Linear solver with infinity numbers (Linsol)

Author: Dimo Chanev
Supervisiors: Doychin Boyadzhiev
Emil Kelevejiev

August 17, 2017

Contents

1 Introduction

1.1 Introduction to the infinity numbers

Let's try to imagine that there is an absolute infinity (∞) (not infinity as a limit but true infinity).

And let's thing a bit what the common reason says:

- Zero multiplied by any number is zero
- Infinity multiplied by any number is $\pm$ infinity

So what is the result of 0∞? This is a contradiction so it is undefined?
Here the infinity numbers come to help us. What is infinity number you might ask?
Let's define **infinity number**. It is a number represented as

$$(a; b) = a + b\infty, where\ a \in \mathbb{R}, b \in \{-1, 0, 1\}$$

∞ is just a symbol, representing the intuitive concept of infinity.

In the above notation:
"a" is called coefficient for the real part.
"b" is called the coefficient for the infinity part.

We introduce this relations ($> < = \neq$) definig that:

$$for\ every\ x,\ y \in \mathbb{R}$$
$$(x; 1) > (y; 0)$$
$$(x; 1) > (y; -1)$$
$$(x; 0) > (y; -1)$$
$$(x; -1) = (y; -1)$$
$$(x; 1) = (y; 1)$$
$$(x; 0) \neq (y; 0),\ where\ x \neq y$$

Let's define the basic mathematical operations with them:

- Addition:

  $(a;b)+(c;d) = a+b\infty+c+d\infty = a+c+(b+d)\infty = (a+c;b+d)$

- Subtraction:

  $(a;b)-(c;d) = a+b\infty-c-d\infty = a-c+(b-d)\infty = (a-c;b-d)$

- Multiplication:

  $(a;b)\times(c;d) = (a+b\infty)\times(c+d\infty) = ac+ad\infty+bc\infty+bd\infty^2 =$

  $ac+ad\infty+bc\infty+bd\infty = ac+(ad+bc+bd)\infty = (ac;ad+bc+bd)$

- Division:

$$\frac{(a;b)}{(c;d)} = \frac{a+b\infty}{c+d\infty} = \frac{b}{d} = (\frac{b}{d};0), d \neq 0$$

$$\frac{(a;b)}{(c;0)} = \frac{a+b\infty}{c} = \frac{a}{c} + \frac{b}{c}\infty = (\frac{a}{c};\frac{b}{c}), d = 0$$

Important remark:

- When $b > 0$ in $(a;b)$ we substitude b by 1
- When $b < 0$ in $(a;b)$ we substitude b by -1

Some examples:

$$(1;0) + (2;1) = (3,1)$$

$$(1.3;-1) - (2;1) = (-0.7;-2) = (3,-1)$$

$$(10;-1) \times (0.1;1) = (1;8.9) = (1,1)$$

$$\frac{(10;-1)}{(0.1;1)} = (-1,0)$$

$$\frac{(10;-1)}{(0.1;0)} = (1,-0.1) = (1,-1)$$

1.2 Introduction to Linsol

Linsol is a library written in Rust programming language which implements the algorithm of the Simplex method to find the minimum or maximum of a linear function with given linear constraints. Let's describe the algorithm for finding the minimum of the function:
First of all we transform the problem into canonical form (each inequality is transformed into equality and there cannot be negative numbers for the free coefficients). This is done in the following way:
$ax + by \geq c$ **can be written as** $ax + by - z = c$ **and z is inserted in the target function with coefficient 0;**
$ax + by \leq c$ **can be written as** $ax + by + z = c$ **and z is inserted in the target function with coefficient 0;**
$ax + by = -c$ **can be written as** $-ax - by = c$
After the canonical form is applied to the problem we must apply the base form (the M form) which has the following condition to be done:
"In each constraint there must be a variable with coefficient 1 and coefficient 0 in the other constraints."
The most common way of doing this is to add a variable with coefficient 1 for each of the contraint and with coefficient 0 for all the others and put it with coefficient ∞ in the target function. Here the infinity numbers come to help to store the data and process it easily.
After that we construct the simplex table, which in every is stored an constraint like coefficients for the variables

in all the others (if a variable is not in the constraint it participates there with coefficient 0). Here we assume that the coefficients are the values of the variables. Then we calculate the deltas:

$$\Delta_V = \sum_{i=0}^{n} V_i b_i$$

where n *is the number of constraints numbered from* 0

V_i *is the value of the variable* V *in the constraint with number* i

and b_i *is the free coefficient in the constraint with number* i

Then we check the optimality condition, which is:

$$\nexists \Delta > 0$$

If it is true the problem is solved for these values of the variables. After that we check the limitlessness condition which is:

$$\exists \Delta_V > 0 \; where \; \nexists V > 0$$

If it is true the minimal value of the target function with the given constraints is $-\infty$.
If the both conditions aren't met we can reconstruct the table to get closer to the result (move in a neighbouring point in the polygon of intersection) as follows:

Select V *where* $\Delta_V \to max$ *: this variable will get into the bases*

Select i *where* $\frac{b_i}{\alpha_{iV}} \to min$; $\alpha_{iV} > 0$ *: this variable will get out the bases*

The variables have already been in the base must have $\Delta = 0$ and $\alpha_{iV} = 1$ and the other $\alpha_{qV} = 0$. So we must make some transformations to do that

2 Description of the library implementation

2.1 InfNum

InfNum is the type that implements the infinity numbers.

