User's Guide for 4ti2 version 1.6

A software package
for algebraic, geometric and combinatorial problems on linear spaces

September 2, 2013

Contents

Chapter 1

Beginner's guide

In this part, we use a few sample problems to introduce you to the basic functionality of `4ti2`. After working through this part, you should know about *linear systems* and their encodings in `4ti2`, and should be able to do computations using the following functions:

- `qsolve`, `rays`, `circuits`
- `zsolve`, `hilbert`, `graver`, `ppi`
- `minimize`, `groebner`, `normalform`
- `genmodel`, `markov`

1.1 Linear systems and their encodings

In this section you learn about the data structure `linear system` and how it is specified in `4ti2`.

1.1.1 Linear systems and integer linear systems

In `4ti2`, a *linear system* is defined by d constraints $Ax \sim b$ in n unknowns x, where each constraint is either $\leq$, $=$ or $\geq$, that is $\sim \in \{\leq, =, \geq\}^d$. Moreover, one may specify sign constraints on the variables that need to be respected in an explicit continuous/integer representation of all solutions.

There is no particular difference in `4ti2` between a linear system and an integer linear system. Currently, the user chooses between one of the two by calling the appropriate functions on the linear system.

1.1.2 Specifying a linear system in `4ti2`

In order to use a linear system as input, we need to specify its parts to `4ti2`. As our running example, take

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \end{pmatrix} x \begin{matrix} \leq \\ \leq \end{matrix} \begin{pmatrix} 6 \\ 10 \end{pmatrix}$$

with sign constraints $(1, 2, 2, 0)$, which we will explain below.

First, we have to give our problem a project name, say `PROJECT`.

- The matrix A has to be put into the file `PROJECT.mat`.

```
2 4
1 1 1 1
1 2 3 4
```

- The relations $\sim$ then have to be specified in `PROJECT.rel`.

```
1 2
< <
```

- The right-hand side vector goes into `PROJECT.rhs`.

```
1 2
6 10
```

- And finally, the sign constraints end up in `PROJECT.sign`.

```
1 4
1 2 2 0
```

Note.

- The input files all have the format of a matrix, preceded by the matrix dimensions. As the dimensions already specify how many symbols have to be read, the matrix could also be given in only one line or even in many lines of different lengths.
- In `4ti2` version 1.3.1, all appearing numbers have to be integers.
- Consequently, this implies that, at the moment, `qsolve` only supports homogeneous linear systems, that is systems with $b = 0$, since minimal inhomogeneous solutions could have rational components.

1.1.3 How does an explicit solution to linear systems look like?

If the system is solved over $\mathbb{R}$ (using `qsolve`), `4ti2` returns two sets of integer vectors:

- a set H of support-minimal homogeneous solutions, and
- a set F defining the linear vector space the solution set lives in.

As only *homogeneous* linear systems are supported in this version of `4ti2`, no list of minimal inhomogeneous solutions is computed. Any solution z of the linear system can now be written as

$$z = \sum \alpha_j h_j + \sum \beta_k f_k \tag{1.1}$$

with $h_j \in H$, $f_k \in F$, and $\alpha_j \geq 0$.

If the system is solved over $\mathbb{Z}$ (using `zsolve`), `4ti2` returns three sets of integer vectors:

- a set H of minimal homogeneous *integer* solutions,
- a set I of minimal inhomogeneous *integer* solutions, and
- a set F defining the sublattice of $\mathbb{Z}^n$ the solution set lives in.

Any solution z of the linear system can now be written as

$$z = i + \sum \alpha_j h_j + \sum \beta_k f_k \tag{1.2}$$

for some $i \in I$ and with $h_j \in H$, $f_j \in F$, and $\alpha_j \in \mathbb{Z}_+$.

Sign file. Let us finally clarify what the sign file `PROJECT.sign` is good for. The sign file may declare a variable to be non-negative (1), to be non-positive (-1), or to consider both cases independently and unite the answers (2). If a nonzero sign has been assigned to a variable, the explicit representations (1.1) and (1.2) above of a solution z have to respect the sign on that variable. The default setting for each variable is 0 (when using `qsolve` and `zsolve`), that is, the sign need not be respected in the explicit representation. In our example above, the first variable is declared to be non-negative, the second and the third one expand to $2 \cdot 2 = 4$ orthant constraints, and the fourth variable is unconstrained. Note, however, that `4ti2` does *not* decompose the problem internally into the four problems with sign patterns $(1, 1, 1, 0)$, $(1, 1, -1, 0)$, $(1, -1, 1, 0)$, and $(1, -1, -1, 0)$, but deals with them more efficiently at the same time.

1.2 Brief tutorial

1.2.1 Solving linear systems over $\mathbb{R}$ and over $\mathbb{Z}$

In this example you learn about the functions `qsolve` and `zsolve`.

