An integrated platform for intuitive mathematical programming modeling using LATEX (#28593)

First submission

Editor guidance

Please submit by **28 Jun 2018** for the benefit of the authors (and your $200 publishing discount).

Structure and Criteria
Please read the 'Structure and Criteria' page for general guidance.

Raw data check
Review the raw data. Download from the materials page.

Image check
Check that figures and images have not been inappropriately manipulated.

Privacy reminder: If uploading an annotated PDF, remove identifiable information to remain anonymous.

Files
Download and review all files from the materials page.

5 Figure file(s)
3 Latex file(s)
1 Raw data file(s)

Structure and Criteria

Structure your review

The review form is divided into 5 sections. Please consider these when composing your review:

1. **BASIC REPORTING**
2. **EXPERIMENTAL DESIGN**
3. **VALIDITY OF THE FINDINGS**
4. General comments
5. Confidential notes to the editor

You can also annotate this PDF and upload it as part of your review

When ready submit online.

Editorial Criteria

Use these criteria points to structure your review. The full detailed editorial criteria is on your guidance page.

BASIC REPORTING

- Clear, unambiguous, professional English language used throughout.
- Intro & background to show context. Literature well referenced & relevant.
- Structure conforms to PeerJ standards, discipline norm, or improved for clarity.
- Figures are relevant, high quality, well labelled & described.
- Raw data supplied (see PeerJ policy).

EXPERIMENTAL DESIGN

- Original primary research within Scope of the journal.
- Research question well defined, relevant & meaningful. It is stated how the research fills an identified knowledge gap.
- Rigorous investigation performed to a high technical & ethical standard.
- Methods described with sufficient detail & information to replicate.

VALIDITY OF THE FINDINGS

- Impact and novelty not assessed. Negative/inconclusive results accepted. *Meaningful* replication encouraged where rationale & benefit to literature is clearly stated.
- Data is robust, statistically sound, & controlled.
- Speculation is welcome, but should be identified as such.
- Conclusions are well stated, linked to original research question & limited to supporting results.

Standout reviewing tips

The best reviewers use these techniques

Tip	*Example*
Support criticisms with evidence from the text or from other sources	*Smith et al (J of Methodology, 2005, V3, pp 123) have shown that the analysis you use in Lines 241-250 is not the most appropriate for this situation. Please explain why you used this method.*
Give specific suggestions on how to improve the manuscript	*Your introduction needs more detail. I suggest that you improve the description at lines 57- 86 to provide more justification for your study (specifically, you should expand upon the knowledge gap being filled).*
Comment on language and grammar issues	*The English language should be improved to ensure that an international audience can clearly understand your text. Some examples where the language could be improved include lines 23, 77, 121, 128 – the current phrasing makes comprehension difficult.*
Organize by importance of the issues, and number your points	*1. Your most important issue* *2. The next most important item* *3. …* *4. The least important points*
Please provide constructive criticism, and avoid personal opinions	*I thank you for providing the raw data, however your supplemental files need more descriptive metadata identifiers to be useful to future readers. Although your results are compelling, the data analysis should be improved in the following ways: AA, BB, CC*
Comment on strengths (as well as weaknesses) of the manuscript	*I commend the authors for their extensive data set, compiled over many years of detailed fieldwork. In addition, the manuscript is clearly written in professional, unambiguous language. If there is a weakness, it is in the statistical analysis (as I have noted above) which should be improved upon before Acceptance.*

An integrated platform for intuitive mathematical programming modeling using LATEX

Charalampos P Triantafyllidis [1] , **Lazaros G. Papageorgiou** [Corresp. 1]

[1] Chemical Engineering, University College London, University of London, London, United Kingdom

Corresponding Author: Lazaros G. Papageorgiou
Email address: l.papageorgiou@ucl.ac.uk

This paper presents a novel prototype platform that uses the same LaTeX mark-up language, commonly used to typeset mathematical content, as an input language for modeling optimization problems of various classes. The platform converts the LaTeX model into a formal Algebraic Modeling Language (AML) representation based on Pyomo through a parsing engine written in Python and solves by either via NEOS server or locally installed solvers, using a friendly Graphical User Interface (GUI). The distinct advantages of our approach can be summarized in i) simplification and speed-up of the model design and development process ii) non-commercial character iii) cross-platform support iv) no limitation on application sector and v) minimization of working knowledge of programming and AMLs to perform mathematical programming modeling. This is the first to the best of our knowledge presentation of a workable scheme on using LaTeX for mathematical programming modeling which assists in furthering our ability to reproduce and replicate scientific work.

An integrated platform for intuitive mathematical programming modeling using LaTeX

Charalampos P. Triantafyllidis[a], Lazaros G. Papageorgiou [a,*]

[a]*Centre for Process Systems Engineering, Department of Chemical Engineering, UCL (University College London), London WC1E 7JE, UK*

Abstract

This paper presents a novel prototype platform that uses the same LaTeX mark-up language, commonly used to typeset mathematical content, as an input language for modeling optimization problems of various classes. The platform converts the LaTeX model into a formal Algebraic Modeling Language (AML) representation based on Pyomo through a parsing engine written in Python and solves by either via NEOS server or locally installed solvers, using a friendly Graphical User Interface (GUI). The distinct advantages of our approach can be summarized in i) simplification and speed-up of the model design and development process ii) non-commercial character iii) cross-platform support iv) no limitation on application sector and v) minimization of working knowledge of programming and AMLs to perform mathematical programming modeling. This is the first to the best of our knowledge presentation of a workable scheme on using LaTeX for mathematical programming modeling which assists in furthering our ability to reproduce and replicate scientific work.

Keywords: LaTeX, Python, Pyomo, Algebraic Modeling Languages, Mathematical Programming; Optimization;

2010 MSC: 90C05, 90C11, 90C90, 97M10, 68T35, 97P30

1. Introduction

Mathematical modeling constitutes a rigorous way of inexpensively simulating complex systems' behavior in order to gain further understanding about the underlying mechanisms

*Corresponding Author

Email addresses: `h.triantafyllidis@ucl.ac.uk` (Charalampos P. Triantafyllidis), `l.papageorgiou@ucl.ac.uk` (Lazaros G. Papageorgiou)

and trade-offs. By exploiting mathematical modeling techniques, one may manipulate the system under analysis so as to guarantee its optimal and robust operation.

The dominant computing tool to assist in modeling is the Algebraic Modeling Languages (AMLs) (Kallrath, 2004). AMLs have been very successful in enabling a transparent development of different types of models, easily distributable among peers and described with clarity, effectiveness and precision. Software suites such as AIMMS (Bisschop and Roelofs, 2011), GAMS IDE (Bruce A. McCarl et. al., 2013), JuMP (Dunning et al., 2017) as the modeling library in Julia (Lubin and Dunning, 2015), Pyomo[1] (Hart et al., 2017, 2011) for modeling in Python[2], (Rossum, 1995) and AMPL (Fourer et al., 1993) are the most popular and widely used in both academia and industry. AMLs usually incorporate the following features:

- a strict and specific syntax for the mathematical notation to describe the models;
- Solver interfaces, the bridge between mathematics and what the solver can *understand* in terms of structural demands;
- a series of available optimization solvers for as many classes of problems as supported (LP, MILP, MINLP etc.) with the associated functional interfaces implemented;
- explicit data file formats and implementation of the respective import/export mechanisms.

