly way.

The first objective of this bachelor thesis is to create a basic overview of available modelling languages and their features. This overview is to give us a transparent comparison of the modelling languages and to show their advantages and drawbacks, which

can be useful for any researcher facing a similar optimizing problem. The second objective is to recommend one modelling language that best suits all requirements that bring optimal scheduling of cogeneration systems. This modelling language will be used in further projects held in the Department of Control Engineering that are aimed at optimal scheduling of real cogeneration plants in the Czech Republic and Slovakia. One of the requirements is an easy implementation of a large scale model (e.g. hundreds of equations) and fast transformation of such a model into an instance of MILP problem solvable by common MILP solvers. A model of a real cogeneration plant usually contains a large number of variables and equations, thus slow transformation of the model into MILP problem can dramatically increase the computing time of the scheduling.

Another important requirement is the ability to operate with MS Excel spreadsheets. MS Excel is one of the most common file formats used for data storage in business applications, thus it is expected that in a real application the data for the scheduling will be stored in this format. Hence it is desirable that the modelling language is capable of both reading and writing to spreadsheets.

The third considerable feature of such a language is a good connectivity with Java programming language. Currently a similar project aimed at scheduling cogeneration systems was held at the Brno University of Technology. This project's graphical front-end was created in Java and possible integration or extension of this project is under consideration. It is expected that the final application for scheduling is to be writen in Java (using its libraries), and Java will link all necessary components. For example, an operator creates a model of a cogeneration plant using user-friendly GUI, the application automatically formulates equations, loads necessary data, creates a MILP problem, then call an appropriate MILP solver and shows/writes the results of scheduling.

The comparison of the modelling languages is divided as follows:

1. Creating a survey of available languages.

   (a) The first step is to create a synoptical overview of modelling languages and its basic features. This overview is divided into four sections:

   General information – platform availability, type of application etc.

   Inputs/Outputs – supported file formats, connectivity with solvers etc.

   Price and licensing – price and licence information.

   Problem formulation – support of non-linear functions and manual setting of branching priorities etc.

(b) This overview mainly consists of literature and Internet research. Previous articles of similar topics are used, as well as modelling languages' manuals. Vendors' homepages are also being searched for additional information. Results of this overview is shown using a lucid spreadsheet. Finally, a list of the most promising modelling languages is created.

(c) The survey is thoroughly described in chapter 2 and the results of this survey can be found in appendix A.

2. Implementation of a basic model.

(a) The second step of the testing consists of the implementation of a basic model of a cogeneration system. The description of this model can be found in section 1.1. Various features of the languages, such as input/output file formats are further inspected. This part of testing is aimed at the following aspects:

Inputs/Outputs – input format of user-defined data, exporting the model into MPS or LP file formats (standard format of MILP problems, for more details see [32, 10]) and connectivity with Java applications.

Declaration of equations – user's experience with model formulation, defining variables, constraints and objective function.

MS Excel connectivity – ability to read and/or write data from/to MS Excel spreadsheets.

Code clarity and debugging – debugging tools, legibility of the final code.

Solver options and branching priorities – changing various solver settings, setting up branching priorities on binary variables etc.

Non-linear functions – formulation and usage of non-linear functions and its limitations.

Price and licensing – discussing various licence and price options.

(b) According to the recommendation of the survey from chapter 2 the most promising languages are used for modelling a basic model. As a result of this section, two of the most suitable languages are chosen for the third part of testing. Source codes of basic model implementation in various modelling languages can be found in appendix D.

(c) A description of each tested language as well as summary of this part of testing can be found in chapter 3, the source codes of the basic model implementation in tested languages can be found in appendix D.

3. Implementation of an extended model.

    - The last step of testing consists of implementing an extended model of a cogeneration system and benchmarking the most promising languages. The choice of languages is based on results from the previous section (e.g. basic model implementation). A detailed description and summary of this part can be found in chapter 4.

4. Summary of retrieved results and a final recommendation are being discussed in chapter 5

1.1 Basic model of a cogeneration system

A cogeneration system has already been mentioned in chapter 1. A more precise definition of cogeneration is the following [25]: *Cogeneration is the combined production of electrical (or mechanical) and useful thermal energy from a single primary energy source.* The mechanical energy produced can be used to drive a turbine or auxiliary equipment such as compressors or pumps while the thermal energy can be used either for heating or cooling. In case of heating, the thermal energy heats water in a district heating system or for industrial applications. Cogeneration systems are also referred to as combined heat and power (CHP) systems [27]. More information about CHP systems can be found for example in [19]. In the rest of this text only CHP systems that generate electrical and thermal energy are considered.

For the purposes of the second part of modelling languages testing a simplified model of a CHP system was used (also refered to as the basic model). This model was inspired by a type of CHP plant that is common both in the Czech and Slovak Republics. The model consists of a set of gas-fuelled boilers, a set of steam turbines, a condenser and a water heater. A scheme of this model is in figure 1.1. Gas-fuelled boilers heat water and turn it into hot and high-pressured steam. Hot steam flows to turbines, where part of the thermal energy of steam is turned into the mechanical energy of turbines. The turbines rotate connected generators, that produce electric power. Cooler low-pressure steam (a part of its thermal energy was turned into mechanical and then electrical energy) outflows from turbines and can be used for heating water in the heating system, or directly cooled in a condenser. No energy loss (apart from the intentional energy loss at the condenser)

is considered and general power and mass balance laws have to be satisfied in the system.

Our aim is to minimize the production cost of the desired amount of electric power [MW] and heat [MW] used for heating households. As satisfying production of both commodities might be unfeasible in certain cases (e.g. large heat demand but low electric power demand) the problem is relaxed allowing certain deviation in el. power production. Every MW of difference between demanded el. power production and total production increases the cost of such production (*contracting penalty*). Positive deviation means that production of el. energy is lower than was planned. Objective function consists of fuel costs of fuel burned by boilers and of a contracting penalty for deviation in electric energy production.

This model has certain features that make it suitable for the testing undertaken, such as:

Absolute value – introduction of the contracting penalty forces us to use deviation in absolute value, otherwise the contracting penalty would make no sense (negative deviation cannot decrease the production cost). Absolute value (further also noted as *abs()*) is a simple non-linear function that is often used during modeling CHP systems. Using absolute value of deviation shows whether a modelling language can handle abs() function with variable as its argument. If not it is necessary to reformulate the function using the following scheme (so as to keep the model linear):

$$\text{instead of } \max |c| \text{ we introduce a new variable } z \text{ such as:} \tag{1.1}$$

$$c \leq z$$

$$c \leq -z$$

Modelling languages that can handle abs() automatically replace it in the same way as mentioned above or use more sophisticated methods.

Non-linear power characteristics of boilers – the gas-fueled boilers used in the model have certain power characteristics (e.g. the relation between power input and output) that are non-linear and have to be linearized using piecewise linear function (PWL). A more detailed description about PWL and its formulation in modelling languages can be found in section 1.3. PWL is important for modelling the linearized characteristics of various pieces of equipment of a CHP system.

The power characteristics of the used boilers with other parameters of the basic model can be found in appendix B.

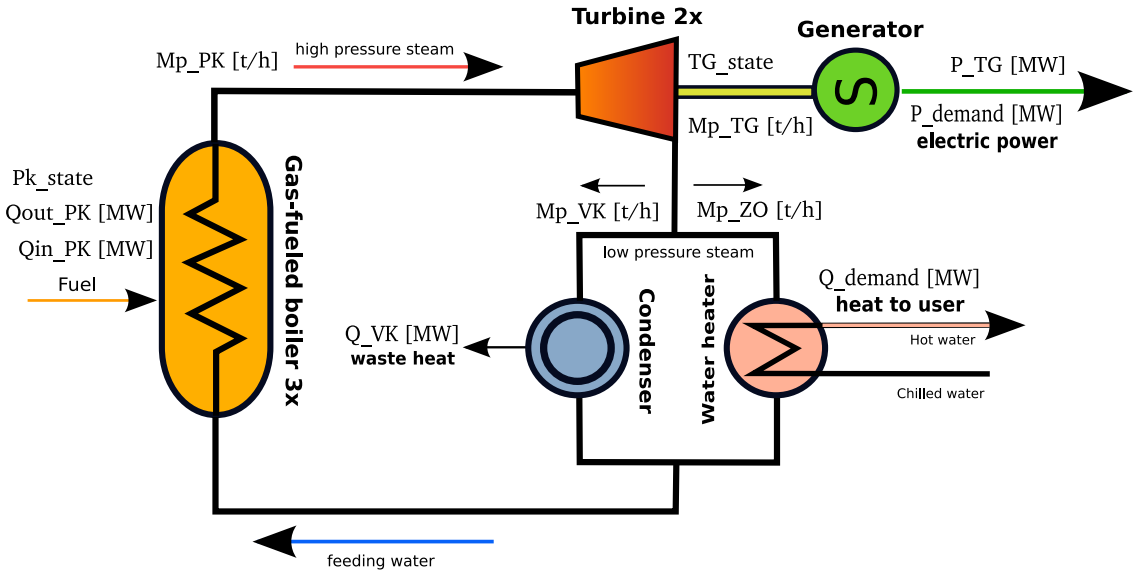

Figure 1.1: Scheme of a simple cogeneration plant

1.2 Mathematical formulation of the model

In this section mathematical equations are introduced describing the basic model of a CHP system from section 1.1.The scheme of the whole model is in figure 1.1 and all parameters and power characteristics of this model can be found in appendix B.

The model contains three gas-fueled boilers and two turbines. The status of boilers (on/off) is described by binary vector $\mathbf{PK}^{state}$. The power input of boilers is represented by vector $\mathbf{Q}_{in}^{PK}$, power output by $\mathbf{Q}_{out}^{PK}$. Parameter m represents the number of boilers (in this case $m = 3$), parameter n represents the number of steam turbines (here $n = 2$). Binary vector $\mathbf{TG}^{state}$ describes the status of turbines. A steam turbine can operate only in limited operation range defined by minimal and maximal possible steam flow. In certain cases it might be more convenient to keep only one turbine operational with higher steam inflow, rather than using both turbines with lower inflow – this would lead to savings on starting costs and to less wear of turbines in a real system, however this effect isn't modelled in this basic model. Energy transformation from thermal energy of hot steam to electrical energy is described by *enthalpy*. Enthalpy [kJ/kg] describes thermal energy of unitary mass [30].

The model contains the following constraints:

1. Boilers heat feeding water to high-pressure steam. The mass of water at the inflow

of boilers is equal to the mass at the outflow of boilers so as to satisfy mass balance.

$$\sum_{i=1}^{n} Mp_i^{PK} = \sum_{j=1}^{m} Mp_j^{TG} \quad [\text{t/h}]$$

2. Steam outflow from turbines is equal to steam inflow to condenser plus steam inflow to water heater. An unlimited maximal flow in the condenser and in the water heater is assumed (minimal flow is zero).

$$\sum_{j=1}^{m} Mp_j^{TG} = Mp^{VK} + Mp^{ZO} \quad [\text{t/h}]$$

3. Each turbine has its minimal and maximal allowed mass flow rate.

$$Mp_j^{TGmin} TG_j^{state} \leq Mp_j^{TG} \leq Mp_j^{TGmax} TG_j^{state} \quad [\text{t/h}]$$

4. The el. energy produced by each turbine is equal to the difference in enthalpy of steam between at the inflow and outflow of turbines. It is a simplified formulation as in fact in turbines thermal energy from inflowing high-pressure steam is turned into mechanical energy (rotary movement of turbines) and then into rotary movement of the generators that produce el. power. Cooler outflowing low-pressure steam is used for heating water in a water heater. In order to keep the same units (MW) on both sides of the equation, division by time (hours) is needed.

$$\sum_{j=1}^{m} P_j^{TG} = \left(\frac{i_{out}^{PK} - i_{out}^{TG}}{3600} \right) \sum_{j=1}^{m} Mp_j^{TG} \quad [\text{MW}]$$

5. Heat produced at water heater has to satisfy the production demanded. It means the difference in enthalpy of steam at the inflow and outflow of the water heater multiplied by the amount of steam is equal to the heat production demanded.

$$Q_{demand} = Mp^{ZO} \left(\frac{i_{out}^{TG} - i_{in}^{PK}}{3600} \right) \quad [\text{MW}]$$

6. The el. power production demanded minus deviation in production is equal to produced el. power. Apart from the positive deviation (lower production) discussed in chapter 1.1 even negative deviation (higher production) can occur. In certain cases using a more powerful (but more effective) boiler leads to lower production cost (even with higher deviation) than satisfying the production requirements with no deviation.

$$P_{demand} = dev + \sum_{j=1}^{m} P_j^{TG} \quad [\text{MW}]$$

7. The Power output of the boiler has to be sufficient for turning feeding water into steam, which is again described using the enthalpy difference of water/steam at the inflow and outflow of boilers.

$$\sum_{k=1}^{n} Q_{out_k}^{PK} = Mp^{PK} \left(\frac{i_{out}^{PK} - i_{in}^{PK}}{3600} \right) \quad [\text{MW}]$$

All variables except from possible deviation have to be non-negative and continuous (apart from state variables that are binary). Our aim is to minimize the production cost of the desired amount of electric power and heat, hence the objective function consists of fuel costs of fuel burned by boilers and of a contracting penalty for deviation in electric energy production as follows ($\mathbf{Q}_{in}^{PK}$ is deducted from the boilers power characteristics) :

$$\min : \quad c_{dev}|dev| + c_{fuel} \sum_{k=1}^{n} Q_{in_k}^{PK}$$

1.3 Piecewise linear functions

As was mentioned in the previous chapter, piecewise linear functions (PWL) are often used for substituting non-linear functions. In modelling and scheduling of CHP systems it is usually necessary to substitute the non-linear power characteristics of boilers, turbines and other parts of the system in order to keep the model linear. A sample linearized power characteristic of a boiler is shown in figure 1.2. Function P stands for power input, $P_1 \dots P_5$ are called *breakpoints* of the function. Function F is the power output of the boiler.

Although it is possible to implement PWL using auxiliary binary variables and function's angular coefficients (as can be seen in basic model implementation in Yalmip modelling language in chapter 3, the source code of the model is in appendix D.1) the more convenient method involves using *Special-ordered sets* type 2 (SOS2). SOS2 is a set of consecutive variables in which no more than two adjacent members may be non-zero in a feasible solution [13]. Implementation of PWL using SOS2 is as follows. Firstly, we define functions F and P as (l stays for number of breakpoints):

$$\lambda_k \in \mathbb{R}^+$$

$$P = \sum_{k=1}^{l} \lambda_k P_k$$

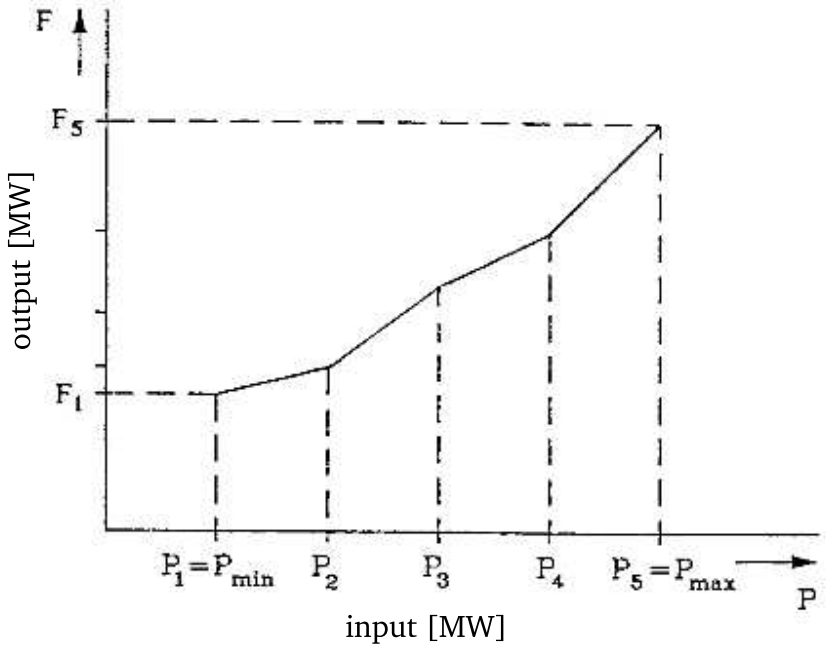

Figure 1.2: Linearized power characteristic of a boiler [26]

$$F = \sum_{k=1}^{l} \lambda_k F_k$$

Secondly, the power output of the boiler is relevant only if the boiler is switched on, thus an additional constraint is introduced (PK^{state} is a binary variable that holds 1 if the boiler is turned on):

$$\sum_{k=1}^{l} \lambda_k - PK^{state} = 0$$

The last necessary equation is $\lambda_k \in$ SOS 2. It means that at most two adjacent variables λ_k may be non-zero. This formulation can be used for formulating any piecewise linear function no matter whether it is convex or concave [26].

It is important to mention that Special Ordered Sets type 1 (SOS1) also exist. SOS1 is a set of variables in which no more than one member from the set may be non-zero in a feasible solution. SOS1 are typically used for representing non-linear functions or for modelling cases where it is necessary to choose "one of many values" (e.g. choosing in which month production should start etc.) [13]. In the basic model no SOS1 is used.

Chapter 2

Modelling languages survey

This chapter introduces the modelling languages survey which represents the first and the most general part of the testing. The aim of this survey is to provide a basic overview of available modelling languages and their features. This overview helps us to determine the most promising languages for further testing. References and sources of this survey are mentioned in chapter 2.1. In chapter 2.2 the investigated features of modelling languages are described. Finally, the results of the survey and the languages suitable for further evaluation are discussed in chapter 2.3. The full overview of tested languages can be found in appendix A.

2.1 The initial survey and other resources

Firstly, a literature and Internet research of articles comparing modelling languages was done. Similar comparisons of modelling languages available at that time (such as [22, 28]) were generally out of date, thus irrelevant for this survey. However, a survey of software for linear programming from Robert Fourer from June 2009 provided a good foundation for our work [21]. Although the information in that survey was provided by software vendors responding to a questionnaire and hence had to be verified, it served as a general overview of available modelling languages. In some cases user manuals of the languages and vendor's web pages were used as sources of additional information. Features of the tested modelling languages on which the survey is focused on are described in the following section.

2.2 Features investigated

The survey was divided into four parts and in each one a different set of features is investigated (these parts are called: **General information, Inputs/Outputs, Price and licensing, Problem formulation**). A description of each part of the survey as well as the modelling language features important for further use in a real business environment (as was mentioned in chapter 1) is provided in the next sections. If certain features aren't mentioned in the final overview (which can be found in appendix A), it is either because they were no longer relevant (e.g. if a modelling language is open-source, it makes no sense talking about the possibility of obtaining a floating licence – see following sections for details) or it wasn't possible to find the appropriate information.

General information

In this part a modelling language platform availability and type of application is investigated. Only availability for MS Windows and Linux/Unix operating systems (OS) were considered, as these OS are the most common environments for solving optimization problems. Mac OS X wasn't taken into account. For performance reasons even the availability for 64-bit systems is considered.

A modelling language that is available as a stand-alone application is distributed as an Integrated Development Environment (IDE) or as a command line application. On the other hand a callable library provides an Application Interface (API) which allows the language to be used in an external program.

A modelling language suitable for an application capable of modelling and optimal scheduling of CHP systems has to be available for MS Windows OS as in real operation mostly this OS is used. A stand-alone IDE isn't necessary as the selected modelling language will be a part of the larger scheduling application (preferably writen in Java). For the same reason a callable library is required (as it provides an easy connectivity of the modelling language and other environments).

Inputs/Outputs

The second part of the survey is focused on data import and data exchange abilities of tested languages. This part investigates three different criteria:

1. **Data input/output related to MILP problem formulation** – whether the modelling language is able to read/save models stored in MPS or LP file format.

2. **Loading model parameters from an external file** – such as a spreadsheet, database, or plain text file.

3. **Connectivity with solvers** – some modelling languages offers their unique solvers (if the modelling language contains modelling environment and a solver), whereas others directly link by embedded application interface (API) with certain solvers only (and can usually access other solvers using MPS/LP files).

A preferred modelling language has to be able to load data from spreadsheets (the reasons for it were explained in chapter 1), and has to be directly linked with two common commercial solvers (CPLEX [15], Gurobi [5]), as these solvers are most likely to be used for scheduling a real system thanks to their performance.

Price and licensing information

The third part of the survey provides information about price and licences of modelling languages. Three different types of licence are usually offered by vendors: *commercial* (for commercial use, the most expensive one), *academic* (for academic and testing purposes only, it offers full functionality) and *demo licence* (usually free but with a limited maximal number of variables). Sometimes also *floating licences* (which permit a certain number of copies to be used anywhere in a large network – e.g a university) and *site licences* (they are unrestricted by the number of copies in a specific location – e.g. a part of a company) are offered.

In some cases the price list is published online, in other cases a request at the vendor's sales department is required, then the price is set on an individual basis. The price of the licence also depends on solvers shipped with the modelling language, the number of CPU's at the licensed machine and other factors.

NEOS Server for Optimization is a free web page where MILP problems can be solved with no size restrictions [12]. Users can choose amongst various solvers and upload their problem using an appropriate file format (usually MPS/LP file is supported). *NEOS Server availability* means that it is possible to solve a MILP problem stored in a native file format of the modelling language (e.g. no export to MPS file is needed). This ability can be useful for testing, as it is free and with no size restriction.

Problem formulation

The features described in this part are helpful for easier model implementation (a user can use simpler formulations in the process of the model creation). The features are as follows:

Special Ordered Sets type 1 (SOS1) – the modelling language is capable of formulating SOS1 (more information about SOS1 can be found in chapter 1.3.

Special Ordered Sets type 2 (SOS2) – the modelling language is capable of formulating SOS2 (more information about SOS2 can be found in chapter 1.3.

Branching priorities of binary variables – if branching priorities of binary variables (which for example are representing boiler status) are set manually (using our categorical knowledge of the system) the computation time can be significantly decreased. Setting up these priorities usually depends on the solver.

Non-linear functions handling – the modelling language can reformulate an absolute value of a variable, minimum and maximum functions with decision variables as their argument, that are used in objective function (or in constraints) and keep the model linear, as these functions are often used in modelling of CHP systems.

The preferred features of the desired modelling language are the formulating of special ordered sets type 2 (which are typically used for representing piecewise linear functions as was mentioned in chapter 1.1), and setting up branching priorities of binary variables (as it might accelerate the computation time). Absolute value can be manually reformulated into a linear form if necessary.

2.3 Modeling languages for further evaluation

In the previous sections of this chapter the modelling languages survey, which serves as the first part of the testing procces, was presented. The sources of the survey were described and each part of the survey was explained, as well as preferred features which a suitable modelling language fulfills. An overview of all compared languages can be found in appendix A. Finally a list of modelling languages recommended for further evaluation, held in the next section, is created.

A suitable modelling language should fulfill the following criteria (a detailed descripton of these features is given in the previous section). The criteria are sorted according to their significance (the more important one goes first):

1. Available for MS Windows.
2. A callable library/API that adds connectivity with other languages/environments.
3. Reading/Saving data from/to spreadsheets.
4. Direct link to Gurobi and CPLEX solvers.
5. Special Ordered Sets type 2 that make formulating piecewise linear functions easier.
6. Branching priorities of binary variables can be manually set.

Modelling language	**MS Windows**	**Callable library**	**Spreadsheets**	**Gurobi/ CPLEX**	**SOS2**	**Branching priorities**
AIMMS	X	X	X	X	X	X
AMPL	X	–	X	X	X	X
GAMS	X	X	X	X	X	X
Gurobi API	X	X	–	/	X	/
LINGO	X	X	X	–	X	/
MPL	X	X	X	–	X	X
OptimJ	X	X	–	X	X	/
Yalmip	X	X	X	X	–	–
Zimpl	X	X	–	–	X	/

Table 2.1: A comparison of the modelling languages recommended for the basic model implementation

A basic comparison of the most suitable languages and their features can be found in table 2.1 ("X" – means that the language supports the feature, "/" – means that the language supports the feature with certain limits, "–" – means no support). As only two languages (AIMMS and GAMS) provides all required features, the requirements were relaxed and all of the languages shown in table 2.1 are recommended for further evaluation. A short comment about additional features that conviced us to recommend the language for further testing is given below:

AIMMS – a modelling language with a sophisticated IDE with a well-developed user-friendly interface. Also fulfils all criteria.

AMPL – well-known and widely-used language that meets almost all requirements.

GAMS – another widely-used and well-known language with an IDE, fulfils all criteria.

Gurobi API – an application interface of a solver, chosen to demonstrate whether it is possible to avoid usage of a modelling language and model the CHP system directly using solver API.

LINGO – a language with an IDE with its own solvers. However, apart from the direct link with CPLEX/Gurobi it meets all requirements.

MPL – a language similar to AMPL, but with its own IDE.

OptimJ – a Java-based modelling language shipped as an extension to Eclipse IDE, promising a very good interaction with Java applications.