Let us have a look at the linear system

$$\begin{array}{rcrcl} x & - & y & \leq & 2 \\ -3x & + & y & \leq & 1 \\ x & + & y & \geq & 1 \\ & & y & \geq & 0 \end{array}$$

and let us solve it over $\mathbb{R}$, we have to create the files encoding the linear system. Let us call our project `system`. Then the input files look as follows:

system.mat	system.rel	system.rhs	system.sign
3 2	1 3	1 3	1 2
1 −1	< < >	2 1 1	0 1
−3 1			
1 1			

and then call

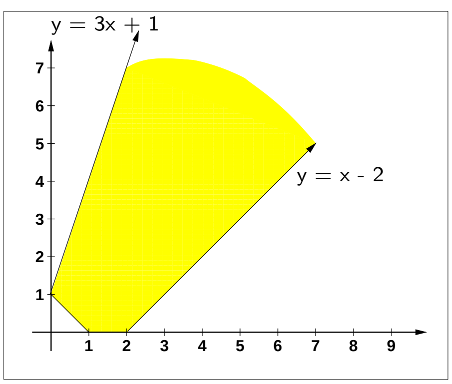
```
./qsolve system
```

This call creates three files

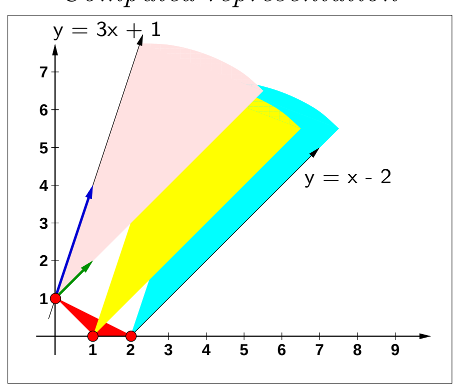
system.qinhom	system.qhom	system.qfree
3 2	2 2	0 2
0 1	1 1	
0 2	1 3	
1 0		

which correspond to the explicit description of all solutions:

Feasible solutions



Computed representation



$$\operatorname{conv}\left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}2\\0\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix}\right)+\operatorname{cone}\left(\begin{pmatrix}1\\1\end{pmatrix},\begin{pmatrix}1\\3\end{pmatrix}\right).$$

Note that in the second picture above, the three colored cones are only a simplification and shall visualize the covering of the feasible region by infinitely many shifted copies of the cone

$$\operatorname{cone}\left(\begin{pmatrix}1\\1\end{pmatrix},\begin{pmatrix}1\\3\end{pmatrix}\right),$$

one appended to each point in

$$\operatorname{conv}\left(\binom{1}{0}, \binom{2}{0}, \binom{0}{1}\right).$$

Let us now turn to the integer situation, that is, let us solve the system

$$\begin{array}{rcrcl} x & - & y & \leq & 2 \\ -3x & + & y & \leq & 1 \\ x & + & y & \geq & 1 \\ & & y & \geq & 0 \end{array}$$

over $\mathbb{Z}$. As the linear system itself is unchanged, we can use the same input files as above in order to specify the linear system.

system.mat	system.rel	system.rhs	system.sign
$\begin{array}{rr} 3 & 2 \\ 1 & -1 \\ -3 & 1 \\ 1 & 1 \end{array}$	$\begin{array}{ccc} 1 & 3 & \\ < & < & = \end{array}$	$\begin{array}{ccc} 1 & 3 & \\ 2 & 1 & 1 \end{array}$	$\begin{array}{cc} 1 & 2 \\ 0 & 1 \end{array}$

Then, however, we call

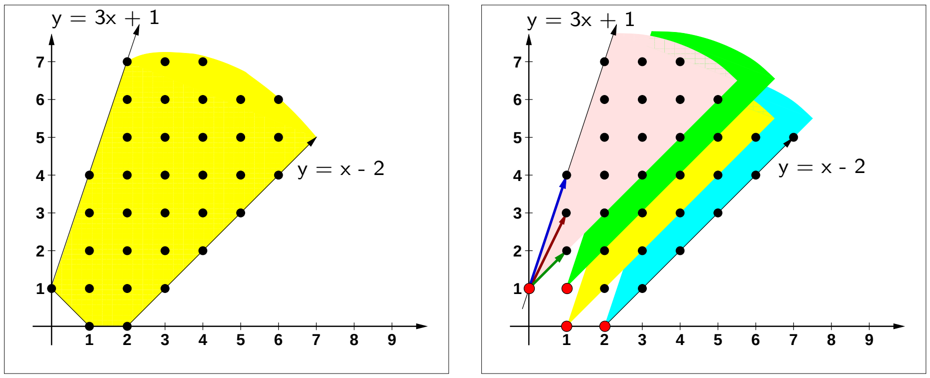
```
./zsolve system
```

This call creates three files

system.zinhom	system.zhom	system.zfree
$\begin{array}{cc} 4 & 2 \\ 0 & 1 \\ 0 & 2 \\ 1 & 0 \\ 1 & 1 \end{array}$	$\begin{array}{cc} 3 & 2 \\ 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{array}$	$\begin{array}{cc} 0 & 2 \end{array}$

which correspond to the explicit description of all integer solutions:

