Module Interface Specification for a Library of Simplex Method Solvers (LoSMS)

Hanane Zlitni

November 24, 2018

1 Revision History

Date	Version	Notes
December 17, 2018	2.0	Final Draft (includes making the document consistent with the rest of the deliverables)
December 16, 2018	1.2	Applied Dr. Smith's Comments
December 06, 2018	1.2	Applied Olu Owojaiye's Comments Posted on GitHub
December 03, 2018	1.1	Corrected solveLP() and pivot() pseudo-code in 8.4.4 and 8.4.5
November 24, 2018	1.0	First draft

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at `https://github.com/hananezlitni/HZ-CAS741-Project/blob/master/docs/SRS/CA.pdf`.

The following are additional symbols, abbreviations or acronyms used in this document:

symbol	**description**
Z	Optimal solution(s) of the objective function
Z'	The negation of the objective function
n	A number in $[0, \mathbb{N})$ representing the rows in the simplex tableau
m	A number in $[0, \mathbb{N})$ representing the columns in the simplex tableau
x	A number in $\mathbb{N}$ representing the size of the list of optimal solutions

Contents

3 Introduction

The following document details the Module Interface Specifications for the Library of Simplex Method Solvers (LoSMS) tool.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at https://github.com/hananezlitni/HZ-CAS741-Project.

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | ... | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by LoSMS.

Data Type	**Notation**	**Description**
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)

The specification of LoSMS uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, LoSMS uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	**Level 2**
Hardware-Hiding Module	
Behaviour-Hiding Module	
Software Decision Module	Simplex Method Solver Exceptions

Table 1: Module Hierarchy

6 MIS of the Simplex Method Solver Module

6.1 Module

SimplexSolverADT

6.2 Uses

Exceptions

6.3 Syntax

6.3.1 Exported Constants

None

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
solveLP	$\mathbb{R}^{n*m}$, $\mathbb{R}^{n}$, $\mathbb{R}^{m}$, $\mathbb{R}^{m}$, gEnum	-	NO_OPTIMAL_SOLUTION
formSimplexTableau	-	-	-
toCanonical	-	-	-
isOptimumFound	-	boolean	-
getOptimum	-	$\mathbb{R}^{i}$	-

[There is no need to define a variable for the linear constraints type because whether it was an equality or inequality, slack/artificial variables would be added —HZ]

[This seems like important information to put in comments. Comments are not displayed with the final documentation, so important information needs to be included with the main text. —SS][done. —HZ]

[For these types you are defining, *lctEnum* and *gEnum*, please do not use integers that map to something meaningful. Mathematically you can define a set for elements of this type. A language like Python gives you enumerated types, so you can implement your new types easily. Hoffmann and Strooper show simple examples of defining new types. You will have something like $gEnum = \{\text{Min}, \text{Max}\}$. You can export these types, so that other modules can use them. If there are many new types like this, you can create a new modules whose purpose is to export the "global" types that you will be using. —SS]

[My comments about these explanation comments and types applies throughout your document. —SS]

6.4 Semantics

6.4.1 State Variables

- objcFunc:$\mathbb{R}^m$
- LCs:$\mathbb{R}^{n*m}$
- constants:$\mathbb{R}^m$
- goal:gEnum ; where gEnum = $\{MAX, MIN\}$
- sTableau:TableauT
- wasMin:boolean

6.4.2 Explanation

gEnum is an enumerated type representing the goal of the linear program: MAX for maximization and MIN for minimization.

The variable *wasMin* is a way to tell whether the original linear program was a minimization problem or not. If it's a minimization problem, the *goal* state variable will be changed to max and *wasMin* will be set to true. Then, the variable will be checked before outputting the solution- if it's True, the optimal solution Z will be multiplied by -1. Otherwise, it will remain as it is.

