Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

Mari Elise Rugland

An object-oriented framework for the optimization of flexible renewable energy systems

Master's thesis in Chemical Engineering and Biotechnology
Supervisor: Johannes Jäschke and Truls Gundersen
Co-supervisor: Avinash S. R. Subramanian
June 2021

NTNU
Norwegian University of
Science and Technology

Mari Elise Rugland

An object-oriented framework for the optimization of flexible renewable energy systems

Master's thesis in Chemical Engineering and Biotechnology
Supervisor: Johannes Jäschke and Truls Gundersen
Co-supervisor: Avinash S. R. Subramanian
June 2021

Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

NTNU
Norwegian University of
Science and Technology

TKP4900 - Chemical Engineering, Master thesis

An object-oriented framework for the optimization of flexible renewable energy systems

Mari Elise Rugland

Submission date: 11.06.2021
Supervisors: Avinash Subramanian, EPT
Truls Gundersen, EPT
Johannes Jäschke, IKP

NTNU
Department of Energy and Process Engineering

NTNU
Department of Chemical Engineering

Norwegian University of Science and Technology

Preface

Acknowledgements

This thesis was written in the spring of 2021 as part of my M.Sc in Chemical Engineering at the Norwegian University of Science and Technology.

I greatly acknowledge supervisors, Truls Gundersen and Johannes Jäschke, for their guidance over the course of the specialization project fall 2020 and the master thesis spring 2021. Additionally, I would like to express special gratitude to co-supervisor Avinash Subramanian. His rigorous feedback, inventive ideas and useful suggestions for improvement has been of great help. Overall, the supervisors' helpfulness and commitment to the project has been encouraging and a great motivator.

Lastly, I want to thank my friends and fellow students at Chemical Engineering and Biotechnology, my room-mates over the years, and all the other amazing people I have had the pleasure of befriending. Without them, my years in Trondheim would not have been as fun and enjoyable as they have been.

Declaration of Compliance

I, Mari Elise Rugland, hereby declare that this is an independent work according to the exam regulations of the Norwegian University of Science and Technology (NTNU).
Signature:

Place and Date: Trondheim - Gløshaugen, 11^{th} of June 2021

Abstract

This thesis describes an object-oriented modeling framework for interfacing an energy system model to the stochastic programming software GOSSIP. Key uncertainties related to optimizing flexible renewable energy systems such as the intermittent output of renewable energy sources, volatile utility market spot prices, and varying end user demand are addressed. In previous work [13], two-stage stochastic formulations were considered to be a promising approach to incorporate uncertainties, and the recently developed GOSSIP software was used for modeling. However, one could not easily extend programs to include new technologies and different kinds of modeling fidelity and levels of complexity. Furthermore, it was revealed that using GOSSIP requires explicit formulation of equations, variables, and constraints in the user input model. The proposed solution presented in this thesis is an object-oriented framework that enables an extensible user input model. An object-oriented structure implies the generalization of system structure and allows for abstraction and encapsulation through classes. The framework includes the following classes: `EnergySystem`, `UncertainParam`, `PrimarySource`, `Conversion`, `Utility`, `EndUser` and `ObjectiveFunction`.

The proposed object-oriented framework divides the model input into two parts: Interface and implementation. The interface is accessible to the user, whereas the implementation includes the class definitions, member functions for energy balance, and attributes. The implementation ensures that a correct two-stage stochastic mixed-integer linear program is formulated and interfaced with the solver. Then, the two algorithms Non-convex Generalized Benders Decomposition and Full Space can find the optimal design of a flexible energy system. Three systems are implemented to verify the framework correctness and show the utility of the resulting object-oriented program: (1) A simple system with wind turbines as energy conversion technology and one end user with electricity demand. (2) A more complex system with wind turbines and solar photo-voltaic panels as energy conversion technologies and one end user with electricity demand. (3) A more complex system with wind turbines, and solar photo-voltaic and solar thermal panels as energy conversion technologies, and two end users with heat and electricity demand.

Expanding the simplest system to the two more complex ones requires few lines of code, no explicit variable and constraint declarations, and no changes to the implementation. The extensibility and scalability that the framework displays encourage pooling together of models, underscoring the suitability of object-oriented programming in optimizing flexible renewable energy systems. To summarize, a framework as the one presented can simplify the process of formulating stochastic programming problems to design flexible renewable energy systems.

Sammendrag

Denne masteravhandlingen beskriver et objektorientert modelleringsrammeverk som et grensesnitt mellom en modell av ett energisystem og den stokastiske programmeringsprogramvaren GOSSIP. Viktige usikkerheter knyttet til optimalisering av fleksible fornybare energisystemer som ustabile spotmarkedspriser, den intermitterende energiproduksjonen til fornybare energikilder, og varierende etterspørsel fra sluttbrukere inkluderes i problemformuleringen. I tidligere arbeider [13] ble to-trinns stokastiske formuleringer ansett for å være en lovende tilnærming for å innlemme usikkerhet, og den nylig utviklede GOSSIP-programvaren ble brukt til modellering. Imidlertid, i tidligere arbeider kunne man ikke enkelt utvide programmene til å omfatte ny teknologi, forskjellige modeller og nivåer av kompleksitet. Videre ble det avslørt at bruk av GOSSIP krever eksplisitt formulering av ligninger, variabler og begrensninger i brukermodellen. Den foreslåtte løsningen, presentert i denne oppgaven, er et objektorientert rammeverk som muliggjør en utvidbar brukermodell. En objektorientert struktur innebærer generalisering av systemstrukturen og gir mulighet for abstraksjon og innkapsling gjennom klasser. Rammeverket inkluderer følgende klasser: `EnergySystem`, `UncertainParam`, `PrimarySource`, `Conversion`, `Utility`, `EndUser` og `ObjectiveFunction`.

Det foreslåtte objektorienterte rammeverket deler modelleringen i to deler: Grensesnitt og implementering. Grensesnittet er tilgjengelig for brukeren, mens implementeringen inkluderer klassedefinisjoner, medlemsfunksjoner for energibalanse og attributter. Implementeringen sørger for den korrekte formuleringen av ett to-trinns stokastisk mixed-integer lineært program og formidler det til GOSSIP. Deretter kan de to algoritmene Non-Generalized Benders Decomposition og Full Space finne den mest gunstige utformingen av et fleksibelt energisystem. Tre systemer er implementert for å verifisere rammeverkets korrekthet og vise nytten av det resulterende objektorienterte programmet: (1) Et enkelt system med vindturbiner til energiomdanning og en sluttbruker med strømbehov. (2) Et mer sammensatt system med vindturbiner og fotovoltaiske solcellepaneler til energiomdanning og en sluttbruker med strømbehov. (3) Et enda mer sammensatt system med vindturbiner, fotovoltaiske og solvarme paneler til energiomdanning, og to sluttbrukere med varme- og strømbehov.

Å utvide det enkleste systemet til de to mer komplekse krever få kodelinjer, ingen eksplisitte variabel- og begrensningserklæringer, og ingen endringer i implementeringen. Muligheten til å utvide og skalere brukermodellen som rammeverket viser, oppmuntrer til å slå sammen energimodeller, og understreker egnetheten til objektorientert programmering i optimering av fleksible fornybare energisystemer. For å oppsummere, det utviklede rammeverket kan forenkle prosessen med å formulere stokastiske programmeringsproblemer for utforming av fleksible fornybare energisystemer.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Nomenclature

Acronyms

BD	Benders Decomposition
ES	Energy System
EVP	Expected Value Problem
EEVP	Expectation of Expected Value Problem
GBD	Generalized Benders Decomposition
GOSSIP	Global Optimization of non-convex two-Stage Stochastic mixed-Integer Programs
IP	Integer Programming
LP	Linear Programming
MIP	Mixed Integer Programming
MICP	Mixed Integer Convex Programming
MILP	Mixed Integer Linear Programming
MINLP	Mixed Integer Non-Linear Programming
NGBD	Nonconvex Generalized Bender Decomposition
OOP	Object-Oriented Programming
P	Programs or Programming (used interchangeably)
RES	Renewable Energy Systems
RP	Recourse Problem
SP	Stochastic Problem
VSS	Value of the Stochastic Solution

Framework abbreviations

CVO	Conversion object
EUO	End User object
ESO	Energy System object
OBJO	Objective function Object
PSO	Primary Source object
UTO	Utility object
UPO	Uncertain Parameter object

CHAPTER
ONE

INTRODUCTION

> We need to put a price on carbon in the markets, and a price on denial in politics
>
> *- Al Gore*

1.1 Motivation

One of the biggest threats to future generations is the ongoing climate crisis and the current over-consumption of natural resources. Accordingly, the global energy demand has increased steadily since industrialization. Historically, this energy surge has been accounted for by increased use of fossil fuels [11]. In solution strategies that limit global warming to the United Nations 1.5 $°C$ target, renewables make up 70-85% of the global energy mix [5]. To motivate governing forces to encourage private companies and capital markets into sustainable growth and energy consumption, the UN has formulated 12 sustainability goals where goal number 7 reads:

"Ensure access to affordable, reliable, sustainable and modern energy for all" [9]

In addition to reducing the negative impact of human activities on the climate, it is advocated that ensuring clean energy to households that are below the poverty line can have positive social-economic effects through increased life-expectancy, educational level, and consequently household income [9].

Unfortunately, except for hydro-power, most widespread renewable energy sources such as wind and solar have an intermittent nature that makes it difficult to provide clean energy with high reliability. In addition, there are varying end user demands and volatile market spot prices of utilities. Consequently, the availability and profitability of renewable energy production and projects are challenging to forecast. This can make governments

and investors hesitant to increase the fraction of renewable energy in the global energy mix.

One approach to address the aforementioned uncertainties related to renewable energy systems is to implement a *flexible design*. Flexible renewable energy systems can adapt operating conditions to external signals such as changes in prices, weather, and demand. Measures for flexible design include combining multiple energy sources, extra production capacity, and energy storage technologies.

Optimization of flexible renewable energy systems requires approaches that take uncertain operating conditions into account. By accounting for uncertain and varying operating conditions in the problem formulation, the resulting flexible design enables the energy system to react to changes in operational circumstances, increasing overall availability. In previous work, a two-stage stochastic programming approach for the optimization of flexible renewable energy systems showed promising results with an increase in energy system profitability and reliability [13]. The program was created in GOSSIP, a stochastic optimization software developed at the Process Systems Engineering Laboratory at MIT by Kannan and Barton [6].

However, the program in [13] was not easily extended to new technologies and different kinds of model fidelity and levels of complexity. Making a small change to the energy system structure, such as adding a wind turbine, required a significant programming effort. Moreover, with the current fast-paced development of renewable energy technologies, both efficiencies and model fidelity are expected to change rapidly, and this needs to be accounted for. In addition, multiple different conversion technologies can be considered at the design stage. As an example, suppose that wind turbines are regarded as the most promising renewable energy technology today, but it is expected that the cost of solar fuel panels will decrease significantly over the next decade. Naturally, one wants to look into the potential of solar energy and possibly test different system configurations. It is currently a non-trivial task to construct stochastic optimization programs extensible enough to new technologies and models.

A solution is to create an object-oriented framework for efficient and scalable modeling of energy systems. Object-oriented programming facilitates the implementation of a user-friendly interface and automates the formulation of the stochastic programming problem. It also enables an implementation to ensure a generalized problem formulation that satisfies energy balances. The result is software that enables an extensible and scalable problem formulation to optimization flexible renewable energy systems.

The resulting framework implementation ensures the correct formulation of a two-stage stochastic mixed-integer linear program to model the flexible design problem. The framework leverages that GOSSIP was developed using the C++ language, which allows for object-oriented programming. The resulting optimization problem can be solved using the two GOSSIP algorithms NGBD and Full-Space. Hence, the development of an object-oriented framework compatible with GOSSIP is a promising step toward creating a user-friendly and extendable program for the optimization of flexible renewable energy systems.

1.2 Scope and objective

The main goal of this thesis is to develop a framework for modeling a variety of renewable energy systems using object-oriented programming and interfacing it to GOSSIP.

The following contributions and tasks were initially considered:

- Investigate different suitable data structures such as linked lists, trees, graphs.
- Attempt to formulate linear models from the literature and previous work [13].
- Investigate ways to extend to a richer class of models, such as disciplined polynomial programming or neural networks.
- Include algorithmic considerations.

However, in agreement with the supervisors, the following contributions and tasks related to the aforementioned goal were prioritized:

- Ensure that the problem formulation in the object-oriented framework is compatible with the two algorithms NGBD and Full-Space embedded in GOSSIP.
- Ensure that the aforementioned framework can handle various levels of modeling complexity:
    1. Complexity of the uncertainty model (e.g., including more scenarios and additional uncertain parameters).
    2. Temporal complexity through the number of time-steps per scenario.
    3. Increasing complexity of the RES by including additional components (e.g, energy sources, conversion technologies, utilities, and end users).
- Show extensibility in the implementation of classes and functions.
- Illustrate extensibility of the user input model, and user-friendliness of the aforementioned framework, through relevant examples of increasing complexity.

1.3 Structure of thesis

First, Chapter 2 presents a brief introduction to optimization and relevant methodologies from stochastic optimization. Thereafter, the applied optimization software GOSSIP is introduced in Chapter 3. The chapter gives a simplified overview of the two relevant optimization algorithms NGBD and Full-Space. In Chapter 4 an overview of object-oriented programming principles and their application in the optimization of energy systems is given. Then, the resulting object-oriented optimization framework is presented with a brief overview of all classes in Section 5.1. Section 5.2 sequentially includes and explains class header files. Section 5.3 gives a detailed description of the framework implementation, including system models and constraints. Lastly, Section 5.4 attempts to explain how to create a viable user input model in the framework. To further explain and illustrate the framework attributes and structure, three case studies are presented in Chapter 6. The examples are presented in order of increasing complexity starting from Section 6.1. In Chapter 7 both quantitative and qualitative results from the case studies in Chapter 6 are presented. Section 7.1 presents numerical results from the examples, whereas Section 7.2 presents algorithmic results. This is followed by a discussion of the resulting OOP framework and numerical results in Chapter 8. Lastly, final remarks and suggestions for future work are asserted in Chapter 9.

CHAPTER

TWO

OPTIMIZATION METHODOLOGIES

2.1 Introduction to optimization

Optimization is a mathematical procedure where the intention is to locate an optimal solution expressed through the minimization or maximization of an objective function. The output from the objective function is a scalar value, but the actual solution to the optimization problem is the value of the independent, feasible, decision variable(s). If the optimization problem does not contain any constraints it is an unconstrained problem. However, most often the problem is constrained by one or multiple equality and/or inequality constraints.

Equation 2.1 shows the formulation of a constrained optimization problem with an objective function, J, and constraints $h(\boldsymbol{x})$ and $g(\boldsymbol{x})$.

$$\begin{aligned} \min_{\boldsymbol{x}\in\boldsymbol{X}} \quad & J(\boldsymbol{x}) \\ \text{s.t.} \quad & \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0} \\ & \boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{0} \end{aligned} \tag{2.1}$$

The vector $\boldsymbol{x}$ contains the independent decision variables. For $\boldsymbol{x}$ to be a feasible point, all equality ($\boldsymbol{h}(\boldsymbol{x})$) and inequality ($\boldsymbol{g}(\boldsymbol{x})$) constraints are satisfied.

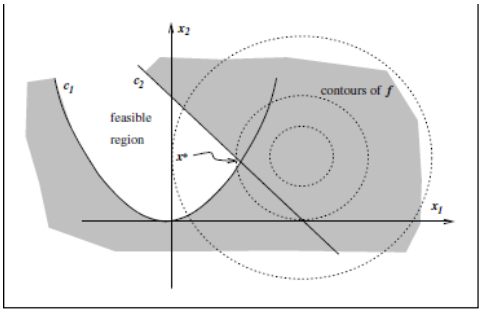

Figure 2.1: Geometrical representation of a constrained optimization problem [1].

Convexity in optimization

A convex set is a set C where for every two points (x, y) in C, a line segment z, as defined in Equation 2.2 must be in the set C. In other words, for C to be a convex set, every interior point on the line segment z must also be in C [2]. A convex set C is illustrated in Figure 2.2 a).

$$z = \lambda x + (1 - \lambda)y, \ \forall \lambda \ \in \ [0, 1] \tag{2.2}$$

If such a line segment z cannot be drawn without crossing any boundaries, the set is by nature non-convex, as illustrated in Figure 2.2 b).

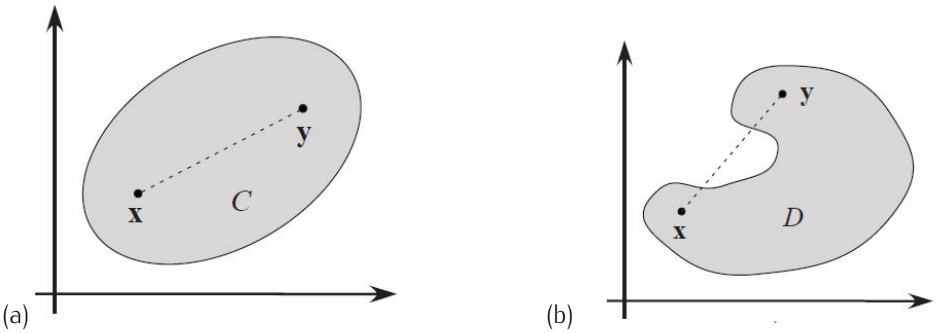

Figure 2.2: Convex (a) and nonconvex (b) set [2].

Following this, a convex function f, is a function defined on a convex domain C where for each two points $x_1, f(x_1)$ and $x_2, f(x_2)$ the line segment between the points lie entirely above the graph of the function f as illustrated in Figure 2.3 a) [2]. In mathematical terms, a function $f : X \to \mathbb{R}$ is a convex function on a convex set X if:

$$f(\lambda x_1 + (1 - \lambda)x_2) \leq \lambda f(x_1) + (1 - \lambda)f(x_2), \quad \forall \lambda \in [0, 1], \quad \forall x_1, x_2 \in X \tag{2.3}$$

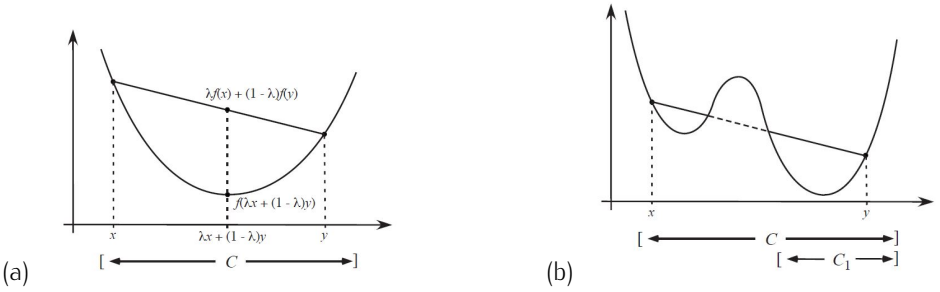

Figure 2.3: Convex (a) and nonconvex (b) function [2].

In a convex optimization problem, a local optimum is guaranteed to be a global optimum. However, if either the objective function or the feasible set is non-convex, then a local optimum is not guaranteed to be a global optimum.

2.2 Mixed-Integer Programming - MIP

Optimization problems are also distinguished by which type of decision variables they contain, where problems with both continuous and discrete variables are defined as mixed-integer programs (MIP).

The scope of this thesis involves mixed-integer linear programming problem (MILP) formulations, illustrated in Equation 2.4.

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{y}} \quad & \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\boldsymbol{y} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{b} \\ & \boldsymbol{x} \quad \in \{0,1\}^p \\ & \boldsymbol{y} \quad \in \mathbb{R}^n_+ \end{aligned} \tag{2.4}$$

$\boldsymbol{A}$ is a $m \times n$ matrix, $\boldsymbol{B}$ is a $m \times p$ matrix, and $\boldsymbol{b}$, $\boldsymbol{c}$ and $\boldsymbol{d}$ are m-,n- and p-dimensional vectors [18]. $\boldsymbol{x}$ is a p-dimensional vector of binary variables and $\boldsymbol{y}$ is a n-dimensional vector with continuous variables.

Mixed-Integer Programming (MIP) is one of the most common approaches to optimize design and planning of production systems in industry. In recent years it has also attained a strong position within the field of flexible renewable energy systems (RES). Even though most energy systems can be somewhat approximated using a linear model, providing a realistic representation of some systems requires the use of nonlinear (and thus nonconvex) model equations. The problem is then considered a nonconvex mixed-integer non-linear programming problem (MINLP).

2.3 Stochastic programming

An optimization problem can contain certain or uncertain input parameters. It is called a deterministic problem if it is assumed that none of the parameters is subject to randomness. On the other hand, if a model or parameter is subject to uncertainty, the problem can be formulated as a stochastic optimization problem and stochastic programming approaches can then be considered.

Contrary to deterministic approaches where all parameters in a model are assumed to be certain, stochastic programs incorporate random variables into the problem formulation to capture the uncertain nature of relevant parameters. The main purpose of a stochastic program is to reduce the risk of undertaking sub-optimal decisions. Probability distributions

or patterns from historical data can be used to describe parametric uncertainty. Although (the solution to) the resulting optimization problem is assumed to be more robust, including uncertainty increases the problem size. Several methods for handling optimization under uncertainty have been developed, and one acclaimed approach is two-stage stochastic programming with scenario generation [7] [10].

2.3.1 Two-stage stochastic programming

Equation 2.5 shows the mathematical formulation of a two-stage stochastic program.

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{y}} \quad & f(\boldsymbol{x}) + E_{\boldsymbol{\xi}}[Q(\boldsymbol{x},\boldsymbol{y},\boldsymbol{\xi})], \\ \text{s.t.} \quad & \mathbf{g}^{(1)}(\boldsymbol{x}) \leq \mathbf{0} \\ & \mathbf{h}^{(1)}(\boldsymbol{x}) = \mathbf{0} \\ & \boldsymbol{x} \in \boldsymbol{X} \end{aligned} \tag{2.5}$$

Here $f(\boldsymbol{x})$ is some function of $\boldsymbol{x}$, and the vector $\boldsymbol{\xi}$ contains the realisations of the uncertain parameters. The functions $\mathbf{g}^{(1)}$ and $\mathbf{h}^{(1)}$ constrain the first stage design variables, and $\boldsymbol{X}$ defines the feasible set for the variables. The expected value of the function $Q(\boldsymbol{x},\boldsymbol{y},\boldsymbol{\xi})$, $E_{\boldsymbol{\xi}}$, is shown in Equation 2.6. It is the sum of the function for all scenarios, h, multiplied by a corresponding probability, p_h, and with parameter values ξ_h. $Q(\boldsymbol{x},\boldsymbol{y},\boldsymbol{\xi})$ represents the recourse problem and is given by Equation 2.7.

$$E_{\boldsymbol{\xi}}[Q(\boldsymbol{x},\boldsymbol{y},\boldsymbol{\xi})] = \sum_{h=1}^{S} p_h \cdot Q(\boldsymbol{x},\boldsymbol{y}_h,\boldsymbol{\xi}_h) \tag{2.6}$$

$$\begin{aligned} \min_{\boldsymbol{y}_h} \quad & Q(\boldsymbol{y}_h,\boldsymbol{x},\boldsymbol{\xi}_h) \\ \text{s.t.} \quad & \mathbf{g}_h^{(2)}(\boldsymbol{y}_h,\boldsymbol{x},\boldsymbol{\xi}_h) \leq \mathbf{0} \qquad \forall\ \mathrm{h} \in \mathrm{S} \\ & \mathbf{h}_h^{(2)}(\boldsymbol{y}_h,\boldsymbol{x},\boldsymbol{\xi}_h) = \mathbf{0} \qquad \forall\ \mathrm{h} \in \mathrm{S} \\ & \boldsymbol{y}_h \in \boldsymbol{Y} \end{aligned} \tag{2.7}$$

Functions $\mathbf{g}^{(2)}$ and $\mathbf{h}^{(2)}$ constrain the $2nd$ stage variables, and $\boldsymbol{Y}$ defines the feasible set for the $2nd$ stage variables.

In two-stage stochastic programs, the number of scenarios is a function of the number of uncertain parameters and the number of possible realizations of these parameters. In the first stage of the program, prior to the realization of the uncertain parameters, a set of immediate decisions are made. In the second stage, corrective actions are made to compensate for the realization of uncertainties. Both first- and second-stage variables are determined with the purpose of minimizing (or maximizing) the value of the objective function.

Contrary to deterministic optimization problems, stochastic programs have an objective function with two separate terms, namely a first- and second-stage term. The aim is to simultaneously minimize (or maximize) the first stage term and the *expected value* of the

second stage term. Determining the optimal decision variables in the second stage is called the recourse problem. Throughout this thesis, the first stage decision variables of a two-stage stochastic program are represented by a vector $\boldsymbol{x}$, while the second stage variables are represented by a vector $\boldsymbol{y}$.

In a two-stage stochastic program, uncertain parameters take values from a finite number of realizations, each with an associated probability. Figure 2.4 illustrates the scenario tree for a two-stage stochastic program.