AMLs provide a level of abstraction, which is higher than the direct approach of generating a model using directly a programming language. The different levels in the design process of a model are depicted in Figure 1. Extending an AML (or even the entire modeling design process) can be done in the following two ways: we can either simplify the present framework (*vertical abstraction*) or extend the embedded functionality (*horizontal abstraction*) (Jackson, 2012). The layers of abstraction between the conception and the semantics of a mathematical model and its computational implementation may not necessarily be *thin*. This means that

[1] http://www.Pyomo.org/
[2] https://www.python.org/

while eventually the aim of the presented platform has the same purpose as an AML that is to generate and solve models, simplification of the required syntax to describe the model is associated with higher complexity. Thus, in order to relax the syntactical requirements, we have to be able to process the same model with limited information (for instance, we do not declare index sets and parameters in the platform). This limited declaration of model components elevates the amount of processing that the platform has to conduct in order to provide equivalent formulations of the input.

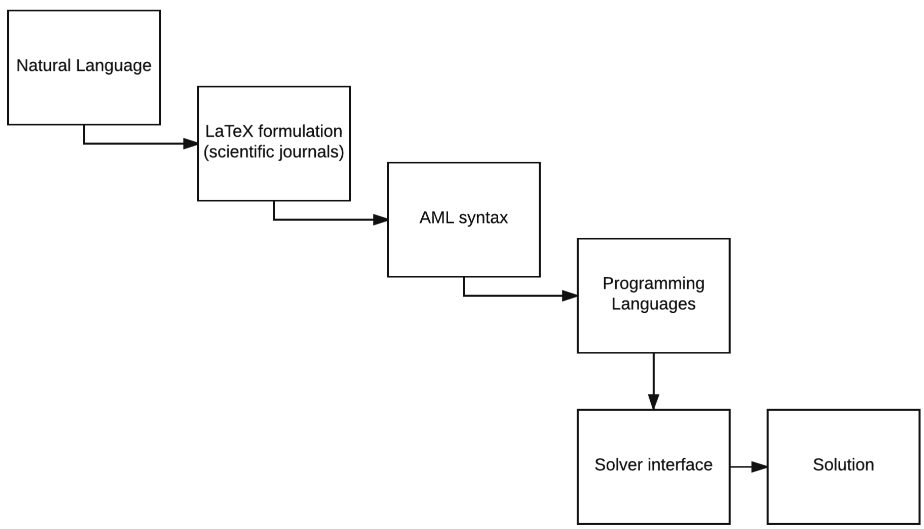

Figure 1: The levels of abstraction in modeling; from natural language to extracting the optimal solution via computational resources.

Our work expands upon two axes : i) the programming paradigm introduced by Donald E. Knuth (Knuth, 1984) on *Literate Programming* and ii) the notions of *reproducible and replicable research*, the fundamental basis of scientific analysis. Literate Programming focuses on generating programs based on logical flow and thinking rather than being limited by the imposing syntactical constraints of a programming language. In essence, we employ a simple mark-up language, LaTeX, to describe a problem (mathematical programming model) and then in turn produce compilable code (Pyomo abstract model) which can be used outside of the presented prototype platform's framework. Reproducibility and the ability to replicate scientific analysis is crucial and challenging to achieve. As software tools become the vessel to unravel the computational complexity of decision-making, developing open-source software is not necessarily sufficient; the ability for the averagely versed developer to reproduce and replicate scientific work is very important to effectively deliver impact (Leek and Peng, 2015; Editorial, 2014). To quote the COIN-OR Foundation [3], *Science evolves when previous results can be easily replicated*.

In the endeavor of simplifying the syntactical requirements imposed by AMLs we have developed a prototype platform. This new framework is materializing a level of modeling design that is higher than the AMLs in terms of *vertical abstraction*. It therefore strengthens the ability to reproduce and replicate optimization models across literature for further analysis by reducing the demands in working knowledge of AMLs or coding. The key capability is that it parses LaTeX formulations of mathematical programs (optimization problems) directly into Pyomo abstract models. The framework then combines the produced abstract model with data provided in the AMPL *.dat* format (containing parameters and sets) to produce a concrete model. This capability is provided through a graphical interface which accepts LaTeX input and AMPL data files, parses a Pyomo model, solves with a selected solver (CPLEX, GLPK, or the NEOS server), and returns the optimal solution if feasible, as the output. The aim is not to substitute ~~AMLS~~ but to establish a link between those using a higher level of abstraction. Therefore, the platform does not eliminate the use of an AML or the advantages emanating from it.

To the best of our knowledge, this is the first prototype workable scheme to address how LaTeX could be used as an input language to perform mathematical programming modeling, and currently supports Linear Programming (LP), Mixed-Integer Linear Programming (MILP) as well as Mixed-Integer Quadratic Programming (MIQP) formulations. Linear Optimization (Bertsimas and Tsitsiklis, 1997; Williams, 1999) has proven to be an invaluable tool for decision support over the past decades. The corpus of models invented for linear optimization over the past decades and for a multitude of domains has been consistently increasing. It can be easily demonstrated with examples in Machine Learning, Supply Chain, Information Security, Environmental Modeling and Energy among many others (Yang et al., 2017, 2016; Tanveer, 2015; Silva et al., 2016; Xu et al., 2007; Grossmann et al., 2016; Papageorgiou and Rotstein, 1998; Jovanović et al., 2016; Sitek and Wikarek, 2015; Triantafyllidis

[3] https://www.coin-or.org/

et al., 2018; Bieber et al., 2018; Wang et al., 2018; Cohen et al., 2017; Mitsos et al., 2009; Melas et al., 2013; Romeijn et al., 2006; Knijnenburg et al., 2016; Kratica et al., 2014; Mouha et al., 2012; Heuberger et al., 2017; Liu and Papageorgiou, 2013, 2017).

This paper is organized as follows: in section 2, we describe the current functionality supported by the platform at this prototype stage. In section 3, we present the implementation details of the parser. Section 4 provides a description of an illustrative example. A discussion follows in section 5. Some concluding remarks are drawn in section 6. Examples of optimization models that were reproduced from scientific papers as well as their corresponding LaTeX formulations and Pyomo models can be found in the Supplementary Information.

2. Functionality

The set of rules that are admissible to formulate models in this platform are formal LaTeX commands and they do not represent in-house modifications. We assume that the model will be in the typical format that optimization programs commonly appear in scientific journals. Therefore, the model must contain the following three main parts and with respect to the correct order as well:

1. the objective function to be optimized (either maximized or minimized);
2. the (sets of) constraints, or else the relationships between the decision variables and coefficients, right-hand side (RHS);
3. the decision variables and their domain space.

We used the programming environment of Python coupled with its modeling library, namely Pyomo. Similar approaches in terms of software selection have been presented for Differential and Algebraic Equations (DAE) modeling and optimization in (Nicholson et al., 2018; Nikolić, 2016). By combining Python and Pyomo we have the ability to transform a simplified representation of a mathematical model initially written in LaTeX into a formal AML formulation and eventually optimize it. In other words, the platform *reads* LaTeX code and then *writes* Pyomo abstract models or *the code generates code*. The resulting *.py* file is usable outside of the platform's frame, thus not making the binding and usage of these two necessary after conversion. The main components that we employed for this purpose are the following:

- Front-end: HTML, JavaScript, MathJax[4] and Google Polymer[5];
- Back-end: Python with Django[6] and Pyomo.