Feasible solutions *Computed representation*



$$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\} + \text{monoid} \left(\begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ 3 \end{pmatrix} \right).$$

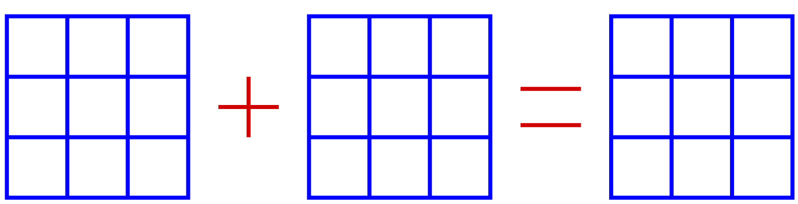
Note that in the pictures above, we are only interested in the *lattice points* inside the colored regions! The full regions are colored only for the purpose of visualizing the covering of all feasible integer solutions by finitely many shifted copies of the monoid

$$\text{monoid} \left(\begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ 3 \end{pmatrix} \right).$$

1.2.2 Computing extreme rays and Hilbert bases

In this example you learn about the functions `rays` and `hilbert`.

Let us consider the set of magic 3×3 squares with non-negative real entries, that is, the set of all 3×3 arrays with non-negative real entries whose row sums, column sums, and main diagonal sums all add up to the same number, the magic constant of the square.



Clearly, addition of two magic squares gives another magic square, as well as does multiplication of a magic square by a non-negative number. Therefore, we may talk about the *cone* of magic 3×3 squares. In fact, this cone is a pointed rational polyhedral cone described by the linear system

$$
\begin{aligned}
x_{11} + x_{12} + x_{13} &= x_{21} + x_{22} + x_{23} \\
&= x_{31} + x_{32} + x_{33} \\
&= x_{11} + x_{21} + x_{31} \\
&= x_{12} + x_{22} + x_{32} \\
&= x_{13} + x_{23} + x_{33} \\
&= x_{11} + x_{22} + x_{33} \\
&= x_{31} + x_{22} + x_{13} \\
& \quad x_{ij} \geq 0, \quad \text{for all } i,j = 1,2,3.
\end{aligned}
$$

Bringing all x_{ij} to the left-hand side of these equations, the matrix $A_{3\times 3}$ defining this linear system is

$$
A_{3\times 3} = \begin{pmatrix}
1 & 1 & 1 & -1 & -1 & -1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & -1 & -1 & -1 \\
0 & 1 & 1 & -1 & 0 & 0 & -1 & 0 & 0 \\
1 & 0 & 1 & 0 & -1 & 0 & 0 & -1 & 0 \\
1 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & -1 \\
0 & 1 & 1 & 0 & -1 & 0 & 0 & 0 & -1 \\
1 & 1 & 0 & 0 & -1 & 0 & -1 & 0 & 0
\end{pmatrix}.
$$

Below, we will deal with the more interesting case of *integer* magic squares. For the moment, however, we wish to compute the extreme rays of the magic square cone $\{z : A_{3\times 3} z = 0, z \geq 0\}$.

In order to call the function `rays`, we only have to create one file, say `magic3x3.mat`, in which we specify the problem matrix $A_{3\times 3}$. The remaining data is set by default to "equations only", to "homogeneous system", and to "all variables are non-negative". Note that we are allowed to change these defaults (except homogeneity) by specifying data in `magic3x3.rel` and `magic3x3.sign`

magic3x3.mat								
7	9							
1	1	1	−1	−1	−1	0	0	0
1	1	1	0	0	0	−1	−1	−1
0	1	1	−1	0	0	−1	0	0
1	0	1	0	−1	0	0	−1	0
1	1	0	0	0	−1	0	0	−1
0	1	1	0	−1	0	0	0	−1
1	1	0	0	−1	0	−1	0	0

Now we call

```
./rays magic3x3
```

which creates the single file

magic3x3.ray								
4	9							
0	2	1	2	1	0	1	0	2
1	2	0	0	1	2	2	0	1
2	0	1	0	1	2	1	2	0
1	0	2	2	1	0	0	2	1

that corresponds to the four extremal rays of the 3×3 magic square cone:

0	2	1
2	1	0
1	0	2

1	2	0
0	1	2
2	0	1

2	0	1
0	1	2
1	2	0

1	0	2
2	1	0
0	2	1

Every magic 3×3 square is a non-negative linear combination of these four magic squares.

