Universidade do Minho
Escola de Engenharia

Hugo Afonso da Gião

LPBlocks - A Block-based Language for Linear Programming

February, 2022

Universidade do Minho
Escola de Engenharia

Hugo Afonso da Gião

LPBlocks - A Block-based Language for Linear Programming

Master's Dissertation
Master's in Informatics Engineering

Work supervised by
Jácome Cunha
Rui Pereira

February, 2022

Acknowledgements

I would like to thank the Univesity of Minho and HASLab/INESC TEC, my advisors Jácome Cunha and Rui Pereira for offering their guidance and experience, as well as all the volunteers that participated in our study and every person that helped in this project.

Abstract

Linear programming is a mathematical optimization technique used in numerous fields including mathematics, economics, and computer science, with numerous industrial contexts, including solving optimization problems such as planning routes, allocating resources, and creating schedules. As a result of its wide breadth of applications, a considerable amount of its user base lacks programming knowledge and experience and thus often resorts to using graphical software such as Microsoft Excel. However, despite its popularity amongst less technical users, the methodologies used by these tools are often *ad-hoc* and prone to errors.

Block-based languages have been successfully used to aid novice programmers and even children in programming. Thus, we created a block-based programming language termed LPBlocks that allows users to create linear programming models using data contained in spreadsheets. This language guides the users to write syntactically and semantically correct programs and thus aids them in a way that current languages do not. We have also implemented a web application where users can define linear programming models, reactively see their mathematical representation and execute them to obtain the optimization values for the variables defined by the users.

To assess the applicability of LPBlocks we used it to successfully express numerous and varied linear programming problems. Furthermore, we designed and ran a qualitative empirical study to understand the experience our tool and language brings to users from various backgrounds. Although we see differences amongst the users, most of them were able to model several problems using LPBlocks.

Keywords: Linear programming, Operations research, Optimization, Block-based languages, Visual languages, Blockly

Resumo

Programação linear é um conjunto de técnicas de otimização matemática utilizada em várias áreas estas incluem matemática, economia, ciências da computação e usos em contextos industriais, incluindo planear rotas, alocar recursos e planear horários. Como resulta das suas aplicações variadas uma grande quantidade dos seus utilizadores não possuem conhecimentos de programação e por isso utilizam software gráfico como o Microsoft Excel. Apesar da sua popularidade este software utiliza metodologias *ad-hoc* e propicias a erros.

As linguagem de programação por blocos tem surgido nos últimos anos com o intuito de ajudar programadores iniciantes, tendo mesmo aplicações no ensino de crianças. Sendo assim nos criamos uma linguagem de programação pro blocos que utiliza dados contidos em folhas de calculo para criar modelos de programação linear chamada LPBlocks. Esta linguagem guia utilizadores na criação de modelos semanticamente e sintaticamente corretos.

Para avaliar a validade de LPBlocks nos implementamos vários problemas utilizando a mesma. Posteriormente implementamos esta linguagem e utilizamo-la num estudo com utilizadores de vários níveis de experiência. Depois utilizamos a informação recolhida durante o estudo para avaliar LPBlocks e propor melhorias.

Palavras-chave: Programação linear, Investigação operacional, Linguagens de blocos, Linguagens visuais, Blockly ...

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Chapter 1

Introduction

This chapter introduces the context of this thesis work and the problem we address (Subsection 1.1). In Subsection 1.2 we expand on our solution to the given problem. Further in Subsection 1.3 we propose several questions that we intend on answering within this work. Then in Subsection 1.4 we showcase relevant contributions from this work and in Subsection 1.5 we describe the organization of the remainder of this document.

1.1 Motivation

Linear programming is a mathematical optimization technique used to find the best possible outcomes in problems specified by linear specification constraints. It originated as a discipline during the 1940s as an effort to tackle complex problems associated with wartime operations. These methods have since found uses in many areas in fields such as mathematics, computer science, business, economics, and engineering and problems such as planning, routing, scheduling, assignment, and design [23].

The versatility of linear programming in formulating all sorts of problems lends itself useful in many industrial contexts from schedule optimization to route planning. Since many of its users have little to no programming or technical knowledge, visual software such as *Microsoft Excel* is often the preferred tool when it comes to specifying and solving this type of problem [11].

Despite its widespread use, end users still face many obstacles when trying to define linear programming models in existing tools since they often require some knowledge of programming to achieve desired outcomes. This is the case of tools such as *MATLAB*[1] or *SAS/OR*[2], and various libraries such as Google OR-Tools[3] or *PuLP*[4]. Spreadsheet software such as *Excel* allows users to formulate and solve both linear

[1] https://www.mathworks.com/products/matlab.html
[2] https://www.sas.com/en_us/software/or.html
[3] https://developers.google.com/optimization
[4] https://pypi.org/project/PuLP/

and non-linear optimization problems using algorithms such as simplex, generalized reduced gradient, and evolutionary algorithms [20]. Users with less technical background will probably find it easier to use this kind of software. However, similarly to other *Excel* tasks this methodology is also *ad-hoc* and could cause problems to novice users due to input errors at the moment of defining variables as well as other errors in the spreadsheet building process due to its lack of methodology. Some of the underlying problems include adding constraints and optimization function from the spreadsheet to the solver, the difficulty in visualizing the complete model, not being able to define constraints by its columns, the inability to call variables by name instead of its cell position, and lack of an interactive model and building process. Other visual tools often use spreadsheet or spreadsheet-like software with the methodologies and problems that come with that, learning linear programming also comes with some difficulties that have been previously mitigated with the use of technology [6].

Some works have used visual languages to tackle aspects of linear programming, however, the majority of them focus on the educational and teaching of mathematical aspects of linear programming [12, 19]. The few existing projects focusing on the applied side of linear programming tend to be several decades old and have dated and unappealing interfaces and do not make use of recent advances in the field of visual languages and human-centered computing [13, 21].

1.2 Our Approach

Numerous projects have applied visual languages to various fields of computing, generally focused on increasing accessibility of novice and non-technical users as well as teaching. A considerable amount of these languages use the *Blockly* framework for their implementation [17]. These languages include *BlockPy* [3], a web-based platform that lets the user write and run *Python* code using a block-based language, and Scratch [15], a block-based visual programming language and educational tool mostly targeted at children.

Our approach involved creating a block-based language which we titled LPBlocks that lets users create linear programming formulations. We also implemented a highly reactive web interface possessing numerous features such as dynamic compilation, error messages, the ability to load data from spreadsheets, and run the created models. Beyond our technical work, we also performed a study to evaluate the applicability of the language and a qualitative empirical study to understand how users interact with the language and tool.

This project comes as the sequence of a project aiming at the creation of visual language for the systematic creation of spreadsheets [16], initially we planned on integrating our language with the tool created by the authors but eventually decided that creating a web interface would be preferable for less technical users and the integration with *Excel* was not a critical aspect of our work.

1.3 Research questions

Considering our motivation and goals our thesis aims to answer the following questions:

- **RQ1** - *Can we use block-based languages to represent linear programming formulations.*
  Currently the tools used in both education and industry to create linear programming models either require pre-existing programming knowledge, use *ad-hoc* methodologies, or cannot be used for formulating general models. Since linear programming has several uses, improving its process for less technical users could be beneficial in improving their productivity and capabilities.

- **RQ2** - *Will using a block-based language and environment allow end users to express linear programming models.*
  We intend to evaluate the created language and environment assessing the experience of users with different backgrounds.

1.4 Contributions

With this work we make the following contributions:

- A visual, block-based language capable of expressing linear programming problems termed LP-Blocks.

- A web application that allows users to build, debug and run linear programming models using LPBlocks.

The source code is available at `https://github.com/h4g0/blockly-spaces` and the web application at `https://lpblocks.herokuapp.com/`.

The scientific publications associated with our work are the following:

- Linear Programming Meets Block-based Languages
  Hugo Gião, Rui Pereira, Jácome Cunha
  IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'21) [9]

- Towards a Block-based Language for Linear Programming
  Hugo Gião, Rui Pereira, Jácome Cunha
  12th National Symposium of Informatics (INForum'21) [10]

1.5 Document organization

To present our work, the document is organized as follows:

- Chapter 2 describes the state of the art when it comes to end-user tooling for linear programming, visual and block-based languages, and projects that bridge visual languages and linear programming.

- Chapter 3 showcases LPBlocks, a block-based language capable of expressing linear programming models. We explain how the data is received, the different blocks, and how to use them.

- Chapter 4 elaborates on the implementation details and compilation process for our language. We also showcase the interface of the tool we have implemented and its features.

- Chapter 5 discusses the applicability of our language by using it to solve a selection of problems taken from class notes and operations research textbooks.

- Chapter 6 presents an empirical study we executed in which the users after a brief tutorial solved several linear programming problems using our tool and then answered a survey where they shared their experiences.

- Chapter 7 analyzes the results of our work and proposes some possible improvements and continuation to this project.

Chapter 2

State of the art

To give a small introduction to linear programming we start with Section 2.1, where we showcase the process of formulating a given linear programming problem using mathematical notation.

We then expose a variety of tools related to our work. To do so we started by researching capable and some of the more user-friendly tools for operations research and linear programming purposes (Section 2.2). These tools are not necessarily the most popular and widely used in industry for the purposes above stated and are not necessarily the most widely used since the more programmatic methodology is often preferred. In the same Section, after exposing the different solutions and some of their methodologies, we proceed to do a comparison and comment on which would be the most appropriate for different user bases according to different aspects such as price and features and then discuss some of their shortcomings and possible solutions.

Then in Section 2.3 we present some works that apply visual languages and human-centered computing to the field of linear programming. The projects approached in this Section focus on both educational and industrial applications of linear programming. Some of them aim to help novice users use and learn linear programming and others aim at seasoned users.

In Section 2.4 we present some existing visual languages projects. These projects tackle various development and computing areas such as data science, application creation, and spreadsheet manipulating with applications in both enterprise and education markets. We present such works as they are related to our solution, which is a visual language.

2.1 Formulating a linear programming problem

There are many steps involved in the linear programming workflow, the first of them being understanding the problem at hand, what value does the user want to know and what this value depends upon. The next steps include defining the decision variables according to the problem, writing an objective function, defining the constraints, writing the constraints in terms of decision variables, and adding non-negativity to the constraints [1].

In this Section, we will use an example taken from an operations research textbook [5] to illustrate the process of creating linear programming formulations using mathematical notation. In this problem, a manufacturer of freeze-dried vegetable mixtures needs to respect certain requirements when it comes to each mixture composition. Each mixture is to be composed of beans, corn, broccoli, cabbage, and potatoes. The mixture needs to contain (by weight) at most 40% beans and at most 32% potatoes. The mixture should contain at least 5 grams of iron, 36 grams of phosphorus, and 28 grams of calcium. The nutrients in each vegetable and the costs can be seen in Table 2.1.

Vegetables	**Iron**	**Phosphorus**	**Calcium**	**Cost per pound**
Beans	0.5	10	200	20
Corn	0.5	20	280	18
Broccoli	1.2	40	800	32
Cabbage	0.3	30	420	28
Potatoes	0.4	50	360	16

Table 2.1: Data for the given problem

The first step to model this problem as a linear programming one is to create a set of variables that can be used to express the restrictions and objective described. We create the variables x_1, x_2, x_3, x_4, and x_5 as the number of pounds of beans, corn, broccoli, cabbage, and potatoes in each mixture. After creating the variables we create the following constraints:

- The percentage of beans in the mixture must be less than 40% of the total and the percentage of potatoes less than 32%. Since the total weight of the mixture is represented as the sum of the variables we get the following constraints in mathematical notation:

$$x_1 <= 0.4(x_1 + x_2 + x_3 + x_4 + x_5) \tag{2.1}$$

$$x_5 <= 0.4(x_1 + x_2 + x_3 + x_4 + x_5) \tag{2.2}$$

- To achieve the required level of nutrients, for each of the nutrients (iron, phosphorus, and calcium, respectively) we need to take into account the amount of each of them in the different vegetables and use this information to generate the following constraints:

$$0.5x_1, +0.5x_2 + 1.2x_3 + 0.3x_4 + 0.4x_5 > 5000 \tag{2.3}$$

$$10x_1, +20x_2 + 40x_3 + 30x_4 + 50x_5 > 36000 \tag{2.4}$$

$$200x_1, +280x_2 + 800x_3 + 420x_4 + 360x_5 > 28000 \tag{2.5}$$

- We also need to guarantee non-negativity by specifying that each of our variables is greater than 0.

$$x_1 >= 0 \tag{2.6}$$

$$x_2 >= 0 \tag{2.7}$$

$$x_3 >= 0 \tag{2.8}$$

$$x_4 >= 0 \tag{2.9}$$

$$x_5 >= 0 \tag{2.10}$$

Finally we define our objective function. Since our goal is to minimize costs and since the cost of the mixture is given by the sum each of our variables multiplied by its cost per pound we get the following objective:

$$minimize(20x_1 + 18x_2 + 32x_3 + 28x_4 + 16x_5) \tag{2.11}$$

All these equations can then be expressed in a linear programming solver to get the solution for each of the variables, thus giving the final solution. In the following Sections we describe some of the existing tools to do so.

2.2 Operations research tooling

2.2.1 Excel

Microsoft Excel[1] is a spreadsheet software developed by *Microsoft* with support for various platforms such as *Windows*, *macOS*, *Android*, and *iOS*. It includes many features related to manipulating and analyzing spreadsheets as well as various features related to business, engineering, and data analytics. *Excel* is closed source and paid for all users except students[2].

Excel allows for formulating and solving linear programming problems using its add-in solver. Using *Excel* for this task comes with some caveat, namely the methodology used for adding the constraints, variables, and the objective is *ad-hoc*, its interface not being user friendly, not having predefined templates for specific use cases and the solver not being native to *Excel*.

To be able to solve linear programming problems in *Excel* users need to follow several steps involved in downloading the solver add-in.

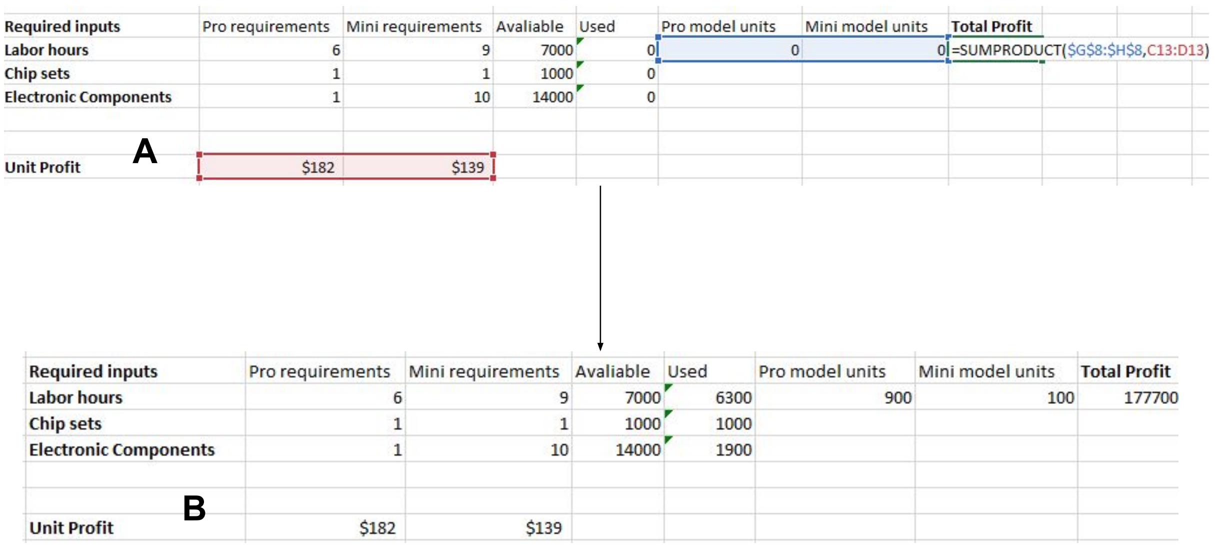
Having the solver add-in installed we can move to the next step. For demonstration purposes we will be using a tablet manufacturing example. This problem states that a tablet manufacturer has to optimize the production of two tablet models to maximize profits. This example was sourced from youtube [14].

To solve this problem we first define the problem variables, constraints, and optimization function inside an *Excel* spreadsheet as seen in Figure 2.1.A In this case, we define the requirements in terms of

[1] https://office.live.com/start/excel.aspx

[2] https://www.microsoft.com/en-us/microsoft-365/p/excel/cfq7ttc0k7dx?activetab=pivot%3aoverviewtab

Labour hours, *Chip sets* and *Electronic components* for each of the models as well as their availability. We then define two variables, one called *Pro Model units* and another called *Mini Model units*, indicating the number of units of each model to be produced. After we define the number of units for each of the requirements as the requirements for each of the model multiplied by the number of produced units, we then define the constant *Unit Profit* for both models and the objective function *Total profit* defined as being the *sumproduct* of each of the units with its profit.

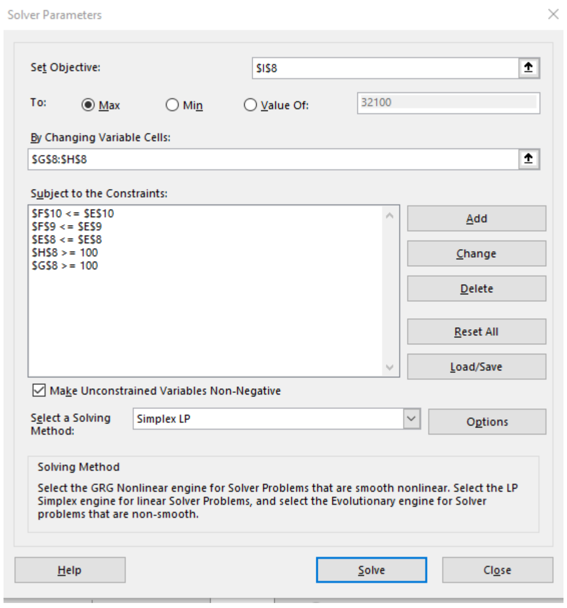


Required inputs	Pro requirements	Mini requirements	Avaliable	Used	Pro model units	Mini model units	Total Profit
Labor hours	6	9	7000	0	0	0	=SUMPRODUCT($G$8:$H$8,C13:D13)
Chip sets	1	1	1000	0			
Electronic Components	1	10	14000	0			
Unit Profit	$182	$139					

Required inputs	Pro requirements	Mini requirements	Avaliable	Used	Pro model units	Mini model units	Total Profit
Labor hours	6	9	7000	6300	900	100	177700
Chip sets	1	1	1000	1000			
Electronic Components	1	10	14000	1900			
Unit Profit	$182	$139					

Figure 2.1: Results outputted from the solver into the spreadsheet.

After defining the problem inside the *Excel* spreadsheet we open the solver add-in. We then are prompted with a window in which we add the different variables, constants, objective function and choose from one of the three available solvers as seen in Figure 2.2.



Set Objective: $I$8

To: [✓ Max] Min Value Of: 32100

By Changing Variable Cells: $G$8:$H$8

Subject to the Constraints:
$F$10 <= $E$10
$F$9 <= $E$9
$E$8 <= $E$8
$H$8 >= 100
$G$8 >= 100

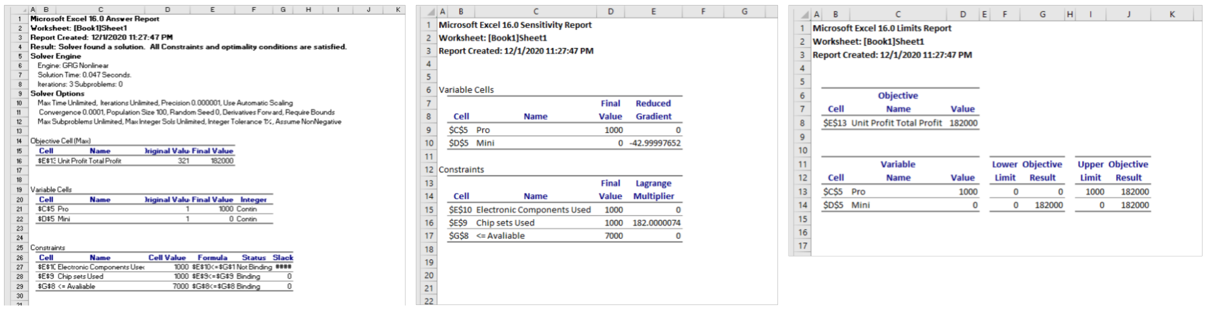
[✓] Make Unconstrained Variables Non-Negative

Select a Solving Method: Simplex LP

Solving Method
Select the GRG Nonlinear engine for Solver Problems that are smooth nonlinear. Select the LP Simplex engine for linear Solver Problems, and select the Evolutionary engine for Solver problems that are non-smooth.

Figure 2.2: Adding variables, constraints and defining the objective in the *Excel* solver.

After solving our problem the *Excel* solver add-in outputs both the solutions seen in Figure 2.1.B and a report detailing the problem specification as well as some of the steps and details about the algorithm used, seen in Figure 2.3.



Microsoft Excel 16.0 Answer Report
Worksheet: [Book1]Sheet1
Report Created: 12/1/2020 11:27:47 PM
Result: Solver found a solution. All Constraints and optimality conditions are satisfied.
Solver Engine
Engine: GRG Nonlinear
Solution Time: 0.047 Seconds.
Iterations: 3 Subproblems: 0
Solver Options
Max Time Unlimited, Iterations Unlimited, Precision 0.000001, Use Automatic Scaling
Convergence 0.0001, Population Size 100, Random Seed 0, Derivatives Forward, Require Bounds
Max Subproblems Unlimited, Max Integer Sols Unlimited, Integer Tolerance 1%, Assume NonNegative

Objective Cell (Max)

Cell	Name	)riginal Valu	Final Value
$E$1:	Unit Profit Total Profit	321	182000

Variable Cells

Cell	Name	)riginal Valu	Final Value	Integer
$C$5	Pro	1	1000	Contin
$D$5	Mini	1	0	Contin

Constraints

Cell	Name	Cell Value	Formula	Status	Slack
E1C	Electronic Components Usec	1000	$E$10<=$G$1	Not Binding	####
$E$9	Chip sets Used	1000	$E$9<=$G$9	Binding	0
$G$8	<= Avaliable	7000	$G$8<=$G$8	Binding	0

Microsoft Excel 16.0 Sensitivity Report
Worksheet: [Book1]Sheet1
Report Created: 12/1/2020 11:27:47 PM

Variable Cells

Cell	Name	Final Value	Reduced Gradient
$C$5	Pro	1000	0
$D$5	Mini	0	-42.99997652

Constraints

Cell	Name	Final Value	Lagrange Multiplier
$E$10	Electronic Components Used	1000	0
$E$9	Chip sets Used	1000	182.0000074
$G$8	<= Avaliable	7000	0

Microsoft Excel 16.0 Limits Report
Worksheet: [Book1]Sheet1
Report Created: 12/1/2020 11:27:47 PM

Cell	Objective Name	Value
$E$13	Unit Profit Total Profit	182000

Cell	Variable Name	Value	Lower Limit	Objective Result	Upper Limit	Objective Result
$C$5	Pro	1000	0	0	1000	182000
$D$5	Mini	0	0	182000	0	182000

Figure 2.3: Example of an *Excel* linear programming report

2.2.2 SAS/OR

The *SAS/OR* software allows the use of various state of the art optimization, simulation, and scheduling algorithms and software. User interfaces and functionalities vary greatly for the different *SAS/OR* features [3]. For some tasks such as *Mathematical optimization* users have access to a algebraic language, however *SAS/OR* provides users with a graphical interface for some specific optimization tasks such as *Discrete event simulation* and *Project and resource scheduling*. This software also boasts state of the art and optimized parallel algorithms.

This software is used across various industries and companies such as *Nestlé*, *Volvo*, and *Honda*. *SAS/OR* is paid and closed source with a free trying period and discounted or free for educational purposes[4].

2.2.3 NCSS

NCSS is a statistical and graphical program with linear and mixed integer programming capabilities. It uses Tableau or Bounds for data visualization and analysis[5]. The *NCCSS* software has many use cases and features for operations research, including solving linear programming and being equipped to solve more specific problems such as *Mixed Integer Programming*, *Assignment*, *Maximum Flow*

To solve linear programming problems using the *NCSS* software users must define the variables in a spreadsheet as seen in figure 2.4a, and subsequently indicate to the solver which are the columns that represent the variables, constraints, and logic as seen in 2.4b. When compared with *Excel*, *NCSS* uses a more structured approach seeing that all variables, logic columns and constraints must be defined in specific columns and does not require that the users build the model by selecting the data in the file using the mouse. The reports seen in 2.4c generated by the solver are similar to the ones generated by *Excel*.

[3]https://www.sas.com/pt_pt/software/or.html#m=features
[4]https://www.sas.com/pt_pt/software/how-to-buy/request-price-quote.html
[5]https://www.ncss.com/software/ncss/operations-research-in-ncss/#Technical

Despite improving on the *Excel* methodology, the *NCSS* methodology could be improved with a more interactive and user-friendly approach to the problem definition.

Pricing wise **NCSS** is paid for all users but possesses a free trial[6]. This software is also closed source.