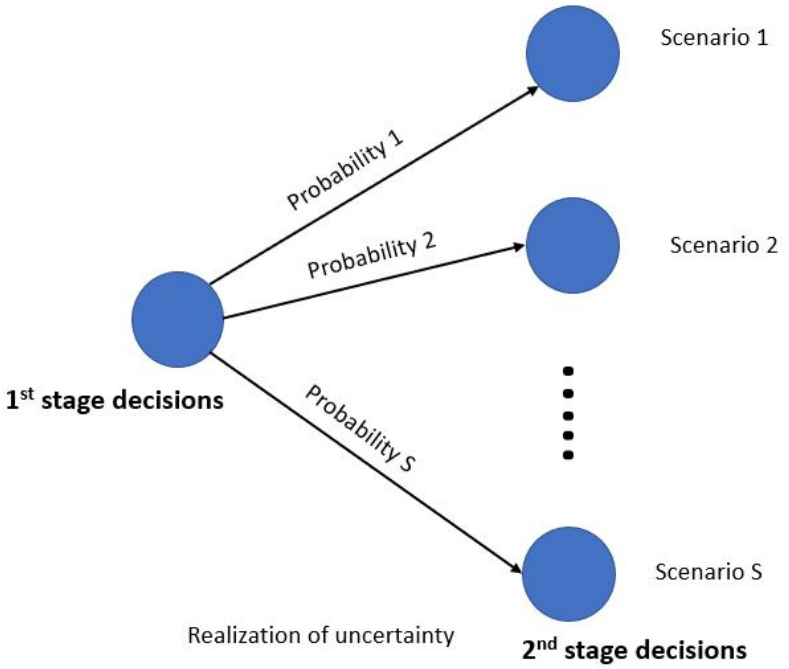

Figure 2.4: Scenario tree for a two-stage stochastic program.

2.3.2 Multi-stage stochastic programming

A multi-stage program is set up in an analog way to Equation 2.5, only that there are several recourse stages. In a multi-stage program, the number of scenarios is, in addition to the number of uncertain parameters and realizations, dependent on the number of time-steps. Multi-stage programs can be used to optimize flexible renewable energy systems that demand sequential, dependent days. A multi-stage approach can be necessary if energy storage is included as the state of charge of the storage technology requires sequential days to be linked. Unfortunately, the number of scenarios increases exponentially with the number of time-steps. As a result, even a low time resolution and horizon such as 10 scenarios per day over the course of one week gives an explosion in scenarios (10^7 scenarios). The scenario tree in a multi-stage approach is illustrated in Figure 2.5.

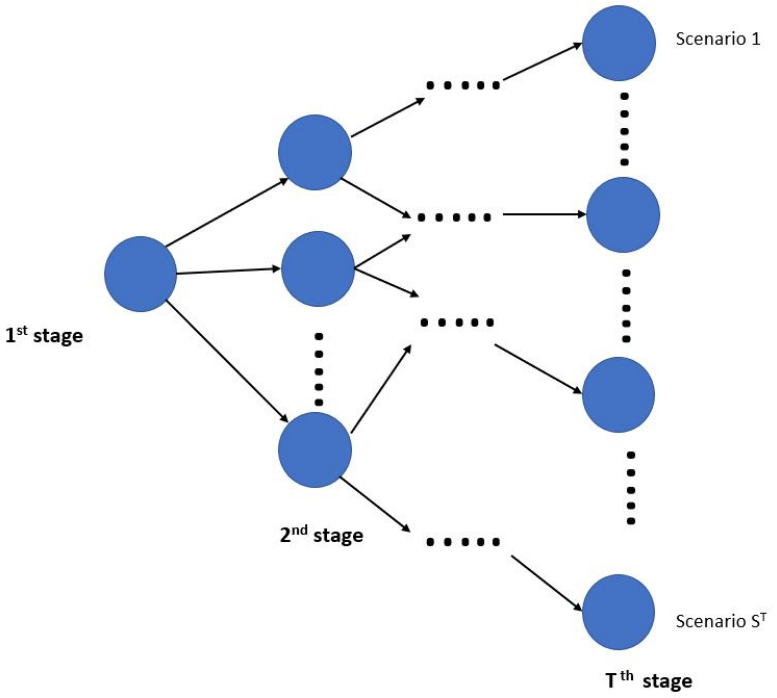

Figure 2.5: Scenario tree for a multi-stage stochastic program.

2.3.3 Multi-period stochastic programming

A potential workaround to the multi-stage formulation is a two-stage multi-period formulation as illustrated in Figure 2.6. Uncertain parameters are aggregated over the entire time period considered. As a result, all uncertain parameters for a scenario branch is assumed to be known in advance. The multi-period formulation can be applied by creating scenario profiles with the desired time resolution and range for each uncertain parameter. For instance, three different hourly solar radiation profiles such as sunny, cloudy and rainy day, each with its corresponding probability and hourly varying value. The downside of the approach is that the solver has perfect information. With multi-period modeling, the solver will, in advance, know the realization of the uncertain parameters in all time-steps. This can reduce the validity of the resulting solution.

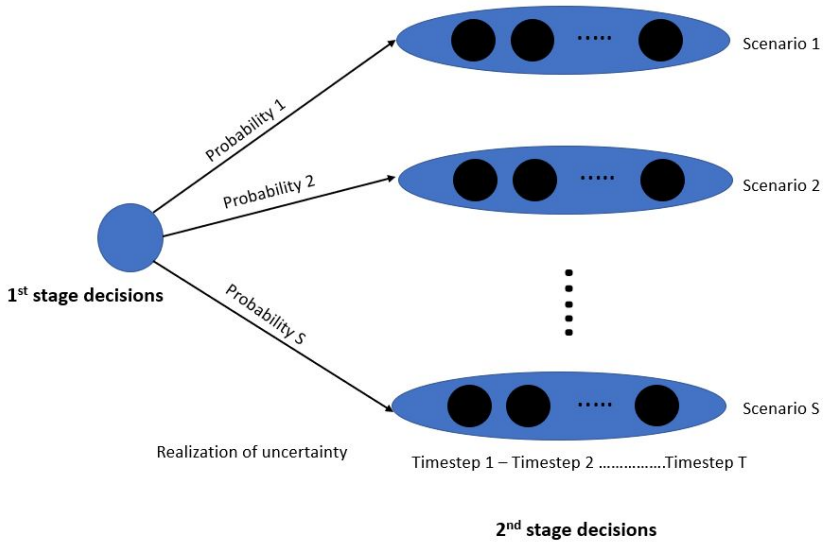

Figure 2.6: Scenario tree for a two-stage multi-period stochastic program.

2.3.4 Value of Stochastic Solution

The value of Stochastic Solution (VSS) can be used to evaluate the performance of a stochastic program. To calculate the VSS an Expected Value Problem (EVP) is formulated.

- An optimization program where all uncertain values are assumed to take on their expected value (mean) is called an Expected Value Problem (EVP). The EVP can be viewed as a single-scenario approach where uncertain parameters are set constant at their mean value. First stage variables are selected to yield the optimal value of the objective function for the single scenario, and the selected first stage variables is referred to as the nominal design.
- The value of the objective function under the nominal design subject to uncertainty is called the Expectation of Expected Value Problem (EEVP).
- The additional value obtained by including uncertainty in the problem formulation is called the Value of Stochastic Solution (VSS) and is defined as the difference between the value of the objective function in the stochastic program (SP) and the EEVP. The VSS is defined in Equation 2.8.

$$VSS = SP - EEVP \tag{2.8}$$

CHAPTER

THREE

GOSSIP SOFTWARE

GOSSIP is a software framework for modeling and solving two-stage stochastic nonconvex MINLPs. It is embedded in a C++ platform and an overview of functionalities and worked out examples for solving two-stage stochastic programs in GOSSIP can be found in the documentation [8]. The GOSSIP software is, under certain requirements, guaranteed to determine the global optimum of a non-convex two-stage stochastic problem. However, GOSSIP can also be used to solve convex two-stage stochastic programs (MICP) as well as large-scale MILP. Multiple solution methods are implemented in GOSSIP, but only the NGBD and Full-space algorithms are applied in this thesis.

Various decomposition approaches have been developed to handle different classes of stochastic programming problems as illustrated in Figure 3.1. The earliest approach was termed 'Benders decomposition (BD)' and was only applicable to the class of two-stage stochastic MILPs. BD was then extended to give Generalized Benders Decomposition (GBD) which could solve the class of two-stage stochastic Mixed-Integer Convex Programs (MICPs). Finally, GBD was extended for the class of two-stage nonconvex MINLPs with the Nonconvex Generalized Benders Decomposition (NGBD) algorithm. We note that NGBD reduces to the GBD algorithm for convex problems and to the BD algorithm for linear problems (MILP).

Next, a brief overview of the GBD, NGBD and Full-Space algorithm is presented. Complete details are presented in [2] and [4].

3.1 Generalized Benders decomposition and extensions

GBD is a method for solving two-stage stochastic MICPs. The GBD strategy involves constructing an equivalent dual representation of the original problem with a large but finite number of constraints. A relaxation of the dual representation is then constructed by only including a small subset of the constraints. The solution of this relaxed problem yields the lower bound on the solution to the original problem. Due to the strong duality for convex problems, the solution to the dual problem itself yields the upper bound to the original

problem. These two steps are done in an iterative manner until a global solution is found [2].

The NGBD extension strategy involves convexifying the MINLP and then applying GBD to give a lower bound to the problem. The upper bound is found by solving the MINLP using a local solver. This procedure is done in an iterative manner shrinking the gap between the lower and upper bound until convergence to a global optimum.

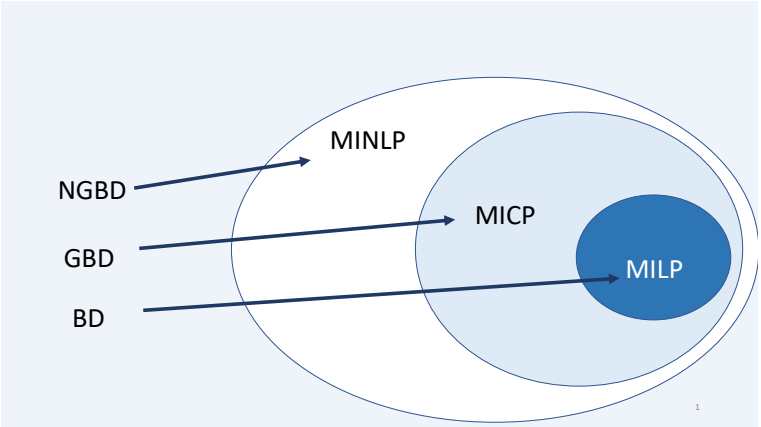

Figure 3.1: A schematic of the different decomposition algorithms and the class of stochastic programs they are applicable to.

Two-stage stochastic problems are decomposed into smaller sub-problems in the NGBD algorithm, e.g. one for each scenario, which provides efficient scaling of solution time with an increasing number of scenarios considered. The aforementioned methods make NGBD a strong tool for the global optimization of MINLP and other non-linear programming problems. However, the NGBD algorithm is only guaranteed to converge for discrete 1^{st} stage variables. Consequently, the $1st$-stage variables in $\boldsymbol{x}$ from Equation 2.5 and 2.7 can only contain variables from a discrete set of integer values. A potential workaround (used in this thesis) is to discretize each continuous 1^{st} stage variable by assuming it can only take on a fixed number of values, x_j, within its interval bounds as defined in Equation 3.1.

$$x_j^{discrete} = j \cdot \frac{x^{UBD}}{d-1} \qquad \forall \text{ j} \in \{0,\ldots,\text{d-1}\} \tag{3.1}$$

$x_j^{discrete}$ denotes the discretized value in the j^{th} interval, d number of intervals, and x^{UBD} the upper bound of the interval, respectively. The lower bound is set to zero and omitted here. A set of binary variables, $\boldsymbol{x}^{binary}$, are implemented to ensure that x only take on one of the fixed variables discretized above. This implies that the first stage variables in a two-stage program needs to be included through a sum of the product of the binary variables and the corresponding discrete value, as shown in equation 3.2.

$$x = \sum_{j=0}^{n} x_j^{binary} \cdot x_j^{discrete} \tag{3.2}$$

3.2 Full-Space algorithm

The scenario-wise decomposition mentioned in Section 3.1 is not executed in the Full-Space algorithm. As the name Full-Space indicates, the algorithm finds the solution to the full-space problem (Problem SP in [8]) through a linked version of ANTIGONE (Algorithms for coNTinuous/Integer Global Optimiztion of Nonlinear EquatioNs). It is a global optimizer for nonconvex MINLP's. The optimization software transforms a user-defined MINLP by reformulation of the model, detects mathematical structure, solves the optimization problem with branch-and-cut global optimization, and returns the model with respect to the original user-defined variables [4]. For more details about ANTIGONE, the reader is referred to [4].

The Full-space algorithm is assumed to be efficient for small problems, but solution time is expected to increase asymptotically faster in CPU time with in increased number of scenarios compared to the NGBD solver [6]. Contrary to NGBD, the Full-Space solver is guaranteed to find the global solution with both continuous and discrete 1st stage variables in a stochastic MILP or MINLP.

CHAPTER

FOUR

OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a powerful programming technique that involves hiding specific details (encapsulation), re-using code (inheritance), and applying interfaces that can represent multiple different types (polymorphism). A short description of basic definitions from computer programming and specifically object-oriented programming is provided below to simplify subsequent explanations.

Basic programming terms [3]

- Procedure(s) - Also called function, method, routine, or subroutine. In general, an input is manipulated to give the desired output. An example is how wind speeds are converted to rated power outputs in a wind turbine model.
- Procedural programming - A computer program with a series of computational steps to be carried out in a predetermined order. A procedure is only operated on the data structure upon which it is called.
- Data structures - Several variables stored together in a type of structure. Some common types are strings (words and/or sentences), lists, vectors, and hash tables.
- Class - A class is a definition of a specific data format (object type). Classes contain attributes and functions that are shared among, and available to, all instances (objects) of the class. E.g objects `SolarRadiation`, `Wind` and `Water` could all belong to the class `RenewableEnergySource` as they all share attributes such as being a renewable energy source.
- Object - An instance of a class. A wind turbine could be an instance of a class named `EnergyConversion`.
- Header file - A header (hpp) file can be thought of as the overview or recipe of a class. The header file contains class declarations, specifically class attributes and class function declarations.

- Implementation file – An implementation/source (cpp) file contains the class function definitions, e.g the class function implementations. The implementation file shows how an instance of the class is constructed, and how input is manipulated to output in class functions.

4.1 Object-oriented programming

4.1.1 An introduction to object-oriented programming

In classic procedural programming, it is common to define a set of relevant parameters and write separate functions that take one, several, or all of the parameters and computes the desired output. However, as the problem complexity, and sub-sequentially computer programs, grow, the procedural programming method can become tedious. Firstly, if an existing function is used as a base to create a new function with slightly different functionality, the programmer has to write an entirely new function even though the two are almost identical. Secondly, making changes in the later stages of the program development often causes the need to change the entire program so as to avoid breaking already defined functions. Lastly, exceptions and multiple possible cases are difficult to handle in an appropriate fashion, increasing the number of lines of code. Object-oriented programming can in part be viewed as a result of trying to overcome these issues.

OOP is a programming paradigm based on the concept of objects that can contain data, data structures, and functions. Specifically, data in the form of fields with attributes or properties. The key idea is that attributes and functionalities that are shared by several objects can be defined within a family, or class, of objects, thus avoiding copy-paste of almost identical code. For instance, all portable computers have certain attributes like a screen, mouse-pad, keyboard, hard-drive, etc. The class would then be called PCs and the aforementioned attributes would automatically be defined for all PC objects through a class constructor.

However, an Apple and DELL PC have different operating systems (OS) and this functionality would be different for an Apple PC and DELL PC object. Either two sub-classes, Apple and DELL computers, would be constructed, or specifications would be set after the construction of the generic PC object through member functions.

Languages that support OOP often use inheritance to reuse code within classes. Two sub-classes Apple and DELL computers from the class `Computer` is an example of this. Class variables and procedures are inherited to all instances (objects) of the base class. Member variables refer to both the class and instance variables that are defined by a particular class. Instance variables are data that belong to a specific object and each object has its own copy, e.g it can be the different DELL and Apple operating systems (OS). As for functions, a class function belongs to the class as a whole whereas instance functions belong to individual objects of the class. The class functions have access to class variables and inputs from the procedure call, whereas the instance functions only have access to the instance variables of the specific object they are called on.

Another feature of objects in OOP is that the object's procedures can modify and change the object's attributes, giving the object a notion of the self and other objects which it can interact with.

4.1.2 The 4 object-oriented programming pillars

The 4 renowned OOP pillars are listed below [19].

- Encapsulation - To group related variables and procedures into units (objects of the same class).
- Abstraction - Ensure that the programmer can ignore certain details during programming and the current implementation. Hide details to the user such that changes to the program can be made without modification of the application (e.g changes related to the implementation of class constructors and functions).
- Polymorphism - Make an object (instance of a class) behave like another instance of the same class as long as the object satisfies the base class specifications.
- Inheritance - Ease the work of encapsulation and polymorphism by allowing the programmer to create objects (derived class) that are more specialized versions of other (base class) objects.

4.2 Object-oriented programming in the optimization of flexible renewable energy systems

A fundamental quality of a flexible (renewable) energy system (RES) is that multiple energy sources, as well as conversion and storage technologies, can be integrated into one system. The primary motivation of flexible design is to mitigate operational challenges associated with related uncertain parameters. Another key property of a flexible RES with multiple conversion technologies is that several end users with multiple different utility demands can be considered. As an example, chemical plants need extensive amounts of both heat and electricity to run certain processes. Furthermore, to comply with environmental due diligence, energy from renewable sources should be included in the plant energy mix. A solution consisting of energy from a natural gas combined cycle (NGCC) and solar PV panels could satisfy the aforementioned utility requirements (heat and electricity) while reducing the negative environmental impact of the plant. However, with solar panels as the only renewable source, the plant is at risk of deficit production of renewable electricity on cloudy days. The inclusion of a wind turbine and/or energy storage could reduce the risk of deficiency. On the other hand, in the event of redundant capacity, the energy producer would want to sell surplus electricity and heat to another end user. The resulting flexible RES could include multiple energy sources, converters, and end users, culminating in a large and complex optimization problem.

OOP is a programming technique that can simplify the construction and problem formulation of large energy systems as it enables the user to break the modeled system into smaller components (objects). Furthermore, OOP promotes user-friendly interfaces through encapsulation and abstraction of classes and functions.

Through a framework interface the user could construct and link components of the RES, call respective member functions and specify certain and uncertain parameters. The user would specify the energy balance models of the different components by implementing an efficiency parameter. Following this, the implementation formulates a two-stage stochastic programming problem based on the user input. To summarize, the implementation of an OOP framework for the formulation of two-stage stochastic programming problems could simplify the process of optimizing flexible renewable energy systems.

CHAPTER

FIVE

OBJECT-ORIENTED FRAMEWORK

This chapter contains the main developments of this work. Section 5.1 describes relevant properties of an object-oriented framework for two-stage stochastic optimization of flexible renewable energy systems. The section includes a flowsheet with the desired framework classes and information flow. In Section 5.2, descriptions and explanations of the resulting framework classes are presented with the respective header files (hpp-files). The associated implementation files (cpp-files) are found in Appendix A. Section 5.3 deals with the model objective, equations and constraints. The section shows how the user input is applied in the implementation to formulate a two-stage stochastic MILP problem. Finally, Section 5.4 shows the minimum number of objects that have to be constructed, and explains which function calls are necessary, to ensure successful compilation and execution of the resulting program.

5.1 Process flowsheet and corresponding objects

Figure 5.1 shows the information flow from the user interface to the solvers in GOSSIP, whereas Figure 5.2 shows the structure and information flow of the OOP framework. In general, it is preferable to keep the information flow simple and avoid creating cycles and inter-dependencies. This way, it is easy to remove or add objects.

Information is shared between the energy system class and the other classes through the inclusion of a pointer to the energy system object in the constructor of the other classes. This ensures consistency of main temporal and system variables, namely the number of time-steps and scenarios, respectively. Pointers are used as the primary link between system objects and as a conveyor of information. Pointers enable easy information flow, avoiding explicit declaration of variables and constraints in the user input model. This way, the majority of the system functionality can be embodied in the implementation, ensuring encapsulation of details that ensure model correctness, such as energy balance equations.

Specifically, the implementation deals with the declaration and naming of system and object variables and constraints. For instance, properties as well as input and resulting energy output from a conversion technology. This way, the client only specifies the actual objects and the relationship between the objects in the flexible RES. Conversely, in writing a procedural program using the native GOSSIP interface, the user would have to specify whether the variable is $1st$ or $2nd$ stage, the variable bounds, name, and type (integer, binary, etc). In addition, the client would have to keep track of the number of variables and constraints. This is a non-trivial task as even a small system can have hundreds of variables and constraints. OOP enables the abstraction of the aforementioned details which normally would require a substantial amount of code in the user input model.

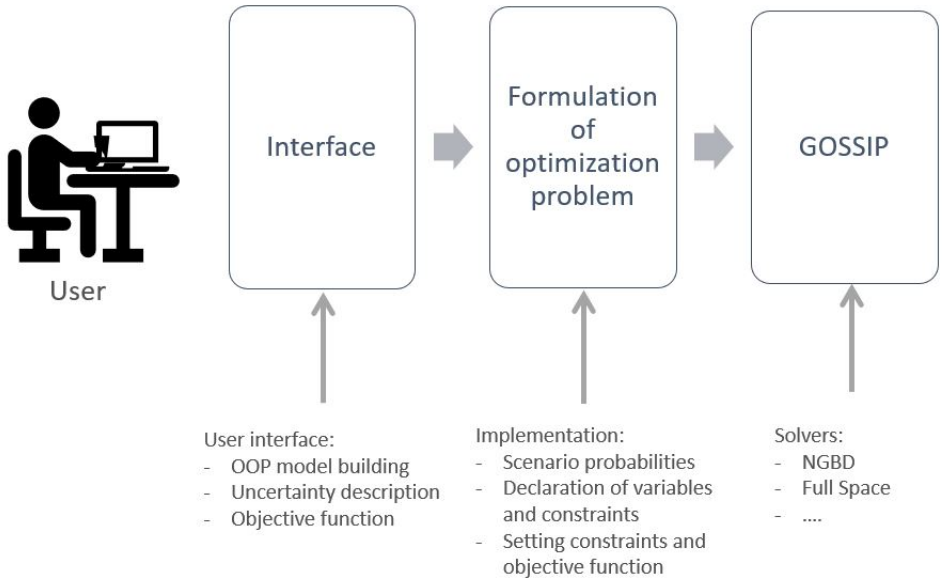

Figure 5.1: An overview of the different layers of code in the framework. User input is sent through the user interface to formulate an optimization problem that can be solved using GOSSIP. The internal information flow in and between *interface* and *formulation of optimization problem* is illustrated in the process flowsheet in Figure 5.2.

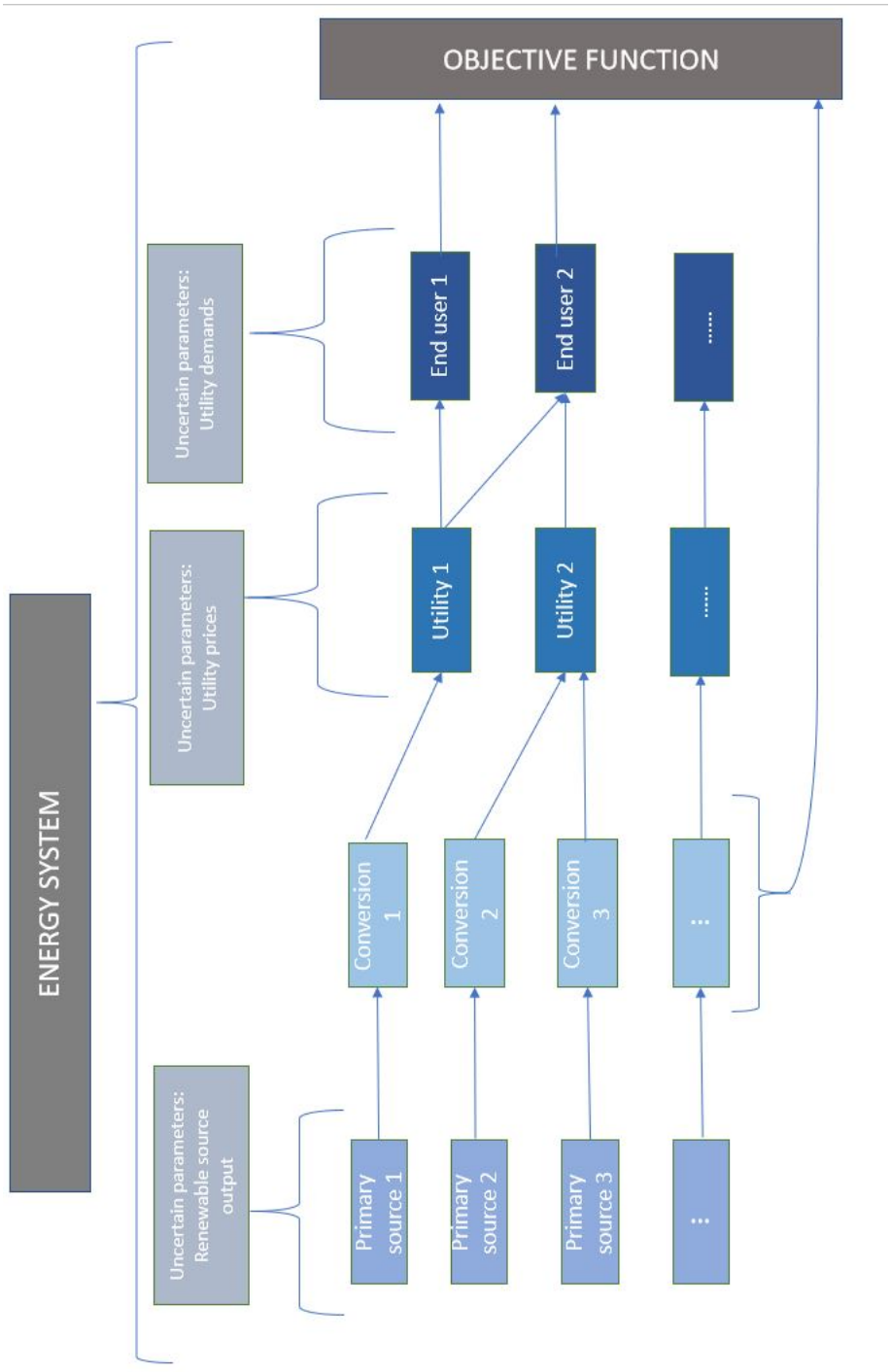

Figure 5.2: Flowsheet illustrating the relationship between the different objects. 22
The rectangles represent objects where the color of the rectangle marks the class that the object is an instance of. Arrows represent flow of values or variables, braces represent bi-directional pointers to and from the object.