In order to increase the effectiveness and user-friendliness of the platform, a Graphical-User Interface (GUI) based on HTML, JavaScript (front-end) and Django as the web-framework (back-end) has been implemented, as shown in Figure 2. The user-input is facilitated mainly via Polymer objects[7]. As the main feature of the platform is to allow modeling in LaTeX language, we used MathJax as the rendering engine. In this way, the user can see the compiled version of the input model. All of these components form a single suite that works across different computational environments. The front-end is plain but incorporates the necessary functionality for input and output, as well as some solver options. The role of the back-end is to establish the communication between the GUI and the parser with the functions therein. In this way the inputs are being processed inside Python in the background, and the user simply witnesses a seamless working environment without having to understand the *black-box* parser in detail.

The main components of the GUI are:

- *Abstract model input*: The input of the LaTeX model, either directly inside the Polymer input text-box or via file upload (a *.tex* containing the required source LaTeX code)
- *Data files*: The input of the data set which follows the abstract definition of the model via uploading the AMPL-format (*.dat*) data file
- *Solver options*: An array of solver - related options such as:
    1. NEOS server job using CPLEX

[4]https://www.mathjax.org/
[5]https://www.polymer-project.org/
[6]https://www.djangoproject.com/
[7]https://www.polymer-project.org/

2. Solve the relaxed LP (if MILP)
3. Select GPLK (built-in) as the optimization solver
4. Select CPLEX (if available) as the optimization solver (currently set to default)

The following is an example of a LaTeX formulated optimization problem which is ready to use with the platform; the well-known Traveling Salesman Problem (TSP) (Applegate et al., 2007):

$$\begin{aligned}
&\texttt{minimize} && \sum_{i,j:i\neq j} (d_{i,j}x_{i,j}) \\
&\texttt{subject to:} && \\
& && \sum_{j:i\neq j} (x_{i,j}) = 1 && \forall i \\
& && \sum_{i:i\neq j} (x_{i,j}) = 1 && \forall j \\
& && u_i - u_j + nx_{i,j} \leq n-1 && \forall i \geq 2, j \leq |j|-1, i \neq j \\
& && u \in Z, x \in \{0,1\}
\end{aligned}$$

and the raw LaTeX code used to generate this was:

```
\text{minimize}  \sum\limits_{i,j: i \neq j}^{} (d_{i,j}x_{i,j})\\
\text{subject to: }\\
\sum\limits_{j: i \neq j}^{} (x_{i,j}) =  1 \quad \quad \forall i\\
\sum\limits_{i: i \neq j}^{} (x_{i,j}) =  1 \quad \quad \forall j\\
u_{i} - u_{j} + nx_{i,j} \leq n - 1  \quad \quad \forall i \geq 2, j \leq |j
    |-1, i \neq j\\
u \in \mathbb Z, x \in \{0,1\}\\
```

which is the input for the platform. The user can either input this code directly inside the Google polymer text box or via a *pre-made .tex* file which can be uploaded in the corresponding field of the GUI. Either way, the MathJax Engine then renders LaTeX appropriately so the user can see the resulting compiled model live. Subject to syntax-errors, the MathJax engine might or might not render the model eventually, as naturally expected. Empty lines or spaces do not play a role, as well as commented-out lines using the standard notation (the percentage symbol %). The model file always begins with the objective function sense, the function itself, and then the sets of constraints follow, with the variables and their respective

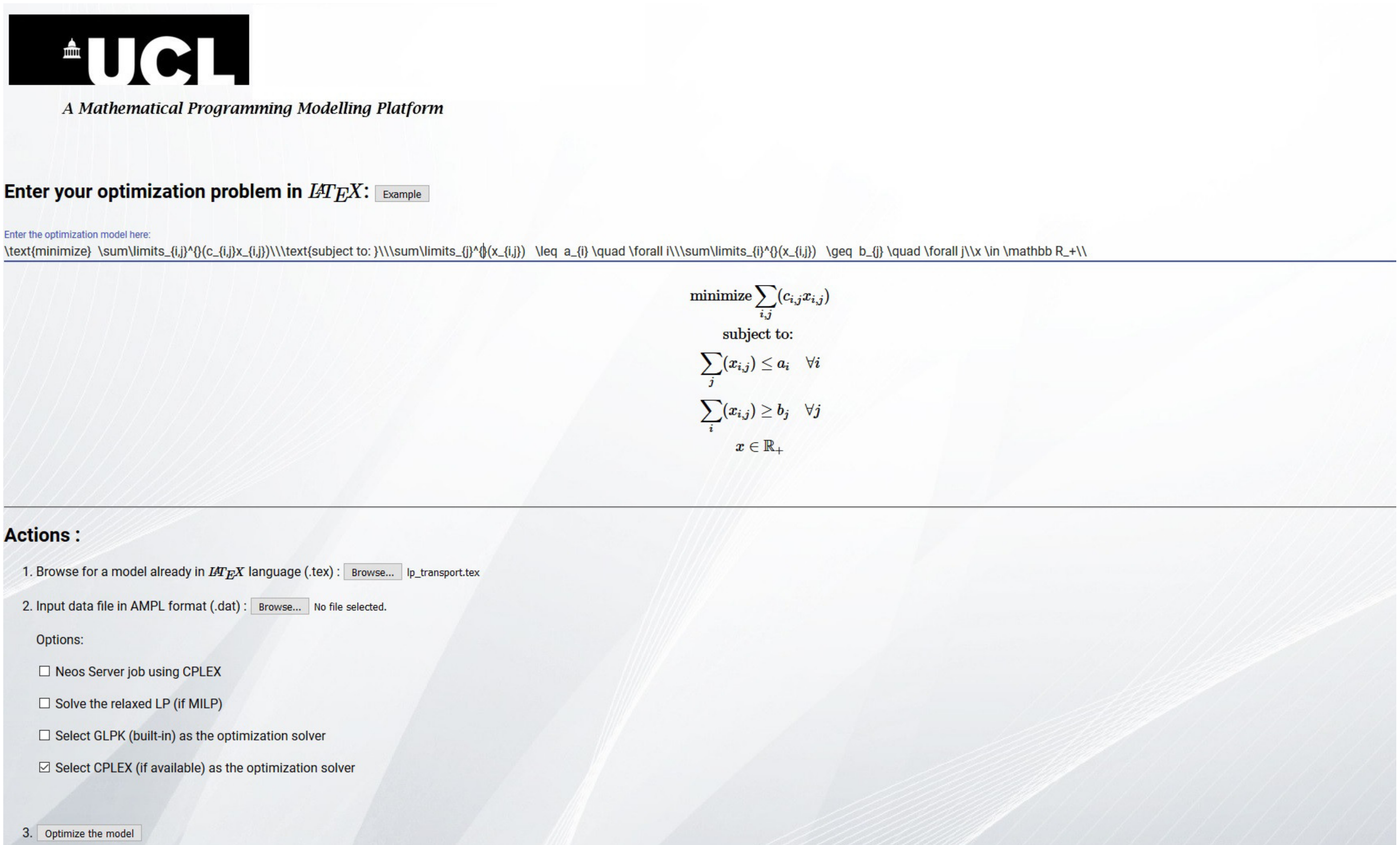

Figure 2: The simplified Graphical User Interface (GUI). The GUI contains the basic but fundamental options to use the platform, such as model input, solver selection and solution extraction.

type at the end of the file.