If we turn now to *integer* magic squares, we are looking for a Hilbert basis of the 3×3 magic square cone. As the default settings for `hilbert` are the same as for `rays`, we can use the same input file magic3x3.mat

7	9							
1	1	1	−1	−1	−1	0	0	0
1	1	1	0	0	0	−1	−1	−1
0	1	1	−1	0	0	−1	0	0
1	0	1	0	−1	0	0	−1	0
1	1	0	0	0	−1	0	0	−1
0	1	1	0	−1	0	0	0	−1
1	1	0	0	−1	0	−1	0	0

for this computation. However, to compute the Hilbert basis, we call

```
./hilbert magic3x3
```

which creates the single output file

magic3x3.hil

4	9							
0	2	1	2	1	0	1	0	2
1	2	0	0	1	2	2	0	1
2	0	1	0	1	2	1	2	0
1	0	2	2	1	0	0	2	1
1	1	1	1	1	1	1	1	1

that corresponds to the five elements in the minimal Hilbert basis of the 3×3 magic square cone:

0	2	1
2	1	0
1	0	2

1	2	0
0	1	2
2	0	1

2	0	1
0	1	2
1	2	0

1	0	2
2	1	0
0	2	1

1	1	1
1	1	1
1	1	1

Every integer magic 3×3 square is a non-negative *integer* linear combination of these five integer magic squares. Note that the all-1 square is in the interior of the magic square cone.

1.2.3 Computing circuits and Graver bases

In this example you learn about the functions `graver`, `ppi`, and `circuits`.

As an example of a Graver basis computation, let us compute the primitive partition identities of order $n = 4$. Before we do the simple computation, let us explain what a primitive partition identity is.

A **partition identity** is any identity of the form

$$a_1 + \ldots + a_k = b_1 + \ldots + b_l$$

with (generally not distinct) integer numbers $0 < a_i, b_j \leq n$. A partition identity is called **primitive** if no proper subidentity exists.

For example,

$$1 + 2 + 3 = 2 + 2 + 2$$

is a partition identity which is not primitive, since it contains the subidentity

$$1 + 3 = 2 + 2$$

which is in fact primitive.

The description of the primitive partition identities for fixed n, however, is exactly the description of the Graver basis of the matrix

$$A_n = \begin{pmatrix} 1 & 2 & 3 & \ldots & n \end{pmatrix}.$$

Let us finally do the computation for $n = 3$. We create an input file `ppi3` for `4ti2` which looks as follows:

ppi3.mat
1 3
1 2 3

and call

```
./graver ppi3
```

This call will create an output file `ppi3.gra` that looks like:

ppi3.gra		
5	3	
3	0	-1
2	-1	0
0	3	-2
1	1	-1
1	-2	1

Thus, there are 5 primitive partition identities of order $n = 3$:

$$\begin{aligned} 1+1+1 &= 3 \\ 1+1 &= 2 \\ 2+2+2 &= 3+3 \\ 1+2 &= 3 \\ 1+3 &= 2+2 \end{aligned}$$

You may try and compute the primitive partition identities for bigger n, say $n = 17$, 20, or 23. Be aware, especially the latter two problems take a long, long time. What is the biggest n for which you can compute the primitive partition identities of order n on your machine within one hour?

Due to the very special structure of the matrix, there are algorithmic speed-ups [2, 3, 5]. The currently fastest algorithm to compute primitive partition identities is implemented in the function `ppi` of `4ti2`. Try running

```
./ppi 17
```

which creates two files `ppi17.mat` (so we do not really have to create this file ourselves) and the file `ppi17.gra` containing the desired identities. Compare this running time with the time taken by

```
./graver ppi17
```

Do you notice the speed-up?

Let us now turn to the question of determining the support-minimal partition identities. This, in fact, is the question of computing the circuits of the matrix

$$A_n = \begin{pmatrix} 1 & 2 & 3 & \ldots & n \end{pmatrix}.$$

We use the same input file

ppi3.mat		
1	3	
1	2	3

as above and call

```
./circuits ppi3
```

This call will create an output file `ppi3.cir` that looks like:

ppi3.cir		
3	3	
3	0	-1
2	-1	0
0	3	-2

Thus, there are 3 support-minimal partition identities of order $n = 3$:

$$\begin{aligned} 1+1+1 &= 3 \\ 1+1 &= 2 \\ 2+2+2 &= 3+3 \end{aligned}$$

Note that support-minimal partition identities are primitive, since the circuits of a matrix are contained in the Graver basis of this matrix.

