# Linear Programming with *Jalinopt*

Luc Hogie (CNRS/INRIA/UNS)

April 22, 2013

## 1 Introduction

### 1.1 The idea

Many mathematical and engineering problems can be expressed as linear programs, and doing so facilitates their resolution. Indeed it is generally more convenient to transform a domain-specific problem into a linear-optimizable one (that can be solved by any solver) than writing a complex domain-specific algorithm.

*Jalinopt* is a Java toolkit for building and solving linear programs. It consists of a straightforward object-oriented model for linear programs, as well as a bridge to most common solvers, including GLPK and CPLEX.

Altought *Jalinopt* is inspired by Mascopt and JavaILP, it provides a significantly different OO model and an utterly different approach to connecting to the solver. In particular this approach offers better portability and the possibility to connect (via SSH) to solvers on remote computers.

The working process of *Jalinopt* is the following:

1. the user defines a linear program as a Java object;
2. *Grph* translates this object into a plain CPLEX LP text;
3. this LP text is piped via standard input to a solver that is executed in another process;
4. on completion of the resolution, the standard output of the solver is piped back to *Grph*;
5. *Grph* parses this output text and builds a result in the form of a Java object;
6. this result object is given to the user, who is invited to extract the information he needs to solve his graph problem.

Only items 1 and 6 require the intervention of the programmer.

## 1.2 JNI vs. piping

Weirdly enough, there does not exist Java implementation of linear solvers. At most one can find implementations of the simplex method. A Java linear solver would be worth having because this would solve portability issues.

As a consequence, Java programs dealing with linear optimization need to interact with native node. Such interaction can be achieve in different ways. Let us consider Java Native Invocation (JNI) and piping.

JNI is a feature of the Java Virtual Machine (JVM) that allows Java programs to run native code in the same process at the JVM itself. To do so, the JVM first dynamically loads a library file and connect Java methods to native functions. JNI is the solution proposed by most linear solvers by providing the Java developer with the adequate Java JNI-enabled interfaces. Unfortunately, in practise JNI's dynamic linkage of natively compiled libraries is a problem: the great variety of processors, operating systems types and versions imposes that JNI-based Java programs must embed all possible JNI-loadable libraries, which proves impracticable.

Piping is an inter-process communication scheme most known in the UNIX family of operation systems. A pipe is an unidirectional binary communication channel. The communication between the source process and the destination one all happens in memory using sophisticated memory management algorithms, making the transfer of data highly efficient. Although Java does not provide standard API for inter-process pipes, they can be found on all operation systems and hence are a very portable solution.

Pipe-based communication imposes that involved processed agree on a coding of the data exchanged. This imposes that the calling process:

- transforms the data to be transferred to a sequence of bytes that the native process can understand;
- parses the data coming from the native process.

Speaking about performance, whether one resort to JNI or piping, the same amount of data must be transferred to the native process. In both case, an adaptation of the data must be done: while JNI must convert types and encoding in both direction, piping-based solutions must convert bytes to native data and vice-versa.

## 1.3 State of the Art

Mascopt is a graph problems optimization library. Among numerous features, it comes with a LP manipulation API. Calls to solvers are done through JNI.

JavaILP provides a clean object-oriented model of linear problems and features bridges to many solvers. Its model has an unnecessary complex representation of variable types, and its hash-table based implementation may pose problems in the cas of large LPs. Also, just like Mascopt, calls to solvers are done through JNI.

# 2 Model for linear programs

This section presents the API for generating linear programs. Please note that this API is not meant to be easy to read/write by humans (this requirement is perfectly met by the standard form of LPs). Instead it is meant to be programmatically handy, making the translation of domain-specific problems into linear ones as natural as possible.

