Towards a Block-based Language for Linear Programming[⋆]

Hugo da Gião[1,2][0000−0003−3798−0367], Rui Pereira[2][0000−0002−5801−7345], and Jácome Cunha[1,2][0000−0002−4713−3834]

[1] University of Minho
[2] HASLab/INESC TEC
hugo.a.giao@inesctec.pt rui.a.pereira@inesctec.pt jacome@di.uminho.pt

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 is lacking in terms of 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 propose creating a block-based programming language termed LPBlocks that allows users to create linear programming models using data contained inside spreadsheets. This language will guide the users to write syntactically and semantically correct programs and thus aid them in a way that current languages do not. As an initial evaluation we have used LPBlocks to model 7 linear programming problems with success.

Keywords: linear programming, spreadsheets, block-based languages, end-user programming

1 Introduction

The versatility of linear programming in specifying 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 [7].

However, the typical methodologies used when solving those types of problems using spreadsheet software often come with underlying problems such as

⋆ This work is supported by the national funds through the Portuguese Funding Agency (FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020).

relying on an imprecise process to feed data from the spreadsheet to the solver, as well as the difficulties visualizing the models. Many tools commonly used by professionals working with linear programming such as MATLAB[3] and GAMS[4] either require considerable programming knowledge or use *ad-hoc* and error-prone methodologies. Some projects related to the use of GUI's for linear programming in MATLAB are despite the use of graphical interface detached from the business logic and more applied use cases [4].

Some projects 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 [16,9]. 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 [10,17].

Numerous projects have applied visual languages to various areas 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 [14]. These languages include BlockPy [1], a web-based platform that lets the user write and run Python code using a block-based language, and Scratch [11], a block-based visual programming language and educational tool mostly targeted at children.

Taking into consideration the potential of Blockly to improve the usability and practice of linear programming and the extensive study of block-based languages and their practices [14,5], we aim to build upon the work already done in this field to the create a visual programming language and tool capable of expressing linear programming models in a high level, safe and intuitive manner.

In this work we extend our previous idea of bringing block-based languages and linear programming [6] by presenting in detail LPBlocks, a block-based language written using Blockly that is tailored for the construction of linear programming models. To illustrate the language's applicability we present its use to model several linear programming problems demonstrating its use and fallbacks.

2 Related work

Blockly is a framework that has been used over the last decade in many projects aiming to improve programming accessibility, in diverse areas going from teaching core concepts [11] to data science [1], robotics [8] and app development [15].

One such project, BlockPy, lets users access several data science libraries and features, and generates the associated Python code. Additionally, it runs inside a browser using a Javascript Python interpreter [1]. This project was shown to be particularly useful in helping introductory level computer science students transition into having full-fledged ideas, as well as offering the students a compelling context for learning programming.

[3] `https://www.mathworks.com/products/matlab.html`
[4] `https://www.gams.com/`

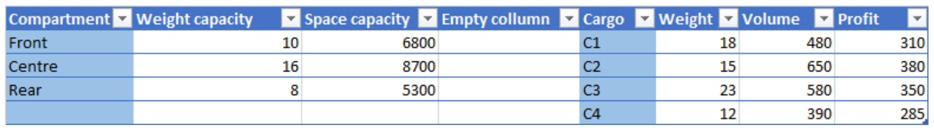
Another such project is the MitApp inventor, a platform that allows users to create full-fledged mobile applications with various graphical interfaces [15]. When using this platform one is able to work on the design of the applications using a graphical user interface, and on the program's logic using a block-based programming language.

Some works tackling various issues associated with the teaching and practice of linear programming using visual programming also exist. One such work presents a tool named GLP-tool which allows users to define two variable linear programming problems using an algebraic language and visualize the solving process graphically [16]. Another relevant work introduces the LPFORM software, which allows managers and operations researchers to formulate large linear programming programs [10]. LPFORM empowers users during the more repetitive steps of building these models by allowing linear programming problems to be represented using objects and relationships, and uses knowledge-based techniques to generate the input given to the solver.

There are many solutions in the market catering to users with different needs and levels of experience although we found that Excel was popular amongst less technical users [7]. The majority of other tooling such as Matlab, GAMS and SAS/OR[5] cater primarily to users with some programming or mathematical knowledge. Some of the problems associated with these more advanced tools relate primarily to the technical complexity of modeling linear programming knowledge, as well as the need to use textual programming languages. When using Excel for modeling users face problems associated with the use of spreadsheet software.

3 A block-based language for linear programming

In this section, we introduce our proposed language, LPBlocks, using an example featured in a Master of Business Administration (MBA) exam [2]. 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 airplain'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 1.