5.2 Classes

The framework developed in this thesis consists of the following classes where the respective header (hpp) and source (cpp) files are placed in the GOSSIP library *lib* in a folder named *OOP*.

- `EnergySystem`
- `UncertainParam`
- `PrimarySource`
- `Conversion`
- `Utility`
- `EndUser`
- `ObjectiveFunction`

To apply the developed OOP framework the user has to add the following declarations to the `inputModel.cpp`[8] in GOSSIP:

1. `#include "headerfiles.hpp"`
2. `#include "MSCfiles.hpp"`

Below, in Sections 5.2.1 to 5.2.7, the classes are described.

5.2.1 The `EnergySystem` class

Each instance of the `EnergySystem` class corresponds to an entire renewable energy system. A single case study would correspond to one object of this class. The `EnergySystem` object is responsible for system consistency and keeping track of the number of system variables and constraints. The user sets the number of scenarios, `scen`, and time-steps, `time`, once, and through pointers to the energy system object, this information is shared with all other system objects. Pointers are the primary method of transferring information between objects. This is to avoid duplicating variables and constraints, and to conceal the actual data structure, thus promoting a user input model with a clear system structure. This benefit is implicit wherever pointers are implemented, and will not be stressed further in later class descriptions.

When a variable or constraint is set in a system object, a global variable or constraint count is incremented in the `EnergySystem` object. A consistent number of scenarios and time-steps as well as the variable and constraint count are basic requirements for running optimization programs in GOSSIP. However, the interface to construct the user defined energy system does not have to involve these details. Thus, related operations and procedures are left in the implementation, hidden from the client.

Because the `EnergySystem` object specifies the number of scenarios and other variables required by GOSSIP, the class is also responsible for importing the scenario probabilities specified by the user. The probabilities have to be stored in, and imported from, a csv-file.

```
//ENERGY SYSTEM CLASS
#ifndef ENERGYSYSTEM_H
#define ENERGYSYSTEM_H
#include "headerFiles.hpp"
//#include "EndUser.hpp"

namespace decomposition
{
  class EnergySystem
{
private:

  int numScen, numTimeSteps;

  int numUncertainParams, numSources, numConverters, numUtilities,
    numUsers; //keep track of objects included in the system


  int varcount; //variable count, GOSSIP essential
  int concount; //constraints count, GOSSIP essential
  vector<double> probabilities;//vector with numScen probabilities,
    should always sum up to 1

public:

  char clabel[70]; //label for GOSSIP variables, GOSSIP essential


  EnergySystem(int scen, int time); //constructor, initialises
    renewablesProduced, nonRenewablesProduced,importedFromGrid;
  void importProbabilities(string filePath, vector<double> & prob); //
    import probabilities from csv file and add to weight vector in
    main file

  int getNumScen(){return numScen;};
  int getNumTimeSteps(){return numTimeSteps;};

  //inline functions to increment variable, constraint and object
    counts
  int addVariable(){++varcount; return varcount;};
  int addConstraint(){++concount; return concount;};

};

}
#endif
```

Listing 5.1: The `EnergySystem` class header file.

In Listing 5.1 we have:

1. `EnergySystem(int scen, int time)`
    - The energy system class constructor on line 27 initialises the aforementioned system variables.
2. `addVariable()` and `addConstraint()`
    - Public member functions `addVariable()` and `addConstraint()` on lines 14 and 15 keeps track of the number of constraints and variables in the entire RES model through member variables varcount and concount.
3. `importProbabilities(string filepath, vector<double>& prob)`
    - Public member-function `importProbabilities(vector<double>& prob)` on line 26 imports probabilities for `numScen` scenarios from a csv-file located at `filePath`.
    - The probabilities are added to the vector `prob`, which is a parameter in the user input model in GOSSIP.
    - In addition, the probabilities are stored locally in the vector named `probabilities` .

5.2.2 The `UncertainParam` class

One uncertain parameter can be relevant to several system objects. To avoid importing the same parameter multiple times, the realisation of a parameter is stored in an `UncertainParam` object. The parameter value can then be used as input to multiple objects. For instance, the market spot price of electricity could be relevant in the case of deficit electricity and/or heat production as electricity can be used for heating as well. Consequently, `Utility` objects for both electricity and heat would depend on the same uncertain parameter.

Uncertain parameter objects import and store the realization of uncertain parameters. Each instance of the class corresponds to one uncertain parameter, which in turn requires one csv-file from the client with `numScen` scenarios and a time-resolution of `numTimeSteps`.

```
1  //UNCERTAIN PARAM CLASS
2  #ifndef UNCERTAINPARAM_H
3  #define UNCERTAINPARAM_H
4  #include "headerFiles.hpp"
5  #include "EnergySystem.hpp"
6
7  namespace decomposition
8  {
9
10 class UncertainParam
11 {
12
13 private:
14
15   int numScen,numTimeSteps;
16
```

```
17    string paramName; //name of imported parameter scenario values, set
        by user
18    string filePath; //filepath to text/csv/excel file with parameter
        scenario values, should be numScen x numTimeSteps values
19
20    vector<vector<double>> output; //vector with outout directly
        imported from csv file, dimensions numTimeSteps and numScen
21
22
23  public:
24    friend EnergySystem;
25
26    EnergySystem* linkToSystem; //linking parameter to main energy
        system
27
28
29    UncertainParam(string path, EnergySystem* enrg, bool print=false);
        //constructor
30
31
32    vector<vector<double>>* getOutput(){return &output;};
33  };
34  }
35  #endif
```

Listing 5.2: The `UncertainParameter` class header file.

In Listing 5.2 we have:

1. `UncertainParameter(string path, EnergySystem* enrg, bool print=false)`
    - The uncertain parameter constructor on line 10 initialises system variables and imports parameter realisations from the csv-file located at filepath `path` and stores them in the 2-dimensional vector named `output`.
2. `getOutput()`
    - Inline function `getOutput()` on line 33 returns a pointer to the imported parameter values in vector `output`.
    - The function is primarily used as a parameter in function calls of other objects. The pointer is later dereferenced to access the actual values.

5.2.3 The `PrimarySource` class

The intention behind the primary source class is that if one renewable energy source (uncertain parameter) has multiple different energy outputs, the client can construct multiple primary sources that link to the same energy source. For instance, solar radiation could be both an electricity and heat source, and the client could then construct primary sources `solarRadiation` and `solarThermalRadiation` from the uncertain parameter object `solar`. As a result, the user can easily specify another energy model and expand from one to multiple conversion technologies and subsequently utilities.

```
1  //PRIMARY SOURCE CLASS
2  #ifndef PRIMARYSOURCE_H
3  #define PRIMARYSOURCE_H
4  #include "headerFiles.hpp"
5  #include "EnergySystem.hpp"
6
7  namespace decomposition
8  {
9
10 class PrimarySource
11 {
12 private:
13   int numScen, numTimeSteps;
14
15   vector<vector<double>> output; //vector with outputs from primary
       energy source
16
17
18 public:
19   friend EnergySystem;
20
21   PrimarySource(vector<vector<double>>* input, EnergySystem* enrg );
       //constructor
22
23   vector<vector<double>>* getOutput(){return &output;}
24
25 };
26 }
27 #endif
```

Listing 5.3: The `PrimarySource` class header file.

In Listing 5.3 we have:

1. `PrimarySource(vector<vector<double>>* input, EnergySystem* enrg)`
    - The constructor on line 21 initialises system variables, and modifies input values from pointer `input` to output stored in the `output` vector.
2. `getOutput()`
    - Inline function `getOutput()` returns a pointer to the vector `output`. It has the same purpose as that of the uncertain parameter class.

5.2.4 The `Conversion` class

The conversion class is used to denote pieces of equipment that change the input from the primary energy source to a useful utility. For instance, photo-voltaic panels and wind turbines are examples of objects in the `Conversion` class. Each object in the `Conversion` class is linked on the input side with an object from the `PrimarySource` or `UncertainParam` class, and on the output side with a `Utility` object.

A considerable amount of the costs in an energy production project is related to the capital expenses associated with energy conversion technologies. Capital cost estimates for renewable energy conversion technologies are continuously being cut as a result of technological advancements. Consequently, the cost equation for the `Conversion` class was developed with the intention of being easy to specify and update.

In addition to cutting capital expenses, technological advancements can result in improved efficiencies and increased energy output. A natural consequence of this is the development of new energy conversion models which in turn motivated the development of functionality to modify the energy conversion model in a `Conversion` object.

Binary $1st$ stage variables are introduced to represent the choice of picking a particular equipment size of a conversion technology. To enforce the selection of exactly one size from the `sizes` vector of each conversion technology, the program has to set first-stage binary design variables and constraints `designVarBin` and `designConBin`, respectively. In addition, to link the binary $1st$ stage variables to the $2nd$ stage recourse problem, variables and constraints `designVarDisc` and `designConDisc` are declared. Both the $1st$ stage binary constraints and the $2nd$ stage linking constraints are of little relevance to the user input model but required to ensure the convergence of the NGBD algorithm in GOSSIP. In accordance with the second OOP-pillar in Section 4.1.2, abstraction, these variables and constraints are set in the implementation files.

```
1  //Conversion class
2
3  #ifndef CONVERSION_H
4  #define CONVERSION_H
5  #include "headerFiles.hpp"
6  #include "PrimarySource.hpp"
7  #include "EnergySystem.hpp"
8
9
10
11
12 namespace decomposition
13 {
14
15
16 class Conversion
17 {
18 private:
19
20
21   vector<vector<double>>* input; //pointer to input vales from primary
       source/energy source, e.g solar irradiation etc
22
23   EnergySystem* linkToSystem;
24
25   double a, b; //parameter values for linear or affine linear
       conversion equation
26
27   int numScen, numTimeSteps, numDiscrete, lifeTime;
28
29   string convTech;
```

```
30
31    double baseCost, CRF,r{0.06},capN{100}, econScaleFac{0.7}, maintFac
        {0.05};
32    //cost0=base cost, cap0=base capacity, capN=max capacity
33    //econScaleFaf= economies of scale factor, eff=efficiency
34
35    vector<double> sizes; //vector with numDiscrete size intervals for
        conversion technology
36
37    vector<vector<double>> output; //vector with output values in unit
        energy MW, will be initialized in setConversionFunction
38
39    vector<Variables> designVarBin;
40    Constraints designConBin;
41
42    vector<Variables> designVarDisc;
43    vector<Constraints> designConDisc;
44
45    vector<vector<Variables>> produced;
46    vector<vector<Constraints>> exported;
47
48  public:
49    friend EnergySystem;
50
51    string unitName;
52
53    Conversion(vector<vector<double>>* inputSource,EnergySystem* enrg);
        //constructor
54
55    Conversion* getPointer(){return this;};
56
57    void setCostFunction(double unitCost, double maxCap, int
        numSizeIntervals, int CRFtime, string type,bool VSS=false, double
        EVPsize=0); //set CAPEX cost function, dependent on installed size
58
59    void setConversionFunction(double constA=1.0, double constB=0.0); //
        set output conversion function, how much energy produced
60
61    vector<vector<Variables>>* getOutput(){return &produced;}; //returns
         output value for timestep t and scenario h, only one double value
62
63    void optimizeCapacity();
64
65    void addCAPEX(vector<Constraints>& objectiveFunction, vector<
        Constraints>& budget);
66
67  };
68
69  }
70
71  #endif
```

Listing 5.4: The `Conversion` class header file.

In Listing 5.4 we have:

1. `Conversion(vector<vector<double>>* inputSource, EnergySystem* enrg)`
   - The constructor initialises system variables `input, linkToSystem, numScen` and `numTimeSteps`.
   - The constructor also declares class variable `produced` and constraint `exported`. `produced` is constrained by `exported`, which equals the energy output of the `Conversion` object, `output`.
2. `setCostFunction(double unitCost, double maxCap, int numSizeintervals, int CRFtime , string type, bool VSS=false, double EVPsize=0)`
   - The function sets the parameters for the discrete cost function described in Section 5.3. The list below shows which class attributes that the user input parameters correspond to.
     I `unitCost=baseCost`
     II `maxCap=capN`
     III `numSizeintervals=numDiscrete`
     IV `CRFtime=lifeTime`
     V `type=unitName`
   - The client specifies both value and unit of the most influential cost parameters in Equation 5.6, namely `maxCap` and `unitCost`.
   - In addition, the function sets the `designVar` vector and calculates the Capital Recovery Factor (CRF), defined in Equation 5.4, from parameters `CRFtime`, and `rate`.
3. `setConversionFunction(double constA=1.0, double constB=0.0)`
   - The member function allows the client to specify or change any affine linear energy conversion model,

     $$Output_{t,h} = A \cdot Input_{t,h} + B$$

     where A is `constA`, ususally denoting an efficiency value relating the energy input and output of the converter. B is `constB`. As an example:

     $$PowerOutSolar = Efficiency_{PV} \cdot SolarRadiation$$

     The $Efficiency_{PV}$ denotes A (`constA`), and B (`constB`) is set to zero.
   - It converts the `inputSource` from a `primarySource` object to `output` in the form of utility.
4. `getOutput()`
   - Inline function `getOutput()` returns a pointer to the vector `produced` and is used to forward the production from the `Conversion` object to other objects.

- The implementation is identical to the `getOutput()` function in the `UncertainParam` class in Section 5.2.2.

5. `optimizeCapacity()`.

   - The function sets the binary $1st$ stage and discrete $2nd$ stage constraint `designConBin` and `designConDisc`, respectively.
   - Output from the conversion object is calculated and added to constraint `exported` together with `produced`.

6. `addCAPEX(vector<Constraints>& objectiveFunction, vector<Constraints>& budget)`

   - It is an auxiliary function that is used back-end in the `ObjectiveFunction` class to add the resulting capital expense from the conversion object.

Currently, the interest rate and economies of scale factor, `rate` and `econScaleFac`, have default values, but it is a simple task to make it modifiable. Moreover, in function `setCostFunction (double unitCost, double maxCap, int numSizeintervals, int CRFtime, string type, bool VSS=false, double EVPsize=0)` there is the option to calculate the Value of Stochastic Solution (VSS) as defined on page 12. To calculate the VSS the client has to set `VSS=true` and specify the nominal design, the design from the Expected Value Problem (EVP), in `EVPsize`. This way, the Expectation of Expected Value Problem (EEVP) and subsequently VSS is obtained. The stochastic solution and the VSS cannot be calculated simultaneously.

5.2.5 The `Utility` class

The purpose of the `Utility` class is to accumulate all the flows of a given utility type such as electricity or heat. The flows originate from multiple `Conversion` objects. From the `Utility` object the utility is forwarded to an `EndUser` object with demand for that particular utility type. The `Utility` class allows the client to include multiple different energy sources and conversion technologies in the flexible RES. This is an essential property of the OOP framework as using multiple energy sources is a potential solution to the reliability problem mentioned in the introduction.

```
1  //UTILITY class
2  #ifndef UTILITY_H
3  #define UTILITY_H
4  #include "headerFiles.hpp"
5  #include "Conversion.hpp"
6  #include "EnergySystem.hpp"
7
8  namespace decomposition
9  {
10
11
12 class Utility
13 {
14 private:
15
16   EnergySystem* linkToSystem;
17
```

```
18    int numScen, numTimeSteps;
19
20    vector<vector<vector<Variables>>*> inputSources;
21    vector<double> inputFractions;
22
23  public:
24    friend Conversion;
25    friend EnergySystem;
26
27    string name;
28    bool uncertainPrice{false};
29    double constImportPrice{300}, constFiT{70}, surplusFiT{0.5};
30    vector<vector<double>> importPrice;
31
32
33    vector<vector<Constraints>> producedCon;
34
35    vector<vector<Variables>> exported;
36    vector<vector<Variables>> imported;
37    vector<vector<Variables>> surplus;
38
39
40    Utility(vector<vector<double>>* price, EnergySystem* enrg, string
        utilityName); //constructor
41
42    void addUtility(vector<vector<Variables>>* input, double fraction);
        //adding energy from conversion to this utility type, fraction
        specifies amount of
43                                      //
44
45    vector<vector<Variables>>* getOutput(){return &exported;};
46    vector<vector<Variables>>* getImportVariable(){return &imported;};
47
48    Variables& getSurplus(int t, int h){return surplus[t][h];};
49    Variables& getExport(int t, int h){return exported[t][h];};
50    Variables& getImport(int t, int h){return imported[t][h];};
51
52    void setFiT(double FiT){constFiT=FiT;};
53    void setSurplusFiT(double FiT){surplusFiT=FiT;};
54
55
56    void setUtilityExportConstraint();
57
58  };
59
60  }
61
62  #endif
```

Listing 5.5: The `Utility` class header file.

In Listing 5.5 we have:

1. `Utility(vector<vector<double>>* price, EnergySystem* enrg, string utilityName)`
    - The constructor on line 40 initialises system and class variables, and imports a user specified import price from an `UncertainParameter` object.
    - Class variables `exported` and `surplus` are defined as the amount of utility exported to end user to cover demand, and say additional exported to the grid, respectively.
    - The utility prices from `price` are stored in vector `importPrice`. However, the `importPrice` can also be set to be a constant through variable `constImportPrice`.
2. `addUtility(vector<vector<Variables>>* input, double fraction)`
    - The member function imports utility from a `PrimarySource` or `Conversion` object and stores a pointer to the source in vector `inputSources`.
    - The input variables are added to the constraints in vector `producedCon` to constrain the total output from the `Utility` object.
3. `getSurplus(int t, int h)`, `getExport(int t, int h)` and `getImport(int t, int h)`
    - Returns the `surplus`, `export` or `import` variable from time-step `t` and scenario `h`.
    - Primarily used in the `EndUser` object to ensure that utility demand is met through production (`export`) and/or import (`import`), and to calculate the potential surplus in the case of additional utility production.
4. `setFiT(double FiT)` and `setSurplusFit(double FiT)`
    - A feed-in tariff is a policy mechanism designed to accelerate investment in renewable energy technologies by offering long-term contracts to renewable energy producers. Under a feed-in tariff, energy producers are paid a cost-based price for the renewable electricity they supply to the grid ($https://en.wikipedia.org/wiki/Feed-in_tariff$, 08.06.21) [15].
    - The two functions let the client specify feed-in-tariff/sales price of utility exported to cover end user demand and any surplus utility, respectively.
5. `setUtilityExportConstraint()`
    - `void` function `setUtilityExportConstraint()` sets the `producedCon` constraints.

The `fraction` parameter in function `addUtility(input, fraction)` makes it possible to distribute the output from one energy source to multiple utility types. `fraction` specifies how much of the `input` that is converted to the utility type which the function is called upon. This property is helpful in cases such as the natural gas combined cycle (NGCC) described in Section 4.2.

Constraint `producedCon` ensures that the variables `exported` and `surplus` are constrained to satisfy energy balances. The sum of `exported` and `surplus` for each time-step and scenario should be equal to the total input of utility in that time-step and scenario. `surplus` is defined as the potential surplus of utility if production surpasses demand. The demand of each utility is found in the end user object (see Section 5.2.6).

To summarize, the back-end implementation ensures consistent energy balances. In other words that the total output of utility does not surpass the total input of utility. This way, the client does not have to specifically specify input-output constraints for the utility object. However, `setUtilityExportConstraint()` has to be called explicitly by the client after all utility sources have been added. This is to avoid the "Out-of-turn definition of variable" error in GOSSIP which is invoked if a variable is set after a constraint.

5.2.6 The `EndUser` class

End users are the energy sinks, or customers, of the energy system. An end user object can require multiple different utilities such as heat, electricity, steam, etc. Pointers to the different demand profiles of the `EndUser` object are stored in the map `demandMap` together with a key that points to the associated `Utility` object. Utility is only imported from an `Utility` object to an `EndUser` object if there is demand for that utility in the `demandMap`.

```
1  //END USER CLASS
2  #ifndef ENDUSER_H
3  #define ENDUSER_H
4  #include "headerFiles.hpp"
5  #include "Utility.hpp"
6  #include "EnergySystem.hpp"
7
8  namespace decomposition
9  {
10
11 class EndUser
12 {
13 private:
14
15   EnergySystem* linkToSystem;
16
17   int numScen,numTimeSteps;
18
19   vector<Utility*> inputUtilities;
20   vector<Utility*> importUtilities;
21
22   map<Utility*,vector<vector<Constraints>> > demandMap;
23
24 public:
25   friend EnergySystem;
26   friend Utility;
27
28   EndUser(EnergySystem* enrg);
29
30   void addUtilityDemand(Utility* utilityType, vector<vector<double>>*
       input);//imports user demand from uncertain parameter object, adds
        value to demandMap with constriants
```

```
31
32    void addUtilitySource(Utility* utilityType); //imports energy from
        given utility source/type, e.g. electricity
33
34    void setUtilityImportConstraint(Utility* utilityType);
35
36    void getUtilityExport(vector<Objective>& objectiveFunction);
37    void getUtilityImport(vector<Objective>& objectiveFunction);
38
39  };
40
41  }
42
43
44  #endif
```

Listing 5.6: The `EndUser` class header file.

In Listing 5.6 we have:

1. `EndUser(EnergySystem* enrg)`
   - The constructor initialises system and class variables `numScen`, `numTimeSteps` and `linkToSystem`.
2. `addUtilityDemand(Utility* utilityType, vector<vector<double>>* input)`
   - Function imports the end user demand of utility `utilityType` from source `input` (uncertain parameter object) and adds it to the constraint with key `utilityType` in `demandMap`.
   - From the utility pointer `utilityType` the `imported` variable is accessed and added to the constraint in `demandMap`.
3. `addUtilitySource(Utility* utilityType)`
   - The function adds the utility type `utilityType` to vector `inputUtilities`.
4. `setUtilityImportConstriant(Utility* utilityType)`
   - The function adds utility from all sources in `inputUtilites` to the respective constraint in `demandMap`, and sets the respective constraints.
5. `getUtilityExport(vector<Objective>& objectiveFunction)` and `getUtilityImport(vector <Objective>& objectiveFunction)`
   - The two inline functions are back-end help functions with the sole purpose of adding operating income and expenses to the objective function in the `ObjectiveFunc` object.

The `imported` variable is added to `demandMap` to ensure that in the case of deficit renewable utility, end user demand is met by importing utility at an import cost. As mentioned in Section 5.2.5, the import price is encapsulated in the utility object.

Because all aforementioned function operations are implemented without input from the client, the client only has to specify "big picture" information flows such as the demand, and input and output sources.

5.2.7 The `ObjectiveFunction` class

By constructing a separate **`ObjectiveFunction`** object, the client can optimize the flexible RES design with respect to several different goals. For instance, the objective could be to minimize total emissions or cost, or maximize reliability of renewable energy. In the current implementation of the framework, the default objective is to minimize project lifetime cost. However, a class member function to set a different objective can be implemented easily without changing the rest of the existing framework.

```
1  //Objective class
2  #ifndef OBJECTIVEFUNC_H
3  #define OBJECTIVEFUNC_H
4  #include "headerFiles.hpp"
5  #include "EndUser.hpp"
6  #include "EnergySystem.hpp"
7
8  namespace decomposition
9  {
10   class ObjectiveFunction
11   {
12   private:
13     int numScen, numTimeSteps;
14
15     EnergySystem* linkToSystem;
16
17     double budget;
18     vector<Constraints> budgetCon;
19
20     vector<Objective> objFunc; //vector to store objective function,
       function to be minimized by default
21
22
23
24   public:
25     friend EnergySystem;
26
27     ObjectiveFunction(EnergySystem* enrg, double setBudget=100000000);
28
29     void addOPEX(EndUser* user);
30
31     void addCAPEX(Conversion* technology);
32
33     int getNumScen(){return numScen;};
34
35     void setObjective(); //setting the objective function,
       minimization by default
36
37   };
38
39
40
```

```
41 }
42
43 #endif
```

Listing 5.7: The `ObjectiveFunciton` class header file.

In Listing 5.7 we have:

1. `ObjectiveFunction(EnergySystem* enrg, double setBudget=20000000)`
   - The constructor on line 25 initialises system variables and the objective-vector `objFunc`.
   - It also adds the potentially user-defined `setBudget` to budget constraint `budgetCon` .
2. `addCAPEX(Conversion* technology)`
   - The member function adds the capital expense of `Conversion` object `technology` to the objective function `objFun` and constraint `budgetCon`.
3. `addOPEX(EndUser* user)`
   - The function adds the operational expenses and revenues from `EndUser` object `user`.
4. `setObjective()`
   - This function is called after all relevant variables has been added by the client. It sets the final objective function and the capital expense budget constraint.