3. Parser - Execution Engine

As *parser* we define the part of the code (a collection of Python functions) in the back-end side of the platform which is responsible for translating the model written in LaTeX to Pyomo, the modeling component of the Python programming language. In order to effectively translate the user model input from LaTeX, we need an array of programming functions to carry out the conversion consistently since preserving the equivalence of the two is implied. The aim of the implementation is to provide minimum loss of generality in the ability to express mathematical notation for different modeling needs.

A detailed description of the implemented scheme is given in Figure 3. A modular design of different functions implemented in Python and the established communication of those (exchanging input and output-processed data) form the basic implementation concept. This type of design allows the developers to add functionality in a more clear and effective way. For instance, to upgrade the parser and support Mixed Integer Quadratic Programming (MIQP) problems, an update only to the parsing function assigned to convert the optimization objective function is required.

Once the *.tex* model file and the *.dat* AMPL formatted data file are given, the platform then starts processing the model. The conversion starts by reading the variables of the model and their respective types, and then follows with component identification (locating the occurrence of the variables in each constraint) and their inter-relationships (multiplication, division, summation etc.). Additionally, any summation and constraint conditional indexing schemes will be processed separately. Constraint-by-constraint the parser gradually builds the *.py* Pyomo abstract model file. It then merges through Pyomo the model with its data set and calls the selected solver for optimization.

3.1. Pre-processing

A significant amount of pre-processing takes place prior of parsing. The minimum and essential is to first tidy up the input; that is, clear empty lines and spaces, as well as reserved (by the platform) keywords that the user can include but do not play any role in functional

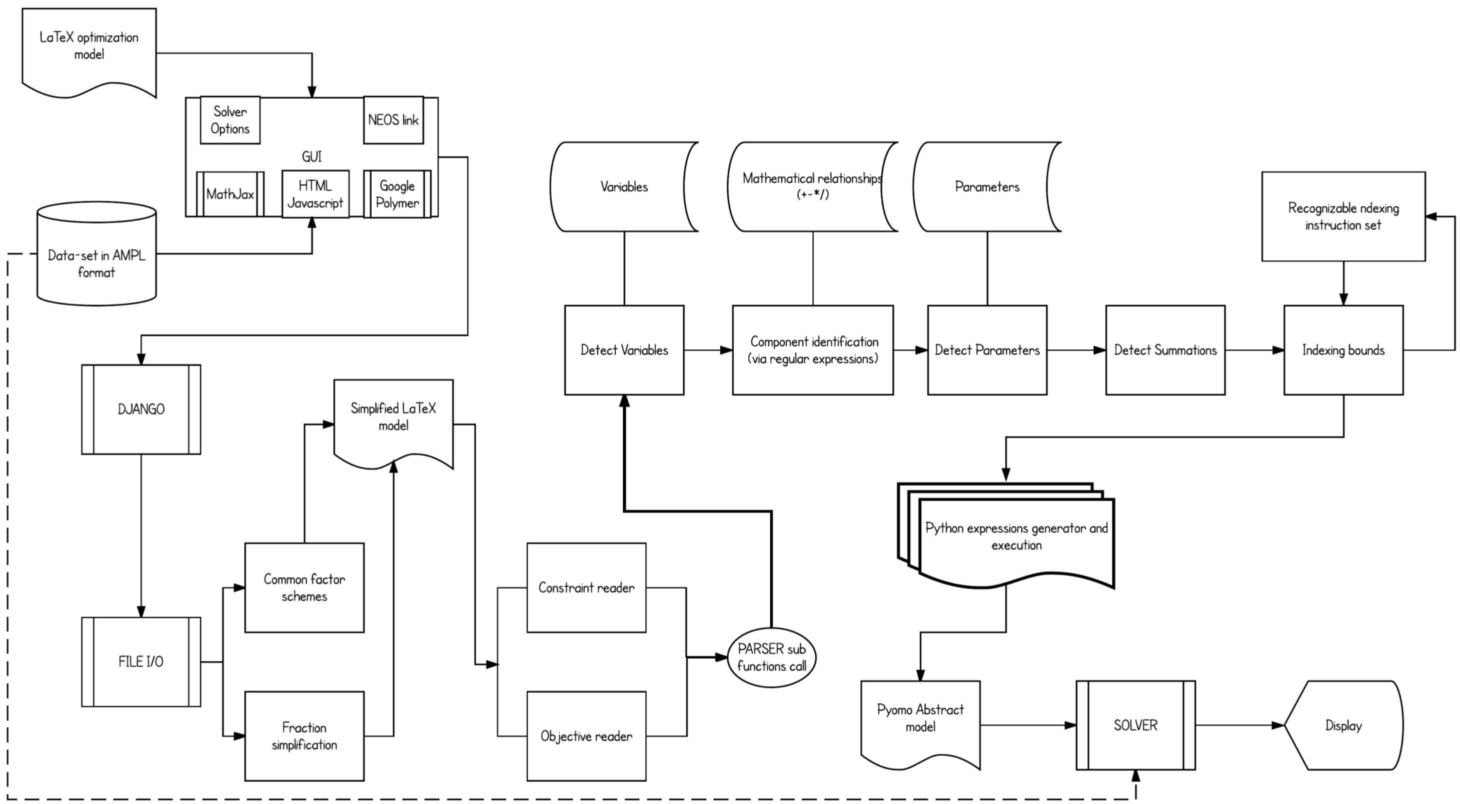

Figure 3: The overall flow of the implementation. From user input to solving the optimization problem or simply exporting the equivalent Pyomo model file.

parsing (such as the *quad* command). The platform also supports the use of Greek letters. For instance if a parameter is declared as α the platform identifies the symbol, removes the backslash and expects to find *alpha* in the data-file. This takes place also in the pre-processing stage.

The user can also opt-out selectively the constraints by putting regular comments in LaTeX, with the insertion of the percentage symbol (%) in the beginning of each expression. Once done, we attempt to simplify some types of mathematical expressions in order to be able to better process them later on. More specifically, we have two main functions that handle fractions and common factor (distributive expressions) simplifications. For example:

$$\frac{A_i B_j}{D_i} \text{ is then converted to: } (A_i B_j)/D_i$$

and

$$\beta(\alpha + 1) \text{ is converted as expected to: } \beta\alpha + \beta$$

When handling fractions, the user can employ the *frac* environment to generate them; however it is easier for the parser to process the analytical form. The same applies with the distributive form of multiplications. While it is more elegant for the eye and serves a compact representation, it is easier for the parser to extract the mathematical relationships from the analytical form.

This keeps the basic component identification functions intact, since their input is transformed first to the acceptable analytical format. Instead of transforming the parsing functions, we transform the input in the acceptable format. However, the user does not lose neither functionality nor flexibility, as this takes place in the background. To put it simply, either the user inputs the analytic form of an expression or the compact, the parser is still able to function correctly.

To frame the capabilities of the parser, we will now describe how the user can define optimization models in the platform with a given example and the successful parsing to Pyomo. The parser first attempts to split the model into its three major distinct parts:

- the objective function

- the sets of constraints
- the types of the variables defined

These three parts are in a way independent but interconnected as well.

3.2. Processing Variables

The parser first attempts to read the variables and their respective domain space (type). The platform is case sensitive since it is based on Pyomo. The processing is done using *string* manipulation functions, therefore the use of *regular expressions* in Python was essential and effective.