Yalmip – a toolbox for Matlab, meets almost all requirements (apart from SOS2 formulation). Currently used for modelling CHP systems in ongoing projects at the Department of Control Engineering.

Zimpl – an open-source solution, included in order to compare commercial and open-source languages.

In the next section each of these selected languages is further evaluated by implementing a basic model of a CHP system.

Chapter 3

Basic model implementation

The aim of this section is to implement a basic model in each of the selected modelling languages, evaluate the models and recommend the most suitable modelling languages for the final part of the testing, which consists of the implementation of an extended model and execution time benchmarking. This final part of testing is described in chapter 4.

Particular features of the languages were closely investigated during the model implementation and are presented in following paragraphs. Each language is described separately with important snippets of code included in the text. The source code of the implemented models can be found in appendix D, and an evaluation of tested languages is provided at the end of this chapter.

3.1 Features investigated

The abilities and features tested are as follows:

Model export and Java connectivity

In this section supported file formats for exporting a model are mentioned. The most common is the MPS/LP file format. The possible linking of the language with Java applications is also questioned here.

Data import/export and MS Excel connectivity

An ability to read model parameters from an external file (especially MS Excel spreadsheets) is important for a real application, as was discussed in chapter 1. Possible ways of importing and exporting model parameters are presented in this section.

Exporting the solution

The process of modelling and optimizing the scheduling of CHP systems requires showing the result of optimization in a user-friendly way (as was presented in chapter 1). Hence if the modelling language is capable of calling an appropriate solver to solve the MILP model, it is logical to investigate its ability to show and export the solution.

Declaration of variables

The languages tested usually provide two ways of declaration. Firstly, variables can be declared anywhere in the code of the model (in the following text just *code* is used) or secondly, they have to be put into the appropriate part of the model. Declaration of parameters is also mentioned in this section.

Declaration of constraints

In a large scale model a way of declaring variables and equations significantly affects the code clarity. An example of declaration of the same constraints in different language is provided. The tested languages offer three different techniques of declaration (both variables and constraints):

- **Programming style** – syntax of the modelling language is similar to any of the programming languages available (e.g. Java). Firstly, a type of the variable has to be set, then the name and optionally a range of the variable. Vectors and matrices are represented using one, two or more dimensional arrays. In case we need to sequentially access elements of an array it is necessary to use *for* cycles. A language that represents this style of declaration is OptimJ (see chapter 3.4 for examples and more information).

- **Matrix-oriented style** – or more familiarly the *Matlab* style. Variables can be declared anywhere in the code, and their type has to be specified. However, matrix multiplication and other operations can be applied, which leads to a very economic code. A typical representation of this style is Yalmip (examples and additional information can be found in chapter 3.2).

- **Set-oriented style** – the last style of declaration is the most common amongst the tested languages. It usually requires variables to be declared in a specific part of the model, their type and a range has to be specified. The basic data structure is a set with its elements. Apart from the matrix-oriented style where sets are represented as vectors, here sets are closer to their mathematical definition. It means a set is a bunch of objects (either ordered or unordered) on which a certain operation can be executed. Although at first sight the set-oriented style looks similar to the matrix-oriented style, it is more suitable for formulating optimization problems, as the set-oriented style allows more intuitive transcription of mathematical equations into the model. This declaration style is used for example by GAMS (which is described in chapter 3.3 with examples).

Debugging and code clarity

An integral part of the model implementation is code debugging, as both syntax and functional errors can occur. Various debugging features of the languages are described in this section. Clarity of the final code is also discussed. Evaluation of these features is strongly subjective and depends on the user's experience, hence all implemented models are shown in appendix D for those with deeper interest.

Setting up solver options

In order to push down the computation time of the solution, a change in default solver settings can speed up the computation. For example, setting up an optimality gap, branching priorities of variables or only the solver verbosity is in the aim of this section. Previously mentioned branching priorities of binary variables (chapter 2.2) usually depend on the chosen solver, but the modelling language can help to set these priorities.

Non-linear functions and special ordered sets

In this section we will describe a language ability for formulating and linearizing absolute value and other simple non-linear functions (such as *min* and *max*). Also formulation of a piecewise linear function using special ordered sets type 2 (as was explained in chapter 1.3) is questioned.

Price and licensing

The vendor's licensing options have been already mentioned in chapter 2. For a real application of the language only a commercial license is relevant. The price of the license varies on the used solver and other parameters, and the up-to-date price list might be different. Usually the price of the basic licence is mentioned for easier comparison.

Summary

In previous paragraphs a description of features investigated during implementation of the basic model was given. At the end of each of the following sections a brief evaluation of results is made. Advantages and disadvantages of the selected language are mentioned, as well as possible recommendations. The sections are sorted in the same order as the languages were tested.

3.2 Yalmip

Yalmip is a modelling language for advanced modelling and solution of convex and non-convex optimization problems [23]. Yalmip is an open-source toolbox for Matlab, which uses all advantages of the Matlab environment.

Model import/export and Java connectivity

Yalmip by default doesn't provide export of the model into MPS/LP file format. A model created in Yalmip can be exported to AMPL model file using *saveampl* function. However, this function works for simple models only [23].

Although Matlab is based on Java and using Java classes in Maltab environment is possible, calling Matlab commands from Java applications is not supported. A workaround currently exists (for details see [7]), but it is a commercial solution that increases the cost of the final modelling application.

Data import/export and MS Excel connectivity

Yalmip variables are accessible in the same way as any other Matlab variable, so it is possible to implement a user's function that handles loading and/or saving parameters of the model or use a suitable Matlab function. Spreadsheets can easily be accessed using Matlab's XLS handler. Data from the spreadsheet can be loaded as simply as follows:

```
% Loading parameters from MS Excel spreadsheet
num = xlsread('model_params.xls');
```

Exporting the solution

Exporting and showing the solution is similar to the data export mentioned before. Matlab or user-defined functions can be used. An example of such showing a solution is here (J stands for the objective function):

```
%%---- SHOW RESULTS ----%%
res.dev = double(dev); % planned deviation [MW]
res.costs = double(J); % total costs of production [CZK/h])
res % showing the results
```

It is possible to change solver verbosity (e.g. how much information is to be printed to the console) using a *sdpsettings* function that can update solver settings. Yalmip is directly linked to a large number of solvers; however both for CPLEX and GUROBI solver require a MEX-interface (MATLAB executable). MEX-files (interfaces) are dynamically linked subroutines produced from C, C++ or Fortran source codes that, when compiled, can be run from within MATLAB in the same way as MATLAB M-files or built-in functions [16]. MEX-interfaces have to be compiled individually for each combination of the Matlab version and an operating system.

Declaration of variables

Yalmip syntax (identical to Matlab syntax, thus anyone familiar with Matlab can start modelling with Yalmip straight away) allows the user to declare variables and parameters anywhere in the code. Yalmip uses a "matrix-oriented style" of declaration as was mentioned in the introduction of this chapter. In the following example a binary vector TG^{state} is declared:

```
% Turbine status
TG_state = binvar(2,1,'full');
```

Default bounds of variables are set according to their type (e.g. binary, integer, continuous). Bounds can be reduced using additional constraints, such as:

```
% Seting non-negative variables:
% Steam generated by boilers [t/h] has to be greater than zero.
Mp_PK>=0;
```

Declaration of constraints

As is natural for the Matlab environment, matrix and vector manipulation is very easy and allows the user to write an economic code. A short example from the basic model becomes handy: when we need to formulate a constraint for each element of a vector (e.g. the third equation – set minimal allowed flow through turbines, see chapter 1.2 for details) no *for* loop is needed, only to naturally write the equation (F stands for a set of constraints, in which all constraints are grouped):

```
% Steam flow in turbines [t/h]
Mp_TG = sdpvar(2,1,'full');


% 3. Minimal allowed flow through turbines
F = F+ [Mp_TG_min.*TG_state <= Mp_TG <= Mp_TG_max.*TG_state];
```

Yalmip can handle double inequality in one constraint, so the equation above doesn't have to be split in two (e.g. $Mp_j^{TGmin}TG_j^{state} \leq Mp_j^{TG}$ and $Mp_j^{TG} \leq Mp_j^{TGmax}TG_j^{state}$). This feature allows the user to write a very economical code. However, if the code isn't well commented it might become harder to understand as there is no index of variables. As a short example compare the previous equations with an equivalent formulation:

```
% 3. Minimal flow allowed through turbines
for i=1:size(TG_state,2)
    F=F+[Mp_TG_min(i)*TG_state(i) <= Mp_TG(i) <= Mp_TG_max(i)*TG_state(i)];
end
```

The objective function is as follows. As can be seen, the *sum* function can be used with no additional parameters.

```
%%---- OBJECTIVE FUNCTION ----%%
J = fuel_cost*sum(sum(Qin_PK)) + deviation_cost*abs(dev);
```

Debugging and code clarity

A complex set of Matlab debugging tools can be used, such as *profiler* for viewing code execution time of code and *debugger* where breakpoints can be set, and the code can be examined "step-by-step." Yalmip itself provides a good documentation and tutorials, which can help to fix errors. Debugging of the Yalmip model is the same as the debugging of any other Matlab script.

Code clarity is at a high rate thanks to the Matlab matrix-oriented syntax. However, the code needs to be well commented.

Setting up solver options

Solver options can be set using function *sdpsettings* and then using the options structure while calling a solver (*solvesdp* function). A sample use of *sdpsettings* follows. Parameters that can be modified depend on the solver used.

```
% Changing solver settings
options = sdpsettings('field',value,'field',value,...)
solvesdp(Constraints, Objective, options)
```

Non-linear functions and special ordered sets

Yalmip can linearize absolute value, minimum and maximum functions. The linearization is done through the *big-M* reformulation and increases the number of variables in the model [23].

On the other hand Yalmip doesn't support special ordered sets type 2 (SOS2), thus a piecewise linear function has to be implemented using auxiliary binary variables and angular coefficients as is shown in this example, where the power characteristics of boilers are formulated. Implemented PWLs have three breakpoints and are presented in appendix B.

```
%%-- Boiler constraints --%%
% an auxiliary variable
PK_regions = binvar(3,3,'full'); % row = boiler, column = section

% Input and ouput connection
% 1. Power output has to be in one section only
Qin_PK_char(:,1:end-1).*PK_regions<=Qin_PK;
Qin_PK<=Qin_PK_char(:,2:end).*PK_regions;

% 2. Power output is set by lines with angular coeficients
coeff = diff(Qout_PK_char,1,2)./diff(Qin_PK_char,1,2);
ofset = Qout_PK_char(:,1:end-1) - coeff.*Qin_PK_char(:,1:end-1);
Qout_PK == coeff.*Qin_PK + ofset.*PK_regions;
```

Price and licensing

Although Yalmip is an open-source product it may not be re-distributed as part of a commercial product [23]. Apart from that it requires the Matlab environment for running. The Matlab commercial licence is being sold for € 1,750,- for one licenced computer or user. An appropriate solver has to be bought separately. MathWorks (Matlab vendor) offers both individual and floating licenses. However due to Yalmip licence limitation its commercial use is cumbersome.

Summary

Yalmip is a versatile modelling language that gains many benefits from its connection with Matlab, such as effective work with vectors and matrices, spreadsheets connectivity and an outstanding IDE. However its inseparable bond with Matlab, complicated connection

with Java and restricted commercial use don't make this language the most suitable for the real modelling application.

3.3 GAMS

The acronym GAMS stands for The General Algebraic Modelling System. It is a high-level modelling system for mathematical programming and optimization. It consists of a language compiler and a set of integrated high-performance solvers [14].

Model export and Java connectivity

The GAMS model can be exported in various formats (e.g. MPS, LP, LINGO, AMPL) using the *convert* utility. It is run like any other GAMS solver from the command line using the following command (the type of exported file has to be specified within the model file):

```
>> gams modelname modeltype=convert
```

GAMS commands can be called from Java applications using *Runtime* class and *Exec()* method, however the model has to be created separately in that case. When calling GAMS, a working and a scratch directory (for temporary files) has to be set:

```
// call gams
String[] cmdArray = new String[5];
cmdArray[0] = "C:\\Program Files\\GAMS\\20.5\\gams.exe";
cmdArray[1] = "D:\\TMP\\gams_model.gms";
cmdArray[2] = "WDIR=D:\\TMP";
cmdArray[3] = "SCRDIR=D:\\TMP";
cmdArray[4] = "LO=2";
Process p = Runtime.getRuntime().exec(cmdArray);
p.waitFor();
```

Data import/export and MS Excel connectivity

Exchange of data files is provided through GAMS Data Exchange (GDX) facilities and files. GDX files are binary files that are portable between different platforms. For example loading data from an MS Excel spreadsheet can be done as follows (it is more complicated than in Yalmip):

```
$CALL GDXXRW.EXE model_params.xls par=TG_min rng=A1:C3
```

The parameter loaded is called TG^{min}, the argument "A1:C3" specifies the range of cells. However, the parameter has to be declared before the load statement:

```
Parameter TG_min(i);
  $GDXIN results.gdx
  $LOAD TG_min
  $GDXIN
```

GAMS is also capable of reading CSV (comma-separated values) files. However, in all cases the GDX facilities have to used for data import/export.

Exporting the solution

When the model is executed, a log file and a solution file are created. The solution file containts model statistics, details about execution time, solver output and the final solution. However, the contents of the solution file can be specified within the model. The solution can be exported using GDX facilities. A sample solution file viewed from the GAMS IDE is shown in figure 3.1

GAMS offers links with various solvers (such as Gurobi and CPLEX), the selection of the solvers actually linked depends on the licence obtained.

Declaration of variables

Unlike Yalmip, GAMS uses "set-oriented" notation (as was mentioned in chapter 3.1). It means that the most important are sets of elements (for example a set of turbines) and the declaration of variables and constraints strongly depends on these sets (e.g. if a new element is added in the set, there is no need to write additional equations). The work with sets is similar to the work with indexes, but allows us to name the elements, which

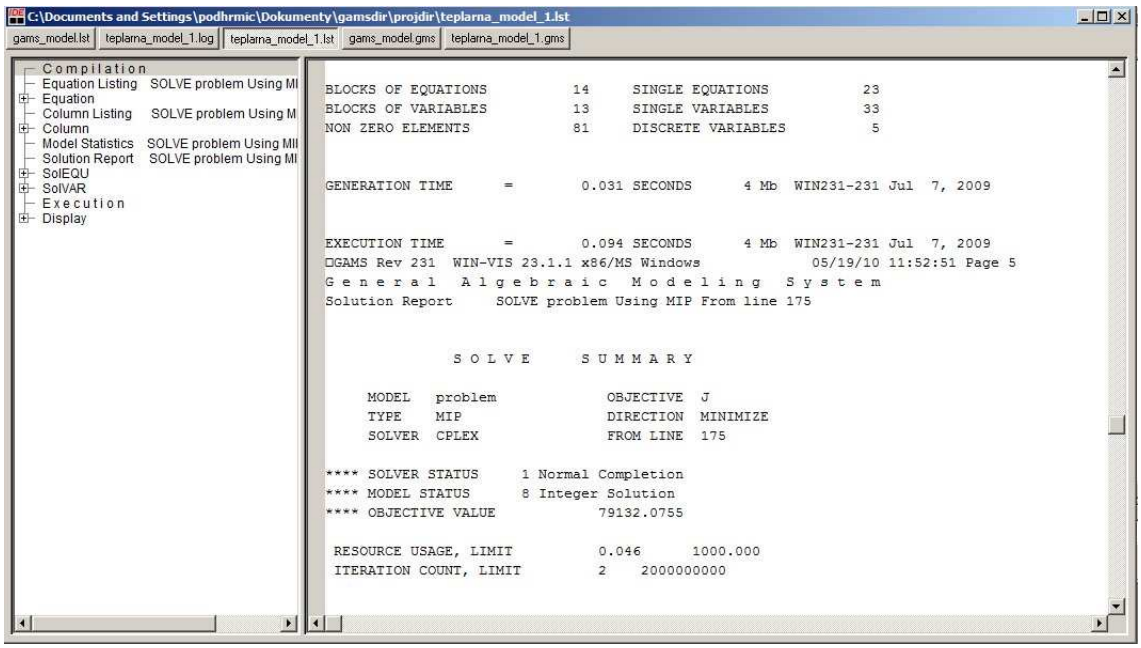

Figure 3.1: Solution file of a GAMS model

makes the problem formulation more natural. For an explanation look at the following example:

```
* The set has to be declared first
* Turbine status
Sets
    TG turbines /TG1, TG2/;

* Then a binary variable representing
* turbine status is introduced:
Binary variables
   TG_state(TG)  turbine status;
```

At first sight we see that the set of turbines contains two elements, a turbine called *TG1* and a turbine *TG2*. Further use of sets in constraints is discussed in the next paragraph. Variables and parameters have to be declared in a specific part of the model, initiated by *Variables* and *Parameters* keyword. GAMS also makes a difference between scalar parameters and matrix parameters (such as a table). Bounds of variables depends on their type (binary, integer, positive, continuous), but can be specified manually in the *Equations* section of the model.

```
* Seting up a non-negative variable
* Steam generated by boilers [t/h]
* has to be greater than zero
Positive variables
    Mp_PK Steam generated by boilers [t per h]
    Mp_TG(TG) Steam flow through turbines [t per h];

* If an upper limit is needed we write:
* MAXLIMIT is a parameter.
Mp_PK.up = MAXLIMIT;
```

Declaration of constraints

The constraints and the objective function have to be declared in the *Equations* section. Each constraint has its name and can be briefly described in the beginning of the section, increasing code clarity. In GAMS no *for* loops are needed, the user only has to specify which set is related to the constraint and GAMS automatically do the rest. See the following example (TG refers to a set of turbines and symbols `=l=, =g=, =e=` refer to $\leq$, $\geq$, $=$ operators):

```
* 3. Minimal and maximal flow through turbines allowed
  constraint3(TG).. Mp_TG_min(TG)*TG_state(TG) =l= Mp_TG(TG);
  constraint4(TG).. Mp_TG(Tg) =l= Mp_TG_max(TG)*TG_state(TG);
```

As we can see, GAMS doesn't support double inequalities, thus the original equation had to be split in two. When summing up variables, a set has to be specified so as to declare which elements are summed up. For example, in the objective function from the basic model a sum of boilers input is required:

```
***** Objective function ****
 costs.. J =e= deviation_cost*v_dev + fuel_cost*sum(PK,Qin_PK(PK));
```

Debugging and code clarity

GAMS is shipped with a simple IDE, which is satisfying for the model implementation and debugging, as can be seen in the following figures. Debugging can be made using

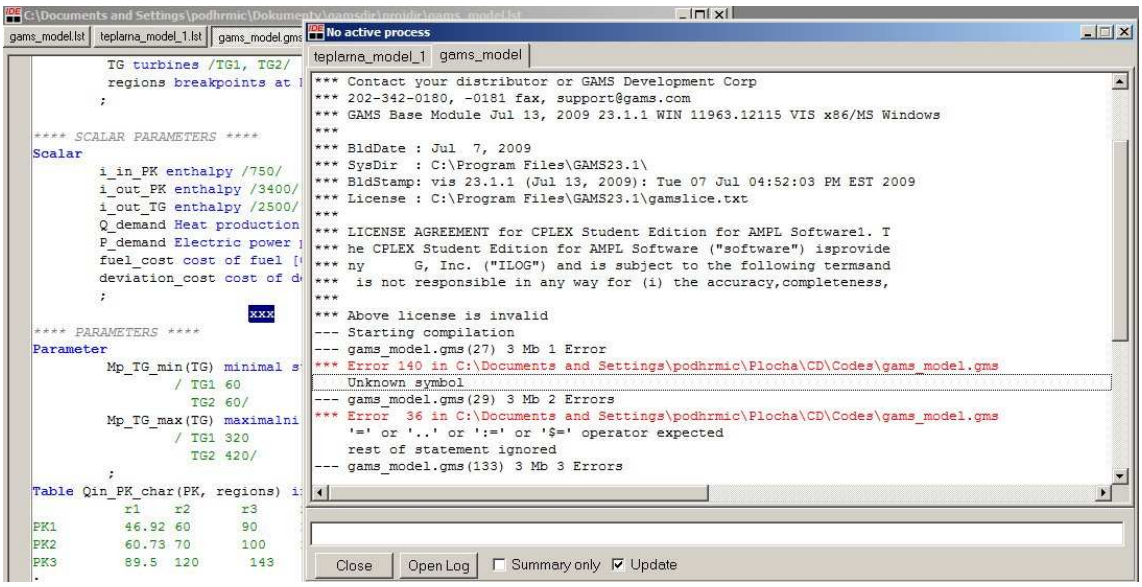

Figure 3.2: A syntax error reported by GAMS IDE

the log file in which all errors are reported. In the GAMS IDE the line of the code where the error occured is marked. A syntax error is shown in figure 3.2. GAMS is a well-documented modelling language, even the IDE contains well organized help topics (as can be seen in figure 3.3).

Due to the "set-oriented" syntax and strict sectioning of the model (e.g. parameters, sets, variables and equations are declared separately) the code is clear and easy to read, as can be seen in appendix D.2.

Setting up solver options

GAMS IDE provides an integrated Option Editor (shown in figure 3.4) where options for different solvers can be set. At the end the option file can be saved and then loaded during the model execution. Another possibility is to specify the options using the *option* command:

```
* create an instance of the problem
Model problem /all/;
* Specify CPLEX as the desired solver
Option MIP = Cplex;
* Copy CPLEX messages to the solution file
```

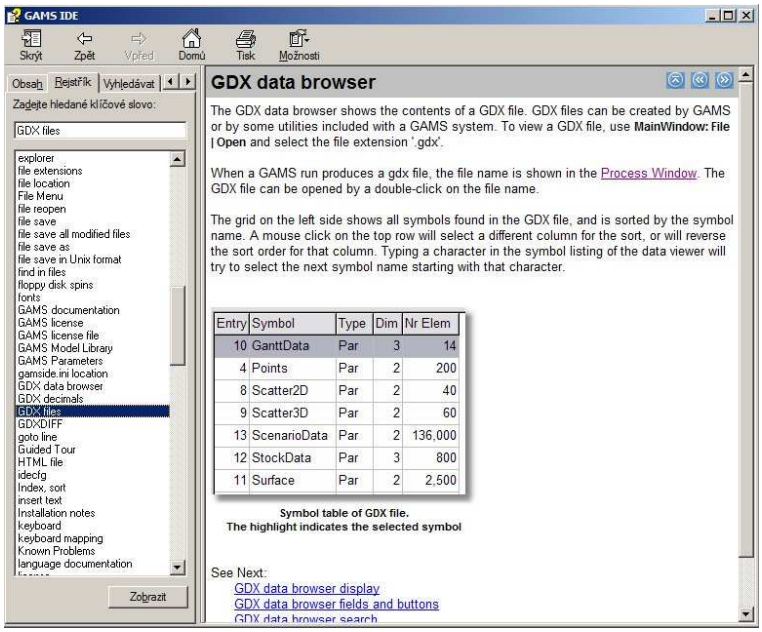

Figure 3.3: GAMS IDE Help topics

```
Option SysOut = On;
* Cplex will read an option file called cplex.opt
problem.OptFile = 1;
```

Non-linear functions and special ordered sets

GAMS doesn't reformulate the non-linear functions, so absolute value and other desired functions need to be linearized by the user. The abs() function had to be reformulated in the way mentioned inchapter 1.1. On the other hand, GAMS supports a declaration of SOS1 and SOS2 variables, thus formulation of PWL is very simple, as can be seen in the following example (the parameter `regions` represents number of breakpoints in boilers power characteristics):

```
** Boiler constraints - using SOS2 **
* first declare the auxiliary variable which belongs to SOS2
SOS2 Variable w;

* Power input definition
constraint8(PK)..Qin_PK(PK)=e=
```

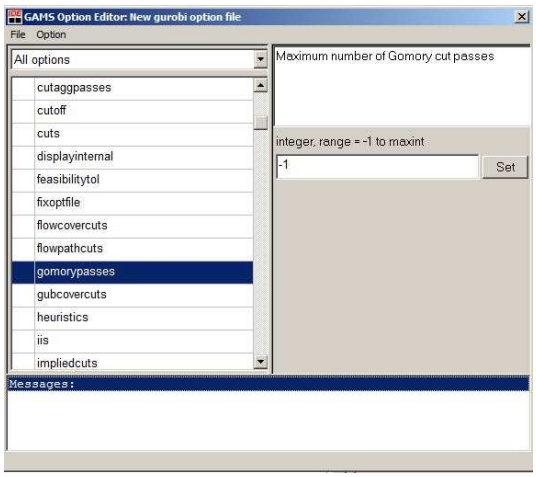

Figure 3.4: GAMS Option Editor editing Gurobi option file

```
     sum(regions,(w(PK,regions)*Qin_PK_char(PK,regions)));

* Power output definition
constraint9(PK)..Qout_PK(PK)=e=
     sum(regions,(w(PK,regions)*Qout_PK_char(PK,regions)));

* Output and input is non-zero only if the boiler is turned on
constraint10(PK)..PK_state(PK) =e= sum(regions, w(PK,regions));
```

Price and licensing

Basic GAMS module for commercial use costs € 2,500,- and solver links (e.g. no license for solvers included, require an appropriate callable library license) is being sold for additional € 2,500,- (CPLEX or Gurobi link), according to the price list from November 2009 [14]. Floating and site licenses are also available.