Compartment	Weight capacity	Space capacity	Empty collumn	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

Fig. 1. Input data for the running example problem

[5] `https://www.sas.com/en_us/software/or.html`

3.1 Input data specification

Our solution requires the input data to follow a specific structure. This structure allows for the definition of **index columns** (as seen highlighted in blue in Figure 1), this are used to reference values and iterate over the **data columns** (in white 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 empty column 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 LPBlocks constructs

The building blocks of the linear programming language we propose can be seen in Figure 2. LPBlocks includes:

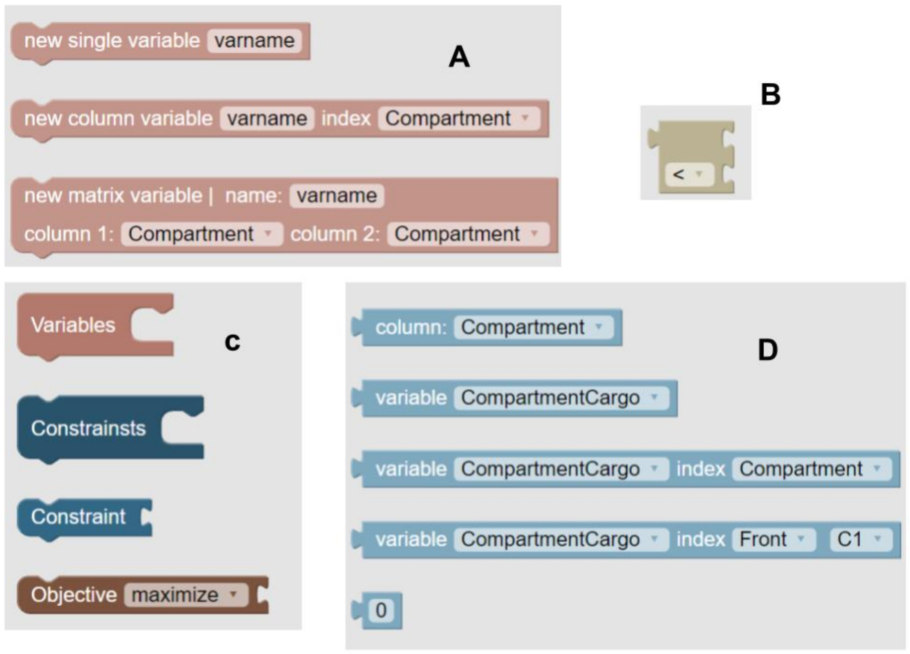

A
new single variable varname
new column variable varname index Compartment
new matrix variable | name: varname
column 1: Compartment column 2: Compartment
B
<
C
Variables
Constrainsts
Constraint
Objective maximize
D
column: Compartment
variable CompartmentCargo
variable CompartmentCargo index Compartment
variable CompartmentCargo index Front C1
0

Fig. 2. Building blocks for LPBlocks

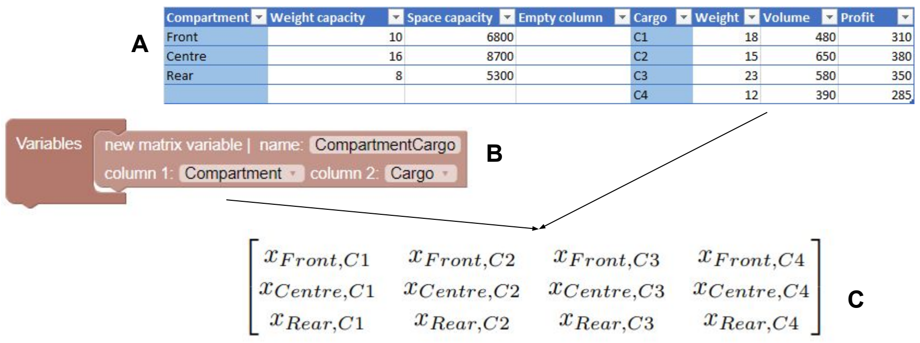
– **Variable blocks** (seen in Figure 2.A): Blocks for creating single, column and matrix variables.

- **Operation block** (seen in Figure 2.B): A block to construct an individual constraint.
- **Building blocks** (seen in Figure 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 2.D): A set of blocks to access the variables created before, that is, values blocks.

To further facilitate this process for novice and inexperienced users, when building a new linear programming model, the variables, constraints and objective blocks already appear and are connected in the workplace when creating a fresh solution. In the next sections we detail each of the blocks.

3.3 Defining variables

To define a mathematical linear programming model for 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.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.B we use such a construct to create the variables for the running example. In the example we use a the 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 column	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

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

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

Fig. 3. Create variables

LPBlocks offers several options to define new variables, using the blocks seen in Figure 2.A:

- single variables through its name;

– column variables defining its name and an index column for which the variable will be iterated and accessed;
– matrix variables that take a name and two index columns for which they can be iterated and those values accessed (used in Figure 3.B).

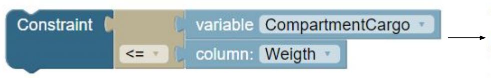
The process of generating the model variables is dependent on the variables block used:

– 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.B).

3.4 Defining constraints

The second step is 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 4.B. There are four constraints, one for each cargo. In each constraint, on the left-hand side of the inequality one should sum the 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.