Reasonably, the focus was on consistency and reliability, rather computational performance mainly due to the lightweight workload of the processing demands in general. In order to do that, the parser uses *keywords* as *identifiers* while scanning from the top to the bottom of the manually curated *.tex* file which contains the abstract model in LaTeX. For the three respective different parts mentioned earlier, the corresponding identifiers are:

1. Objective function: $\{minimize, maximize\}$
2. Sets of constraints: $\{\backslash leq, \backslash geq, =\}$
3. Variables and their types: $\{\backslash mathbb\ , \{0,1\}\}$

This helps separate the processing into sections. Each section is analyzed and passes the information in Pyomo syntax in the *.py* output model file. Variable types can appear in the following way:

- `\in \mathbb R`

  for Real numbers ($\in R$)

- `\in \mathbb R_+`

  for non-negative Real numbers ($\in R_+$)

- `\in \mathbb R_{*}^{+}`

  for positive Real numbers ($\in R_*^+$)

- `\in \{0,1\}`

  for binary variables ($\in \{0, 1\}$)

- `\in \mathbb Z`

  for integers ($\in Z$)

- `\in \mathbb Z_+`

  for non-negative integers ($\in Z_+$)

- `\in \mathbb Z_{*}^{+}`

  for positive integers ($\in Z_*^+$)

In order to avoid confusion between lowercase and uppercase, the identifiers are converted to uppercase prior of comparison. Upon locating these keywords, the parser separates the processing and starts calling the corresponding functions. Once the variables and their types are processed (expected to be found at the bottom of the mathematical definition of the model), the parser then creates a list of strings for the names of the variables. This is one of the crucial structures of the parser and utilized alongside the entire run-time of the conversion process. A list of the same length, which holds the types of each respective variable, is also created. The platform in general uses Python lists to store information about variables, index sets, parameters, scalars etc.

3.3. Decomposing constraints and objective function expressions

Our approach for understanding the inter-mathematical relationships between the variables and the parameters relied on exploiting the fundamental characteristics of Linear Programming:

- Proportionality
- Additivity
- Divisibility

These mathematical relationships can help us understand the structure of the expressions and how to *decompose* them. By *decomposition* we define the fragmentation of each mathematical expression at each line of the .tex input model file into the corresponding variables, parameters, summations etc. so as we can process the given information accordingly. A simple graphical example is given in Figure 4.

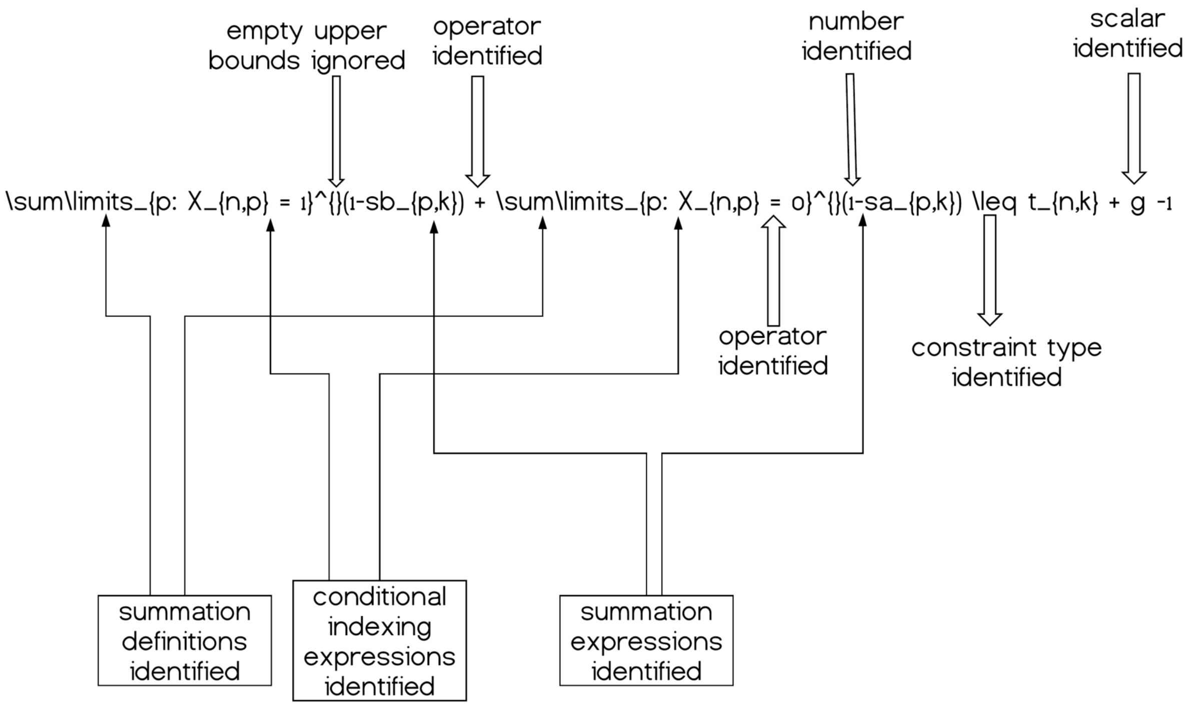

Figure 4: A simple constraint having its components (partially) decomposed and therefore identified; summations, operators, scalars and numerical quantities.

The decomposition with the regular expressions is naturally done via the strings of the possible operators found, that is: addition, subtraction, division $(+, -, /)$, since the asterisk to denote multiplication ($*$ or $\cdot$) is usually omitted in the way we describe the mathematical expressions (e.g. we write ax to describe coefficient a being multiplied by variable x). In some cases however it is imperative to use the asterisk to decompose a multiplication. For example, say Ds is a parameter and s is also a variable in the same model. There is no possible way to tell whether the expression Ds actually means $D{*}s$ or if it is about a new parameter altogether, since the parameters are not explicitly defined in the model definition (as in AMLs). Adding to that the fact that for the scalars there is no associated underscore character to identify the parameter as those are not associated with index sets, the task is even more challenging. Therefore, we should write $D*s$ if D is a scalar. As for parameters with index sets, for example Ds_is_i causes no confusion for the parser because the decomposition based on the underscore character clearly reveals two separate components. In this way, the platform also identifies new parameters. This means that since we know for instance that s is a variable but Ds is not we can dynamically identify Ds on the fly (as we scan the current constraint) as being a parameter which is evidently multiplied with variable s, both having index set i associated with them. However, we need to pay attention on components appearing iteratively in different or in the same sets of constraints; did we have the component already appearing previously in the model again? In that case we do not have to *declare* it again in the Pyomo model as a new quantity, as that would cause a modeling error.

By *declaration* we mean the real-time execution of a Python command that creates the associated terms inside the Pyomo abstract objected-oriented (OO) model. For instance if a set i is identified, the string $model.i = Set(dimen = 1)$ is first written inside the text version of the Pyomo model file, and then on-the-fly executed independently inside the already parsing Python function using the *exec* command. The execution commands run in a sequential manner. All the different possible cases of relationships between parameters and variables are dynamically identified, and the parser keeps track of the local (per constraint) and global (per model) list of parameters identified while scanning the model in dynamically growing lists.

