latter: LattE and 4ti2 in R

David Kahle, Ruriko Yoshida, and Luis Garcia

2016-05-03

Getting started with latter

latter is an R package that provides back-end connections to LattE and 4ti2 executables and front-end tools facilitating their use in the R ecosystem. Many of the functions in **latter** are improved versions of previous implementations found in the **algstat** package **algstat** package. In the future, **algstat** will depend on these connections and functions to perform algebraic statistical tasks.

Note: **latter** assumes you have LattE and 4ti2 installed and **latter** has registered it (i.e. has found its path). If you don't have them, take one of the following two paths to get them. In either case, note that 4ti2 comes with LattE now, so you'll just need to get LattE.

1. If you're used to compiling and installing programs from the command line, just build them from LattE's source files.
2. If you're not used to doing this, check out LattE's binaries.

When you load **latter** (with `library(latter)`) it will try to determine where LattE and 4ti2 executables are on your computer. Where it looks depends on your machine. If you have a Mac, it will look in `/Applications/` and your home directory. If you have a PC and you've installed LattE and 4ti2 with Cygwin, it will use `whereis` to find the executables. If you're not using one of these, you'll have to point to it yourself with `set_latte_path()` and `set_4ti2_path()`. If you're in an active session, and you don't give these functions an argument, it'll open a window through which you can navigate to one of the corresponding packages executables (e.g. `count` for latte and `markov` for 4ti2). If not, you'll need to find the path to these executables yourself and then give them as a character string to one of those functions. For example, `set_latte_path("/Applications/latte/dest/bin/")`.

```r
library(latter)
```

Note: R doesn't know your user profile's path, so it is possible for LattE and 4ti2 functions to work from the command line but not work with **latter** in R. This will happen if you have LattE and 4ti2 installed in unanticipated directories but have put them on your path. In this case, just use the `set_*_path()` functions to manually set the paths.

LattE/4ti2 file formatting

All the connections from R to LattE/4ti2 work by writing temporary files in LattE's format, running the executable which creates more files, and then reading those files back into standard R data structures as needed. The `format_latte()`, `write.latte()`, and `read.latte()` functions do the formatting and saving for **latter**.

The basic geometric object of interest in LattE/4ti2 is a polytope, here defined by a vector inequality $Ax \leq b$, where $A \in \mathbb{Z}^{m \times n}$, $x \in \mathbb{Z}^n$ and $b \in \mathbb{Z}^m$. LattE's primary interest is in counting the number of integer points in such a polytope, but it does much more.

LattE uses a special file format to describe the system $Ax \leq b$, which essentially just represents the inequality as a matrix $[b\ (-A)] \in \mathbb{Z}^{m \times (n+1)}$ and puts the dimensions as the first line of the file. This is the so-called hyperplane representation of the polytope, for more details check out the LattE manual.

Here's an example on the formatting.

```r
(mat <- matrix(1:9, nrow = 3))
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```r
format_latte(mat)
```

```
## [1] "3 3\n1 4 7\n2 5 8\n3 6 9"
```

This may look kind of strange. The problem is that it prints the line breaks as characters instead of actual line breaks, a problem that is fixed when `write.latte()` uses `format_latte()` to write the code to a file. So here's what it looks like normally:

```r
cat(format_latte(mat))
```

```
## 3 3
## 1 4 7
## 2 5 8
## 3 6 9
```

If in addition to inequalities in some of the rows of $Ax \leq b$ you wanted to specify equalities, you could do this by specifying two different rows in A, but that'd be a little redundant. To make things easier LattE formatting allows for setting equality rows (`linearity`) and non-negative rows. The way **latter** deals with this is as attributes to the matrix. Here's an example:

```r
attr(mat, "linearity") <- c(1, 3)
attr(mat, "nonnegative") <- 2
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
## attr(,"linearity")
## [1] 1 3
## attr(,"nonnegative")
## [1] 2
```

```r
cat(format_latte(mat))
```

```
## 3 3
## 1 4 7
## 2 5 8
## 3 6 9
## linearity 2 1 3
## nonnegative 1 2
```

`write.latte()` and `read.latte()` work just as you'd expect. Their names are the only names in **latter** separated with a `.` because they are modeled after the more common read/write functions `read.csv()`/`write.csv()` (or `table`, or other elements); **latter**'s main naming convention is to break words with underscores `_`.

```
(filename <- tempfile())
```

```
## [1] "/var/folders/r3/126_d6t55f5d32tplbg5mk1d0c48s9/T//Rtmpfp35o1/file923229fc5c04"
```

```
write.latte(mat, filename) # the output here is invisible, it's the same as format_latte()
read.latte(filename)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
## attr(,"linearity")
## [1] 1 3
## attr(,"nonnegative")
## [1] 2
```

You can even use `write.latte()` to get the A/b structure back:

```
read.latte(filename, format = "Ab")
```

```
## $A
##      [,1] [,2]
## [1,]   -4   -7
## [2,]   -5   -8
## [3,]   -6   -9
##
## $b
## [1] 1 2 3
```

Most of the time you will never use `format_latte()`, `read.latte()`, and `write.latte()`, because higher level functions will use them behind the scenes for you. Nevertheless, if you need to use them, there exposed for your use.