A



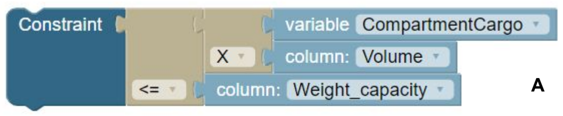
B

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

Fig. 4. Defining constraints for the cargoes weight

In LPBlocks, each constraint is defined by dragging a constraint block inside the constraints block (second and third blocks from the top in Figure 2.C) and then using the value blocks (blocks in Figure 2.D) and operation blocks (blocks following the constraint block in Figure 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.



A

B

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

Fig. 5. Defining constraints - second example

The variables can be accessed using different blocks and options. This as well as the way the columns are used influence how the constraints will be generated. As an example a user can access a matrix variable with a single slot variable block in Figure 4.A to generate multiple constraints or use the three slot variable blocks to access a particular value of the given variable.

Our solution possesses other features such as using the positioning and index columns of the variables and columns used in the constraint construction automatically deducing summations, sumproducts and sets of linear programming constraints from the high level user defined visual constraints.

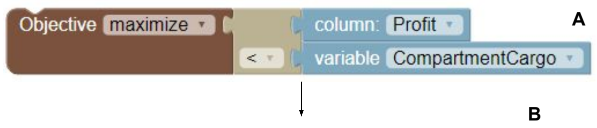
The first constraint in Figure 4.A is defined in our language by using: *i)* an `operation block` with the inequality sign <=; *ii)* a `variable block` with the option `CompartmentCargo` and; *iii)* a column block with the option `Weight`. Since the `constraints block` only appears after the `variables 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 expressed in Figure 4.B.

Another example constraint can be expressed in natural language as "the volume (space) capacity of each compartment must be respected". This constraint (in Figure 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 5.B.

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 fit the `objective block` into the `constraints block` and use several `value` and `operation blocks` to define the function.



Objective maximize < column: Profit variable CompartmentCargo

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

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

Fig. 6. Objective function for the running example

In the example seen in Figure 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 option `CompartmentCargo`. The objective function generated by this statement is the one featured in 6.B which would be the one written in a mathematical model.

3.6 Implementation

This language is included in a previously existing software prototype developed in the context of a work exploring the creation of systematic spreadsheet processes using a process-like notation [12,13]. This tool currently supports some of the operations associated with creating and editing spreadsheets such as adding columns, sorting, filtering or and creating charts. Our goal is to extend the current software with functionalities that would allow users to build correct and robust linear programming models.

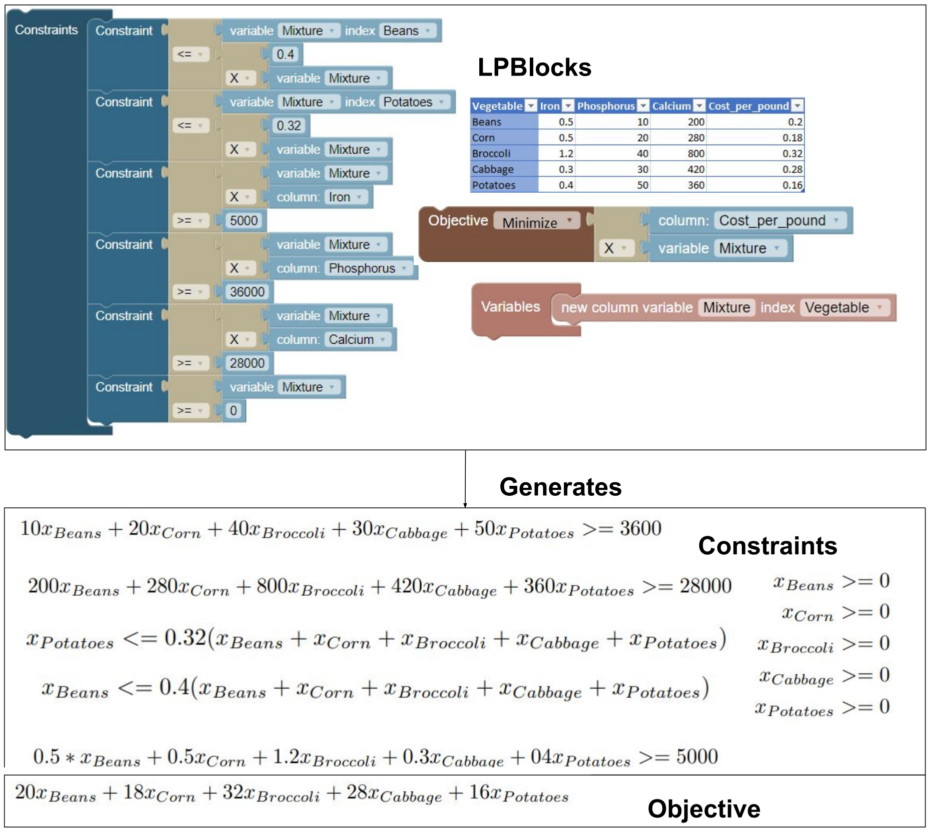
We use the Blockly framework to define our language syntax. This is then compiled into an OR-Tools valid linear programming model.