Summary

In comparison with Yalmip, GAMS offers a simple IDE with basic functionality only and limited debugging options. Also GDX facilities for importing/exporting data and Java connectivity are limiting. On the other hand, a "set-oriented" syntax, clear and legible code and easy setting of solver parameters make GAMS a promising modelling language.

3.4 OptimJ

OptimJ is a modelling language developed by Ateji and combines the advantages of mathematical language and object-oriented programming. It extends Java language and allows users to create models directly in a Java application. An OptimJ model interacts directly with any Java-based application, without the need for any interface code. All Java APIs, whether standard or home-grown, can be used directly in an OptimJ model [18]. OptimJ is distributed as a plug-in into Eclipse IDE [3].

Model export and Java connectivity

OptimJ works as code compiler, which translates OptimJ model file into pure Java source code. As a result, OptimJ models and Java classes can coexist in the same project, thus the Java connectivity couldn't be better as OptimJ is "part of" Java. However, an export of the OptimJ model to an LP/MPS format can be done. An example of outputting the model into MPS file is the following (mps solver is an integrated OptimJ solver capable of exporting model files) [18]:

```
model MPSModel solver mps
{
  // decision variables and constraints go here
  /* This method outputs the model into
  *  a standard text format. The FileWriter
  *  must be opened and closed by the caller.
  */
  static void writeModel(FileWriter out) throws IOException
  {
    // instanciate the model
    MPSModel myModel = new MPSModel();
    // extract it
    myModel.extract();
    // output it
    out.write(myModel.solver().toString());
  }
}
```

Should the user prefer the LP file to be exported, only a change from "solver mps" to "solver lp" is needed.

Data import/export and MS Excel connectivity

OptimJ doesn't provide any special tools for importing or exporting data. On the contrary if the user implements his own Java method (or use any of the available Java libraries) various file formats can be accessed. For example for connecting OptimJ models and Java applications with MS Excel files it is possible to use the *HSSF-API* library, which provides a set of functions for manipulating spreadsheets [6]. The final application looks as follows (low-level methods are not shown, but the whole Eclipse project can be found on the enclosed CD):

```
/*---- MAIN METHOD ----*/
public static void main (String []args) {
    // create an instance of the model
    optimj_model problem = new optimj_model();
    // a class for data handling
    Data d = new Data();
    LoadData ld = new LoadData();
    // loads data from the selected spreadsheet
    ld.init("params.xls", d);
    ...
```

Exporting the solution

The solver output can be printed to console (using `problem.solver().setOut(System.Out);` command) or saved into a log file using standard Java functions. A solution can be accessed as any other Java variable; OptimJ provides two functions for obtaining the solution information:

- `value(var variable)` – returns a value of the selected variable.
- `objValue()` – returns the value of the objective function.

A convenient way of displaying results is, for example to override the `toString()` function of the model. OptimJ provides links to the following solvers: CPLEX, Gurobi, glpk, lpsolve and Mosek.

Declaration of variables

As OptimJ is part of Java programming language, it uses the "programming" style of declaring variables and equations. Parameters are declared like any other Java variable; however variables for MILP are introduced by the keyword `var`. Unfortunately, even model formulation is affected by the used solver, which makes the formulation of a solver-independent model troublesome. For example a comparison in declaration of binary variables in a model with two different solvers:

```
// CPLEX solver
// Turbine status
   final var boolean[] TG_state[2];

// Gurobi solver
// Turbine status
   final var int[] TG_state[2] in 0 .. 1;
```

The bounds of variables can be specified when the variables are declared. If not specified, the default bounds (according to the type of variable – double, int, boolean...) are used. Declaration of non-negative continuous variables is as follows:

```
// Steam generated by boilers [t/h]
final var double Mp_PK in 0 .. Double.MAX_VALUE;
// Steam flow through turbines [t/h]
final var double[] Mp_TG[2] in 0 .. Double.MAX_VALUE;
```

Declaration of constraints

Constraints have to be declared in the `constraints` section of the model. During constraints declaration any Java or user-defined function can be used, as long as it keeps the model linear. Non-linear functions and SOS2 are discussed later. Another limitation is that in *for* loops a keyword *forall* needs to be used, as is shown in this example (OptimJ doesn't support double inequalities in the equations):

```
// 3. Minimal allowed flow through turbines
forall(int i : 0 .. Mp_TG.length-1) {
Mp_TG_min[i]*?TG_state[i] <= Mp_TG[i];
Mp_TG[i] <= Mp_TG_max[i]*?TG_state[i];
}
```

It is obvious that for accessing each element of the array we have to use the parameter *array length* in *for* loop. For summing elements of an array, OptimJ provides *sum* function with similar use as *forall* cycle. The objective function is introduced by a *minimize* or *maximize* keyword.

```
/*---- OBJECTIVE FUNCTION ----*/
minimize
java.lang.Math.abs(dev)*deviation_cost +
sum{int i : 0 .. Qin_PK.length-1}{Qin_PK[i]*fuel_cost};
```

Debugging and code clarity

Being incorporated into the Eclipse IDE, OptimJ can use sophisticated the Eclipse debugger and the code can be examined "step-by-step." OptimJ doesn't generate byte-code, but a standard Java source code, thus using JUnit (a framework for writing a repeatable tests [31]) or Javadoc (a tool from Sun Microsystems for generating API documentation in HTML [8]) is possible. An overview of the IDE is in figure 3.5. On the left is a project explorer window which makes maintaining even a larger project relatively simple. At the bottom is the command line output, and the biggest part of the environment is taken by the code editor.

On the contrary, OptimJ offers only one brief language manual and four sample projects for each supported solver. Javadoc documentation of OptimJ classes and methods is missing. As a result, the language documentation is rather poor.

As Java is object-oriented language, the model can be composed from different objects (e.g. boilers, turbines) which have their unique characteristics and as a result the code is developed faster (especially when we talk about large-scale models) and the code is even simpler. The code of the model is Java-like, thus easy to read for anybody who has experience of Java programming. Javadoc (if properly used) can make the code even clearer.

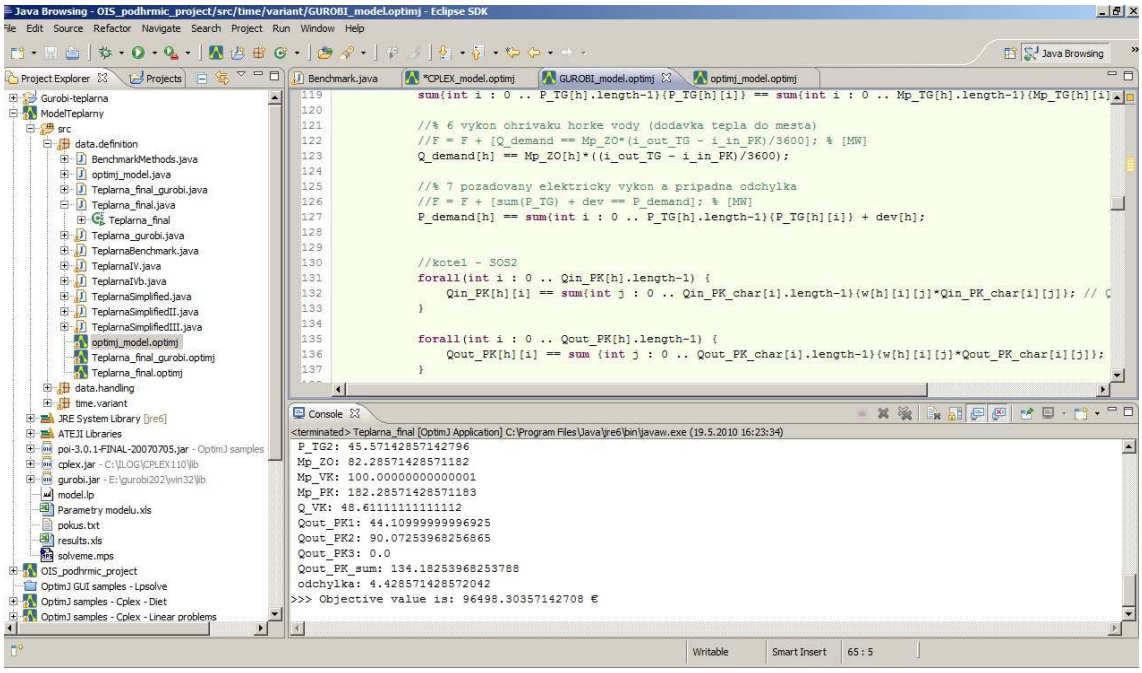

Figure 3.5: An OptimJ model developed in the Eclipse IDE

Setting up solver options

For setting up solver parameters an instance of the solve is needed. It can be obtained using `solver()` function. All functions of the solver API are then available. These functions are different for each solver; see solver documentation for further details. A simple example of changing CPLEX options is provided:

```
// get an instance of the solver
ilog.cplex.IloCplex m = problem.solver();
// print solver output into the command line
m.setOut(Syste.Out);
```

Non-linear functions and special ordered sets

Any non-linear function from the *java.lang.Math* package can be used, as well as any user-defined function. However OptimJ doesn't provide any linearizing. As a result, non-linear function that can be used in the model depends on the type of the solver. For example, CPLEX can extract the following functions: *abs, min, max* and PWL. Gurobi can handle only PWL (using SOS2).

A similar situation occurs with the formulation of PWL. OptimJ supports PWL only in connection with CPLEX or Gurobi solvers. The syntax depends on the solver, as can be seen in the following example (this part is identical for both solvers):

```
// Auxiliary variable for SOS2 (w substitutes lambda)
final var double[][] w[3][4] in 0 .. Double.MAX_VALUE;

/*** Boiler constraints - using SOS2 ***/
/* Power input definition */
forall(int i : 0 .. Qin_PK.length-1) {
Qin_PK[i] ==
sum{int j : 0 .. Qin_PK_char[i].length-1}{w[i][j]*Qin_PK_char[i][j]};
}
/* Power output definition */
forall(int i : 0 .. Qout_PK.length-1) {
Qout_PK[i] ==
sum {int j : 0 .. Qout_PK_char[i].length-1}{w[i][j]*Qout_PK_char[i][j]};
}
/* Output and input is non-zero only if the boiler is turned on */
forall(int i : 0 .. PK_state.length-1) {
sum {int k : 0 .. w[i].length-1} {w[i][k]} == ?PK_state[i];
}
```

A declaration of SOS2 for each boiler follows; a different function is used in each case:

```
/*** A - CPLEX ***/
forall(int i : 0 .. w.length-1) {
cplex11.SOS2(w[i], Qin_PK_char[i]);
}
/*** B - Gurobi ***/
forall(int i : 0 .. w.length-1) {
gurobi.addSOS(w[i], Qin_PK_char[i],2);
}
```

Price and licensing

A commercial licence for OptimJ starts at € 3,000,- for a basic package with linkers to solvers (solver licences have to be bought separately). OptimJ is licenced per developer seat, e.g. only a licence for code compilation is necessary, but no licence is needed for deploymend [18], thus no floating or site licences are available. For more information about licencing and price options the Ateji sales department has to be contacted.

Summary

The great advantage of OptimJ is its integration into Java language, which allows the user to create a model using an object-oriented environment. Another advantage is OptimJ's integration into Eclipse IDE, which provides a sophisticated platform for development. However, the solver-specific syntax requires models to be developed for one solver only and coupled poor documentation, these are the biggest drawbacks.

3.5 Gurobi API

The Gurobi Optimizer is a state-of-the-art linear programming and mixed-integer programming solver [5]. The Gurobi Optimizer provides APIs for C, C++, Python and Java programming languages. In this case Java API was chosen as interaction with Java is an important feature of the modelling language sought. In order to use Gurobi API the Gurobi library has to be imported into the Java project and then referenced in Java class using the command `import gurobi.*`.

Model export and Java connectivity

A Gurobi model can be exported into MPS/LP files using the `GRBModel.write()` method. Gurobi API is a library imported into Java project, thus the Java connectivity is perfect and out of question in this case.

Data import/export and MS Excel connectivity

Similarly to OptimJ (described in chapter 3.4) various file formats can be accessed, but user-defined methods are required. For connecting MS Excel files the *HSSF-API* library can also be used. However, Gurobi API provides its own function for loading data files. A function `GRBModel.read()` can read start file for MIP models (MST file), or Gurobi parameter files. A function `GRBModel.write()` writes the solution file.

Exporting the solution

The solver messages can be saved in a log file and further or printed on command line output. Solution can be accessed like any other Java variable (identical to OptimJ) using `GRBModel.get()` function. A sample use is as follows:

```
// Retrieving an objective value
 double objval = model.get(GRB.DoubleAttr.ObjVal);
```

Declaration of variables

Gurobi API syntax is identical to Java syntax and doesn't resemble any modelling language. Variables are always associated with a particular model and are created using the `GRBModel.addVar()` method. For example:

```
// Turbine status
 GRBVar TG_state_1 = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "TG_state_1");
 GRBVar TG_state_2 = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, "TG_state_2");
```

Bounds of the variables are set during their declaration. The first argument of the `addVar()` function is the lower bound, the second argument is the upper bound of the variable.

```
// Steam generated by boilers [t/h]
GRBVar Mp_PK=model.addVar(0.0,Double.MAX_VALUE,0.0,GRB.CONTINUOUS,"Mp_PK");
```

Declaration of constraints

Constraints are added to the model using the `GRBModel.addConstr()` function. Firstly, terms (variables) on both sides of the equation have to be added (using the function

`GRBLinExpr.addTerm()`), then the function `addConstr()` called. Only linear expressions can be added. A sample example is as follows (the original equation stands as $Mp_j^{TGmin} TG_j^{state} \leq Mp_j^{TG}$):

```
// 3. Minimal and maximal flow through turbines allowed
// left side of the equation
exprLeft = new GRBLinExpr();
exprLeft.addTerm(Mp_TG_min[0], TG_state_1);
// Right side of the equation
exprRight = new GRBLinExpr();
exprRight.addTerm(1.0, Mp_TG);
model.addConstr(exprLeft, GRB.LESS_EQUAL, exprRight, "c3");
```

Obviously such a declaration isn't very practical. However user-defined methods can simplify this declaration (e.g. functions like `addVariables(int numberOfVars, String[] typeOfVars, String[] names)`). The objective function is defined by objective coefficient of each variable, e.g. how many times the particular variable appears in the objective function. An important detail is that the objective function is always minimized, thus if we are solving a problem of "maximization" the objective coefficients have to be negative. Previously declared variables had their objective coefficients zero, as they don't appear in the objective function (min : $c_{dev}|dev| + c_{fuel} \sum_{k=1}^{n} Q_{in_k}^{PK}$). As we can see in the following example, the declaration of constraints is cumbersome):

```
/***** OBJECTIVE FUNCTION ****/
// the third argument is the objective coefficient
GRBVar v_dev = model.addVar(Double.MIN_VALUE,Double.MAX_VALUE,
           deviation_cost,GRB.CONTINUOUS,"v_dev");
GRBVar[] Qin_PK = new GRBVar[PK];
for(int i = 0; i < Qin_PK.length; i++) {
Qin_PK[i] = model.addVar(0, Double.MAX_VALUE,
            fuel_cost, GRB.CONTINUOUS, "Qin_PK" + i);
}
```

Debugging and code clarity

Debugging is done in the same way as it was with OptimJ and relies on the used environment (for example Eclipse IDE). On the contrary, Gurobi is well-documented, with

a satisfying number of examples (around 20 sample files). However, Javadoc is also not included.

As can be seen in previous examples, the model development is based on "low-level" programming (in comparison with high-level modelling as was presented by other modelling languages) and leads to a longer and more complicated code.

Setting up solver options

Setting up and chancing solver options can be done using the function `GRBEnv.set()` which belongs to `GRBEnv` class. For example:

```
// create the Gurobi environment first
GRBEnv env = new GRBEnv();
// Switch off the presolve feature
env.set(GRB.IntParam.Presolve, 0);
```

Non-linear functions and special ordered sets

Gurobi doesn't offer any kind of non-linear function reformulation, so the user is forced to declare only linear functions. However, PWL can be formulated using SOS2, as can be seen in this example (only a part of the formulation is shown, the original function is $P = \sum_{k=1}^{l} \lambda_k P_k$, at the end of the example $\lambda_k \in$ SOS 2 is set):

```
/*** Boiler constraints - using SOS2 ***/
/* Power input definition - w is an auxiliary variable for SOS2 */
GRBVar[][] w = new GRBVar[3][4];
for(int i=0; i<w.length; i++ {
    w[i] = model.addVars(4, GRB.CONTINOUS);
}
/* Create left and right-hand sides of the equation */
 GRBLinExpr[] exprLeft = new GRBLinExpr[PK];
 GRBLinExpr[] exprRight = new GRBLinExpr[PK];
for(int i = 0; i < expr1.length; i++) {
    exprLeft[i] = new GRBLinExpr();
    exprRight[i] = new GRBLinExpr();
    // on the left-hand side is a variable Qin
```

```
exprLeft[i].addTerm(1.0, Qin_PK[i]);
for (int j = 0; j < w[i].length; j++) {
// on the right-hand side is lambda_k * Qin_k
   exprRight[i].addTerm(Qin_PK_char[i][j],w[i][j]);
}
model.addConstr(exprLeft[i], GRB.EQUAL, exprRight[i], "cSOS2_" + i);
// set w as SOS2 variable
model.addSOS(w[i], Qin_PK_char[i],2);}
```

Price and licensing

A commercial licence for the Gurobi solver costs € 7,000,- and both floating and site licences are available.

Summary

Gurobi is primarily a solver and not a modelling language, and the difference in variables and constraints declaration is obvious. However, Gurobi API can easily be embedded into any Java application and developed in a suitable IDE (such as Eclipse). Good documentation is another advantage. If the user implements his own methods for easier variable and constraint declaration, model development might be faster. Nonetheless, the model implementation using the standard solver API is extremely complicated and leads to a long code which is hard to read, as can be seen from the previous examples. For this reason an implemented model isn't included in the appendix.

3.6 LINGO

LINGO is a comprehensive tool designed for building and solving linear, nonlinear and integer optimization models faster, easier and more efficiently [9]. LINGO provides its own set of solvers, LINGO modelling language, a stand-alone IDE, LINDO API for accessing the LINGO solvers from other applications and an MS Excel plug-in called "What's Best!" which allows users to formulate and solve models within the MS Excel application.

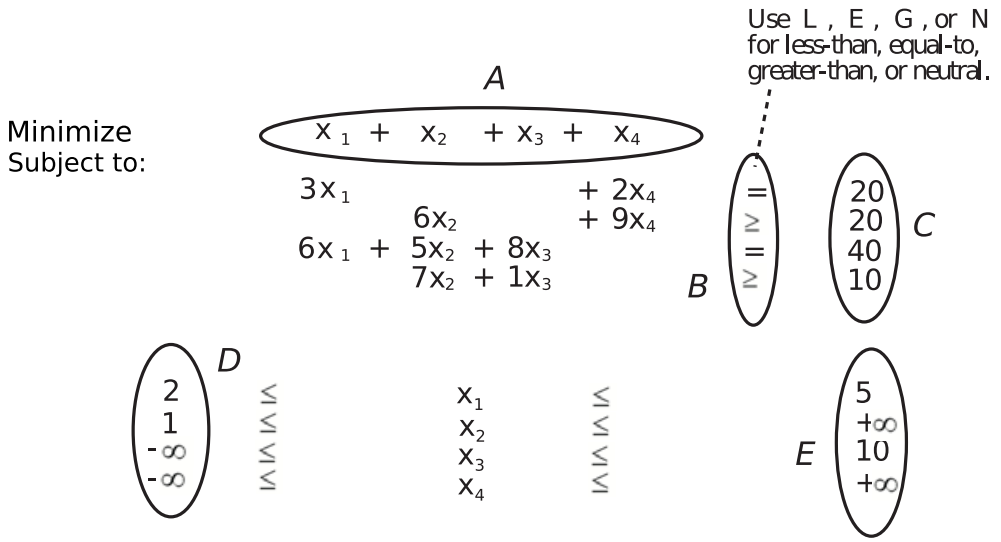

Figure 3.6: LINDO API Array representation of an LP model

Model export and Java connectivity

LINGO is capable of exporting models into MPS file format. Connectivity with Java applications can be done by LINDO API. LINDO API is a library imported into a Java project, and allows easy connection between a Java application and LINGO solvers and models. LINDO API provides functions for loading (MPS or LINGO native format), creating and solving a model. However, similar to Gurobi API for creating a model it provides only "low-level" functions. An LP or MILP model has to be characterised by 5 arrays (as is shown in figure 3.6) and by linear expressions only as constraints. Obviously it is not as comfortable as formulating the model in LINGO modelling language (which is described in later paragraphs).

The sample model from figure 3.6 is translated into the vector representation as follows (the constraints have to be defined separately):

```
A = [ 1 1 1 1 ].
B = [ E G E G ].
C = [ 20 20 40 10 ].
D = [ 2 1 -LS_INFINITY -LS_INFINITY].
E = [ 5 LS_INFINITY 10 LS_INFINITY ].
```

Data import/export and MS Excel connectivity

LINGO can load/save data from/to an MS Excel spreadsheet using its *Object Linking and Embedding* (OLE) interface. An example of a spreadsheet linking is following (the `@OLE()` function is calling the OLE interface with an address of the file as its parameter):

```
DATA:
! Load enthalpy values from MS Excel file
i_in_PK,i_out_PK,i_out_TG = @OLE('params.xls');
ENDDATA
```

Exporting the solution

LINGO generates a solution report file, with an objective value and values of variables. Using LINDO API the objective solution can be retrieved using the following function (`ls` stands for a Lingo environment):

```
// Copy an objective value from solved model into obj variable
ls.LSgetInfo(model, LS_DINFO_POBJ, obj);
```

Declaration of variables

LINGO, similar to GAMS, uses "set-oriented" notation. However, it extends this notation into an upper level, as each element of the set can have multiple attributes. These attributes can be both parameters and variables. For example, elements of the set TG are turbines used in the model. Attributes of the set TG then describe all features of these turbines (its status, minimal and maximal steam flow allowed and the el. power generated).

```
SETS:
! Turbine parameters, states, el.power output and steam flow;
     TG:Mp_TG_min, Mp_TG_max, TG_state, Mp_TG, P_TG;
ENDSETS
```

Sets have to be declared in the SETS section, parameters appear in the DATA section. At the point of declaration, it isn't known which attributes are variables and which are parameters. It is declared in the DATA section, where all parameters have to be set,

for example loaded from an MS Excel file. The number of elements of the set depends on the number of parameters in the spreadsheet. For example, in our sample file two Mp^{TGmin} and Mp^{TGmax} values are used, thus set TG contains two elements (turbine one and turbine two).

Attributes that aren't specified in the DATA section are treated as variables. The variables are by default treated as continuous or non-negative (depends on the global settings). Their bounds can be changed using the `@BND` command or by changing their type (for example to binary using `@BINARY()` command or to unbounded continuous using `@FREE()` command):

```
! Steam generated by boilers [t per h]
* UpperBound is a parameter;
@BND(0, Mp_PK, UpperBound);
```

Variables can be declared at the point of their first use (e.g. a constraint), however, then their type is specified according to global model settings. An example is given in the next paragraph.