Dynamic identification of the parameters and index sets is one of the elegant features of the platform, since in most Algebraic Modeling Languages (AMLs) the user explicitly defines the model parameters one-by-one. In our case, this is done in an intelligent automated manner. Another important aspect of the decomposition process is the identification of the constraint type ($<=, =, >=$), since the position of the operator is crucial to separate the left and the right hand side of the constraint. This is handled by an independent function. Decomposition also helps identify Quadratic terms. By automatic conversion of the caret symbol to $**$ (as this is one of the ways to denote power of a variable in Pyomo language) the split function carefully transfers this information intact to the Pyomo model.

3.4. Summations and conditional indexing

Summation terms need to be enclosed inside parentheses $(\cdots)$, even with a single component. This accelerates identification of the summation terms with clarity and consistency. Summations are in a way very different than processing a simplified mathematical expression in the sense that we impose restrictions on how a summation can be used. First of all, the corresponding function to process summations tries to identify how many summation expressions exist in each constraint at a time. Their respective indexing expressions are extracted and then sent back to the index identification functions to be processed. The assignment of conditional indexing with the corresponding summation is carefully managed. Then, the summation commands for the Pyomo model file are gradually built. Summations can be expressed in the following form, and two different fields can be utilized to exploit conditional indexing (upper and lower brackets):

```
\sum\limits_{p: X_{n,p} = 1}^{}(1-sb_{p,k})
```

which then compiles to: $\sum\limits_{p:X_{n,p}=1}(1-sb_{p,k})$

This means that the summation will be executed for all values of p, (that is for $p = 1 : |p|$) but only when $X_{n,p} = 1$ at the same time. If we want to use multiple and stacked summations (double, triple etc.) we can express them in the same way by adding the indexes for which the summation will be generated, as for example:

```
\sum\limits_{i,j}^{}(X_{i,j})
```

which then compiles to: $\sum\limits_{i,j}(X_{i,j})$

and will run for the full cardinality of sets i, j. Dynamic (sparse) sets imposed on constraints can be expressed as:

```
X_{i,j} = Y_{i,j} \forall (i,j) \in C \\
```

which then compiles to: $X_{i,j} = Y_{i,j} \quad \forall (i,j) \in C$

This means that the constraint is being generated only for those values of (i, j) which belong to the dynamic set C. In order to achieve proper and precise processing of summations and conditional indexing, we have built two separate functions assigned for the respective tasks. Since specific conditional indexing schemes can take place both for the generation of an entire constraint or just simply for a summation inside a constraint, two different sub-functions process this portion of information. This is done using the *forall* command at the end of each constraint, which changes how the indexes are being generated for the *vertical* expansion of the constraints from a specific index set. Concerning summations it is done with the bottom bracket information for *horizontal* expansion, as we previously saw for instance with $p : X_{n,p} = 1$.

A series of challenges arise when processing summations. For instance, which components are inside a summation symbol? A variable that might appear in two different summations at the same constraint can cause confusion. Thus, using a binary list for the full length of variables and parameters present in a constraint we identify the terms which belong to each specific summation. This binary list gets re-initialized for each different summation expression. From the lower bracket of each summation symbol, the parser is expecting to understand the indexes for which the summation is being generated. This is done by either simply stating the indexes in a plain way (for instance a, b or if a more complex expression is used, the for-loop indexes for the summations are found *before* the colon symbol (:).

3.5. Constraint indexing

At the end of each constraint, the parser identifies the "∀"' (*forall*) symbol which then helps understand for which indexes the constraints are being sequentially generated (vertical expansion). For instance $\forall (i, j) \in C$ makes sure that the constraint is not generated for all combinations of index sets i, j, but only the ones appearing in the sparse set C. The sparse sets are being registered also on the fly, if found either inside summation indexing brackets or in the constraint general indexing (after the ∀ symbol) by using the keywords *in*, *notin*. The simplest form of constraint indexing is for instance:

$$\sum_{j:i \neq j} (x_{i,j}) = 1 \qquad \forall i,$$

where the constraint is vertically expanding for all elements of set i and the summation is running for all those values of set j such that i is not equal to j. More advanced cases of constraint conditional indexing are also identified, as long as each expression is separated with the previous one by using a comma. For example in:

$$\forall i < |i|, j \geq i + 1$$

we see each different expression separated so the parser can process the corresponding indexing. Three different functions handle identification on constraint- level and the input for the general function that combines these three, accepts as input the whole expression. We process each component (split by commas) iteratively by these three functions:

1. to identify left part (before the operator/reserved keyword/command)
2. the operator and
3. the right-hand part

For example, in $i < |i|$, the left part is set i, the operator is $<$ and the right-hand part is the cardinality of set i. In this way, by adding a new operator in the acceptable operators list inside the code, we allow expansion of supported expressions in a straightforward manner.

4. An illustrative parsing example

Let us now follow the sequential steps that the parser takes to convert a simple example. Consider the well-known *transportation problem*:

$$\begin{aligned}
&\texttt{minimize} && \sum_{i,j}(c_{i,j}x_{i,j}) \\
&\texttt{subject to:} && \\
& && \sum_{j}(x_{i,j}) \leq a_i \quad \forall i \\
& && \sum_{i}(x_{i,j}) \geq b_j \quad \forall j \\
& && x \in R_+
\end{aligned}$$

We will now provide in-depth analysis of how each of the main three parts in the model can be processed.

4.1. Variables

The parser first attempts to locate the line of the *.tex* model file that contains the variable symbols and their respective domains. This is done by trying to identify any of the previously presented reserved keywords specifically for this section. The parser reaches the bottom line by identifying the keyword $mathbbR\_+$ in this case. Commas can separate variables belonging to the same domain, and the corresponding parsing function splits the collections of variables of the same domain and processes them separately.

In this case, the parser identifies the domain and then rewinds back inside the string expression to find the variable symbols. It finds no commas, thus we collect only one variable with the symbol x. The platform then builds two Python lists with the name of the variables found and their respective types.

4.2. Objective function

The parser then reads the optimization sense (by locating the objective function expression using the keywords, in this case *minimize*) and tries to identify any involved variables in the objective function. In a different scenario, where not all of the model variables are present in the objective function, a routine identifies one-by-one all the remaining variables and their associated index sets in the block of the given constraint sets.

The parser first attempts to locate any summation symbols. Since this is successful, the contained expression is extracted as $c_{\{i,j\}}x_{\{i,j\}}$, by locating the parentheses bounds (). In case of multiple summations, or multiple expressions inside the parentheses, we process them separately. The bounds of the summation symbol (the lower and upper brackets) respectively will be analysed separately. In this case, the upper one is empty, so the lower one contains all the indexes for which the summation has to scale. Separated by commas, a simple extraction gives i, j to be used for the Pyomo for-loop in the expression. There is no colon identified inside the lower bracket of the summation, thus no further identification of conditional indexing is required.

A split function is then applied on the extracted mathematical expression $c\_\{i, j\}x\_\{i, j\}$ to begin identification of the involved terms. Since there are no operators $(*, +, -, /)$ we have a list containing only one item; the combined expression. It follows that the underscore characters are used to frame the names of the respective components. It is easy to split on these characters and then create a list to store the pairs of the indexes for each component. Thus, a sub-routine detects the case of having more than just one term in the summation-extracted expression. In this example, c is automatically identified as a parameter because of its associated index set which was identified with the underscore character and since it does not belong to the list of variables.