4 Language applicability

In this section we present a set of linear programming problems taken from an MBA exam [2] and from a Operations Research textbook [3] and modeled using LPBlocks. With this we intend to illustrate the applicability of our language to a broad set of examples. In Appendix A we include the solution in LPBlocks of other problems.

4.1 Vegetable mixture

This example shown in Figure 7 comes from the operations research textbook [3]. 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



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

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

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

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

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

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

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

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

Fig. 7. Definition of the vegetable mixture problem using LPBlocks

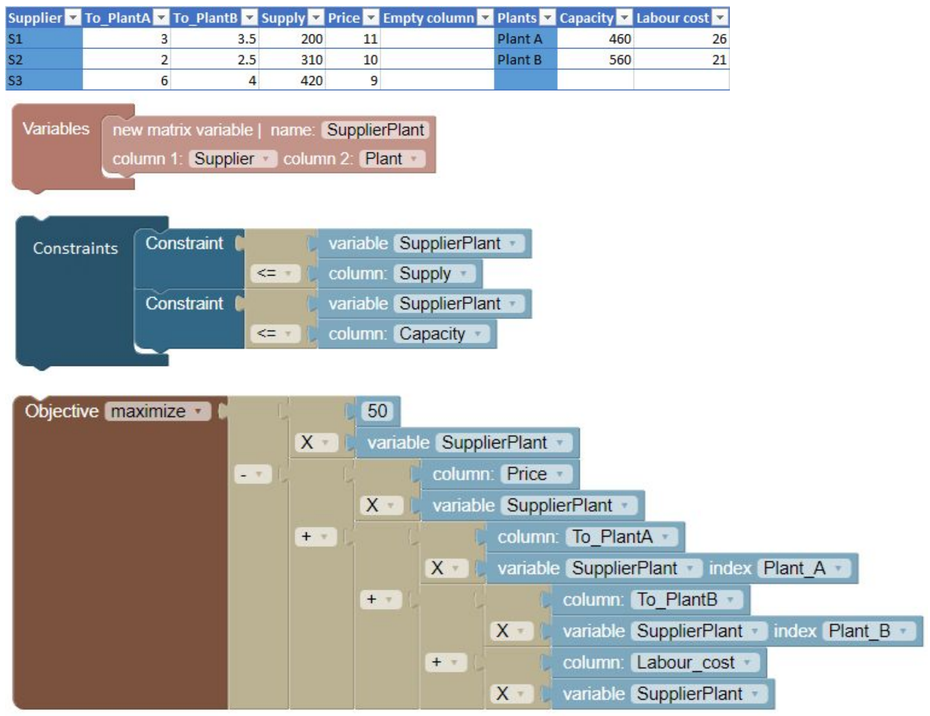
pound in the tabular data. We have a maximum percentage for certain vegetables and the lower bounds for certain nutrients.

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% for Beans and 32% for Potatoes. The following three constraints define the lower bounds for the given nutrients and the last adds non-negativity. The objective is to minimize the cost per pound of the mixture. The verbosity of this model could be improved if support for inputing data in the form of matrices in the spreadsheet was added. This would decrease the necessity of using single value blocks since iteration through both indexes would be possible.

When compared with implementing this model using mathematical notation LPBlocks allows for the creation of a more concise and intuitive model. We feel that our language lends itself to this sort of problem since it allows for the generation of sizable mathematical constraints using more concise business logic.

4.2 Fruit canning plants

In this example taken from an MBA exam [2] and shown in Figure 8 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 of \$50.



Supplier	To_PlantA	To_PlantB	Supply	Price	Empty column	Plants	Capacity	Labour cost
S1	3	3.5	200	11		Plant A	460	26
S2	2	2.5	310	10		Plant B	560	21
S3	6	4	420	9				

```
Variables
    new matrix variable | name: SupplierPlant
    column 1: Supplier  column 2: Plant

Constraints
    Constraint  variable SupplierPlant
                <=
                column: Supply
    Constraint  variable SupplierPlant
                <=
                column: Capacity

Objective maximize
    -
        X
            50
            variable SupplierPlant
        +
            X
                column: Price
                variable SupplierPlant
            +
                X
                    column: To_PlantA
                    variable SupplierPlant index Plant_A
                +
                    X
                        column: To_PlantB
                        variable SupplierPlant index Plant_B
                    X
                        column: Labour_cost
                        variable SupplierPlant
```

Fig. 8. Fruit canning plant example modeled using LPBlocks

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 considerably more complex since it needs to take into account the selling price and all the costs to represent the profit. The objective verbosity could be mitigated by adding the support for defining matrices inside the spreadsheet for the data. To improve the reusability of the created models adding support to define data variables inside the spreadsheet could also be done. To help users better visualize the model we will add support for intermediate values to store portions of more verbose components. More concretely in this model we could create blocks for the different operating costs used in the objective and reference these blocks at the moment of its definition.