Declaration of constraints

Constraints can be declared anywhere in the model after the SETS and DATA section. The `@FOR()` command accesses each element of the specified set (this set is given as an argument of the command). Summation of set attributes is done by `@SUM()` command, it has two arguments - a set and a parameter of the set that has to be summed up. As was mentioned above, variables can be declared straight in the equation (e.g. no bounds or type has to be specified, LINGO provides it automatically). See the following example (TG is a set of turbines, other variables are attributes of the set):

```
Declaring binary variables;
@FOR( TG:
@BIN( TG_state);
);

! 3. Minimal and maximal flow through turbines allowed;
@FOR(TG:
Mp_TG_min*TG_state <= Mp_TG;
```

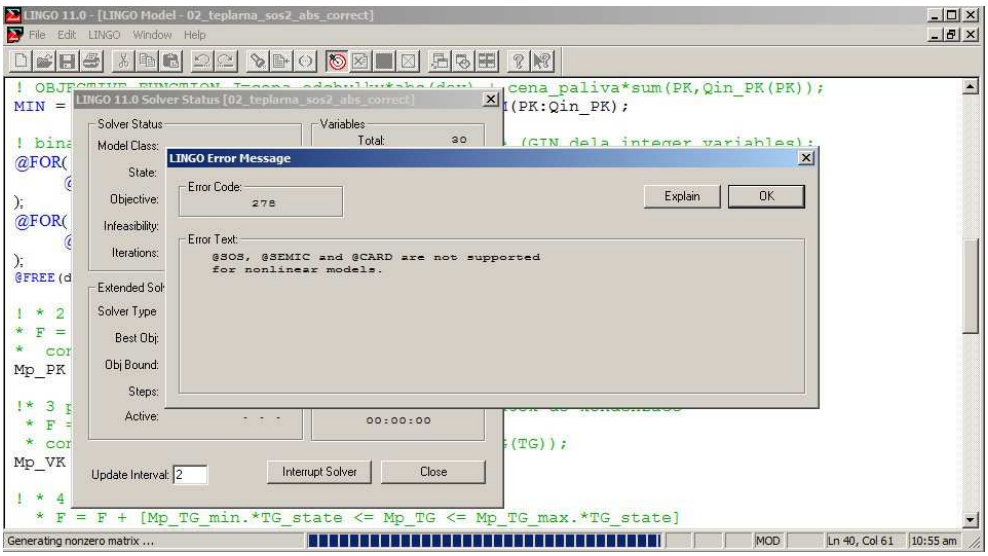

Figure 3.7: LINGO IDE with the error message pop-up

```
Mp_TG <= Mp_TG_max*TG_state;
);
```

LINGO doesn't support double inequalities in the equations. Keywords `MIN` and `MAX` denote the objective function, which can be set as follows:

```
!**** OBJECTIVE FUNCTION ****;
MIN = deviation_cost*@ABS(dev) + fuel_cost*@SUM(PK:Qin_PK);
```

Debugging and code clarity

LINGO provides good documentation and tens of sample models as an example. Its IDE is simple, it is basically a text editor that provides syntax highlighting and a graphical user interface (GUI) for various solver and model settings. Unfortunately simultaneous work with more models is not very comfortable, as LINGO IDE doesn't show tabs for opened files thus the user doesn't see all opened files. For debugging, comprehensive help is incorporated into IDE, and in case of an error, a message with a short explanation and a link to help topics appears, as shown in figure 3.7

LINGO syntax, which allows the user to declare sets with attributes makes the declaration of variables and parameters especially simple (for example, all variables and parameters of the basic model from chapter 1.1 were declared in 4 rows only). Also

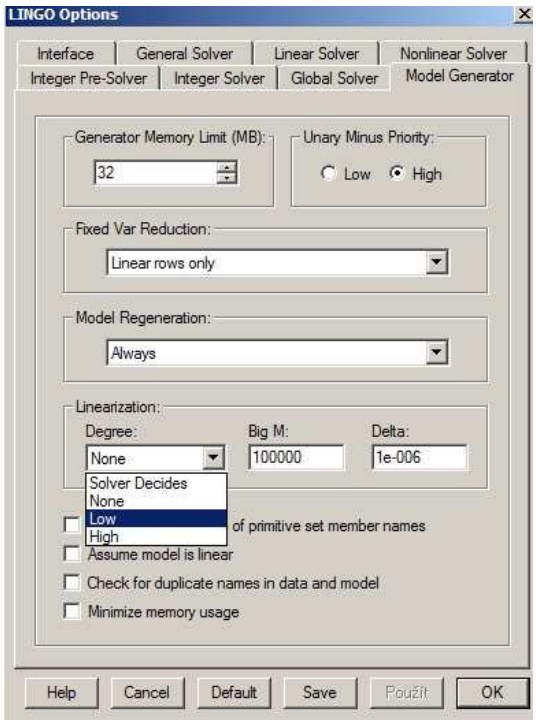

Figure 3.8: LINGO IDE and the solver options tab

constraints can easily be declared and as a result the code is economical and easy to read.

Setting up solver options

LINGO allows the user to specify the solver options using LINGO IDE, which provides interface to all LINGO solvers (integer, linear, non-linear and other). The solver options tab is shown in the figure 3.8. The solver can also be accessed through LINDO API functions.

Non-linear functions and special ordered sets

Unlike the other languages, LINGO provides its own linearization tool, which is done through a *big-M* operator (the same method Yalmip uses) and can linearize absolute value, minimum and maximum functions. Two levels of linearization are offered, the low level linearizes the functions mentioned above, on the high level even logical operators are linearized (e.g. $a \leq b$, $a \geq b$, $a \neq b$). However, linearization can substantially increase

the number of variables and constraints of the model [9].

As SOS2 are supported, a PWL can be formulated using the `@SOS2()` function as is shown in the following example:

```
!--- Boiler constraints - using SOS2 ----!;
@FOR(PK(I):
! Declaration of SOS2 for each boiler;
    @FOR(Q_PK_Char(I,J):
        @SOS2('W_' + PK( I), W( I, J))
     );
! Power input definition;
    Qin_PK(I) = @SUM(Q_PK_Char(I,J) : value_in(I,J)*W(I,J));
! Power output definition;
    Qout_PK(I) = @SUM(Q_PK_Char(I,J) : value_out(I,J)*W(I,J));
! Output and input is non-zero only if the boiler is turned on;
    @SUM(Q_PK_Char(I,J):W(I,J)) = PK_state(I);
);
```

Price and licensing

LINGO offers various prices for licences according to the maximum number of variables and constraints that the model can contain. For scheduling a real CHP system only the most expensive (so called *Extended*) licence is applicable (i.e. unlimited number of variables and constraints). The cheaper licences are too restrictive and don't allow the user to formulate a sufficiently large model.. A single commercial licence for LINGO Extended costs € 4,000,- and a commercial licence for LINDO API Extended (also the unlimited model size) costs € 3,300,- according to [9]. Both LINGO and LINDO API contain LINGO solvers, the aforementioned price is for the basic package only (e.g. LP/MILP solver). Floating and site licences are also offered.

Summary

The advantages of LINGO is its good connectivity with spreadsheets, a comprehensive documentation and its own linearizing tool. LINGO solvers might be an interesting alternative to other commercial solvers, although their performance is questionable. LINGO

IDE provides only basic functions and isn't as sophisticated as other modelling environments.

3.7 AIMMS

AIMMS is an advanced development environment for building optimization based operations research applications and advanced planning systems. It is a complex modelling environment offering linkage to many solvers, visualisation of results and MS Excel plug-in [1].

Although AIMMS can be run from command line (through AimmsCmd.exe), it is strongly connected with AIMMS IDE which provides full functionality and sophisticated visualisation abilities. AIMMS is available in two versions – ASCII and Unicode, the difference is the charset type used. Unfortunately AIMMS projects created in the Unicode version cannot be imported into the ASCII version of AIMMS and vice versa.

Model export and Java connectivity

AIMMS doesn't support export to MPS/LP files, only a special *flat model* option is offered which stores the model in a text file with specific formatting (it can be found in appendix D.4 and might be useful for debugging purposes). Import of MPS/LP files is not supported.

AIMMS is accessible from other applications using Component Object Model (COM) objects. COM is a binary-interface standard for communication between various software components [29]. Java by default doesn't communicate with COM objects, however various commercial and freeware Java toolkits are available (such as [20]).

Data import/export and MS Excel connectivity

AIMMS offers both MS Excel add-in, which allows users to change and load the model data directly from MS Excel (a configuration wizard is shown in figure 3.9) and an Excel function library, that provides functions for data import/export from spreadsheets.

Unlike other modelling languages data for the model are stored in so-called "cases" – e.g. a special file that stores all parameter and sets values. This feature is useful when we

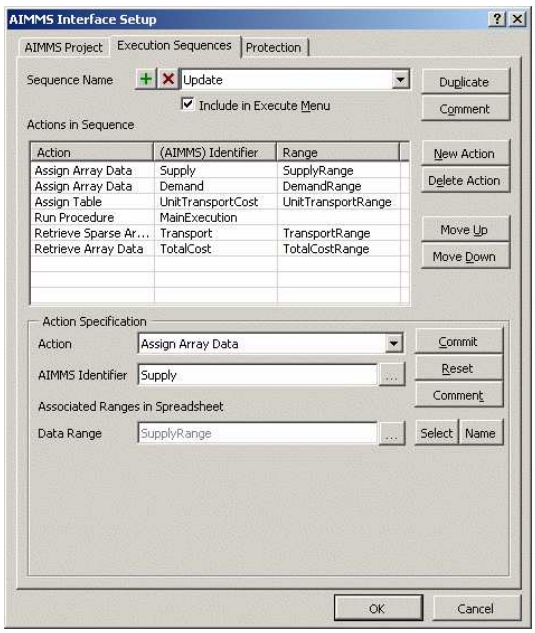

Figure 3.9: MS Excel configuration wizard for importing data into an AIMMS model

need to quickly change or update data of the whole model (for example, if new turbines with different characteristics were installed). The data set wizard is shown in figure 3.10. These cases can be exported or imported from a different project, using AIMMS native format.

Exporting the solution

By default the solution and solver messages are printed on command line output of the AIMMS IDE. The first possibility is to create a graphical user interface (GUI) within AIMMS IDE. This user-made GUI can provide visualisation of results as can be seen in figure 3.11. However, an AIMMS IDE is necessary for running such applications.

The second possibility is to access the AIMMS model and the solution from other applications using AIMMS COM functions.

Declaration of variables

The AIMMS syntax is "set-oriented" and similar to GAMS's notation. AIMMS IDE doesn't allow the user to "program" the model, all variables, parameters and constraints

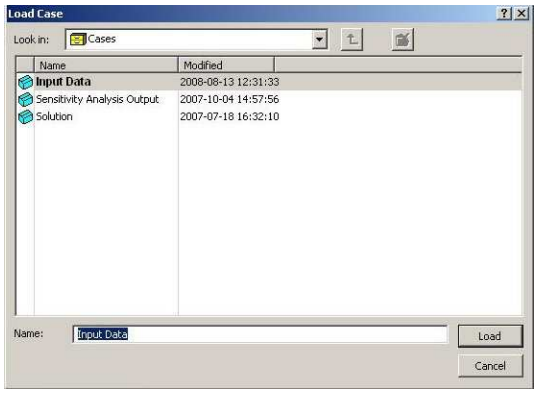

Figure 3.10: AIMMS IDE Data set wizard

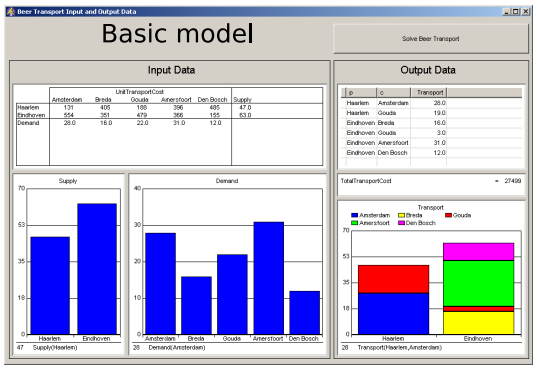

Figure 3.11: A simple GUI created in AIMMS IDE

have to be declared "clicking." For example if we want to declare a new binary variable TG^{state}, instead of typing:

```
* GAMS declaration *;
Binary variables
   TG_state(TG) turbine states;
```

the user has to click on the "new variable" button, then set the variable name, then click on the "range" button and choose "binary", then set variable domain (TG) and then click to the "commit and close" button, which is much more time demanding. A similar procedure applies for parameter and constraints declaration. The whole process is shown in figure 3.12. The main page of the AIMMS IDE is shown, on the left side is the model explorer, where all parameters, variables and constraints can be seen. Fields where the name, index and range of a variable are, are also shown. Finally the "commit and close" button is in the top right corner. A white box called "Definition" suits for constraint (we write equations) or data (we write data values) declaration.

Bounds of variables are set in the tab shown in figure 3.13 - it is possible to use default bounds (e.g. binary, non-negative) or define a specific bounds.

Declaration of constraints

Constraints are declared in a similar way to variables, only their definition has to be provided. The syntax is similar to GAMS syntax and AIMMS also doesn't support more inequalities in one equation. An example is as follows:

```
3. Minimal steam flow through turbines [t/h]
 Mp_TG(TG) >= Mp_TG_min(TG)*TG_state(TG);
```

The objective function is declared as:

```
Sum[PK, Qin_PK(PK)] * fuel_cost + v_dev * deviation_cost
```

Debugging and code clarity

Thanks to the project explorer utility (shown in figure 3.12) all variables, parameters and constraints of the model can easily be viewed. The project is divided into sections, one for variables, one for parameters etc. Sections for data loading and additional options

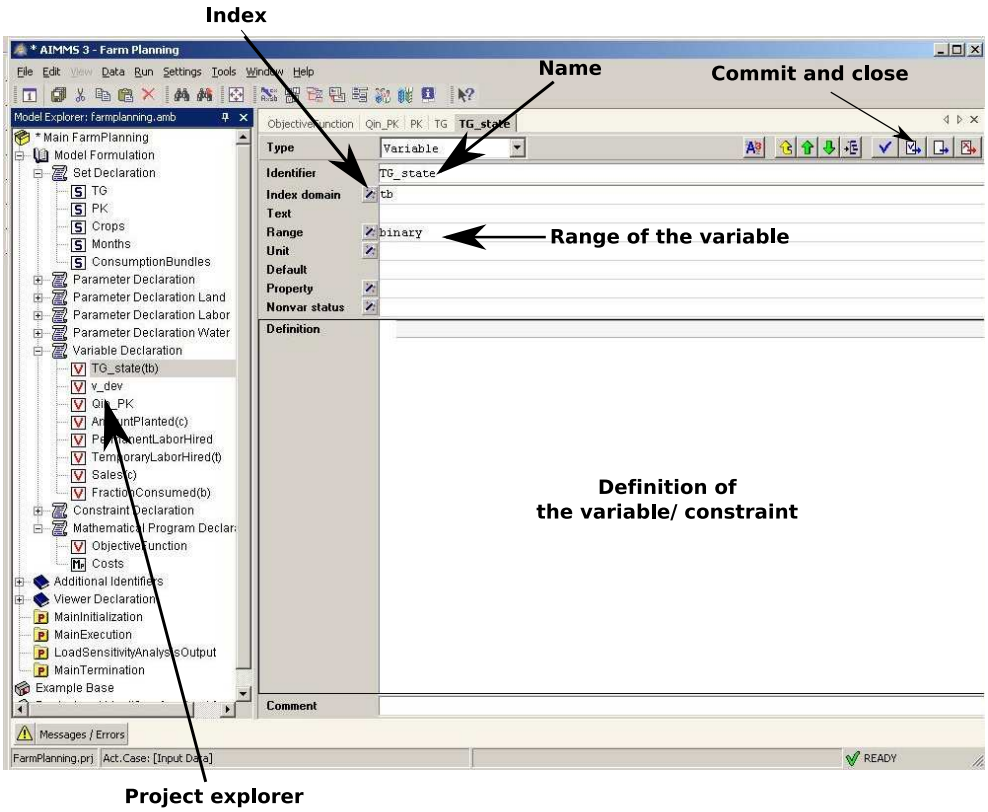

Figure 3.12: AIMMS IDE - declaration of variables/constraints

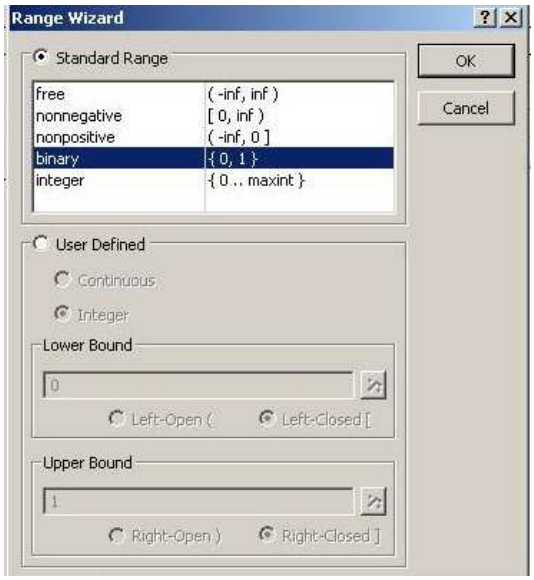

Figure 3.13: Defining the type of variable and its bounds

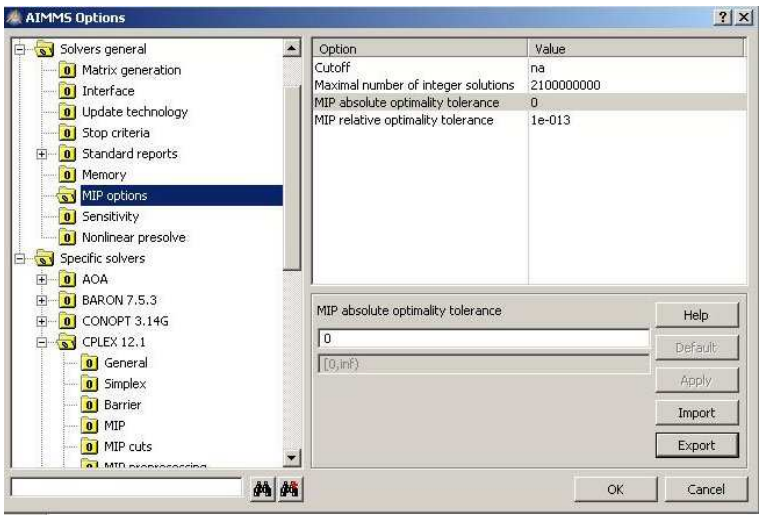

Figure 3.14: Setting up an optimality gap using the property editor

are also included. However, as the source code of the model cannot be directly edited, working with a large scale model might be complicated (as a large number of variables and equations will be introduced).

Error messages are printed to the command line output, and good documentation and integrated IDE help is provided, thus debugging options are satisfying.

Setting up solver options

AIMMS offers links to various solvers (Gurobi, CPLEX and many others) and both general options and options for a specific solver can be edited using the integrated property editor (shown in figure 3.14). Customized options can be both imported and exported into an option file.

Non-linear functions and special ordered sets

AIMMS doesn't handle the non-linear functions of a decision variable, thus the user has to reformulate absolute value and similar functions by himself. PWL can be implemented using SOS2. In AIMMS it is needed to specify sos2 in the property attribute of the constraint in which the λ's are added up to 1. It is the third constraint, where the sum of λ variables is equal to PK^{state}. The equation is as follows:

```
PK_state(PK) == sum[regions, w(PK,regions)];
```

Price and licensing

AIMMS offers an enduser licence which applies for AIMMS IDE on one machine, or a component licence which applies for applications that uses AIMMS COM library (e.g. a deployed Java scheduling application without AIMMS IDE). The commercial enduser licence for unlimited size of the model costs € 6,000,- and the same but component licence is being sold for € 5,000,- (solvers or solver linkers have to be bought separately). More information about licences, pricing and offered site and float licenses can be found at [1].

Summary

AIMMS provides a sophisticated IDE with wide visualisation options, data-set management and a property editor which allows easy modification of solver settings. Although various Java-to-COM linkers are available, connectivity with Java applications using AIMMS COM objects is questionable. Another disadvantage are limited model exporting options.

3.8 AMPL

The acronym AMPL stands for *A Mathematical Programming Language*. AMPL is a comprehensive and powerful algebraic modeling language for linear and nonlinear optimization problems, in discrete or continuous variables [2]. By default AMPL package offer a command line application only, for developing the model it is necessary to find suitable IDE or use an arbitrary text editor, as is discussed below.

Model export and Java connectivity

AMPL strictly separates model and data files and works as a linker between the solvers and the model file. A functional diagram can be seen in figure 3.15 – at the beginning a model file and data files are sent to AMPL and at the end a solution file is obtained. Another option is to write an executable AMPL script that automates this process.

AMPL can export the model into an MPS file. The Java application can call AMPL commands (e.g. load model, load data, solve, etc.) using the *Runtime()* class (in the same way as GAMS, in chapter 3.3). No AMPL Java API is provided.

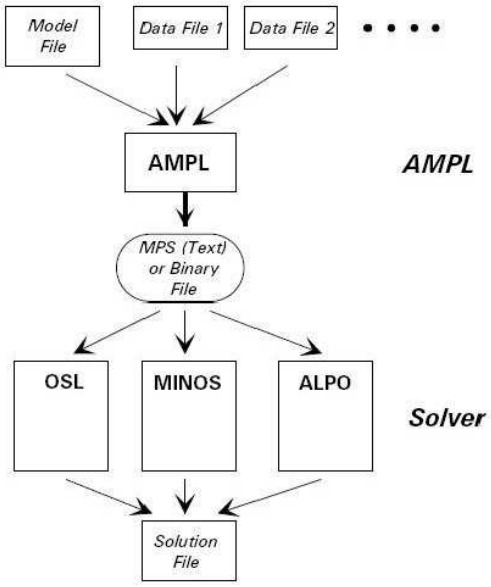

Figure 3.15: AMPL functional diagram

Data import/export and MS Excel connectivity

AMPL can load and save data from a special data file or from an MS Excel spreadsheet. The data file is a text file with declared sets, parameters and their values (in the model file the parameters and sets can only be referenced). To access spreadsheets AMPL uses the Open Database Connectivity (ODBC) standard library. The data loading procedure is as follows: Firstly a source file, ODBC handler and the name of the table has to be set. Then set which parameters have to be loaded (in this case turbine parameters are be loaded from the table params.xls). Finally the command `read table` *table_name* loads specified data from the table. Writing data into spreadsheets works on a similar basis (*"ODBC"* command is used without the keyword `IN` and `write table` *table_name* is used).

```
table TG_params IN "ODBC" "C:\temporary\params.xls" "TG_params":
TG <- [TG], Mp_TG_min, Mp_TG_max;
read table TG_params;
```

Exporting the solution

By default the solution and solver message are printed to the command line output, however it is possible to write solution into a data file or spreadsheet. A command `display` print variable values into the default output (a command line or a file).

Declaration of variables

AMPL is another "set-oriented" language, similar for example, to GAMS. Sets, parameters and variables can be declared anywhere in the code; the only limitation is that they have to be declared before their first reference. Declaration of a variable or parameter for each element of a set can be done by adding the set name into the variable declaration, as is shown below:

```
# turbines
set TG;
# Turbine states
var TG_state {i in TG} binary;
# Equivalent notation:
var TG_state {TG} binary;
```

More dimensional parameters and variables can be formulated using additional index, for example:

```
# Power input and output boiler characteristics
param Qin_PK_char{i in PK, j in regions};
```

Default bounds of variables are set according to their type (e.g. binary, continuous), however user-defined bounds can be used:

```
# Steam generated on boilers [t / h]
# MaxValue is a parameter
var Mp_PK >= 0 <= MaxValue;
```

Declaration of constraints

The AMPL syntax is similar to the GAMS syntax; the only difference is that each constraint has to be introduced by `subject to` keywords. An example is given below:

```
# Steam flow in turbines [t / h]
var Mp_TG {i in TG} >= 0;

# 3. Minimal and maximal flow through turbines allowed
subject to c3Min{j in TG}: Mp_TG[j] >= Mp_TG_min[j]*TG_state[j];
subject to c3Maxk{j in TG}: Mp_TG[j] <= Mp_TG_max[j]*TG_state[j];
```

The objective function can be introduced either by the `minimize` or `maximize` keyword:

```
##-- OBJECTIVE FUNCTION --#
minimize Cost: deviation_cost*v_dev + fuel_cost * sum{i in PK} Qin_PK[i];
```

Debugging and code clarity

Thanks to the "set-oriented" syntax and strict division between the model file and the data file, the code (as can be seen in appendix D.5) is economical and easy to read. However, there are two major concerns about AMPL language.

The first is a lack of an appropriate IDE. Debugging and developing a model using a text editor and a command line is possible, however for obvious reasons it is not very convenient for large models. Another option is to use a commercial IDE (such as OptiRisk AMPL Studio [24]), but choosing this option means additional costs. The third option is to develop a simple IDE from scratch (sample projects of AMPL GUI already exist), but this option leads to increased time costs.

The second concern is about AMPL documentation. The language is poorly documented and all available manuals refer to the AMPL Book from Robert Fourer, offered at the vendor's homepage [2]. This book contains a detailed description of AMPL modelling language and many examples. However, no free documentation in a similar quality is offered.

Setting up solver options

Solver settings can be set by `option` command, for example `option solver cplex` set CPLEX as an actual solver. Other options can be stored in the option file (different for each solver) which has to be loaded before solving the model.

Non-linear functions and special ordered sets

AMPL doesn't handle non-linear functions of decision variables. PWL has to be declared using special AMPL commands without introducing SOS2 variables (AMPL handles it automatically). The boiler power characteristics can be set as follows (*coeff* stands for angular coefficients of each part of the function):

```
##-- BOILER CONSTRAINT - PIECEWISE LINEAR FUNCTION --##
# the syntax is: << breakpoints, slope_list >> variable;
subject to Sos2 {i in PK}:
<<{j in regions}Qin_PK_char[i,j];0,{j in PK}coef[i,j],0>> Qout_PK[i]*PK_state[i]>=0
```

Price and licensing

According to [21] a single user commercial licence for AMPL costs € 3,200,- for a basic AMPL package. In order to find out the price of AMPL linked with a specific solver, a request at an AMPL sales department is necessary. Apart from single user licenses, floating licences are also available. The additional cost is the above-mentioned AMPL Book (around € 60,-). The OptiRisk AMPL Studio licence price is available upon request at an OptiRisk sales department.