2.2.4 MATLAB

MATLAB is a software tool supporting a visual environment created for iterative analysis and design process and a programming language that expresses matrices and arrays directly as well as a live editor and the capability to output text, graphics, and formatted notebooks[7]. The main appeal behind *MATLAB* stands from it requiring lower configuration and setup compared to other programming environments. Other factors distinguishing *MATLAB* from the alternatives stand from its performance, state of the art algorithms and scalability, its ease of integration, and industry-focused features such as its support for model-based design and easy deployment with business systems. One of its biggest setbacks stands from its pricing, at time of writing it does not have any free tier, *MATLAB* is closed source as well[7].

Users can use *MATLAB* to model and solve linear programming problems using the $linprog$ function. Users can use this function by expressing both variables and constants in a matrix form. As an example, the following inequalities constraint

$$x(1) + x(3) \leq 2 \tag{2.12}$$

$$x(1) + x(2)/4 \leq 1 \tag{2.13}$$

$$x(1) - x(2) \leq 2 \tag{2.14}$$

$$-x(1)/4 - x(2) \leq 1 \tag{2.15}$$

$$-x(1) - x(2) \leq -1 \tag{2.16}$$

$$-x(1) + x(2) \leq 2 \tag{2.17}$$

can be expressed in *MATLAB* in the following way:

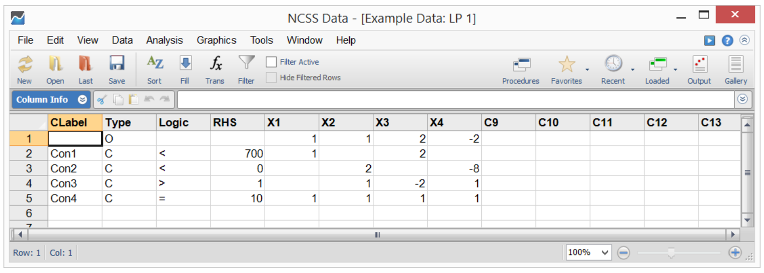
```
A = [1 1
   1 1/4
   1 -1
   -1/4 -1
   -1 -1
   -1 1];

b = [2 1 2 1 -1 2];
```

After defining the inequalities constraints, the next step is to define the objective function. For this example, the following function will be used:

$$-x(1) - x(2)/3 \tag{2.18}$$

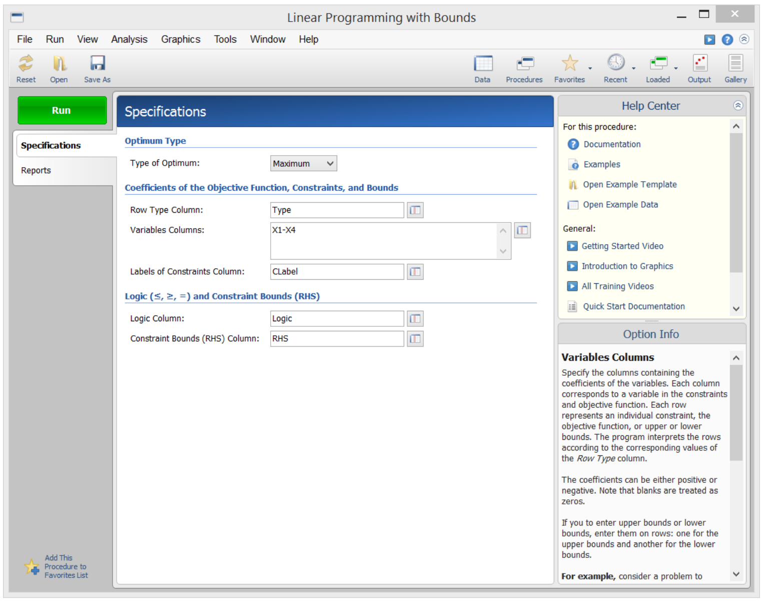
[6]https://www.ncss.com/online-store/

[7] https://www.mathworks.com/products/matlab.html



	CLabel	Type	Logic	RHS	X1	X2	X3	X4	C9	C10	C11	C12	C13
1		O			1	1	2	-2					
2	Con1	C	<	700	1		2						
3	Con2	C	<	0		2		-8					
4	Con3	C	>	1		1	-2	1					
5	Con4	C	=	10	1	1	1	1					
6													

(a) Problem construction inside the spreadsheet



Specifications

Optimum Type

Type of Optimum: Maximum

Coefficients of the Objective Function, Constraints, and Bounds

Row Type Column: Type

Variables Columns: X1-X4

Labels of Constraints Column: CLabel

Logic (≤, ≥, =) and Constraint Bounds (RHS)

Logic Column: Logic

Constraint Bounds (RHS) Column: RHS

Variables Columns

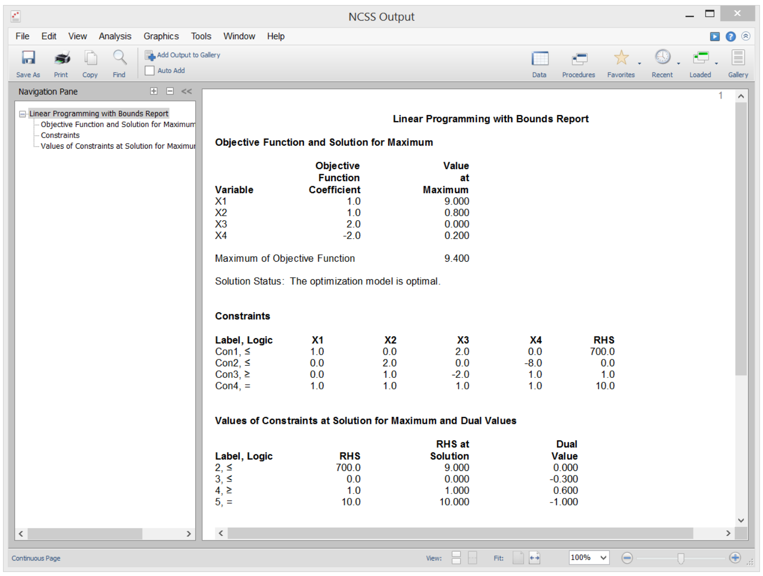
Specify the columns containing the coefficients of the variables. Each column corresponds to a variable in the constraints and objective function. Each row represents an individual constraint, the objective function, or upper or lower bounds. The program interprets the rows according to the corresponding values of the *Row Type* column.

The coefficients can be either positive or negative. Note that blanks are treated as zeros.

If you to enter upper bounds or lower bounds, enter them on rows: one for the upper bounds and another for the lower bounds.

For example, consider a problem to

(b) Adding the problem to the solver



```
                    Linear Programming with Bounds Report

Objective Function and Solution for Maximum

                 Objective             Value
                  Function                at
Variable       Coefficient           Maximum
X1                     1.0             9.000
X2                     1.0             0.800
X3                     2.0             0.000
X4                    -2.0             0.200

Maximum of Objective Function          9.400

Solution Status:  The optimization model is optimal.


Constraints

Label, Logic       X1          X2          X3          X4          RHS
Con1, ≤           1.0         0.0         2.0         0.0        700.0
Con2, ≤           0.0         2.0         0.0        -8.0          0.0
Con3, ≥           0.0         1.0        -2.0         1.0          1.0
Con4, =           1.0         1.0         1.0         1.0         10.0


Values of Constraints at Solution for Maximum and Dual Values

                                       RHS at              Dual
Label, Logic           RHS           Solution             Value
2, ≤                 700.0              9.000             0.000
3, ≤                   0.0              0.000            -0.300
4, ≥                   1.0              1.000             0.600
5, =                  10.0             10.000            -1.000
```

(c) Solver output

Figure 2.4: Linear programming problem solving in NCSS

Defined in *MATLAB*:

```
f = [-1 -1/3];
```

After defining the inequalities and objective function the next step is to call the *linprog* function:

```
x = linprog(f,A,b)
```

We then get the optimal solution found by the solver:

```
x = 2 x 1
   0.6667
   1.3333
```

The $linprog$ function accepts equality constraints; as an example the $x(1)+x(2)/4 = 1/2$ constraint can be added to our model:

```
Aeq = [1 1/4];
beq = 1/2;

x = linprog(f,A,b,Aeq,beq)
```

The $linprog$ function can accept lower and upper bound constraints and different solver algorithms and parameters. *MATLAB* contains more advanced features such as modeling problems that can be formulated in a problem-based approach and more complete outputs.

2.2.5 LINGO and What'sBest!

LINGO and *What'sBest!* are two tools created by the *LINDO SYSTEMS INC* to solve linear programming problems using data contained in spreadsheets. These tools constitute two different approaches to solve this problem. While *LINGO* uses the data in spreadsheets and uses an *SQL* like language to express the problem and call the solver, *What'sBest!* defines the problem inside an *Excel* spreadsheet and calls the solver using a GUI.

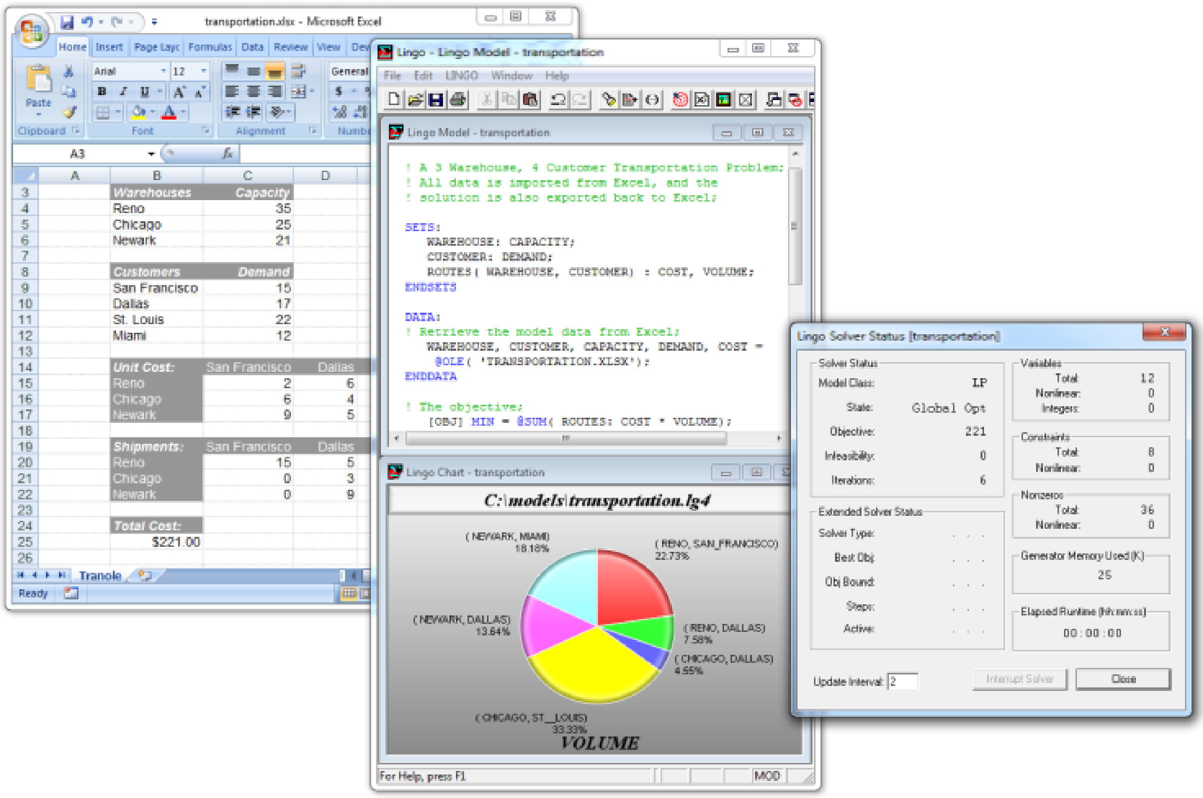
What'sBest! offers users a form layout within spreadsheets where users can define variables, constraints, and optimization functions, to finalize the problem definition and call the solver. Users then use the GUI provided by the *Excel* add-in to define the different variables, constraints, and functions in the spreadsheet cells as well as choosing the optimization solver and different report options[8].

LINGO, whose snapshots of the user interface can be seen in 2.5, is an optimization modeling software for linear, nonlinear, and integer programming. Lingo is a comprehensive tool designed to make solving various optimization problems simpler, faster, and more efficient. This tool provides an integrated package comprising a powerful language capable of expressing various optimization models, editing problems, and fast solvers. *LINGO* allows for reading data from spreadsheet files using its *SQL* like language[9].

2.2.6 Summary and comparisons

In this Section we summarize the findings and compare the different solutions according to five different categories:

[8]https://www.lindo.com/index.php/products/what-sbest-and-excel-optimization
[9]https://www.lindo.com/index.php/products/lingo-and-optimization-modeling



	A	B	C	D
3		*Warehouses*	*Capacity*	
4		Reno	35	
5		Chicago	25	
6		Newark	21	
7				
8		*Customers*	*Demand*	
9		San Francisco	15	
10		Dallas	17	
11		St. Louis	22	
12		Miami	12	
13				
14		*Unit Cost:*	San Francisco	Dallas
15		Reno	2	6
16		Chicago	6	4
17		Newark	9	5
18				
19		*Shipments:*	San Francisco	Dallas
20		Reno	15	5
21		Chicago	0	3
22		Newark	0	9
23				
24		*Total Cost:*		
25		$221.00		
26				

```
! A 3 Warehouse, 4 Customer Transportation Problem;
! All data is imported from Excel, and the
! solution is also exported back to Excel;

SETS:
   WAREHOUSE: CAPACITY;
   CUSTOMER: DEMAND;
   ROUTES( WAREHOUSE, CUSTOMER) : COST, VOLUME;
ENDSETS

DATA:
! Retrieve the model data from Excel;
   WAREHOUSE, CUSTOMER, CAPACITY, DEMAND, COST =
    @OLE( 'TRANSPORTATION.XLSX');
ENDDATA

! The objective;
   [OBJ] MIN = @SUM( ROUTES: COST * VOLUME);
```

Lingo Solver Status [transportation]

Solver Status
Model Class: LP
State: Global Opt
Objective: 221
Infeasibility: 0
Iterations: 6

Extended Solver Status
Solver Type: . . .
Best Obj: . . .
Obj Bound: . . .
Steps: . . .
Active: . . .

Variables
Total: 12
Nonlinear: 0
Integers: 0

Constraints
Total: 8
Nonlinear: 0

Nonzeros
Total: 36
Nonlinear: 0

Generator Memory Used (K)
25

Elapsed Runtime (hh:mm:ss)
00 : 00 : 00

Update Interval: 2

Interrupt Solver
Close

Figure 2.5: Lingo user interface

- **Cost:** We compare the different tools according to their pricing for both enterprise, education, and student users.
- **Code availability:** This metric compares the different tooling according to the availability of their code to the general public.
- **User interface:** In this category we compare the user interfaces available for each of the different tools and the tasks end users can accomplish using them.
- **Ease of use:** This metric is used to compare the ease of use of the solutions.
- **Features:** We use this metric to compare the applications in terms of available features.

Looking at Table 2.2 we can see that all of the tools above carry costs for enterprise users, however all of them have free trials, *Excel* and *SAS/OR* are free for students and all of the others are heavily discounted for students.

Excel	Paid with free trial and free student tier
SAS/OR	Paid with free trial and free student tier
NCSS	Paid with free trial and discounts for education and government use
MATLAB	Paid with heavy discounts for home, education and student users and free trial
Lingo and What'sBest!	Paid with free trial with education discounts

Table 2.2: Different solutions comparison cost

When it comes to user interfaces users have different options according to their needs. One can see in Tables 2.3, 2.4 and 2.5 that tools such as *Excel*, *NCSS*, and *What'sBest!* would make a better choice for users with little to no programming experience wanting to solve linear programming problems. For more experienced users wanting more freedom and capabilities when solving linear programming problems tools such as *MATLAB*, *Lingo* and *SAS/OR* would be a preferable choice and for users wanting to solve more advanced optimization problems, *SAS/OR* or *MATLAB* would be the tooling of choice.

However, despite the array of features offered by the different solutions and the availability of user interfaces for select tasks, there are still improvements that could be made in the usability, flexibility, and reliability of these interfaces. More notably various interfaces used for linear programming tasks such as the ones used in *Excel*, *NCSS*, and *What'sBest!* assume that users know the various steps involved in linear programming problem solving. The *Excel* tool has some downsides with its approach since its methodology involves a considerable amount of pointing, clicking and dragging and does not provide users with easy visualization of the model being built. Other tools such as *NCSS* and *What'sBest!* improve on the scenario by having the references to the cells done manually. However none of the previous tools have proactive measures to avoid user errors such as interactively showing the different constraints, variables, and optimization as the model is being constructed. Some of the more useful tools such as Excel lack some more advanced features such as more advanced solvers and options.

Excel	*Excel* has a user interface for linear programming and most tasks
SAS/OR	Possesses a graphical user interface for some operations research tasks, but linear programming can only be done using their algebraic language
NCSS	Yes similar to *Excel*
MATLAB	Like *SAS/OR* has some GUI, but not for linear programming
Lingo and What'sBest!	Yes very similar to *Excel*

Table 2.3: Different solutions comparison when it comes to User interface

Excel	Numerous features with many spreadsheet related uses; lacking in terms of operations research related features when compared with the other solutions few linear programming solvers and options
SAS/OR	Numerous optimization and operations research features
NCSS	Numerous features related to linear and mixed integer programming; various solvers and options
MATLAB	Numerous operations research features and a wide array of tools and a general purpose programming language
Lingo and What'sBest!	Numerous linear and mixed integer tools

Table 2.4: Different solutions comparison when it comes to features

Excel	Relatively easy to use for seasoned *Excel* users; requires the installation of additional add-ins and lacks a coherent interface
SAS/OR	Some optimization tasks are accessible to end users but others such as linear programming require some knowledge of mathematical notations and some programming
NCSS	Process very similar to *Excel* with a better and less prone to errors interface
MATLAB	Some tasks can be accomplished without programming but similarly to *SAS/OR* requires programming knowledge for others such as linear programming; however users with a mathematical background should be familiar with significant portions of the syntax.
Lingo and What'sBest!	The difficulty level of *Lingo* can be positioned between *SAS/OR* and *Excel* and *What'sBest!* has a similar workflow to *Excel*.

Table 2.5: Different solutions comparison when it comes to ease of use

2.3 Projects involving visual languages and linear programming

2.3.1 A graphics interface for linear programming

In previous works researchers introduced an interface for a software system that guides users when graphically building linear programming models instead of using a mathematical formulation [13]. Their interface (seen in Figure 2.6) boasts multiple features such as hierarchical decomposition, multiple model representations, alternative formulation approaches, the use of model templates, and database and model management features.

The authors identified the following steps to solve a linear programming problem: investigation, model formulation, data management, algorithmic solution and report generation and analysis. In their work the authors discuss only the first four stages. The first step, problem investigation, is due to the nature of the task being done manually. In their solution *LPFORM*, the second step being the model formulation in an

```
                                                       Sink
Source                  Conversion                     {residential,
{domestic,              {refineries,                   transportation,
foreign}                electric-utilities}            industrial}
       TSC                                TCS
          so,co,re                           co,si,pe
  [ ] ----------------> [=|-] ----------------------> [ ]
   |   Raw_energy        X        Processed_energy     ^
                          co,pe
   |   (oil,gas,coal)         (gasoline,electricity)  |
   |                                                   |
   v                    TSS                            |
                           so,si,re
   ------------------------------------------------->
                      Raw_energy
                      (oil,gas,coal)
```

Figure 2.6: Energy problem represented in LPFORM

algebraic language, is automated. The third step involves human input, this being necessary to input, store, and retrieve data from a relational database. The fourth step, solving the model, is done automatically.

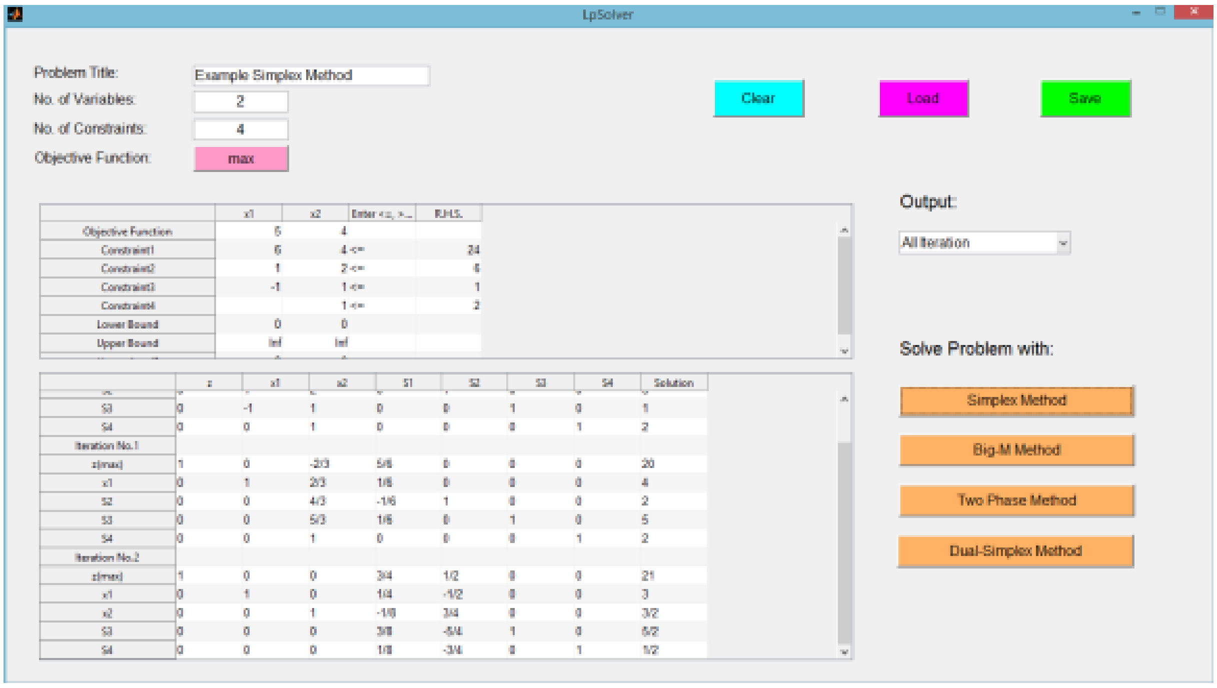
2.3.2 Creating a GUI Solver for Linear Programming Models in MATLAB

Researchers have introduced *LpSolver*, a Graphical User Interface (GUI) for linear programming problems using *MATLAB* [22]. Their solution was created for classroom-sized problems and boasts features such as computing expressions with symbolic variables and fractions and allowing the users to trace the optimization process.

When building models using the *LpSolver* users have access to a graphical interface seen in Figures 2.7,2.8 and 2.9 where they can input the data manually or load the data from a previously created file. However, users are discouraged to use *LpSolver* to solve problems with more than 50 constraints or 100 variables. Users can also save the created model and have access to different algorithms for solving the problem, this includes the Simplex Method, the Big-M Method, the Two-Phase Method, and the Dual-Simplex Method.

2.3.3 gLPS: A graphical tool for the definition and manipulation of linear problems

Collaud and Pasquier-Boltuck introduced *gLPS* (graphical Linear Programming System) [7]. This tool allows users to express linear problems using graphical objects (circles for restrictions, squares for variables, etc.) networked according to specific rules to form a model. The major strength of *gLPS* is grounded on it being able to express linear programming models belonging to a wide array of domains. *gLPS* is not only a

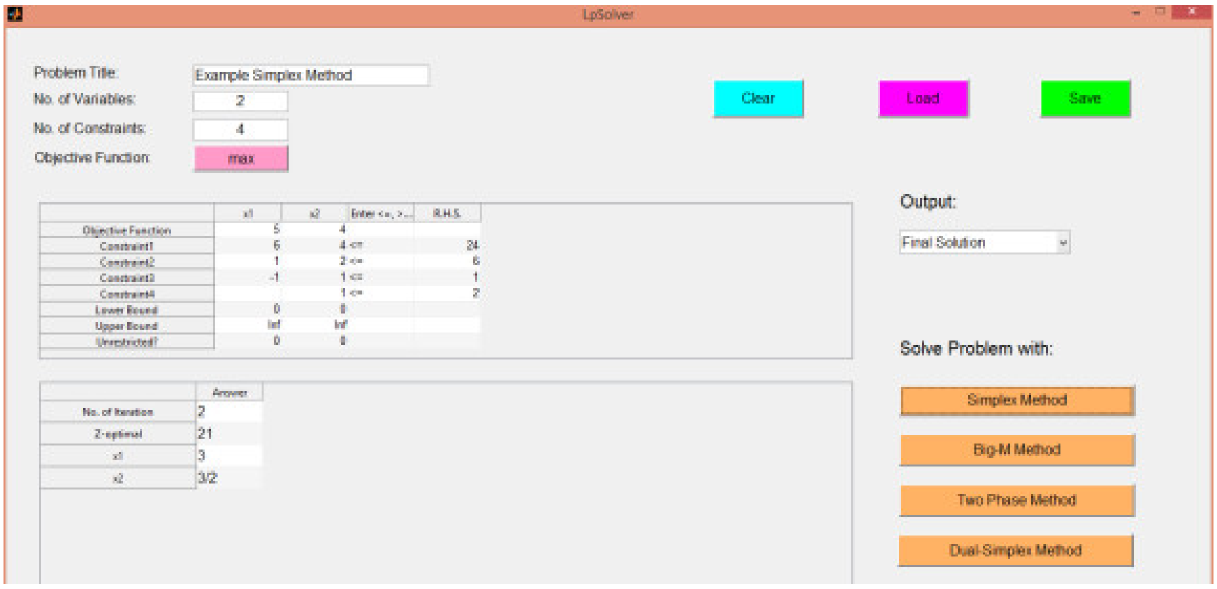


Problem Title: Example Simplex Method
No. of Variables: 2
No. of Constraints: 4
Objective Function: max
Output: All Iteration

	x1	x2	Enter <=, >...	R.H.S.
Objective Function	5	4		
Constraint1	6	4	<=	24
Constraint2	1	2	<=	6
Constraint3	-1	1	<=	1
Constraint4		1	<=	2
Lower Bound	0	0		
Upper Bound	Inf	Inf		

	z	x1	x2	S1	S2	S3	S4	Solution
S3	0	-1	1	0	0	1	0	1
S4	0	0	1	0	0	0	1	2
Iteration No.1								
z(max)	1	0	-2/3	5/6	0	0	0	20
x1	0	1	2/3	1/6	0	0	0	4
S2	0	0	4/3	-1/6	1	0	0	2
S3	0	0	5/3	1/6	0	1	0	5
S4	0	0	1	0	0	0	1	2
Iteration No.2								
z(max)	1	0	0	3/4	1/2	0	0	21
x1	0	1	0	1/4	-1/2	0	0	3
x2	0	0	1	-1/8	3/4	0	0	3/2
S3	0	0	0	3/8	-5/4	1	0	5/2
S4	0	0	0	1/8	-3/4	0	1	1/2

Figure 2.7: LpSolver graphical interface creating a formulation with output for all iterations



Problem Title: Example Simplex Method
No. of Variables: 2
No. of Constraints: 4
Objective Function: max
Output: Final Solution

	x1	x2	Enter <=, >...	R.H.S.
Objective Function	5	4		
Constraint1	6	4	<=	24
Constraint2	1	2	<=	6
Constraint3	-1	1	<=	1
Constraint4		1	<=	2
Lower Bound	0	0		
Upper Bound	Inf	Inf		
Unrestricted?	0	0		

	Answer
No. of Iteration	2
Z-optimal	21
x1	3
x2	3/2

Figure 2.8: LpSolver graphical interface with final solution output

modeling language but also an integrated software system for the creation, modification, and running of linear programming models.