When comparing with traditional mathematical notation or even other solutions LPBlocks allowed for considerable savings in terms of syntax for the expression of constraints however we found that due to its complexity the objective function specification in LPBlocks was similar to the mathematical form.

4.3 Machine allocation

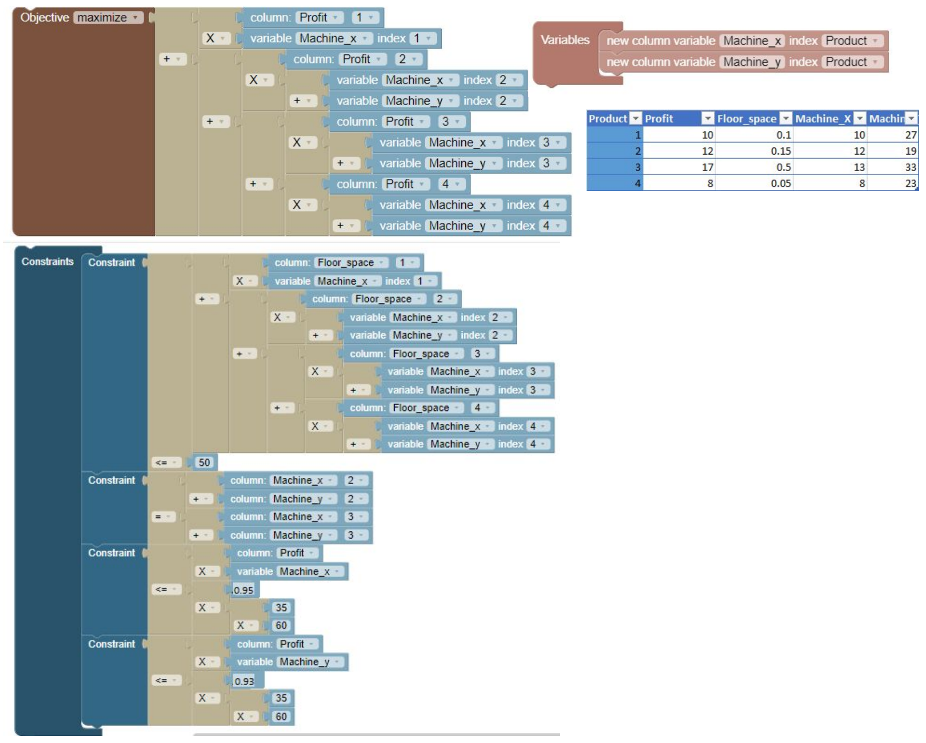
In the problem shown in Figure 9 (taken from [2]) the goal is to maximize a factory's profit by allocating the production of different goods among two machines. In this problem we are given information about each product's profitability, use of floor space and manufacturing time in minutes taken by each 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.

In this example we create `column variables` for each machine taking the `Proudct` column as the index as opposed to previous examples where we created `matrix variables`. In this we did not create a `matrix variable` as could be assumed do to the fact that the values for the machines are not used as index columns and using them for defining a `matrix variable` would mandate the referencing of one of the values for every use of the variable 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 of $50m^2$. If LPBlocks supported matrices as data input we possibly could express this constraint in a less verbose manner. In the second constraint we express that the production of product 2 is the same as 3. In constraints three and four we take into account the downtime of 5% for machine 1 and 7% for 2 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. This is the reason we were not able to use a more generic representation for this constraint.

Similarly to the previous example we found some benefits in terms of smaller footprint but since the creation of the model required the use of complex mathematical operations it couldn't make use of most of LPBlocks features.

4.4 Improving LPBlocks

We have specified 7 linear programming models using LPBlocks. Most of these examples were easily encoded using LPBlocks. However, we found that some improvements could be made in terms of usability, this center mainly on providing support for receiving data in the form of matrices and variables. In testing the



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

Fig. 9. Machine allocation problem in LPBlocks

language applicability the value of these changes was evident. Despite posing some challenges in terms of parsing and changes in the context of our application, this addition would greatly benefit the writing of less verbose and more concise programs in LPBlocks. Other features such as using intermediate blocks could add some benefit to users but come with possible negative impacts when it comes to ease of use and the creation of correct models.

5 Concluding remarks

In this work we present a language to aid end users defining linear programming models. With our language, users are forced to create correct programs as the constructs are based on the inputs of the problems and it possesses numerous advantages associated with Block-based such as having and forcing strict input, output , next and previous statement data types and having other features such as imposing the selection of various inputs from lists extracted from input spreadsheet. However, it is still possible to build models with problems and thus we intend to include error-handling in the support tool. We will also design and run empirical evaluations to assess the usability of the language.

