Solving Systems of Affine (In)Equalities: PIP's User's Guide

Paul Feautrier

Additions by Jean-François Collard and Cédric Bastoul
Fourth Version, October 18, 2001

Abstract

This document is the User's Manual of PIP, a program which solves Parametric Integer Programming problems. That is, PIP finds the lexicographic minimum of the set of integer points which lie inside a convex polyhedron, when that polyhedron depends linearly on one or more integral parameters.

1 Introduction

The semantic analysis of programs accessing arrays often boils down to finding integer solutions to parametric linear programming problems. This is due to two main phenomena:

- Array subscripts are very often linear functions of surrounding loop counters ;
- The program's execution order enforces an order on possible solutions.

Let us consider the following example:

```
for i:= 0 to m do
    for j := 0 to n do              {I}
        a[2*i+j] := i+j;
```

After completion of execution, for which values of k is `A[`k`]` defined, and which instances of the assignment wrote into this array element? We can easily check that answering this question is equivalent to finding the solutions of the following system, where i, j and k are the unknowns:

$$0 \leq \quad i \quad \leq m, \tag{1}$$
$$0 \leq \quad j \quad \leq n, \tag{2}$$
$$2i + j \quad = \quad k. \tag{3}$$

Moreover, if we want to know which instance gave its *final* value to `A[`k`]`, that is if we are looking for the *last* instance writing into `A[`k`]`, then we have to look for the maximal value of vector (i, j) according to lexicographic order. We thus consider the following *polyhedron* $\mathcal{F}(k, m, n)$:

$$\mathcal{F}(k, m, n) = \{< i, j > | 0 \leq i \leq m, 0 \leq j \leq n, 2i + j = k\}. \quad (4)$$

What is the lexicographical maximum of the integer-valued vectors included in $\mathcal{F}(k, m, n)$? The aim of PIP is to solve such problems. The reader is refered to [**?**] for a mathematical description of the method.

1.1 General formulation

Let $\mathcal{F}$ be a polyhedron:

$$\mathcal{F}(\vec{z}) = \{\vec{x} | \vec{x} \geq 0, \mathbf{A}\vec{x} + \mathbf{B}\vec{z} + \vec{c} \geq 0\}. \quad (5)$$

In this formula, $\vec{x}$ is a vector with n entries: the vector of all unknowns. $\vec{z}$, $\vec{z} \geq 0$, is the vector built from parameters and has p entries. Polyhedron $\mathcal{F}(\vec{z})$ is a subset of $\mathbf{R}^n$ and is defined by $n + l$ inequalities: n inequalities expressing

$$\vec{x} \geq 0$$

and the l inequalities correponding to rows of matrix $\mathbf{A}$ of size $l \times n$, matrix $\mathbf{B}$ of size $l \times p$, and constant vector $\vec{c}$ of size l.

Size parameters can themselves be constrained by a set of affine inequalities

$$\mathbf{M}\vec{z} + \vec{h} \geq 0,$$

which is called the *context* of the problem. $\mathbf{M}$ is an $m \times p$ matrix and $\vec{h}$ a vector of dimension m. All data of a PIP problem: $(\mathbf{A}, \mathbf{B}, \mathbf{M}, \vec{c}, \vec{h})$ are assumed to be integer-valued.

2 Using the PIP Software

2.1 Writing the Input File

The input text file follows the following context-free grammar:

Grammar 1 *:*

```
File     ::= Problem ...
Problem  ::= ( Comments Nn Np Nl Nm Bg Nq Tableau Context )
Comments ::= List
List     ::= Atom | ( List ... )
```

```
Tableau  ::= ( Vector ... )
Context  ::= ( Vector ... )
Vector   ::= #[ Integer ... ]
Nn ::= Integer
Np ::= Integer
Nl ::= Integer
Nm ::= Integer
Bg ::= Integer
Nq ::= 0 | 1
```

This syntax was chosen so as to ease the generation of problems by a Lisp program. In particular, each `Problem` is a balanced list, as far as both parentheses and brackets are concerned.

- `Comments` are arbitrary lists. These comments are written verbatim to the output file, and are useful to keep track of problems and solutions.

  Note that several `Problem`s can be given to PIP in the same file. The problems may be separated by any text that does not contain a parenthesis. By using Unix FIFO's as input and output files, it is easy to convert the present implementation of PIP into a linear programming server.

- `Nn` is the number of unknowns in the program (which was denoted by n in the first section).

- `Np` is the number of (symbolic) parameters (p)

- `Nl` is the number of inequalities defining the domain of the unknowns (l).

- `Nm` is the number of inequalities satisfied by the parameters (m).

- `Bg` is the index of a "Big" parameter whose value is assumed to be infinitely large. That is, if the big parameter appears with a positive coefficient in a form ϕ, then we can immediately deduce that $\phi > 0$. If `Bg` is set to a nonpositive value, then there is no big parameter in the problem to be solved.