Contrary to other projects presented in this Subsection *gLPS* was created for experts capable of formulating linear programming models algebraically. The author's name *LPForm* (Subsection 2.3.1) as being one language focused on improving linear programming for non-experts. The symbolism used by *gLPS* is a direct translation of the algebraic notation, a choice whose authors assert restricts the potential users of *gLPS*. The authors justify this choice by claiming that this notation would be a more natural approach

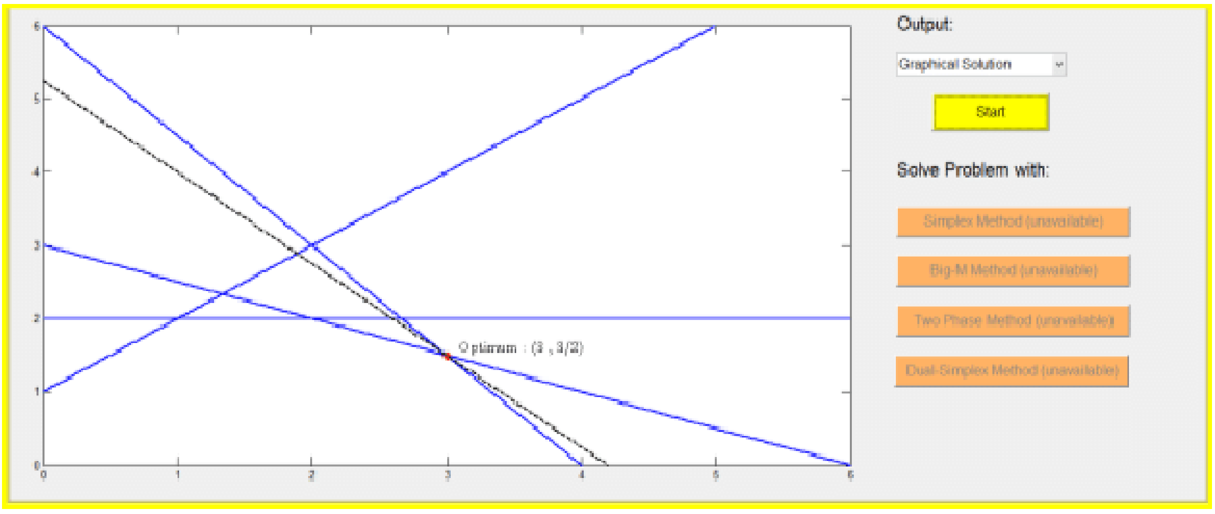

Output:
Graphical Solution
Start
Solve Problem with:
Simplex Method (unavailable)
Big-M Method (unavailable)
Two Phase Method (unavailable)
Dual-Simplex Method (unavailable)
Optimum : (3 , 3/2)

Figure 2.9: LpSolver graphical solution

to operations research specialists. When creating a restriction using gLPS, users are presented with an interface as seen in Figure 2.10. Using this interface one can drag, drop, and connect different components to create a linear programming model.

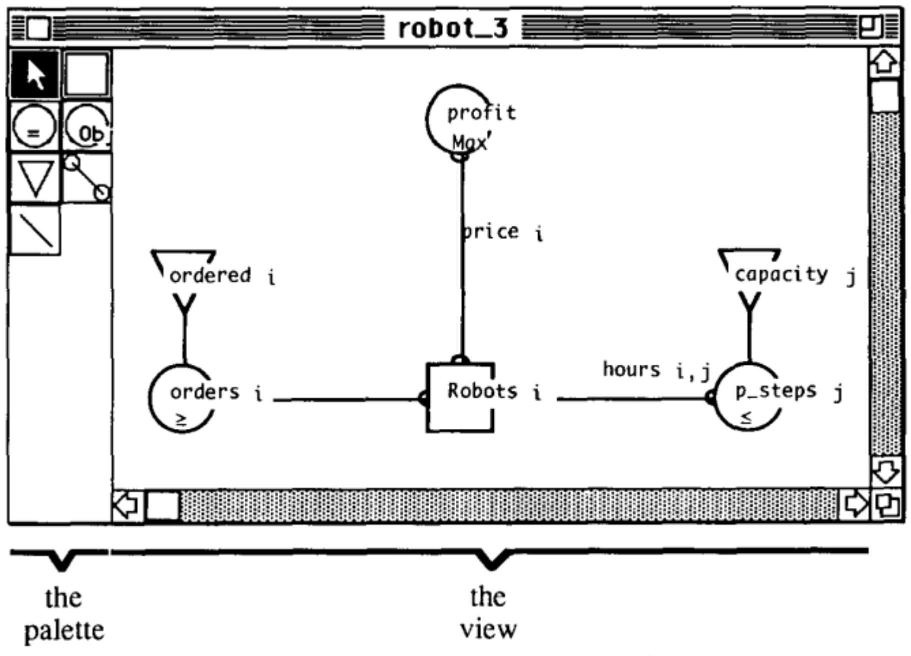


Figure 2.10: gLPS graphical interface

2.3.4 Two-variable Linear Programming: A Graphical Tool with mathematica

Previous work introduced GLP-Tool, a graphical interface designed to help users to understand fundamental concepts of linear programming [19]. The author's goal is to leverage active learning to increase student engagement during the learning process. This tool is a dynamic, interactive, and visual tool that allows solving user-defined linear programming problems with two variables. In particular, the user can explore different objective functions and constraint sets, obtain graphical and numerical information on optimal solutions and intuitively perform post-optimal and sensitivity analysis. The author's focus is clearly on education and teaching the mathematical component behind linear programming and the author justifies some of their choices on classroom requirements.

The interface shown in Figure 2.11 lets the users input the objective and constraints by using selection and sliding inputs, it allows for the definition of non-negativity by choosing the option in the variable, choosing the values of the constants using a sliding option as well as the variables in the objective and constraints. Users can also choose to maximize or minimize the objective. Users can then visualize the feasible area and solution.

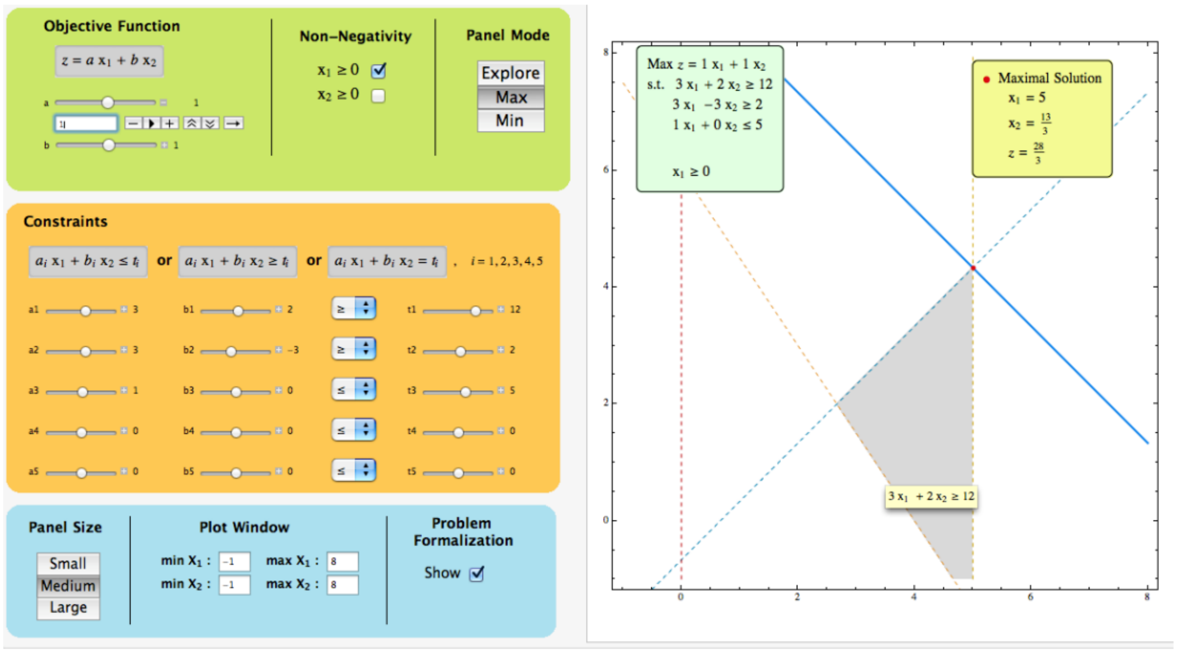


Figure 2.11: GUI linear programming interface for MATLAB

2.3.5 Conclusions

Overall we found that the projects referenced in this Section either have a mathematical approach to linear programming and its methods and do not offer business-focused features like *Excel* does by allowing users to work with data to create the formulations. This is seen in Subsections 2.11 and 2.3.2. Or despite focusing on similar goals to us, their projects possess outdated and not very intuitive interfaces, this is seen in Subsections 2.3.1 and 2.3.3.

Considering the above we believe that taking inspiration from projects that use block-based languages for various other computing tasks could be useful to tackle our problem.

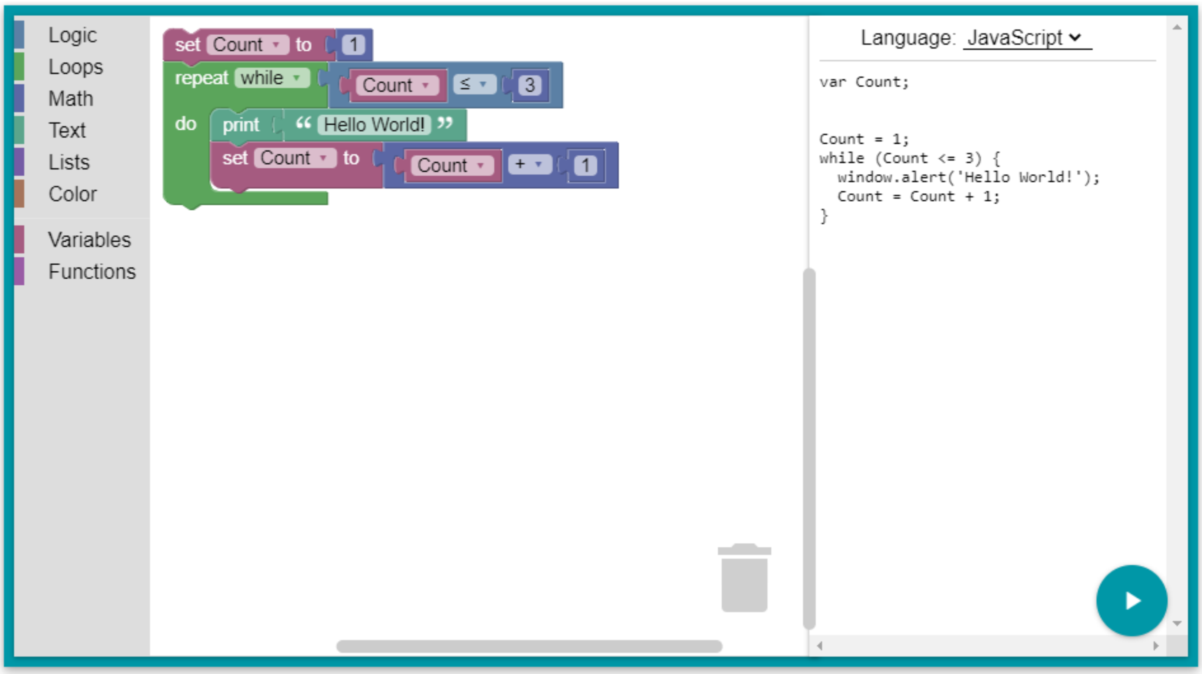
2.4 Frameworks and notable visual languages projects

2.4.1 Blockly

Blockly (whose interface can be seen in Figure 2.12) is a google created library created to build visual applications that output syntactically correct code. This library is written in pure *JavaScript*, is 100% client-side and without server-side dependencies, can be used with all major browsers, and is highly customizable and extendable[10].

Various projects have been built using this technology, having found success particularly in the educational space. These applications include Blockly Games[11], code.org[12], MIT App Inventor[13] and BlockPy[14].

The creation of this library was greatly influential to the field of visual programming languages, not only due to the projects that were built using it, but as well as the insights gained into the difficulties and mistakes made when creating visual programming languages [8] which can be helpful in any project involving block-based languages.



```
var Count;


Count = 1;
while (Count <= 3) {
  window.alert('Hello World!');
  Count = Count + 1;
}
```

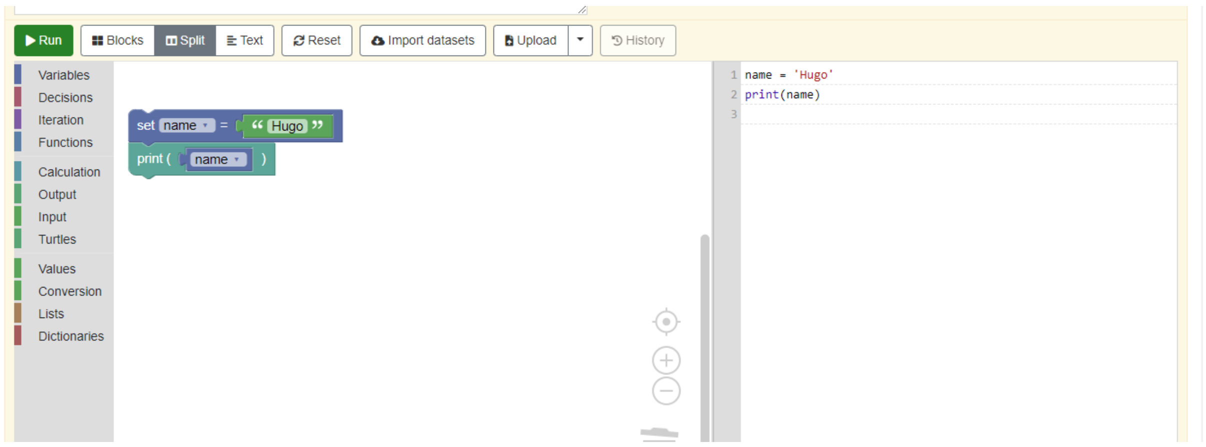
Figure 2.12: A simple Blockly program.

2.4.2 BlockPy

The *BlockPy* (whose interface can be seen in Figure 2.13) project is a web-based, open-access Python programming environment made for introductory programming and data science education. This project

[10] https://developers.google.com/blockly
[11] https://blockly.games/
[12] https://code.org/
[13] https://appinventor.mit.edu/
[14] https://think.cs.vt.edu/blockpy/

came about due to the increasing need that professionals and non-computer science students feel to learn computer science related skills and due to the contextualization of introductory computer science education focusing mostly on game design and media computation thus alienating potential learners due to a perceived pointlessness to their professional activity. Since data science skills such as data processing are widely needed across a variety of fields the subject is more appealing to users from business or other technical fields wanting to change careers than other computing fields [3].



```
name = 'Hugo'
print(name)
```

Figure 2.13: Hello World program using the BlockPy interface

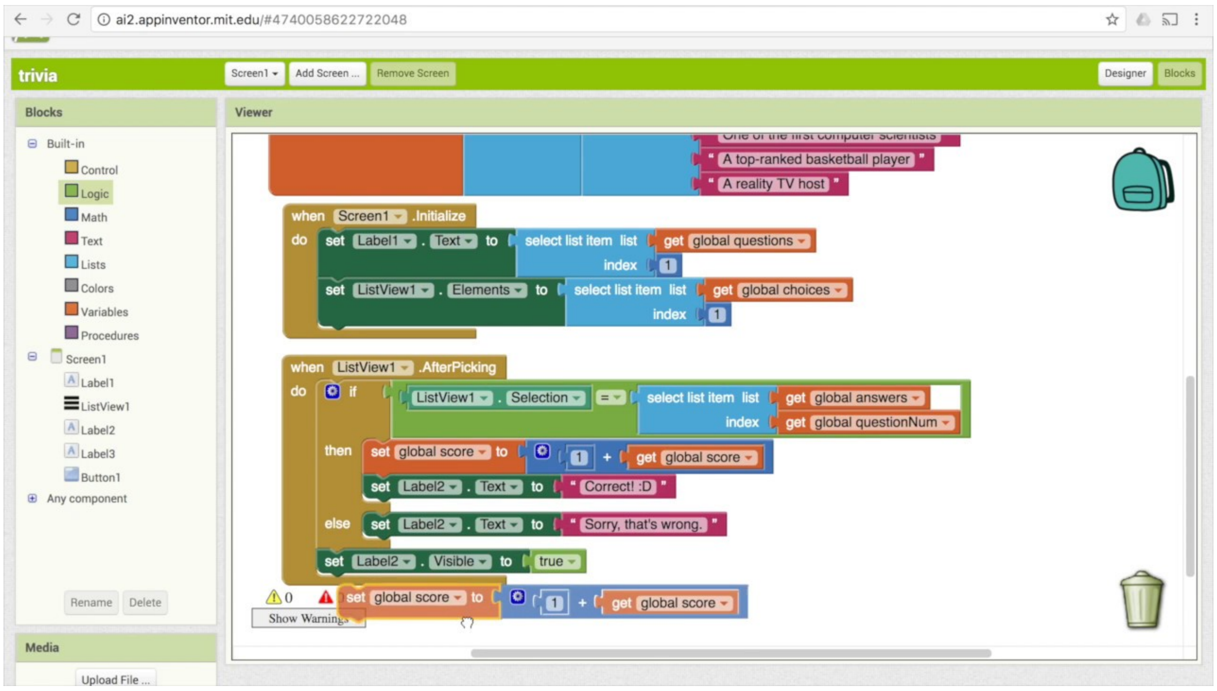
The choice of *Python* for this educational tool was because of its explicit syntax, strong support for data science libraries such as pandas, *Matplotlib*, and *scikit-learn*. Despite its ease of use when compared with other computer languages, using an intermediate visual language was an improvement for the users of the study. This was found to improve the usability of the language for novice programmers. One feature that distinguishes *BlockPy* from other projects allowing users to program in Python using visual languages is the possibility of freely oping between text and visual programming. This feature was found to improve the transition process between the two types of languages [2].

Accessibility is one of *BlockPy's* project goals. For this reason *BlockPy* is an easily accessible web-based platform, all of its code is open source and leverages an array of open-source libraries. The visual language portion of this project was done using the *Blockly* library [3].

2.4.3 MIT app Inventor

The MIT App Inventor, whose user interface can be seen in figure 2.14, is an educational platform that uses android app development to teach introductory computer science concepts. This concept was born from the observation that smartphones play an intrinsic role in our everyday lives. However the majority of people do not understand how the technology they use works. This project tries to solve this problem by providing a more user-friendly approach to building smartphone apps. The interface uses a drag and drop approach for different components and the *Blockly* library for the program logic [18].

This project's contributions to the user interface and visual programming fields stem from the real-world context and applications instead of more traditional programming aspects such as loops, arrays,



```
" One of the first computer scientists "
" A top-ranked basketball player "
" A reality TV host "

when Screen1 .Initialize
do  set Label1 . Text to  select list item  list  get global questions
                                             index 1
    set ListView1 . Elements to  select list item  list  get global choices
                                                   index 1

when ListView1 .AfterPicking
do  if  ListView1 . Selection = select list item  list  get global answers
                                                  index get global questionNum
    then  set global score to  1 + get global score
          set Label2 . Text to " Correct! :D "
    else  set Label2 . Text to " Sorry, that's wrong. "
    set Label2 . Visible to true

set global score to  1 + get global score
```

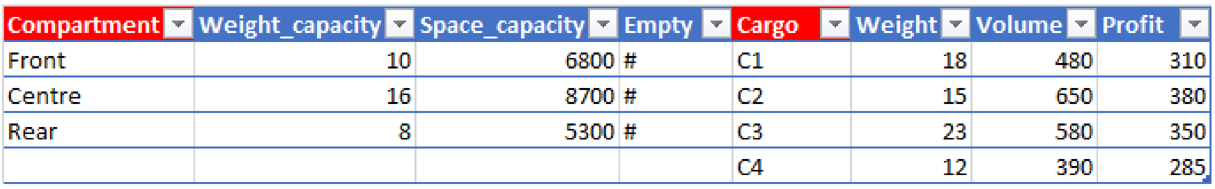
Figure 2.14: Mit app Inventor user interface.

operators, and conditionals [18].

Chapter 3

A block-based language for linear programming

In this Chapter we introduce our proposed language LPBlocks. We will describe how LPBlocks processes the data, which blocks were designed, and how to define the variables, constraints, and linear programming model's objective. Additionally, to make it easier for the reader to understand the language constructs and format, we will use a running example featured in a Master of Business Administration (MBA) exam [4]. This example problem aims to increase the profit of deliveries by airplanes. The problem statement provides values for the weight and space capacity of three different airplane's compartments (front, rear, and center) and maximum values for the weight, volume, and profit for four different cargoes (C1, C2, C3, and C4) as seen in Figure 3.1.



Compartment	Weight_capacity	Space_capacity	Empty	Cargo	Weight	Volume	Profit
Front	10	6800	#	C1	18	480	310
Centre	16	8700	#	C2	15	650	380
Rear	8	5300	#	C3	23	580	350
				C4	12	390	285

Figure 3.1: Input data for the running example problem

3.1 Data structure

Our language requires the input data to follow a specific structure. This structure allows for the definition of *index columns* (as seen highlighted in red in Figure 3.1). These are used to reference values and iterate over the *data columns* (in the blue columns of the same Figure) this being always associated with one index column. To distinguish between the two we assume that the *data columns* addressed by a given *index column* appear in the spreadsheet immediately after the said *index column*, and that different sets of *index* and *data columns* are separated by an column filled with the character # as can be seen in the

Figure (fourth column). In this case there are two sets, the first being for the three plane compartments and the second for the four types of cargo.

3.2 Blocks

The building blocks of the linear programming language we propose can be seen in Figure 3.2. LPBlocks includes:

- *Variable blocks* (seen in Figure 3.2.A): Blocks for creating single, column and matrix variables.
- *Operation block* (seen in Figure 3.2.B): A block to construct an individual constraint or the objective.
- *Building blocks* (seen in Figure 3.2.C): These blocks include two nesting blocks for the `Variables` and `Constraints`, a nesting block to add an individual `Constraint` to a `Constraints` block, and an `Objective` block to define the objective function.
- *Value blocks* (seen in Figure 3.2.D): A set of `value blocks` to access the variables created before.