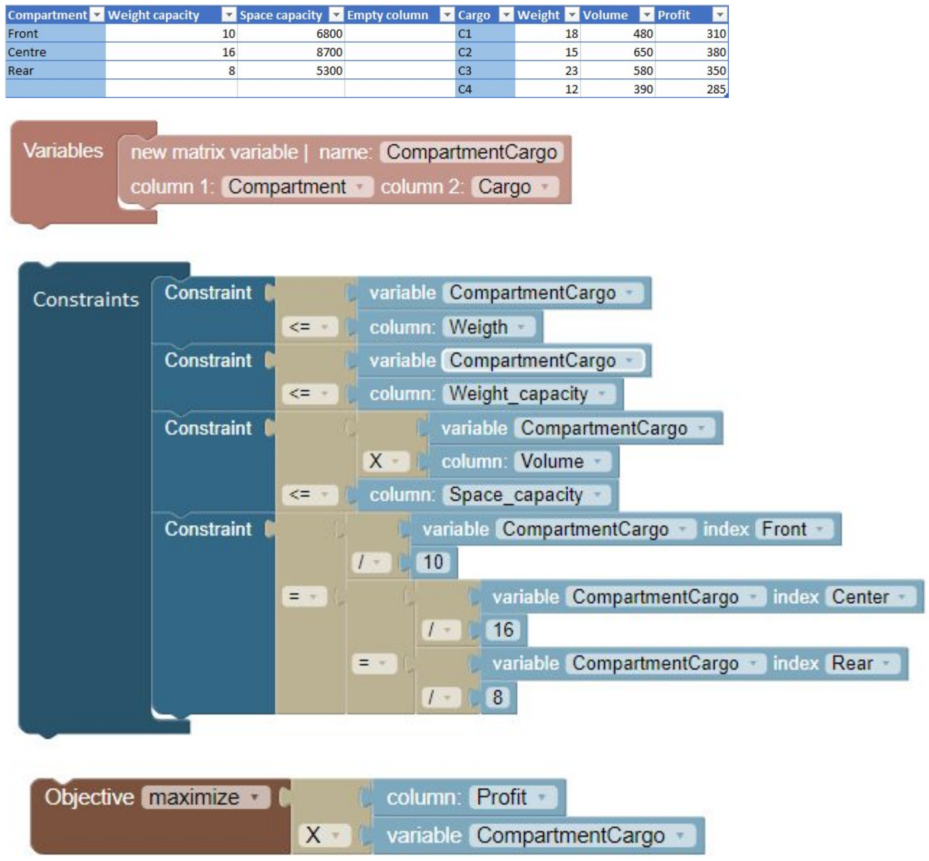
References

1. Bart, A.C., Tibau, J., Tilevich, E., Shaffer, C.A., Kafura, D.: Blockpy: An open access data-science environment for introductory programmers. Computer **50**(5), 18–26 (2017). https://doi.org/10.1109/MC.2017.132
2. Beasley, J.E.: Or-notes, `http://people.brunel.ac.uk/~mastjjb/jeb/or/lpmore.html`
3. Carter, M., Price, C.C.: Operations Research: A Practical Introduction. CRC Press (2000)
4. Chong, L.S., Xin, C.J.: Creating a gui solver for linear programming models in matlab. Journal of Science and Technology **10** (2018)
5. Fraser, N.: Ten things we've learned from blockly. In: 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond). pp. 49–50 (2015). https://doi.org/10.1109/BLOCKS.2015.7369000
6. da Gião, H., Cunha, J., Pereira, R.: Linear programming meets block-based languages. In: 2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (2021), to appear.
7. Guerrero, H.: Excel Data Analysis: Modeling and Simulation. Springer (2010), `https://www.springer.com/gp/book/9783642108341`
8. Krishnamoorthy, S.P., Kapila, V.: Using a visual programming environment and custom robots to learn c programming and k-12 stem concepts. In: Proceedings of the 6th Annual Conference on Creativity and Fabrication in Education. p. 41–48. FabLearn '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/3003397.3003403
9. Lazaridis, V., Paparrizos, K., Samaras, N., Sifaleras, A.: Visual linprog: A web-based educational software for linear programming. Comput. Appl. Eng. Educ. **15**(1), 1–14 (2007). https://doi.org/10.1002/cae.20084
10. Ma, P.C., Murphy, F.H., Stohr, E.A.: A graphics interface for linear programming. Commun. ACM **32**(8), 996–1012 (Aug 1989). https://doi.org/10.1145/65971.65978
11. Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E.: The scratch programming language and environment. ACM Trans. Comput. Educ. **10**(4) (Nov 2010). https://doi.org/10.1145/1868358.1868363
12. Mendes, J., Cunha, J., Duarte, F., Engels, G., Saraiva, J., Sauer, S.: Towards systematic spreadsheet construction processes. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). pp. 356–358 (2017). https://doi.org/10.1109/ICSE-C.2017.141
13. Mendes, J., Cunha, J., Duarte, F., Engels, G., Saraiva, J., Sauer, S.: Systematic spreadsheet construction processes. In: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 123–127 (2017). https://doi.org/10.1109/VLHCC.2017.8103459
14. Pasternak, E., Fenichel, R., Marshall, A.N.: Tips for creating a block language with blockly. In: 2017 IEEE Blocks and Beyond Workshop (B B). pp. 21–24 (2017). https://doi.org/10.1109/BLOCKS.2017.8120404
15. Patton, E.W., Tissenbaum, M., Harunani, F.: MIT App Inventor: Objectives, Design, and Development, pp. 31–49. Springer Singapore, Singapore (2019). https://doi.org/10.1007/978-981-13-6528-7_3
16. Pereira, J., Fernandes, S.: Two-variable linear programming: A graphical tool with mathematica. In: SYMCOMP 2013 - 1st International Conference on Algebraic and Symbolic Computation. pp. 159–173 (09 2013)