  Be aware that `Bg` is the column rank of the corresponding parameter in the `Tableau`, and that the first valid value for it is `Nn+1`.

- `Nq` is an integer but should be interpreted as a boolean value *à la* C, that is, it denotes "true" if its value is nonzero. If `Nq` is true, then an integer-valued solution is looked for. Otherwise, PIP finds the lexicographic minimum rational solution to the problem.

- `Tableau` stores the set of inequalities defining the domain of unknowns. Each `Vector` represents one inequality. The entries in `Vector` are, in this order:

- the coefficients of the unknowns (I.e., a row of matrix $\mathbf{A}$),
- the (additive) constant, (I.e., an entry of vector $\vec{c}$),
- the coefficients of the parameters (I.e., a row of matrix $\mathbf{B}$)

This notation heavily depends on the positions given to unknowns and parameters: it is the responsability of the user to enforce a coherent ordering of coefficients and to set a coefficient to zero when the corresponding unknown/parameter does not appear.

There are l such `Vector`s in `Tableau`, and each `vector` exactly has $n+1+p$ entries.

- In a similar way, `Context` is a list of `Vector`s. Each `Vector` represents a row of Matrix $\mathbf{M}$ followed by the corresponding entry in vector $\vec{h}$. `Context` thus includes m `Vector`s of $p+1$ entries.

2.1.1 Example

This example is taken from [**?**]. We consider the loop nest below:

```
for i:= 0 to m do
   for j := 0 to n do                {II}
      for k := 0 to i+j do ...
```

and we wish to rewrite this nest in the order `k, j, i`. The bounds on `k` can easily be guessed ($0 \leq k \leq m+n$), so let's look for the lower bound on `j` in the rewritten nest. This lower bound on `j` can be found by solving the following problem:

$$\mathcal{D}_2(k) = \{ <j,i> \ | i \leq m, j \leq n, k \leq i+j \}.$$

This problem is to be solved in the context $k \leq m+n$. The input file may thus look like this:

```
( (Lower bound on j after loop inversion
     (unknowns j i)
     (parameters k m n))
  2 3 3 1 -1 1
  ( #[0 -1 0 0 1 0]
    #[-1 0 0 0 0 1]
    #[1 1 0 -1 0 0]
   )
  ( #[-1 1 1 0])
)
```

The first sequence of integers should be read as: This problem has 2 unknowns (i and j) and 3 parameters (k, m and n). The domain is defined by 3 inequalities, the context by 1 inequality. There is no (-1) big parameter and it is true (1) that we are looking for an integer solution.

2.2 Calling PIP

PIP is called by the following command:

```
pip [-s|-v] [-z] [input [output]]
```

PIP prints some information on the screen after having solved a problem. The `-s` (silent mode) switches this feature off. On the contrary, the verbose `-v` option tells PIP to copy, in a file, all the input data and all the intermediary results. The name of this file is given either by the variable `DEBUG` in the environment or is built by `mktemp`.

If the `-z` option is given, then the solution is somewhat simplified (see below).

`input` and `output` are the names of the input (data) and output (results) files, respectively. If no `output` (`input`) file is given, then the results are printed to the standard output (input).

2.2.1 Messages

- `Version X.x`. Currently, `E.2`.
- `cross : <n>, alloc : <m>` This message is output after solving each problem. The value of `<n>` gives an idea of the complexity of the problem.

Errors related to the input

- `Syntax error`: unbalanced parentheses in the input.
- `Your computer doesn't have enough memory`: self explanatory.

Errors related to the solution

- `Integer Overflow`: A number has been generated that is too large to be accommodated in a 32 bit integer. Check the input and/or switch to Zbigniew Chamski's infinite precision PIP.
- `The solution is too complex`: the solution quast has grown beyond the memory allocated to it. Check the input and/or change the value of constant `SOL_SIZE` in file `type.h`, then rebuild PIP.
- `Memory overflow`: self explanatory.
- `<file> unaccessible`: one of the input, output or debug file cannot be opened.

Dimension errors

- `Too much variables`
- `Too much parameters` : Check the input and/or change the value of constants `MAXCOL` and `MAXPARM` in file `type.h`, then rebuild PIP.

Implementation errors All such error messages begin by the word `Syserr`. These messages indicate a bug in the implementation. You should report such events by sending a copy of the input file by e-mail to the author, `Paul.Feautrier@prism.uvsq.fr` who will endeavor to solve the problem as soon as possible.