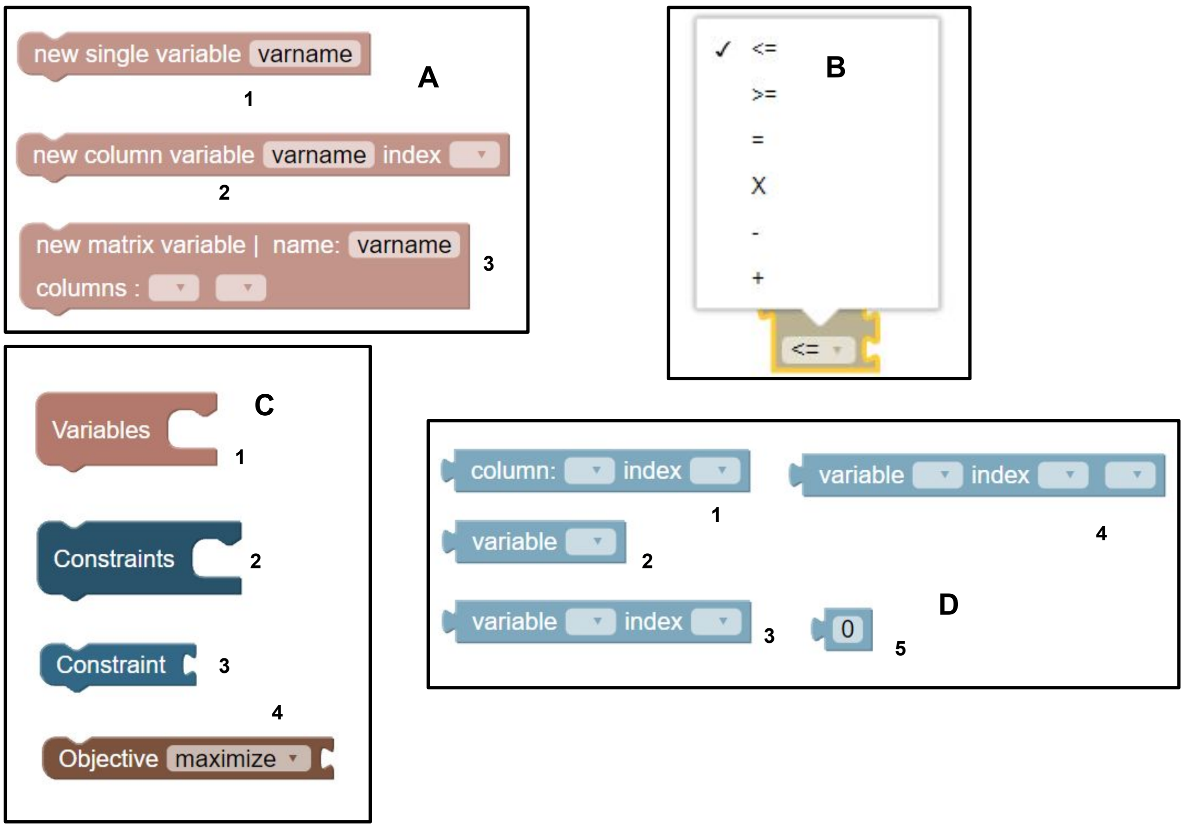
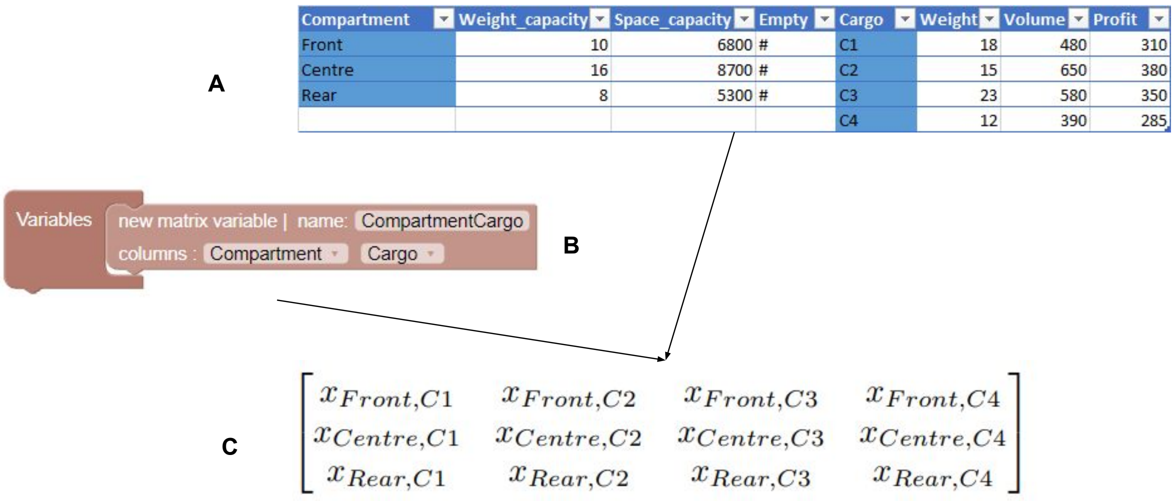


Figure 3.2: Building blocks for LPBlocks

3.3 Defining variables

To define a mathematical linear programming model, considering our running example, one would start by creating a set of variables iterating over the the airplane sections and the cargoes as shown in Figure 3.3.C (we refer to the problem's original website for a more common variable naming). Since this is a very common scenario, LPBlocks includes a construct that can be used to define all these variables which we call a matrix. In Figure 3.3.B we use such a construct to create the variables for the running example. In the example, we use a `matrix variable block` to create a new $N \times M$ matrix variable named `CompartmentCargo`, with N being equal to the length of the column `Compartment` and M to the length of the column `Cargo` with these columns serving as its indexes.



Compartment	Weight_capacity	Space_capacity	Empty	Cargo	Weight	Volume	Profit
Front	10	6800	#	C1	18	480	310
Centre	16	8700	#	C2	15	650	380
Rear	8	5300	#	C3	23	580	350
				C4	12	390	285

Variables
new matrix variable | name: CompartmentCargo
columns : Compartment Cargo

$$\begin{bmatrix} x_{Front,C1} & x_{Front,C2} & x_{Front,C3} & x_{Front,C4} \\ x_{Centre,C1} & x_{Centre,C2} & x_{Centre,C3} & x_{Centre,C4} \\ x_{Rear,C1} & x_{Rear,C2} & x_{Rear,C3} & x_{Rear,C4} \end{bmatrix}$$

Figure 3.3: Creating an example matrix variable

LPBlocks offers several options to define new variables, using the blocks seen in Figure 3.2.A respectively:

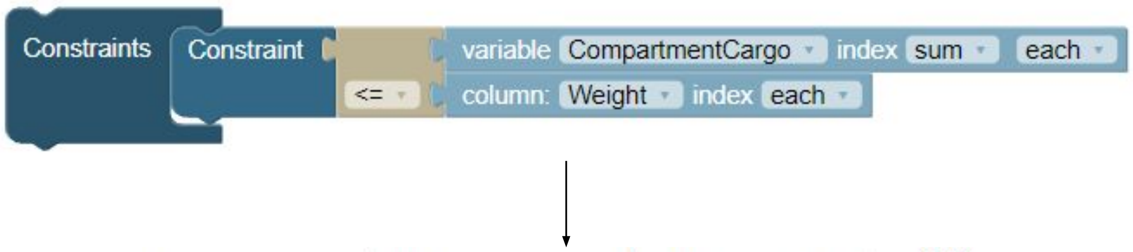
- single variables through its name (as seen in Figure 3.2.A.1);
- column variables defining its name and an index column for which the variable will be iterated and accessed (as seen in Figure 3.2.A.2);
- matrix variables that take a name and two index columns for which they can be iterated and those values accessed ((as seen in Figure 3.2.A.3) and used in Figure 3.3.B).

The process of generating the model variables is dependent on which `variable blocks` we used:

- For the `single variable block`, a variable is generated with the chosen name.
- For `column variable` blocks, an array of variables is created.
- For `matrix variables blocks`, a matrix of variables is created (as shown in Figure 3.3.B).

3.4 Defining constraints

The second step to define the mathematical model would be to create a set of constraints, using the variables created before, and encoding the restrictions of the underlying problem. A constraint of the running example is that one “cannot pack more of each of the four cargoes than their available quantity”. The mathematical encoding would be as shown in Figure 3.4. There are four constraints, one for each cargo. For each constraint, the left-hand side of the inequality displays the sum of variables referring to the corresponding cargo (e.g. C1 for the first constraint) and for the three different airplane sections. On the right-hand side one would write the cargo weight limit.


Constraints Constraint <= variable CompartmentCargo index sum each
column: Weight index each

$$x_{Front,C1} + x_{Centre,C1} + x_{Rear,C1} <= 18$$
$$x_{Front,C2} + x_{Centre,C2} + x_{Rear,C2} <= 15$$
$$x_{Front,C3} + x_{Centre,C3} + x_{Rear,C3} <= 23$$
$$x_{Front,C4} + x_{Centre,C4} + x_{Rear,C4} <= 12$$

Figure 3.4: Defining constraints for the cargoes weight

The first constraint in Figure 3.4 is defined in our language by using: *i)* an `operation block` with the inequality sign <=; *ii)* a `variable block` with the option `CompartmentCargo` and indexes *sum* and *each*; *iii)* a column block with the option `Weight` and index *each*. Since the `constraints block` only appears after the `variables block` the compiler knows the index values for both the column and variable used and thus can generate the correct constraints which in this case are also expressed in Figure 3.4.

In LPBlocks, each constraint is defined by dragging a constraint block inside the constraints block (second and third blocks from the top in Figure 3.2.C) and then using the `value blocks` (blocks in Figure 3.2.D) and `operation blocks` (blocks following the constraint block in Figure 3.2.B) to express the constraints. In our language, `operation blocks` represent relations between blocks and are used to express several operations including arithmetic operations and inequalities. The `value blocks` can represent:

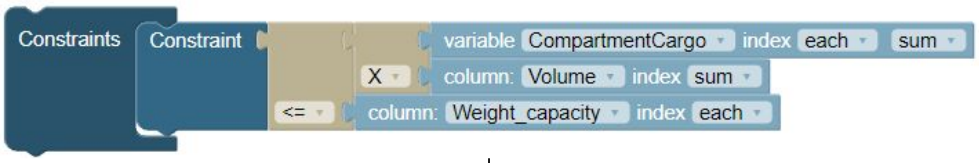
- Columns;
- Previously defined variables;

- Numbers.

The variables can be accessed using the analogous `matrix variable value block` to the `matrix variable creation block` used for its creation. As an example when a user creates a variable using a *matrix variable creation block* (seen in Figure 3.3) a user can only access the created variable using a *matrix variable value block* (as seen in Figure 3.4).

Our solution includes other features such as allowing for the expression of sums and iterations by selecting the option *each* for iterations and *sum* for sums. This can be seen in Figure 3.4 where the user selects the option *each* to generate an iteration of the column `Compartment` and the option *sum* for generating sums of the column `Cargo`.

Another example constraint can be expressed in natural language as “the volume (space) capacity of each compartment must be respected”. This constraint (in Figure 3.5.A) uses `X` (multiplication) and <= `operation blocks` and `value blocks` to express the more complex constraints. This constraint differs from the previous ones since the use of the `X operation block` leads to the generation of *sumproduct* constraints instead of sum. For this constraint, our compiler generates the linear programming constraints featured in Figure 3.5.


Constraints
Constraint
variable CompartmentCargo index each sum
X
column: Volume index sum
<=
column: Weight_capacity index each

$$\downarrow$$

$$480x_{Front,C1} + 650x_{Front,C2} + 580x_{Front,C3} + 390x_{Front,C4} <= 6800$$
$$480x_{Centre,C1} + 650x_{Centre,C2} + 580x_{Centre,C3} + 390x_{Centre,C4} <= 8700$$
$$480x_{Rear,C1} + 650x_{Rear,C2} + 580x_{Rear,C3} + 390x_{Rear,C4} <= 5300$$

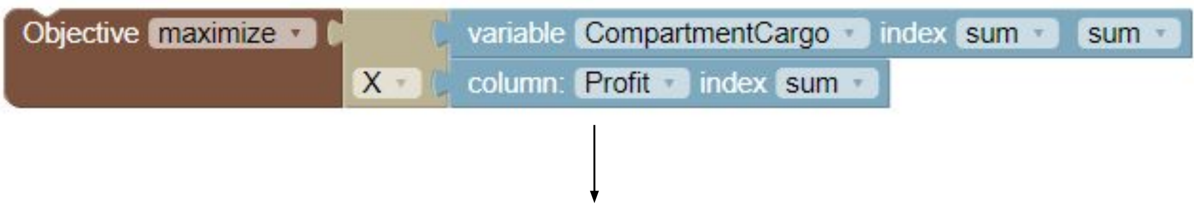
Figure 3.5: Defining constraints for the weight capacity

3.5 Defining the objective function

The final step in a linear programming model is the definition of an objective function. For our running example, one intends to maximize the profit of the airplane usage.

To define the objective function users must connect the `objective block` and the `constraints block` together and use several `value` and `operation blocks`.

In the example seen in Figure 3.6.A the objective function is created by using an `operation block` with value <=, a `column block` with option `Profit`, and a `variable block` with the matrix variable



$$310[x_{Front,C1} + x_{Centre,C1} + x_{Rear,C1}] + 380[x_{Front,C2} + x_{Centre,C2} + x_{Rear,C2}] +$$

$$350[x_{Front,C3} + x_{Centre,C3} + x_{Rear,C3}] + 285[x_{Front,C4} + x_{Centre,C4} + x_{Rear,C4}]$$

Figure 3.6: Objective function for the running example

`Compartment-Cargo`. The objective function generated by this statement is the one featured in Figure 3.6 which would be the one written in a mathematical model.

Chapter 4

Implementation and architecture

To allow LPBlocks to be used we created a web application that lets users interact with the language and use it with their own datasets. When using our tool users can load data from spreadsheets, interact with the data and create linear programming formulations using our language, visualize the model being created in real-time and errors being committed during the construction process and finally run the model and get the results of running the created model by a solver.

Our implementation possesses three main components: the first is a web application, used to provide the user access to our language, and with an interface to select the data, build and run the models. The other components are *APIs* used by the web application. The second component is a *Rest API* to read spreadsheet files and the third a *Rest API* to run the model and get a solution. A component diagram with the architecture of our implementation can be seen in Figure 4.1.

4.1 Web application

For the users to be able to access our language we created a web application that allows users to build linear programming models using our language seen in Figure 4.2.F and Figure 4.2.G. The data loaded from a spreadsheet can be seen in Figure 4.2.E. The users can also run the model and get the results as can be visualized in Figure 4.2.D. Figure 4.2.A, B and C illustrates the mathematical representation of the model created in section G of the same figure.

We chose *React* and *react-blockly* since it was well suited for the highly dynamic data generated when the users load data from spreadsheets, create and run the model. The *Web application* was built using various web technologies. The blocks and workspace were implemented using the *react-blockly* library[1].

[1] https://www.npmjs.com/package/react-blockly

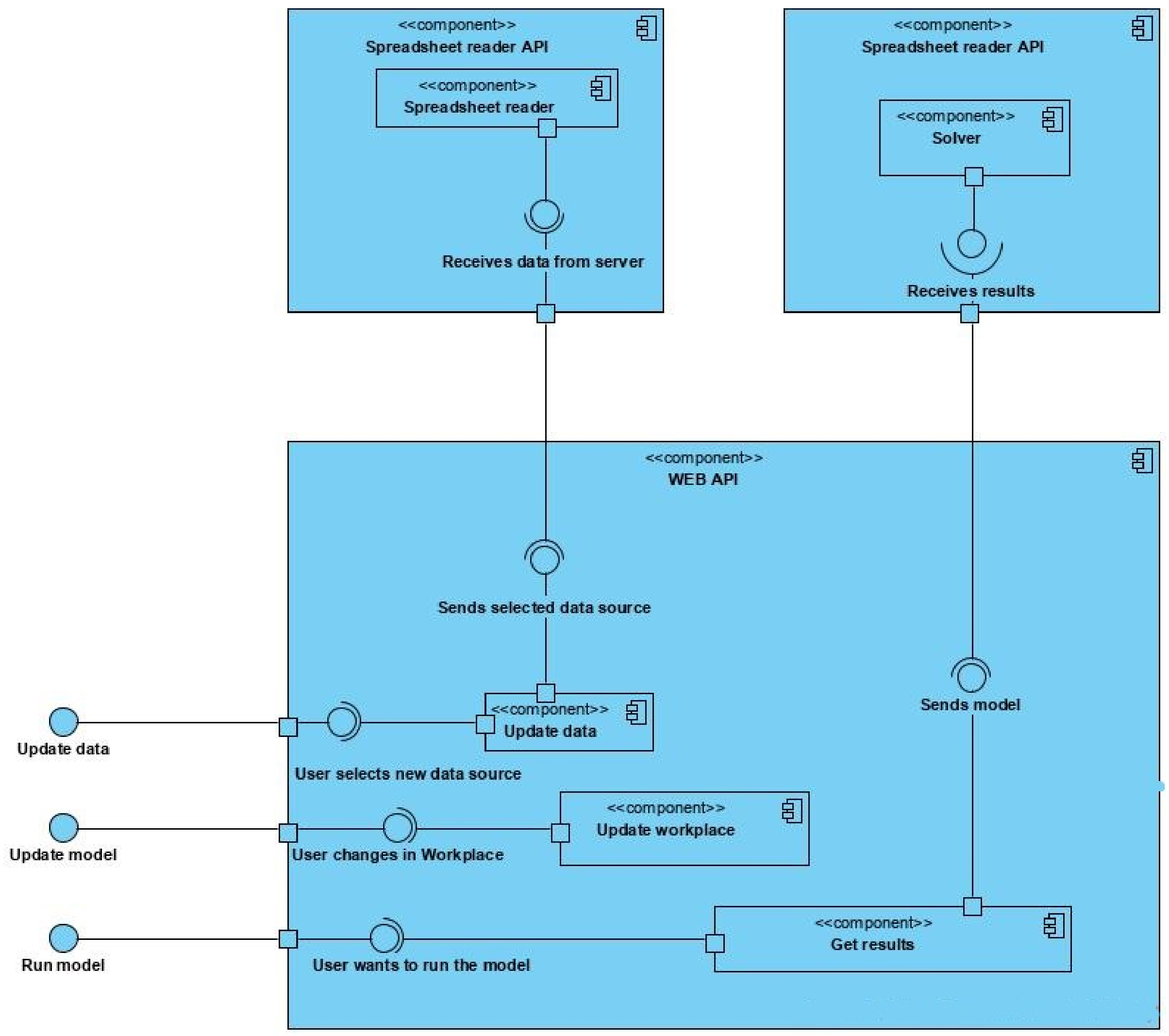
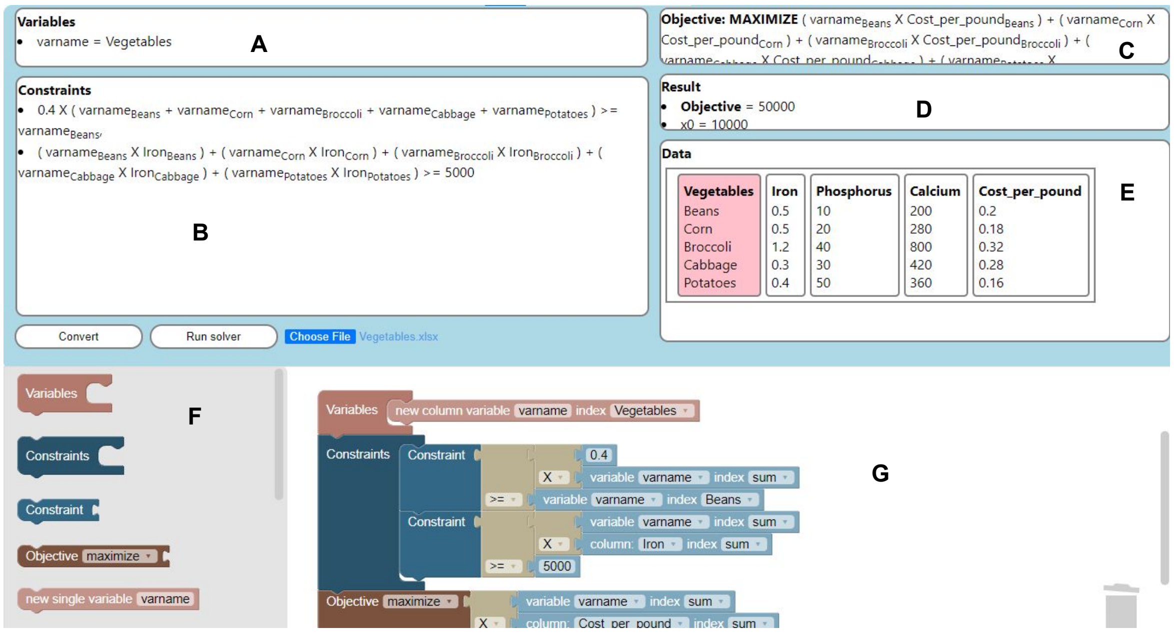


Figure 4.1: Component diagram for our application

The interface was built using *Typescript*[2], *React*[3] and *Redux*[4]. Our goal was to create a highly dynamic and interactive way for users to build their models and to do so we used the previously referred libraries. We decided to use *Typescript* for this project since it offers better reliability when compared to plain *JavaScript* especially when it comes to tasks related to generating the linear programming formulation.

Because of performance issues and to guarantee near-instantaneous feedback for the user when creating the models in the block-based language, we tried to do most of the computing in the front-end since this allows for lower communication costs. It was possible to convert the visual model built by the user to the mathematical formulation, since the code generation component of our LPBlocks *Blockly* implementation was built by us. The running of the model by a solver and the reading of data from the spreadsheets is done on the backend. This was done to better existing libraries in *Node.js* when compared to libraries that run in the browser. The communication between the *Web Application* and the services are done using the *Rest* protocol.

[2]https://www.typescriptlang.org/
[3]https://reactjs.org/
[4]https://redux.js.org/



Variables

- varname = Vegetables

Constraints

- 0.4 X ($\text{varname}_{\text{Beans}}$ + $\text{varname}_{\text{Corn}}$ + $\text{varname}_{\text{Broccoli}}$ + $\text{varname}_{\text{Cabbage}}$ + $\text{varname}_{\text{Potatoes}}$) >= $\text{varname}_{\text{Beans}}$,
- ($\text{varname}_{\text{Beans}}$ X $\text{Iron}_{\text{Beans}}$) + ($\text{varname}_{\text{Corn}}$ X $\text{Iron}_{\text{Corn}}$) + ($\text{varname}_{\text{Broccoli}}$ X $\text{Iron}_{\text{Broccoli}}$) + ($\text{varname}_{\text{Cabbage}}$ X $\text{Iron}_{\text{Cabbage}}$) + ($\text{varname}_{\text{Potatoes}}$ X $\text{Iron}_{\text{Potatoes}}$) >= 5000

Objective: MAXIMIZE ($\text{varname}_{\text{Beans}}$ X $\text{Cost\_per\_pound}_{\text{Beans}}$) + ($\text{varname}_{\text{Corn}}$ X $\text{Cost\_per\_pound}_{\text{Corn}}$) + ($\text{varname}_{\text{Broccoli}}$ X $\text{Cost\_per\_pound}_{\text{Broccoli}}$) + (varname X Cost_per_pound) + (varname X

Result

- **Objective** = 50000
- x0 = 10000

Data

Vegetables	Iron	Phosphorus	Calcium	Cost_per_pound
Beans	0.5	10	200	0.2
Corn	0.5	20	280	0.18
Broccoli	1.2	40	800	0.32
Cabbage	0.3	30	420	0.28
Potatoes	0.4	50	360	0.16

Figure 4.2: Our web application interface

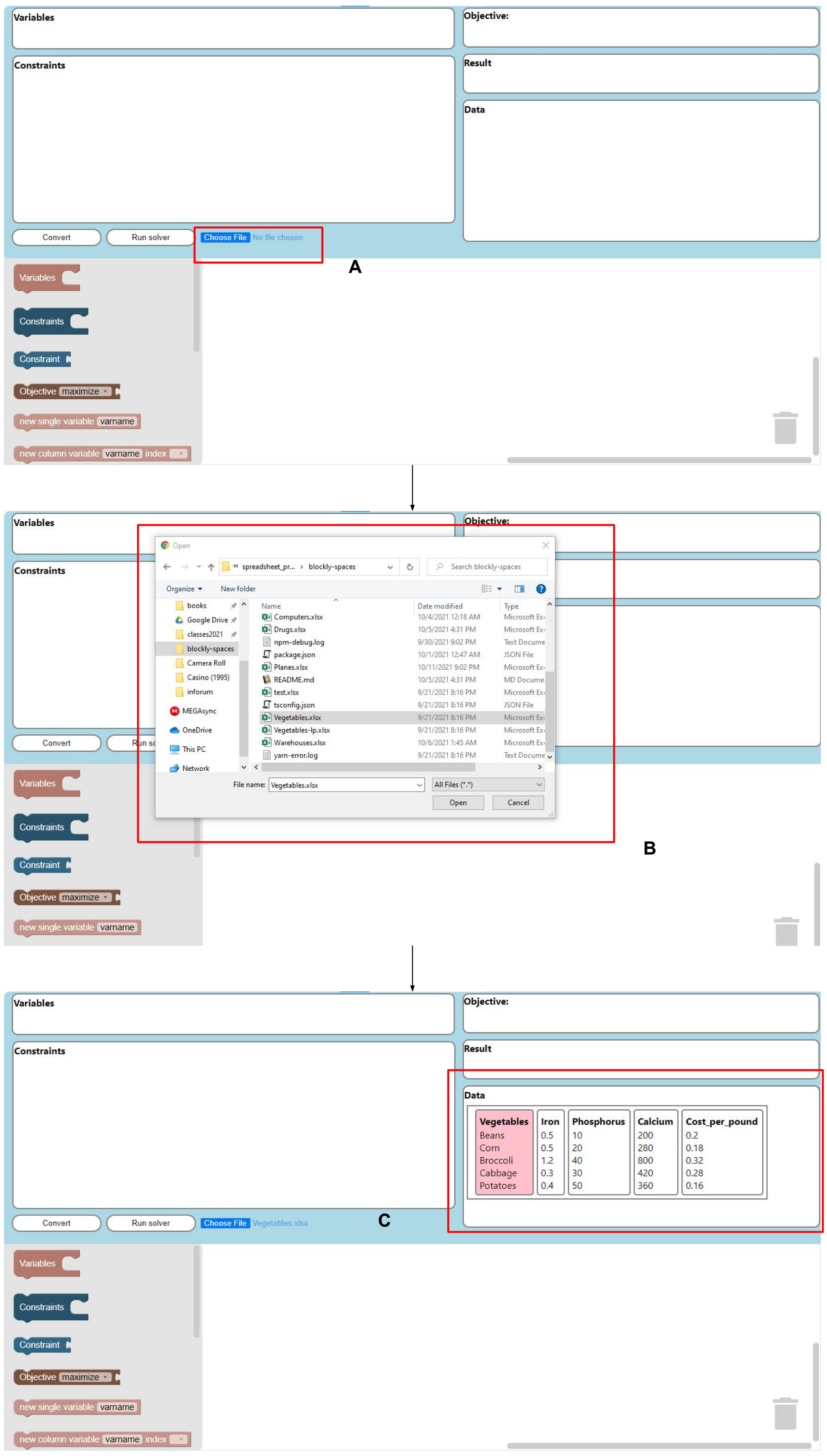
4.1.1 Features

Load and visualize data from spreadsheets One of the features of our interface is the ability that the users have to use spreadsheets to load the data used to build new models. This feature can be used to reuse the model created with compatible data.