1.2.4 Integer programming and toric Gröbner bases

In this example you learn about the functions `minimize`, `groebner`, and `normalform`.

The following neat example is based on the example presented in [4]. Let us assume that we want to give change worth 99 cents using only pennies (1ct), nickels (5ct), dimes (10ct), and quarters (25ct). Clearly,

$$4 \cdot 1 + 4 \cdot 5 + 0 \cdot 10 + 3 \cdot 25 = 99$$

would be one way to do it. Is this there another choice of 11 coins that sums up to 99ct but uses fewer nickels and quarters (in total)? In other words, we would like to solve

$$\min\{x_2 + x_4 : x_1 + x_2 + x_3 + x_4 = 11, x_1 + 5x_2 + 10x_3 + 25x_4 = 99, x_1, x_2, x_3, x_4 \in \mathbb{Z}_+\}$$

Let us set up the problem in `4ti2`.

4coins.mat	4coins.rhs	4coins.sign	4coins.cost
2 4 1 1 1 1 1 5 10 25	1 2 11 99	1 4 1 1 1 1	1 4 0 1 0 1

Note that we do not have to specify a relations file `4coins.rel`, since already by default all relations are assumed to be equations. Now we simply call

```
./minimize 4coins
```

which creates the single output file

4coins.min
1 4 4 1 4 2

From this, we conclude that

$$4 \cdot 1 + 1 \cdot 5 + 4 \cdot 10 + 2 \cdot 25 = 99$$

is an optimal choice, using only 3 instead of 7 nickels and quarters.

Remark. We could also specify a list of right-hand sides in `4coins.rhs`. The call

```
./minimize 4coins
```

then creates a file `4coins.min` containing minima to the corresponding integer programs. □

Since we already know a feasible solution, there is another way we might attack this problem, namely via toric Gröbner bases. For this, we first need to specify the matrix A and the cost vector c in the two files `4coins.mat` and `4coins.cost`:

4coins.mat	4coins.cost
2 4	1 4
1 1 1 1	0 1 0 1
1 5 10 25	

Then we compute the Gröbner basis of the toric ideal

$$I_A = \langle x^u - x^v : Au = Av, u, v \in \mathbb{Z}_+^4 \rangle$$

with respect to a term ordering $\prec$ compatible with c, that is, $c^\intercal v < c^\intercal u$ implies $x^v \prec x^u$. This toric Gröbner basis is computed by

```
./groebner 4coins
```

and gives the output file

4coins.gro
1 4
4 4 0 3

Then we specify our feasible solution in

4coins.zfeas
1 4
4 4 0 3

and call

```
./normalform 4coins
```

to produce the file

4coins.nf			
1	4		
4	1	4	2

that also contains the desired optimal solution.

Remark. We could also specify a list of feasible solutions in `4coins.zfeas`. Then the call

```
./normalform 4coins
```

creates a file `4coins.nf` containing the minima to the corresponding integer programs. (If z_0 is a feasible solution, the corresponding integer program is defined by putting the right-hand side to Az_0.) □

Rename `4coins.zfeas` to `4coins.zfea`?!

1.2.5 Markov Bases in Statistics

In this example you learn about the functions `markov` and `genmodel`.

Let us consider the following 4×4 table of non-negative integer numbers together with all row and column sums.

$$\begin{array}{cc} \begin{pmatrix} 11 & 23 & 34 & 3 \\ 4 & 15 & 12 & 11 \\ 17 & 2 & 3 & 25 \\ 16 & 12 & 22 & 7 \end{pmatrix} & \begin{array}{c} 71 \\ 42 \\ 47 \\ 57 \end{array} \\ \begin{array}{cccc} 48 & 52 & 71 & 46 \end{array} & \end{array}$$

In statistics, one wishes to sample among arrays that have fixed counts, say fixed row and column sums. In order to sample, one needs a set of moves that, in particular, do not change the counts when added to the current table. Clearly, these moves must have counts 0 and thus quite naturally lead us to the toric ideal

$$I_A = \langle x^u - x^v : Au = Av, u, v \in \mathbb{Z}_+^{16} \rangle,$$

where

$$A = \begin{pmatrix} 1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1 \\ 1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0 \\ 0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0 \\ 0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0 \\ 0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1 \end{pmatrix}.$$

It turns out that for any set of fixed counts, a (minimal) *Markov basis* is given by a minimal generating set of this toric ideal. Note that a Markov basis connects all non-negative tables with these counts in the sense that for any two non-negative tables T_1 and T_2 with these counts, there is a sequence of non-negative tables $T_1 = S_0, \ldots, S_N = T_2$ with the same counts as T_1 and T_2 and such that $S_i - S_{i-1}$ or $S_{i-1} - S_i$ is in the Markov basis for $i = 1, \ldots, N$.