2.3 Output Data

The output file can be described by the following grammar:

Grammar 2 *:*

```
File ::= Result ...
Result :: ( Comments Solution )
Solution ::= Quast_group
    | void
Quast_group ::= Quast
    | Newparm ... Quast
Quast ::= Form
    | (if Vector Quast_group Quast_group)
Form ::= (list Vector ...)
    | nil
Newparm ::= (newparm Integer Vector Integer)
Vector ::= #[ Coefficient ... ]
Coefficient ::= Integer | Integer / Integer
```

The `Comments` are copied from the input file. The `Solution` is said to be `void` when the initial context is void. Otherwise, it is given as a quast written *à la* Lisp. The quast may possibly be preceded by the definition of one or several new parameters.

The vector coefficients may be either integers or rationals written as `num/denom`. The latter case occurs if `Nq` had been set to 0 in the input file.

In the solution, a `Vector` represents an affine form; each entry is the coefficient of the corresponding parameter (the parameter of the same rank). The last entry is the additive constant.

The definition of a new parameter begins with the key-word `newparm`, then a rank number, a vector of coefficients, and a denominator. The new parameter is equal to the integer division of the vector by the denominator. The new

parameter can only appear in the `Quast` following its definition. Introducing a new parameter adds one entry in the list of parameters, so the length of vectors in the solution is not constant. However, this length is always equal to 1 plus the number of original parameters plus the number of new parameters currently defined.

The solution is a multi-level conditional expression (a tree of nested conditionals.) A predicate expression p should be understood as the boolean expression $p \geq 0$. Leaves of the conditional tree are either `nil`, meaning that the input problem has no solution, or a `Form`. A `Form` is a list of vectors, each vector giving the value of the corresponding unknown.

2.3.1 Example

The output of PIP is not intended for human consumption. No attempt has been made to implement a pretty-printer. In the interest of readability, some of the result files in this paper have been beautified by hand. The reader should not be surprised if he gets results with different layouts when running the examples.

Here is the output solution file for the example above (2.1.1):

```
( (Lower bound on j after loop inversion
      (unknowns j i)
      (parameters k m n) 1 )(if #[ -1 1 0 0]
(list #[ 0 0 0 0]
#[ 1 0 0 0]
)
(list #[ 1 -1 0 0]
#[ 0 1 0 0]
)
)
)
```

To express this solution, no new parameter had to be introduced. The form associated to the first conditional is:

$$-1 \times k + 1 \times m + 0 \times n + 0 \times 1 = m - k$$

so the test should be read as $m - k \geq 0$. If this inequality holds, then the solution is $< 0, k >$. Otherwise, the solution is $< m - k, m >$.

To sum things up, the lexicographical minimum of $\mathcal{D}_2$ is:

```
if m-k >= 0 then <0, k> else <m-k, m>.
```

Hence the lower bound on the first coordinate:

```
if m-k >= 0 then 0 else m-k
```

2.3.2 Simplifying the solution

The solution of a parametric problem may be in the form of a quast all of whose leaves are nil. This means in fact that the original polyhedron is empty whatever the values of the parameters. An example, due to Dirk Fimmel, is the following:

```
(((i j 1)(m n))
 2 2 7 0 -1 1
 (#[2 6 -9 0 0]
  #[5 -3 0 0 0]
  #[2 -10 15 0 0]
  #[-2 6 -3 0 0]
  #[-2 -6 17 0 0]
  #[0 1 0 -1 0]
  #[1 0 0 0 -1]
 )
 ()
)
```

Without the `-z` option, the solution is:

```
(((i j 1)(m n) -1 )
 (if #[ -4 0 5]
  (if #[ 0 -4 3]
   ()
   (if #[ 0 -2 9]
    (if #[ 0 -2 3]
     (newparm 2 (div #[ 0 2 3] 6))
     (newparm 3 (div #[ 0 2 10 7] 12))
     (newparm 4 (div #[ 0 4 0 2 1] 6))
     ()
     (if #[ 0 -2 7]
      (newparm 2 (div #[ 0 4 3] 6))
      (if #[ 0 -8 6 11] () ())
      ()))
    ()))
  (if #[ -1 0 3]
   (if #[ -1 0 2]
    (if #[ 10 -2 -15] ()())
    ())
  ()))
)
```

Inspection reveals that all leaves are `()`. With the `-z` option, the solution is much simpler:

```
(((i j 1)(m n) -1 )()
)
```

2.4 The Power of PIP

In the following sections, we explain how PIP can be used to solve extended classes of problems:

- Problems where equalities occur.
- Problems where a lexicographical *maximum* has to be found.
- Cases when linear cost functions are to be optimized.
- Problems where unknowns and/or parameters may be negative

2.4.1 Handling Equalities

When the input problem contains r affine equalities $f_i = 0$,$1 \leq i \leq r$, one may just write r inequalities $f_i \geq 0$ and r inequalties $f_i \leq 0$, thus satisfying PIP's input syntax. However, one may notice that only $r + 1$ inequalities are needed: $f_i \geq 0$, $1 \leq i \leq r$, and the following inequality:

$$\sum_{i=1}^{r} f_i \leq 0.$$

2.4.2 The bigparm trick

In some cases, it is useful to suppose that one parameter in a PIP problem grows "very large". Some examples will be given in the following sections. Let B be the name of this parameter. Suppose that in the solution, one of the predicates is:

$$aB + b \geq 0,$$

where b may depend on all other parameters. For B large enough, if $a > 0$ then the predicate is true, and if $a < 0$ then the predicate is false. One can find the limit shape of the solution by removing such tests and replacing them by their true of false branch, as appropriate. This can be done *a posteriori* on the results of PIP, or PIP can do it "on the fly" while solving the problem. This last method is more efficient, since it tends to simplify the solution.