17. Senne, E., Lucas, C., Taylor, S.: Towards an intelligent graphical interface for linear programming modelling. Journal of Intelligent Systems **6**(1), 63–94 (1996). https://doi.org/doi:10.1515/JISYS.1996.6.1.63

A Examples of the use of LPBlocks

A.1 Cargo allocation



Compartment	Weight capacity	Space capacity	Empty column	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

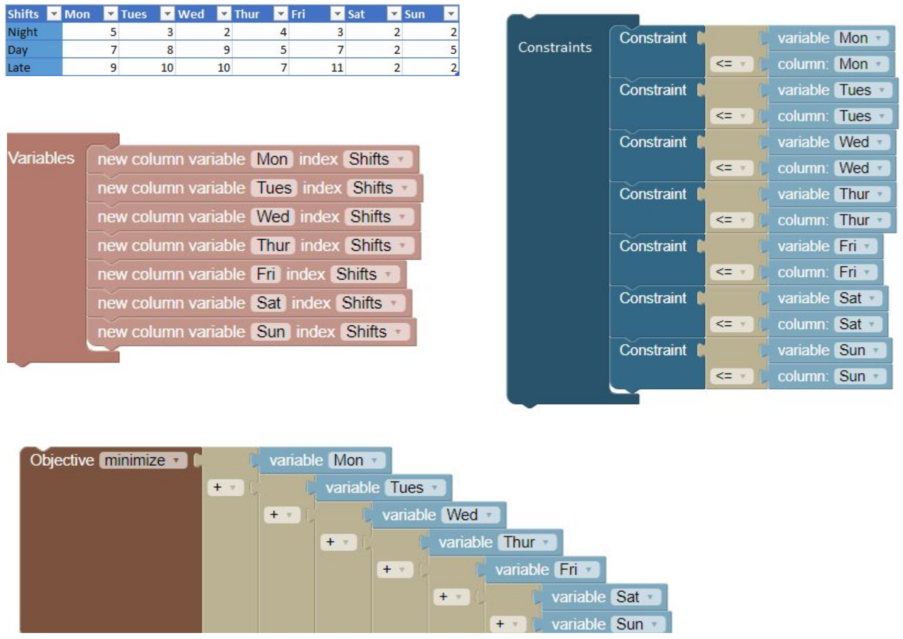
```
Variables  new matrix variable |  name: CompartmentCargo
           column 1: Compartment  column 2: Cargo

Constraints
  Constraint  variable CompartmentCargo
          <=  column: Weigth
  Constraint  variable CompartmentCargo
          <=  column: Weight_capacity
  Constraint      variable CompartmentCargo
              X   column: Volume
          <=  column: Space_capacity
  Constraint      variable CompartmentCargo  index Front
              /   10
          =           variable CompartmentCargo  index Center
                  /   16
              =       variable CompartmentCargo  index Rear
                  /   8

Objective maximize    column: Profit
                  X   variable CompartmentCargo
```

Fig. 10. Cargo allocation

A.2 Shift allocation



Shifts	Mon	Tues	Wed	Thur	Fri	Sat	Sun
Night	5	3	2	4	3	2	2
Day	7	8	9	5	7	2	5
Late	9	10	10	7	11	2	2