5.3 Model objective, equations and constraints

By constructing necessary objects and calling relevant functions through the user interface, a two-stage stochastic MILP problem is formulated in the implementation. This includes formulating the problem objective function, as defined in Section 5.3.1, and constraints that are necessary for a consistent model formulation, as defined in Section 5.3.2.

5.3.1 The objective

Currently, the default objective is to minimize the expected lifetime cost of the energy system as defined by Equation 5.1. This can be split into the capital costs, $CAP(x)$, and the expected operating cost over the different scenarios, where OP_h is the operational cost in scenario h. The capital costs are determined in the 1^{st} stage, and operational costs are incurred in the 2^{nd} stage.

$$OBJ_{min} = CAP(\boldsymbol{x}) + \sum_{h} p_h \cdot OP_h(\boldsymbol{y}_h) \qquad \forall \text{ h} \in \text{S} \tag{5.1}$$

Here $\boldsymbol{x}$ is a vector of binary decision variables in the 1^{st} stage, p_h is probability of scenario h and $\boldsymbol{y}_h$ is a vector of scenario dependent operational decision variables in the 2^{nd} stage.

S defines the set of possible scenarios h. The size of $\boldsymbol{x}$ is dependent on the number of different conversion technologies installed and the number of different sizes to chose from for each conversion technology. The size of $\boldsymbol{y}_h$ depends on number of scenarios, time-steps, utilities produced, end users and possible conversion technologies.

Capital cost estimation [13] [17]

The project capital cost can be written as,

$$CAP(\boldsymbol{x}) = \sum_{i \in I} FC_i(x_i) \tag{5.2}$$

where FC_i is defined in Equation 5.3 and I is the set of constructed conversion technologies. The total capital cost of conversion technology i consists of an investment cost $C_{i,base}$ multiplied by capacity installed J_i, a maintenance cost factor $F_{m,i}$ and the Capital Recovery Factor, CRF_i.

$$FC_i(x_i) = J_i(x_i) \cdot C_{i,base} \cdot F_{m,i} \cdot CRF_i \tag{5.3}$$

The Capital Recovery Factor is defined as in Equation 5.4 where r is the interest rate and T_i denotes lifetime of technology i.

$$CRF_i = r \cdot \frac{(1+r)^{T_i}}{(1+r)^{T_i} - 1} \tag{5.4}$$

As mentioned in Chapter 3, the 1^{st} stage variables must be discrete for guaranteed convergence of the NGBD algorithm. Consequently, the capacity J_i is a function of the binary decision variable $z_{i,j}$ and a discrete capacity function, $S_{i,j}$, where subscript i denotes conversion technology and j capacity interval.

$$J_i(x_i) = \sum_{j=0}^{d} S_{i,j} \cdot x_{i,j} \qquad \mathrm{x}_{i,j} \in \{0,1\} \tag{5.5}$$

The capacity, $S_{i,j}$, is calculated from Equation (5.6) where $S_{i,max}$ is the user-defined maximum capacity that can be installed.

$$S_{i,j} = j \cdot \left(\frac{S_{i,max}}{d-1}\right) \qquad \forall\ \mathrm{j} \in \{0,\ldots,\mathrm{d}\} \tag{5.6}$$

Operating cost estimation

The operating costs in the second term of Equation 5.1 is in each scenario summed up from discrete time t_0 to t_{final} for each utility-type, u. It calculates the expected revenue from production over the entire project lifetime T and is formulated in Equation 5.7.

$$\begin{aligned} OP_h(\boldsymbol{y}) = \sum_{u \in U} \sum_{t_0}^{t_{final}} & y_{u,t,h}^{import} \cdot UiP_{u,h} \\ & - y_{u,t,h}^{demand} \cdot FiT_u \\ & - y_{u,t,h}^{redundant} \cdot FiT_u^{redundant} \\ & \forall\ h \in S \end{aligned} \tag{5.7}$$

Utility flow $y_{u,t,h}^{import}$ is energy that is imported externally from grid to end user if production is less than demand, and $y_{u,t,h}^{demand}$ and $y_{u,t,h}^{redundant}$ is energy exported from energy system to end users. Here $demand$ and $redundant$ indicates if the energy is exported to cover end user demand or if it is redundant production exported to grid, respectively. All energy flows are indexed by scenario h and time t. UiP$_{u,t,h}$ is the price of imported utility u in scenario h whereas FiT_u and $FiT_u^{redundant}$ are constant feed-in tariffs for exported utility, u, to end user and back to grid, respectively.

5.3.2 Framework constraints

The number of constraints in the user input model depends on the number of scenarios, time-steps, conversion technologies, utilities, and end users. Some constraints are individual, e.g specific for one object such as the design constraint in Equation 5.8, whereas equations such as 5.11 involve different classes and multiple objects. Constraints are generated automatically and declared in the implementation, but the client has to call `setConstraint(...)` functions for several of the framework objects. This is to ensure that $1st$ stage constraints are set after $1st$ and $2nd$ stage variables and before any $2nd$ stage constraints. Nonetheless, the tedious and repetitive work of resizing vectors and constraining simple input-output variables is done back-end.

Design constraints

To ensure that only one size, $S_{i,j}$, is selected for each `Conversion` object i, the variable constraint formulated in Equation 5.8 is imposed on the system.

$$\sum_{j=0}^{d} x_{i,j} = 1 \qquad \mathrm{x}_{i,j} \in \{0,1\} \tag{5.8}$$

In addition, the client can specify a budget constraint that limits the total investment cost related to conversion technologies. The budget constraint is defined by Equation 5.9 where i denotes a `Conversion` object from the set I of installed conversion technologies. J_i denotes the installed capacity of `Conversion` object i with unit cost $C_{i,base}$.

$$CAPEX_{budget} \geq \sum_{i \in I} J_i \cdot C_{i,base} \tag{5.9}$$

Energy balance constraints

Each `Conversion` object, i, has a production constraint as defined by Equation 5.10 to satisfy energy balance constraints, i.e that the amount of exported energy does not surpass the amount of produced energy in each time-step t for each scenario h. $output_{t,h}$ is energy output from the `Conversion` object in time-step t and scenario h. J_i scales the energy output to reflect the installed capacity.

$$y_{u,t,h}^{export,i} = output_{t,h} \cdot J_i, h \tag{5.10}$$

Furthermore, each `Utility` object u has a production constraint as defined by Equation 5.11. This is to ensure that the amount of utility exported to end user (demand) and back to grid (redundant) is equal to the sum of utility (export) from all relevant utility sources (`Conversion` objects i).

$$\sum_{i \in I} y_{u,t,h}^{export,i} = y_{u,t,h}^{demand} + y_{u,t,h}^{redundant} \tag{5.11}$$

Demand and import constraint

It is assumed that there are no arbitrage opportunities in the energy market. This entails that it is not beneficial for the system to import utility from the market and export this to the end users. However, to ensure that the end user demand is met, there must be a possibility to cover deficit utility production through import. To enforce the arbitrage assumption and simultaneously ensure satisfied end user demand, an end user demand constraint is defined for each type of utility that an end user requires. The demand constraint is defined in Equation 5.12 where $demand_{u,t,h}$ is the end user demand of utility u, $y_{u,t,h}^{demand}$ is utility exported to the end user and $y_{u,t,h}^{import}$ is utility imported from the market. All variables are indexed by `Utility` object/type u, time t and scenario h.

$$demand_{u,t,h} = y_{u,t,h}^{demand} + y_{u,t,h}^{import} \tag{5.12}$$

5.4 User input model: Setting up a simple program

The coming sections explain the objects and function calls required for any program within the developed OOP framework to compile and run. Abbreviations for the different objects, for instance the term UPO for an `UncertanParameter` object, are listed on page xii.

5.4.1 Uncertain parameter objects and related input-files

The list below shows the framework classes that can involve uncertain parameters and lists where typical uncertainties in a flexible RES are included in the OOP framework.

1. One `UncertanParameter` object (UPO) per `PrimarySource` object (PSO), representing the input to the PSO. Example: Wind speed for primary source wind.
2. One UPO per `Utility` object (UTO), representing the import price of that utility. Example: The volatile market spot price for electricity, e.g the cost related to importing electricity from the grid.
3. One UPO for each utility type that an `EndUser` object needs. Example: An end user that needs heat and electricity results in two UPOs to give the user demand profile for heat and electricity, respectively.

5.4.2 Energy system objects

In addition to the number of related uncertain parameters described above, the following objects have to be constructed to ensure the compilation and correct execution of a program in the OOP framework. Note that construction of the objects should follow the line of order below.

1. One energy system object (ESO).
2. One primary source object (PSO), representing an intermediary energy source.
3. One conversion object (CVO) that turns the input from a PSO into output to a UTO.
4. One utility object (UTO), an intermediary sink, to accumulate utility from multiple conversion objects.
5. One end user object (EUO) representing the final sink.
6. One objective object (OBJO) to define a function for the program to minimize/maximize.

5.4.3 Function calls

The following function calls have to be made to set the necessary system variables and constraints. The order in which the calls are made depends on function input parameters and whether the function involves $1st$ or $2nd$ stage variables, or $1st$ or $2nd$ stage constraints. $1st$ stage variables must be defined and set before $2nd$ stage variables, which in turn must be defined and set before $1st$ and $2nd$ stage constraints.

```
ESO.importProbabilities(string filepath, vector<double>& prob)

CVO.setCostFunction(double unitCost, double maxCap, int
numSizeIntervals, double efficiency, string type)

CVO.setConversionFunction(double constA, double constB)

EUO.addUtilityDemand(Utilty* type=&UT, UP.getOutput())

EUO.addUtilitySource(Utility* type=&UT, type.getOutput())

CVO.optimizeCapacity()

UTO.addUtility(CV.getOutput(), double fraction)

UTO.setUtilityExportConstraint()

EUO.setUtilityImportConstraint(Utiliy * type=&UT)

OBJ.addOPEX(EndUser * user=&EU)

OBJ.addCAPEX(Conversion* technology=&CV)

OBJ.setObjective()
```

Listing 5.8: Necessary function calls to set system and create an executable program. The capital letters in front of the function call in Listing 5.8 refers to the class which the function is a member of. See the list of abbreviations on page xii for information about which object the capital letters symbolize.

CHAPTER

SIX

CASE STUDIES

This chapter illustrates and further explains how the object-oriented framework works through three examples with increasing complexity. First, a small but viable system is presented in Section 6.1 to show the essential components of a solvable user input model. In Section 6.2 the model is expanded to replicate the model in [17] without the implementation of batteries. This is to verify the correctness of the OOP framework by showing that the results obtained are identical. Lastly, in Section 6.3 the user input model is further expanded to include multiple utility types and end users. This is to show the extensibility of framework components and scalability of the user input model.

The objective in all three examples is to determine the optimal design and operation under uncertainty that minimizes lifetime cost. The intermittent nature of both renewable energy sources and user demand, as well as the volatile market spot price for electricity, introduces uncertainty into the system.

The examples are, listed after increasing complexity:

- **Example 1:** A simple system
- **Example 2:** A system consisting of multiple energy sources
- **Example 3:** A more complex system with multiple energy sources, utilities and end users

6.1 A small system

6.1.1 The system

The model consists of renewable energy source wind, energy conversion through wind turbines, and an end user with one utility demand (electricity). The system has three uncertain variables, namely wind speed, the market spot price of electricity, and user demand. However, it should be noted that the user can set both the import price and demand to a constant. The physical system is illustrated in Figure 6.1.

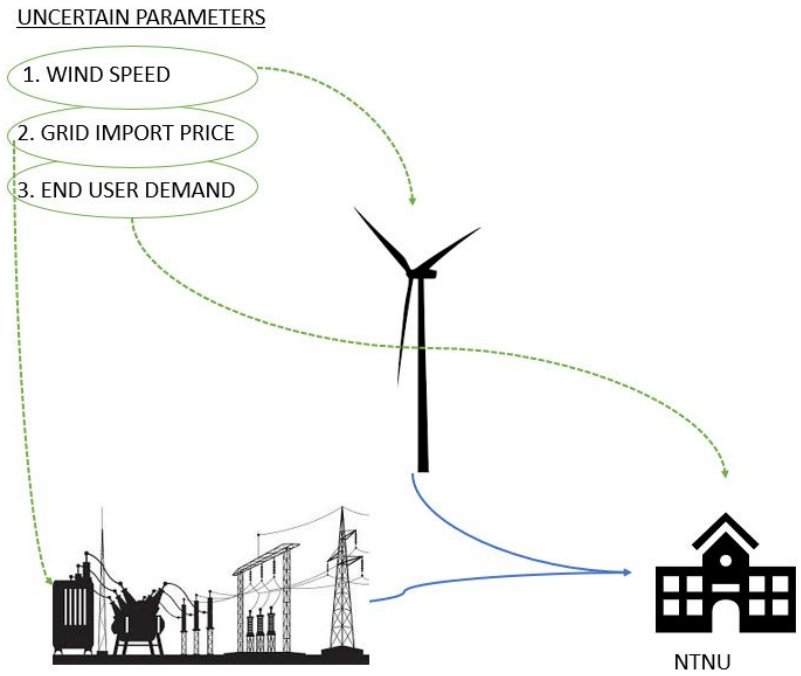

Figure 6.1: A simple illustration of the physical system in Example 1. Blue arrows represent electricity flow.

The wind turbine model in Equation 6.1 and 6.2 is nonlinear with respect to the wind speed, thus, the expression in the $2nd$ case in Equation 6.2 is rearranged to fit the affine linear input-output relationship in the `Conversion` class object. The resulting wind turbine model and input parameters (`constA` and `constB`) are listed in Table 6.1 and 6.2, respectively.

The rated power output, $P_{t,h}^{WT}$, from a wind turbine is modelled after Equation 6.1.

$$P_{t,h}^{WT} \leq q_{t,h} \cdot Z^{WT} \tag{6.1}$$

$q_{t,h}$ is a function of wind speed defined by Equation 6.2, and specific for the model [14]. Z^{WT} is a function of the first stage decision variables and denotes the number of wind turbines installed.

$$q_{t,h} = \begin{cases} 0, & \text{if } W_{t,h} \leq W_{min} \\ q_d \cdot \frac{W_{t,h}^3 - W_{min}^3}{W_d^3 - W_{min}^3}, & \text{if } W_{min} \leq W_{t,h} < W_d \\ q_d, & \text{if } W_d \leq W_{t,h} < W_{max} \\ 0, & \text{if } W_{t,h} \geq W_{max} \end{cases} \tag{6.2}$$

q_d is the maximum production capacity of one wind turbine, obtained if the wind speed is between w_d and w_{max}. W_{min} and W_{max} is the cut-in and cut-out wind speed, respectively, and $W_{t,h}$ the wind speed in time-step t and scenario h. The wind model parameters are listed in Table 6.1.

Symbol	Unit	Value
q_d	MW	8
W_d	m/s	14
W_{min}	m/s	4
W_{max}	m/s	25
$\frac{q_d}{W_d^3 - W_{min}^3}$	MW/$(m/s)^3$	0.002985
$\frac{-q_d \cdot W_{min}^3}{W_d^3 - W_{min}^3}$	MW	-0.1910

Table 6.1: Parameters used in the wind model in Equation 6.2.

The overall objective is to minimize project costs. Renewable energy is exported to end user at a constant feed-in tariff until the end user demand is met. Any redundant renewable energy is exported to the grid at a lower feed-in tariff, and any deficit production is covered by importing electricity from the grid at market spot price. The key result from the program is the $1st$ stage decision variables, namely the number of wind turbines installed. The process flowsheet is illustrated in Figure 6.2, and the user input parameters and model is listed in Table 6.2 and shown in Listing 6.1, respectively.

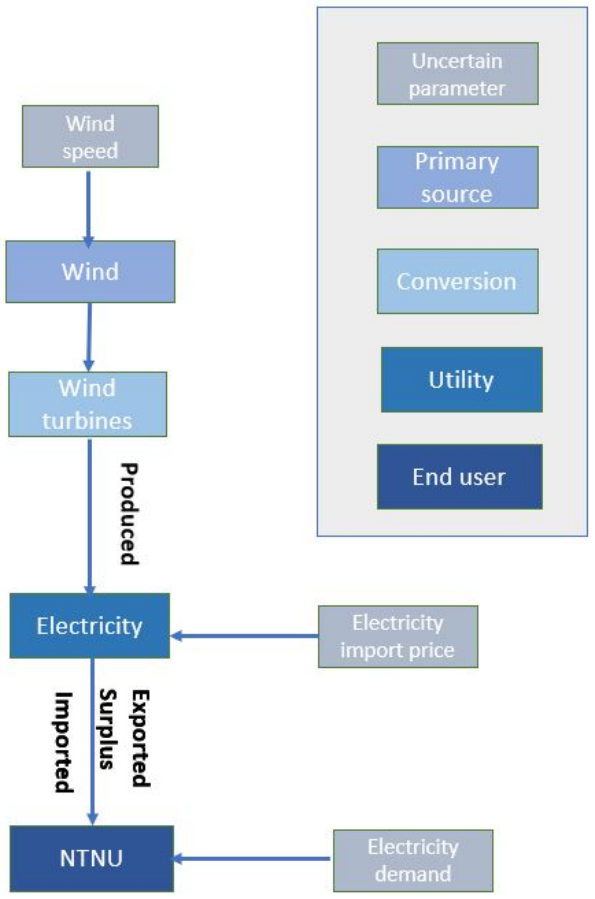

Figure 6.2: Flowsheet illustrating the relationship between objects in the renewable energy system illustrated in Figure 6.1. The corresponding user input model is shown in Listing 6.1. Produced, exported, surplus and imported are system variables.

6.1.2 User model

Listing 6.1 shows the user input model of the system illustrated in Figure 6.1 and with corresponding process flowsheet in Figure 6.2. User input parameters are listed in Table 6.2. The code listing has also been split into segments to show the order of declaration for $1st$ and $2nd$ stage variables and constraints.

Table 6.2: User input parameters in Listing 6.1. Stage denotes whether the parameter is used in the $1st$ or $2nd$ stage of the resulting two-stage MILP problem formulation.
([1]time-steps per day, [2]no. wind turbines)

Stage	Name	Unit	Value
`RES` object			
1	time	- [1]	24
1	scen	-	12
`WT` object			
1	unitCost	$	1 200 000
1	maxCap	- [2]	18
1	numSizeIntervals	-	10
1	CRFtime	years	20
2	constA	$MW/(m/s)^3$	0.002985
2	constB	MW	-0.1910
`Electricity` object			
2	fraction	-	1.0
`minimize` object			
2	setBudget	$	20 000 000

```
#include "headerFiles.hpp"
#include "MSCfiles.hpp"

using namespace decomposition;

//----------Simplest possible program-------------//
int inputmodel(std::vector<double> &weights)
{
  //Setting energy system time-steps=24 hours and scenarios specs=12
    scenarios per hour
  EnergySystem RES(12,24);
  //Importing probabilities for each of the 12 scenarios, parameter is
     filepat and vector weights
  RES.importProbabilities("Examples/GOSSIP_library/MSC_v3/ImportFIles/
    UncertaintyData_v2/probabilities.csv", weights);

  //----------Importing scenario profiles for all relevant uncertain
    parameters-------------//

  //Electricity demand profile
  UncertainParam ELdemand("Examples/GOSSIP_library/MSC_v3/ImportFIles/
    UncertaintyData_v2/param4.csv",&RES);
  //Electricity import price from grid
  UncertainParam GridImportPrice("Examples/GOSSIP_library/MSC_v3/
    ImportFIles/UncertaintyData_v2/param3.csv",&RES);
  //EFfective power throughput
  UncertainParam WindSpeed("Examples/GOSSIP_library/MSC_v3/ImportFIles
    /UncertaintyData_v2/param2new.csv",&RES);
  //Need one uncertain parameter for each primary source = energy
    source input
  PrimarySource Wind(WindSpeed.getOutput(),&RES);

  //.-----------Setting 1st stage variables---------//

  //Input(vector with wind output, link to energy system)
  Conversion WT(Wind.getOutput(),&RES);
  //Setting cost function: Input (unit price, max capacity, number of
    discrete intervals, name)
  WT.setCostFunction(1200000,18,10,20,"WT");

  //-----------Setting 2nd stage variables---------//

  //Input(constant a, constant b)
  WT.setConversionFunction(0.002985,-0.1910);
  //Need one uncertain parameter for each type of utility = price of
    utility import
  //Input(vector with price, link to system, name)
  Utility Electricity(GridImportPrice.getOutput(),&RES,"EL");
  //Simple constructor
  EndUser NTNU(&RES);
  //Need one uncertain parameter for each type of utility demand for
    each end user
  //Input(link to relevant utility, vector with demand profile for
    that utiliy)
  NTNU.addUtilityDemand(&Electricity,ELdemand.getOutput());
  //Input(link to utility source for which there is demand)
  NTNU.addUtilitySource(&Electricity);
```

```
46   //Setting 1st stage capacity/design constraints
47   WT.optimizeCapacity();
48
49   //-------------------------Setting 2nd stage operational constraints
       ---------------------//
50
51   //Input(vector with energy outout from converion object, fraction of
       energy output to that utility type)
52   Electricity.addUtility(WT.getOutput(),1.0);
53   //Constraint to ensure utility exported does not surpass utility
       produced
54   Electricity.setUtilityExportConstraint();
55   //Constraint to ensure utility improted does not surpass utility
       produced
56   //Input(link to utility source for which there is a demand)
57   NTNU.setUtilityImportConstraint(&Electricity);
58
59   //----------Objective function-------//
60
61   //Simple constructor, sets CAPEX budget
62   ObjectiveFunction minimize(&RES, 20000000);
63   //Adding OPEX (operational expenses and income)
64   //Input(link to end user that the system exports energy to)
65   minimize.addOPEX(&NTNU);
66   //Adding CAPEX for energy system (capital expense for each
       conversion technology possibly installed)
67   //Input(link to conversion technology)
68   minimize.addCAPEX(&WT);
69   //Setting the final objective function after adding all expenses and
       revenues
70   minimize.setObjective();
71
72   //Returning number of scenarios as output of inputmodel
73   return minimize.getNumScen();
74
75 };
```

Listing 6.1: Resulting user input model for the system illustrated in Figure 6.1

6.2 Solar PV and wind turbine system

6.2.1 The system

In Figure 6.3, the system from Figure 6.1 is expanded to include renewable energy source solar radiation with corresponding conversion technology solar photo voltaic panels. The process flowsheet is illustrated in Figure 6.4.

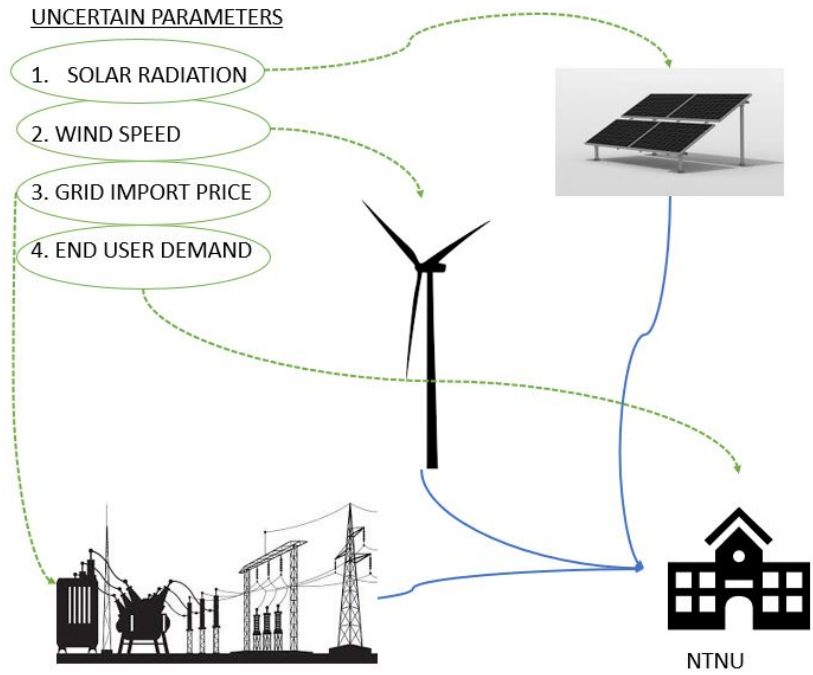

Figure 6.3: A simple illustration of the physical system in Example 2. Blue arrows represent electricity flow.

The power output per square meter of solar panels, $P_{t,h}^{PV}$, is modelled as,

$$P_{t,h}^{PV} = \eta^{PV}(I_{t,h}, \theta) \cdot I_{t,h} \cdot Z^{PV} \tag{6.3}$$

where η^{PV} is the efficiency, $I_{t,h}$ is solar radiation at time t and scenario h [MW/km^2], and θ the solar incidence angle. Z^{PV} is a function of the first stage decision variable and denotes how many m^2 of solar PV panels that is installed. For simplicity η^{PV} and θ are assumed constant.

The user input model in Listing 6.2 verifies the correctness of the framework as it gives the same results as the model in [17] run without the inclusion of batteries. The user demand and import price of electricity are the same as in the case study in Section 6.1, but now a sufficient amount of renewable energy can be ensured from two different sources and by two conversion technologies.