For two-way tables the situation is still very simple as our computations with 4×4 tables will now demonstrate. Write the matrix that defines our toric ideal in the file `4x4.mat`:

4x4.mat
```
8 16
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0
0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0
0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0
0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1
```

Let us compute the Markov basis via the call

```
./markov 4x4
```

which creates a single output file `4x4.mar` containing the 36 Markov basis elements. Up to symmetry (swapping rows or columns), the Markov basis consists of the single move

$$\begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

In fact, this elementary move is (up to symmetry) the only representative of the minimal Markov moves for arbitrary $m \times n$ tables using row and column counts.

Creating the matrices for statistical models may be pretty cumbersome. `4ti2` provides a litte function, `genmodel`, that helps the user with creating matrices for hierarchical models defined by a complex.

The $m \times n$ tables problem above corresponds to the complex $\{\{1\},\{2\}\}$ on two nodes with levels m and n, respectively. Let us encode the complex for $3x6$ tables with 1-marginals (row and column sums) in `3x6.mod`.

3x6.mod	
3	
3	6
2	
1	1
1	2

and call

```
./genmodel 3x6
```

to produce the desired matrix file `3x6.mat`.

The encoding of the complex should be obvious from the example: first we state the number of nodes and their levels, then we give the number of maximal faces. Finally, we list each maximal face by first specifying the number of nodes on it and then by listing these nodes.

Thus, a $3x4x6$ table with 2-marginals (that is, again only counts along coordinate axes) corresponds to the complex $\{\{(1,2)\},\{(2,3)\},\{(3,1)\}\}$ on 3 nodes with levels 3, 4, and 6, respectively. Thus, its encoding is in `4ti2` would look like:

3x4x6.mod			
	3		
	3	4	6
	3		
	2	1	2
	2	2	3
	2	3	1

A binary model on the bipartite graph $K_{2,3}$ then reads as

3x4x6.mod					
	5				
	2	2	2	2	2
	6				
	2	1	3		
	2	1	4		
	2	1	5		
	2	2	3		
	2	2	4		
	2	2	5		

Chapter 2

Advanced guide

In this part, we deal with several more advanced problem specifications in `4ti2`.

First we introduce *affine systems* and their encodings. In fact, affine systems are the basic objects used in `4ti2`, since every linear system is transformed into an affine system. However, in the integer situation, it is not always possible to transform an affine system back into a linear system without adding variables or modulo constraints.

Next, we demonstrate how one can exploit bounds on integer variables to truncate the solution set.

Again, we use a few sample problems to demonstrate how to use `4ti2`. After working through this part, you should know about how to run the following functions on affine systems:

- `qsolve`, `rays`, `circuits`
- `zsolve`, `hilbert`, `graver`
- `markov`, `groebner`, `normalform`, `minimize`

2.1 Affine systems and their encodings

In `4ti2`, the definition of an *affine system* is slightly different for the continuous and for the integer situation. In fact, the definition for the integer case is simply the integer analogue to the definition for the continuous case.

2.1.1 Continuous affine systems

Let $a + \mathcal{L}_{\mathbb{R}}$ be a linear affine space given by the vector a and by generators for the linear space $\mathcal{L}_{\mathbb{R}}$. We wish to find a finite sign-compatible description for the set of all vectors $x \in a + \mathcal{L}_{\mathbb{R}}$.

2.1.2 Integer affine systems

Let $a + \mathcal{L}_{\mathbb{Z}}$ be an "integer linear affine space" given by the vector $a \in \mathbb{Z}^n$ and by generators for the lattice $\mathcal{L}_{\mathbb{Z}} \subseteq \mathbb{Z}^n$. We wish to find a finite sign-compatible description for the set of all (integer) vectors $x \in a + \mathcal{L}_{\mathbb{Z}}$.

2.1.3 Specifying an affine system in `4ti2`

Currently, only homogeneous affine systems can be solved in `4ti2` over $\mathbb{R}$ and over $\mathbb{Z}$ using the functions `qsolve` and `zsolve`, respectively. In order to call these functions, one needs to specify an affine system to `4ti2`.

As an example, let consider the linear space $\mathcal{L}_{\mathbb{R}}$ and the lattice $\mathcal{L}_{\mathbb{Z}}$ both spanned by the two vectors $(1, -2, 1, 0)$ and $(2, -3, -0, 1)$. Moreover, consider the sign-constraints $(1, 2, 2, 0)$. Thus, we are looking for a finite explicit sign-compatible description for all x that can be written as

$$x = \begin{pmatrix} 1 & 2 \\ -2 & -3 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \lambda,$$

with $\lambda \in \mathbb{R}^2$ and $\lambda \in \mathbb{Z}^2$, respectively.