6.4.3 Environment Variables

None

6.4.4 Assumptions

None

6.4.5 Access Routine Semantics

solveLP(*LCs*, *constants*, *objcFunc*, *LCsType*, *goal*):

- transition:
    1. ($\neg$(self.goal = MAX) $\Rightarrow$ self.wasMin = True, self.objcFunc = self.objcFunc * -1, self.goal = MAX)
    2. self.formSimplexTableau()
    3. self.updateEnteringDepartingVars(self.sTableau)
    4. self.findPivot()

5. (pivot $< 0 \Rightarrow$ NO_OPTIMAL_SOLUTION)
6. self.pivot(pivotIndex)
7. Repeat 4, 5 and 6 until there are no negative values in the last row (excluding the last column)
8. self.getOptimum()

- output: -
- exception: exc :=

  (self.getOptimum() = 0 $\vee$ self.getOptimum() = "" $\Rightarrow$ NO_OPTIMAL_SOLUTION)

formSimplexTableau():

- transition:

  1. Initialize *sTableau*
  2. Update its state by appending *LCs* to it
  3. Call toCanonical() to convert the tableau to the canonical form by adding the slack/artificial variables
  4. Update the state of *sTableau* by appending *objcFunc* to its bottom row
  5. Update the state of *sTableau* by appending *constants* to its last column

- output: -
- exception: -

toCanonical():

- transition:

  1. Initialize *slackVariables* matrix
  2. Initialize *row* list
  3. Compare between two counters i and j
  4. ($i = j \Rightarrow$ append 1 to *row* $\mid i \neq j \Rightarrow$ append 0 to *row*)
  5. Repeat 3 and 4 for the length of *sTableau*
  6. Append *row* to *slackVariables*
  7. Update the state of *sTableau* by appending *slackVariables* to it

- output: -
- exception: -

getOptimum():

- transition:
    1. Initialize the tuple *optimum*
    2. Get the optimal solution and the points they occur from *sTableau* and set *optimum* to the values
    3. (wasMin = True ⇒ optimum['z'] = optimum['z'] * -1)
- output: *out* := optimum
- exception: -

isOptimumFound()

- transition:
    1. Initialize the variable *optimumFound* and set it to True
    2. Look for negative values in the last row of *sTableau* excluding the last column and set *optimumFound* to False if found
- output: *out* := optimumFound
- exception: -

6.4.6 Local Functions

getEnteringVariable():

```
    #find the most negative value in the last row of sTableau

        start
            initialize lastRow
            lastRow = self.sTableau[len(sTableau) - 1]
            initialize iMostNegative
            iMostNegative = 0
            initialize mostNegative
            mostNegative = lastRow[iMostNegative]
            for each entry in lastRow
                if entry < mostNegative
                    if mostNegative = entry
                update iMostNegative
            return iMostNegative
        end
```

findPivot():

```
        start
            return self.getEnteringVariable(), self.getDepartingVariable()
```

```
    end

pivot($iPivot$):
    start
        set $i$, $j$ to $iPivot$
        set $pivot$ to $sTableau[i][j]$
        for each row in pivot column
            perform a row operation to make entry 0
    end
```

[You shouldn't be defining types here. Seeing that these types are coming up again, I suggest you add a module to your design that exports types. —SS]

[Why do you have two modules with the same state variables (goal etc.). This makes it seem like the design is not completely thought out. Why can't your Tableau module use your input module to get the values it needs? Or maybe you don't need an input module. As we discussed in class, most of your design could be encapsulated in the tableau module. —SS]

[Add a brief statement on what each state variables is. I remember what Z is, but I forget what K means. —SS]

[Using the types $\mathbb{R}^x$ is confusing, since x usually represents a real value. —SS]

[Would it be easier if you added a solver method to your tableau module? —SS]

[I have updated the MIS to reflect the changes in the design. —HZ]

7 MIS of the Exceptions Module

7.1 Module

Exceptions

7.2 Uses

None

7.3 Syntax

7.3.1 Exported Constants

None

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
MISSING_INPUT	Error	-	-
NO_OPTIMAL_SOLUTION	Error	-	-

7.4 Semantics

7.4.1 State Variables

None

7.4.2 Environment Variables

None

7.4.3 Assumptions

None

7.4.4 Access Routine Semantics

MISSING_INPUT():

- transition:
- output: *out* := "At least one input is missing."
- exception: -

NO_OPTIMAL_SOLUTION():

- transition:
- output: *out* := “This linear program does not have an optimal solution.”
- exception: -

7.4.5 Local Functions

None

References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL `http://citeseer.ist.psu.edu/428727.html`.

8 Appendix

There are no additional information to provide.