PIP is notified of the presence of a big parameter by setting the `Bg` argument to a positive value. This value is the rank of the big parameter in the problem tableau. Hence, the lowest admissible value for `Bg` is `Nn + 1`.

The reader should convince himself that in the presence of two big parameters, no such simplifications are possible unless one has some information on the relative size of the parameters. Such situations should be handled by giving PIP

ordinary parameters, and doing the simplification on the solution in the light of extra knowledge.

2.4.3 Computing Lexicographical Maxima

To get the maximum of an unknown x, minimize $B - x$, where B is a new "big" parameter. Adding a parameter just adds one column in the problem tableau. The fact that this column corresponds to a Big parameter is specified by setting the 5-th switch to a positive value, this value being the position of the column of B in the problem tableau.

These cases can be handled systematically in the following way. Suppose that we are asked for the integer maximum of the polyhedron:

$$\begin{aligned} x &\geq 0, \\ y &\geq 0, \\ 3y &\leq x + 12, \\ y &\geq 2x - 3. \end{aligned} \tag{6}$$

Let us introduce the new unknowns:

$$x' = B - x, \;\; y' = B - y,$$

where B is the big parameter. System (6) translates to:

$$\begin{aligned} -x' + B &\geq 0, \\ -y' + B &\geq 0, \\ -x' + 3y' + 12 - 2B &\geq 0, \\ 2x' - y' + 3 - B &\geq 0. \end{aligned}$$

Finding the maximum of $(x, y)^T$ is equivalent to finding the minimum of $(x', y')^T$, provided B is large enough. The solution of the above problem is:

```
((a maximization problem 1 )
 (if #[ -1 6]
  (if #[ -1 3]
   (list #[ 0 0]
         #[ 0 0])
   (if #[ -5 27]
    (newparm 1 (div #[ 1 1] 2))
    (list #[ 1 -1 -1]
          #[ 0 0 0])
    (list #[ 1 -4]
          #[ 1 -5])))
  (list #[ 1 -4]
        #[ 1 -5])))
```

Suppose we tell PIP that B is a large parameter. The input file is now:

```
((a maximization problem)
 2 1 4 0 3 1
 (#[-1 0 0 1]
  #[0 -1 0 1]
  #[-1 3 12 -2]
  #[2 -1 3 -1]
 )
 ()
)
```

and the solution is much simpler:

```
((a maximization problem 1 )
 (list #[ 1 -4]
       #[ 1 -5]))
```

The reader may care to check that this result is equivalent to the previous one as soon as $B > 5$. The position of the minimum is: $x' = B - 4, y' = B - 5$, from which we deduce: $x = 4, y = 5$. As expected, B has disappeared from the solution. If this does not happen, we observe first that B must have a positive coefficient in the result (if not, one of the inequalities $x, y \geq 0$ would be violated for B large enough). This means that the original polyhedron is not bounded, since, whatever B, it contains a point whose coordinates are $O(B)$, and hence has no maximum.

2.4.4 Optimizing Linear Cost Functions

The problem here is to compute the minimum of a linear function cx in a polyhedron P, where c is a vector with integer coefficients. Let us introduce a new unknown y. Solve the linear programming problem obtained by adding the constraint $y \geq cx$ to the defining constraints of P. y should be the first unknown in the lexicographic ordering. Let y_s, x_s be the solution. Suppose that the minimum of cx in P is obtained at x_m and set $y_m = cx_m$. Since x_s is in P, and $y_s \geq cx_s$, it is clear that $y_s \geq y_m$. Conversely, (y_m, x_m) satisfies the constraints of the problem of which (y_s, x_m) is the lexicographic minimum. Hence $(y_s, x_s) \ll (y_m, x_m)$, and, since y is the first unknown, $y_s \leq y_m$. Hence, $y_m = y_s$. There is no guarantee, however, that $x_s = x_m$.

2.4.5 Negative Unknowns and Parameters

Suppose we want to find the minimum of $f(i, j) = i - 2j$ over the square domain represented in Figure 1[1].

[1] This example was proposed and solved by Pierre Boulet.