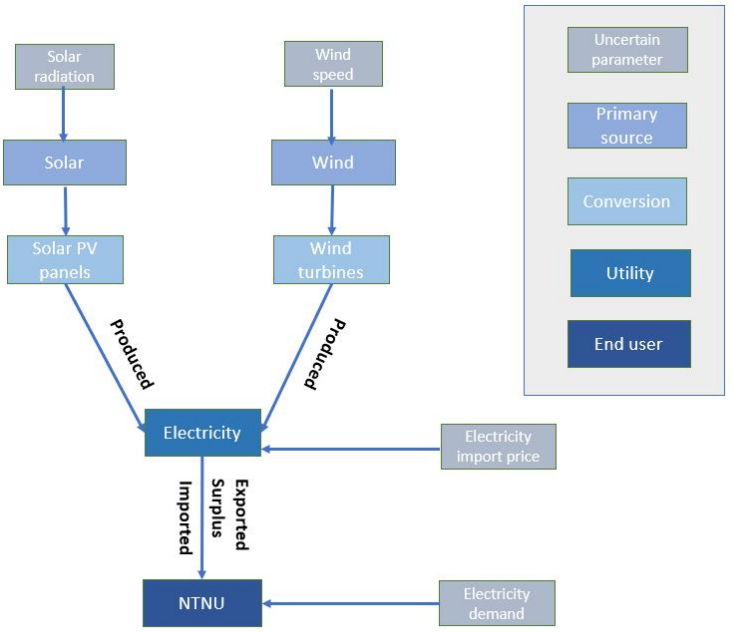

Figure 6.4: Flowsheet illustrating the relationship between objects in the renewable energy system illustrated in Figure 6.3. The corresponding user input model is shown in Listing 6.2. Produced, exported, surplus and imported are system variables.

6.2.2 User model

Some explanatory comments have been omitted in this example and the reader is referred to Chapter 5 for extensive explanations. The program in Listing 6.2 is the object-oriented equivalent of the model in [17] without implementation of batteries. User input parameters are listed in Table 6.3.

Table 6.3: User input parameters in Listing 6.2. Stage denotes whether the parameter is used in the $1st$ or $2nd$ stage of the resulting two-stage MILP problem formulation.

Stage	Name	Unit	Value
`RES` object			
1	time	- [1]	24
1	scen	-	12
`WT` object			
1	unitCost	$	1 200 000
1	maxCap	- [2]	18
1	numSizeIntervals	-	10
1	CRFtime	years	20
2	constA	$MW/(m/s)^3$	0.002985
2	constB	MW	-0.1910
`SolarPV` object			
1	unitCost	$	130
1	maxCap	m^2	72 000
1	numSizeIntervals	-	10
1	CRFtime	years	30
2	constA	-	$2{\cdot}10^{-7}$
2	constB	-	0.0
`Electricity` object			
2	fraction	-	1.0
2	fraction	-	1.0
`minimize` object			
2	setBudget	$	20 000 000

```
#include "headerFiles.hpp"
#include "MSCfiles.hpp"

using namespace decomposition;

int inputmodel(std::vector<double> &weights)
{
  EnergySystem RES(12,24);

  RES.importProbabilities("Examples/GOSSIP_library/MSC_v3/ImportFIles/
    UncertaintyData_v2/probabilities.csv", weights);

  UncertainParam SolarRadiation("Examples/GOSSIP_library/MSC_v3/
    ImportFIles/UncertaintyData_v2/param1.csv",&RES);
  UncertainParam WindSpeed("Examples/GOSSIP_library/MSC_v3/ImportFIles
    /UncertaintyData_v2/param2new.csv",&RES);
  UncertainParam GridImportPrice("Examples/GOSSIP_library/MSC_v3/
    ImportFIles/UncertaintyData_v2/param3.csv",&RES);
  UncertainParam ELdemand("Examples/GOSSIP_library/MSC_v3/ImportFIles/
    UncertaintyData_v2/param4.csv",&RES);

  PrimarySource Solar(SolarRadiation.getOutput(),&RES);
  PrimarySource Wind(WindSpeed.getOutput(),&RES);

  //-------------Setting 1st stage variables--------//
  Conversion SolarPV(Solar.getOutput(), &RES);
  Conversion WT(Wind.getOutput(),&RES);

  SolarPV.setCostFunction(130,72000,10,30,"PV");
  WT.setCostFunction(1200000,18,10,20,"WT");

  //-----------Setting 2nd stage variables---------//
  SolarPV.setConversionFunction(0.0000002,0);
  WT.setConversionFunction(0.002985,-0.1910);

  Utility Electricity(GridImportPrice.getOutput(),&RES, "EL");

  EndUser NTNU(&RES);

  NTNU.addUtilityDemand(&Electricity,ELdemand.getOutput());

  NTNU.addUtilitySource(&Electricity);
  SolarPV.optimizeCapacity();
  WT.optimizeCapacity();

  //--------Setting 2nd stage constraints--------//
  Electricity.addUtility(SolarPV.getOutput(),1.0);
  Electricity.addUtility(WT.getOutput(),1.0);

  //Constraint to ensure utility exported does not surpass utility
    produced
  Electricity.setUtilityExportConstraint();

  //Constraint to ensure utility imported does not surpass utility
    produced
  NTNU.setUtilityImportConstraint(&Electricity);

```

```
51   ObjectiveFunction minimize(&RES, 20000000);
52
53   //Adding OPEX (operational expenses and income) for end user
54   minimize.addOPEX(&NTNU);
55
56   //Adding CAPEX for energy system, for each technology possibly
       installed
57   minimize.addCAPEX(&SolarPV);
58   minimize.addCAPEX(&WT);
59
60   //Setting objective function
61   minimize.setObjective();
62
63   return minimize.getNumScen();
64 };
```

Listing 6.2: Resulting user input model for the system illustrated in Figure 6.3.

6.3 System with heat and electricity demand

6.3.1 The system

The system in Figure 6.5 with corresponding process flowsheet in Figure 6.6 is an expansion of the system in Figure 6.3. Another utility type and end user is introduced to show extensibility of the object-oriented framework. The end users now consist of the electricity requirement of a medium-sized university (`NTNU`) and the heating demand of a small neighbourhood (`TidemandsGate`). Solar panels (`solarPV`) and wind turbines (`WT`) can be used for generating electricity to `NTNU`, and solar thermal panels (`solarTP`) can be used for generating heat to `TidemandsGate`.

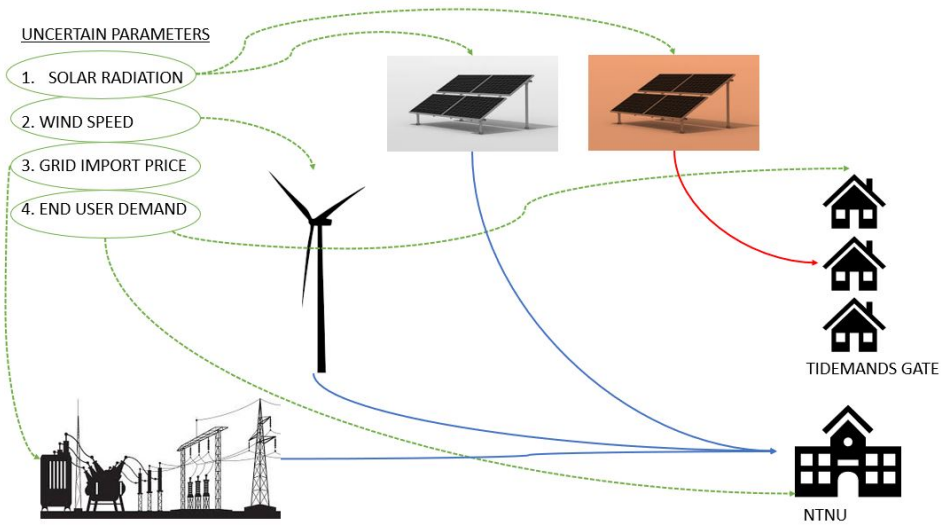

Figure 6.5: A simple illustration of the physical system in case study 3. The grey and red solar panels represent solar photo voltaic and thermal panels, respectively. Blue and red arrows represent electricity and heat flow, respectively.

The power output from the solar thermal panels, $P^{PT}_{t,h}$, is calculated from the surrogate model in Equation 6.4.

$$P^{PT}_{t,h} = \eta^{PT}(I_{t,h}) \cdot I_{t,h} \cdot Z^{PT} \tag{6.4}$$

η^{PT} is the efficiency and $I_{t,h}$ solar radiation at time t and scenario h [MW/km^2]. Z^{PT} is the first stage decision variable and denotes the installed area of solar thermal panels [m^2]. For simplicity η^{PT} is assumed constant.

If there is a deficit production of electricity and/or heat, electricity is imported from the grid at market spot price. Either for direct consumption or as an intermediary to generate heat. On the other hand, if user demand is satisfied, abundant electricity and heat can be exported to the grid or end user at a lower feed-in tariff.

Listing 6.3 is the problem formulation for the system illustrated in Figure 6.6. For simplicity the electricity demand of `NTNU` and the heating demand of `TidemandsGate` are equal, but the client can specify individual demand profiles. Also, because electricity can be used to produce heat in the case of deficit heat from renewables, the import prices of heat and electricity are assumed to be the same.

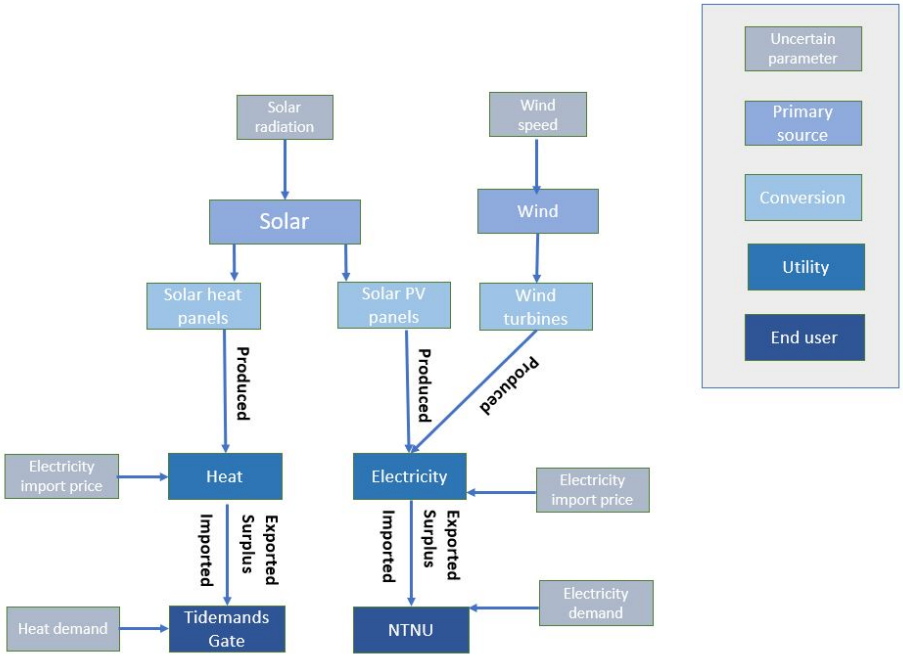

Figure 6.6: Flowsheet of a more complex system illustrating the different objects and relationships in the program listed on page 58. Produced, exported, surplus and imported are system variables.

6.3.2 User model

Listing 6.3 shows the user input model for the system illustrated in Figure 6.5. Explanatory comments have been removed and the reader is referred to Chapter 5 and Appendix A for more details about the individual class implementations. User input parameters are listed in Table 6.4.

Table 6.4: User input parameters in Listing 6.3. Stage denotes whether the parameter is used in the $1st$ or $2nd$ stage of the resulting two-stage MILP problem formulation.

Stage	Name	Unit	Value
`RES` object			
1	time	-[1]	24
1	scen	-	12
`WT` object			
1	unitCost	$	1 200 000
1	maxCap	-[2]	18
1	numSizeIntervals	-	10
1	CRFtime	years	20
2	constA	$MW/(m/s)^3$	0.002985
2	constB	MW	-0.1910
`SolarPV` object			
1	unitCost	$	130
1	maxCap	m^2	72 000
1	numSizeIntervals	-	10
1	CRFtime	years	30
2	constA	-	$2{\cdot}10^{-7}$
2	constB	-	0.0
`SolarTP` object			
1	unitCost	$	100
1	maxCap	m^2	72 000
1	numSizeIntervals	-	10
1	CRFtime	years	30
2	constA	-	$5{\cdot}10^{-7}$
2	constB	-	0.0
`Electricity` object			
2	fraction	-	1.0
2	fraction	-	1.0
`Heat` object			
2	fraction	-	0.9
`minimize` object			
2	setBudget	$	20 000 000

```
#include "headerFiles.hpp"
#include "MSCfiles.hpp"

using namespace decomposition;

int inputmodel(std::vector<double> &weights)
{
  EnergySystem RES(12,24);

  RES.importProbabilities("Examples/GOSSIP_library/MSC_v3/ImportFIles/
    UncertaintyData_v2/probabilities.csv", weights);

  UncertainParam SolarRadiation("Examples/GOSSIP_library/MSC_v3/
    ImportFIles/UncertaintyData_v2/param1.csv",&RES);
  UncertainParam WindSpeed("Examples/GOSSIP_library/MSC_v3/ImportFIles
    /UncertaintyData_v2/param2new.csv",&RES);
  UncertainParam GridImportPrice("Examples/GOSSIP_library/MSC_v3/
    ImportFIles/UncertaintyData_v2/param3.csv",&RES);
  UncertainParam ELdemand("Examples/GOSSIP_library/MSC_v3/ImportFIles/
    UncertaintyData_v2/param4.csv",&RES);
  UncertainParam HeatDemand("Examples/GOSSIP_library/MSC_v3/
    ImportFIles/UncertaintyData_v2/param4.csv",&RES);

  PrimarySource Solar(SolarRadiation.getOutput(),&RES);
  PrimarySource Wind(WindSpeed.getOutput(),&RES);

  //-------------Setting 1st stage variables--------//
  Conversion SolarPV(Solar.getOutput(), &RES);
  Conversion SolarTP(Solar.getOutput(),&RES);
  Conversion WT(Wind.getOutput(),&RES);

  SolarPV.setCostFunction(130,72000,10,30,"PV");
  SolarTP.setCostFunction(100,72000,10,30,"PT");
  WT.setCostFunction(1200000,18,10,20,"WT");

  //-----------Setting 2nd stage variables---------//
  SolarPV.setConversionFunction(0.0000002,0);
  SolarTP.setConversionFunction(0.0000005,0);
  WT.setConversionFunction(0.002985,-0.1910);

  Utility Electricity(GridImportPrice.getOutput(),&RES, "EL");
  Utility Heat(GridImportPrice.getOutput(),&RES,"HEAT");

  EndUser NTNU(&RES);
  EndUser TidemandsGate(&RES);

  NTNU.addUtilityDemand(&Electricity,ELdemand.getOutput());
  TidemandsGate.addUtilityDemand(&Heat,HeatDemand.getOutput());

  NTNU.addUtilitySource(&Electricity);
  TidemandsGate.addUtilitySource(&Heat);

  SolarPV.optimizeCapacity();
  SolarTP.optimizeCapacity();
  WT.optimizeCapacity();

  //--------Setting 2nd stage constraints--------//
```

```
52   Electricity.addUtility(SolarPV.getOutput(),1.0);
53   Heat.addUtility(SolarTP.getOutput(),0.9);
54   Electricity.addUtility(WT.getOutput(),1.0);
55
56   //Constraint to ensure utility exported does not surpass utility
       produced
57   Electricity.setUtilityExportConstraint();
58   Heat.setUtilityExportConstraint();
59
60   //Constraint to ensure utility imported does not surpass utility
       produced
61   NTNU.setUtilityImportConstraint(&Electricity);
62   TidemandsGate.setUtilityImportConstraint(&Heat);
63
64   ObjectiveFunction minimize(&RES,20000000);
65
66   //Adding OPEX (operational expenses and income) for end user
67   minimize.addOPEX(&NTNU);
68   minimize.addOPEX(&TidemandsGate);
69
70   //Adding CAPEX for energy system, for each technology possibly
       installed
71   minimize.addCAPEX(&SolarPV);
72   minimize.addCAPEX(&SolarTP);
73   minimize.addCAPEX(&WT);
74
75   //Setting objective function
76   minimize.setObjective();
77
78   return minimize.getNumScen();
79 };
```

Listing 6.3: Resulting user input model for the more complex system illustrated in Figure 6.5.

CHAPTER

SEVEN

RESULTS

In Section 7.1 $1st$ stage design results from the three examples in Chapter 6 are presented together with the corresponding Net Present Value and Value of Stochastic Solution. The Section also contains results from running the program in Listing 6.1 without a capital expense budget constraint. Next, Section 7.2 presents a limited number of results related to the performance of the two algorithms Full Space and NGBD.

7.1 Optimal design of energy system

The NGBD and Full-Space algorithm produced the same results for both the Expected Value and Stochastic Problem. Consequently, results from the Full-Space algorithm are omitted. Sections 7.1.1, 7.1.2 and 7.1.3 presents the design from the Expected Value Problem and the Stochastic Problem for all three case studies as well as the Net Present Value and the Value of Stochastic Solution. In addition to verifying the framework, the results from the examples illustrate the value of stochastic programming.

7.1.1 Example 1

The optimal design for the energy system in Section 6.1 is listed in Table 7.1. Table 7.2 lists the resulting Net Present Value and the Value of Stochastic Solution. The program in Listing 6.1 was run without the budget constraint on capital expenses to see whether it was the lack of measures for flexibility or maximization of the budget that resulted in the identical nominal and stochastic design in Table 7.1.

Table 7.1: First stage design results from Example 1 in Chapter 6 solved with the NGBD algorithm. N/A indicates that the conversion technology was not included in the user input model. SP and EVP indicates the solution to the Stochastic and Expected Value Problem, respectively.

Example 1					
No. scenarios		W/budget 12		W/O budget 12	
Variable	Unit	SP	EVP	SP	EVP
Solar PV	m^2	N/A	N/A	N/A	N/A
Solar TP	m^2	N/A	N/A	N/A	N/A
Wind turbines	no. turbines	16	16	18	16

Table 7.2: The Net Present Value (NPV) from the stochastic problem, and the Value of Stochastic Solution (VSS) of the renewable energy system in Example 1 with and without the budget constraint.

Example 1			
Value	Unit	W/budget	W/O budget
NPV	\$	153 975	205 063
VSS	\$	0	51 088

For the simple system illustrated in Figure 6.2 the nominal and stochastic optimal designs are identical. The practical implication of the result is that it is indifferent whether the mean wind speed or a 12-scenario profile with an hourly resolution is used in the modeling and selection of the energy system design. However, a zero VSS for a system with only one energy source, i.e., zero measures for flexibility, is expected. Moreover, the capital expense budget is maximized with the installment of 16 wind turbines. Thus, it could be that both the EVP and SP are constrained financially. Results from Table 7.1 show that without a budget constraint, the nominal design is unchanged, whereas the stochastic design maximizes the number of wind turbines installed. The resulting positive VSS underscores the value of stochastic programming even for small energy systems with zero measures for flexibility.

7.1.2 Example 2

The optimal design for the energy system in Section 6.2 is listed in Table 7.3. Resulting Net Present Value and the Value of Stochastic Solution is listed in Table 7.4.

Table 7.3: First stage design results from Example 2 in Chapter 6 solved with the NGBD algorithm. N/A indicates that the conversion technology was not included in the user input model. SP and EVP indicate the solution to the Stochastic and Expected Value Problem, respectively.

Example 2			
No. scenarios		12	
Variable	Unit	SP	EVP
Solar PV	m^2	40 000	72 000
Solar TP	m^2	N/A	N/A
Wind turbines	no. turbines	12	8

Table 7.4: The Net Present Value (NPV) from the stochastic problem, and the Value of Stochastic Solution (VSS) of the renewable energy system in Example 2.

Example 2		
Value	Unit	
NPV	\$	370 840
VSS	\$	58 389

The results in Table 7.3 correspond with the results from the model in [17] without implementation of batteries. Specifically, both models output 40 000 m^2 of solar PV panels and 12 wind turbines as the optimal $1st$ stage design variables for the system illustrated in Figure 6.3.

7.1.3 Example 3

The optimal design for the energy system in Section 6.3 is listed in Table 7.5. Resulting Net Present Value and the Value of Stochastic Solution is listed in Table 7.6.

Table 7.5: First stage design results from Example 3 in Chapter 6 solved with the NGBD algorithm. SP and EVP indicates the solution to the Stochastic and Expected Value Problem, respectively.

Example 3			
No. scenarios		12	
Variable	Unit	SP	EVP
Solar PV	m^2	24 000	48 000
Solar TP	m^2	72 000	64 000
Wind turbines	no. turbines	8	6

Table 7.6: The Net Present Value (NPV) from the stochastic problem and the Value of Stochastic Solution (VSS) of the renewable energy system in Example 3.

Example 3		
Value	Unit	
NPV	$	672 490
VSS	$	81 323

The positive value of the stochastic solution (VSS) for Example 2 and 3 implies that accounting for uncertainty in the design of the systems in Figure 6.4 and 6.6 increases the expected profitability. In addition, the VSS increases with the expansion of the system in Example 2 to the system in Example 3. The added value with an increase in system flexibility, and accompanied associated uncertainty, substantiates the importance of including uncertainty in formulating the flexible design problem. It suggests that the value of stochastic programming improves with increased related uncertainty. In other words, the related uncertainty should be significant.

An approach to indicate which parameters have a substantial impact on the solution, and value of stochastic programming, is to execute sensitivity analyses for all relevant and potentially uncertain parameters. Sensitivity analyses could indicate the parameters that can be set constant at their mean value and which parameters to model as uncertain.

7.2 Algorithmic results

The three examples in Chapter 6 were run with both the NGBD and Full Space algorithm. How to select and switch between algorithms in GOSSIP is explained in the documentation [8]. Output from the solvers is listed in Table 7.7. Complicating variables are the first stage decision variables, e.g, variables that must be set before the realization of uncertainties. Contrary to what was mentioned in Chapter 3 the results show that there are minor differences between the run-times of the NGBD and Full Space algorithm despite an increase in problem size (variables and constraints). The problems are too small to give a meaningful difference in solution times. Moreover, the scope of this thesis was on efficient *modeling* of two-stage stochastic programs for RES and not on testing out different solution strategies.

Table 7.7: Solver output from the NGBD and Full-Space algorithm in the three examples in Chapter 6. Number of variables and constraints are after pre-processing.
*(The NGBD algorithm reduces to BD for MILP's as described in Chapter 3.)

	Example 1		Example 2		Example 3	
Algorithm	NGBD$^{(*)}$	Full Space	NGBD$^{(*)}$	Full Space	NGBD$^{(*)}$	Full Space
No. scenarios	12	12	12	12	12	12
No. time-steps	24	24	24	24	24	24
No. variables	1186	1186	1496	1496	2670	2670
No. complicating variables	10	10	20	20	30	30
No. constraints	889	889	1202	1202	2079	2079
Total solver time (s)	0.03	0.05	0.08	0.11	0.2	0.15
Total time (s)	1.87	1.87	1.95	1.98	2.31	2.23

CHAPTER

EIGHT

DISCUSSION

The object-oriented framework aims to simplify the formulation of a two-stage stochastic program for the optimization of flexible renewable energy systems. More specifically, the framework is developed as a user-friendly interface to the stochastic programming software GOSSIP. A user-friendly program should be easy to learn, understand and use. However, due to the relatively complex nature of stochastic programming, it is expected that any user applying the framework and GOSSIP software is familiar with or can understand basic concepts and methods from optimization. Specifically, this entails a basic understanding concerning the order of declaration of $1st$ and $2nd$ stage variables and constraints in a stochastic MILP and how to ensure consistency in the problem formulation. In addition, the user should be familiar with object-oriented programming and the C++ language.

Framework correctness

It is difficult to validate the numerical results in Section 7.1.1, 7.1.2 and 7.1.3 as there is no accurate answer when uncertain parameters are involved. The scenario profiles are assumed to be one of the most prominent sources of inaccuracy as yearly weather profiles (wind speed and solar radiation) are geo-specific and prone to large deviations from one year to another. Consequently, the results give an approximate estimate of how much energy a renewable conversion technology could recover. Furthermore, it is challenging to isolate simple input-output relationships in renewable energy conversion models such as wind turbines and solar PV panels. The direction of the wind and the angle of the solar radiation affects the actual output of the turbines and panels, respectively. As a result, the (affine) linear model structure in the `Conversion` class is a simplification of reality. However, it should be noted that detailed energy models are beyond the scope of this thesis and that the developed framework is a helpful tool in deciding whether it is profitable or not to install a certain renewable conversion technology.

The correctness of the framework is verified by constructing a program in the OOP framework identical to the model in [17] without the implementation of batteries. The results from [17] is used as a benchmark for the $1st$ stage decision variables from Example 2 in Section 6.2. $1st$ stage decision variables from Example 2 in Section 6.2 listed in Table 7.3 give the same energy system design as the model in [17]. The identical designs indicate that the framework implementation is correct and ensures consistency between the user input model and the resulting problem formulation interfaced to GOSSIP.

Extensibility of framework

One of the virtues of an OOP framework for the optimization of flexible RES is its extensibility. From the user perspective it should be easy to change the model objective, and expand the system by adding system components.

Expanding the system in Example 1 (Section 6.1) to the system in Example 2 (Section 6.2) requires exactly 8 lines of code, increasing the size of the user input model from 22 lines to 30 lines as shown in the resulting Listing 6.2. The extra lines of code specify `primarySource`, cost and conversion function, as well as the size and utility type of the added conversion technology. In other words, the client does not have to re-write any code, only add new connections resulting from the added conversion technology. Contrary to the small change in user input model size, the results in Table 7.7 show that the program complexity grows through a substantial increase in variables and constraints.