Another possibility is to use the NEOS Server [12] for solving large scale AMPL problems for free. Nonetheless this option is good only for testing, as server availability at all times cannot be ensured, further more it is not desirable to send private data (e.g. CHP plant parameters) to the third side.

Summary

AIMMS offers a subtle modelling language capable of rapid model development, complemented by an ODBC handler for simple MS Excel data exchange. On the contrary, a lack of suitable IDE and poor documentation are serious disadvantages.

3.9 MPL

MPL (Mathematical Programming Language) is an advanced modeling system that allows the model developer to formulate complicated optimization models in a clear, concise, and efficient way [11].

Model export and Java connectivity

MPL offers export of models into an MPS/LP file format. MPL can be accessed from Java using the *OptiMax* library, the functionality offered is then similar to Gurobi API or LINDO API.

Data import/export and MS Excel connectivity

MPL can load data from a CSV text file or an MS Excel spreadsheet, using the ODBC handler and `EXCELRANGE` command. A sample use is as follows (reading turbine parameters):

```
! minimal allowed steam flow through turbines
Mp_TG_min[TG] = EXCELRANGE("params.xls", "MP_TG_min");
```

Exporting the solution

MPL by default creates a solution file and prints the solution and solver messages to command line output. In the case of accessing MPL from Java using the OptiMax library, the solution can be retrieved and manipulated in the same way as any other Java variable. For example the command:

```
model.getSolution().getObjectValue());
```

returns the objective value of the solved problem.

Declaration of variables

As is obvious even from the name, the MPL syntax is similar to AMPL language and also "set-oriented." In MPL sets are called *indexes*, because they serve as indexes for

parameters and variables. Sets have to declared in the `INDEX` section, parameters in the `DATA` section. A sample declaration:

```
INDEX
TG = (TG1,TG2);
! Turbine states;
TG_state[TG];
! it is a binary variable!
BINARY
TG_state[TG];
```

The bounds of variable are set according to their type, and can be narrowed introducing additional constraints.

```
! Steam generated by boilers [t/h]
Mp_PK;
! Add a new constraint
Mp_PK >= 0;
```

Declaration of constraints

The declaration of constraints is similar as that used in AMPL language; the constraints section has to be introduced by a `SUBJECT TO` keyword. A short example follows:

```
! Steam flow in turbines [t/h]
Mp_TG[TG];
SUBJECT TO
! Mp_TG is non-negative!
bounds_Mp_TG[TG]: Mp_TG[TG] >= 0;
! 3. Minimal and maximal flow through turbines allowed
min_tg[TG] : Mp_TG_min[TG]*TG_state[TG] <= Mp_TG[TG];
max_tg[TG] : Mp_TG_max[TG]*TG_state[TG] >= Mp_TG[TG];
```

The objective function is almost identical to the AMPL model:

```
!--- OBJECTIVE FUNCTION ---!
MIN Cost = deviation_cost*v_dev+ fuel_cost*sum(PK : Qin_PK);
```

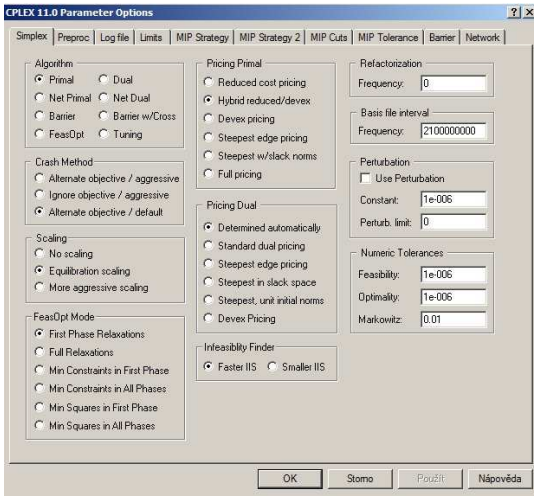

Figure 3.16: MPL IDE solver options tab (CPLEX)

Debugging and code clarity

Because MPL uses almost identical syntax to AMPL, the code is economical and easy to read. Error messages are printed into the command line; an interesting feature is the "check syntax" button, which checks whether the current model is syntactically correct. MPL documentation is available online and is satisfactory.

Setting up solver options

Solvers can be set using the solver option tab in the MPL IDE (figure 3.16 shows a tab with available CPLEX settings). General settings and settings for each separate solver can be used.

Non-linear functions and special ordered sets

MPL doesn't handle non-linear equations. PWL can be set using SOS2, the following (the implementation of boiler characteristics) is an example:

```
!--- Boiler constraints - using SOS2 ----!
! Power input definition
sos_1[PK] :
Qin_PK(PK)=sum(regions : (w[PK, regions]*Qin_PK_char[PK, regions]));
```

```
! Power output definition
sos_2[PK] :
Qout_PK(PK)=sum(regions : (w[PK, regions]*Qout_PK_char[PK, regions]));

! Output and input is non-zero only if the boiler is turned on
sos_3[PK] : PK_state(PK) = sum(regions : w[PK,regions]);

! Specify the SOS2 variables
SOS2
 s[PK]: SET( regions : w[PK,regions]);
```

Price and licensing

MPL offers a subscription licence and its price is based on the length of the subscripted period. During this period there are no restrictions on the software. MPL also offers both floating and site licences. However, to find out the price of a commercial MPL licence a request to an MPL sales department has to be made.

Summary

MPL is a language similar to AMPL, offering its own IDE, OptiMax library for accessing MPL from external applications and good connectivity with spreadsheets. However, it doesn't excel in any of these attributes,

3.10 Zimpl

Zimpl is an open-source modelling language created for academic purposes and offers only a command line application [17].

Model export and Java connectivity

Zimpl can export a model into MPS/LP file format. From a Java application it is only possible to call Zimpl commands using *Runtime()* class, in a similar ways as GAMS commands (see chapter 3.3).

Data import/export and MS Excel connectivity

Zimpl doesn't provide any kind of MS Excel connectivity or data import/export features. The only possible way is to import data from a text file with a specific format or from a CSV file.

Exporting the solution

Zimpl works as a modelling language only, its function is to export a Zimpl model into a standard MPS/LP file format. The rest, such as solving the model etc. has to be done by an external application.

Declaration of variables

Zimpl uses "set-oriented" notation, which strongly resembles AMPL syntax (described in chapter 3.8). For example a binary variable TG^{state} is declared as follows:

```
# A set of turbines
set TG := {"TG1", "TG2"};
# Turbine states
var TG_state[TG] binary;
```

Bounds of variables can be set during their declaration, otherwise the default bounds are used:

```
# Steam generated by boilers [t / h]
var Mp_PK >= 0 <= infinity;
```

Declaration of constraints

The syntax is similar to AMPL language, each constraint has to be introduced by the keyword `subto`. Zimpl doesn't support double inequalities in equations. An example of syntax follows:

```
# 3. Minimal and maximal flow through turbines allowed
subto c3min:
forall <j> in TG do Mp_TG[j] >= Mp_TG_min[j]*TG_state[j];
subto c3max:
forall <j> in TG do Mp_TG[j] <= Mp_TG_max[j]*TG_state[j];
```

The objective function can be introduced by the keywords `minimize` or `maximize`:

```
##-- OBJECTIVE FUNCTION --##
minimize cost: deviation_cost*v_dev  + fuel_cost * sum <i> in PK: Qin_PK[i];
```

Debugging and code clarity

In case of a syntax or any other error an error message is printed into the command line output, pointing to a line where the error occured. A simple user manual is offered; however for such a simple language as Zimpl certainly is, the documentation is satisfactory. Due to syntax similar to AMPL language, the code is economical and easy to read.

Setting up solver options

As Zimpl doesn't link with any solvers, setting up solver options is out of question.

Non-linear functions and special ordered sets

Non-linear functions of decision variables aren't supported and thus these functions have to be linearized by the user. However, Zimpl allows PWL functions to be formulated using SOS2. An example of boiler constraints is as follows:

```
##-- Boiler constraints - using SOS2 --##
# Power input definition
subto total_cost_x:
```

```
forall <i> in PK:
Qin_PK[i] == sum <j> in regions: (Qin_PK_char[i,j]*w[i,j]);

# Power output definition
subto weights_of_fx:
forall <i> in PK:
Qout_PK[i] == sum <j> in regions: (Qout_PK_char[i,j]*w[i,j]);

# Output and input
subto weights_must_sum_to_PK_state:
forall <i> in PK:
PK_state[i] == sum <j> in regions: w[i,j];

# Declaration of SOS2 for each boiler
sos s1: forall <j> in regions: type2: w["PK1",j];
sos s2: forall <j> in regions: type2: w["PK2",j];
sos s3: forall <j> in regions: type2: w["PK3",j];
```

Price and licensing

Zimpl is an open-source program released under GNU/GPL licence [4].

Summary

Zimpl might be a suitable language for educational purposes and an academic environment. However, due to its very limited functionality Zimpl can be hardly used for commercial purposes (no MS Excel connectivity nor utilities for data import/export). As source-codes are provided, Zimpl can become useful for deploying a modelling language for a special hardware architecture or if a specific customization is needed. The basic model implemented in Zimpl can be found in appendix D.8.

3.11 Modelling languages for the final testing

On the previous pages the basic model of a CHP system was implemented into the languages recommended in chapter 2. A detailed description of each language was given and the investigated language features were described at the beginning of this chapter. Finally, source codes of the implemented models can be found in appendix D.

The most important features (a good Java and MS Excel connectivity and links to Gurobi or CPLEX solvers, as described in chapters 1 and 2.3) which a suitable language should fulfill were met by a couple of languages. However, only three languages were selected for benchmarking, performed in the next chapter. Each of these three languages has a different syntax and thus a different style of formulating problems, which can affect the speed of the model formulating process.

The selected languages are the following:

Yalmip – currently used for modelling CHP systems in ongoing projects at the Department of Control Engineering. Despite its obvious disadvantages (such as licencing options) which exclude Yalmip from commercial use, Yalmip has been selected for further testing for comparison with other promising languages.

OptimJ – a promising Java-based language. Its biggest advantage is a direct integration with Java, which makes developing Java-based applications for scheduling very easy and makes OptimJ worth further testing. The second Java-related product, the Gurobi API doesn't provide such high-level programming functions as OptimJ, thus making Gurobi API unsuitable for formulating a large scale model. OptimJ uses the syntax of a programming language, hence making it easy for anyone with programming experience to learn.

GAMS, LINDO, Zimpl – the third tested languages have to use "set-oriented" syntax so as to compare all three types of languages (i.e. *programming, set-oriented* and *matrix-oriented*, as described in chapter 3.1). LINGO and GAMS are languages with similar abilities (apart from the fact that LINGO provides its own solvers and GAMS does not integrate well with Java). However, both languages offer only limited free licences (demo versions) unsuitable for benchmark testing (due to low maximal number of variables and constraints implementing a long-term scheduling model was not possible). In view of this fact an open-source solution was chosen – Zimpl, even though it offers only limited functionality.

Modelling language	**Java connectivity**	**MS Excel connectivity**	**Code clarity/ debugging options**	**Syntax style**
GAMS	low	good	good	*set-oriented*
LINGO	good	good	good	*set-oriented*
OptimJ	excellent	good	good	*programming*
Yalmip	poor	excellent	good	*matrix-oriented*
Zimpl	low	none	low	*set-oriented*

Table 3.1: A comparison of the modelling languages selected for benchmarking

Regarding the other languages, AMPL does not offer a suitable IDE, MPL does not excel in any of the investigated areas and AIMMS does not provide a satisfying Java connectivity. The languages recommended for benchmarking are briefly compared in table 3.1. As each feature of the language is described by one word only, it might be subjective and the reader should look up the details provided in chapters 3.2 and further.

An attribute *Java connectivity* describes how easy it is to interact the language with Java applications. In the *MS Excel connectivity* column,the ability of languages to read/write to spreadsheets is described, and the *Code clarity / debugging options* column summarizes the same named features of the languages. Finally, the *Syntax style* column describes the aforementioned language syntax. GAMS and LINGO languages are also included in the comparison, even though they were not benchmarked (due to the licence limitation).

In the next section the model extraction speed of the selected languages is evaluated, using an extended model for a longer planning horizon.

Chapter 4

Extended model implementation

The aim of this section is to implement an extended model of a CHP system, benchmark the extraction time of the selected modelling languages and evaluate the results obtained. The extended model and benchmarking methods are described in sections 4.1 and 4.2, the source code of the implemented models is enclosed in appendix E. Summarized results as well as the recommended and the most suitable modelling language are discussed in section 4.3.

4.1 Extended model of a cogeneration system

The extended model enhances the basic model from chapter 1.1 so the planning horizon can be extended to an arbitrary number of hours (i.e. the scheduling of the model can be made not only for one hour but for a significantly longer time) and an arbitrary number of boilers and turbines can be added. It was done by adding a time dimension to each variable (e.g. the deviation in el.power produced is no longer a scalar variable, but a vector which contains the deviation for each hour of the planning horizon). As for constraints, a *for* loop was added, in which the constraints for each hour were formulated (OptimJ and Yalmip models). In the Zimpl model the *for* loop was added to each constraint separately.

The number of boilers and turbines used in the system, as well as the planning horizon is set by appropriate model parameters (m for the number of boilers, n for the number of turbines, t for the number of hours). In the OptimJ model the command line arguments set these parameters, in the Yalmip model it is necessary to modify `benchmark.m` file (or

the values can be loaded from Matlab workspace). The Zimpl model reads the parameters from the file `data.dat`.

Each piece of system equipment (i.e. turbines and boilers) can have its specific power characteristics (this situation is likely in a real CHP system), however as a necessary simplification only one characteristic was used during benchmarking for the whole model.

4.2 Benchmarking methods

The aim of the benchmarking was to measure the extraction time of the model. An extraction time is a time that the language needs for formulating the problem itself into a solver-readable format, before the solver is called. This can be done internally (using an embedded solver API of the language) or externally, generating an MPS or LP file. The extraction time can be indispensable for large and complicated problems (i.e. tens thousands of variables and hundreds of constraints) and can dramatically increase the computation time of the optimal schedule. After the language formulates the model a solver is not called as the solution of the problem is not important in this case.

Benchmarking was proceeded for planning horizon from 1 up to 4,500 hours and for three models. The first one contains 3 boilers and 2 turbines (i.e. it is the basic model with a longer planning horizon) and its performance is shown in figure 4.1, the second model consists of 14 boilers and 8 turbines (figure 4.2) and the third model consists of 20 boilers and 14 turbines (figures 4.3 and 4.4). In the following section the details of each tested language are given. Results of the testing are discussed in section 4.3.

Yalmip

As was mentioned in the previous chapter, Yalmip is included in the benchmarking because it is used in certain optimization projects held at the Department of Control Engineering and is desirable to have a comparison of Yalmip and other languages. The processor time (also CPU time) necessary for declaring the model variables and constraints was measured. At the beginning of the model formulation is called the function `yalmip('clear')`, which clears the internal yalmip cache and should provide a better performance of Yalmip [23]. For measuring the Matlab functions `tic` and `toc` were used.

The function `solvesdp()` is not called, as the solution of the model isn't needed. The additional time spend by the `solvesdp()` function before it calls the solver itself (necessary operations before the solver can be called, also referred as "yalmiptime") is negligible in comparison to the extraction time, thus it wasn't measured.

The Yalmip performance can be seen in the following figures and is further discussed in section 4.3.

OptimJ

OptimJ extends the Java language and can easily be incorporated into any Java application. In OptimJ, the model is transformed into solver-readable format using the `extract()` function (this function has to be called before the model can be solved). Both CPU and the system time needed by this function was measured. *CPU* or *processor time* says how much processor time was spent on running an application (or function) code, *system time* says how much time was spent on associated I/O operations (e.g. writing auxiliary files on disk, reading data from disk).

In this case the system time took around 1/10 of the CPU time, thus wasn't counted in the extraction time.

Zimpl

Zimpl is an open-source set-oriented modelling language. It was chosen because Zimpl has no licence restrictions, unlike GAMS or LINGO, as was discussed in chapter 3.11. Zimpl extracts the model file into an MPS/LP file, which can be read by an appropriate solver. As was described in chapter 3.10, Zimpl cannot load data from spreadsheets and does not integrate well with Java. On the contrary, Zimpl can read text and CSV files and the parameter values of the extended model (t, m, n) are read from such a file (`data.dat`). An MPS file format was chosen for output file, however it is interchangeable with LP file format and Zimpl provides identical extraction time for both file types.

Turbine parameters and boiler characteristics are "hard-wired" into the code in order to simplify the data-loading process. The benchmarking is done from the Matlab environment using the `tic` and `toc` functions and `run.m` script. The Zimpl performance is

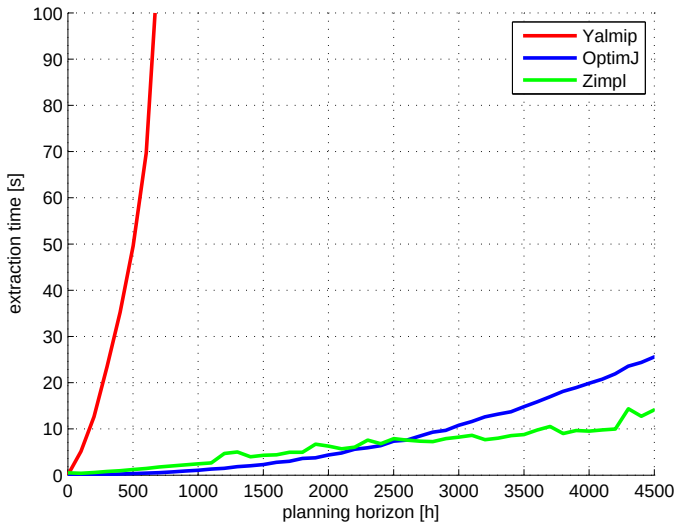

Figure 4.1: An extraction time of the model with 3 boilers and 2 turbines

discussed in the next section.

4.3 Results and recommendations

In this section results of performed benchmarking are discussed and a final evaluation of the tested languages is made.

Yalmip

As can be seen in the following figures, Yalmip provides significantly worse performance even on a relatively small problem (figure 4.1). The Yalmip model formulation complexity is exponential and will cause a significantly longer computation time of large scale models (as can be seen in figure 4.3 the extraction time reaches 10 minutes for 1000 hours of planing horizon). A smart formulation of the constraints (e.g. avoiding using *for* loops) might slightly improve the performance, but the model formulation complexity will still stay exponential. Yalmip is suitable for educational or academic purposes, but definitely not convenient for commercial use.

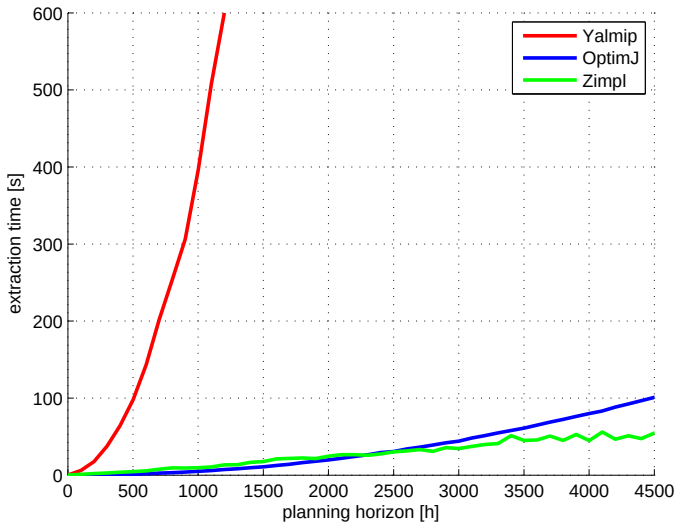

Figure 4.2: An extraction time of the model with 14 boilers and 8 turbines

OptimJ

The performance of OptimJ is significantly better. For example the extraction time of the largest tested problem (i.e. a model with 20 boilers, 14 turbines scheduled for 4,500 hours) is made within 3 minutes, which is a reasonable time. As can be seen in figure 4.4 the OptimJ model formulation complexity is polynomial and thus provides good results even for large scale problems and makes OptimJ a good candidate for operational use.

Zimpl

Zimpl surprisingly provides the best extraction time and linear model formulation complexity for all tested models, as can be seen in the following figures. In figure 4.4 a comparison between OptimJ's and Zimpl's extraction time shows that Zimpl is around two times faster than OptimJ for the larger models and longer planning horizon which keeps the extraction time below two minutes even for the largest tested models (i.e. a model with 20 boilers, 14 turbines and planning horizon 4,500 hours).

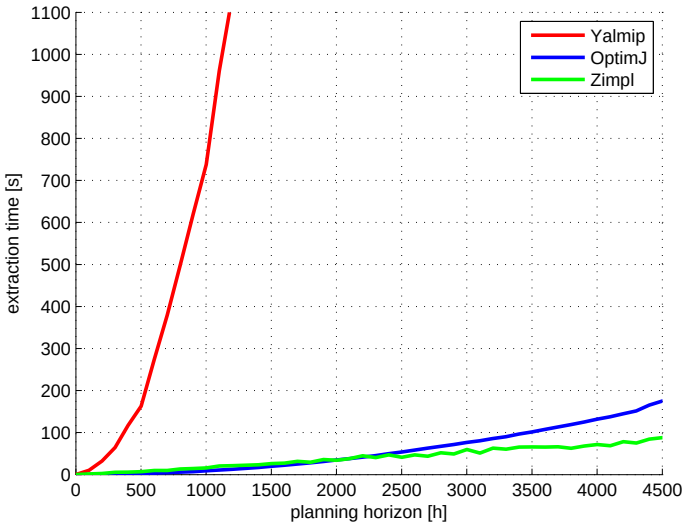

Figure 4.3: An extraction time of the model with 20 boilers and 14 turbines

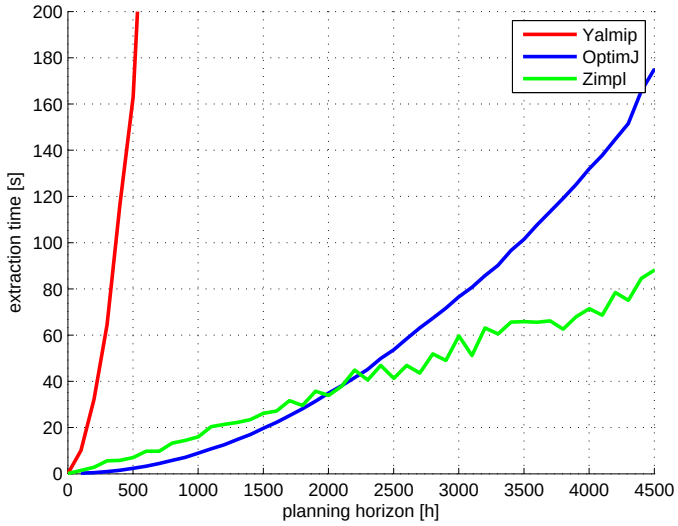

Figure 4.4: A close-up view of the model with 20 boilers and 14 turbines

Summary

During this benchmark the fastest model extraction time provided Zimpl with linear model formulation complexity. The second fastest was OptimJ providing a polynomial model formulation complexity and around twofold extraction time compared to Zimpl. The worst performance was provided by Yalmip, excluding this language from modelling large-scale problems. The extraction time raised linearly with the model size for all three languages (i.e. a model containing four times more turbines and boilers has a four-times-longer extraction time).

Zimpl represents languages which use "set-oriented" syntax and although Zimpl cannot be recommended due to its limited functionality and poor Java connectivity, its results make other set-oriented languages worth considering (as it is reasonable to assume that languages with similar syntax – such as AMPL or GAMS – have a similar performance). On the contrary, the time that a solver needs to load the MPS/LP model file is questionable. For example the extended model exported to an MPS file can be found on the enclosed CD.

OptimJ provides a satisfactory extraction time performance and thanks to its nature its very easy to incorporate OptimJ models into any Java application. This feature definitely makes OptimJ worth considering as a modelling language suitable for modelling and scheduling CHP systems, such as the projects currently being undertaken in the Department of Control Engineering.

Chapter 5

Conclusion

A thorough comparison of available modelling languages for Mixed-Integer Linear Programing has been addressed in this thesis. The overall aim of this work was to test, evaluate and recommend a modelling language that is the most suitable for the task of modelling and optimal scheduling of CHP systems. A desired modelling language has to give a good performance while extracting the model into a solver-readable format, has to integrate well with a Java environment and has to be capable of reading and writing to MS Excel spreadsheets. However, the comparison is useful for anybody facing similar optimization and scheduling problems.