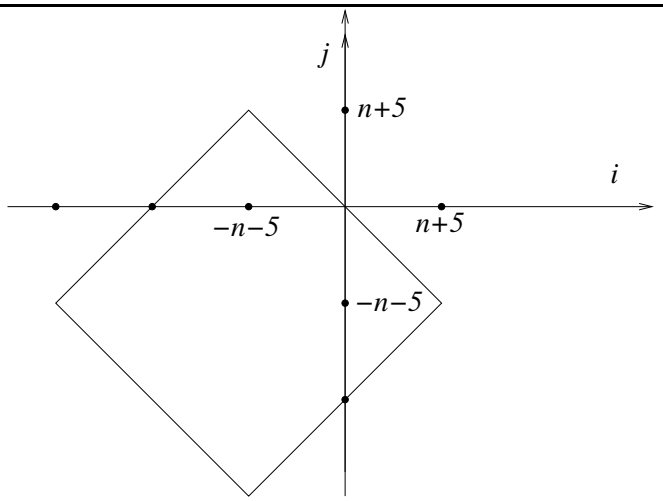

Figure 1: Problem domain

As above, we introduce a new unknown f and the inequality $f - i + 2j \geq 0$. Since we want to optimize f, f will appear as the first unknown.

The trick for solving the problem in Z is to introduce the following parameters:

- $G \geq max(0, -i, -j, -f)$.
- $P = max(0, -n)$.

This choice insure that the new variables and parameters:

$$\begin{aligned} f' &= G + f \\ i' &= G + i \\ j' &= G + j \\ n' &= P + n \end{aligned}$$

are all positive. This property stays true if G grows. Hence, G is again a big parameter. However, P must be considered as an ordinary parameter. After replacement of i, j, n, f by the new variables i', j', n', f', we get a system which corresponds to the following input:

```
(
 ( Solving MIN(i-2.j) under the following constraints:
   Unknowns may be negative.
   Order:
   f' i' j' constant G P n'
  )
 3 3 5 0 4 1
 (
```

```
  #[ 0 1 1 20 -2 -4 4 ]
  #[ 1 -1 2 0 -2 0 0 ]
  #[ 0 -1 -1 0 2 0 0 ]
  #[ 0 1 -1 10 0 -2 2 ]
  #[ 0 -1 1 10 0 -2 2 ]
   )
 ( ))
```

The result is:

```
(
 ( Solving MIN(i-2.j) under the following constraints:
   Unknowns may be negative.
   Order:
   f' i' j' constant G P n'
   -1 )(if #[ 0 -1 1 5]
(list #[ 1 3 -3 -15]
#[ 1 1 -1 -5]
#[ 1 -1 1 5]
)
()
)
)
```

which should be read as:

$$\begin{aligned}(f', i', j') \;=\; & \texttt{if } -P + n' - 5 \geq 0 \\ & \texttt{then } (G + 3P - 3n' - 15, G + P - n' - 5, G - P + n' + 5) \\ & \texttt{else } \bot\end{aligned}$$

That is, in the original coordinate system:

$$(f, i, j) = \texttt{if } n \geq 5 \texttt{ then } (-3n - 15, -n - 5, n + 5) \texttt{ else } \bot$$

I.e., the minimum value for function f is $-3n - 15$, and this value is reached at point $(-n - 5, n + 5)$. This minimum exists only if $n \geq 5$; otherwise, the feasible set is empty.

2.4.6 Mixed Programming

A mixed program is a program in which some variables are constrained to be integers while others may take rational values. Suppose for instance that we have to solve:

$$\begin{aligned}S \;=\; & \min ax + by, \\ & Ax + By + c \geq 0,\end{aligned}$$

where y is the vector of the integer variables. First, solve

$$\begin{aligned} T &= \min ax, \\ & \quad Ax + By + c \geq 0, \end{aligned}$$

in rational, with y as parameters. The result is a quast. To each leaf i is associated a linear function $f_i(y)$ and a set of inequalities $C_i y + d_i \geq 0$. T is equal to f_i when y is such that the corresponding inequalities are satisfied. For each i, solve the problem:

$$\begin{aligned} S_i &= \min f_i(y) + by, \\ & \quad C_i y + d_i \geq 0, \end{aligned}$$

in integers. The final result is the minimum of all S_i. Obviously, the method can accomodate parameters in the constraints. The S_i will be functions of these parameters, and the minimum must be computed symbolically.

3 Using the PIP Library

The PIP Library (PipLib for short) was implemented to allow the user to call PIP directly from his programs, without file accesses or system calls. The user only needs to link his programs with C libraries. The PipLib mainly provides one function which takes as input the problem description and some options, and returns a `Quast` (see grammar 2 in section 2.3) corresponding to the solution. Some other functions are provided for convenience reasons ; they are described in section 3.2. Most of them require some specific structures to represent the problem or the solution ; these structures are described in section 3.1.