To create the system in Example 3 (Section 6.3), the client has to add one more conversion technology, utility type, and end user. Once again, the client does not have to re-write the input model, only add 13 lines of code. An addition of 13 lines is a relatively trivial amount compared to the actual change in problem size. The total program interfaced to GOSSIP is almost doubled in size as it grows with 2051 variables and constraints, as can be seen from the results in Table 7.7. The large increase in variables and constraints despite small changes to the user input model results from the encapsulation and abstraction that defines an object-oriented program. OOP pillar one, encapsulation, is achieved by grouping relevant attributes, energy production export and import variables, and constraints in objects. The encapsulation enables the programmer to change class implementations without interfering with the user input model. This is an illustration of OOP pillar two, abstraction of the program.

An effect of the encapsulation and abstraction is that isolating which lines of code that are essential to which object can be challenging for the client. One function call can involve two objects from different classes. For instance, the function call `NTNU.addUtilitySource (&Electricity)` on line 44 in Listing 6.3 sets end user object `NTNU` as a potential sink for utility object `Electricity`, and `Electricity` as a potential source to `NTNU`. The upside to this is the aforementioned small addition of code, and the downside is that the client is at risk of over- or under-specifying sources and sinks in the system.

Without necessarily linking the objects to a source or sink, the three examples show that the work related to creating the framework objects can be summarized as follows:

- A renewable energy source (`UncertainParam` or `PrimarySource`) requires minimum 1 line of code.
- A conversion technology requires minimum 5 lines of code.
- An utility requires minimum 3 lines of code.
- An end user requires minimum 5 lines of code.

As expected, the amount of code is proportional to the object's relative position in the framework. With an increase in connections comes an increase in specifications. The uncertain parameter objects only have one output connection, whereas the utility and end user objects can have multiple inputs and output connections.

Adaptability of framework

The advantage of changeable connections and models in the optimization of flexible renewable energy systems was stressed in Section 4.2. The desired framework adaptability and extensibility is achieved through user-defined energy and cost models, and modifiable input-output relationships between objects, respectively. Two of the most fundamental framework properties for model extensibility are the multiple input and output sources the `Utility` and `EndUser` objects can have, and the unlimited number of capital and operating expenses that can be added to the `ObjectiveFunction` object.

Lessons from previous attempts

The final framework structure is the result of testing multiple different back-end structures. Appendix C contains the first attempt at formulating a two-stage stochastic MILP through OOP in GOSSIP. Appendix D contains a second attempt with linked-lists and a specialized data structure named `ESvector(2D)`. Both attempts showed promising results. However, attempt one in Appendix C was too similar to a procedural program with a large number of variable-, constraint- and vector declarations in the user input model. Attempt 2 in Appendix D was an improvement as it had a higher degree of abstraction through the use of linked lists and the `ESvector` data structure. Both frameworks were discarded in favor of the framework presented, but ideas from the attempts should be investigated further.

As can be seen in Listing 6.3 there is currently no automatic connecting structure between objects of the same class. For instance, adding capital and operating expenses is a repetitive process where the client has to call `OBJO.addCAPEX(Conversion* technology)` and `OBJO.addOPEX(EndUser* user)` for each `Conversion` and `endUser` object. One potential solution could be to create a specialized container to group related objects. The `OBJO.addCAPEX(Conversion* technology)` and `OBJO.addOPEX(EndUser* user)` functions would then be called with a pointer to a container instead of an individual object. Objects would then be added by iterating through the container. The `ESvector` class could be used as inspiration for the implementation of this container. Another solution, also proposed in

the framework in Appendix D, could be a singly or doubly linked list structure within the classes to give the objects a notion of other instances from the class. Then the `EnergySystem` and `objectiveFunction` object would have a notion of how many objects exist of each class. However, a linked list would imply implementing specialized functions to remove and add objects such that objects can be added or deleted without breaking other connections.

CHAPTER

NINE

CONCLUSION AND DIRECTIONS FOR FUTURE WORK

Final remarks

Flexible renewable energy systems have been singled out as a promising solution to obtain a larger fraction of renewable energy in the global energy mix. However, investigating which and how much renewable technologies to include in a flexible design is a non-trivial task. An object-oriented framework was developed to simplify the formulation of two-stage stochastic programs to optimize flexible renewable energy system design. The resulting framework works as a user-friendly interface to the two-stage stochastic programming software GOSSIP.

To obtain desirable framework properties such as extendability and scalability of energy models and system, respectively, object-oriented programming was chosen as the preferred programming paradigm. Object-oriented programming enabled the encapsulation of variables and constraints through classes and abstraction in implementing the framework structure. The use of pointers for information flow enabled data structures and values to be transferred between objects without being explicitly declared in the user input model. However, advanced techniques such as inheritance and polymorphism were not implemented in the current version. The resulting framework interface show signs of user-friendliness with simple function calls and zero variable and constraint declarations. Chapters 5 and 6 represents a user documentation and manual with class and user input model descriptions, respectively.

The three different examples of increasing complexity illustrated the framework functionality and verified the resulting problem formulation. The corresponding two-stage stochastic MILP problems were solved using the two algorithms NGBD and Full Space. Results from Example 2 corresponded with results from the model in [17] without implementation of batteries. The identical results verified the two-stage problem formulation and subsequently the framework correctness.

Furthermore, results from Example 2 and 3 illustrated the value of stochastic programming with a positive VSS of $58,389$ and $81,323$ \$, respectively. The system in Example 1, on the other hand, had a VSS of zero. However, without a budget constraint on the capital expenses, the VSS for Example 1 was positive. A positive VSS indicates that stochastic programming is valuable even for small energy systems without measures for flexibility such as multiple energy sources and energy storage technologies.

Overall, the framework shows promising tendencies regarding user friendliness, and produces valid and valuable results for programs related to complex energy systems.

Future work

Inheritance

The fourth OOP-pillar in Section 4.1.2 states that inheritance is one of the main attributes of an object-oriented program. Inheritance has not been implemented in the OOP framework presented in Chapter 5. However, implementation of inheritance could entail recycling of the generic parts of functions like the multiple `setConstraint(...)` and `addImport/Export/Utility(...)` functions instead of having to fully define an almost identical procedure for each class and variable type. Moreover, inheritance could be applied to create more specialized versions of already existing classes. Implementation of inheritance could increase the customization from the user perspective and simultaneously ensure a general problem formulation.

Extension of classes

A flexible design can only be achieved partly with the inclusion of multiple energy sources and conversion technologies. Even though it technically would be possible to produce energy under every circumstance, it is expected that the amount produced would vary. Simultaneously the cost of storage technologies such as batteries is decreasing [16]. However, it is a fairly complicated task to implement an `EnergyStorage` class as it would require constraints linking one time-step to another as well as periodicity constraints [17]. Nevertheless, it is expected that energy storage will play an important role in flexible renewable energy systems and the implementation of an `EnergyStorage` class should be investigated.

Furthermore, in the current framework implementation, the `import` variable with the related import price is part of the `Utility` class. This implementation opposes OOP methodology and thinking. Even though the imported variable is a utility type, creating an `ImportUtility` class should be investigated. This way, there could be different instances of one import utility type. For instance, imported electricity can be produced by both renewable and non-renewable sources, thus have varying emission rates and import prices. Furthermore, an import utility can also be used as an intermediary for another utility, as in Example 3 in Section 6.3 where electricity is used to generate heat. With an `ImportUtility` class the client could specify individual `ImportUtility` objects for the aforementioned applications.

Extension to MINLP formulations

Another shortcoming of the current framework implementation is that the energy models are limited to affine linear functions. Many linear models are evident simplifications of the real world. With the implementation of an `EnergyStorage` class, batteries, with their inherently non-linear models [12], would be a highly relevant storage technology to include in a flexible RES.

To represent contemporary and future energy systems, the framework implementation should include a two-stage MINLP formulation. This way, batteries and other technologies with non-linear models can be more accurately represented. It is expected that future implementation of a polynomial surrogate model is manageable as the framework is compatible with the MINLP solvers described in Chapter 3.

BIBLIOGRAPHY

[1] Stephen M Robinson. *Numerical Optimization*. 2006. ISBN: 9780387303031. DOI: 10.1007/978-0-387-40065-5.

[2] Paul I. Barton. "Mixed-Integer and Nonconvex Optimization". In: *Notes* (2007).

[3] Walter Savitch. *Absolute C++*. 4th. USA: Addison-Wesley Publishing Company, 2009. ISBN: 9780136083818.

[4] Ruth Misener and Christodoulos A. Floudas. "ANTIGONE: Algorithms for coNTinuous / Integer Global Optimization of Nonlinear Equations". In: *Journal of Global Optimization* 59.2-3 (2014), pp. 503–526. ISSN: 15732916. DOI: 10.1007/s10898-014-0166-2.

[5] IPCC. *Global Warming of 1.5°C. An IPCC Special Report on the impacts of global warming of 1.5°C above pre-industrial levels and related global greenhouse gas emission pathways, in the context of strengthening the global response to the threat of climate change, sustainable development, and efforts to eradicate poverty*. Tech. rep. Intergovernmental Panel on Climate Change, 2018.

[6] R. Kannan. "Algorithms, analysis and software for the global optimization of two-stage stochastic programs". PhD thesis. Massachusetts Institute of Technology, 2018.

[7] Georgios Mavromatidis, Kristina Orehounig, and Jan Carmeliet. "Design of distributed energy systems under uncertainty: A two-stage stochastic programming approach". In: *Applied Energy* 222.April (2018), pp. 932–950. ISSN: 03062619. DOI: 10.1016/j.apenergy.2018.04.019. URL: https://doi.org/10.1016/j.apenergy.2018.04.019.

[8] P. I. Barton R. Kannan. "GOSSIP documentation". In: (2018).

[9] United Nations. "AFFORDABLE AND CLEAN ENERGY : energy efficient". In: *The sustainable Development Goals Report* (2018). URL: http://www.un.org/sustainabledevelopment/energy/.

[10] M. Karmellos, P. N. Georgiou, and G. Mavrotas. "A comparison of methods for the optimal design of Distributed Energy Systems under uncertainty". In: *Energy* 178 (July 2019), pp. 318–333. ISSN: 03605442. DOI: 10.1016/j.energy.2019.04.153.

[11] R. West and B. Fattouh. *The Energy Transition and Oil Companies' hard choices*. Tech. rep. The Oxford Instiute for Energy Studies, 2019.

[12] Arpit Maheshwari et al. "Optimizing the operation of energy storage using a non-linear lithium-ion battery degradation model". In: *Applied Energy* 261 (2020), p. 114360. ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2019.114360. URL: https://www.sciencedirect.com/science/article/pii/S0306261919320471.

[13] Nordby, Petter Engblom and Rugland, Mari Elise. *Optimization of flexible renewable energy systems using stochastic programming*. 2020.

[14] Oersted. *Our offshore wind capabilitites*. Oct. 2020.

[15] Wikipedia.ord. *Feed-in-Tariff*. June 2020.

[16] David L. Chandler. In: https://news.mit.edu/2021/lithium-ion-battery-costs-0323 (Mar. 2021).

[17] Petter Engblom Nordby. *Optimization of flexible renewable energy systems*. MSc Thesis, Norwegian University of Science, Technology, Department of Energy, and Process Engineering, 2021.

[18] NTNU. URL: https://www.itk.ntnu.no/_media/emner/fordypning/ttk16/introductiontomip2015.pdf.

[19] *The Four Pillars of Object-Oriented Programming*. URL: https://www.freecodecamp.org/news/four-pillars-of-object-oriented-programming/.

APPENDIX

A

SOURCE FILES

A.1 Energy system class

```
#include "EnergySystem.hpp"

namespace decomposition
{

EnergySystem::EnergySystem(int scen, int time) : numScen{scen},
    numTimeSteps{time}
  {
     numUncertainParams=0;
     numSources=0;
     numConverters=0;
     numUtilities=0;
     numUsers=0;

     varcount=-1;
     concount=-1;
  };

void EnergySystem::importProbabilities(string filePath, vector<double
    >& prob)
  {
    //Scenario probabilities from csv to program
      int it_line=0;
    int it_cell=0;
      ifstream file;
      string filepath;
      string cell;
      string line;
    file.open(filePath);
    bool printFile{false};
    if (!file.is_open())
    {
        cout<<"Error opening file for probabilities"<<endl;

```

```
33      }
34      while(getline(file, line))
35      {
36        string comma(",");
37          if(printFile){ cout<<line<<endl;}
38          if(it_line!=0)
39          {
40              stringstream ss(line);
41              while (getline(ss,cell,';'))
42              {
43                  if(it_cell == 1)
44                  {
45                    if(cell.find(comma)!=string::npos)
46                    {
47                        probabilities.push_back(stod(cell.replace(cell.
    find(comma),comma.length(),".")));
48                    }
49                      else
50                      {
51                        probabilities.push_back(stod(cell));
52                      }
53                  }
54                  it_cell++;
55              }
56          }
57          it_line++;
58          it_cell=0;
59
60      }
61      file.close();
62      if (file.is_open())
63      {
64          cout<<"Error closing file for probabilities"<<endl;
65
66      }
67      for(int h=0;h<numScen;++h)
68      {
69        prob.push_back(probabilities[h]);
70      }
71    };
72
73
74  }
```

A.2 Uncertain parameter class

```
#include "UncertainParam.hpp"

namespace decomposition
{

UncertainParam::UncertainParam(string path, EnergySystem* enrg, bool print=false): filePath{path}, linkToSystem{enrg}, numScen{enrg->getNumScen()}, numTimeSteps{enrg->getNumTimeSteps()}
  {
      // initializing
      int t=0;
      int h=0;
      int it_line=0;
      int it_cell=0;
      ifstream file;
      string cell;
      string line;

      bool printFile{print};

      vector<vector<double>> temp(numTimeSteps,vector<double>(numScen));

      //uncertain parameter values from csv to program
      file.open(filePath);
      string comma(",");
      if (!file.is_open())
      {
          cout<<"Error opening file at "<< filePath<<endl;
      }
      while(getline(file, line))
      {
          if (printFile){cout <<line<<endl;}
          if(it_line!=1)
          {
              stringstream ss(line);
              while (getline(ss,cell,';'))
              {

                  if(it_line==0 && it_cell==1)
                  {
                     paramName=cell;
                  }
                  else if (it_line>1 && it_cell==0)
                  {
                      t = stoi(cell);
                  }
                  else if(it_line>1 && it_cell>0 )
                  {
                    if(cell.find(comma)!=string::npos)
                    {
                      temp[t-1][it_cell-1]=stod(cell.replace(cell.find(comma),comma.length(),","));
                    }
                    else
```

```
                    {
                        temp[t-1][it_cell-1] = stod(cell);
                    }
                }
                it_cell++;
            }
        }
        it_line++;
        it_cell=0;
    }
    file.close();
    if (file.is_open())
    {
        cout<<"Error closing file at "<< filePath<<endl;

    };

    output.resize(numTimeSteps);
    for(int t=0;t<numTimeSteps;++t)
    {
      output[t].resize(numScen);
      for(int h=0;h<numScen;++h)
      {
        output[t][h]=temp[t][h];
      };

    };
    cout<<"Imported uncertain parameter values from csv-file "<<
   filePath <<endl;
  };
}
```

A.3 Primary Source class

```
#include "PrimarySource.hpp"

namespace decomposition
{
PrimarySource::PrimarySource(vector<vector<double>>* input,
    EnergySystem* enrg ): numScen{enrg->getNumScen()}, numTimeSteps{
    enrg->getNumTimeSteps()}
  {
    output.resize(numTimeSteps);
    int t=0;
    for(vector<vector<double>>::iterator itRow=input->begin();itRow!=
    input->end();++itRow)
    {
      output[t].resize(numScen);
      int h=0;
      for(vector<double>::iterator itCol=itRow->begin();itCol!=itRow->
    end();++itCol)
      {
        output[t][h]=(*itCol);
        h++;
      }
      t++;
    };

  };
}
```

A.4 Conversion class

```
#include "Conversion.hpp"

namespace decomposition
{

  Conversion::Conversion(vector<vector<double>>* inputSource,
    EnergySystem* enrg): input{inputSource}, linkToSystem{enrg},
    numScen{enrg->getNumScen()}, numTimeSteps{enrg->getNumTimeSteps()}
  {
    output.resize(numTimeSteps);
    produced.resize(numTimeSteps);
    exported.resize(numTimeSteps);

    designVarDisc.resize(numScen);

    for(int t=0;t<numTimeSteps;++t)
    {
      output[t].resize(numScen);
      produced[t].resize(numScen);
      exported[t].resize(numScen);
    }
  };

  void Conversion::setCostFunction(double unitCost, double maxCap, int
     numSizeIntervals,int CRFtime, string type, bool VSS=false, double
     EVPsize=0)
  {
    lifeTime=CRFtime;
    convTech=type;
    char* name=&type[0];
    baseCost=unitCost;
    capN=maxCap;
    CRF=r*pow(1+r,lifeTime)/(pow(1+r,lifeTime)-1);
    cout<<"CRF for "<<type<< " is :"<<CRF<<endl;
    if(!VSS)
    {
      numDiscrete=numSizeIntervals;
      sizes.resize(numDiscrete);
      designVarBin.resize(numDiscrete);

      for(int n=0;n<numDiscrete;++n)
      {
        sizes[n]=n*capN/(numDiscrete-1);
        sprintf(linkToSystem->clabel, "%s[%d]", name, n+1);
        designVarBin[n].setIndependentVariable(linkToSystem->
    addVariable(), compgraph::BINARY, I(0,1),0.,-1,linkToSystem->
    clabel);
      }
    }
    else
    {
      numDiscrete=1;
      sizes.resize(numDiscrete);
      designVarBin.resize(numDiscrete);
      sizes[numDiscrete-1]=EVPsize;
```

```
50      sprintf(linkToSystem->clabel, "%s[%d]", name, numDiscrete);
51      designVarBin[numDiscrete-1].setIndependentVariable(linkToSystem
    ->addVariable(), compgraph::BINARY, I(0,1),0.,-1,linkToSystem->
    clabel);
52    }
53
54
55  };
56
57  void Conversion::setConversionFunction(double constA=1.0, double
    constB=0.0)
58  {
59    a=constA;
60    b=constB;
61    string varName1=convTech+"produced";
62    char* name1=&varName1[0];
63
64    int t=0;
65    for(vector<vector<double>>::iterator itRow=input->begin();itRow!=
    input->end();++itRow)
66    {
67      int h=0;
68      for(vector<double>::iterator itCol=itRow->begin();itCol!=itRow->
    end();++itCol)
69      {
70        sprintf(linkToSystem->clabel,"%s[%d][%d]",name1,t+1,h+1);
71        produced[t][h].setIndependentVariable(linkToSystem->
    addVariable(),compgraph::CONTINUOUS,I(0,1000),0.,h+1,linkToSystem
    ->clabel);
72        if((*itCol)>0)
73        {
74          output[t][h]=a*(*itCol)+b;
75        }
76        else
77        {
78          output[t][h]=0;
79        }
80        cout<<output[t][h]<<endl;
81        h++;
82      }
83      t++;
84    };
85
86    string varName2=convTech+"zDisc";
87    char* name2=&varName2[0];
88    for(int h=0;h<numScen;++h)
89    {
90      sprintf(linkToSystem->clabel,"%s[%d]",name2,h+1);
91      designVarDisc[h].setIndependentVariable(linkToSystem->
    addVariable(),compgraph::CONTINUOUS,I(0,capN),0.,h+1,linkToSystem
    ->clabel);
92    }
93  };
94
95  void Conversion::optimizeCapacity()
96  {
97    designConBin=-1;
```

```
98      designConDisc.resize(numScen);
99
100     for(int n=0;n<numDiscrete;++n)
101     {
102       designConBin+=designVarBin[n];
103     }
104     designConBin.setDependentVariable(linkToSystem->addConstraint(),
        compgraph::EQUALITY,false,-1);
105
106     for(int h=0;h<numScen;++h)
107     {
108       designConDisc[h]=designVarDisc[h];
109       for(int n=0;n<numDiscrete;++n)
110       {
111         designConDisc[h]-=designVarBin[n]*sizes[n];
112       }
113       designConDisc[h].setDependentVariable(linkToSystem->
      addConstraint(),compgraph::EQUALITY,true,h+1);
114     }
115
116     for(int t=0;t<numTimeSteps;++t)
117     {
118       for(int h=0;h<numScen;++h)
119       {
120         exported[t][h]=produced[t][h];
121         exported[t][h]-=output[t][h]*designVarDisc[h];
122         exported[t][h].setDependentVariable(linkToSystem->
      addConstraint(),compgraph::EQUALITY,true,h+1);
123       };
124     }
125   };
126
127   void Conversion::addCAPEX(vector<Objective>& objectiveFunction,
      vector<Constraints>& budget)
128   {
129     for(int h=0;h<numScen;++h)
130     {
131
132       objectiveFunction[h]+=baseCost*designVarDisc[h]*CRF*(1+maintFac)
      ;
133       budget[h]+=baseCost*designVarDisc[h];
134
135     }
136
137   };
138
139 }
```

A.5 Utility class

```
#include "Utility.hpp"

namespace decomposition
{

Utility::Utility(vector<vector<double>>* price,EnergySystem* enrg,
    string utilityName): linkToSystem{enrg}, numScen{enrg->getNumScen
    ()}, numTimeSteps{enrg->getNumTimeSteps()}, name{utilityName}
{
  string varName1=utilityName+"exported";
  string varName2=utilityName+"imported";
  string varname3=utilityName+"surplus";
  char* name1=&varName1[0];
  char* name2=&varName2[0];
  char* name3=&varname3[0];

  exported.resize(numTimeSteps);
  imported.resize(numTimeSteps);
  surplus.resize(numTimeSteps);

  producedCon.resize(numTimeSteps);
  importPrice.resize(numTimeSteps);

  for(int t=0;t<numTimeSteps;++t)
  {
    exported[t].resize(numScen);
    imported[t].resize(numScen);
    surplus[t].resize(numScen);

    producedCon[t].resize(numScen);
    importPrice[t].resize(numScen);

    for(int h=0;h<numScen;++h)
    {
      sprintf(linkToSystem->clabel,"%s[%d][%d]",name1,t+1,h+1);
      exported[t][h].setIndependentVariable(linkToSystem->addVariable
    (),compgraph::CONTINUOUS,I(0,1000),0.,h+1,linkToSystem->clabel);
      sprintf(linkToSystem->clabel,"%s[%d][%d]",name2,t+1,h+1);
      imported[t][h].setIndependentVariable(linkToSystem->addVariable
    (),compgraph::CONTINUOUS,I(0,1000),0.,h+1,linkToSystem->clabel);
      sprintf(linkToSystem->clabel,"%s[%d][%d]",name3,t+1,h+1);
      surplus[t][h].setIndependentVariable(linkToSystem->addVariable()
    ,compgraph::CONTINUOUS,I(0,1000),0.,h+1,linkToSystem->clabel);
    }
  };

  cout<<"Initialized utility object "<<name<<".\n";
  uncertainPrice=true;
  vector<vector<double>>::iterator it1;
  int t=0;
  for(it1=price->begin();it1!=price->end();++it1)
  {
    vector<double>::iterator it2;
    int h=0;
    for(it2=it1->begin();it2!=it1->end();++it2)
```

```
51      {
52        importPrice[t][h]=(*it2);
53        ++h;
54      }
55      ++t;
56    };
57    cout<<"Imported import price for utility "<<utilityName<<".\n";
58
59
60  };
61
62  void Utility::addUtility(vector<vector<Variables>>* input, double
      fraction)
63  {
64    inputSources.push_back(input);
65    inputFractions.push_back(fraction);
66
67    cout<<"Added utility input to "<<name<<".\n";
68    int t=0;
69    for(vector<vector<Variables>>::iterator itRow=input->begin();itRow!=
      input->end();++itRow)
70    {
71      int h=0;
72      for(vector<Variables>::iterator itCol=itRow->begin();itCol!=itRow
      ->end();++itCol)
73      {
74        if(inputSources.size()<=1)
75        {
76          producedCon[t][h]=0;
77          producedCon[t][h]+=(*itCol)*inputFractions[inputFractions.size
      ()-1];
78        }
79        else
80        {
81          producedCon[t][h]+=(*itCol)*inputFractions[inputFractions.size
      ()-1];
82        }
83        h++;
84      }
85      t++;
86    };
87
88  };
89
90  void Utility::setUtilityExportConstraint()
91  {
92    for(int t=0;t<numTimeSteps;++t)
93    {
94      for(int h=0;h<numScen;++h)
95      {
96        producedCon[t][h]-=exported[t][h];
97        producedCon[t][h]-=surplus[t][h];
98        producedCon[t][h].setDependentVariable(linkToSystem->
      addConstraint(),compgraph::EQUALITY,true,h+1);
99      }
100   }
101 };
```

```
102
103 }
```