As seen in Section 3.1 the data used to build the model is extracted from a compatible spreadsheet and the data is then shown in Figure 4.2.E. The data can be used in the workplace when building the model as seen in Figure 4.2.G.

Dynamic model generation Being able to visualize the generated mathematical linear programming formulation is crucial for users learning how to use our language and to add a layer of safety between the model creation in our language and the execution of the generated model. Users have the ability to visualize the data loaded from spreadsheets using the process show in Figure 4.3, during this process the user first selects a file by clicking in the appropriate button (this is show in Figure 4.3.A), then selecting the file (shown in Figure 4.3.B) and then can visualize the data as seen in Figure 4.3.C. To further improve this process users can visualize the model being created in the workspace (in Figure 4.2.G) using our language in its mathematical formulation (in Figure 4.2.A, Figure 4.2.B and Figure 4.2.C). This feature is also useful for experimentation since users can observe the impact of each change in the workplace in the resulting model and thus can correct possible mistakes and achieve better efficiency when building their models.

The dynamic generation feature can be seen in Figure 4.4 in various steps of building the model. In Figure 4.4.A the user created a `column variable` and can expand on the variables generated by hovering with the mouse the list of generated variables (in the section seen in Figure 4.2.A). The user then



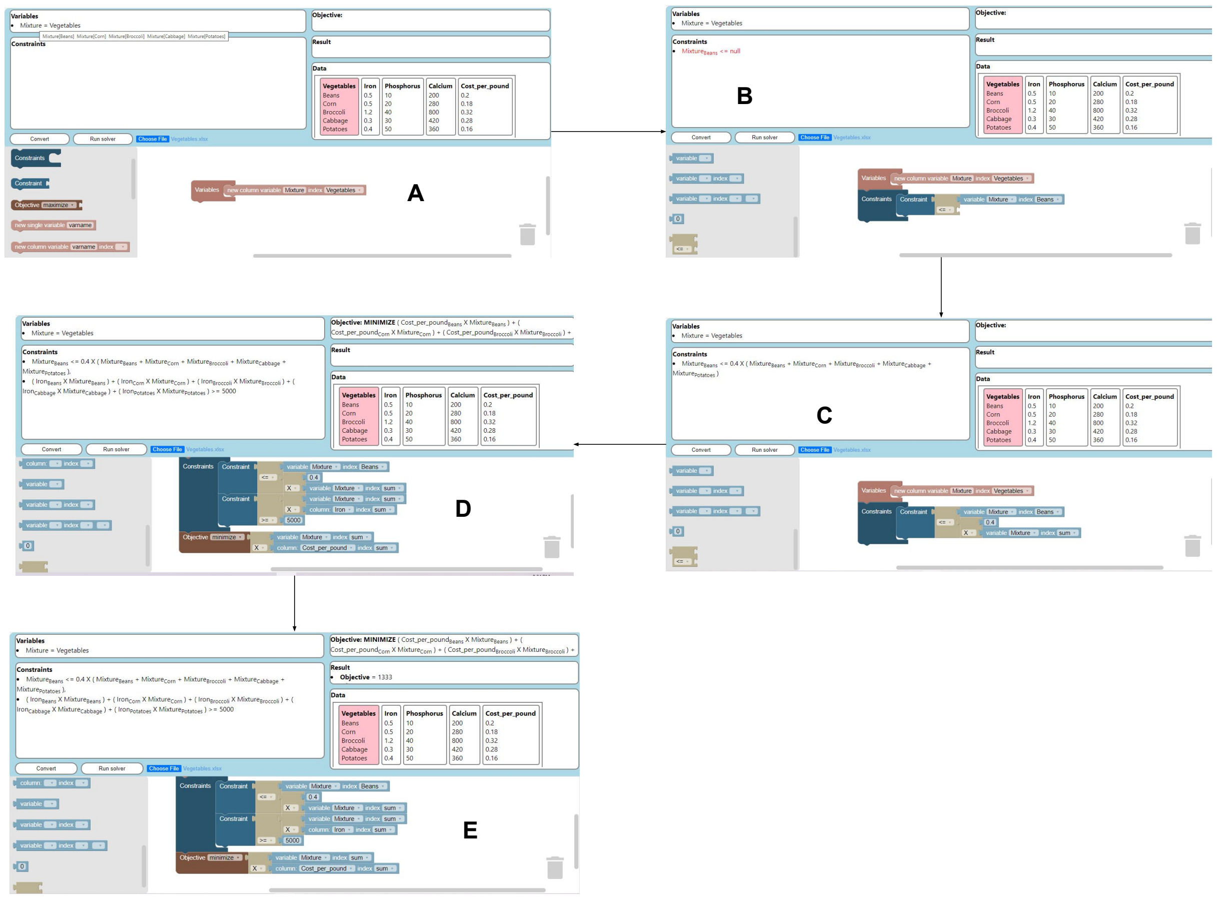
Name	Date modified	Type
Computers.xlsx	10/4/2021 12:18 AM	Microsoft Ex
Drugs.xlsx	10/5/2021 4:31 PM	Microsoft Ex
npm-debug.log	9/30/2021 9:02 PM	Text Docume
package.json	10/1/2021 12:47 AM	JSON File
Planes.xlsx	10/11/2021 9:02 PM	Microsoft Ex
README.md	10/5/2021 4:31 PM	MD Docume
test.xlsx	9/21/2021 8:16 PM	Microsoft Ex
tsconfig.json	9/21/2021 8:16 PM	JSON File
Vegetables.xlsx	9/21/2021 8:16 PM	Microsoft Ex
Vegetables-lp.xlsx	9/21/2021 8:16 PM	Microsoft Ex
Warehouses.xlsx	10/6/2021 1:45 AM	Microsoft Ex
yarn-error.log	9/21/2021 8:16 PM	Text Docume

File name: Vegetables.xlsx

Vegetables	Iron	Phosphorus	Calcium	Cost_per_pound
Beans	0.5	10	200	0.2
Corn	0.5	20	280	0.18
Broccoli	1.2	40	800	0.32
Cabbage	0.3	30	420	0.28
Potatoes	0.4	50	360	0.16

Figure 4.3: Loading data in our application

created the constraints (in Figure 4.4.B and Figure 4.4.C) and the constraints generated by our tool can be seen in real-time. In Figure 4.2.D the user added the objective and in Figure 4.4.E the user ran the model and received the results.



Variables
- Mixture = Vegetables

Constraints
- $Mixture_{Beans}$ <= 0.4 X ($Mixture_{Beans}$ + $Mixture_{Corn}$ + $Mixture_{Broccoli}$ + $Mixture_{Cabbage}$ + $Mixture_{Potatoes}$).
- ($Iron_{Beans}$ X $Mixture_{Beans}$) + ($Iron_{Corn}$ X $Mixture_{Corn}$) + ($Iron_{Broccoli}$ X $Mixture_{Broccoli}$) + ($Iron_{Cabbage}$ X $Mixture_{Cabbage}$) + ($Iron_{Potatoes}$ X $Mixture_{Potatoes}$) >= 5000

Objective: MINIMIZE ($Cost\_per\_pound_{Beans}$ X $Mixture_{Beans}$) + ($Cost\_per\_pound_{Corn}$ X $Mixture_{Corn}$) + ($Cost\_per\_pound_{Broccoli}$ X $Mixture_{Broccoli}$) +

Result
- **Objective** = 1333

Data

Vegetables	Iron	Phosphorus	Calcium	Cost_per_pound
Beans	0.5	10	200	0.2
Corn	0.5	20	280	0.18
Broccoli	1.2	40	800	0.32
Cabbage	0.3	30	420	0.28
Potatoes	0.4	50	360	0.16

Figure 4.4: Dynamic model generation in our application

Error messages Another feature of our tool is the addition of dynamic error checking and messages, our goal with this feature is to help novice and more experienced users debug the constructed models as well as helping users learn our language and catch mistakes commonly associated with creating linear programming models and in using block-based languages. LPBlocks offers the following error messages:

- **Column with the given name already exists** (Figure 4.5a): This error is given when the user attempts to create a variable with a name already used for an existing column. When outputting this message the variable generated by our model is colored red and the user can visualize more information about the error by hovering the mouse above the given variable.

- **Missing inequation in constraint** (Figure 4.5b): This error is given when a user attempts to create a constraint without using an `operation block` with an inequation option.

- **More than one inequation in constraint** (Figure 4.5c): This error is generated when the user attempts to create a constraint with two or more inequalities. This error is shown by highlighting the constraints associated with this attempt in red with a more descriptive message appearing when the user hovers the constraints.

- **Null values in expression** (Figure 4.5d): This message is given when an expression contains Null values.

- **Variable multiplication** (Figure 4.5e): This error is generated when the user attempts to multiply two variables together. This message consists of a red highlight of the affected expressions with a more descriptive message consisting of the names of the variables that the user is attempting to multiply.
- **Empty fields** (Figure 4.5f): This error is outputted when the user did not select the field in a selectable block.
- **Inequation in objective** (Figure 4.5g): Given when the user attempts to use an inequation when creating the objective.
- **Each used in objective** (Figure 4.5h): When the user attempts to use the option `each` in an objective.

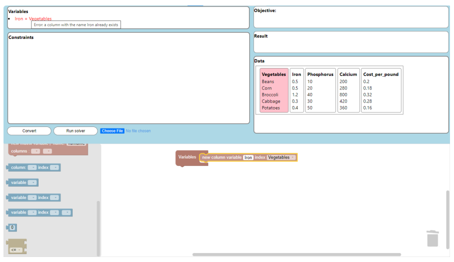
4.1.2 Compilation process

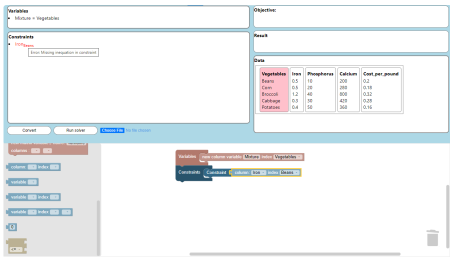
In this Subsection, we elaborate on the different aspects of the compilation process used to generate the linear programming mathematical formulations from our language. In Paragraph 4.1.2 we explain how the variables are generated from a block statement and the data coming from a spreadsheet. We then, in Paragraph 4.1.2, explain how we generate the constraints and the objective function.

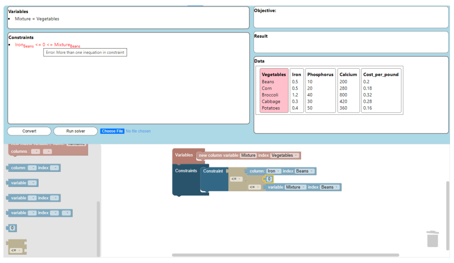
Generating variables To generate the variables into our internal representation when reading a `variable block`, our compilation process consists in reading the name of the given variable and depending on the blocks used we either: *i)* stop there for a `single variable block`, (*ii)*) read the variables index columns. The compiler then sets the variable name as a key to a list with the index columns used to create the variables as its value. The individual variables created with various iterations are computed when necessary.

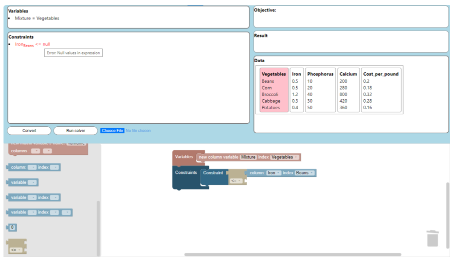
The process of computing the variables when given their name consists in retrieving the index columns array for the given variable, and calculating all possible permutations for the contents of the columns. An example of the outcome of this process can be visualized in Figure 3.3, in which the it was used a `Matrix Variable block` to define the variable and our compiler then iterates over both the columns passed as the input to generate the variables.

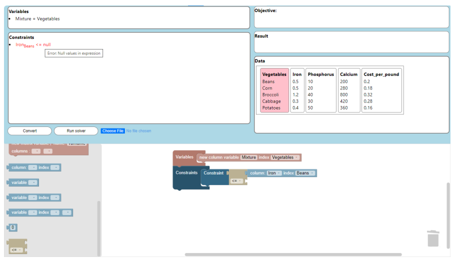
Generating the constraints and objective The process of generating the constraints is not as forward as for generating the variables since creating the constraints in our language involves more blocks and complex operations that support polymorphism such as `operation blocks` with the `X` option. The first step to compute a constraint from our language to the mathematical formulation is to compute the `value blocks`, the output of the `value blocks` is dependent on the block used and its options and requires that information relative to some of the operations passes from the `value blocks` to the `operation blocks`, this information consists in indications for generating sums and iterations. To do so when accessing a particular value of a given column or variable we represent the value in our internal representation.

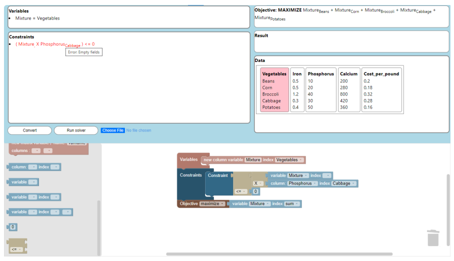

(a) Column with the given name already exists

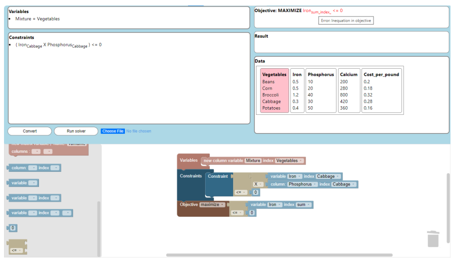

(b) Missing inequation in constraint


(c) More than one inequation in constraint


(d) Null values in expression


(e) Variable multiplication


(f) Empty fields


(g) Inequation in objective

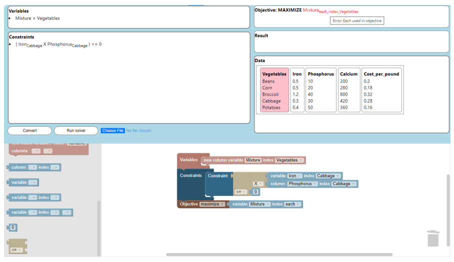
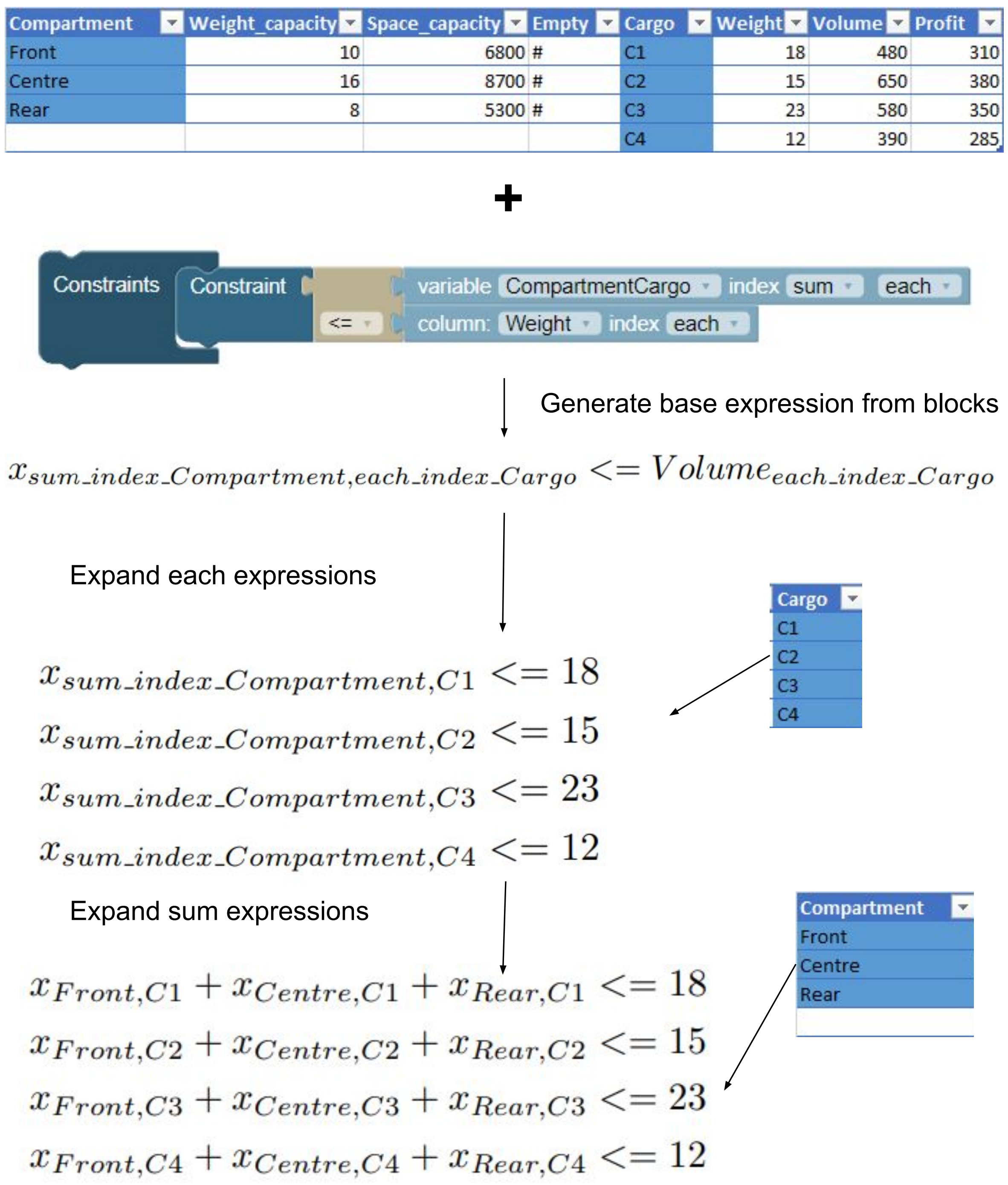

(h) Each used in objective

Figure 4.5: LPBlocks dynamic error messages

If the user selects `sum` or `each` our internal representation contains indications that the user selected this option in the position of the index column for the value selected. We expand the `each` operation at the moment of parsing the inequation. This is due to the possibility that an iteration is applied to more than one variable or column and when this happens the compiler generates a single iteration. We expand the `sum` when parsing an operation or the objective since this operation can be used in more than one context that changes its meaning such as using it as part of a `operation block` with the option *X* to create a *sumproduct* (as seen in Figure 3.4) of two columns or one column and one variable. Figure 4.6 showcases the compilation process of block that uses an `each` and `sum` operation.

After compiling the `value blocks` to their respective location, sum, or iterations of a column or variable, we then compile the different `operation blocks`. The compilation of the `operation blocks` differs from the given operations: for the inequalities we first retrieve the different `each` indication and



Compartment	Weight_capacity	Space_capacity	Empty	Cargo	Weight	Volume	Profit
Front	10	6800	#	C1	18	480	310
Centre	16	8700	#	C2	15	650	380
Rear	8	5300	#	C3	23	580	350
				C4	12	390	285

+

Constraints
Constraint
<=
variable CompartmentCargo index sum each
column: Weight index each
Generate base expression from blocks

$$x_{sum\_index\_Compartment,each\_index\_Cargo} <= Volume_{each\_index\_Cargo}$$

Cargo
C1
C2
C3
C4

$$x_{sum\_index\_Compartment,C1} <= 18$$
$$x_{sum\_index\_Compartment,C2} <= 15$$
$$x_{sum\_index\_Compartment,C3} <= 23$$
$$x_{sum\_index\_Compartment,C4} <= 12$$

Compartment
Front
Centre
Rear

$$x_{Front,C1} + x_{Centre,C1} + x_{Rear,C1} <= 18$$
$$x_{Front,C2} + x_{Centre,C2} + x_{Rear,C2} <= 15$$
$$x_{Front,C3} + x_{Centre,C3} + x_{Rear,C3} <= 23$$
$$x_{Front,C4} + x_{Centre,C4} + x_{Rear,C4} <= 12$$

Figure 4.6: Defining constraints - second example from Chapter 3

use them to generate the multiple constraints as seen in Algorithm 4.2 by first treating them as a set and generating constraints for all the permutations of the given columns. When generating the operation performed using the X option, the operation computed is dependent on the input given. Thus, if we are given either single values on both sides or a single value on one side and multiple values on the other, the single value is multiplied by each of the multiple values. When multiplying both values generated with the option `sum` a *sumproduct* as seen in Algorithm 4.3 operation is generated. We opted for this option relative to doing a multiplication since we considered that it would be the most intuitive for our target audience.

For the + and - operation options we use the first input as the left side statement and the second as the right side statement. When a `sum` is not expanded as a *sumproduct* operation; it is replaced by a sum of the different values of the given index column as seen in Algorithm 4.1. The way we compile the objective is similar to the constraints, differentiating in not allowing for the use of `each` in `value blocks` and inequalities in the `operation blocks`.

Algorithm 4.1 Expand sum expression

```
1:  function expand_sum_expression($expr$)
2:      $Values_{sum} \leftarrow finds\ columns\ associated\ with\ the\ sum\ keywords$
3:      $new_{expr} \leftarrow expr$
4:      for $value\ of\ value_{each}$ do
5:          $col_{value} \leftarrow get\ column\ data\ for\ column\ value$
6:          $sum_{elements} \leftarrow []$
7:          for $cell\ of\ col_{value}$ do
8:              $sum_{elements}.add(cell)$
9:          end for
10:         $new_{expr} \leftarrow new_{expr}.replace(''sum'' + value, sum_{elements}.join('' + ''))$
11:     end for
        return $new_{expr}$
12: end function
```

Algorithm 4.2 Expand each expression

```
1:  function expand_each_expression($expr$)
2:      $Values_{each} \leftarrow finds\ columns\ associated\ with\ the\ each\ keywords$
3:      $new_{exprs} \leftarrow []$
4:      for $value\ of\ value_{each}$ do
5:          $col_{value} \leftarrow get\ column\ data\ for\ column\ value$
6:          $new_{exprs}.add(expr.replace(''each'' + value, cell))$
7:      end for
        return $new_{exprs}$
8:  end function
```

Algorithm 4.3 Generate sumproduct

```
1:  function generate_sumproduct($expr,$)
2:      $Values_{each} \leftarrow finds\ columns\ associated\ with\ the\ each\ keywords$
3:      $new_{exprs} \leftarrow []$
4:      for $value\ of\ value_{each}$ do
5:          $col_{value} \leftarrow get\ column\ data\ for\ column\ value$
6:          $new_{exprs}.add(expr.replace(''each'' + value, cell))$
7:      end for
        return $new_{exprs}$
8:  end function
```

4.2 Spreadsheet reading service

This service was created as a *Rest API* using *Express.js*[5] that accepts the upload of a given spreadsheet file and then reads it using the *node-xlsx*[6] library and sends back the contents of spreadsheet in *JSON* format as seen in Appendix A.

To translate the data in the spreadsheet into our internal representation used to generate the constraints (this process can be seen in Figure 4.7) we load the data from the file into memory in the *JSON* format. Subsequently, we parse the *JSON* data into our internal representation. This process comprises the following steps:

- Finding the index columns as seen in Algorithm 4.4: we do this by iterating over the list of columns names and selecting the first column and the first ones after an "Empty" column as seen in Algorithm 4.4
- Finding the indexes for each column as seen in Algorithm 4.5: to do this we iterate over the column names and set each column index column as the previous index column.
- Finding the column data as seen in Algorithm 4.6: to do this task we first iterate over the column names and set the data of the column to its contents in the *JSON* readings.