The three levels of testing were set in this thesis. At the first level, a survey of available languages and their basic features was created. According to this survey the most promising languages were chosen for further testing. The second level of testing consisted of implementing a model of a simple CHP system and closely investigating the model formulating features of each language. Regarding the results of the second level, three languages were selected for the final part of testing – implementation of a more complicated model and benchmarking.

The modelling languages survey provides a basic overview and a quick comparison of available languages. The basic model implementation brings a closer look at the languages and shows their biggest advantages and drawbacks. Finally, the extended model implementation compares the performance and model extraction times of selected languages.

The modelling languages survey helped to determine 9 promising languages for further testing. During the basic model implementation three general categories of languages were found, based on the syntax they use – *programming language* style, *set-oriented* and *matrix-oriented* style. One language from each category was recommended for bench-

marking. Due to strict licence options the languages originally chosen cannot be benchmarked (GAMS and LINDO), thus Zimpl as an open-source alternative was used.

The best performance was provided by Zimpl, which extracts models with linear complexity. OptimJ took second place with reasonable polynomial model formulation complexity. The worst results came from Yalmip as it extracts models with exponential complexity and is thus unsuitable for formulating large-scale models. According to these results there are two recommended languages. The first one is OptimJ, which is especially interesting for its easy integration with Java applications and a reasonable performance. The second one is not Zimpl, because of its limited functionality, but a similar "set-oriented" language, such as AMPL or GAMS – but under certain conditions. As the best performance was made by Zimpl, a set-oriented modelling language, it is reasonable to expect that other set-oriented languages give a similar performance. Nonetheless, this assumption was not proved.

Zimpl generates MPS/LP files whereas the other two languages call solvers using an embedded API. Loading large models stored in MPS/LP files by appropriate solvers might be time-consuming and not as fast as using a solver API embedded into the modelling language. Another interesting question is the speed of OptimJ extraction time for MPS or LP files (for Zimpl the performance for both file formats is identical). It is also questionable whether there is a possible difference in solver computation time if different MILP problem formats are used (e.g. an MPS or LP file format). A recommended step is to extend the model by adding additional constraints (such as setting up time and starting costs of boilers) and benchmark this more complex model.

References

[1] AIMMS Website, 2010, [online].
⟨http://www.aimms.com/⟩.

[2] AMPL , A Modeling Language for Mathematical Programming, 2010, [online].
⟨http://www.ampl.com/⟩.

[3] Eclipse Website, 2010, [online].
⟨http://www.ateji.com/optimj.html⟩.

[4] GNU General Public Licence, 2007, [online].
⟨http://www.gnu.org/licenses/gpl.html⟩.

[5] Gurobi Optimization Website, 2010, [online].
⟨http://www.gurobi.com/⟩.

[6] HSSF - Java API To Access MS Excel Format Files, 2010, [online].
⟨http://poi.apache.org/spreadsheet/index.html⟩.

[7] J-Integra for COM, 2010, [online].
⟨http://j-integra.intrinsyc.com/support/com/doc/other_examples/Matlab.htm⟩.

[8] Javadoc Tool Home Page, 2010, [online].
⟨http://java.sun.com/j2se/javadoc/⟩.

[9] LINDO Systems Website, 2010, [online].
⟨http://www.lindo.com/⟩.

[10] LP file format, 2010, [online].
⟨http://www.gurobi.com/html/doc/refman/node386.html⟩.

[11] MPL Modeling System Website, 2010, [online].
⟨http://www.maximal-usa.com/mpl/⟩.

[12] NEOS Server for Optimization, 2010, [online].
⟨http://www-neos.mcs.anl.gov/⟩.

[13] Special Ordered Sets, 2010, [online].
⟨http://lpsolve.sourceforge.net/5.5/SOS.htm⟩.

[14] The General Algebraic Modeling System (GAMS) Website, 2010, [online].
⟨http://www.gams.com/⟩.

[15] The ILOG CPLEX Website, 2010, [online].
⟨http://ilog.com/products/cplex/⟩.

[16] The MathWorks MATLAB Website, 2010, [online].
⟨http://www.mathworks.com/products/matlab/⟩.

[17] ZIMPL Website, 2009, [online].
⟨http://zimpl.zib.de/⟩.

[18] ATEJI. OptimJ: A Java-based Modeling Language, 2010, [online].
⟨http://www.ateji.com/optimj.html⟩.

[19] Boyce, M. P. *Handbook for cogeneration and combined cycle power plants*. The American Society of Mechanical Engineers, 2002.

[20] EZJCom. Easy Java COM Connectivity, 2009, [online].
⟨http://www.ezjcom.com/⟩.

[21] Fourer, R. Linear programming software survey. *OR/MS Today*, June 2009.

[22] Linderoth, J. *Integer programming: theory and practice*, chapter Noncommercial Software for Mixed-Integer Linear Programming. CRC Press, 2006.

[23] Löfberg, J. Yalmip : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004.

[24] Optirisk Systems. AMPL Studio, 2010, [online].
⟨http://www.optirisk-systems.com/products_ amplstudio.asp⟩.

[25] Ramsay, B. et al. EDUCOGEN: The European Educational Tool for cogeneration. In *COGEN Europe, The European Association for the Promotion of Cogeneration*, Brussels, 2003.

[26] Seeger, T. and Verstege, J. Short term scheduling in cogenerative systems. In *Power Industry Application Conference*, Baltimore, Maryland, USA, 1991.

[27] Simovic, T. Optimal production planning of a cogeneration system. Master's thesis, Czech Technical University Faculty of Electrotechnical Engineering, 2008.

[28] Steiger, D. and Sharda, R. Lp modeling languages for personal computers: A comparison. *Annals of Operations Research*, March 1993.

[29] Wikipedia: The Free Encyclopedia. Component Object Model, 2010, [online]. ⟨http://en.wikipedia.org/wiki/Component_Object_Model⟩.

[30] Wikipedia: The Free Encyclopedia. Enthalpy, 2010, [online]. ⟨http://en.wikipedia.org/wiki/Enthalpy⟩.

[31] Wikipedia: The Free Encyclopedia. JUnit, 2010, [online]. ⟨http://en.wikipedia.org/wiki/JUnit⟩.

[32] Wikipedia: The Free Encyclopedia. Mathematical Programming System file format, 2010, [online]. ⟨http://en.wikipedia.org/wiki/MPS_(format)⟩.

Appendix A

Results of modelling languages survey

modelling sofware	General Information					
	Platforms				Availability	
	PC Windows		PC Linux			
	32-bit	64-bit	32-bit	64-bit	Stand alone app.	Callable library
AIMMS the modelling systém	Yes	Yes	Yes	Yes	Yes	Yes (*.DLL library)
AMPL	Yes	Yes	Yes	Yes	Yes	No But it is possible to call ampl from external application
GAMS	Yes	Yes	Yes	Yes	Yes	Yes
GUROBI API	Yes	Yes	Yes	Yes	Yes	Yes
ILOG OPL Dev.studio	Yes	Yes	Yes	Yes	Yes	-
Jmodelica	Yes	Yes	Yes	Yes	Yes (Python environment)	Yes (Python based, provides C, Java API)
LINGO	Yes	Yes	Yes	Yes	Yes	Yes (LINDO API DLL library), What's Best! as a plug-in to MS Excel
Microsoft Solver Foundation	Yes	Yes	No	No	No, has to be imported to MS Excel or Visual Studio	Yes, plug-in to MS Excel 2007 or Visual Studio, other ways: C#, C++, IronPython, OML
MOSEK	Yes	Yes	Yes	Yes	Yes	Yes (provides: Java API, C API, .NET API, Python API, optimJ,MS Excel plug-in)
MPL	Yes	Yes	No	No	Yes	Yes (OptiMax library)
OptimJ	Yes	Yes	Yes	Yes	No	Yes (Eclipse plug-in)
YALMIP	Yes	Yes	Yes	Yes	No	Yes (Matlab Add-in)
ZIMPL	Yes	No	Yes	Yes	Yes (command line)	Yes

Figure A.1: General information

modelling software	Inputs/outputs					
	Integrated modelling environment/solver	Supported formats			LP/MPS format	Other solvers connectivity
		Spreadsheets	Database	Text		
AIMMS the modelling systém	Modelling environment only	Yes	Yes	Yes	No/No	CPLEX, GUROBI, XPRESS, XA, CONOPT, KNITRO, LGO, BARON and more; plus AIMMS Open Solver Interface
AMPL	Modelling environment only	Yes	Yes	Yes	Yes/Yes	35+ solvers listed at www.ampl.com/solvers.html
GAMS	Modelling environment only	Yes	Yes	Yes	Yes/Yes	ALPHAECP, BARON, CONOPT, CPLEX, DECIS, DICOPT, GUROBI, KNITRO, LGO, LINOGLOBAL, MINOS, MOSEK, MPSGE, MSNLP, OONLP, OSL, PATH, SBB, SNOPT, XA, XPRESS
GUROBI API	Solver	No	No	Yes	Yes/Yes	It is a solver itself
ILOG OPL Dev.studio	Model + solver	Yes	Yes	Yes	-	IBM ILOG CPLEX, IBM ILOG CP Optimizer
Jmodelica	Modelling environment only	No	No	Yes	No/No	IPOPT
LINGO	Model + solver	Yes	Yes	Yes	MPS	LINDO API -> Matlab, C, Java... LINGO and What'sBest has its own solvers
Microsoft Solver Foundation	Model + solver	Yes	-	-	MPS	It has its own solvers (plus GUROBI), other solver could be accessible through .MPS file
MOSEK	Solver	Depends on modelling software			MPS	It is a solver itself
MPL	Modelling environment only	Yes	Yes	Yes	Yes/Yes	CPLEX, GUROBI, XPRESS, OSL, XA, MOPS, LINDO, FORTMP, CoWHIZ, COINMP, GLPK, LPSOLVE, CONOPT, KNITRO, LGO, PATH, EXCEL
OptimJ	Modelling environment only	No (could be implemented)	No (could be implemented)	Yes	Yes/Yes	Directly: lp_solve, Mosek, GUROBI, CPLEX, glpk Others using MPS/LP files
YALMIP	Modelling environment only	Yes (depends on Matlab)	Yes (depends on Matlab)	Yes (depends on Matlab)	through Matlab only	BINTPROG, BPMPD, CDD, CLP, CPLEX, CSDP, DSDP, FMINCON, GLPK, GPPOSY, KYPD, IPOPT, LINPROG, LMILAB, LMIRANK, LPSOLVE, MAXDET, MOSEK, MPT, NAG, OOQP, PENBMI, PENSDP, QPC, QSopt, QUADPROG, SDPA, SDPLR, SDPT3, SeDuMi, SNOPT, VSDP, XPRESS
ZIMPL	Modelling environment only	No	No	Yes	Yes/Yes	SCIP, LP_SOLVE,SoPlex + others using MPS file

Figure A.2: Inputs/Outputs

modelling software	Price and licensing												
	Commercial			Educational/Academic			Demo/Student	Demo version limits					NEOS server availability
	Single	Floating license	Site license	Single	Floating license	Site license	Single	Constraints	Variables	Integer Variables			
AIMMS the modelling system	From 1400€	Yes	Yes	450€	Yes	Yes	Free	300	300	300			No
AMPL	$4,000.00	Yes	No	$400.00	Yes	No	Free	300	300	300			Yes
GAMS	www.gams.com/sales/commercialp.pdf	Yes	Yes	www.gams.com/sales/commercialp.pdf	Yes	Yes	Free	300	300	50			Yes
GUROBI API	See website	Yes	Yes	$100 – $850 (depending on options)	Yes	Yes	Free	500	500	500			No
ILOG OPL Dev.studio	info@ilog.fr	Yes	Yes	995-24955$	Yes	Yes	Free	500	500	-			No
Jmodelica	Free	-	-	Free	-	-	Free	unlimited	unlimited	unlimited			No
LINGO	From 495$	Yes	Yes	From 295$	Yes	Yes	Free	150	300	30			No
Microsoft Solver Foundation	www.solverfoundation.com	Yes	Yes	Free (MicrosoftSoftware Aliance)	-	-	Free	500	500	500			No
MOSEK	$4,000.00	Yes	No	$800.00	Yes	No	Free	unlimited	unlimited	unlimited			Yes
MPL	Free MPL for Development, contact for runtime/solver prices	Yes	Yes	Free MPL for Academics, contact for standard prices	Yes	Yes	Free	500	500	500			No
OptimJ	Starts at €3000 (depends on solver)	-	-	Free	-	-	Free	unlimited	unlimited	unlimited			No
YALMIP	Free	-	-	Free	-	-	Free	unlimited	unlimited	unlimited			No
ZIMPL	Free (open source)	-	-	Free	-	-	Free	unlimited	unlimited	unlimited			No

Figure A.3: Price and lincensing

modelling sofware	Problem formulation			
	SOS1	SOS2	Branching priorities of binary variables	Non-linear functions (absolute value)
AIMMS the modelling systém	Yes	Yes	Yes	No
AMPL	Yes	Yes	Yes, even for integers (using „priority“ value), needs CPLEX library	No
GAMS	Yes	Yes	Yes	No
GUROBI API	Yes	Yes	solver only	solver only
ILOG OPL Dev.studio	Yes	Yes	-	-
Jmodelica	No	No	No	Yes
LINGO	Yes	Yes	Yes, but limited – it is possible only to choose whether to branch binary variables first	Yes, using Lingo linearization tool
Microsoft Solver Foundation	-	-	-	-
MOSEK	No	No	solver only	solver only
MPL	Yes	Yes	Yes	No
OptimJ	Yes	Yes	depends on solver	Yes (based on Java.Math)
YALMIP	No	No	(According to the YALMIP Wiki) No	Yes
ZIMPL	Yes	Yes	Yes (using *priority* keyword)	No

Figure A.4: Problem formulation

Appendix B

Basic model parameters and power characteristics

B.1 Parameters

TG	Mp_{min}^{TG} [t/h]	Mp_{max}^{TG} [t/h]
TG1	60	320
TG2	60	420

Table B.1: Minimal and maximal steam flow in turbines

Enthalpy [kJ/kg]	
i_{in}^{PK}	750
i_{out}^{PK}	3400
i_{out}^{TG}	2500

Table B.2: Steam enthalpy values

B.2 Power characteristics of boilers

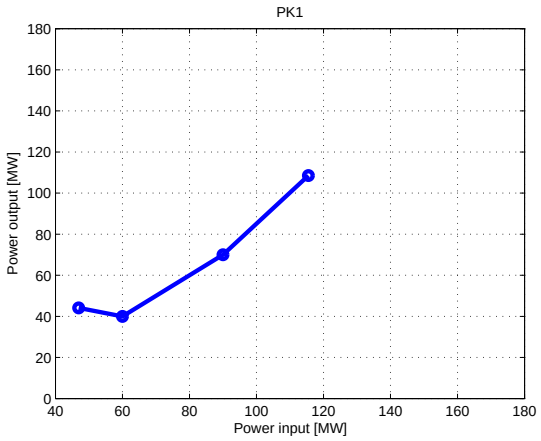

Figure B.1: Power characteristic of boiler PK1

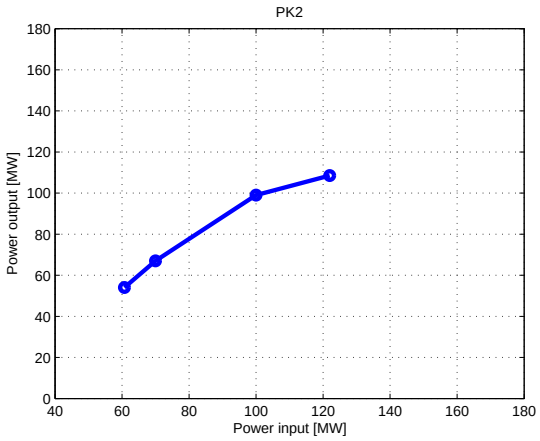

Figure B.2: Power characteristic of boiler PK2

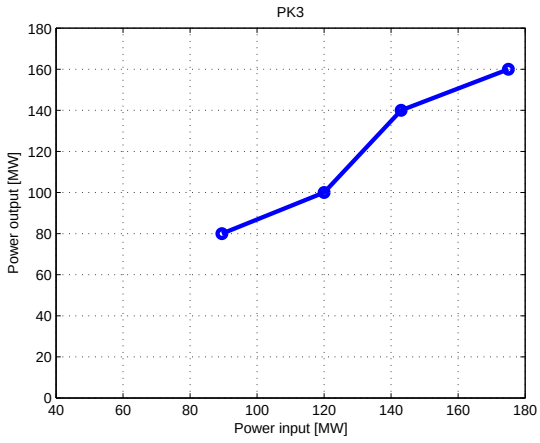

Figure B.3: Power characteristic of boiler PK3

Appendix C

Contents of the enclosed CD

The enclosed CD contains the following:

- The text of this report (*podhrm_2010_modelling_languages_for_optimization.pdf*).
- The text of this report – print version.
- A copy of the official requirements for this work (*podhrm_2010_requirements.pdf*).
- Source codes of the implemented basic models.
- Source codes of the implemented extended models.
- MS Excel spreadsheet containing parameters of the basic model (power characteristics of boilers, steam enthalpy and turbine parameters).
- Other sample files such as log files, Eclipse projects and used Java classes (to be found in the appropriate folders).

Appendix D

Source codes of basic model implementation

D.1 Yalmip

```
%% BASIC MODEL IMPLEMENTATION %%
% Simplified cogeneration system:
% - Three gas-fueled boilers with non-linear characteristics.
% - Two turbines with linear characteristics.
% - Condenser.
% - Minimizing production cost of desired power and heat production.
clear;

%%---- PARAMETERS ----%%
% Production demanded
Q_demand = 200; % Heat/hot-water production [MW]
P_demand = 250; % Electric power pruduction [MW]
fuel_cost = 600; % [CZK/MWh]
deviation_cost = 3000; % [CZK/MWh]

% Loading parameters from MS Excel spreadsheet
num = xlsread('model_params.xls');

% Mininal and maximal steam flow through turbines allowed
Mp_TG_min = num(1,2:3)';
Mp_TG_max = num(2,2:3)';

% Enthalpy
i_in_PK = num(6,2);
```

```
i_out_PK = num(7,2);
i_out_TG = num(8,2);

% Power input and output boiler characteristics
Qin_PK_char = [num(14:17,1) num(21:24,1) num(28:31,1)]';
Qout_PK_char = [num(14:17,2) num(21:24,2) num(28:31,2)]';

%%----VARIABLES ----%%
% Boiler status (3 boilers, on/off)
PK_state = binvar(3,1,'full');

% Power output of boilers [MW]
% row = boiler, column = output in a given section
Qout_PK = sdpvar(3,3,'full');

% Power input of boilers [MW]
% row = boiler, column = input in a given section
Qin_PK = sdpvar(3,3,'full');

% Steam generated by boilers [t/h]
Mp_PK = sdpvar(1);

% Turbine status
TG_state = binvar(2,1,'full');

% Steam flow in turbines [t/h]
Mp_TG = sdpvar(2,1,'full');

% Electric power generated by turbines [MW]
P_TG = sdpvar(2,1,'full');

% Possible difference in the electric power supplied
% (positive = production is lower than planned)
dev = sdpvar(1);

% Steam flow to condenser [t/h]
Mp_VK = sdpvar(1);

% Steam flow to water heater [t/h]
Mp_ZO = sdpvar(1);

% Auxiliary variables for boiler's non-linear characteristics
```

```
% It is not a very efective formulation and it is desirable to
% reformulate it using SOS2
% Currently each boiler's power characteristic
% is altered using three linear sections
PK_regions = binvar(3,3,'full'); % row = boiler, column = section

%%---- CONSTRAINTS  ---%%
% Set of constraints
F = [];

% Non-negative variables
F = F + [Mp_PK>=0; Mp_TG>=0; Mp_VK>=0; Mp_ZO>=0;];

% 1. Steam generated by boilers is equal to the steam flow to turbines
F = F + [sum(Mp_PK) == sum(Mp_TG)];

% 2. Steam flow through turbines is equal to the flow to the condenser plus
% flow to the water heater
F = F + [sum(Mp_TG) == Mp_VK + Mp_ZO];

% 3. Minimal and maximal flow through turbines allowed
F = F + [Mp_TG_min.*TG_state <= Mp_TG <= Mp_TG_max.*TG_state];

% 4. The amount of electric energy produced is equal to the entalpy difference in
% the steam (eg. before and after the steam leaves turbines)
F = F + [P_TG == Mp_TG*(i_out_PK - i_out_TG)/3600]; % [MW]

% 5. Power of heating unit (heating water for further use in buildings)
F = F + [Q_demand == Mp_ZO*(i_out_TG - i_in_PK)/3600]; % [MW]

% 6. Desired power and possible deviation
F = F + [sum(P_TG) + dev == P_demand]; % [MW]

%% Boiler constraints %%
% Relation between power input and output of boiler
% non-linear characteristics
% (desirable to reformulate using SOS2)

% 7.Total power of boilers must be sufficient to boil desired amount of
% steam
F = F + [sum(sum(Qout_PK)) == Mp_PK*(i_out_PK - i_in_PK)/3600]; % [MW]
```

```
% If a boiler is in operation its power could vary only in one power section
F = F + [sum(PK_regions,2) == PK_state];

% Input and ouput connection
% 1. Power output has to be in one section only
F=F+[Qin_PK_char(:,1:end-1).*PK_regions<=Qin_PK<=Qin_PK_char(:,2:end).*PK_regions];

% 2. Power output is set by lines with angular coeficients
coeff = diff(Qout_PK_char,1,2)./diff(Qin_PK_char,1,2);
ofset = Qout_PK_char(:,1:end-1) - coeff.*Qin_PK_char(:,1:end-1);
F = F + [Qout_PK == coeff.*Qin_PK + ofset.*PK_regions];

%%---- OBJECTIVE FUNCTION ----%%
% J = fuel costs + electric power production costs
J = fuel_cost*sum(sum(Qin_PK)) + deviation_cost*abs(dev);

%%---- SOLVE PROBLEM ----%%
diag = solvesdp(F,J,sdpsettings('solver','cplex'))

%%---- SHOW RESULTS ----%%
res.P_demand = P_demand;
res.Q_demand = Q_demand;
res.P_TG1 = double(P_TG(1));
res.P_TG2 = double(P_TG(2));
res.Mp_ZO = double(Mp_ZO);
res.Mp_VK = double(Mp_VK);
res.Mp_PK = double(Mp_PK);
res.Q_VK = double(Mp_VK)*(i_out_TG-i_in_PK)/3600; %power loss on condensator[MW]
res.Qout_PK1 = sum(double(Qout_PK(1,:))); % power of boiler one [MW]
res.Qout_PK2 = sum(double(Qout_PK(2,:))); % power of boiler two [MW]
res.Qout_PK3 = sum(double(Qout_PK(3,:))); % power of boiler three [MW]
res.Qout_PK_sum = sum(sum(double(Qout_PK))); % total power of boilers [MW]
res.dev = double(dev); % planned deviation [MW]
res.costs = double(J); % total costs of production [Kc/h])
res % showing the results
```