A.6 End user class

```
#include "EndUser.hpp"


namespace decomposition
{

  EndUser::EndUser(EnergySystem* enrg): linkToSystem{enrg}, numScen{
    enrg->getNumScen()}, numTimeSteps{enrg->getNumTimeSteps()}
  {};

  void EndUser::addUtilityDemand(Utility* utilityType, vector<vector<
    double>>* input)
  {
    map<Utility*,vector<vector<Constraints>>>::iterator it1=demandMap.
    find(utilityType);
    if(it1!=demandMap.end())
    {
      vector<vector<double>> temp(numTimeSteps,vector<double>(numScen)
    );

      int t=0;
      for(vector<vector<double>>::iterator itRow=input->begin();itRow
    !=input->end();++itRow)
      {
        int h=0;
        for(vector<double>::iterator itCol=itRow->begin();itCol!=itRow
    ->end();++itCol)
        {
          temp[t][h]=(*itCol);
          h++;
        }
        t++;
      };
      cout<<"Added utility user demand to exisiting utility type "<<
    utilityType->name<<"."<<endl;
      for(int t=0;t<numTimeSteps;++t)
      {
        for(int h=0;h<numScen;++h)
        {
          it1->second[t][h]+=temp[t][h];
        }
      }
    }
    else
    {
      vector<vector<Constraints>> temp2(numTimeSteps,vector<
    Constraints>(numScen));

      int t=0;
      for(vector<vector<double>>::iterator itRow=input->begin();itRow
    !=input->end();++itRow)
      {
        int h=0;
        for(vector<double>::iterator itCol=itRow->begin();itCol!=itRow
    ->end();++itCol)
```

```
46          {
47            temp2[t][h]=(*itCol);
48            h++;
49          }
50          t++;
51        };
52        importUtilities.push_back(utilityType);
53        demandMap.insert(pair<Utility*,vector<vector<Constraints>>>(
    utilityType,temp2));
54        cout<<"Added user demand and import price for new utility type "
    <<utilityType->name<<"."<<endl;
55    };
56
57
58  }
59
60
61  void EndUser::addUtilitySource(Utility* utilityType)
62  {
63    vector<Utility*>::iterator it=find(inputUtilities.begin(),
    inputUtilities.end(),utilityType);
64    if(it==inputUtilities.end())
65    {
66      inputUtilities.push_back(utilityType);
67      cout<<"Added utility type "<<utilityType->name<<"."<<endl;
68    }
69    else
70    {
71      cout<<"Error, utility already added. Check system.\n";
72
73    };
74  };
75
76  void EndUser::setUtilityImportConstraint(Utility* utilityType)
77  {
78    map<Utility*,vector<vector<Constraints>>>::iterator it1=demandMap.
    find(utilityType);
79    if(it1!=demandMap.end())
80    {
81      cout<<"Found user demand for the requested utilitytype "<<
    utilityType->name<<"."<<endl;
82      vector<Utility*>::iterator it3=find(inputUtilities.begin(),
    inputUtilities.end(),utilityType);
83      vector<Utility*>::iterator it4=find(importUtilities.begin(),
    importUtilities.end(),utilityType);
84      if(it3!=inputUtilities.end() && it4!=importUtilities.end())
85      {
86        for(int t=0;t<numTimeSteps;++t)
87          {
88            for(int h=0;h<numScen;++h)
89            {
90              it1->second[t][h]-=(*it3)->getExport(t,h);
91              it1->second[t][h]-=(*it4)->getImport(t,h);
92            }
93          }
94      }
95      for(int t=0;t<numTimeSteps;++t)
```

```
96        {
97          for(int h=0; h<numScen;++h)
98          {
99            it1->second[t][h].setDependentVariable(linkToSystem->
      addConstraint(),compgraph::EQUALITY,true,h+1);
100         }
101       }
102     }
103     else
104     {
105       cout<<"Could not find any demand constraint for the requested
      utility type "<<utilityType->name<<"."<<endl;
106     };
107   };
108
109
110   void EndUser::getUtilityExport(vector<Objective>& objectiveFunction)
111   {
112     vector<Utility*>::iterator it;
113     for(it=inputUtilities.begin();it!=inputUtilities.end();++it)
114     {
115       double FiT=(*it)->constFiT;
116       double FitExtra=((*it)->surplusFiT)*FiT;
117       for(int t=0;t<numTimeSteps;++t)
118       {
119         for(int h=0;h<numScen;++h)
120         {
121           objectiveFunction[h]-=(*it)->getExport(t,h)*FiT*365;
122           objectiveFunction[h]-=(*it)->getSurplus(t,h)*FitExtra*365;
123         }
124       }
125     }
126
127   };
128
129   void EndUser::getUtilityImport(vector<Objective>& objectiveFunction)
130   {
131     vector<Utility*>::iterator it;
132     for(it=importUtilities.begin();it!=importUtilities.end();++it)
133     {
134       for(int t=0;t<numTimeSteps;++t)
135       {
136         for(int h=0;h<numScen;++h)
137         {
138           objectiveFunction[h]+=(*it)->getImport(t,h)*(*it)->
      importPrice[t][h]*365;
139         }
140       }
141     }
142
143   };
144
145 }
```

A.7 Objective function class

```
#include "ObjectiveFunc.hpp"


namespace decomposition
{
  ObjectiveFunction::ObjectiveFunction(EnergySystem* enrg, double
    setBudget=100000000): numScen{enrg->getNumScen()}, numTimeSteps{
    enrg->getNumTimeSteps()}, linkToSystem{enrg}, budget{setBudget}
  {
    objFunc.resize(numScen);
    budgetCon.resize(numScen);
    for(int h=0;h<numScen;++h)
    {
      objFunc[h]=0;
      budgetCon[h]=-budget;
    };
  };


  void ObjectiveFunction::addOPEX(EndUser* user)
  {
    user->getUtilityExport(objFunc);
    user->getUtilityImport(objFunc);
  };

  void ObjectiveFunction::addCAPEX(Conversion* technology)
  {
    technology->addCAPEX(objFunc, budgetCon);
  };

  void ObjectiveFunction::setObjective()
  {
    for(int h=0;h<numScen;++h)
    {
      objFunc[h].setDependentVariable(linkToSystem->addConstraint(),
    compgraph::OBJ,true,h+1);
      budgetCon[h].setDependentVariable(linkToSystem->addConstraint(),
    compgraph::LEQ,true,h+1);
    };
  };

};
```

APPENDIX

B

SCENARIO PROFILES

B.1 Probabilities

Scenario	Probability
1	0.125
2	0.125
3	0.125
4	0.125
5	0.0625
6	0.0625
7	0.0625
8	0.0625
9	0.0625
10	0.0625
11	0.0625
12	0.0625

B.2 Wind speed

h	1	2	3	4	5	6	7	8	9	10	11	12
t	W	Spr	Sum	Fa	W	Spr	Sum	Fa	W	Spr	Sum	Fa
1	11.31	14.01	4.02	9.02	7.35	9.11	0	5.87	15.27	18.91	5.43	12.18
2	12.17	11.56	5.07	7.7	7.91	7.52	0	5.01	16.43	15.61	6.84	10.4
3	10.47	14.86	5.07	9.71	6.81	9.66	0	6.31	14.14	20.06	6.84	13.11
4	11.31	14.01	5.07	11.86	7.35	9.11	0	7.71	15.27	18.91	6.84	16.01
5	12.17	10.02	6.19	9.71	7.91	6.52	4.02	6.31	16.43	13.53	8.35	13.11
6	11.31	6.51	5.07	9.02	7.35	4.23	0	5.87	15.27	8.79	6.84	12.18
7	10.47	8.56	5.62	11.86	6.81	5.56	0	7.71	14.14	11.55	7.58	16.01
8	12.17	7.86	7.38	9.71	7.91	5.11	4.8	6.31	16.43	10.6	9.97	13.11
9	10.47	5.87	8.01	9.02	6.81	3.82	5.21	5.87	14.14	7.93	10.81	12.18
10	11.31	7.17	9.98	9.02	7.35	4.66	6.49	5.87	15.27	9.68	13.47	12.18
11	11.31	5.87	10.67	11.86	7.35	3.82	6.93	7.71	15.27	7.93	14.4	16.01
12	8.09	6.51	12.09	9.02	5.26	4.23	7.86	5.87	10.93	8.79	16.32	12.18
13	8.09	5.25	12.82	9.71	5.26	3.41	8.33	6.31	10.93	7.09	17.31	13.11
14	10.47	3.54	12.82	11.86	6.81	2.3	8.33	7.71	14.14	4.78	17.31	16.01
15	10.47	7.17	10.67	11.13	6.81	4.66	6.93	7.23	14.14	9.68	14.4	15.02
16	8.09	5.87	9.98	9.71	5.26	3.82	6.49	6.31	10.93	7.93	13.47	13.11
17	6.63	7.17	10.67	9.71	4.31	4.66	6.93	6.31	8.94	9.68	14.4	13.11
18	9.66	5.87	10.67	7.7	6.28	3.82	6.93	5.01	13.04	7.93	14.4	10.4
19	11.31	6.51	5.07	6.46	7.35	4.23	0	4.2	15.27	8.79	6.84	8.72
20	10.47	10.78	4.02	6.46	6.81	7.01	0	4.2	14.14	14.56	5.43	8.72
21	11.31	13.18	4.02	5.28	7.35	8.56	0	0	15.27	17.79	5.43	7.13
22	11.31	13.18	5.62	5.28	7.35	8.56	0	0	15.27	17.79	7.58	7.13
23	10.47	14.01	4.53	5.28	6.81	9.11	0	0	14.14	18.91	6.12	7.13
24	12.17	13.18	4.02	7.7	7.91	8.56	0	5.01	16.43	17.79	5.43	10.4

B.3 Solar radiation

h	1	2	3	4	5	6	7	8	9	10	11	12
t	W	Spr	Sum	Fall	W	Spr	Sum	Fall	W	Spr	Sum	Fall
1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0
6	0	4	11	0	0	5	15	0	0	3	7	0
7	0	25	42	1	0	34	57	1	0	16	27	1
8	5	100	123	25	7	135	166	34	3	65	80	16
9	51	219	243	109	69	296	328	147	33	142	158	71
10	141	357	380	218	190	482	513	294	92	232	247	142
11	211	461	497	312	285	622	671	421	137	300	323	203
12	255	511	569	368	344	690	768	497	166	332	370	239
13	257	520	586	387	347	702	791	522	167	338	381	252
14	230	508	588	354	311	686	794	478	150	330	382	230
15	173	466	558	299	234	629	753	404	112	303	363	194
16	89	388	484	219	120	524	653	296	58	252	315	142
17	24	296	395	136	32	400	533	184	16	192	257	88
18	2	190	281	71	3	257	379	96	1	124	183	46
19	0	92	165	21	0	124	223	28	0	60	107	14
20	0	28	67	2	0	38	90	3	0	18	44	1
21	0	5	20	0	0	7	27	0	0	3	13	0
22	0	0	2	0	0	0	3	0	0	0	1	0
23	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0

B.4 Electricity and heat demand

h	1	2	3	4	5	6	7	8	9	10	11	12
t	W	Spr	Sum	Fall	W	Spr	Sum	Fall	W	Spr	Sum	Fall
1	3.93	3.07	2.53	2.97	3.93	3.07	2.53	2.97	3.93	3.07	2.53	2.97
2	3.86	3	2.49	2.89	3.86	3	2.49	2.89	3.86	3	2.49	2.89
3	3.78	2.93	2.45	2.83	3.78	2.93	2.45	2.83	3.78	2.93	2.45	2.83
4	3.7	2.91	2.48	2.86	3.7	2.91	2.48	2.86	3.7	2.91	2.48	2.86
5	3.58	3.17	2.81	3.16	3.58	3.17	2.81	3.16	3.58	3.17	2.81	3.16
6	3.61	3.66	3.32	3.68	3.61	3.66	3.32	3.68	3.61	3.66	3.32	3.68
7	3.96	4	3.68	4.04	3.96	4	3.68	4.04	3.96	4	3.68	4.04
8	4.41	4.19	3.89	4.22	4.41	4.19	3.89	4.22	4.41	4.19	3.89	4.22
9	4.68	4.28	3.97	4.33	4.68	4.28	3.97	4.33	4.68	4.28	3.97	4.33
10	4.93	4.31	4.02	4.37	4.93	4.31	4.02	4.37	4.93	4.31	4.02	4.37
11	5.04	4.31	4.03	4.39	5.04	4.31	4.03	4.39	5.04	4.31	4.03	4.39
12	5.09	4.25	3.97	4.34	5.09	4.25	3.97	4.34	5.09	4.25	3.97	4.34
13	5.11	4.19	3.91	4.29	5.11	4.19	3.91	4.29	5.11	4.19	3.91	4.29
14	5.06	4.14	3.88	4.26	5.06	4.14	3.88	4.26	5.06	4.14	3.88	4.26
15	5.03	4.18	3.94	4.35	5.03	4.18	3.94	4.35	5.03	4.18	3.94	4.35
16	5.04	4.26	3.97	4.57	5.04	4.26	3.97	4.57	5.04	4.26	3.97	4.57
17	5.31	4.25	3.85	4.72	5.31	4.25	3.85	4.72	5.31	4.25	3.85	4.72
18	5.58	4.29	3.72	4.68	5.58	4.29	3.72	4.68	5.58	4.29	3.72	4.68
19	5.52	4.29	3.63	4.47	5.52	4.29	3.63	4.47	5.52	4.29	3.63	4.47
20	5.29	4.15	3.6	4.18	5.29	4.15	3.6	4.18	5.29	4.15	3.6	4.18
21	5.01	3.82	3.42	3.8	5.01	3.82	3.42	3.8	5.01	3.82	3.42	3.8
22	4.66	3.41	3.03	3.4	4.66	3.41	3.03	3.4	4.66	3.41	3.03	3.4
23	4.29	3.09	2.74	3.09	4.29	3.09	2.74	3.09	4.29	3.09	2.74	3.09
24	3.94	3.1	2.62	3.05	3.94	3.1	2.62	3.05	3.94	3.1	2.62	3.05

B.5 Electricity import price

h	1	2	3	4	5	6	7	8	9	10	11	12
t	W	Spr	Sum	Fall	W	Spr	Sum	Fall	W	Spr	Sum	Fall
1	41	27	26	33	41	27	26	33	41	27	26	33
2	39	25	25	31	39	25	25	31	39	25	25	31
3	37	23	23	29	37	23	23	29	37	23	23	29
4	35	22	22	28	35	22	22	28	35	22	22	28
5	36	23	23	29	36	23	23	29	36	23	23	29
6	40	26	26	33	40	26	26	33	40	26	26	33
7	52	40	34	43	52	40	34	43	52	40	34	43
8	66	51	44	54	66	51	44	54	66	51	44	54
9	69	52	47	57	69	52	47	57	69	52	47	57
10	65	49	44	54	65	49	44	54	65	49	44	54
11	61	46	41	51	61	46	41	51	61	46	41	51
12	60	46	40	50	60	46	40	50	60	46	40	50
13	56	43	38	46	56	43	38	46	56	43	38	46
14	54	42	37	45	54	42	37	45	54	42	37	45
15	53	42	36	45	53	42	36	45	53	42	36	45
16	54	43	37	45	54	43	37	45	54	43	37	45
17	56	44	39	47	56	44	39	47	56	44	39	47
18	64	50	44	53	64	50	44	53	64	50	44	53
19	70	54	47	58	70	54	47	58	70	54	47	58
20	71	55	47	58	71	55	47	58	71	55	47	58
21	64	50	42	52	64	50	42	52	64	50	42	52
22	57	44	38	46	57	44	38	46	57	44	38	46
23	52	40	35	43	52	40	35	43	52	40	35	43
24	44	33	29	36	44	33	29	36	44	33	29	36

APPENDIX
C

ITERATIVE ATTEMPT 1

Ensure OOP-formulation compatible with GOSSIP

C.1 Main program

```
#include <iostream>
#include <string>
#include <vector>
#include "definitions.hpp"
#include "CompGraph.hpp"
#include "GenerateScenarios.hpp"
#include "inputmodel.hpp"
#include "RES_classes.hpp"

using namespace std;



int inputmodel(vector<double> &weights)
{
  /*
  1. initalize scenarios and realisations
  2. initialize 1st stage choices
  3. initalize 2nd stage choices
  4. initialize constraints
  5. initialize objective function
  */
  cout<<"OOP optimization of flexible RES version three"<<endl;
  int num_scen=0;
  int num_design_days=0;

  vector<double> means{100,15.0}; //mean values demand and grid price
    OCgrid
  vector<double> std_devs{0.1*100,0.1*15}; //std deviation demand and
    grid price OCgrid
```

```
29  vector<int> num_reals{3,3}; //number of realisations demand and grid
       price OCgrid
30
31  vector<Energy_source*> sources; //vector of pointers to all energy
      sources, this to create one scenario-tree
32
33  Energy_source solar_radiance(0.00015,0.000015,3,sources);
34  Energy_source wind_speed(7.5,0.75,3,sources);
35
36  Scenario scen(sources,means,std_devs,num_reals); //merge all
      uncertain params, sources of energy as well as demand and oc_gric
37  vector<double> scenarios=scen.initiate_scenarios(weights,num_scen);
      //initates scenarios from automatically generate scenarios GOSSIP
      function
38
39  for(int n=0;n<num_scen;++n)
40  {
41    cout<<scenarios[n]<<"-----"<<scenarios[n+num_scen]<<"----"<<
      scenarios[n+num_scen*2]<<"----"<<scenarios[n+num_scen*3]<<endl;
42  }
43
44  vector<Energy_convert*> conversion;
45
46  Energy_convert solar_PV(true,false,121,conversion); //create solar
      PV panels with 10 discrete size and cost intervals
47  Energy_convert wind_turbine(false,true,121,conversion); //create
      wind turbine with 10 discrete size and cost intervals
48
49  int varcount = -1;
50  int concount=-1;
51
52  vector<Variables> design;//vector to store 1st stage design decision
       variables
53
54  solar_PV.set_design_decision(design, varcount);
55  wind_turbine.set_design_decision(design, varcount);
56
57
58  vector<Variables> energy_import;
59  vector<Variables> energy_export;
60
61  Energy_balance balance(energy_import,energy_export, num_scen,
      varcount);
62
63  vector<Constraints> design_lim;
64  vector<Constraints> produced;
65  vector<Constraints> demand;
66
67  solar_PV.set_design_constraint(design_lim,design,concount);
68  wind_turbine.set_design_constraint(design_lim,design,concount);
69
70  balance.get_production(produced, demand, num_scen, scenarios,
      conversion, design, energy_import, energy_export, concount);
71
72  //Declaring objective function
73  vector<Objective> minimize_cost(num_scen);//[$]
```

```
  Cost cost_func(minimize_cost, design, energy_import,energy_export,
    num_scen, scenarios,conversion, concount);

  cout<<"End of program"<<endl;
  return num_scen;
};
```

C.2 Inclusion file headers

```
#pragma once
#include <iostream>
#include <string>
#include <vector>
#include "definitions.hpp"
#include "CompGraph.hpp"
#include "GenerateScenarios.hpp"
#include "inputmodel.hpp"
#include "Energy_source.hpp"
#include "Scenario.hpp"
#include "Energy_conversion.hpp"
#include "Scenario.hpp"
#include "Energy_balance.hpp"
#include "Cost.hpp"

using namespace std;
```

C.3 Energy source class

```
#pragma once

#include "RES_classes.hpp"

//Energy source class

using namespace std;

class Energy_source
{
private:
  //Nominal power model params
  double const Inom{0.00015}; //[MW/M ] solar radiance intensity
  double const Wnom{7.5}; //[m/s] wind speed
public:
  double mean, std_dev;
  int num_realisations;

  Energy_source(double m, double s, int num, vector<Energy_source*> &
    src): mean{m}, std_dev{s}, num_realisations{num}{
    src.push_back(this);
  };
  Energy_source(): mean{0}, std_dev{0}, num_realisations{0}{};

};
```

C.4 Energy conversion class

```
#pragma once

#include "RES_classes.hpp"


//Energy conversion class
class Energy_convert
{
private:
  //Technology cost parameteres ={PV,WT}
  vector<double> C0={27000,49840000}; //$
  vector<int> Smax={1200000,120}; //Area PV and number of wind
    turbines;
  vector<int> S0={180,1}; //Reference value for cost function
  vector<double> xi={0.05,0.05}; //Maintenance cost (factor)
  vector<double> sfi={0.7,0.7}; //Cost scaling factor

public:
  bool solar, wind;
  int num_discrete;

  vector<double> sizes;//list of discrete sizes 1st stage var
  vector<double> cost;//list of discrete associated cost 1st stage var

  Energy_convert(bool s, bool w, int num, vector<Energy_convert*> &
    conv): solar{s}, wind{w}, num_discrete{num}
  {
    sizes.resize(num);
    cost.resize(num);
    if(s)
    {
      cout<<"Solar panel sizes and costs: \n";
      for(int n=0;n<num;++n)
      {
        int i=0;
        this->sizes[n]=n*Smax[i]/(num-1);
        this->cost[n]=(1+xi[i])*C0[i]*pow(this->sizes[n]/S0[i],sfi[i])
    ;
        cout<<this->sizes[n]<<"--"<<this->cost[n]<<endl;
      }
      cout<<"Constructed solar panel \n";
    }
    if(w)
    {
      cout<<"Wind turbine sizes and costs: \n";
      for(int n=0;n<num;++n)
      {
        int i=1;
        this->sizes[n]=n*Smax[i]/(num-1);
        this->cost[n]=(1+xi[i])*C0[i]*pow(this->sizes[n]/S0[i],sfi[i])
    ;
        cout<<this->sizes[n]<<"--"<<this->cost[n]<<endl;
      }
      cout<<"Constructed wind turbine\n";
    }
```

```
    conv.push_back(this);

  };

  void set_design_decision(vector<Variables> &z,int &varcount)
  {
    char clabel[30];
    z.resize(z.size()+num_discrete);
    for(int n=(z.size()-num_discrete);n<z.size();++n)
    {
      sprintf(clabel,"z[%d]",n+1);
          z[n].setIndependentVariable(++varcount,compgraph::BINARY,I
    (0,1),0.,-1,clabel);  //first stage variable, does not belong to
    specific scenario
    }
  };

  void set_design_constraint(vector <Constraints> &z_lim, const vector
    <Variables>& z, int& concount)
  {
    z_lim.resize(z_lim.size()+1);
    z_lim[z_lim.size()-1]=-1;
      for(int n=(z_lim.size()-1)*num_discrete; n<(num_discrete*z_lim.
    size());++n)
      {
        z_lim[z_lim.size()-1]+=z[n];
      }
    z_lim[z_lim.size()-1].setDependentVariable(++concount,compgraph::
    EQUALITY,false,-1);
  };
};
```

C.5 Scenario class

```
#pragma once

#include "RES_classes.hpp"

//Scenario class

class Scenario
{
public:
  vector<double> mean_vector, std_dev_vector;
  vector<int> num_realisations_vector;
  vector<double> realisations;

  Scenario(vector<Energy_source*> energy_sources, vector<double>
    mean_params,vector<double> std_dev_params, vector<int>
    num_realisations_params,bool seasonal=false)
  {
    mean_vector.resize(mean_params.size()+energy_sources.size());
    std_dev_vector.resize(std_dev_params.size()+energy_sources.size())
    ;
    num_realisations_vector.resize(num_realisations_params.size()+
    energy_sources.size());
```

```
19
20      for(int i=0;i<mean_params.size();++i)
21      {
22        mean_vector[i]=mean_params[i];
23        std_dev_vector[i]=std_dev_params[i];
24        num_realisations_vector[i]=num_realisations_params[i];
25      }
26      for(int j=mean_params.size();j<mean_vector.size();++j)
27      {
28        int i=j-mean_params.size();
29        mean_vector[j]=energy_sources[i]->mean;
30        std_dev_vector[j]=energy_sources[i]->std_dev;
31        num_realisations_vector[j]=energy_sources[i]->num_realisations;
32      }
33    };
34
35    vector<double> initiate_scenarios(vector<double> &weights, int&
        num_scen)
36    {
37
38      num_scen=decomposition::generateScenarios(decomposition::NORMAL,
        this->mean_vector.size(),this->num_realisations_vector,this->
        mean_vector,this->std_dev_vector, weights,this->realisations);//
        need weights to be taken in from  inputmodel
39      return realisations;
40    };
41
42    void add_seasonal_variation(vecto<double>& realisations, int&
        num_scen, , int& num_design_days, double seasonal_sun{0.2}, double
         seasonal_wind{0.2})
43    {
44      num_design_days=4; //spring, summer, fall, winter
45      struct season
46      {
47        double sun, wind;
48        int count;
49      };
50      struct season spring{}
51
52
53    };
54
55    Scenario(){};
56
57  };
```

C.6 Energy balance class

```
1  #pragma once
2
3  #include "RES_classes.hpp"
4
5
6  //Energy balance
7
8  class Energy_balance
```

```
{
private:
  //Wind power model params
  double Wmin=3.5; // [m/s]
  double Wmax=25;
  double Wd=13;
  double qd=8; //[MW] for one wind turbine

  //Efficiencies
  double etaPV=0.15;//[-]
  double etaWT=0.85;
public:

  Energy_balance(vector<Variables>& imp, vector<Variables>& exp, const
     int& num_scen, int& varcount)
  {
    char clabel[30];
    imp.resize(num_scen);
    exp.resize(num_scen);
    for(int s=0;s<num_scen;++s)
    {
      sprintf(clabel,"f_import[%d]",s+1);
      imp[s].setIndependentVariable(++varcount,compgraph::CONTINUOUS,I
    (0,10000),0.,s+1,clabel);
      sprintf(clabel,"f_export[%d]",s+1);
      exp[s].setIndependentVariable(++varcount,compgraph::CONTINUOUS,I
    (0,100000),0.,s+1,clabel);
    }
  };

  Energy_balance(){};

  void get_production(vector<Constraints> &produced,
            vector<Constraints> &demand,
            const int& num_scen,
            const vector<double>& scenarios,
            vector<Energy_convert*> &conv,
            vector<Variables>& z,
            vector<Variables>& imp,
            vector<Variables>& exp,
            int& concount)
  {
    produced.resize(num_scen);
    demand.resize(num_scen);
    for(int s=0;s<num_scen;++s)
    {
      produced[s]=0;
      for(int i=0;i<conv.size();++i)
      {
        for(int n=0;n<conv[i]->num_discrete;++n)
        {
          int j=n+(conv[i]->num_discrete)*i;
          if(conv[i]->solar==true)
          {
            produced[s]+=this->etaPV*scenarios[s+num_scen*2]*conv[i]->
    sizes[n]*z[j]*5; //approx 5 hours of sun each day
          }
```

```
62            if(conv[i]->wind==true)
63            {
64              if(scenarios[s+num_scen*3]>this->Wmin && scenarios[s+
    num_scen*3]<this->Wd)
65              {
66                produced[s]+=this->etaWT*this->qd*((pow(scenarios[s+
    num_scen*3],3)-pow(this->Wmin,3))/(pow(this->Wd,3))-pow(this->Wmin
    ,3))*conv[i]->sizes[n]*z[j]*24;
67              }
68              if(scenarios[s+num_scen*3]>=this->Wd && scenarios[s+
    num_scen*3]<=this->Wmax)
69              {
70                produced[s]+=this->etaWT*this->qd*scenarios[s+num_scen
    *3]*conv[i]->sizes[n]*z[j]*24;
71              }
72              else //if(scenarios[s+num_scen*3]<this->Wmin || scenarios[
    s+num_scen*3]>this->Wmax)
73              {
74                produced[s]+=0;
75              }
76            }
77          }
78        }
79        produced[s]-=exp[s];
80        produced[s].setDependentVariable(++concount,compgraph::EQUALITY,
    true,s+1);
81        demand[s]=scenarios[s]-exp[s]-imp[s];
82        demand[s].setDependentVariable(++concount,compgraph::LEQ,true,s
    +1);
83      }
84    };
85  };
```