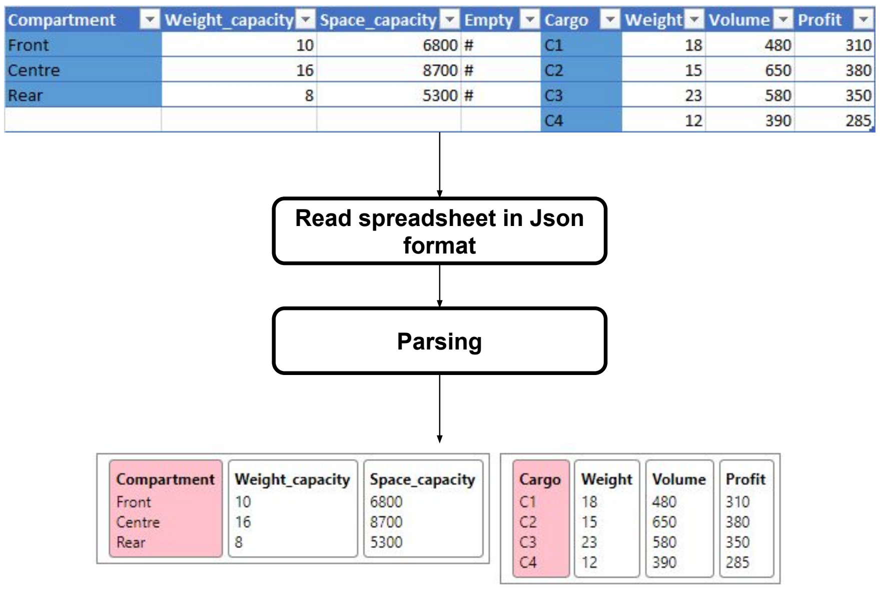
Listing 4.1: Input for the Solver API

```
{
  model: `Maximize
      obj: + 0.6 x1 + 0.5 x2
      Subject To
      cons1: + x1 <= 1
      cons2: + 3 x1 + x2 <= 2
      End`
}
```

Algorithm 4.6 Find column data

```
1: function find_column_data($cols$)
2:     $column_{data} \leftarrow Map < string, string > ()$
3:     for $col\ of\ cols$ do
4:         $column_{data} \leftarrow cols[col]$
5:     end for
       return $column_{indexes}$
6: end function
```

[5] https://expressjs.com/
[6] https://www.npmjs.com/package/node-xlsx



Compartment	Weight_capacity	Space_capacity	Empty	Cargo	Weight	Volume	Profit
Front	10	6800	#	C1	18	480	310
Centre	16	8700	#	C2	15	650	380
Rear	8	5300	#	C3	23	580	350
				C4	12	390	285

Compartment	Weight_capacity	Space_capacity
Front	10	6800
Centre	16	8700
Rear	8	5300

Cargo	Weight	Volume	Profit
C1	18	480	310
C2	15	650	380
C3	23	580	350
C4	12	390	285

Figure 4.7: Input data for the running example problem

Algorithm 4.4 Find index columns

```
 1: function find_indexes($cols$)
 2:     $index_{cols} \leftarrow []$
 3:     $index \leftarrow True$
 4:     for col of colnames do
 5:         if $index == True$ then
 6:             $index_{cols} += col$
 7:             $index \leftarrow False$
 8:         end if
 9:         if $col == "Empty"$ then
10:             $index \leftarrow True$
11:         end if
12:     end for
        return $index_{cols}$
13: end function
```

4.3 Optimization service

The optimization service uses *Express.js* once again to create a *Rest API*. This *API* receives a *JSON* structure that includes the model in a format that the solver can run. The *API* then runs the model using the *clp-wasm*[7] library and sends the output back. This library is a port of the *COIN-OR* linear programming solver to *WebAssembly*. To use this service we also transform the formulation in our web application internal structure to the format seen in Listing 4.1.

[7] https://www.npmjs.com/package/clp-wasm

Algorithm 4.5 Find column indexes

```
1:  function find_column_indexes($cols$)
2:      $column_{indexes} \leftarrow Map < string, string > ()$
3:      $index \leftarrow True$
4:      $index_{column} \leftarrow$ ""
5:      for $col\ of\ cols$ do
6:          if $index == True$ then
7:              $column_{indexes}[col] = index$
8:              $index \leftarrow False$
9:          end if
10:         if $col ==$ "$Empty$" then
11:             $index \leftarrow True$
12:         end if
13:     end for
        return $column_{indexes}$
14: end function
```

Chapter 5

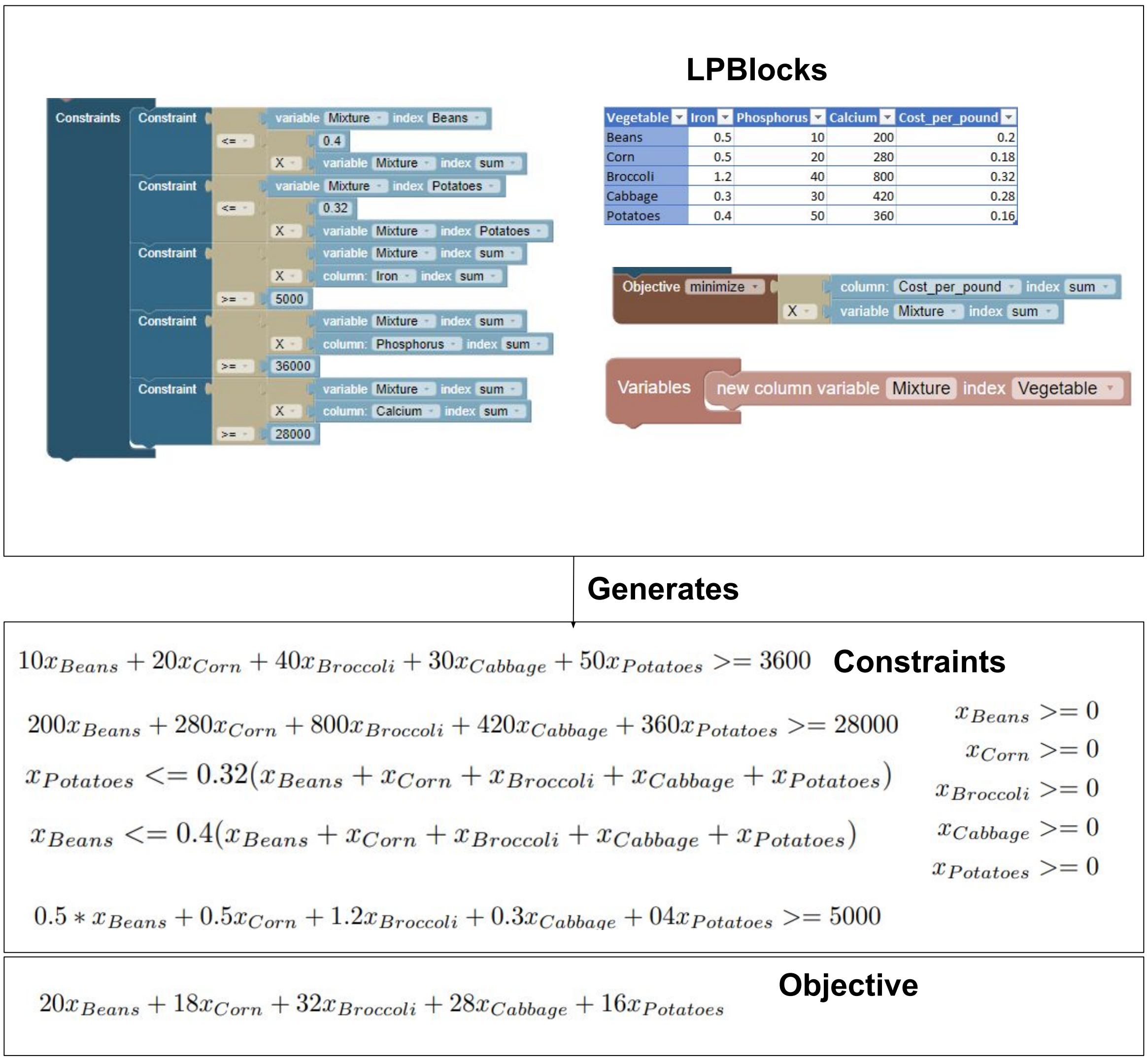
Language applicability

In this chapter we intend to illustrate the applicability of our language to a broad set of examples. Doing so gives us some assurance that our language could be used as a replacement of other linear programming tools, still allowing users to solve their problems. We present a set of linear programming problems mostly taken from an MBA exam [4] and from an Operations Research textbook [5], and their modeling using LPBlocks. Our goal when choosing those problems was to have a set of problems representative of different linear programming workloads, difficulty levels and size, that would allow us to test and assess some possible shortcomings of LPBlocks.

5.1 Vegetable mixture

This example shown in Figure 5.1 comes from the operations research textbook [5]. In this example, a manufacturer of freeze-dried vegetables aims at reducing production costs while adhering to various nutrition criteria and guidelines. We are given nutritional data for each of the vegetables as well as their cost per pound in the tabular data shown in Figure 5.1. We have a maximum percentage for certain vegetables and lower bounds for certain nutrients.

Since our goal is to find the ratio of each vegetable that goes into the mixture, we created a `column variable` named `Mixture` that takes as its input the column `Vegetable`. For the constraints we start by creating constraints imposing limits of 40% (represented by the multiplication of 0.4) for beans and 32% for potatoes (represented by the multiplication of 0.32). The following three constraints define the lower bounds for the given nutrients. The objective is to minimize the cost per pound of the mixture.


LPBlocks
Constraints
Constraint
variable Mixture index Beans
<=
0.4
X
variable Mixture index sum
variable Mixture index Potatoes
0.32
variable Mixture index Potatoes
variable Mixture index sum
column: Iron index sum
>=
5000
column: Phosphorus index sum
36000
column: Calcium index sum
28000
Objective minimize
column: Cost_per_pound index sum
variable Mixture index sum
Variables
new column variable Mixture index Vegetable
Generates
Constraints
Objective

Vegetable	Iron	Phosphorus	Calcium	Cost_per_pound
Beans	0.5	10	200	0.2
Corn	0.5	20	280	0.18
Broccoli	1.2	40	800	0.32
Cabbage	0.3	30	420	0.28
Potatoes	0.4	50	360	0.16

$$10x_{Beans} + 20x_{Corn} + 40x_{Broccoli} + 30x_{Cabbage} + 50x_{Potatoes} >= 3600$$

$$200x_{Beans} + 280x_{Corn} + 800x_{Broccoli} + 420x_{Cabbage} + 360x_{Potatoes} >= 28000$$

$$x_{Potatoes} <= 0.32(x_{Beans} + x_{Corn} + x_{Broccoli} + x_{Cabbage} + x_{Potatoes})$$

$$x_{Beans} <= 0.4(x_{Beans} + x_{Corn} + x_{Broccoli} + x_{Cabbage} + x_{Potatoes})$$

$$0.5 * x_{Beans} + 0.5x_{Corn} + 1.2x_{Broccoli} + 0.3x_{Cabbage} + 04x_{Potatoes} >= 5000$$

$$x_{Beans} >= 0 \quad x_{Corn} >= 0 \quad x_{Broccoli} >= 0 \quad x_{Cabbage} >= 0 \quad x_{Potatoes} >= 0$$

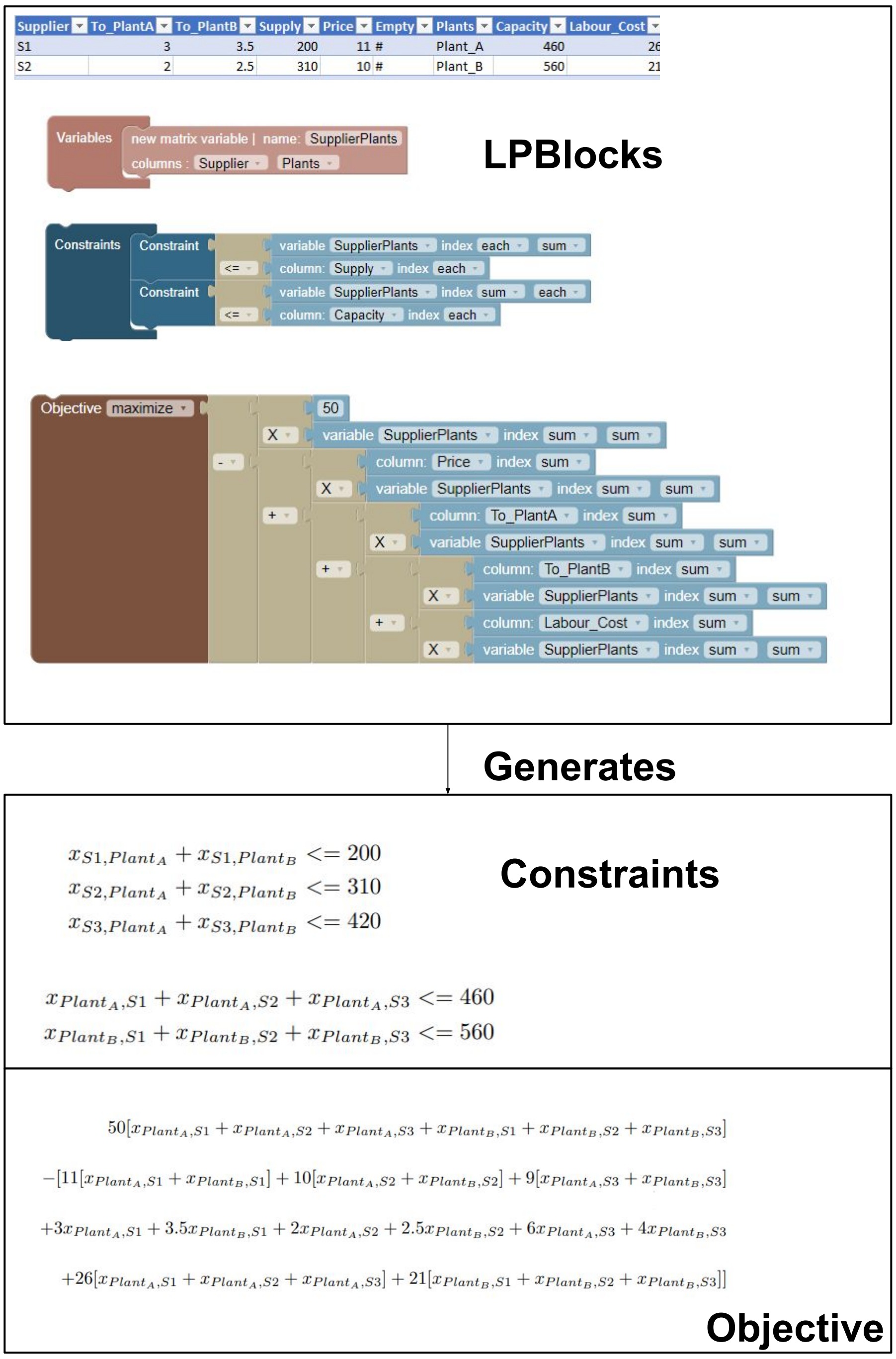
$$20x_{Beans} + 18x_{Corn} + 32x_{Broccoli} + 28x_{Cabbage} + 16x_{Potatoes}$$

Figure 5.1: Definition of the vegetable mixture problem using LPBlocks

5.2 Fruit canning plants

In this example taken from an MBA exam [4] and shown in Figure 5.2 we are given information associated with different suppliers and fruit canning plants with the goal of maximizing its profits. The information includes shipping, labor and operating costs, buying prices and maximum production capacities. Despite not being in the spreadsheet, the problem definition states that the selling price for each tonne is $50.

To generate the formulation we create a `matrix variable` with indexes `Supplier` and `Plant`. The constraints for this problem are straightforward and can be generically specified, this consisting of the upper bounds for the supply and capacity for each of the plants. The objective function is more complex since it needs to take into account the selling price and all the costs to represent the profit.



Supplier	To_PlantA	To_PlantB	Supply	Price	Empty	Plants	Capacity	Labour_Cost
S1	3	3.5	200	11	#	Plant_A	460	26
S2	2	2.5	310	10	#	Plant_B	560	21

LPBlocks
Variables
new matrix variable | name: SupplierPlants
columns : Supplier Plants
Constraints
Constraint
variable SupplierPlants index each sum
<=
column: Supply index each
Constraint
variable SupplierPlants index sum each
<=
column: Capacity index each
Objective maximize
50
X
variable SupplierPlants index sum sum
-
column: Price index sum
X
variable SupplierPlants index sum sum
+
column: To_PlantA index sum
X
variable SupplierPlants index sum sum
+
column: To_PlantB index sum
X
variable SupplierPlants index sum sum
+
column: Labour_Cost index sum
X
variable SupplierPlants index sum sum
Generates
Constraints
Objective

$$x_{S1,Plant_A} + x_{S1,Plant_B} <= 200$$
$$x_{S2,Plant_A} + x_{S2,Plant_B} <= 310$$
$$x_{S3,Plant_A} + x_{S3,Plant_B} <= 420$$

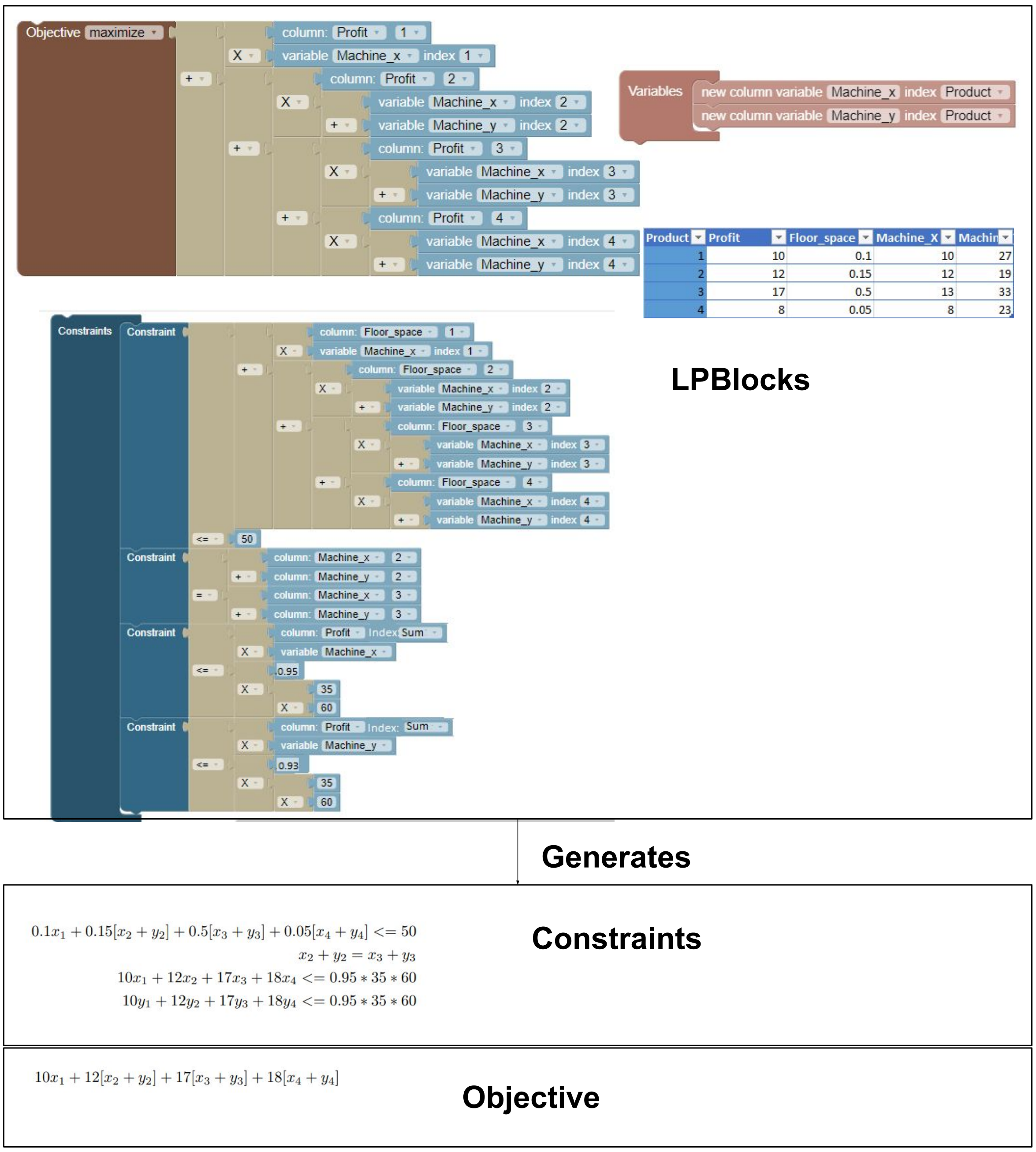
$$x_{Plant_A,S1} + x_{Plant_A,S2} + x_{Plant_A,S3} <= 460$$
$$x_{Plant_B,S1} + x_{Plant_B,S2} + x_{Plant_B,S3} <= 560$$

$$\begin{aligned}
&50[x_{Plant_A,S1} + x_{Plant_A,S2} + x_{Plant_A,S3} + x_{Plant_B,S1} + x_{Plant_B,S2} + x_{Plant_B,S3}]\\
&-[11[x_{Plant_A,S1} + x_{Plant_B,S1}] + 10[x_{Plant_A,S2} + x_{Plant_B,S2}] + 9[x_{Plant_A,S3} + x_{Plant_B,S3}]\\
&+3x_{Plant_A,S1} + 3.5x_{Plant_B,S1} + 2x_{Plant_A,S2} + 2.5x_{Plant_B,S2} + 6x_{Plant_A,S3} + 4x_{Plant_B,S3}\\
&+26[x_{Plant_A,S1} + x_{Plant_A,S2} + x_{Plant_A,S3}] + 21[x_{Plant_B,S1} + x_{Plant_B,S2} + x_{Plant_B,S3}]]
\end{aligned}$$

Figure 5.2: Fruit canning plant example modeled using LPBlocks

5.3 Machine allocation

In the problem shown in Figure 5.3 (taken from [4]) the goal is to maximize a factory's profit by allocating the production of different goods among two machines. In this problem we are given information about each product's profitability, use of floor space and manufacturing time in minutes taken by each machine. We are also given other rolls related to the machines down time, the total floor space of $50m^2$, the time of a work week of 35 hours, the ratios of which some products have to be produced relatively to others and that `Product` 1 can only be manufactured in the second machine.


Objective maximize
Variables
new column variable Machine_x index Product
new column variable Machine_y index Product
Constraints
Constraint
LPBlocks
Generates
Constraints
Objective

Product	Profit	Floor_space	Machine_X	Machin
1	10	0.1	10	27
2	12	0.15	12	19
3	17	0.5	13	33
4	8	0.05	8	23

$$0.1x_1 + 0.15[x_2 + y_2] + 0.5[x_3 + y_3] + 0.05[x_4 + y_4] <= 50$$
$$x_2 + y_2 = x_3 + y_3$$
$$10x_1 + 12x_2 + 17x_3 + 18x_4 <= 0.95 * 35 * 60$$
$$10y_1 + 12y_2 + 17y_3 + 18y_4 <= 0.95 * 35 * 60$$

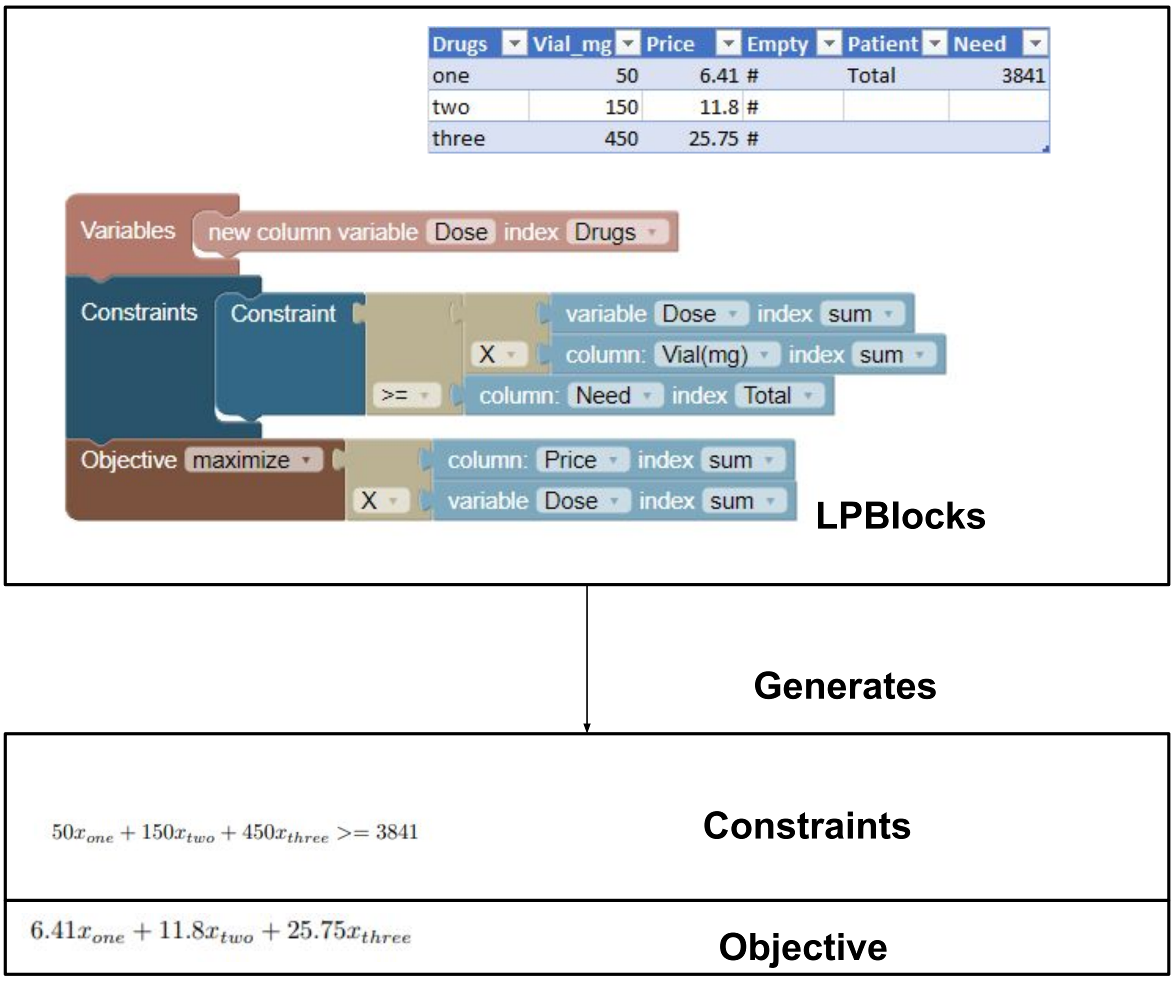
$$10x_1 + 12[x_2 + y_2] + 17[x_3 + y_3] + 18[x_4 + y_4]$$

Figure 5.3: Machine allocation problem in LPBlocks

In this example we create `column variables` for each machine taking the `Product` column as the index as opposed to previous examples where we created `matrix variables`. In this model we did not create a `matrix variable` as could be assumed due to the fact that the values for the machines are not used as index columns. Using them for defining a `matrix variable` would imply that at least one of the variables is referenced doing so and would offer nothing in terms of iterability and generalization. In terms of constraints we use the first constraint to express that the maximum floor space use is $50m^2$. If LPBlocks supported matrices as data input we could express this constraint in a less verbose manner. In the second constraint we express that the production of 2 is the same as 3. In constraints three and four we take into account the downtime of 5% for machine x and 7% for y by modeling that the total running time of each machine must be lower or equal to 95% and 93% of the total work week. The objective aims to maximize the profit and takes into account that `Product` 1 can only be manufactured in the second machine to create the objective function. Alternatively we could add a constraint to express that machine x produces 0 units of product 1 and use a generic approach to create the constraint.