3.1 PipLib data structures description

3.1.1 PipMatrix structure

```
struct pipmatrix
{ unsigned NbRows, NbColumns ;
  Entier ** p ;
  Entier * p_Init ;
} ;
typedef struct pipmatrix PipMatrix ;
```

The `PipMatrix` structure is devoted to represent a constraints matrix in the PolyLib shape [?]. The whole matrix is arranged row after row at the `p_Init` adress. `p` is an array of pointers in which `p[i]` points to the beginning of the i^{th} row. `NbRows` and `NbColumns` are respectively the number of rows and columns of

the matrix. We use this structure to carry polyhedrons. Each row corresponds to a constraint which the polyhedron must satisfy. The constraint is an equality if the first element is 0, an inequality $p(x) \geq 0$ if the first element is 1. The next elements are the unknown coefficients, followed by the parameter coefficients. The last element is the constant factor. For instance, in the problem of section 2.1.1 the domain is defined by 3 constraints:

$$\begin{cases} -i + m \geq 0 \\ -j + n \geq 0 \\ j + i - k \geq 0 \end{cases}$$

the rows corresponding to these constraints would be:

```
# eq/in  i   j   k   m   n   cst
    1    0  -1   0   1   0    0
    1   -1   0   0   0   1    0
    1    1   1  -1   0   0    0
```

The context is defined by one constraint:

$$\{ \ -k + m + n \geq 0$$

the row corresponding to this constraint would be:

```
# eq/in  k   m   n   cst
    1   -1   1   1    0
```

3.1.2 PipVector structure

```
struct pipvector
{ int nb_elements ;
  Entier * the_vector ;
  Entier * the_deno ;
} ;
typedef struct pipvector PipVector ;
```

The `PipVector` structure represents a `Vector` as described in grammar 2 in section 2.3. `nb_elements` is the number of vector elements, `the_vector` is an array which contains the numerators of these elements and `the_deno` is an array which contains their denominators: the i^{th} element is `the_vector[i]/the_deno[i]`.

3.1.3 PipNewparm structure

```
struct pipnewparm
{ int rank ;
  PipVector * vector ;
```

```
  Entier deno ;
  struct pipnewparm * next ;
} ;
typedef struct pipnewparm PipNewparm ;
```

The `PipNewparm` structure represents a `NULL` terminated linked list of `Newparm` as described in grammar 2 in section 2.3. For each `Newparm`, the rank is `rank`, the vector of coefficients is pointed by `vector`, and the denominator is `deno`. `next` is a pointer to the next `PipNewparm` structure.

3.1.4 PipList structure

```
struct piplist
{ PipVector * vector ;
  struct piplist * next ;
} ;
typedef struct piplist PipList ;
```

The `PipList` structure represents a `NULL` terminated linked list of `Vector` as described in grammar 2 in section 2.3. `vector` is a pointer to the vector of the current node and `next` is a pointer to the next `PipList` structure.

3.1.5 PipQuast structure

```
struct pipquast
{ PipNewparm * newparm ;
  PipList * list ;
  PipVector * condition ;
  struct pipquast * next_then ;
  struct pipquast * next_else ;
  struct pipquast * father ;
} ;
typedef struct pipquast PipQuast ;
```

The `PipQuast` represents a `Quast` as described in grammar 2 in section 2.3. Each `Quast` has a tree structure and begins with a list of `Newparm` (field `newparm`). If the pointer `condition` is not `NULL`, the list of `Newparm` is followed by a conditional structure : if the condition pointed by `condition` is true, then the solution continues in the `Quast` pointed by `next_then`, in the `Quast` pointed by `next_else` otherwise. If the pointer `condition` is `NULL`, the list of `Newparm` is followed by a list of vectors (field `list`). For `Quast` manipulation convenience, a pointer to the father in the tree is provided (field `father`), obviously the father of the root is `NULL`.

3.2 PipLib functions description

3.2.1 pip_solve function

```
PipQuast * pip_solve
( PipMatrix * domain,
  PipMatrix * context,
  int Bg,
  int Nq,
  int Verbose,
  int Simplify,
  int Max
) ;
```

The `pip_solve` function solves a linear problem provided as input. The first three parameters describe the problem that the user wants to solve. The last four parameters describe the options that the user has to set. These parameters are:

1. `domain`: a pointer to the equations and inequations system which describes the unknown domain in the PolyLib constraints matrix shape,
2. `context`: a pointer to the equations and inequations system satisfied by the parameters context in the PolyLib constraints matrix shape (it can be NULL if there is no context),
3. `Bg`: the column rank of the bignum (first column rank is 0), or a negative value if there is no big parameter in the problem to be solved,
4. `Nq`: a boolean set to 1 if an integer solution is needed, 0 otherwise,
5. `Verbose`: a boolean set to 1 if debug informations are needed (in this case, most of the processing will be printed in a file. The file name is generated at random (by *mkstemp*) or set with constant DEBUG), 0 if not,
6. `Simplify`: a boolean set to 1 if some trivial quast simplifications are needed (recursive elimination of degenerated patterns like `if #[...] () ()`), 0 otherwise,
7. `Max`: not in use yet, please set it to 0 for future compatibility.