D.2 GAMS

```
$Ontext
***** BASIC MODEL IMPLEMENTATION *****
```

```
 Simplified cogeneration system:
 - Three gas-fueled boilers with non-linear characteristics.
 - Two turbines with linear characteristics.
 - Condenser.
 - Minimizing production cost of desired power and heat production.
$Offtext

**** SETS *****
Sets
         PK boilers /PK1,PK2,PK3/
         TG turbines /TG1, TG2/
         regions breakpoints at boiler's power characteristics /r1,r2,r3,r4/
        ;

**** SCALAR PARAMETERS ****
Scalar
        i_in_PK enthalpy /750/
        i_out_PK enthalpy /3400/
        i_out_TG enthalpy /2500/
        Q_demand Heat/hot-water production [MW] /120/
        P_demand Electric power pruduction [MW] /120/
        fuel_cost cost of fuel [CZK/MWh] /600/
        deviation_cost cost of deviation [CZK/MWh] /3000/
        ;

**** PARAMETERS ****
* data are "hard-wired" into the code so as to show
* how to define data array in GAMS
Parameter
         Mp_TG_min(TG) minimal steam flow on turbines
                 / TG1 60
                   TG2 60/
         Mp_TG_max(TG) maximalni steam flow on turbines
                 / TG1 320
                   TG2 420/
         ;
Table Qin_PK_char(PK, regions) input of boilers
           r1    r2      r3     r4
PK1        46.92 60      90     115.48
PK2        60.73 70      100    121.97
PK3        89.5  120      143    175
;
```

```
Table Qout_PK_char(PK, regions) outputs of boilers
           r1    r2      r3     r4
PK1        44.11 40      70     108.56
PK2        54.05 67      99     108.56
PK3        80    100     140    160
;

**** VARIABLES ****
Variables
         dev   possible difference in the electric power supplies
         v_dev auxiliary variable for linearizing abs() function
         w(PK,regions) variable for SOS2
         J total cost
         ;
Binary variables
         PK_state(PK)    boiler states
         TG_state(TG)    turbine states
         ;
* Declaring non-negative variables
Positive variables
         Mp_VK Steam flow to condenser [t per h]
         Mp_ZO Steam flow to water heater [t per h]
         Mp_PK Steam generated by boilers [t per h]
         Mp_TG(TG) Steam flow in turbines [t per h]
         P_TG(TG) Electric power generated by turbines [MW]
         Qin_PK(PK) Power input of boilers [MW]
         Qout_PK(PK) Power output of boilers [MW]
         ;

SOS2 Variable w;

**** CONSTRAINTS ****
Equations
         costs              production cost
         constraint1        steam generation
         constraint2        mass balance
         constraint3(TG)    minimal steam flow on turbines
         constraint4(TG)    maximal steam flow on turbines
         constraint5(TG)    el.power production
         constraint6        water heater power
         constraint7        desired el.power production
         constraint8(PK)    boiler constraint
```

```
            constraint9(PK)    boiler constraint
            constraint10(PK)   boiler constraint
            constraint11       steam generation by boilers
            abs1               linearizing abs
            abs2               linearizing abs
            ;

    * Objective function = min: fuel costs + electric power production costs
            costs.. J =e= deviation_cost*v_dev + fuel_cost*sum(PK,Qin_PK(PK));

    * 1. Steam generated on boilers is equal to the steam flow to turbines
            constraint1.. Mp_PK =e= sum(TG, Mp_TG(TG));

    * 2. Steam flow through turbines is equal to flow to condensation plus
    * flow to water heater
            constraint2.. Mp_VK + Mp_ZO =e= sum(TG, Mp_TG(TG));

    * 3. Minimal and maximal flow through turbines allowed
            constraint3(TG).. Mp_TG_min(TG)*TG_state(TG) =l= Mp_TG(TG);
            constraint4(TG).. Mp_TG(Tg) =l= Mp_TG_max(TG)*TG_state(TG);

    * 4. The amount of electric energy produced is equal to the entalpy difference
    * in the steam (eg. before and after the steam leaves turbines) [MW]
            constraint5(TG).. P_TG(TG) =e= Mp_TG(TG)*(i_out_PK - i_out_TG)/3600;

    * 5. Power of heating unit (heating water for further use in buildings) [MW]
            constraint6.. Q_demand =e= Mp_ZO*(i_out_TG - i_in_PK)/3600;

    * 6. Desired power and possible deviation [MW]
            constraint7.. P_demand =e= dev + sum(TG,P_TG(TG));

    ** Boiler constraints - using SOS2
    * Power input definition Qin = \sum_k \lambda_k QinPK_k
    constraint8(PK)..Qin_PK(PK)=e=sum(regions,(w(PK,regions)*Qin_PK_char(PK,regions)));

    * Power output definition Qout = \sum_k \lambda_k QoutPK_k
    constraint9(PK)..Qout_PK(PK)=e=sum(regions,(w(PK,regions)*Qout_PK_char(PK,regions)));

    * Output and input is non-zero only if the boiler is turned on
    constraint10(PK)..PK_state(PK) =e= sum(regions, w(PK,regions));

    * 7.Total power of boilers must be sufficient to boil desired
```

```
* amount of steam [MW]
constraint11.. Mp_PK*(i_out_PK - i_in_PK)/3600 =e= sum(PK, Qout_PK(PK));

* Linearizing abs()
        abs1.. v_dev >= dev;
        abs_2.. v_dev >= -1*dev;

**** SOLVE PROBLEM ****
Model problem /all/;
Option MIP = Cplex;
Solve problem using mip minimizing J;

**** SHOW RESULTS ****
Display dev.l, J.l;
```

D.3 OptimJ

```
/** --- BASIC MODEL IMPLEMENTATION --- **
Simplified cogeneration system:
- Three gas-fueled boilers with non-linear characteristics.
- Two turbines with linear characteristics.
- Condenser.
- Minimizing production cost of desired power and heat production.
*/
public model optimj_model solver cplex11 {
/*---- PARAMETERS ----*/
// min/max steam flow on turbines
        protected double[] Mp_TG_min[2];
        protected double[] Mp_TG_max[2];

// Power characteristics of boilers
        protected double[][] Qin_PK_char[3][4];
        protected double[][] Qout_PK_char[3][4];

// Enthalpy
        protected double i_in_PK;
        protected double i_out_PK;
        protected double i_out_TG;

// Scalar parameters
```

```
        protected double Q_demand = 40;
        protected double P_demand = 50;
        protected double fuel_cost = 600;
        protected double deviation_cost = 3000;

/*---- VARIABLES ----*/
// Boiler states (3 boilers, on/off)
        final var boolean[] PK_state[3];
// Turbine states
        final var boolean[] TG_state[2];
// Steam flow in turbines [t/h]
        final var double[] Mp_TG[2] in 0 .. Double.MAX_VALUE;
// Electric power generated on turbines [MW]
        final var double[] P_TG[2] in 0 .. Double.MAX_VALUE;
// Steam flow to condensator [t/h]
        final var double Mp_VK;
// Steam flow to water heater [t/h]
        final var double Mp_ZO;
// Steam generation on boilers [t/h]
        final var double Mp_PK;
/* Possible difference in the electric power supplied
 (positive = production is lower than planned)*/
        final var double dev;
// Power input of boilers [MW]
        final var double[] Qin_PK[3] in 0 .. Double.MAX_VALUE;
// Power output of boilers [MW]
        final var double[] Qout_PK[3] in 0 .. Double.MAX_VALUE;
// Auxiliary variable for SOS2
        final var double[][] w[3][4] in 0 .. Double.MAX_VALUE;

/*---- CONSTRAINTS ----*/
constraints {
// Non-negative variables
        Mp_PK >=0;
        Mp_ZO >=0;
        Mp_VK >=0;

// 1. Steam generated on boilers is equal to the steam flow to turbines
        Mp_PK == sum{int i : 0 .. Mp_TG.length-1}{Mp_TG[i]};

/*  2. Steam flow through turbines is equal to flow to condensation plus
 flow to water heater */
```

```
        Mp_VK + Mp_ZO == sum{int i : 0 .. Mp_TG.length-1}{Mp_TG[i]};

// 3. Minimal and maximal flow through turbines allowed
forall(int i : 0 .. Mp_TG.length-1) {
        Mp_TG_min[i]*?TG_state[i] <= Mp_TG[i];
        Mp_TG[i] <= Mp_TG_max[i]*?TG_state[i];
}

/* 4. The amount of electric energy produced is equal to the entalpy difference in
 the steam (eg. before and after the steam leaves turbines) [MW] */
sum{int i : 0 .. P_TG.length-1}{P_TG[i]} ==
        sum{int i : 0 .. Mp_TG.length-1}{Mp_TG[i]}*(i_out_PK - i_out_TG)/3600;

/* 5. Power of heating unit (heating water for further use in buildings) [MW]*/
        Q_demand == Mp_ZO*(i_out_TG - i_in_PK)/3600;

/*  6. Desired power and possible deviation [MW] */
        P_demand == sum{int i : 0 .. P_TG.length-1}{P_TG[i]} + dev;

/*-- Boiler constraints - using SOS2 -*/
/* Power input definition Qin = \sum_k \lambda_k QinPK_k */
forall(int i : 0 .. Qin_PK.length-1) {
Qin_PK[i] ==
        sum{int j : 0 .. Qin_PK_char[i].length-1}{w[i][j]*Qin_PK_char[i][j]};
}
/* Power output definition Qout = \sum_k \lambda_k QoutPK_k */
forall(int i : 0 .. Qout_PK.length-1) {
Qout_PK[i] ==
        sum {int j : 0 .. Qout_PK_char[i].length-1}{w[i][j]*Qout_PK_char[i][j]};
}
/* Output and input is non-zero only if the boiler is turned on */
forall(int i : 0 .. PK_state.length-1) {
        sum {int k : 0 .. w[i].length-1} {w[i][k]} == ?PK_state[i];
}
/* Declaration of SOS2 for each boiler (using special cplex function */
forall(int i : 0 .. w.length-1) {
        cplex11.SOS2(w[i], Qin_PK_char[i]);
}

/* 7.Total power of boilers must be sufficient to boil desired amount of
steam [MW] */
        sum{int i : 0 .. Qout_PK.length-1}{Qout_PK[i]} ==
```

```
                                                    Mp_PK*(i_out_PK - i_in_PK)/3600;
    }

    /*---- OBJECTIVE FUNCTION ----*/
    // min: fuel costs + electric power production costs
    minimize
    java.lang.Math.abs(dev)*deviation_cost +
                          sum{int i : 0 .. Qin_PK.length-1}{Qin_PK[i]*fuel_cost};

    /*---- MAIN METHOD ----*/
    public static void main (java . lang . String []args) {
            optimj_model problem = new optimj_model();
            Data d = new Data();
            LoadData ld = new LoadData();
            ld.init("params.xls", d);
            problem.fillData(d);
    try {
    /*---- EXTRACT MODEL ----*/
            problem.extract();
    /*--- SOLVE MODEL ----*/
            if (problem.solve()) {
    /*--- SHOW RESULTS ----*/
                    System.out.println(problem.objValue());
            }
            else {
                    System.out.println("No solution.");
            }
        }
    finally {
            problem.dispose();
        }
    }
    // other functions, whole java class can be found at the enclosed CD...
    //...
    }
```

D.4 AIMMS

```
1   ******** BASIC MODEL IMPLEMENTATION - EXPORTED TO TXT FILE **********
    * Simplified cogeneration system:
```

```
* - Three gas-fueled boilers with non-linear characteristics.
* - Two turbines with linear characteristics.
* - Condenser.
* - Minimizing production cost of desired power and heat production.

MAIN MODEL Main_model
******** SETS ********
  DECLARATION SECTION SetDeclaration
    SET:
       identifier :  boilers
       index      :  PK ;
    SET:
       identifier :  boiler_regions
       index      :  regions ;
    SET:
       identifier :  turbines
       index      :  TG ;
  ENDSECTION  ;

******** PARAMETERS ********
  DECLARATION SECTION ParameterDeclaration
* Minimal steam flow through turbines
    PARAMETER:
       identifier   :  Mp_TG_min
       index domain :  (TG) ;
* Maximal steam flow through turbines
    PARAMETER:
       identifier   :  Mp_TG_max
       index domain :  (TG) ;
* Power output boiler characteristics
    PARAMETER:
       identifier   :  Qout_PK_char
       index domain :  (PK,regions) ;
* Power input boiler characteristics
    PARAMETER:
       identifier   :  Qin_PK_char
       index domain :  (PK,regions) ;
  ENDSECTION  ;

***** SCALAR PARAMETERS *****
  DECLARATION SECTION ScalarDeclaration
    PARAMETER:
```

```
45          identifier   :  Q_demand ;
        PARAMETER:
           identifier   :  P_demand ;
        PARAMETER:
           identifier   :  deviation_cost ;
50      PARAMETER:
           identifier   :  fuel_cost ;

    * Enthalpy
        PARAMETER:
55         identifier   :  i_in_PK ;
           initial data : 750;
    * Enthalpy
        PARAMETER:
           identifier   :  i_out_PK
60         initial data :  3400 ;
    * Enthalpy
        PARAMETER:
           identifier   :  i_out_TG
           initial data :  2500 ; ;
65    ENDSECTION  ;

    ***** VARIABLES *****
      DECLARATION SECTION VariableDeclaration
    * Possible difference in the electric power supplied
70  * (positive = production is lower than planned)
        VARIABLE:
           identifier   :  dev ;
    * Auxiliary variables for SOS2
        VARIABLE:
75         identifier   :  w
           index domain :  (PK,regions) ;
    * Auxiliary variables for SOS2
        VARIABLE:
           identifier   :  w_bin
80         index domain :  (PK,regions)
           range        :  binary ;
    * Boiler states (3 boilers, on/off)
        VARIABLE:
           identifier   :  PK_state
85         index domain :  (PK)
           range        :  binary ;
```

```
* Turbine states (2 boilers, on/off)
    VARIABLE:
       identifier   :  TG_state
       index domain :  (TG)
       range        :  binary ;
* Steam flow to condensator [t/h]
    VARIABLE:
       identifier   :  Mp_VK
       range        :  nonnegative ;
* Steam flow to water heater [t/h]
    VARIABLE:
       identifier   :  Mp_ZO
       range        :  nonnegative ;
* Steam generated on boilers [t/h]
    VARIABLE:
       identifier   :  Mp_PK
       range        :  nonnegative ;
* Steam flow in turbines [t/h]
    VARIABLE:
       identifier   :  Mp_TG
       index domain :  (TG)
       range        :  nonnegative ;
* Electric power generated on turbines [MW]
    VARIABLE:
       identifier   :  P_TG
       index domain :  (TG)
       range        :  nonnegative ;
* Power input of boilers [MW]
    VARIABLE:
       identifier   :  Qin_PK
       index domain :  (PK)
       range        :  nonnegative ;
* Power output of boilers [MW]
    VARIABLE:
       identifier   :  Qout_PK
       index domain :  (PK)
       range        :  nonnegative ;
  ENDSECTION  ;

**** CONSTRAINTS ****
* Constraints are in different order than in Yalmip model
  DECLARATION SECTION ConstraintDeclaration
```

```
* 1. Steam generated on boilers is equal to the steam flow to turbines
    CONSTRAINT:
       identifier   :  constraint2
       definition   :  Mp_PK = sum(TG, Mp_TG(TG))  ;
* 3. Minimal and maximal flow through turbines allowed
    CONSTRAINT:
       identifier   :  constraint3
       index domain :  (TG)
       definition   :  Mp_TG_min(TG)*TG_state(TG) <= Mp_TG(TG)  ;
* 3. Minimal and maximal flow through turbines allowed
    CONSTRAINT:
       identifier   :  constraint4
       index domain :  (TG)
       definition   :  Mp_TG(TG) <= Mp_TG_max(TG)*TG_state(TG)  ;
* 4. The amount of electric energy produced is equal to the entalpy difference in
* the steam (eg. before and after the steam leaves turbines)
    CONSTRAINT:
       identifier   :  constraint5
       index domain :  (TG)
       definition   :  P_TG(TG) = Mp_TG(TG)*(i_out_PK - i_out_TG)/3600  ;
* 5. Power of heating unit (heating water for further use in buildings)
    CONSTRAINT:
       identifier   :  constraint6
       definition   :  Q_demand = Mp_ZO*(i_out_TG - i_in_PK)/3600  ;
* 6. Desired power and possible deviation
    CONSTRAINT:
       identifier   :  constraint7
       definition   :  P_demand = dev + sum(TG,P_TG(TG))  ;
* 2. Steam flow through turbines is equal to flow to condensation plus
* flow to water heater
    CONSTRAINT:
      identifier    : constraint13
      definition    : Mp_VK + Mp_ZO = sum(TG, Mp_TG(TG))   ;
** BOILER CONSTRAINTS
* 7.Total power of boilers must be sufficient to boil desired amount of steam
    CONSTRAINT:
       identifier   :  constraint12
       definition   :  Mp_PK*(i_out_PK - i_in_PK)/3600 = sum(PK, Qout_PK(PK))  ;
* Power input definition
    CONSTRAINT:
    identifier   :  constraint8_SOS2
    index domain :  (PK)
```

```
    definition   :  Qin_PK(PK)= sum(regions,(w(PK,regions)*Qin_PK_char(PK,regions)));
* Power output definition
    CONSTRAINT:
    identifier   :  constraint9_SOS2
    index domain :  (PK)
    definition   :  Qout_PK(PK)=sum(regions,(w(PK,regions)*Qout_PK_char(PK,regions)));
* Output and input is non-zero only if the boiler is turned on
    CONSTRAINT:
       identifier   :  constraint10_SOS2
       index domain :  (PK)
       property     :  Sos2
       definition   :  PK_state(PK) = sum(regions, w(PK,regions))  ;
  ENDSECTION  ;

***** OBJECTIVE FUNCTION *****
  DECLARATION SECTION MathematicalProgramDeclaration
    MATHEMATICAL PROGRAM:
       identifier :  costs
       objective  :  ObjectiveFunction
       direction  :  minimize
       type       :  MILP ;

    VARIABLE:
       identifier :  ObjectiveFunction
       definition :  deviation_cost*abs(dev) + fuel_cost*sum[PK,Qin_PK(PK)]  ;
  ENDSECTION  ;
  **********************************
```

D.5 AMPL

```
### BASIC MODEL IMPLEMENTATION ###
# Simplified cogeneration system:
# - Three gas-fueled boilers with non-linear characteristics.
# - Two turbines with linear characteristics.
# - Condenser.
# - Minimizing production cost of desired power and heat production.
#
# This is a model file, the files ampl_model.dat and ampl_model.run
# are needed in order to run this model

```

```
#-- SETS --#
set PK; # boilers
set TG; # turbines
# number of regions in boilers's power characteristic
set regions ordered;

##-- PARAMETERS --##
# scalar parameters
param deviation_cost;
param fuel_cost;
param Q_demand;
param P_demand;

# Enthalpy
param i_in_PK;
param i_out_PK;
param i_out_TG;

# Mininal and maximal flow through turbines allowed
# equivalent notation
param Mp_TG_min {TG};
param Mp_TG_max {i in TG};

# Power input and output boiler characteristics
param Qin_PK_char{i in PK, j in regions};
param Qout_PK_char{i in PK, j in regions};
param coef{i in PK, j in 1..3}; # auxiliary parameter

##-- VARIABLES --##
var TG_state {i in TG} binary; # Turbine states
var PK_state {i in PK} binary; # Boiler states (3 boilers, on/off)
# Possible difference in the electric power supplied
# (positive = production is lower than planned)
var dev;
var v_dev; # auxiliary variable for linearizing abs()

var Mp_VK >= 0; # Steam flow to condensator [t / h]
var Mp_ZO >= 0; # Steam flow to water heater [t / h]
var Mp_PK >= 0; # Steam generated on boilers [t / h]
var Mp_TG {i in TG} >= 0; # Steam flow in turbines [t / h]
var P_TG {i in TG} >= 0; # El. power generated on turbines [MW]
var Qin_PK {PK} >= 0; # Power input of boilers [MW]
```

```
var Qout_PK {i in PK} >= 0; # Power output of boilers [MW]

##-- OBJECTIVE FUNCTION --#
# min: fuel costs + electric power production costs
minimize Cost: deviation_cost*v_dev + fuel_cost * sum{i in PK} Qin_PK[i];

##-- CONSTRAINTS --##
subject to abs_1: v_dev >= dev;
subject to abs_2: v_dev >= -1*dev;

# 1. Steam generated on boilers is equal to the steam flow to turbines
subject to c1: Mp_PK = sum{i in TG} Mp_TG[i];

# 2. Steam flow through turbines is equal to flow to condensation plus
# flow to water heater
subject to c2: Mp_VK + Mp_ZO = sum{i in TG} Mp_TG[i];

# 3. Minimal and maximal flow through turbines allowed
subject to c3Min{j in TG}: Mp_TG[j] >= Mp_TG_min[j]*TG_state[j];
subject to c3Maxk{j in TG}: Mp_TG[j] <= Mp_TG_max[j]*TG_state[j];

# 4. The amount of electric energy produced is equal to the entalpy difference in
# the steam (eg. before and after the steam leaves turbines) [MW]
subject to c4{g in TG}: P_TG[g] = Mp_TG[g]*(i_out_PK - i_out_TG)/3600;

# 5. Power of heating unit (heating water for further use in buildings)
subject to c5: Q_demand = Mp_ZO*(i_out_TG - i_in_PK)/3600; # [MW]

# 6. Desired power and possible deviation [MW]
subject to c6: P_demand = dev + sum{g in TG} P_TG[g];

# 7.Total power of boilers must be sufficient to boil desired amount of
# steam [MW]
subject to c7: Mp_PK*(i_out_PK - i_in_PK)/3600 = sum{i in PK} Qout_PK[i];

##-- PIECEWISE LINEAR FUNCTION --##
# instead of SOS2
# << breakpoints, slope_list >> variable;
subject to Sos2 {i in PK}:
<<{j in regions}Qin_PK_char[i,j];0,{j in 1..3}coef[i,j],0>>Qout_PK[i]*PK_state[i]>=0;
```

D.6 LINGO

```
!!--- BASIC MODEL IMPLEMENTATION ---!!
 Simplified cogeneration system:
 - Three gas-fueled boilers with non-linear characteristics.
 - Two turbines with linear characteristics.
 - Condenser.
 - Minimizing production cost of desired power and heat production.;

!---- SETS ---- !;
SETS:
! Boilers + their power characteristics;
PK/PK1,PK2,PK3/:PK_state, Qin_PK, Qout_PK;
regions/r1,r2,r3,r4/; ! regions - auxiliary set;
! Power input and output of boilers, w is auxiliary variable for SOS2;
Q_PK_char(PK,regions):value_in, value_out, w;
! Turbine parameters, states, el.power output and steam flow;
TG:Mp_TG_min, Mp_TG_max, TG_state, Mp_TG, P_TG;
ENDSETS

!---- PARAMETERS ---!;
DATA:
! Load parameters from MS Excel file
Absolute address has to be used
eg. 'C:\Documents and Settings\podhrmic\...file.xls!;
Mp_TG_min, Mp_TG_max,value_in,value_out = @OLE('params.xls');
i_in_PK,i_out_PK,i_out_TG = @OLE('params.xls');

! scalar parameters;
deviation_cost = 3000;
fuel_cost = 600;
P_demand = 40;
Q_demand = 50;
ENDDATA

!---- OBJECTIVE FUNCTION ----!
min: fuel costs + electric power production costs
abs() needs to be linearized in order to keep the model
as linear. Go to LINGO-Options-Model Generator-Linearization;
MIN = deviation_cost*@ABS(dev) + fuel_cost*@SUM(PK:Qin_PK);

!---- VARIABLES ----!
```

```
! Variables are declared in the equations
 (constraints section);
Declaring binary variables;
@FOR( TG:
        @BIN( TG_state);
);
@FOR( PK:
        @BIN( PK_state);
);

!---- CONSTRAINTS ----!
! 1. Steam generated on boilers is equal to the steam flow to turbines;
Mp_PK = @SUM(TG:Mp_TG);

! 2. Steam flow through turbines is equal to flow to condensation plus
 flow to water heater;
Mp_VK + Mp_ZO = @SUM(TG:Mp_TG);

! 3. Minimal and maximal flow through turbines allowed;
@FOR(TG:
        Mp_TG_min*TG_state <= Mp_TG;
        Mp_TG <= Mp_TG_max*TG_state;
);

! 4. The amount of electric energy produced is equal to the entalpy difference
in the steam (eg. before and after the steam leaves turbines) [MW];
@FOR(TG:
        P_TG = Mp_TG*(i_out_PK - i_out_TG)/3600;
);

! 5. Power of heating unit (heating water for further use in buildings) [MW];
Q_demand = Mp_ZO*(i_out_TG - i_in_PK)/3600;

! 6. Desired power and possible deviation;
P_demand = dev + @SUM(TG:P_TG);

! 7.Total power of boilers must be sufficient to boil desired amount of
 steam [MW];
Mp_PK*(i_out_PK - i_in_PK)/3600 = @SUM(PK: Qout_PK);

!--- Boiler constraints - using SOS2 ----!;
@FOR(PK(I):
```

```
! Power input definition Qin = \sum_k \lambda_k QinPK_k;
        Qin_PK(I) = @SUM(Q_PK_Char(I,J) : value_in(I,J)*W(I,J));
! Power output definition Qout = \sum_k \lambda_k QoutPK_k ;
        Qout_PK(I) = @SUM(Q_PK_Char(I,J) : value_out(I,J)*W(I,J));
! Output and input is non-zero only if the boiler is turned on;
        @SUM(Q_PK_Char(I,J):W(I,J)) = PK_state(I);
! Declaration of SOS2 for each boiler;
        @FOR(Q_PK_Char(I,J):
                @SOS2('W_' + PK( I), W( I, J))
        );
);
```