In order to solve this affine system using `qsolve` or `zsolve`, we create the following input files to encode the affine system:

PROJECT.lat	PROJECT.sign
2 4 1 −1 1 0 2 −3 0 1	1 4 1 2 2 0

and then call

```
./qsolve PROJECT
```

and

```
./zsolve PROJECT
```

In the continuous case, this creates the files `PROJECT.qhom` and `PROJECT.qfree`, and in the integer case this creates the files `PROJECT.zhom` and `PROJECT.zfree`.

In contrast to calling `qsolve` and `zsolve` on a inear system, these calls create only two files, since at the moment only homogeneous affine systems (with $a = 0$) are supported by `4ti2`.

Chapter 3

README: Instructions on configuring and building 4ti2

```
4ti2 -- A software package for algebraic, geometric and combinatorial
problems on linear spaces.

Copyright (C) 2006 4ti2 team.

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

COMPILING 4ti2
==============
```

Run the following commands with the 4ti2 directory:

```
./configure --prefix=INSTALLATION-DIRECTORY
   make
   make check
make install-exec
```

The final command will install 4ti2 in a directory tree below the INSTALLATION-DIRECTORY that you gave with the first command. If you omit the --prefix option, 'make install' will install 4ti2 in the /usr/local hierarchy.

You will need glpk and gmp installed first (see below).

The first command, 'make', compiles all the executables. The second command, 'make check', runs a lot of automatic checks. This will take a while. If a check fails, then please notify the 4ti2 team.

You will need gcc version 3.4 or higher.

You will need an installed version of glpk (linear programming software). See the website http://www.gnu.org/software/glpk for more information. The version 4.7 has been tested. If you do not have glpk installed or 4ti2 cannot find glpk, then the compilation will fail saying that it cannot find the file "glpk.h". If you have installed glpk but not in a location that 4ti2 finds by default, then you will need to invoke

```
./configure --with-glpk=/ROOT/OF/GLPK/INSTALLATION/HIERARCHY
```

You will also need an installed version of gmp, The GNU MP Bignum Library, with c++ support enabled (see http://www.swox.com/gmp/ for more details). Versions 4.2.1 and 4.1.4 have been tested. If you are compiling a version of gmp from the source, make sure that you enable c++ support (--enable-cxx configure option). If you have installed gmp but not in a location that 4ti2 finds by default, then you

```
will need to invoke

     ./configure --with-gmp=/ROOT/OF/GMP/INSTALLATION/HIERARCHY

If you have gmp but not with c++ support, then ./configure will
fail with an error saying that the file "gmpxx.h" cannot be found.


USING MACPORTS ON MAC OS X
==========================

Use the following commands.

     sudo port install gmp glpk
     ./configure --with-gmp=/opt/local --with-glpk=/opt/local
     make
     sudo make install


DOCUMENTATION
=============

See the website http://www.4ti2.de for information on using 4ti2.
```

Chapter 4

NEWS: Changes to 4ti2 since version 1.2

```
News in 4ti2 version 1.6, compared to 1.5.2:

 * Restore the functionality of "hilbert" in versions up to 1.3.2 to
   accept "rel" files.  This signalled an error in the 1.4 and 1.5 series.
   (Note that "zsolve" did accept "rel" files in the 1.4 and 1.5 series.)

 * When the cone is not pointed, "hilbert" now outputs a "zfree" file,
   containing a lattice basis, in addition to the "hil" file.

   Note that in the non-pointed case, Hilbert bases are not uniquely
   determined.  Let zfree_1, ..., zfree_k be the vectors in the "zfree"
   file and hil_1, ..., hil_l be the vectors in the "hil" file.
   Then a Hilbert basis of the non-pointed cone is
   hil_1, ..., hil_l, -(hil_1 + ... + hil_l), zfree_1, ..., zfree_k.

   (In the 1.3 series, "hilbert" silently appended the lattice
   generators to the "hil" file.  Thus the list of vectors in the
   "hil" file was not a Hilbert basis of the non-pointed cone; this
   was a bug.  Note that "zsolve" did work correctly in the 1.3
   series.)
```

```
 * Fix a bug of zsolve and hilbert on 64-bit platforms (where
   sizeof(unsigned long) > sizeof(int)), which affected problems with
   more than 32 variables and could lead to wrong results. (Testcases
   a1, dutour-testcase-2013-08-21).

 * Accept longer filenames.

 * Enable shared library builds on the Cygwin platform (using the
   libtool -no-undefined flag).  (However, this requires that shared
   libraries of GMP, GLPK are available.)

 * Use gnulib to provide getopt_long if not available in the
   system libraries.

 * If the C++ compiler does not have int32_t and int64_t, use int and
   long int instead.

 * Fixed bug in lattice transformation with too few rows.
   (Reported by Jerry James for Fedora.)

 * Fix a build failure with gcc 4.7.
   (Patch by Jerry James for Fedora.)


News in 4ti2 version 1.5.2, compared to 1.5.1:

 * Build a GMP-only 4ti2 if the C++ compiler does not have int32_t and int64_t.


News in 4ti2 version 1.5.1, compared to 1.5:

 * Fix a build problem with --enable-shared.


News in 4ti2 version 1.5, compared to 1.4:
```

```
 * Latest version of new qsolve.


News in 4ti2 version 1.4, compared to 1.3.2:

 * Portability fixes

 * New abstract C and C++ API (callable library), header files in 4ti2/

 * New implementation of zsolve in C++


News in 4ti2 version 1.3.2, compared to 1.3.1:

 * New build system, using GNU Autoconf, Automake, and Libtool.

   This allows 4ti2 to be built using the standard "./configure &&
   make && make install" sequence.

 * Bug fixes

 * Portability fixes (for GCC versions 4.3.x and 4.4.x)


News in 4ti2 version 1.3.1, compared to 1.2:

 * 'groebner' and 'markov' are again heavily improved.

 * 'groebner' and 'markov' allow non-homogeneous lattice ideals.

 * 'groebner' and 'markov' allow truncation.

 * There is a new function 'walk' performing a Grbner walk.

 * There are new functions 'qsolve' and 'zsolve' for solving linear
   systems over the reals or the integers, respectively.
```