This function returns a pointer to a `PipQuast` structure containing the solution, it will be `NULL` if the context is `void`.

3.2.2 pip_alloc_matrix function

```
PipMatrix * pip_alloc_matrix
( unsigned nb_rows,
  unsigned nb_columns
) ;
```

The `pip_alloc_matrix` function allocates the memory space for a `PipMatrix` structure with `nb_rows` rows and `nb_columns` columns. It fills the `Nb_Rows`, `Nb_Columns` and `p` fields and initializes the matrix entries to 0, then it returns a pointer to this structure.

3.2.3 pip_read_matrix function

```
PipMatrix * pip_read_matrix(FILE *) ;
```

The `pip_read_matrix` function read a matrix from a file. It takes as input a pointer to the file it has to read (possibly `stdin`), and returns a pointer to a `PipMatrix` structure. The input has the following syntax:

- some optional comment lines which begin with `#`,
- the row numbers and column numbers, possibly followed by comments, on a single line,
- the matrix rows, each row must be on a single line and is possibly followed by comments.

For instance, in the problem of section 2.1.1 the domain is defined as follows

```
# This is the domain
3 7                 # 3 lines and 7 columns
1  0 -1  0  1  0  0 # -i + m >= 0
1 -1  0  0  0  1  0 # -j + n >= 0
1  1  1 -1  0  0  0 # j + i - k >= 0
```

3.2.4 Printing Functions

```
void pip_print_matrix(FILE *, PipMatrix *) ;
void pip_print_vector(FILE *, PipVector *) ;
void pip_print_newparm(FILE *, PipNewparm *, int indent) ;
void pip_print_list(FILE *, PipList *, int indent) ;
void pip_print_quast(FILE *, PipQuast *, int indent) ;
```

There is a printing function for each structure of the PipLib. They all take as input a pointer to a file (possibly `stdout`) and a pointer to a structure. Some of them takes as input an indent step. They print the structure contents to the file without indent if `indent` < 0, with an indentation step of `indent` otherwise.

3.2.5 Memory Deallocation Functions

```
void pip_free_matrix(PipMatrix *) ;
void pip_free_vector(PipVector *) ;
void pip_free_newparm(PipNewparm *) ;
void pip_free_list(PipList *) ;
void pip_free_quast(PipQuast *) ;
```

There is a memory deallocation function for each structure of the PipLib. They free the allocated memory for the structure.

3.3 Example

Here is a simple example showing how one can use the PipLib, assuming that a basic installation was done. The following C program reads a domain and its context on the standard input then prints the solution on the standard output. Options are preselected : there is no bignum, we are looking for an integral solution without simplification and we don't want debug informations.

```
/* example.c */
# include <stdio.h>
# include <piplib.h>

int main()
{ PipMatrix * domain, * context  ;
  PipQuast * solution ;

  domain = pip_read_matrix(stdin) ;
  context = pip_read_matrix(stdin) ;

  solution = pip_solve(domain,context,-1,1,0,0,0) ;

  pip_free_matrix(domain) ;
  pip_free_matrix(context) ;

  pip_print_quast(stdout,solution,0) ;
}
```

The compilation command could be :

```
gcc example.c -lpiplib -o example
```

Supposing that the user wants to solve the problem of section 2.1.1, he will type:

```
3 7
1  0 -1  0  1  0  0
1 -1  0  0  0  1  0
1  1  1 -1  0  0  0
1 5
1 -1  1  1  0
```

And the program will print :

```
(if #[ -1 1 0 0]
 (list
  #[ 0 0 0 0]
  #[ 1 0 0 0]
 )
 (list
  #[ 1 -1 0 0]
  #[ 0 1 0 0]
 )
)
```

4 Installing PIP

4.1 Implementation Notes

The main problem with any integer programming software is that, since one must distinguish between integer and rationals, all computations are to be done exactly. Rationals must be represented as quotients with an integer numerator and an integer denominator. In the preceding version, there was only one denominator for the whole tableau. The consequence was that simplifications where most unlikely, and that the integers in the tableau were growing until overflows occured.

In the present version, there is one denominator per row of the tableau. Reduction to lower terms occurs frequently, and the growth of numbers in the problem tableau is limited. As a consequence, much larger problems can be solved. This has had the unfortunate consequence that several bugs, which were beyond the domain of the old version, have now surfaced. These bugs have been corrected. As far as the author can tell, these bugs mainly gave correct results which were not in simplest form: the quast had extraneous leaves. In some cases, the result was wrong but the error was self evident: for instance, there were denominators in integer results.