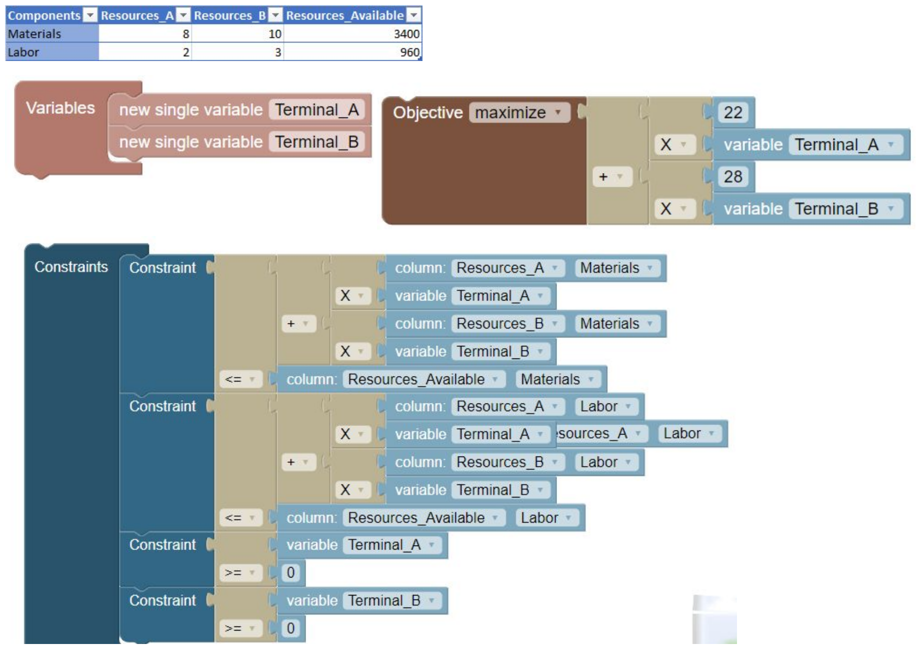
```
Variables
  new column variable Mon index Shifts
  new column variable Tues index Shifts
  new column variable Wed index Shifts
  new column variable Thur index Shifts
  new column variable Fri index Shifts
  new column variable Sat index Shifts
  new column variable Sun index Shifts

Constraints
  Constraint  variable Mon   <=  column: Mon
  Constraint  variable Tues  <=  column: Tues
  Constraint  variable Wed   <=  column: Wed
  Constraint  variable Thur  <=  column: Thur
  Constraint  variable Fri   <=  column: Fri
  Constraint  variable Sat   <=  column: Sat
  Constraint  variable Sun   <=  column: Sun

Objective minimize
  variable Mon
  + variable Tues
    + variable Wed
      + variable Thur
        + variable Fri
          + variable Sat
            + variable Sun
```

Fig. 11. Shift allocation problem

A.3 Terminal manufacture



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

```
Variables
    new single variable Terminal_A
    new single variable Terminal_B

Objective maximize
    +
        X   22
            variable Terminal_A
        X   28
            variable Terminal_B

Constraints
    Constraint
        <=
            +
                X   column: Resources_A  Materials
                    variable Terminal_A
                X   column: Resources_B  Materials
                    variable Terminal_B
            column: Resources_Available  Materials
    Constraint
        <=
            +
                X   column: Resources_A  Labor
                    variable Terminal_A  sources_A  Labor
                X   column: Resources_B  Labor
                    variable Terminal_B
            column: Resources_Available  Labor
    Constraint
        >=
            variable Terminal_A
            0
    Constraint
        >=
            variable Terminal_B
            0
```

Fig. 12. Terminal manufacture

A.4 Satellite launching

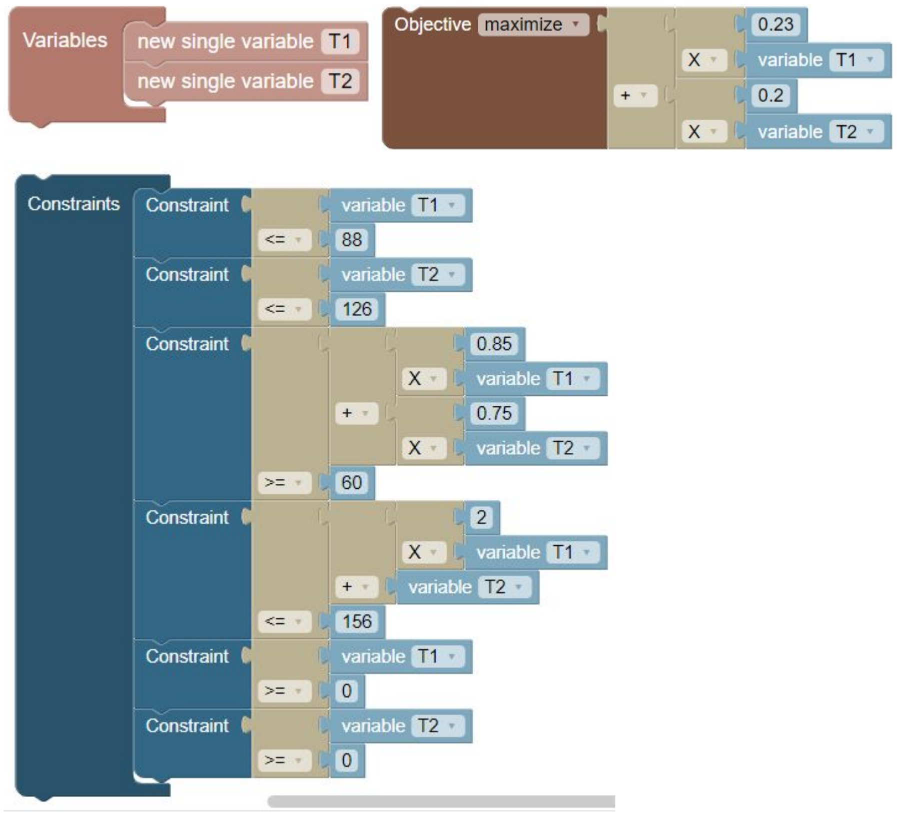


Fig. 13. Satellite launching