5.4 Pharmaceutical company

This problem was taken from an *Excel* optimization tutorial[1] and seen in Figure 5.4. Our goal in this problem is to reduce the manufacturing costs associated with a given medicine. The data that we are given consists in three packages of the same medicine. The quantity of medicine per vial for each type of drug is also given. We also have information about the price per package and total needed in milligrams by the patients. In this model the variables are the quantity of each of the packages, the constraint is that the total amount produced of the drugs in milligrams must be higher or equal than the total requested by the patients. The objective is to minimize the production costs of the packages.



Drugs	Vial_mg	Price	Empty	Patient	Need
one	50	6.41	#	Total	3841
two	150	11.8	#		
three	450	25.75	#		

$$50x_{one} + 150x_{two} + 450x_{three} >= 3841$$

$$6.41x_{one} + 11.8x_{two} + 25.75x_{three}$$

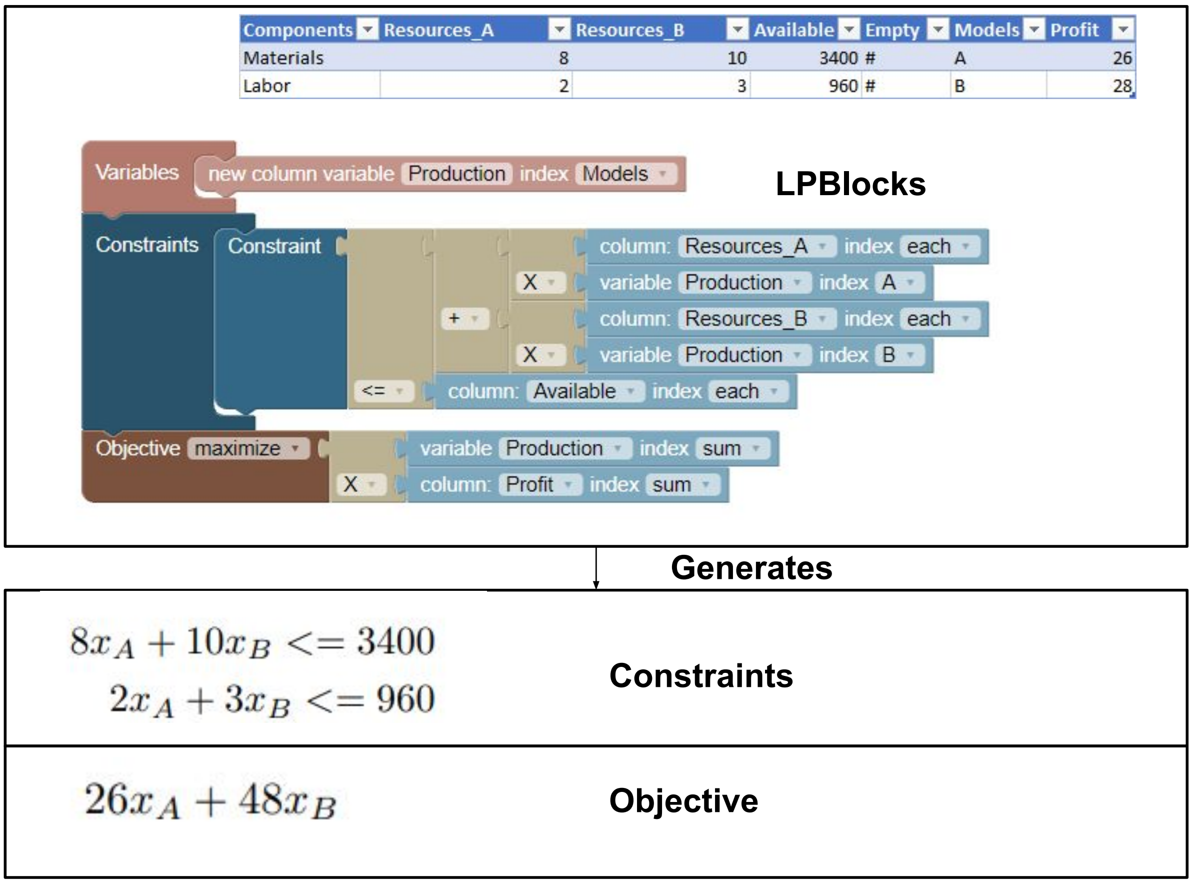
Figure 5.4: Drug manufacturing problem in LPBlocks

[1] www.exceltactics.com/using-solver-to-optimize-solutions-to-costing-problems-in-excel

To represent the variables we create a `column variable` with the index `Drugs` and called this variable `Dose`. To represent our constraint we use an `operation block` with the option <=. This block's first input is another `operation block` with the option X, in this block we fit a `variable block` with index sum and a `column block` with index sum. Using this blocks represent a *sumproduct* operation between the variables `Dose` and the column `Vial_mg`. The second input of the `operation block operation block` is a `column block` representing value *Total* of the column `Need`. The objective was created by doing a *sumproduct* of column `Price` and the variables `Dose`.

5.5 Computer manufacturing

In the problem shown in Figure 5.5 (taken from [4]),a computer manufacturer needs to decide the quantities of each of two computer models to manufacture. For this we are given two components necessary to manufacture a computer model, this being *Materials* and *Labor*. We also get for each of the components the costs for the manufacture of one model A and for one model B, and we also get the available quantity of each of these components. The data also include the computer models' profit associated with each of them. Our constraint in this problem is to guarantee the resources used do not surpass their values. The objective is to maximize profits.



Components	Resources_A	Resources_B	Available	Empty	Models	Profit
Materials	8	10	3400	#	A	26
Labor	2	3	960	#	B	28

LPBlocks
Variables
new column variable Production index Models
Constraints
Constraint
column: Resources_A index each
X
variable Production index A
+
column: Resources_B index each
X
variable Production index B
<=
column: Available index each
Objective maximize
variable Production index sum
X
column: Profit index sum
Generates
Constraints
Objective

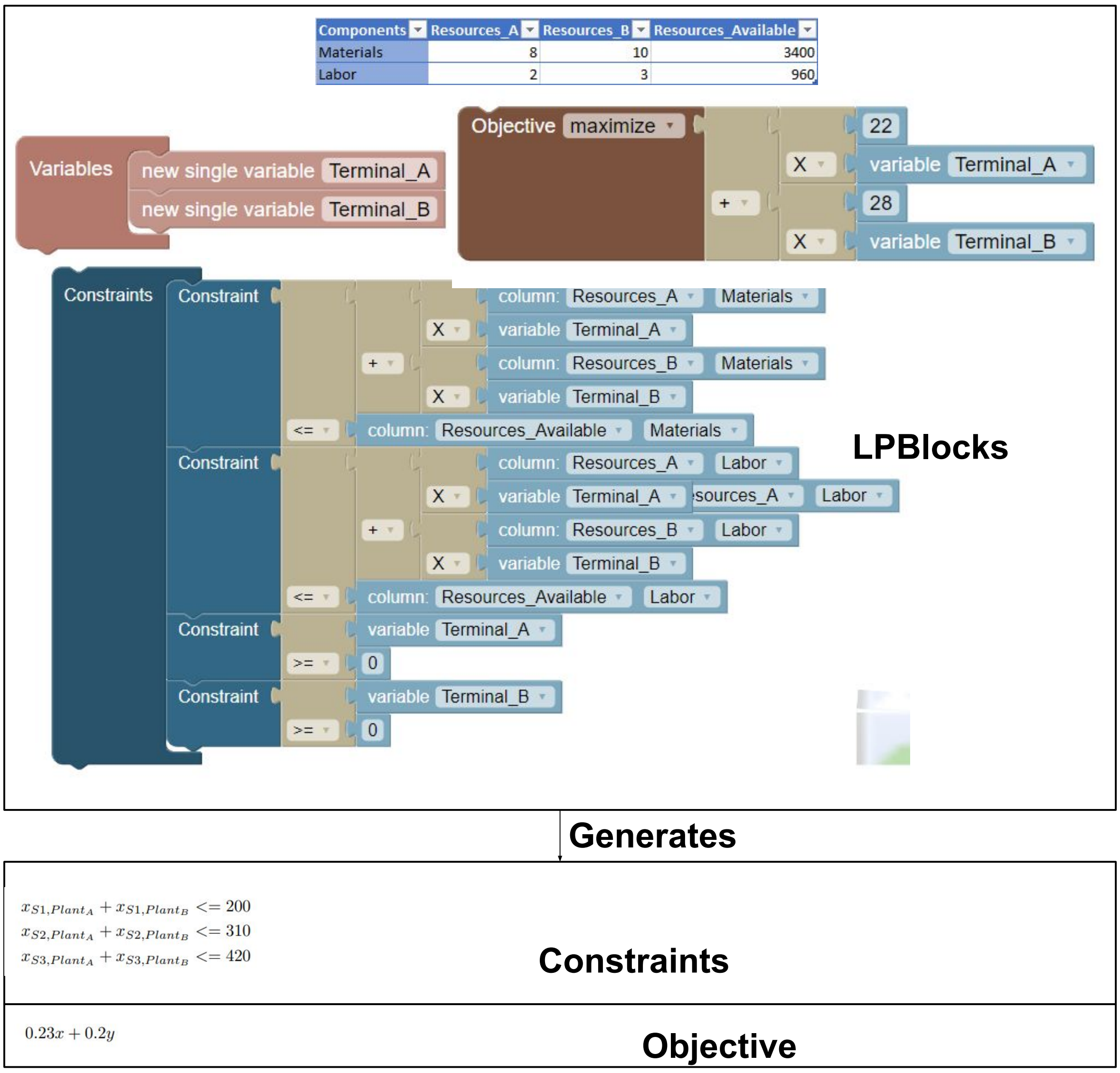
$$8x_A + 10x_B <= 3400$$
$$2x_A + 3x_B <= 960$$

$$26x_A + 48x_B$$

Figure 5.5: Computer manufacturing problem in LPBlocks

To represent this problem we first start with the variables. For this we start by creating a `column variable` named `Production` with the column `Models` as its index. Doing it allows us to create a variable for each of the two models. To create the constraint we use `operation blocks` with the X option and `value blocks` with the *each* option as the index of column `Available` and the variable `Production` to multiply the units produced by the costs Labor and Materials.

5.6 Satellite launching



Components	Resources_A	Resources_B	Resources_Available
Materials	8	10	3400
Labor	2	3	960

$x_{S1,Plant_A} + x_{S1,Plant_B} <= 200$
$x_{S2,Plant_A} + x_{S2,Plant_B} <= 310$
$x_{S3,Plant_A} + x_{S3,Plant_B} <= 420$

$0.23x + 0.2y$

Figure 5.6: Satelite launching problem in LPBlocks

This problem was taken from an operations research textbook [5]. In this problem, we did not use spreadsheet data for our formulation. For this problem we know that a company has two payloads (T1 and T2) and wants to calculate the number of satellites that carry each payload. We also have access to other information such as the success rate of satellites carrying payload T1 and T2 and the profit for each successful payload transportation.

We define two `single variables` in LPBlocks for our variables. The constraints express various launch success rates, launch times, and maximum launches for each of the loads. The objective is to maximize the profit for the loads.

5.7 Cargo allocation

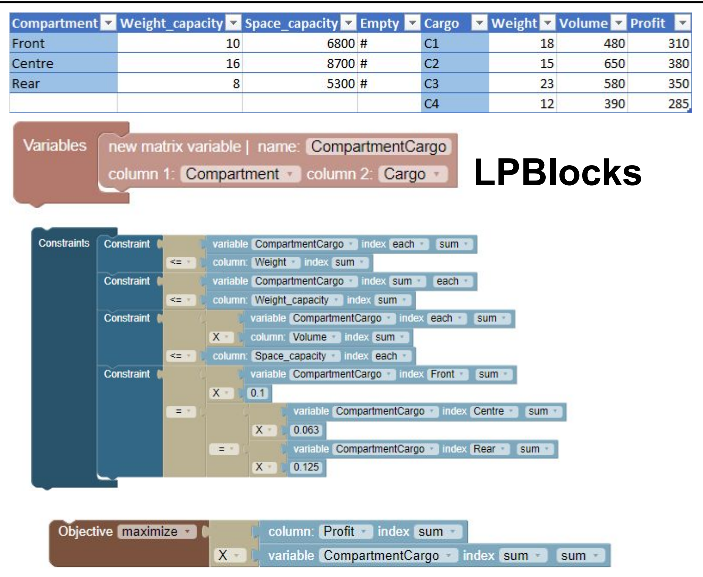
This is the example used for presenting the language in previous chapters and can be observed in Figure 5.7. To recall our goal in this problem is to maximize a shipping plane's profit by allocating four types of cargo amongst three plane sections.

The variable created is a `matrix variable` that takes the `index columns` `Compartment` and `Cargo` as its input.

For the constraints, the first three relate to space, volume, and weight capacity limitations. LPBlocks shines in these three constraints in the sense that when compared with writing a standard mathematical specification we need to write considerably less specification as doing it in our language allows us to write a generic specification of the constraint we want to represent and the compiler can generate the associated constraints in the mathematical notation. The last constraint related to balancing the plane's cargo in the different sections had to be done manually since LPBlocks does not possess a block capable of representing multiple equalities using a single block. We considered adding this block to our language but found that this specific use case was not prevalent to a level warranting adding the block. The objective function aims to maximize profits by calculating the *sumproduct* of the `Profit` with the sum of our variables elements indexed by the given `Profit` element.

5.8 Threats to validity

The main point of objection to the results of this Chapter come from the reduced number of problems used and to the fact that those problems could have been chosen to better fit our language and possible lack of coverage. To mitigate this we source our problems from different sources and strive to include problems that would use different features of our language, represent different use cases and have varying difficulty levels. The number of sources used could also be higher but despite only having problems from three different sources two of those sources were used in teaching in linear programming and purposely contained problems of different difficulties.



Compartment	Weight_capacity	Space_capacity	Empty	Cargo	Weight	Volume	Profit
Front	10	6800	#	C1	18	480	310
Centre	16	8700	#	C2	15	650	380
Rear	8	5300	#	C3	23	580	350
				C4	12	390	285

```
Variables  new matrix variable |  name: CompartmentCargo
           column 1: Compartment   column 2: Cargo

Constraints
  Constraint  variable CompartmentCargo index each  sum
          <=  column: Weight index sum
  Constraint  variable CompartmentCargo index sum  each
          <=  column: Weight_capacity index sum
  Constraint      variable CompartmentCargo index each  sum
              X   column: Volume index sum
          <=  column: Space_capacity index each
  Constraint      variable CompartmentCargo index Front  sum
              X   0.1
          =           variable CompartmentCargo index Centre  sum
                  X   0.063
              =       variable CompartmentCargo index Rear  sum
                  X   0.125

Objective maximize      column: Profit index sum
                    X   variable CompartmentCargo index sum  sum
```

Generates

Constraints

$$
\begin{aligned}
x_{Front,C1} + x_{Centre,C1} + x_{Rear,C1} &<= 18\\
x_{Front,C2} + x_{Centre,C2} + x_{Rear,C2} &<= 15\\
x_{Front,C3} + x_{Centre,C3} + x_{Rear,C3} &<= 23\\
x_{Front,C4} + x_{Centre,C4} + x_{Rear,C4} &<= 12
\end{aligned}
$$

$$
\begin{aligned}
480x_{Front,C1} + 650x_{Front,C2} + 580x_{Front,C3} + 390x_{Front,C4} &<= 6800\\
480x_{Centre,C1} + 650x_{Centre,C2} + 580x_{Centre,C3} + 390x_{Centre,C4} &<= 8700\\
480x_{Rear,C1} + 650x_{Rear,C2} + 580x_{Rear,C3} + 390x_{Rear,C4} &<= 5300
\end{aligned}
$$

$$
\begin{aligned}
&[x_{Front,C1} + x_{Front,C2} + x_{Front,C3} + x_{Front,C4}] * 0.1\\
= &[x_{Centre,C1} + x_{Centre,C2} + x_{Centre,C3} + x_{Centre,C4}] * 0.063\\
= &[x_{Rear,C1} + x_{Rear,C2} + x_{Rear,C3} + x_{Rear,C4}] * 0.125
\end{aligned}
$$

$$
\begin{aligned}
x_{Front,C1} + x_{Front,C2} + x_{Front,C3} + x_{Front,C4} &<= 10\\
x_{Centre,C1} + x_{Centre,C2} + x_{Centre,C3} + x_{Centre,C4} &<= 16\\
x_{Rear,C1} + x_{Rear,C2} + x_{Rear,C3} + x_{Rear,C4} &<= 8
\end{aligned}
$$

$$
310[x_{Front,C1} + x_{Centre,C1} + x_{Rear,C1}] + 380[x_{Front,C2} + x_{Centre,C2} + x_{Rear,C2}] +
$$

$$
350[x_{Front,C3} + x_{Centre,C3} + x_{Rear,C3}] + 285[x_{Front,C4} + x_{Centre,C4} + x_{Rear,C4}]
$$

Objective

Figure 5.7: Cargo allocation problem in LPBlocks

Chapter 6

Empirical evaluation

We intend to assess if our language and tool can be used by people with little or even no linear programming experience. We want to understand how people react to its features and how users from different backgrounds interact with our language, tool and the concepts behind it. Thus, in this chapter we showcase an empirical study we designed and ran.

6.1 Design

As said previously our aim is to observe the experience that different users have when using our tool to solve a plethora of problems and to collect their reactions and feedback.

To achieve the desired outcome in this study we planned on doing hour-long sessions with each of the participants. During each of the sessions we plan on doing an initial introduction to linear programming and how our language can be used to model linear programming problems. We then introduce our tool and its features. After doing so we solve a demo exercise in front of the user, followed by a joint exercise where the user solves the problem with possibly extensive help from us and two individual exercises where the user solves the problem with our input only when necessary.

Beyond solving the problems we created an individual follow-up questionnaire about their experience with our language and tool, prior experience and education, and other basic information such as age and gender. The questionnaire can be seen in Appendix B for questions related to the users background and Appending C for user feedback.

Since the goal of this study is to get qualitative feedback from users and to see their difficulties and assess our solution viability with different types of users, we aim to have participants from different backgrounds and with different levels of experience with linear programming. Thus, we contacted via email and word of mouth possible participants with diverging academic and professional experiences.

6.2 Instrumentation

During the session, we use several problems taken from the book and class notes referenced in Chapter 5 and whose datasets in a spreadsheet format can be found at `https://drive.google.com/drive/folders/1SAle03APg4JZefFIleS3-1hYGJ3gURI`. We use a web implementation of our tool that can be found at `https://lpblocks.herokuapp.com`.

We used the same set of problems in all our sessions and those where:

- **Vegetable mixture** - We used a portion of the problem seen in Section 5.1 containing only constraints retaining the maximum ratio of beans in the mixture and the absolute content of iron in the mixture. We used this problem as the tutorial problem. We decided to use this problem as the introductory problem since it showed a balance of complexity and relatability with the number of features of our tool shown during its resolution.
- **Fruit canning plants** - We used this problem as presented in Section 5.2 to use as a joint exercise. This exercise is arguably more complex than the ones to be solved more independently by the user but allows the user to experience most of the features needed for solving the future exercises.
- **Pharmaceutical company** - This problem is the first individual problem that the participants are asked to solve. This problem introduced in Section 5.4 is notably simpler than the previous problem and its simplicity lets the user focus on some of the core features of our tool and has the added benefit of decreasing the fear and intimidation for users less familiar with linear programming.
- **Computer manufacturer** - This problem seen in Section 5.5 is the last of the individual problems. It is more complex than the first of the individual problems but allows the participants to put into practice various of our language features such as *sumproducts* between variables and columns and the use of the *each* option, requiring a more thorough thinking process for the interpretation and expression of the constraints in our language.

6.3 Execution

To find participants for our study we contacted people in our circles as well and student groups of relevant subjects as said in Section 6.1. Our goal was to have a diverse group of participants and in our case that entailed increasing the efforts to acquire participants from non-computing backgrounds. Overall we were able to find 7 participants. Of this group 4 of the participants had a higher education degree in a computing field and were either currently Master or Ph.D. students. The other 3 had no computing degree, one of them worked in industrial engineering consulting and the other 2 work entailed doing web marketing, customer service, and logistics for e-commerce companies. All participants had at least a bachelor's degree and all participants from the computing background were male and from the business background were female.

The study was conducted with each participant at a time, in a think-allowed setting, using a video-conference software.

The first step when starting the session was to ask participants to download the datasets for the session. Afterwards, we gave a briefing on linear programming and some of its uses and some of the difficulties associated with its use. After doing so we introduce our solution, explain that the results of the study will be anonymous, and reassure the participants that our goals are to collect qualitative feedback about our tool, that our tool has a learning curve and that we are not passing any judgment on the participants.

After our introduction we start with the tutorial which is composed of an explanation and introduction to our language. This includes how the data is loaded from spreadsheets and its representation in the tool and our language blocks and their uses. We also give an explanation of some aspects of the compilation process and some of the caveats of our tools, we explain to the participant how to navigate our interface and finally do a live demonstration of one linear programming model creation using our tool. During this process, especially during the live demo, we encourage the participants to ask as many questions as they need.

Our next step after the tutorial is to let the user solve one of our problems. To do so we ask participants to share their screen and then after an explanation of the problem and data we guide the user on how to solve the problem using our solution. During this step we allow the user to do some exploration and to solve as much of the problem as he can.

Following the joint exercise, we start the first evaluation exercise. Both evaluation exercises are easier than the explanation exercise, the first one being easier than the second. After explaining the problem and data we allow the users to do as much as they can, only intervening when the user asks for help or it is necessary to avoid the user going on a wrong path. We then do the second problem similarly.

After solving the problems we ask the user for any feedback and proceed to end the session and direct them to the form.

6.4 Data collection

In this section we present the data collected from the forms: in Subsection 6.4.1 the portion relating to the participants background and in Subsection 6.4.3 the results relating to the feedback given by the users. In Subsection 6.4.2 we describe the individual session with each user.

6.4.1 Background data

In this subsection, we present the background information about the participants of our study. The questions in the form relating to the subject are available in Appendix B.

From the information in Table 6.1 (whose questions can be seen in Appendix B), we were able to observe that our participants have an age distribution predominantly between 21-24 and 25-35. The distribution is split evenly between those two groups, beyond those we had one person over the age of 35 . In terms of gender we have 4 males and 3 females. In terms of academic background, we have 4 people with a computing background, 2 people with a business background, and 1 person with both a business and engineering background . In terms of degree, all our participants had post-secondary degrees, having slightly more participants with a Masters than with Bachelors and none with Ph.D.. When it comes to prior experience with linear programming we had 4 persons who took college classes and 3 persons with no experience at all.

Participant	**Age**	**Gender**	**Degree**	**Years of university**	**Education field**	**Linear programming experience**	**Linear programming tools**
1	25-35	Male	Master	8	Computing	Took college classes	Programming tools
2	25-35	Male	Master	8	Computing	Took college classes	Excel
3	25-35	Male	Master	5	Computing	Took college classes	Programming tools, Other visual tools such as GAMS
4	21-24	Male	Bachelor	5	Computing	Took college classes	Programming tools, Excel
5	more than 35	Female	Master	6	Engineering and Business	None	None
6	21-24	Female	Master	5	Business	None	None
7	21-24	Female	Bachelor	3	Business	None	Excel

Table 6.1: Participants background gathered from the survey

6.4.2 Sessions

In this subsection we describe the sessions with the different participants.

Participant 1 During the tutorial the participant was able to understand the language and our goals, despite special emphasis had to be made when explaining the data and variables. The participant was proactive and was able to solve a considerable portion of the joint exercise. The participant also demonstrated interest in our tool and language.

Relatively to the first individual problem, the participant tried to maximize the objective instead of minimizing. This error could have been made since the user might not have understood that our goal with this problem was not to increase the profit but to decrease production costs since the participant might have been more accustomed to maximization problems. Other than that the user did not need considerable amounts of help.