4.2 Description

The PIP software, while not public domain, is freely available for teaching and research, under the condition that the author is cited. Any commercial use is forbid-

den. A copy of PIP can been obtained by anonymous ftp to `ftp.prism.uvsq.fr`. Alternatively, PIP is referenced through the author home page, `http://www.prism.uvsq.fr/~paf/index.html`. PIP is distributed as a single compressed tarfile, `pip.tar.Z`.

4.3 Adjusting the Precision

Pip is an all integer version of the dual simplex algorithm. As such, it has to handle integers whose size may grow exponentially as the computation proceeds. Integer overflow may occur and have to be checked. Since the hardware integer overflow exception is usually masked by the operating system or the compiler, overflow is detected by checking that a division somewhere in the algorithm, which can be proved to be exact by mathematical arguments, is indeed exact. If not, an error is reported and the computation stops.

The size of the numbers to be handled depends strongly on the size of the constraint matrix and on the size of its coefficients. The precision of the integer representation in the Pip code can be adjusted by modifying two lines in the file `type.h`. In the distribution, these lines are

```
#define Entier long long
#define FORMAT "%lld"
```

This results in a 64 bits Pip. For small systems, one may use:

```
#define Entier long
#define FORMAT "%ld"
```

resulting in a faster running time. The use of:

```
#define Entier short
#define FORMAT "%d"
```

is deprecated.

For very large systems, an arbitrary precision version is needed. See Sect. **??**.

4.4 Building the Executable and the Library

To build PIP, first copy the above tarfile to any convenient directory. Expand the tarfile using:

```
zcat pip.tar.Z | tar xvf -
```

You should obtain the following files:

- header files in the `include` directory,
- C code files in the `source` directory,

- a lot of data files `*.dat` and of result files `*.ll` in the `test` directory (you should then run at least some of the `*.dat` files and compare the results to the corresponding `*.ll` file),
- a simple example showing how to use the PipLib in the `example` directory,
- a postscript version of the present document, `pip.ps` in the `doc` directory,
- files needed for compilation and installation in the PIP root directory.

4.4.1 Basic Installation

The `configure` shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a `Makefile`. The file `configure.in` is used to create `configure` by a program called `autoconf`. You only need `configure.in` if you want to change it or regenerate `configure` using a newer version of `autoconf`.

The simplest way to compile this package is:

- `cd` to the directory containing the package's source code and type `./configure` to configure the package for your system (while running, `configure` prints some messages telling which features it is checking for),
- type `make` to compile the package and install the program and the library,
- you can remove the program binaries and object files from the source code directory by typing `make clean`. To also remove the files that `configure` created (so you can compile the package for a different kind of computer) type `make distclean`.

PIP and the PipLib have been successfully compiled on the following systems:

- PC's under Linux, with the `gcc` compiler,
- SparcStations, with the `gcc` compiler.

4.4.2 Optional Features

- By default, `make` will install the package's files in `/usr/local/bin`, `/usr/local/lib`, etc. You can specify an installation prefix other than `/usr/local` by giving `configure` the option `--prefix=PATH`.
- By default, both PIP and the PipLib are compiled and installed. By giving `configure` the option `--without-pip` you disable the compilation and installation of PIP. By giving `configure` the option `--without-lib` you disable the compilation and installation of the PipLib.

4.4.3 Uninstallation

You can easily remove PIP and the PipLib from your system by typing `make uninstall`.

5 Multiple Precision Pip

Multiple Precision Pip is built on top of the GMP library. Each integer in the program is represented as a list of 32 bits numbers. All computations are done exactly, and the size of the numbers increases as needed to preserve exactness. It follows that no overflow is possible. However, when the size of numbers increases, computations get slower and slower, and memory overflow may occur in extreme cases. In well behaved problems, 32 bits are enough for the initial data, the size of intermediate results first increases up to a maximum, then decreses, and 32 bits are again enough for the results. Hence, it has been possible to keep the input format and output format of Multiple Precision Pip completely compatible with the formats of the bounded precision versions.

To install Multiple Precision Pip, first install Gmp according to the directions found at the above URL. Usually, the library is installed in `/usr/local/lib`, and the header files are in `/usr/local/include`. If this is not the case, you must adjust the `Makefile` in `Pip/mpPip`.

When this is done, go to the directory `Pip/mpPip`. Say:

```
make clean
make pipMP
```

If everything runs properly, there should be an executable file, `pipMP` in the directory. Test it by saying:

```
pipMP rairoi.dat
```

and comparing the results with those of a bounded precision Pip.

Report all bugs, problems, inaccuracies in the documentation to: `Paul.Feautrier@ens-lyon.fr`. Praise is also appreciated.

Let the power of parametric integer programming be with you.