C.7 Cost and objective class

```
1   #pragma once
2
3   #include "RES_classes.hpp"
4
5   //Cost class, where objective function is set
6   class Cost
7   {
8     public:
9       //Grid parameters
10      double const FiT_nom=1.45; //feed-in tariff [$/MWh]
11
12    Cost(vector<Objective>& obj,
13      vector<Variables>& z,
14      vector<Variables>& imp,
15      vector<Variables>& exp,
16      const int& num_scen, const vector<double>& scen,
17      vector<Energy_convert*> conv,
18      int& concount)
19      {
20        for(int s=0;s<num_scen;++s)
21        {
```

```
22          obj[s]=0;
23          for(int i=0;i<conv.size();++i)
24          {
25            for(int n=0;n<(conv[i]->num_discrete);++n)
26            {
27              int j=n+i*conv[i]->num_discrete;
28              obj[s]+=conv[i]->cost[n]*z[j];
29            }
30          }
31          obj[s]+=imp[s]*scen[s+num_scen]*365*30; //[$]
32          obj[s]-=exp[s]*FiT_nom*365*30; //[$]
33          obj[s].setDependentVariable(++concount,compgraph::OBJ,true,s
      +1);
34        }
35      };
36
37    Cost(){};
38
39  };
```

APPENDIX

D

ITERATIVE ATTEMPT 2

OOP-formulation with linked lists and ESvector class

D.1 Main program

```
#include "RES_classes_2.hpp"

#include "inputmodel.hpp"

using namespace std;

/*
1. initalize scenarios and realisations
2. initialize 1st stage choices
3. initalize 2nd stage choices
4. initialize constraints
5. initialize objective function
*/

int inputmodel(vector<double> &weights)
{

  Scenario scenarios("Examples/GOSSIP_library/RES_master/Petter/
    UncertaintyData/probabilities.csv","Examples/GOSSIP_library/
    RES_master/Petter/UncertaintyData/", 24, 4,3);
  scenarios.set_probabilities(weights);

  Conversion_technology CT1(&scenarios,"IR_PV","SOLAR");

  Conversion_technology CT2(&scenarios,"W_WT","WIND",CT1.get_pointer()
    );

  CT1.set_cost_function(1000,1,10,50);
  CT2.set_cost_function(1000,1, 10, 50);
  CT1.set_conversion_function('L');
  CT2.set_conversion_function('L');
```

```
29
30    //End_user factory()
31    Energy_system RES(&CT1);
32
33    RES.set_FiT();
34
35    RES.set_demand(&scenarios,"L_DEM");
36    RES.set_import_cost(&scenarios,"OC_GRID");
37    RES.set_objective("COST",30);
38
39
40    return scenarios.num_scen;
41 };
```

D.2 Inclusion file

```
1  #pragma once
2  #include <iostream>
3  #include <cstdio>
4  #include <string>
5  #include <vector>
6  #include <map>
7  #include <fstream>
8  #include <sstream>
9  #include "definitions.hpp"
10 #include "CompGraph.hpp"
11 #include "GenerateScenarios.hpp"
12 #include "inputmodel.hpp"
13 #include "counter.hpp"
14 #include "Scenario.hpp"
15 #include "conversion_technology.hpp"
16 #include "energy_system.hpp"
17 //#include "energy_system.cpp"
18 //#include "ES_vector.hpp"
19 //#include "ES_vector_2D.hpp"
20 #include "end_user.hpp"
21 #include "grid.hpp"
22 //#include "energy_storage.hpp"
23
24
25 using namespace std;
```

D.3 Scenario class

```
#pragma once
#include <iostream>
#include <cstdio>
#include <string>
#include <vector>
#include "RES_classes_2.hpp"

//Scenario class

class Scenario
{
  public:
    int num_t, num_p, num_s;
    int num_scen{0};

    vector<double> probabilities;
    map<string,vector<vector<double>>> uncertain_params;
    bool print_file{false};

    Scenario(const string probabilities_filepath, const string
    user_filepath, int num_time_steps=1, int num_params=0, int
    num_scenarios=0): num_t{num_time_steps}, num_p{num_params}, num_s{
    num_scenarios}
    {
      if(num_time_steps==0)
      {
        cout<<"Invalid time format, specify number of time-steps to be
     larger than 0.\n";
      }
      else
      {
        vector<vector<double>> temp(num_t,vector<double>(num_s));
        string param = "param0";
        string param_name;
          int t=0;
          int it_line=0;   int it_cell=0;
          ifstream file;
          string filepath;
          string cell;
          string line;

          //Scenario probabilities from csv to program
          file.open(probabilities_filepath);
          if (!file.is_open())
          {
            cout<<"Error opening file for probabilities"<<endl;
          }
          while(getline(file, line))
          {
              if(print_file){ cout<<line<<endl;}
              if(it_line!=0)
              {
                  stringstream ss(line);
                  while (getline(ss,cell,';'))
                  {
```

```
                    if(it_cell == 1)
                    {
                        probabilities.push_back(stod(cell));
                        num_scen++;
                    }
                    it_cell++;
                }
            }
            it_line++;
            it_cell=0;

        }
        file.close();
        if (file.is_open())
        {
            cout<<"Error closing file for probabilities"<<endl;
        }
      //uncertain parameter values from csv to program
        for (int i=0;i<num_p;i++)
        {
            it_line =0;
            //opening file
            param.pop_back();
            param+=to_string(i+1);
            filepath = user_filepath+param+".csv";
            file.open(filepath);
            if (!file.is_open())
            {
                cout<<"Error opening file for "<<param<<endl;
            }

            //reading file and updating map
            if (print_file){cout << param <<":"<<endl;}

            while(getline(file, line))
            {
                if (print_file){cout <<line<<endl;}
                if(it_line!=1)
                {
                    stringstream ss(line);
                    while (getline(ss,cell,';'))
                    {

                        if(it_line==0 && it_cell==1)
                        {
                            param_name=cell;
                            uncertain_params.insert(pair<string,
vector<vector<double>>>(param_name,temp));
                        }
                        else if (it_line>1 && it_cell==0)
                        {
                            t = stoi(cell);
                        }
                        else if(it_line>1 && it_cell>0 )
                        {
                            uncertain_params[param_name][t-1][
it_cell-1] = stod(cell);
```

```
                        }
                        it_cell++;
                    }
                }
                it_line++;
                it_cell=0;
            }
            file.close();
            if (file.is_open())
            {
                cout<<"Error closing file for "<< param_name<<endl;
            }
        }

        cout <<"-------printing data---------"<<endl;
        cout<<"probabilities: "<< probabilities.size()<<endl;
        for (int h=0;h<num_s;h++){
            cout << "scen "<< h<<": "<< probabilities[h]<<endl;
        }

        map<string,vector<vector<double>>>::iterator it_map;
        for (it_map = uncertain_params.begin(); it_map !=
uncertain_params.end(); it_map++)
        {
            cout << it_map->first << ":"<<endl;   // string (key)
            for (int h=0;h<num_s;h++)
            {
                cout << " scen: "<<h+1<<endl;
                for (int t=0;t<num_t;t++)
                {
                    cout <<" "<< it_map->second[t][h]<< " ";  //
string's value
                }
                cout << endl;
            }
        }

    }
};

Scenario* get_this()
{
  return this;
};

map<string,vector<vector<double>>>& get_scenarios()
{
  return uncertain_params;
};

vector<vector<double>>* get_param_pointer(string map_key)
{
  return &uncertain_params.at(map_key);
};

void set_probabilities(vector<double>& probability_weights)
{
```

```
162       for(int s=0; s<num_scen;++s)
163       {
164         probability_weights.push_back(probabilities[s]);
165       };
166     };
167
168     int get_num_scen()
169     {
170       return num_scen;
171     };
172 };
```

D.4 Uncertain parameter class

```
#pragma once
#include <iostream>
#include <cstdio>
#include <string>
#include <vector>
#include "std_lib_facilities.h"
#include "Uncertain_param.hpp"
#include "Scenario.hpp"
//#include "RES_classes_2.hpp"

using namespace std;

class Uncertain_param
{
  friend class Scenario;
  vector<vector<vector<vector<double>>>> param_values;
  string name;

  public:
    Uncertain_param(const Scenario& scen, int param_number, string
    param_name): name{param_name}
    {
      param_values.resize(scen.seasons);
      for(int ss=0;ss<scen.seasons;++ss)
      {
        param_values[ss].resize(scen.days);
        for(int d=0;d<scen.days;++d)
        {
          param_values[ss][d].resize(scen.time_steps);
          for(int t=0; t<scen.time_steps;++t)
          {
            param_values[ss][d][t].resize(scen.num_scen);
            for(int s=0;s<scen.num_scen;++s)
            {
              param_values[ss][d][t][s]=scen.realisations[param_number
    -1][ss][d][t][s];
              //cout<<param_values[ss][d][t][s];
            }
          }

        }
      }
      cout<<"Uncertain parameter "<<this->name<< " created. \n";
    };

    Uncertain_param* get_pointer()
    {return this;};

};
```

D.5 ES_vector and ES_vector_2D class

```
#pragma once
#include "RES_classes_2.hpp"
using namespace std;

    class ES_vector
    {
    public:

      int num_discrete;

      vector<Variables> array_var;
      vector<Constraints> array_con;

      Energy_system* system_link{NULL};

      ES_vector(Energy_system* enrg): num_discrete{enrg->CT->n_d},
    system_link{enrg}{};

      virtual void set_variable(string name, double max_val)
      {
        Conversion_technology* temp=system_link->CT;
        char clabel[30];
        while(temp->next!=nullptr)
        {
          int j=array_var.size();
          array_var.resize(array_var.size()+num_discrete);
          for(int i=j;i<(j+num_discrete);++i)
          {
            sprintf(clabel,"%s[%d]",name,i+1);
            array_var[i].setIndependentVariable(system_link->
    varcount_ptr++,compgraph::BINARY, I(0,1),0.,-1,clabel);
          };
        temp=temp->next;
        }

      };

      virtual void set_constraint()
      {
        Conversion_technology* temp=system_link->CT;
        while(temp->next!=nullptr)
        {
          array_con.resize(array_con.size()+1);
          array_con[array_con.size()-1]=-1;
          for(int i=(array_con.size()-1)*num_discrete;i<(num_discrete*
    array_con.size());++i)
          {
            array_con[array_con.size()-1]+=array_var[i];
          };
          array_con[array_con.size()-1].setDependentVariable(
    system_link->concount_ptr++,compgraph::EQUALITY,false,-1);
          temp=temp->next;
        };

      };
```

```
52
53      ES_vector* get_pointer()
54      {
55        return this;
56      };
57
58    };
```

```
1  #pragma once
2  #include "RES_classes_2.hpp"
3  using namespace std;
4
5  class ES_vector_2D : public ES_vector
6  {
7  public:
8
9    int num_s, num_t;
10
11   ES_vector_2D(Energy_system* enrg): ES_vector{enrg}, num_s{enrg->CT->
       s}, num_t{enrg->CT->t}
12   {
13     vector<vector<Variables>> array_var(num_t,array_var{num_s});
14     vector<vector<Constraints>> array_con(num_t,array_con{num_s});
15   };
16
17   void set_variable(string name, double max_val) override
18   {
19     char clabel[30];
20     array_var.resize(enrg->t);
21     for(int i=0;i<enrg->t;++i)
22     {
23       array_var[i].resize(enrg->s);
24       for(int j=0;j<enrg->s;++j)
25       {
26         sprintf(clabel,"%s[%d][%d]",name,i+1,j+1);
27         array_var[i][j].setIndependentVariable(system_link->
       varcount_ptr++,CONTINUOUS, I(0,max_val),0.,j+1,clabel);
28       };
29     };
30
31   };
32
33   void set_constraint() override
34   {
35     array_var.resize(enrg->t);
36     for(int i=0;i<enrg->t;++i)
37     {
38       array_var[i].resize(enrg->s);
39       for(int j=0;j<enrg->s;++j)
40       {
41         array_con[i][j].setDependentVariable(system_link->concount_ptr
       ++,compgraph::EQUALITY,false,j+1);
42       };
43     };
44   };
45
46 };
```

D.6 Energy conversion class

```
#pragma once
#include "RES_classes_2.hpp"
//Energy conversion class

class Conversion_technology
{

private:
  double a, b;

public:
  friend Scenario;
  char function;
  int t,s,n_d;

  Scenario* related_scenario;

  double C0, cap_0, cap_n{100}, econ_scale{0.7}, maint_fac{0.05},
    efficiency{0.99};
  vector<double> sizes;
  vector<double> cost;

  Conversion_technology* next=nullptr;
  Conversion_technology* prev;

  string input_param; //pointer to param value from scenario
  vector<vector<double>> output; //vector with output values energy

  string unit_name, par;

  Conversion_technology(Scenario* scen, string related_param, string
    name="noname", Conversion_technology* link=nullptr): t{scen->num_t
    },s{scen->num_s}, related_scenario{scen},input_param{related_param
    } ,unit_name{name}, par{related_param}
  {
    output.resize(t);
    for(int i=0;i<t;++i)
    {
      output[i].resize(s);
    }
    cout<<"Created conversion technology "<<unit_name<<" with
    efficiency "<<efficiency<<", and associated uncertainty "<<par<<
    endl;

    if(link!=nullptr)
    {
      this->prev=link;
      link->next=this;
      this->next=nullptr;

    }
    else
    {
      this->prev=nullptr;
    }
```

```
};

Conversion_technology* get_pointer()
{
  return this;
};

void set_cost_function(double base_cost, double base_cap, double
  maximum_cap, int num_discrete)
{
  C0=base_cost;
  cap_0=base_cap;
  cap_n=maximum_cap;
  n_d=num_discrete;

  sizes.resize(n_d);
  cost.resize(n_d);
  for(int n=0;n<n_d;++n)
  {
    sizes[n]=n*cap_n/(n_d-1);
    cost[n]=C0*pow(sizes[n]/cap_0,econ_scale);
    cout<<unit_name <<" cost & capacity " <<cost[n]<< ", "<<sizes[n
  ]<<endl;
  }
};

void set_conversion_function(char function_type='L', double const_a
  =1, double const_b=0)
{
  //case/switch statements here
  a=const_a;
  b=const_b;
  map<string,vector<vector<double>>>::iterator it_map;
  for (it_map = related_scenario->uncertain_params.begin(); it_map
  != related_scenario->uncertain_params.end(); it_map++)
      {

          if(it_map->first==input_param)
         {
          cout<<"Import param: "<<it_map->first<<endl;
            for (int i=0;i<s;i++)
            {
                for (int j=0;j<t;j++)
                {
                    output[j][i]=it_map->second[j][i]*a+b;

                }
            }
        }
        else
        {
          cout<< it_map->first <<" is not imported.\n";
        }
      };
  switch(function_type)
  {
  case 'L':
```

```
103         cout<<"Linear energy conversion model for "<<this->unit_name<<".
        \n";
104           break;
105       case 'P':
106         cout<<"Polynomial model not possible atm.\n";
107         break;
108 
109       case 'E':
110         cout<<"Exponential model not possible atm.\n";
111         break;
112       }
113     };
114 
115 
116 };
```

D.7 Energy system class

```
#pragma once
#include "RES_classes_2.hpp"
using namespace std;

class Energy_system
{
public:
  //-------------------------------------------------------------
  class ES_vector
  {
  public:

    int num_discrete;

    vector<Variables> array_var;
    vector<Constraints> array_con;

    Energy_system* system_link;

    ES_vector(){};

    void set_params(Energy_system* enrg)
    {
      system_link=enrg;
      num_discrete=enrg->CT->n_d;

    };

    void set_variable(char name[], double max_val=0)
    {
      Conversion_technology* temp=system_link->CT;
      while(temp!=nullptr)
      {
        int j=array_var.size();
        array_var.resize(array_var.size()+num_discrete);
        for(int i=j;i<(j+num_discrete);++i)
        {
          sprintf(system_link->clabel,"%s[%d]",name,i+1);
          system_link->varcount+=1;
          array_var[i].setIndependentVariable(system_link->varcount,
    compgraph::BINARY, I(0,1),0.,-1,system_link->clabel);
        };
      temp=temp->next;
      }

    };

    void set_constraint()
    {
      Conversion_technology* temp=system_link->CT;
      while(temp!=nullptr)
      {
        array_con.resize(array_con.size()+1);
        array_con[array_con.size()-1]=-1;
        for(int i=(array_con.size()-1)*num_discrete;i<(num_discrete*
```

```
    array_con.size());++i)
        {
          array_con[array_con.size()-1]+=array_var[i];
        };
        system_link->concount+=1;
        array_con[array_con.size()-1].setDependentVariable(system_link
    ->concount,compgraph::EQUALITY,false,-1);
        temp=temp->next;
      };

    };

    ES_vector* get_pointer()
    {
      return this;
    };

  };
  //------------------------------------------------------------

  class ES_vector_2D
  {
  public:

    int num_s, num_t;
    Energy_system* system_link;

    vector<vector<Variables>> array_var;
    vector<vector<Constraints>> array_con;

    ES_vector_2D(){};

    void set_params(Energy_system* enrg)
    {
      num_s=enrg->CT->related_scenario->num_s;
      num_t=enrg->CT->related_scenario->num_t;
      system_link=enrg;
    };

    void set_2D_variable(char name[], double max_val)
    {
      array_var.resize(num_t);
      for(int i=0;i<num_t;++i)
      {
        array_var[i].resize(num_s);
        for(int j=0;j<num_s;++j)
        {
          sprintf(system_link->clabel,"%s[%d][%d]",name,i+1,j+1);
          system_link->varcount+=1;
          array_var[i][j].setIndependentVariable(system_link->varcount
    ,compgraph::CONTINUOUS, I(0,max_val),0.,j+1,system_link->clabel);
        };
      };
    };

    void set_2D_constraint()
    {
```

```
109       array_con.resize(num_t);
110       for(int i=0;i<num_t;++i)
111       {
112         array_con[i].resize(num_s);
113         for(int j=0;j<num_s;++j)
114         {
115           array_con[i][j]=0;
116         };
117       };
118     };
119
120   };
121   //-------------------------------------------------------------
122
123   int num_scen,time,num;
124
125   int varcount=-1;
126   int concount=-1;
127   char clabel[70];
128
129   Conversion_technology* CT;
130
131   double FiT, FiT_extra;
132
133   ES_vector des;
134   ES_vector_2D prod;
135   ES_vector_2D dem;
136
137   vector<vector<double>> price;
138
139   double constant_demand, constant_price, project_lifetime;
140
141   vector<Objective> obj;
142
143   Energy_system(Conversion_technology* conv_link) : num_scen{conv_link
        ->related_scenario->num_s}, time{conv_link->related_scenario->
        num_t}, num{conv_link->n_d}, CT{conv_link}
144   {
145
146     //initialize first stage variables
147     des.set_params(this);
148     prod.set_params(this);
149     dem.set_params(this);
150
151     des.set_variable("z",0.0);
152     //initialize 2nd stage variables
153
154     prod.set_2D_variable("energy_export",1000);
155
156     dem.set_2D_variable("energy_deficit",1000);
157     //initialize constraints
158     des.set_constraint();
159     prod.set_2D_constraint();
160     dem.set_2D_constraint();
161
162     cout<<"Initialized energy system with conversion units: \n";
163
```

```
164     Conversion_technology* temp=CT;
165     while(temp!=nullptr)
166     {
167       cout<<temp->unit_name<<endl;
168       temp=temp->next;
169     };
170
171   };
172
173   void set_demand(Scenario* scen, string related_param="", int
        const_val=0)
174   {
175     Conversion_technology* temp=CT;
176     int count=0;
177     /*cout<<"Size of con:" <<prod.array_con.size() <<" -- " <<prod.
        array_con[0].size()<<endl;
178     cout<<"Size of var "<<prod.array_var.size() << " --"<<prod.
        array_var[0].size()<<endl;
179     cout<<"Size of con:" <<dem.array_con.size() <<" -- " <<dem.
        array_con[0].size()<<endl;
180     cout<<"Size of des: "<<des.array_var.size()<<endl;
181     cout<<"Size of temp: "<<temp->output.size() << "--"<<temp->output
        [0].size()<<endl;
182     cout<<"Size of sizes: "<<temp->sizes.size()<<endl;
183     */
184     while(temp!=nullptr)
185     {
186       //cout<<"Counter: "<<count<<endl;
187       for(int i=0; i<scen->num_t;++i)
188       {
189         for(int j=0;j<scen->num_s;++j)
190         {
191           int k=0;
192           for(int n=count*num;n<(num*(count+1));++n)
193           {
194             //cout << i<<" " << j<<" "<< k<<" "<< n<<endl;
195             prod.array_con[i][j]+=(temp->output[i][j]*temp->sizes[k]*
        des.array_var[n]);
196             k+=1;
197           };
198         };
199       };
200       count+=1;
201       temp=temp->next;
202     };
203
204
205         for (int i=0;i<scen->num_s;i++)
206         {
207           for (int j=0;j<scen->num_t;j++)
208             {
209         map<string,vector<vector<double>>>::iterator it_map;
210         for (it_map =scen->uncertain_params.begin(); it_map !=scen->
        uncertain_params.end(); it_map++)
211           {
212
213               if(it_map->first==related_param)
```

```
214             {
215                     dem.array_con[j][i]-=it_map->second[j][i];
216             };
217           };
218           concount+=1;
219           prod.array_con[j][i]-=prod.array_var[j][i];
220           prod.array_con[j][i].setDependentVariable(concount,compgraph
    ::EQUALITY,false,j+1);
221           concount+=1;
222           dem.array_con[j][i]+=prod.array_var[j][i]+dem.array_var[j][i
    ];
223         dem.array_con[j][i].setDependentVariable(concount,compgraph::
    EQUALITY,false,j+1);
224         };
225     };
226   };
227
228   void set_import_cost(Scenario* scen=nullptr, string related_param=""
    , int const_val=0)
229   {
230     if(scen!=nullptr && related_param!="")
231     {
232       price.resize(time);
233       for(int i=0;i<time;++i)
234         {
235           price[i].resize(num_scen);
236         };
237       map<string,vector<vector<double>>>::iterator it_map;
238       for (it_map =CT->related_scenario->uncertain_params.begin();
    it_map !=CT->related_scenario->uncertain_params.end(); it_map++)
239         {
240             if(it_map->first==related_param)
241           {
242               for (int i=0;i<num_scen;i++)
243               {
244                   for (int j=0;j<time;j++)
245                   {
246                     price[j][i]=it_map->second[j][i];
247                   }
248               }
249           }
250         }
251     }
252     else
253     {
254       constant_price=const_val;
255     };
256
257   };
258
259   void set_FiT(double tariff=150, double tariff_extra=0)
260   {
261     FiT=tariff;
262     if(tariff_extra==0)
263       {
264         FiT_extra=tariff;
265       }
```

```
266       else
267         {
268           FiT_extra=tariff_extra;
269         };
270     };
271
272     void set_objective(string objective="", double lifetime=10)
273     {
274       //initialize objective function
275       project_lifetime=lifetime;
276       obj.resize(num_scen);
277       if(objective=="COST" || "")
278       {
279         for(int i=0;i<num_scen;++i)
280         {
281           obj[i]=0;
282           Conversion_technology* temp=CT;
283           int count=des.array_var.size()/num;
284           int it=0;
285           while(temp!=nullptr)
286           {
287             int k=0;
288             for(int n=it*num;n<des.array_var.size()/count;++n)
289             {
290               obj[i]+=des.array_var[n]*temp->cost[k];
291               k+=1;
292             };
293             temp=temp->next;
294             count-=1;
295             it+=1;
296           }
297           for(int j=0;j<time;++j)
298           {
299             obj[i]-=prod.array_var[j][i]*FiT*project_lifetime*365;
300             obj[i]+=dem.array_var[j][i]*price[j][i]*project_lifetime
      *365;
301
302           };
303           concount+=1;
304           obj[i].setDependentVariable(concount, compgraph::OBJ,true,i+1)
      ;
305         };
306       };
307     };
308
309 };
```

Mari Elise Rugland

An OOP framework for optimization of flexible RES

NTNU
Norwegian University of
Science and Technology