When solving the second problem the user had some difficulties distinguishing between the option `sum` and `each` in the constraints and objectives. This user did demonstrate more difficulty in solving this problem than the previous one especially on how to express the costs in our language, but with some help the user was able to get to the correct solution.

Overall the user had a positive reaction to our tool and was able to do a good portion of the exercises and we felt that the user understood the core of our language and tool and that with some more experience would be able to achieve mastery of our tool.

Participant 2 During the language presentation the subject found himself interested in the language and its features, this being demonstrated by the subject asking several questions.

When doing the first demonstration problem the user did have some difficulties on where to place the blocks. After the initial error, he was able to understand the logic behind the variable creation. Other problems subsequent during the demonstration were related to differences between `each` and `sum`, confusion between `sum` values from one column or variable, and using the `operation block` with the value `+`. This was solved. Despite some initial confusion, the participant was able to understand the language.

During the first exercise, the major difficulty was representing the production in milligrams as the *sumproduct* of the vial quantity and the produced vials. The user was able to get the answer and understand the reasoning with some help.

The main difficulty in the second problem was using two constraints, one for the labor and the another for the materials instead of using an `each` for both. The user ended up understanding the logic behind the model and did the rest of the model by himself.

Overall we found that the user understood the core concepts of our language and with some training could become proficient in using it.

Participant 3 The participant was attentive and asked several questions when watching the tutorial.

When doing the first exercise the user was proactive and was able to solve a significant portion of it. Some difficulties involved the use of `each` and the use of columns and variables.

When doing the second exercise the user was able to solve the problem on his own, the biggest problem being in using the correct block for addressing variables and discerning between `column` and `variables blocks`.

When doing the third exercise the user experienced some confusion between the use of `each` and `sum`, but was able to solve the exercise.

After our session, he manifested interest in our tool and gave some of his opinions related to our tool such as adding more solver options and exporting data to *Excel* files.

Participant 4 The participant was interested in the language and tool and asked several questions. During the example exercise, the participant asked questions about block positioning and language features, particularly about the `sumproduct` operation.

During the joint exercises, the user was proactive and found some initial difficulties when trying to represent the constraints and objectives in our language but ended up being able to continue with some help. The main problems were related to understanding how the data correlated with the desired goal of the constraints and objectives. This might be related to the user not using linear programming regularly.

In the first exercise, there were some difficulties in creating the variables, particularly in creating 3 variables since the user tried to use 3 `column variable blocks` to create 3 variables, after visualizing the formulation generated by our tool we were able to understand the correct way to build the model. Understanding how to represent the constraints with the data took some tries but with some guidance and visualizing the output the user got to the correct answer. For the constraints, the user was able to create them correctly by visualizing the result, error messages and some trial and error.

In the second exercise the participant's main difficulty was in representing the variables. The rest of the model took some trial and error but with minor guidance and output visualization, the user was able to get to the solution. The user did do a manual *sumproduct* instead of using the *sumproduct* operation but that was a result of the trial and error process. With more practice, the user might gain some dexterity with the tool.

6.4.2.1 Participant 5

During the tutorial the participant was able to understand the purpose of our language and tool, although special emphasis had to be made when explaining the data and variables. Despite not being from a computing background this participant also had an engineering background and understood some of the concepts that our solution tackles, however, the user did demonstrate some skepticism to our language and its complexity for non-technical users.

When doing the tutorial the user did not ask as many questions as the previous ones but did ask some. We tried to explain the concepts and how our tool could be used and the user seemed to understand the language's purpose. The user did complement the interface and the features, but did feel intimidated by our tool and doubted her ability to solve problems such as the tutorial problem without our help. In response we reassured that after doing an exercise with us and gaining some experience the tool becomes easier and more intuitive.

During the solving of the joint exercise we allow the user to try to understand the problem and ask any questions and give her input. Some of the difficulties the user had were in understanding the differences between the blocks for variable access and variable creation. We answered that and were able to convey their functions. Some of the operations such as *sumproducts* were not obvious. The user at the end of this exercise expressed some doubts about the virtue of our tool for users without any prior programming experience and admitted that despite her prior programming experience in one or two classes in college she still had difficulties understanding and working with our language.

During the first individual problem the user was able to create the variables, understanding the mapping of our language to the mathematical formulation and how to write the expression was something that did not occur naturally but after the initial awkwardness and further explanation, the user was able to get over those hurdles. And once again understanding the differences between variables and columns and how the constraint in the natural language would be expressed in terms of data were only possible with some help and further explanations, but the user was able to get there on her own afterward.

We were not able to do the last problem due to time constraints since we had planned on doing a one-hour session but by the time we finished solving the first individual problem the time was almost over and the user could not continue.

6.4.2.2 Participant 6

The user was interested during the presentation and during the demonstration exercise, where we gave special emphasis to explaining clearly the uses of linear programming and optimization. However, the participant did manifest some level of fear related to using our tool. Some of this previous hesitation had to do with the user not having a strong mathematical background. Despite the hesitation, the user did manifest some excitement with our tool and despite not doing any optimization in her line of work, she currently does use *Excel* for tasks related to marketing and business in general, including keeping track of stocks and inventory and interacting with logistics software.

We did most of the joint exercise with this participant, but we saw that during the explanation the user was interested in the different features of our language. She was able to understand some of the solutions we created and the logic behind it. She also showed some appreciation for the interface with special relevance for the dynamic features, data visualization, and blocks.

During the first individual problem the user showed some initial difficulty understanding what variable to create, but after explaining the purpose of the problem again and some trial and error with different blocks the user was able to get to the correct solution. For creating the constraint, understanding where to use the *inequation* was an initial problem, but was quickly corrected with some explanation. Creating the *sumproduct* was the harder part and both the blocks and which *column* and *variable blocks* and options to use require some extensive explanation. In the end, the user got to the solution and was able to understand the logic behind the use. For creating the objective the user still needed help but was able to get to the solution faster than for the constraint.

When doing the last problem the user faced some difficulties when understanding the problem and creating the variable, this time the blocks were not an issue. For the constraints, there were still issues of blocks such as trying to use the wrong `variable block` to access a variable and a considerable amount of help was needed for the user to solve the problem, but, eventually, she did and there was some improvement compared with the first problem especially in understanding the blocks and how they were used. After solving this exercise the user expressed her liking for the interface and the blocks. She also affirmed liking the block-based language and made some comments about possible improvements such as improving the distinction between the `constraint` and `variable value blocks`.

6.4.2.3 Participant 7

When introducing the language the participant demonstrated interest in our solution, since she had work experience with *Excel* and found it to have some potential, however since the user did have little to no prior experience with linear programming the user felt intimidated by it at first. We made a special emphasis on explaining what linear programming entails and in explaining its components and how they relate to our language. The user during the presentation and tool showcase also expressed her liking for the interface and the language aesthetics and how the data was displayed.

We did most of the joint exercise with this participant, but we saw that during the explanation the user was receptive to the different features of our language and was able to understand the solution we created and the logic behind it.

During the first exercise, there was some difficulty in choosing which variable to use, but after re explaining the problem and how variables are created the user was able to get to the correct block, and using our visualization features the user was assured she made the correct choice. There were also some difficulties in representing the value in milligrams from the variable created and the value in milligrams for each vial, but the user was able to eventually understand how to do it, although with some help. The user was able to get the value in the spreadsheet to represent the total necessary medication in milligrams. The user was able to get the objective after some trial and help.

During the second problem, the user was able to get the variable and part of the constraints. Once again the user needed help in doing the *sumproduct* and there was still some confusion in using the `variable` and `column block` and some help was warranted to achieve the correct solution. Also, some help was necessary to get to the objective but the user was able to solve portions of the problem. Despite the lack of previous experience with linear programming the user still appreciated the tool and was able to understand some concepts especially using our visualization and dynamic features. She also gave some advice consisting especially in improving the differentiation between `variable` and `column blocks`. When creating the constraints there was still some confusion on how columns and variables were used, but with some help she was able to get to the correct answer.

6.4.3 Participants feedback

During the survey we asked for feedback about our tool from the users as well as some ideas and possible improvements that the users might believe would benefit our tool. To do so we added the questions in Appending C to our survey, for which the results can be seen in Table 6.2. The details about the questions asked are in Appendix C.

6.5 Analysis and conclusions

In this section, we present some conclusions and critical analysis related to the data gathered from our survey and the sessions and their implication on our work and possible future paths, changes, and applications.

According to the data gathered from our sessions and survey, we can separate our participants into two different groups, one being the users 1 to 4 or the users with a computing background and 5 to 7 or the users with a business background. By coincidence, the computing group was all male and the business

Participant	Other tools	Positive aspects	Negative aspects
1	Better usability.	Usability; Performance;	
2	This tool has a strong visual component. Although Excel is algorithmically "more complete"	UX is nice. Drag and drop works well. Also nice shortcuts: delete, ctrl+c ctrl+v. Constraints and Objective "sub-windows"are useful to understand if the inequations match the user's reasoning.	Took me a while to understand the difference between "light-blue" variable block and "light-blue" column block. Each and sum are also a little confusing at first (but the previously mentioned sub-windows help fixing mistakes).
3	Easy to learn interface, good visual aid (in the formula section - we can compare what the blocks translate to in terms of mathematical equations)	Easy to learn interface, good visual aid (in the formula section - we can compare what the blocks translate to in terms of mathematical equations). No installation needed is a plus.	No option export to file. For bigger problems / harder to solve problems, the lack of alternative solvers could result in not being able to solve the problem.
4	Is not drag and drop like this app.	Simpler	Is not intuitive to beginners
5		Easy to use, to grasp and get together.	Terms of the buttons. Depending on the client if he doesn't have the right knowledge it will be difficult to think in a problem solving logic.
6		Visualy atractive; Well organized; All the information is well explained when building the model.	The operations ordering is confusing.
7		Very intuitive, easy to use, visualy appelative and well organized. We can visualize the model being created	The order of the operations is not very intuitive.

Table 6.2: Participants feedback gathered from the survey

group all female, the computing group had on average more years of higher education and probably due to their field of study had prior experience with linear programming. Age-wise the differences were not starling.

From our sessions, we were able to gather that all the participants were interested in our tool and interface and across the board (although possibly due to selection bias) asked various questions. The questions were different according to the user background, being slightly more technical in nature from the computing group. We also found that users universally liked the interface and how the data was accessible. This included the business group where the participants particularly enjoyed having all the data, workspace and mathematical formulation in the same place. However, during the tutorial and introduction to our solution, we found that the business group showed either skepticism for participant 5 or hesitation for participants 6 and 7. This was expected due to them not having experience in the field of linear programming.

In terms of actual dexterity and level of proficiency acquired when using our tool during this session we can conclude that the computing group did considerably better than the business group. This was evident by the considerably higher portion of the problems they were able to build and less input needed. However we still found some difficulties that prevailed across the users of this group. We found that the relation between the data and the specification in natural language was not a considerable problem for this group except in some less obvious cases. We found that distinguishing between *sum* and *each* initially might cause difficulties in some cases, but are quickly resolved after creating a couple of expressions using these options. We also found in some cases that the users of this group did not use the language as intended, but that quickly resolved it. Distinguishing between columns and variables was initially a problem but was eventually solved. For this group, we also found that the dynamic compilation to the mathematical formulation was of considerable help in learning our language.

When doing the exercises with the business group we found that the mastery of our tool acquired by the users varied considerably. We found that user 7 did better than user 6 and that user 5 despite having only done the first individual exercise. However, with this group, we found some common problems such as misusing blocks, confusing variables, and columns, difficulties doing the operations such as *sumproducts*, difficulties with block order, and how to construct mathematical operations using the blocks. We also found that these users also had difficulties in interpreting the exercises and in understanding how the data correlated with the goals of the problems, however, we found that as the session progressed their ability to do so improved.

From the feedback gathered from the participants, seen in Table 6.2, we found that for the computing group the negative aspects focus on our tool having some learning curve for beginners, the difficulty in distinguishing between `variable` and `column blocks`. Other negative aspects for this group related to the lack of features such as no options to export or import models and the lack of different solvers. The positive aspects for this group were related to our interface features and usability such as the ability to drag and drop, the ability to visualize the mathematical formulation being built in real-time, the ability to visualize the data, our tool performance, and not requiring installation. When compared to other tools this group found that it had better usability, the interface was easy to learn and was especially helpful due to the dynamic compilation and errors allowing users to learn the language as they go along. The drag and drop capabilities were also something that our application did better than the ones previously used by the participants. The participants in this group had previously used programming tools, *Excel* and one of them had used graphical software to encode linear programming programs.

The business group in the feedback questions expressed that the negative aspects were that the language could be difficult for people who do not have experience in technical fields and that the order in which blocks have to be inserted in the workplace to create an expression differs from the mathematics.

This group also found positive aspects in our language such as ease of use and starting building models, the language being visually appealing, being able to see the model being built in real-time and our tool being intuitive and well organized. These participants had no prior experience with other optimization tools.

Overall we found that some features of our tool such as a graphical interface that displays everything needed by the user on a single page, the dynamic compilation, and errors were universally liked. We also found that despite the users in the computing group having done better than the users in the business group both ended our session with some knowledge on how to use our tool. Some of the challenges felt by the participants were for the majority distinguishing between certain blocks such as columns and variables and for users without a computing background, the block order to create expressions was not intuitive. We feel that for the users with less experience in the field our tool can serve as an introductory experience to linear programming and operations research and also feel that despite having solved only three problems the users did improve in all groups and had positive reactions to LPBlocks.

6.6 Threats to validity

This study's main threats validity stand from the number of participants and their relationship with the research team and the number of problems used. When it comes to the participants the first threat is the fact that the participants were known colleagues, students or acquaintances of the research team, to mitigate this threat we insisted that the participants gave their true feedback and that the goal of the study was to evaluate our tool with users. The second threat comes from the low number of participants, and while we had only 7 participants they came from different backgrounds and feel that their experiences can give us insights into how our tool works with users of different backgrounds and especially allow us to see some of the difficulties felt by less technical users comparatively to more savvy ones.

When it comes to the problems, one threat could be the fact that the participants did not solve the problems using *Excel* or another tool. Since we did not have time to find inexperienced users and train them on how to solve optimization problems using *Excel* We mitigate this problem by having a group of users from a computing background that had experience with other tools and could give us some insight on how our tool compared to others. The participants also solved only three problems, while this is true the problems varied in size and showcased most of LPBlocks features.

Chapter 7

Conclusions and future work

In this chapter we present our final conclusions and answer the research questions asked in Chapter 1. This is done in Section 7.1 and in Section 7.2 we present our expected future work.

7.1 Conclusions

Current tools for creating linear programming models often require previous programming knowledge or use *ad-hoc* methodologies and lack some features that would benefit the less technical user. In our work we were able to design a block-based language that can express those problems and create a tool that coupled this language with various features to create an environment that lent itself to users of different technical backgrounds.

We were able to successfully use LPBlocks to express numerous and varied linear programming problems in Chapter 5. Further, we took our implementation of LPBlocks, contacted possible participants of different backgrounds, designed and ran a study to collect data referent to their experience with our tool and language.

During this study, we were able to gather that some features were universally liked such as the dynamic compilation and errors, drag and drop, the ability to see everything in one window, and the use of a block-based language. We also found that some aspects of LPBlocks cause more problems such as differentiating between `column blocks` and `variable blocks`, some of the logic behind the language and thinking in terms of data to solve problems was not obvious for less technical users and some users would have liked to see more features such as data loading and exporting and more solver options.

We now answer the research questions posed in Section 1.3:

- **RQ1** - *Can we use block-based languages to represent linear programming formulations?*

Yes, as seen in Chapter 5 LPBlocks can be used to represent numerous linear programming formulations. While we do not have a formal proof that guarantees that our language can represent all linear programming problems, by solving a diverse array of problems we have a reasonable assurance that most problems could be solved using our LPBlocks.

- **RQ2** - *Will using a block-based language provide users an easier environment for linear programming?*
  From our study we were able to conclude that users that had previously used Excel and other optimization software compared ours favorably when it came to the graphical interface. However, not all users in our study had experience with those tools and none of the users without a computing background did. Despite the lack of comparison for those users, they did enjoy and benefit from the graphical and block-based features of our tool, features that do not exist on most of the other tools.

7.2 Future work

Our future work aims are to use the data gathered in Chapter 6 to continue improving our tool and language from a user experience perspective and further test our tool using a higher number of participants. Taking the previously said into account we defined the following tasks as possible future work:

1. Improving the differentiation of `variable blocks` and `column blocks` in LPBlocks. This could be done by changing the color scheme used for those blocks and making it more similar to the `variable creation blocks` for the `variable blocks` and to the index columns for the `column blocks`.
2. Adding more features to our tool such as loading and exporting data and mode solver options.
3. Prompting the user with automatic suggestions and templates when building models.
4. Doing further research with a higher number of participants and more time dedicated to teaching the language and some of the processes necessary to create linear programming models, especially for users with less experience in the field.

Bibliography

[1] D. R. Anderson, D. J. Sweeney, T. A. Williams, J. D. Camm, and J. J. Cochran. *Quantitative Methods for Business*. Ed. by C. Valentine. url: https://www.amazon.com/Quantitative-Methods-Business-David-Anderson/dp/0840062346.

[2] A. C. Bart, E. Tilevich, C. A. Shaffer, and D. Kafura. "Position paper: From interest to usefulness with BlockPy, a block-based, educational environment." In: *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 2015, pp. 87–89. doi: 10.1109/BLOCKS.2015.7369009.

[3] A. C. Bart, J. Tibau, E. Tilevich, C. A. Shaffer, and D. Kafura. "BlockPy: An Open Access Data-Science Environment for Introductory Programmers." In: *Computer* 50.5 (2017), pp. 18–26. doi: 10.1109/MC.2017.132.

[4] J. E. Beasley. *Or-notes*. url: http://people.brunel.ac.uk/~mastjjb/jeb/or/lpmore.html.

[5] M. Carter and C. C. Price. *Operations Research: A Practical Introduction*. CRC Press, 2000. isbn: 9780849322563.

[6] M. Chikwature. *Challenges faced by pupils in the learning of linear programming at ordinary level: A case of a secondary school in Umguza District*. 2018. url: http://liboasis.buse.ac.zw:8080/xmlui/bitstream/handle/123456789/10979/chikwature-margaret-curriculum.pdf?sequence=1&isAllowed=y.

[7] G. Collaud and J. Pasquier-Boltuck. "gLPS: A graphical tool for the definition and manipulation of linear problems." In: *European Journal of Operational Research* 72.2 (1994), pp. 277–286. issn: 0377-2217. doi: https://doi.org/10.1016/0377-2217(94)90309-3. url: https://www.sciencedirect.com/science/article/pii/0377221794903093.

[8] N. Fraser. "Ten things we've learned from Blockly." In: *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 2015, pp. 49–50. doi: 10.1109/BLOCKS.2015.7369000.

[9] H. D. Giao, J. Cunha, and R. Pereira. "Linear Programming Meets Block-based Languages." In: *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2021, pp. 1–3. doi: 10.1109/VL/HCC51201.2021.9576449. url: https://doi.ieeecomputersociety.org/10.1109/VL/HCC51201.2021.9576449.

[10] H. da Gião, J. Cunha, and R. Pereira. "Towards a Block-based Language for Linear Programming." In: 12^{th} *National Symposium of Informatics (INForum'21)*. 2021, pp. 36–49.

[11] H. Guerrero. *Excel Data Analysis: Modeling and Simulation*. Springer, 2010. url: https://www.springer.com/gp/book/9783642108341.

[12] V. Lazaridis, K. Paparrizos, N. Samaras, and A. Sifaleras. "Visual LinProg: A web-based educational software for linear programming." In: *Comput. Appl. Eng. Educ.* 15.1 (2007), pp. 1–14. doi: 10.1002/cae.20084.

[13] P.-C. Ma, F. H. Murphy, and E. A. Stohr. "A Graphics Interface for Linear Programming." In: *Commun. ACM* 32.8 (Aug. 1989), pp. 996–1012. issn: 0001-0782. doi: 10.1145/65971.65978.

[14] M. Macarty. *Linear Programming (LP) Optimization with Excel Solver*. url: https://www.youtube.com/watch?v=6xa1x_Iqjzg&ab_channel=MattMacarty.

[15] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. "The Scratch Programming Language and Environment." In: *ACM Trans. Comput. Educ.* 10.4 (Nov. 2010). doi: 10.1145/1868358.1868363.

[16] J. Mendes, J. Cunha, F. Duarte, G. Engels, J. Saraiva, and S. Sauer. "Towards systematic spreadsheet construction processes." In: (2017). Ed. by S. Uchitel, A. Orso, and M. P. Robillard, pp. 356–358. doi: 10.1109/ICSE-C.2017.141. url: https://doi.org/10.1109/ICSE-C.2017.141.

[17] E. Pasternak, R. Fenichel, and A. N. Marshall. "Tips for creating a block language with blockly." In: *2017 IEEE Blocks and Beyond Workshop (B B)*. 2017, pp. 21–24. doi: 10.1109/BLOCKS.2017.8120404.

[18] E. W. Patton, M. Tissenbaum, and F. Harunani. "MIT App Inventor: Objectives, Design, and Development." In: *Computational Thinking Education*. Ed. by S.-C. Kong and H. Abelson. Singapore: Springer Singapore, 2019, pp. 31–49. isbn: 978-981-13-6528-7. doi: 10.1007/978-981-13-6528-7_3.

[19] J. Pereira and S. Fernandes. "Two-variable Linear Programming: A Graphical Tool with Mathematica." In: *SYMCOMP 2013 - 1st International Conference on Algebraic and Symbolic Computation*. Sept. 2013, pp. 159–173.

[20] D. Saleh and T. Latif. "Solving LProg Problems By Using Excel's Solver." In: *Tikrit Journal of Pure Sc.* Vol. 14 (Mar. 2009).

[21] E. Senne, C. Lucas, and S. Taylor. "Towards an Intelligent Graphical Interface for Linear Programming Modelling." In: *Journal of Intelligent Systems* 6.1 (1996), pp. 63–94. doi: doi:10.1515/JISYS.1996.6.1.63.

[22] L. Siaw Chong and C. Jia Xin. "Creating a GUI Solver for Linear Programming Models in MATLAB." In: *Journal of Science and Technology* 10.4 (Dec. 2018). url: https://publisher.uthm.edu.my/ojs/index.php/JST/article/view/3653.

[23] D. Solow. *Linear Programming: Second edition*. Dover Publications, 2014.

Appendix A

Example spreadsheet data in JSON format

Listing A.1: Spreadsheet data in JSON format

```
{
  [
 {
   Vegetables: 'Beans',
   Iron: 0.5,
   Phosphorus: 10,
   Calcium: 200,
   Cost_per_pound: 0.2
 },
 {
   Vegetables: 'Corn',
   Iron: 0.5,
   Phosphorus: 20,
   Calcium: 280,
   Cost_per_pound: 0.18
 },
 {
   Vegetables: 'Broccoli',
   Iron: 1.2,
   Phosphorus: 40,
   Calcium: 800,
   Cost_per_pound: 0.32
 },
 {
   Vegetables: 'Cabbage',
   Iron: 0.3,
   Phosphorus: 30,
   Calcium: 420,
   Cost_per_pound: 0.28
 },
 {
   Vegetables: 'Potatoes',
   Iron: 0.4,
   Phosphorus: 50,
   Calcium: 360,
   Cost_per_pound: 0.16
 }
]
}
```

Appendix B

LPBlocks study form participant background questions

1. **(Age) How old are you** - In this question we give the participant the option to select one of the following options:
   a) Less than 18
   b) 18-20
   c) 21-24
   d) 25-35
   e) More than 35
   f) Rather not answer
2. **(Gender) What is your gender identity** - In this question we give the participant the option to choose one of the following options:
   a) Male
   b) Female
   c) Other
   d) Rather not answer
3. **(Education degree) Highest degree** - We allow the participant to choose form one of the following options:
   a) Bellow High School
   b) High School

   c) Bachelor
   d) Post-graduation
   e) Master
   f) PhD

4. **(Years of university) Years of post-secondary education** - We allow the participant to choose one of the following options:
   a) 0
   b) 1
   c) ...
   d) 9
   e) 10+

5. **(Education field) Main fields of study** - We allow the user to choose multiple of the following options:
   a) Computing (Computer Science, Informatics Engineering, Information systems, Information Technology, etc...)
   b) Life Science(Biology, Chemistry, etc...)
   c) Engineering(Electrical Engineering, Mechanical Engineering, Civil Engineering, etc...)
   d) Business(Economics, Business, Marketing, etc...)
   e) Humanities(History, Languages, etc...)
   f) Mathematics(Mathematics, Statistics)
   g) Other: (fill in option)

6. **(Linear programming experience) What's your academic experience with linear programming and operations research** - We allow the particpants to choose from multiple of the following options:
   a) None
   b) Took college classes
   c) Degree in related field(ex Industrial engineering or statistics)
   d) Professional experience
   e) Other: (fill in otpion)

7. **(Linear programming tools) Experience with optimization tools programming and otherwise** - We allow the participants to choose from mutiple of the following options:

a) Programming tools

b) Excel

c) Other visual tools such as GAMS

d) Other: (fill in option)

Appendix C

LPBlocks study form user feedback

1. **(Other tools) If you used any of the tools above how do those compare to ours** - This is a write in question.
2. **(Positive aspects) What were in your opinion the positive aspects of our tool** - This is a write in question.
3. **(Negative aspects) What where in your opinion the negative aspects of our tool** - This is a write in question.