The global list of parameters is then updated by adding c, as well as the parameters for the current constraint/objective expression. This helps us clarify which parameters are present in each constraint as well as the set of parameters (unique) for the model thus far, as scanning goes on. Once the parameter c and variable x are identified and registered with their respective index sets, we proceed to read the constraint sets. The parser creates expressions as the ones shown below for this kind of operations:

```
model.i = Set(dimen=1) \\
model.j = Set(dimen=1) \\
model.c = Param(model.i,model.j, initialize = 0) \\
model.x = Var(model.i,model.j, domain=NonNegativeReals) \\
```

Since the objective function summation symbol was correctly identified with the respective indexes, the following code is generated and executed:

```
def obj_expression(model):
  model.F = sum(model.c[i,j]*model.x[i,j] for i in model.i for j in model.j)
  return model.F
model.OBJ = Objective(rule=obj_expression, sense = minimize)
```

4.3. Constraints

Since the constraints sets are very similar, for shortness we will only analyze the first one. The parser first locates the constraint type by finding either of the following operators $\leq, \geq, =$. It then splits the constraint in two parts, left and right across this operator. This is done to carefully identify the position of the constraint type operator for placement into the Pyomo constraint expression later on.

The first component the parser gives is the terms identified raw in the expression $([' x'_{i,j}, ' a'_i])$. Parameter a is identified on the fly and since x is already registered as a variable and the parser proceeds to only register the new parameter by generating the following Pyomo expressions:

```
model.a = Param(model.i, initialize = 0)
```

The platform successfully identifies which terms belong to the summation and which do not and separates them carefully. Eventually the $\forall$ symbol gives the list of indexes for which the constraints are being generated. This portion of information in the structure of a Pyomo constraint definition goes in replacing X in the following piece of code:

```
def axb_constraint_rule_1(model,X):
```

and the full resulting function is:

```
def axb_constraint_rule_1(model,i):
  model.C_1= sum(model.x[i,j] for j in model.j) <= model.a[i]
  return model.C_1
model.AxbConstraint_1=Constraint(model.i,rule=axb_constraint_rule_1)
```

5. Discussion

Developing a parser that would be able to *understand* almost every different way of writing mathematical models using LaTeX is nearly impossible; however, even by framing the way the user could write down the models, there are some challenges to overcome. For instance, the naming policy for the variables and parameters. One would assume that these would cause no problems but usually this happens because even in formal modeling languages, the user states the names and the types of every component of the problem. Starting from the sense of the objective function, to the names and the types of the variables and parameters as well as their respective sizes and the names of the index sets, everything is explicitly defined. This is not the case though in this platform; the parser recognizes the parameters and index sets with no prior given information. This in turn imposes trade-offs in the way we write the mathematical notation. For instance multiple index sets have to be separated by commas as in $x_{i,j}$ instead of writing x_{ij}.

By scanning a constraint, the parser quickly identifies as mentioned the associated variables. In many cases parameters and variables might have multiple occurrences in the same constraint. This creates a challenging environment to locate the relationships of the parameters and the variables since they appear in multiple locations inside the string expressions and in different ways. On top of this, the name of a parameter can cause identification problems because it might be a sub/super set of the name of another parameter, e.g. parameter AB, and parameter ABC. Therefore naming conflicts are carefully resolved by the platform by meticulously identifying the exact location and occurrences of each term.

Challenges also arise in locating which of the terms appearing in a constraint belong to summations, and to which summations; especially when items have multiple occurrences inside a constraint, there needs to be a unique identification so as to include a parameter (or a variable) inside a specific summation or not. We addressed this with the previously introduced binary lists. Then for each of those summation symbols, the items activated (1) are included in the summation or not (0) and the list is generated for each different summation within the expression.

Finally, it is worth mentioning that the amount of lines/characters to represent a model in LaTeXin comparison with the equivalent model in Pyomo is substantially smaller. In this respect, the platform accelerates the modeling development process.

6. Conclusions

We presented a platform for rapid model generation using LaTeX as the input language for mathematical programming, starting with the classes of LP, MILP and MIQP. The platform is based on Python and parses the input to Pyomo to successfully solve the underlying optimization problems. It uses a simple GUI to facilitate model and data input based on Django as the web-framework. The user can exploit locally installed solvers or redirect to NEOS server. This prototype platform delivers transparency and clarity, speedup of the model design and development process (by significantly reducing the required characters to type the input models) and abstracts the syntax from programming languages and AMLs. It therefore delivers reproducibility and the ability to replicate scientific work in an effective manner from an audience not necessarily versed in coding. Future work includes full expansion of the platform's capabilities to support nonlinear terms as well as differential and algebraic equations.

Author Contributions

Conceived and designed the experiments: LGP. Analyzed the data: LGP, CPT. Performed the computational work and prepared figures and tables: CPT. Wrote the paper: CPT and LGP. Approved the final draft: LGP.

7. Acknowledgments

We gratefully acknowledge financial support from *The Leverhulme Trust* under Grant number RPG-2015-240 and the UK Engineering and Physical Sciences Research Council (EPSRC) under project EP/M027856/1. We would also wish to thank Prof. Eric Fraga and Dr. Aristotelis Kittas for useful discussions.

References

Applegate, D. L., Bixby, R. E., Chvatal, V., Cook, W. J., 2007. The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics). Princeton University Press, Princeton, NJ, USA.

Bertsimas, D., Tsitsiklis, J., 1997. Introduction to Linear Optimization, Third printing edition Edition. Athena Scientific.

Bieber, N., Ker, J. H., Wang, X., Triantafyllidis, C., van Dam, K. H., Koppelaar, R. H., Shah, N., 2018. Sustainable planning of the energy-water-food nexus using decision making tools. Energy Policy 113, 584 – 607.
URL http://www.sciencedirect.com/science/article/pii/S0301421517307838

Bisschop, J., Roelofs, M., 2011. AIMMS language reference, version 3.12. Paragon Decision Technology.

Bruce A. McCarl et. al., 2013. McCarl Expanded GAMS User Guide, GAMS Release 24.2.1. GAMS Development Corporation, Washington, DC, USA.
URL http://www.gams.com/mccarl/mccarlhtml/gams_user_guide_2005.htm

Cohen, M. C., Leung, N.-H. Z., Panchamgam, K., Perakis, G., Smith, A., 2017. The impact of linear optimization on promotion planning. Operations Research 65 (2), 446–468.
URL https://doi.org/10.1287/opre.2016.1573

Dunning, I., Huchette, J., Lubin, M., 2017. Jump: A modeling language for mathematical optimization. SIAM Review 59 (2), 295–320.
URL https://doi.org/10.1137/15M1020575

Editorial, 2014. Software with impact. Nature Methods 11 (211).
URL http://dx.doi.org/10.1038/nmeth.2880

Fourer, R., Gay, D., Kernighan, B., 1993. AMPL: A Modeling Language for Mathematical Programming. Scientific Press.
URL https://books.google.co.uk/books?id=8vJQAAAAMAAJ

Grossmann, I. E., Apap, R. M., Calfa, B. A., Garca-Herreros, P., Zhang, Q., 2016. Recent advances in mathematical programming techniques for the optimization of process systems under uncertainty. Computers & Chemical Engineering 91, 3 – 14.
URL http://www.sciencedirect.com/science/article/pii/S0098135416300540

Hart, W. E., Laird, C. D., Watson, J.-P., Woodruff, D. L., Hackebeil, G. A., Nicholson, B. L., Siirola, J. D., 2017. Pyomo–optimization modeling in python, 2nd Edition. Vol. 67. Springer Science & Business Media.