2.1.1 new () -> InfNum

Returns the infinity number (0.0; 0.0).

2.1.2 from (real: f64, inf: f64) -> InfNum

Returns the infinity number (real; inf).

2.1.3 Redefining +, -, *, /

Returns the result of the corresponding operation.
Example:
let a = InfNum::from (1.2, 1.0);
let b = InfNum::from (2.2, 1.0);
let c = a + b; // c = InfNum::from (3.4, 1.0)

2.1.4 Redefining +=, -=, *=, /=

Sets the results from the corresponding operation to the left variable.
Example:
let mut a = InfNum::from (1.2, 1.0);
a -= InfNum::from (2.2, 1.0); // a = InfNum::from (-1.0, 0.0)

2.2 Function

Function is the type that implements a function with infinity numbers.

2.2.1 new ()-> Function

Returns a basic function (no variables and coefficients).

2.2.2 add_variable (name: String, coefficient: InfNum) -> bool

Pushes a variable and its' coefficient to the function.

2.2.3 change_coefficient (name: String, new_coefficient: InfNum)

Sets the coefficient in front of the variable.

2.2.4 get_coefficient (name: String) -> InfNum

Returns the coefficient in front of the variable.
Example:
let mut a = Function::new ();
a.add_variable (String::from ("x"), InfNum::from (1.0, 0.0));
a.add_variable (String::from ("y"), InfNum::from (-1.0, 0.0));
a.change_coefficient (String::from ("y"), InfNum::from (-2.0, 0.0));
let c = a.get_coefficient (String::from ("x")); // InfNum::from (1.0, 0.0)
let d = a.get_coefficient (String::from ("y")); // InfNum::from (-2.0, 0.0)

2.2.5 get_value (values: &$HashMap < String, InfNum >$)-> InfNum

Calculates and gives the result formed my replacing the variables as described in the HashHap.
Example:
let mut inst = Function::new ();
inst.add_variable (String::from ("y"), InfNum::from (2.0, -1.0));
let mut vals = HashMap::$< String, InfNum >$::new ();
vals.insert (String::from ("y"), InfNum::from (1.0, 0.0));
inst.get_value (&vals) // InfNum::from (2.0, -1.0)

2.3 Constraint

Constraint is the type that is for modeling constraints.

2.3.1 new ()-> Constraint

Returns a basic constraint (new function, equality, and free coefficient as new InfNum).

2.3.2 from (left: Function, sign: Sign, right: InfNum)-> Constraint

Returns a Constraint constructed with components as the given ones.

2.3.3 check (variables: &$HashMap < String, InfNum >$) -> bool

Checks if the Constraint is met with the given values for the variables.
Example:

```
let mut inst = Consraint::new ();
inst.left.add_variable (
String::from ("x"),
InfNum::from (1.0, 1.0),
);
inst.left.add_variable (
String::from ("y"),
InfNum::from (1.0, 0.5),
);
inst.sign = Sign::GreaterOrEqual;
inst.right = InfNum::from (-1.0, 0.0);
let mut vals = HashMap::< String, InfNum >::new ();
vals.insert (String::from ("x"), InfNum::from (0.0, 0.0));
vals.insert (String::from ("y"), InfNum::from (0.0, 1.0));
inst.check (&vals) // true
```

2.4 Solver

Constraint is the type that is for modeling the problems.

2.4.1 new ()-> Solver

Returns a basic constraints (new function, equality, and free coefficient as new InfNum).

2.4.2 from (function: Function, value: TargetValue, constraints: Vec< *Consraint* >)

Returns a Solver constructed with components as the given ones.

2.4.3 canonical_form ()

Makes the problem in the canonical form.

2.4.4 base_form ()-> Vec< *InfNum* >

Makes the problem in the base form and returns a vector containing the base variables.

2.4.5 get_simplex_table ()-> SimplexTable

Makes the problem in canonical form, then in base form and constructs the simplex table.

2.4.6 check_optimality (table: &SimplexTable)-> bool

Returns true if the data describing the problem is optimal (the problem is solved). Otherwise returns false.

2.4.7 check_limitlessness (table: &SimplexTable)-> bool

Returns true if the problem is unlimited (no solution). Otherwise returns false.

2.4.8 improve_table (table: &mut SimplexTable)

Improves the table as described above.

3 Future

First of all I have to make my code work. After that i will optimize the memory consumption and the speed of the algorithm. When this is done I'm planning to implement integer solving (to return only itegers as values of the variables). While I am doing all of this I will clarify the conception of the infinity numbers.

4 Code

GitHub: https://github.com/Ro6afF/linsol
BitBucket: https://bitbucket.org/Ro6afF/linsol

5 Acknowledgments

Special thanks to:

- Doychin Boyadjiev for clarifying and improvement the idea and explaining the algorithm.
- Emil Kelevejiev for improvement of the idea and documentation

Thanks also to:

- Bulgarian Academy of Sciences
- High School Students Institute of Mathematics and Informatics
- Sofia High School of Mathematics