D.7 MPL

```
!!--- BASIC MODEL IMPLEMENTATION ---!!
! Simplified cogeneration system:
! - Three gas-fueled boilers with non-linear characteristics.
! - Two turbines with linear characteristics.
! - Condenser.
! - Minimizing production cost of desired power and heat production.
Title
mpl_model;

!---- SETS ----!
INDEX
PK = (PK1,PK2,PK3);
TG = (TG1,TG2);
regions = (r1,r2,r3,r4);

!----- PARAMETERS ----!
DATA
! scalar values !
Q_demand := 140;
P_demand := 120;
fuel_cost := 600;
deviation_cost := 3000;

!--- LOADING PARAMS FROM EXCEL --- !
! Enthalpy !
i_in_PK = EXCELRANGE("param_MPL.xls", "C9");
```

```
i_out_PK = EXCELRANGE("param_MPL.xls", "C10");
i_out_TG = EXCELRANGE("param_MPL.xls", "C11");

! Power characteristics of boilers
Qin_PK_char[PK, regions] = EXCELRANGE("param_MPL.xls", "value_in");
Qout_PK_char[PK, regions] = EXCELRANGE("param_MPL.xls", "value_out");

! min/max steam flow on turbines
Mp_TG_min[TG] = EXCELRANGE("param_MPL.xls", "MP_TG_min");
Mp_TG_max[TG] = EXCELRANGE("param_MPL.xls", "MP_TG_max");

!--- VARIABLES ---!
DECISION VARIABLES
dev; ! Possible difference in the electric power supplied
v_dev; ! auxiliary variable for linearizing abs()
Mp_VK; ! Steam flow to condensator [t / h]
Mp_ZO; ! Steam flow to water heater [t / h]
Mp_PK; ! Steam generated on boilers [t / h]
Mp_TG[TG]; ! Steam flow in turbines [t / h]
P_TG[TG]; ! Electric power generated on turbines [MW]
Qin_PK[PK]; ! Power input of boilers [MW]
Qout_PK[PK]; ! Power output of boilers [MW]
PK_state[PK]; ! Boiler states (3 boilers, on/off)
TG_state[TG]; ! Turbine states
w[PK,regions]; ! auxiliary variable for SOS2

!-- Special Ordered Set type2 declaration--!
SOS2
 s[PK]: SET( regions : w[PK,regions]);

!--- OBJECTIVE FUNCTION ---!
MODEL
! min: fuel costs + electric power production costs
MIN Cost = deviation_cost*v_dev+ fuel_cost*sum(PK : Qin_PK);

!--- CONSTRAINTS ---- !
SUBJECT TO
! 1. Steam generated on boilers is equal to the steam flow to turbines
Mp_PK = sum(TG : Mp_TG);

!  2. Steam flow through turbines is equal to flow to condensation plus
! flow to water heater
```

```
Mp_VK + Mp_ZO = sum(TG : Mp_TG);

! 3. Minimal and maximal flow through turbines allowed
min_tg[TG] : Mp_TG_min[TG]*TG_state[TG] <= Mp_TG[TG];
max_tg[TG] : Mp_TG_max[TG]*TG_state[TG] >= Mp_TG[TG];

! 4. The amount of electric energy produced is equal to the entalpy difference in
! the steam (eg. before and after the steam leaves turbines) [MW]
power_tg[TG] : P_TG[TG] = Mp_TG[TG]*(i_out_PK - i_out_TG)/3600;

! 5. Power of heating unit (heating water for further use in buildings) [MW]
Q_demand = Mp_ZO*(i_out_TG - i_in_PK)/3600;

! 6. Desired power and possible deviation
P_demand = dev + sum(TG : P_TG);

! 7.Total power of boilers must be sufficient to boil desired amount of
! steam [MW]
Mp_PK*(i_out_PK - i_in_PK)/3600 = sum(PK : Qout_PK);

!--- Boiler constraints - using SOS2 ----!
! Power input definition Qin = \sum_k \lambda_k QinPK_k
sos_1[PK] : Qin_PK(PK)=sum(regions : (w[PK, regions]*Qin_PK_char[PK, regions]));

! Power output definition Qout = \sum_k \lambda_k QoutPK_k
sos_2[PK] : Qout_PK(PK)=sum(regions : (w[PK, regions]*Qout_PK_char[PK, regions]));

! Output and input is non-zero only if the boiler is turned on
sos_3[PK] : PK_state(PK) = sum(regions : w[PK,regions]);

! Linearizing abs()
v_dev >= dev;
v_dev >= -dev;

!--- BINARY VARIABLES --- !
! Needed to be declared here !
BINARY
        PK_state[PK];
        TG_state[TG];
```

D.8 ZIMPL

```
#### BASIC MODEL IMPLEMENTATION ###
# Simplified cogeneration system:
# - Three gas-fueled boilers with non-linear characteristics.
# - Two turbines with linear characteristics.
# - Condenser.
# - Minimizing production cost of desired power and heat production.
##-- SETS --##
set PK := { "PK1" , "PK2", "PK3"};
set TG := {"TG1", "TG2"};
set regions := {"r1","r2","r3","r4"};

##-- PARAMETERS --##
# parameters can be loaded from a text file, but
# data are "hard-wired" into the code so as to show
# how to define data array in Zimpl
# scalar parameters
param Q_demand := 50;
param P_demand := 40;
param deviation_cost := 3000;
param fuel_cost := 600;

# Enthalpy
param i_in_PK := 750;
param i_out_PK := 3400;
param i_out_TG := 2500;

# Turbines min/max steam flow
param Mp_TG_min[TG] := <"TG1"> 60, <"TG2"> 60;
param Mp_TG_max[TG] := <"TG1"> 320, <"TG2"> 420;

# Power characteristics of boilers
param Qin_PK_char[PK*regions] := | "r1", "r2", "r3", "r4" |
                              |"PK1"|46.92, 60  ,  90 ,115.48|
                              |"PK2"|60.73, 70  , 100 ,121.97|
                              |"PK3"| 89.5, 120 , 143 ,175   |;
param Qout_PK_char[PK*regions] := | "r1", "r2", "r3", "r4" |
                               |"PK1"|44.11, 40  ,  70 ,108.56|
                               |"PK2"|54.05, 67  ,  99 ,108.56|
                               |"PK3"| 80  , 100 , 140 ,160   |;

```

```
##-- VARIABLES --##
var TG_state[TG] binary; # Turbine states
var PK_state[PK] binary; # Boiler states (3 boilers, on/off)
var dev; # Possible difference in the el.power supplied [MW]
var v_dev; # auxiliary variable

var Mp_VK >= 0 <= infinity; # steam flow to condensator [t / h]
var Mp_ZO >= 0; # steam flow to water heater [t / h]
var Mp_PK >= 0; # Steam generated on boilers [t / h]
var Mp_TG [TG] >= 0; # Steam flow in turbines [t / h]
var P_TG [TG] >= 0; # El.power generated on turbines[MW]
var Qin_PK [PK] >= 0; # Power input of boilers [MW]
var Qout_PK [PK] >= 0; # Power output of boilers [MW]

var w [PK*regions]; # auxiliary variable for SOS2

##-- OBJECTIVE FUNCTION --##
# min: fuel costs + electric power production costs
minimize cost: deviation_cost*v_dev  + fuel_cost * sum <i> in PK: Qin_PK[i];

##-- CONSTRAINTS --##
subto abs_1: v_dev >= dev;
subto abs_2: v_dev >= -1*dev;

# 1. Steam generated on boilers is equal to the steam flow to turbines
subto c1: Mp_PK == sum <i> in TG: Mp_TG[i];

# 2. Steam flow through turbines is equal to flow to condensation plus
# flow to water heater
subto c2: Mp_VK + Mp_ZO == sum <i> in TG: Mp_TG[i];

# 3. Minimal and maximal flow through turbines allowed
subto c3min:
forall <j> in TG do Mp_TG[j] >= Mp_TG_min[j]*TG_state[j];

# 3. Minimal and maximal flow through turbines allowed
subto c3max:
forall <j> in TG do Mp_TG[j] <= Mp_TG_max[j]*TG_state[j];

# 4. The amount of electric energy produced is equal to
# the entalpy difference in the steam
# (eg. before and after the steam leaves turbines) [MW]
```

```
subto c4:
forall <g> in TG do P_TG[g] == Mp_TG[g]*(i_out_PK - i_out_TG)/3600;

# 5. Power of heating unit (heating water for further use in buildings) [MW]
subto Heat_demand: Q_demand == Mp_ZO*(i_out_TG - i_in_PK)/3600;

# 6. Desired power and possible deviation [MW]
subto Power_demand: P_demand == dev + sum <g> in TG: P_TG[g];

# 7.Total power of boilers must be sufficient to boil desired amount of
# steam [MW]
subto Boiler_power: Mp_PK*(i_out_PK-i_in_PK)/3600==sum <i> in PK: Qout_PK[i];

##-- Boiler constraints - using SOS2 --##
# Power input definition Qin = \sum_k \lambda_k QinPK_k
subto total_cost_x:
forall <i> in PK:
        Qin_PK[i] == sum <j> in regions: (Qin_PK_char[i,j]*w[i,j]);

# Power output definition Qout = \sum_k \lambda_k QoutPK_k
subto weights_of_fx:
forall <i> in PK:
        Qout_PK[i] == sum <j> in regions: (Qout_PK_char[i,j]*w[i,j]);

# Output and input is non-zero only if the boiler is turned on
subto weights_must_sum_to_PK_state:
forall <i> in PK:
        PK_state[i] == sum <j> in regions: w[i,j];

# Declaration of SOS2 for each boiler
sos s1: forall <j> in regions: type2: w["PK1",j];
sos s2: forall <j> in regions: type2: w["PK2",j];
sos s3: forall <j> in regions: type2: w["PK3",j];
```

Appendix E

Source codes of extended model implementation

E.1 Yalmip

```
%% --- EXTENDED MODEL IMPLEMENTATION --- **
%* Simplified cogeneration system:
%* @param t - number of hours
%* @param m - number of gas boilers with non-linear characteristic
%* @param n - number of turbines with linear characterictic.
%* @param l - number of breakpoints in PWL function
%*
%* Boilers characteristics are described by piecewise linear function,
%* with 4 breakpoints. Deviation and fuel cost are constant, as well as
%* Q_demand and P_demand.
%*
%* Parameters are set in benchmark.m script
%*
%*/

% Clear internal yalmip cache for better performance
yalmip('clear')

%% Start counting
tic

% Boiler status
PK_state = binvar(m,t,'full');
```

```
% Auxiliary variables for boiler's non-linear characteristics
% It is not a very efective formulation and it is desirable to
% reformulate it using SOS2
% Currently each boiler is apporximated using three linear sections
PK_regions = binvar(m,l,'full'); %

% Turbine status
TG_state = binvar(n,t,'full');

% Steam generated by boilers [t/h]
Mp_PK = sdpvar(t,1,'full');

% Power output of boilers [MW]
% 1st dimension = time, 2nd dim = boiler, 3rd dim = output in a given section
Qout_PK = sdpvar(m,l,t,'full');

% Power input of boilers [MW]
% 1st dimension = time, 2nd dim = boiler, 3rd dim = input in a given section
Qin_PK = sdpvar(m,l,t,'full');

% Steam flow through turbines [t/h]
Mp_TG = sdpvar(n,t,'full');

% Electric power generated by turbines [MW]
P_TG = sdpvar(n,t,'full');

% Possible difference in the electric power supplied
% (positive = production is lower than planned)
dev = sdpvar(t,1);

% Steam flow to condenser [t/h]
Mp_VK = sdpvar(t,1);

% Steam flow to water heater [t/h]
Mp_ZO = sdpvar(t,1);


%% ---- CONSTRAINTS  ---%%
% Set of constraints
F = [];

% smernice primek
```

```
smernice = diff(Qout_PK_char,1,2)./diff(Qin_PK_char,1,2);
ofset = Qout_PK_char(:,1:end-1) - smernice.*Qin_PK_char(:,1:end-1);

% formulate these constraints for each hour
for i=1:t
    % do
% Non-negative variables
F = F + [Mp_PK(i)>=0; Mp_TG(:,i)>=0; Mp_VK(i)>=0; Mp_ZO(i)>=0;];

% 1. Steam generated on boilers is equal to the steam flow to turbines
F = F + [sum(Mp_PK(i)) == sum(Mp_TG(:,i))];

% 2. Steam flow through turbines is equal to flow to condensation plus
% flow to water heater
F = F + [sum(Mp_TG(:,i)) == Mp_VK(i) + Mp_ZO(i)];

% 3. Minimal and maximal flow through turbines allowed
F = F + [Mp_TG_min.*TG_state(:,i) <= Mp_TG(:,i) <= Mp_TG_max.*TG_state(:,i)];

% 4. The amount of electric energy produced is equal to the entalpy difference in
% the steam (eg. before and after the steam leaves turbines)
F = F + [P_TG(:,i) == Mp_TG(:,i)*(i_out_PK - i_out_TG)/3600]; % [MW]

% 5. Power of heating unit (heating water for further use in buildings)
F = F + [Q_demand(i) == Mp_ZO(i)*(i_out_TG - i_in_PK)/3600]; % [MW]

% 6. Desired power and possible deviation
F = F + [sum(P_TG(:,i)) + dev(i) == P_demand(i)]; % [MW]

%% Boiler constraints %%
% Relation between power input and output of boiler
% non-linear characteristics
% (desirable to reformulate using SOS2)

% 7.Total power of boilers must be sufficient to boil desired amount of
% steam
F = F + [sum(sum(Qout_PK(:,:,i))) == Mp_PK(i)*(i_out_PK - i_in_PK)/3600]; % [MW]

% If a boiler operates only its power could vary only on one power section
F = F + [sum(PK_regions,2) == PK_state(:,t)];

% Input and ouput connection
```

```
% 1. Power output has to be in one section only
F = F + [Qin_PK_char(:,1:end-1).*PK_regions
                        <= Qin_PK(:,:,i) <= Qin_PK_char(:,2:end).*PK_regions];

% 2. Power output is set by lines with angular coeficients
F = F + [Qout_PK(:,:,i) == smernice.*Qin_PK(:,:,i) + ofset.*PK_regions];
end


%% ---- OBJECTIVE FUNCTION ----%%
% J = fuel costs + electric power production costs
J = fuel_cost*sum(sum(sum(Qin_PK))) + deviation_cost*abs(sum(dev));

%% Stop counting
res.time = toc;
res;
```

E.2 OptimJ

```
package time.variant;

import data.handling.*;
import data.definition.*;

import java.awt.print.Printable;
import java.util.Random;


/** --- EXTENDED MODEL IMPLEMENTATION --- **
* Simplified cogeneration system:
* @param t - number of hours
* @param m - number of gas boilers with non-linear characteristic
* @param n - number of turbines with linear characterictic.
*
* Boilers characteristics are described by piecewise linear function,
* with 4 breakpoints. Deviation and fuel cost are constant, as well as
* Q_demand and P_demand.
*
*/
public model extended_model extends ExtendedModelParams solver cplex11 {
```

```
/**
 * Constructor
 * @param t - number of hours
 * @param m - number of gas boilers with non-linear characteristic
 * @param n - number of turbines with linear characterictic.
 */
extended_model(int hours, int boilers, int turbines) {
 super(hours, boilers, turbines);
}
/*---- PARAMETERS ----*/
//min/max steam flow through turbines [t/h]
protected double[] Mp_TG_min[n];
protected double[] Mp_TG_max[n];
//  Power characteristics of boilers
// row -> boiler, column -> input/output
protected double[][] Qin_PK_char[m][breakpoints];
protected double[][] Qout_PK_char[m][breakpoints];
// Enthalpy
protected double i_in_PK;
protected double i_out_PK;
protected double i_out_TG;
// Desired heat and el. power production
protected double[] Q_demand[t]; // Heat production [MW]
protected double[] P_demand[t]; // Electric power pruduction [MW]

protected double deviation_cost; // [CZK]
protected double fuel_cost; //  [CZK]

/*---- VARIABLES ----*/
// Turbine status
final var boolean[][] TG_state[t][n];
// Boiler status
final var boolean[][] PK_state[t][m];
// Steam flow through turbines [t/h]
final var double[][] Mp_TG[t][n] in 0 .. Double.MAX_VALUE;
// Electric power produced by generators [MW]
final var double[][] P_TG[t][n] in 0 .. Double.MAX_VALUE;
// Steam flow to condenser [t/h]
final var double[] Mp_VK[t];
// Steam flow to the water heater [t/h]
final var double[] Mp_ZO[t];
// Steam flow from boilers [t/h]
```

```
final var double[] Mp_PK[t];
// contracting penalty [CZK for each MW]
final var double[] dev[t];
//       Power input of boilers [MW]
// row -> hour, column -> boiler
final var double[][] Qin_PK[t][m] in 0 .. Double.MAX_VALUE;
//       Power output of boilers [MW]
// row -> hour, column -> boiler
final var double[][] Qout_PK[t][m] in 0 .. Double.MAX_VALUE;
//       Auxiliary variable for SOS2 (lambda)
// 1st dimension -> hour, 2nd dimension -> boiler, 3rd dimension -> lambda variables
final var double[][][] w[t][m][breakpoints] in 0 .. Double.MAX_VALUE;

/*---- CONSTRAINTS ----*/
constraints {
// Formulate the following constraints for each hour:
forall(int h : 0 .. t-1) {
// Assume variables to be non-negative
  Mp_PK[h] >=0;
  Mp_ZO[h] >=0;
  Mp_VK[h] >=0;

// 1. Steam generated on boilers is equal to the steam flow to turbines
  Mp_PK[h] == sum{int i : 0 .. Mp_TG[h].length-1}{Mp_TG[h][i]};

// 2. Steam flow through turbines is equal to flow to condensation plus
// flow to water heater
  Mp_VK[h] + Mp_ZO[h] == sum{int i : 0 .. Mp_TG[h].length-1}{Mp_TG[h][i]};

// 3. Minimal and maximal steam flow through turbines allowed
forall(int i : 0 .. Mp_TG[h].length-1) {
  Mp_TG_min[i]*?TG_state[h][i] <= Mp_TG[h][i];
  Mp_TG[h][i] <= Mp_TG_max[i]*?TG_state[h][i];
}

// 4. The amount of electric energy produced is equal to the entalpy difference
// in the steam (eg. before and after the steam leaves turbines) [MW]
sum{int i : 0 .. P_TG[h].length-1}{P_TG[h][i]} ==
     sum{int i : 0 .. Mp_TG[h].length-1}{Mp_TG[h][i]}*(i_out_PK - i_out_TG)/3600;

// 5. Power of heating unit (heating water for further use in buildings) [MW]
  Q_demand[h] == Mp_ZO[h]*(i_out_TG - i_in_PK)/3600;
```

```
// 6. Desired power and possible deviation [MW]
  P_demand[h] == sum{int i : 0 .. P_TG[h].length-1}{P_TG[h][i]} + dev[h];

//----  Boiler constraints - using SOS2 ---- //
// Power input definition
forall(int i : 0 .. Qin_PK[h].length-1) {
  Qin_PK[h][i] ==
    sum{int j : 0 .. Qin_PK_char[i].length-1}{w[h][i][j]*Qin_PK_char[i][j]};
}

// Power output definition
forall(int i : 0 .. Qout_PK[h].length-1) {
  Qout_PK[h][i] ==
   sum {int j : 0 .. Qout_PK_char[i].length-1}{w[h][i][j]*Qout_PK_char[i][j]};
}

// Output and input is non-zero only if the boiler is turned on.
forall(int i : 0 .. PK_state[h].length-1) {
  sum {int k : 0 .. w[h][i].length-1} {w[h][i][k]} == ?PK_state[h][i];
}

// SOS2 declaration
forall(int i : 0 .. w[h].length-1) {
  cplex11.SOS2(w[h][i], Qin_PK_char[i]);
}

// 7.Total power of boilers must be sufficient to boil desired
// amount of steam [MW]
  sum{int i : 0 .. Qout_PK[h].length-1}{Qout_PK[h][i]} ==
                                      Mp_PK[h]*(i_out_PK - i_in_PK)/3600;
 }
}

/*---- OBJECTIVE FUNCTION ----*/
// min: fuel costs + electric power production costs
minimize
sum{int h : 0 .. t-1}{
 java.lang.Math.abs(dev[h])*deviation_cost+
        sum{int i : 0 .. Qin_PK[h].length-1}{Qin_PK[h][i]*fuel_cost}
 };
}
```

```
// Superclass
class ExtendedModelParams {
final int t; // number of hours
final int m; // number of boilers
final int n; // number of turbines
final int breakpoints = 4; // number of breakpoints

 ExtendedModelParams(int time, int boilers, int turbines) {
  this.t = time;
  this.m = boilers;
  this.n = turbines;
 }
}
```

E.3 Zimpl

```
#### EXTENDED MODEL IMPLEMENTATION ###
# Simplified cogeneration system:
# - m gas boilers with non-linear characteristic.
# - n turbines with linear characterictic.
# - t hour planning horizon
# - one Condensator.
# - for extraction time benchmark.


# -----------
# SETS
# -----------
param hours := read "data.dat" as "1n";
param boilers := read "data.dat" as "2n";
param turbines := read "data.dat" as "3n";

set PK := { 1..boilers};
set TG := { 1..turbines};
# regions = breakpoints in boilers power char.
set regions := {"r1","r2","r3","r4"};
set t := { 1..hours };

# -----------
```

```
# PARAMETERS
# -----------
# scalar parameters
param Q_demand := 250;
param P_demand := 240;
param deviation_cost := 3000;
param fuel_cost := 600;

# Enthalpy
param i_in_PK := 750;
param i_out_PK := 3400;
param i_out_TG := 2500;

# Turbines min/max steam flow
# same params for all turbines
param Mp_TG_min := 60;
param Mp_TG_max := 320;

# Power characteristics of boilers
## same characteristics for all boilers
param Qin_PK_char[regions] :=  <"r1"> 46.92, <"r2"> 60, <"r3"> 90, <"r4">  115.48;
param Qout_PK_char[regions] := <"r1"> 44.11, <"r2"> 40, <"r3"> 70, <"r4"> 108.56;

# -------------
# VARIABLES
# -----------
var TG_state[t*TG] binary; # Turbine status
var PK_state[t*PK] binary; # Boiler status
var dev[t]; # Possible difference in the el.power supplied [MW]
var v_dev[t]; # auxiliary variable for reformulation abs()

var Mp_VK[t] >= 0 <= infinity; # flow to condenser [t per h]
var Mp_ZO[t] >= 0; # flow to water heater [t per h]
var Mp_PK[t] >= 0; # Steam generated on boilers [t per h]
var Mp_TG [t*TG] >= 0; # Steam flow in turbines [t per h]
var P_TG [t*TG] >= 0; # El.power generated on turbines[MW]
var Qin_PK [t*PK] >= 0; # Power input of boilers [MW]
var Qout_PK [t*PK] >= 0; # Power output of boilers [MW]

var w [t*PK*regions]; #  auxiliary variable for SOS2
```

```
# -------------
# CONSTRAINTS
# ------------
## min: fuel costs + electric power production costs
minimize cost: deviation_cost*sum<j> in t: v_dev[j]  +
            fuel_cost*sum<j> in t: (sum <i> in PK: Qin_PK[j,i]);

# reformulating absolute value
subto abs1:
        forall <h> in t do
                        v_dev[h] >= dev[h];

# reformulating absolute value
subto abs2:
        forall <h> in t do
                        v_dev[h] >= -1*dev[h];

# 1. Steam generated on boilers is equal to the steam flow to turbines
subto c1:
        forall <h> in t do
                        Mp_PK[h] == sum <i> in TG: Mp_TG[h,i];

# 2. Steam flow through turbines is equal to flow to condensation plus
# flow to water heater
subto c2:
        forall <h> in t do
                Mp_VK[h] + Mp_ZO[h] == sum <i> in TG: Mp_TG[h,i];

# 3. Minimal allowed steam flow through turbines
subto c3min:
        forall <h> in t do
                forall <j> in TG do Mp_TG[h,j] >= Mp_TG_min*TG_state[h,j];

# 3. Maximal allowed steam flow through turbines
subto c3max:
        forall <h> in t do
                forall <j> in TG do Mp_TG[h,j] <= Mp_TG_max*TG_state[h,j];

# 4. The amount of electric energy produced is equal to
# the entalpy difference in the steam
# (eg. before and after the steam leaves turbines) [MW]
subto c4:
```

```
        forall <h> in t do
                forall <g> in TG do P_TG[h,g] == Mp_TG[h,g]*(i_out_PK - i_out_TG)/3600;

# 5. Power of heating unit (heating water for further use in buildings) [MW]
subto c5:
        forall <h> in t do
                 Q_demand == Mp_ZO[h]*(i_out_TG - i_in_PK)/3600;

# 6. Desired power and possible deviation [MW]
subto c6:
        forall <h> in t do
                P_demand == dev[h] + sum <g> in TG: P_TG[h,g];

# 7.Total power of boilers must be sufficient to boil desired amount of
# steam [MW]
subto c7:
        forall <h> in t do
                 Mp_PK[h]*(i_out_PK - i_in_PK)/3600 == sum <i> in PK: Qout_PK[h,i];

# -------
# Boiler constraints - using SOS2
# ------
# Power input definition
subto total_cost_x:
        forall <h> in t do
                forall <i> in PK:
        Qin_PK[h,i] == sum <j> in regions: (Qin_PK_char[j]*w[h,i,j]);

# Power output definition
subto weights_of_fx:
        forall <h> in t do
                forall <i> in PK:
        Qout_PK[h,i] == sum <j> in regions: (Qout_PK_char[j]*w[h,i,j]);

# Output and input is non-zero only if the boiler is turned on
subto weights_must_sum_to_PK_state:
        forall <h> in t do
                forall <i> in PK:
        PK_state[h,i] == sum <j> in regions: w[h,i,j];

#  Declaration of SOS2 for each boiler
sos s1: forall <h> in t:
```

```
150                 forall <k> in PK:
                            forall <j> in regions:
                            type2: w[h,k,j];
```