* There are new functions 'rays' and 'circuits' to compute extreme
  rays and circuits.

* The functions 'circuits' and 'graver' allow to fix certain
  orthants.

* One may compute with projections by specifying variables to be
  ignored.

* There is a new function 'minimize' to solve integer linear
  programs.

AUTHORS: The 4ti2 team

Ralf Hemmecke
Raymond Hemmecke
Matthias Koeppe
Peter Malkin
Matthias Walter

Bibliography

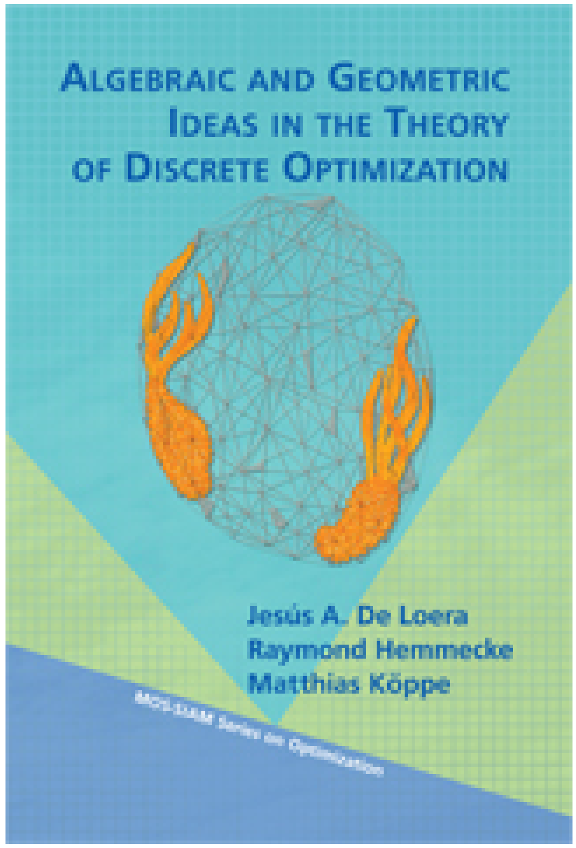
[1] Jesús A. De Loera, Raymond Hemmecke, and Matthias Köppe. *Algebraic and Geometric Ideas in the Theory of Discrete Optimization.* MOS–SIAM Series on Optimization. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2013.

[2] Utz-Uwe Haus, Matthias Köppe, and Robert Weismantel. A primal all-integer algorithm based on irreducible solutions. *Math. Programming, Series B*, 96(2):205–246, 2003.

[3] Matthias Köppe. Erzeugende Mengen für gemischt-ganzzahlige Programme. Diploma thesis, Otto-von-Guericke-Universität Magdeburg, 1999. available from URL `http://www.math.ucdavis.edu/~mkoeppe/art/mkoeppe-diplom.ps`.

[4] Bernd Sturmfels. *Algebraic recipes for integer programming.* arXiv:math.OC/0310194, 2003.

[5] Regina Urbaniak. *Decomposition of Generating Sets and of Integer Programs.* Dissertation, Technische Universität Berlin, 1998.


ALGEBRAIC AND GEOMETRIC
IDEAS IN THE THEORY
OF DISCRETE OPTIMIZATION
Jesús A. De Loera
Raymond Hemmecke
Matthias Köppe
MOS-SIAM Series on Optimization