## 2.1 Creating a new problem

```
1 LP lp = new LP();
```

## 2.2 Setting the objective

```
1 LinearExpression objective = lp.getObjective();
2 objective.addTerm(coefficient, variable);
```

## 2.3 Setting the type of optimization

```
1 lp.setOptimizationType("min");
```

## 2.4 Adding a new constraint

A constraint is a triplet $lhs, operation, scalar$. For example:

$$4a + b + 2c > 10$$

```
1 Constraint constraint = lp.addConstraint();
2 LinearExpression lhs = constraint.getLeftHandSide();
3 lhs.addTerm(4, "a");
4 lhs.addTerm(1, "b");
5 lhs.addTerm(2, "c");
6 constraint.setOperator(">");
7 constraint.setRightHandSide(10);
```

*Jalinopt* accepts this shorter form for the definition of constaints:

```
1 Constraint constraint = lp.addConstraint(">", 10);
2 constraint.addTerm(4, "a");
3 constraint.addTerm(1, "b");
4 constraint.addTerm(2, "c");
```

## 2.5 Defining the type of the variables

# 3 Solving a problem

The resolution of a problem gives the optimal value for the objective function, as well as the value of each of the variables when the objective is optimized.

## 3.1 Using the default solver

Once a problem is defined, it can be solved. To do so, one need to

```
1 Result r = lp.solve();
```

The `Result` object then encapsulates the value of the optimized objective as well as the value of every variable which was used to reach the objective.

The default solver is defined by iteratively looking for the following components, and by using the first found:

1. local installation of CPLEX;
2. an installation of CPLEX on any of the SSH-accessible hosts defined in the `/.jalinopt/hosts` file;
3. the Java Apache LP Solver (but this one does not support ILP).

*Jalinopt* first checks if CPLEX is installed on the local host. To do so it looks at the `PATH` environment variable. If the CPLEX executable cannot be found, *Jalinopt* considers the `LPSolver.CPLEX_HOST` variable. If this variable is set to null

## 3.2 Forcing the solver

```
1 Result r = lp.solve(new LocalCplex());
```

Available servers include:

**jalinopt.ApacheSolver** the solver that comes with the Apache Commons optimization toolkit;

**jalinopt.cplex.LocalCPLEX** uses the cplex application, that is installed on the local computer; the `cplex` command must be in the system PATH;

**jalinopt.cplex.CPLEX_SSH** uses the cplex application installed on another computer; the SSH environment should be configured to as the user can connect to the remote host without having to type his password.

## 3.3 On a remote computer

This feature assumes than SSH is properly configured, so that the user does not need to provide a password when connecting to the remote computer. It also assumes that the solver executable is located in the PATH on this remote computer.

For the moment, only CPLEX can be invoked in a remote fashion. To process, one need to invoke the `CPLEX_SSH` solver. This solver needs information on which host to connect to, and the username on this host. This information is passed to the solver by using a `SSHConnectionInfo` object. This object needs a hostname and a user name. If the user name is omitted, *Jalinopt* considers the local username.

The following example solve the given LP using CPLEX on the host `musclotte`:

```
SSHConnectionInfo sshInfo = new SSHConnectionInfo("musclotte");
Result r = lp.solve(new CPLEX_SSH());
```

The following shortcut is also accepted:

```
Result r = lp.solve(new CPLEX_SSH("musclotte")));
```

# 4 Example

Let us consider the following LP.

```
Minimize
        obj: 3x + 2y
Subject To
        4x + y < 15
        5y > 10
General
        y
End
```

This LP can be programmatically modelled by the following Java code:

```
        LP lp = new LP();
        lp.setOptimizationType(OptimizationType.MIN);

        // define the objective
        LinearExpression objective = lp.getObjective();
        objective.addTerm(3, "x");
        objective.addTerm(2, "y");

        // 4x + 1y < 15
        Constraint c1 = lp.addConstraint("<", 15);
        c1.getLeftHandSide().addTerm(4, "x");
        c1.getLeftHandSide().addTerm(1, "y");

        // another way to formulate a constraint (5y > 10)
        Constraint c2 = lp.addConstraint(">", 10);
        c2.getLeftHandSide().addTerm(5, "y");

        // define the type for variable
        lp.getVariableByName("y").setType(TYPE.INTEGER);
```