Hart, W. E., Watson, J.-P., Woodruff, D. L., 2011. Pyomo: modeling and solving mathematical programs in python. Mathematical Programming Computation 3 (3), 219.
URL https://doi.org/10.1007/s12532-011-0026-8

Heuberger, C. F., Rubin, E. S., Staffell, I., Shah, N., Dowell, N. M., 2017. Power capacity expansion planning considering endogenous technology cost learning. Applied Energy 204 (Supplement C), 831 – 845.
URL http://www.sciencedirect.com/science/article/pii/S0306261917309479

Jackson, M., 2012. Aspects of abstraction in software development. Software & Systems Modeling 11 (4), 495–511.
URL https://doi.org/10.1007/s10270-012-0259-7

Jovanović, I., Savić, M., Živković, Ž., Boyanov, B. S., Peltekov, A., 2016. An linear programming model for batch optimization in the ecological zinc production. Environmental Modeling & Assessment 21 (4), 455–465.
URL https://doi.org/10.1007/s10666-015-9485-z

Kallrath, J., 2004. Modeling Languages in Mathematical Optimization (APPLIED OPTIMIZATION). Kluwer Academic Publishers, Norwell, MA, USA.

Knijnenburg, T. A., Klau, G. W., Iorio, F., Garnett, M. J., McDermott, U., Shmulevich, I., Wessels, L. F. A., 2016. Logic models to predict continuous outputs based on binary inputs with an application to personalized cancer therapy. Scientific Reports 6, 36812.
URL http://dx.doi.org/10.1038/srep36812

Knuth, D. E., 1984. Literate programming. Comput. J. 27 (2), 97–111.
URL http://dx.doi.org/10.1093/comjnl/27.2.97

Kratica, J., Dugoija, D., Savi, A., 2014. A new mixed integer linear programming model for the multi level uncapacitated facility location problem. Applied Mathematical Modelling 38 (7), 2118 – 2129.
URL http://www.sciencedirect.com/science/article/pii/S0307904X13006240

Leek, J. T., Peng, R. D., 2015. Opinion: Reproducible research can still be wrong: Adopting a prevention approach. Proceedings of the National Academy of Sciences 112 (6), 1645–1646.
URL http://www.pnas.org/content/112/6/1645

Liu, S., Papageorgiou, L. G., 2013. Multiobjective optimisation of production, distribution and capacity planning of global supply chains in the process industry. Omega 41 (2), 369 – 382.
URL http://www.sciencedirect.com/science/article/pii/S0305048312000813

Liu, S., Papageorgiou, L. G., 2017. Fair profit distribution in multi-echelon supply chains via transfer prices. Omega.
URL http://www.sciencedirect.com/science/article/pii/S0305048316307897

Lubin, M., Dunning, I., 2015. Computing in operations research using Julia. INFORMS Journal on Computing 27 (2), 238–248.
URL https://doi.org/10.1287/ijoc.2014.0623

Melas, I. N., Samaga, R., Alexopoulos, L. G., Klamt, S., 2013. Detecting and removing inconsistencies between experimental data and signaling network topologies using integer linear programming on interaction graphs. PLOS Computational Biology 9 (9), 1–19.
URL https://doi.org/10.1371/journal.pcbi.1003204

Mitsos, A., Melas, I. N., Siminelakis, P., Chairakaki, A. D., Saez-Rodriguez, J., Alexopoulos, L. G., 2009. Identifying drug effects via pathway alterations using an integer linear programming optimization formulation on phosphoproteomic data. PLOS Computational Biology 5 (12), 1–11.
URL https://doi.org/10.1371/journal.pcbi.1000591

Mouha, N., Wang, Q., Gu, D., Preneel, B., 2012. Differential and Linear Cryptanalysis Using Mixed-Integer Linear Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 57–76.
URL https://doi.org/10.1007/978-3-642-34704-7_5

Nicholson, B., Siirola, J. D., Watson, J.-P., Zavala, V. M., Biegler, L. T., 2018. pyomo.dae: a modeling and automatic discretization framework for optimization with differential and algebraic equations. Mathematical Programming Computation 10 (2), 187–223.
URL https://doi.org/10.1007/s12532-017-0127-0

Nikolić, D. D., 2016. Dae tools: equation-based object-oriented modelling, simulation and optimisation software. PeerJ Computer Science 2, e54.
URL https://doi.org/10.7717/peerj-cs.54

Papageorgiou, L. G., Rotstein, G. E., 1998. Continuous-domain mathematical models for optimal process

plant layout. Industrial & Engineering Chemistry Research 37 (9), 3631–3639.
URL http://dx.doi.org/10.1021/ie980146v

Romeijn, H. E., Ahuja, R. K., Dempsey, J. F., Kumar, A., 2006. A new linear programming approach to radiation therapy treatment planning problems. Operations Research 54 (2), 201–216.
URL https://doi.org/10.1287/opre.1050.0261

Rossum, G., 1995. Python reference manual. Tech. rep., Amsterdam, The Netherlands, The Netherlands.

Silva, J. C., Bennett, L., Papageorgiou, L. G., Tsoka, S., 2016. A mathematical programming approach for sequential clustering of dynamic networks. The European Physical Journal B 89 (2), 39.
URL https://doi.org/10.1140/epjb/e2015-60656-5

Sitek, P., Wikarek, J., 2015. A hybrid framework for the modelling and optimisation of decision problems in sustainable supply chain management. International Journal of Production Research 53 (21), 6611–6628.
URL http://dx.doi.org/10.1080/00207543.2015.1005762

Tanveer, M., 2015. Robust and sparse linear programming twin support vector machines. Cognitive Computation 7 (1), 137–149.
URL https://doi.org/10.1007/s12559-014-9278-8

Triantafyllidis, C. P., Koppelaar, R. H., Wang, X., van Dam, K. H., Shah, N., 2018. An integrated optimisation platform for sustainable resource and infrastructure planning. Environmental Modelling Software 101, 146 – 168.
URL http://www.sciencedirect.com/science/article/pii/S1364815217301391

Wang, X., Guo, M., Koppelaar, R. H. E. M., van Dam, K. H., Triantafyllidis, C. P., Shah, N., 2018. A nexus approach for sustainable urban energy-water-waste systems planning and operation. Environmental Science & Technology 52 (5), 3257–3266, pMID: 29385332.
URL https://doi.org/10.1021/acs.est.7b04659

Williams, H. P., 1999. Model Building in Mathematical Programming, 4th Edition, 4th Edition. Wiley.
URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0471997889

Xu, G., Tsoka, S., Papageorgiou, L. G., 2007. Finding community structures in complex networks using mixed integer optimisation. The European Physical Journal B 60 (2), 231–239.
URL https://doi.org/10.1140/epjb/e2007-00331-0

Yang, L., Liu, S., Tsoka, S., Papageorgiou, L. G., 2016. Mathematical programming for piecewise linear regression analysis. Expert Systems with Applications 44, 156–167.
URL http://dx.doi.org/10.1016/j.eswa.2015.08.034

Yang, L., Liu, S., Tsoka, S., Papageorgiou, L. G., 2017. A regression tree approach using mathematical programming. Expert Systems with Applications 78, 347 – 357.
URL http://www.sciencedirect.com/science/article/pii/S0957417417300957