Lattice point counting

Most LattE/4ti2 programs are available as functions in **latter** and (to the extent possible) they have the same names. For example, LattE's `count` function is implemented in **latter**'s `count()` to determine the number of integer points in a polytope. In this example we count the number of integer points satisfying the constraints $x \geq 0$, $y \geq 0$, and $x + y \leq 10$. In matrix notation, this is

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 10 \end{bmatrix}$$

To use LattE's count to count the number of integer points in the polytope (here polygon), we simply define the A and b and create the matrix:

```r
A <- matrix(c(-1, 0, 0, -1, 1, 1), byrow = TRUE, nrow = 3)
b <- c(0, 0, 10)
(mat <- cbind(b, -A))
```

```
##        b
## [1,]   0  1  0
## [2,]   0  0  1
## [3,]  10 -1 -1
```

We then format it with `format_latte()`...

```r
code <- format_latte(mat)
cat(code)
```

```
## 3 3
##  0  1  0
##  0  0  1
## 10 -1 -1
```

...and pass it to **latter**'s `count()` function:

```r
count(code)
```

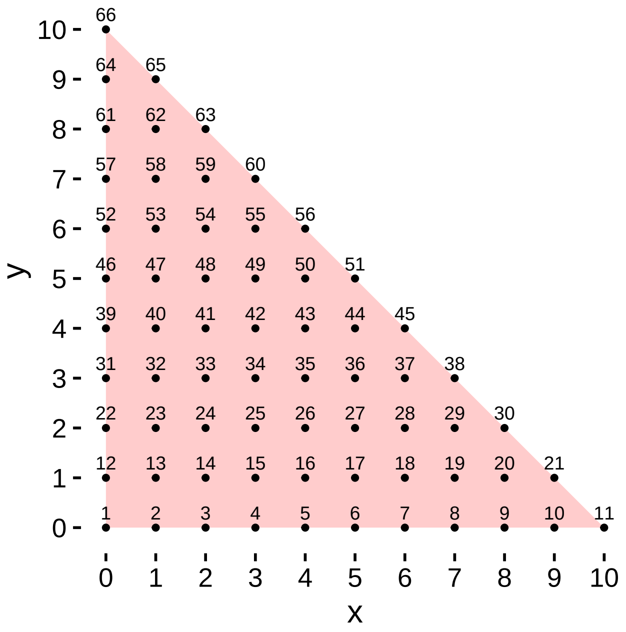
```
## [1] 66
```

It's easy to confirm the solution with a simple visualization:

```r
library(ggplot2); theme_set(theme_classic()); library(magrittr)
polytope <- data.frame(x = c(0, 10, 0), y = c(0, 0, 10))

points   <- expand.grid(x = 0:10, y = 0:10) %>% dplyr::filter(x + y <= 10) %>%
  dplyr::mutate(., number = 1:nrow(.))

ggplot(aes(x = x, y = y), data = polytope) +
  geom_polygon(fill = "red", alpha = .2, size = 1) +
  geom_text(aes(label = number), nudge_y = .3, size = 2.5, data = points) +
  geom_point(data = points, size = .75) +
  scale_x_continuous(breaks = 0:10, minor_breaks = NULL) +
  scale_y_continuous(breaks = 0:10, minor_breaks = NULL) +
  coord_equal()
```



Thus, the `count()` function executes the LattE program `count` on code formatted with `format_latte()` and reads the output back into R. The result is an integer in R, unless the integer is "too large" (10 or more digits), in which case it is returned as a character string. (In this case, you may want to look into R's **gmp** package and the function `as.bigz()`.)

This process can be a bit tedious, so we provide three other ways to do the same thing with far less typing. The first uses a back-end connection to **mpoly** to facilitate more user-friendly specifications. For example, instead of having to create the A and b yourself, combine them, and format them, you can use the following much more simple syntax:

```
count(c("x + y <= 10", "x >= 0", "y >= 0"))
```

```
## [1] 66
```

The two other specifications are a list of vertices. . .

```
vertices <- list(c(0,0), c(10,0), c(0,10))
count(vertices)
```

```
## setting opts = "--vrep"; see ?count
```

```
## [1] 66
```

. . . or a list containing the A and b matrices:

```
count(list(A = A, b = b))
```

```
## [1] 66
```

Ehrhart polynomials

`count()` can also be used to compute the Ehrhart polynomial of an integral polytope. Ehrhart polynomials capture information about the relationship between a polytope's volume and the number of integer points it contains. **latter** uses **mpoly** to represent the polynomials symbolically:

```
count(c("x + y <= 10", "x >= 0", "y >= 0"), opts = "--ehrhart-polynomial")
```

```
## 15 t  +  50 t^2  +  1
```

Integer programming

In addition to table counting, it can also do integer programming with LattE's `latte-maximize` and `latte-minimize` programs. To do this, it (again) uses tools from **mpoly** to do behind-the-scenes rearrangements. For example, if we defined the function $f(x, y) = -2x + 3y$ over the integer points in the triangular region defined in the above example, and wanted to maximize or minimize it, we can use `latte_max()` and `latte_min()`:

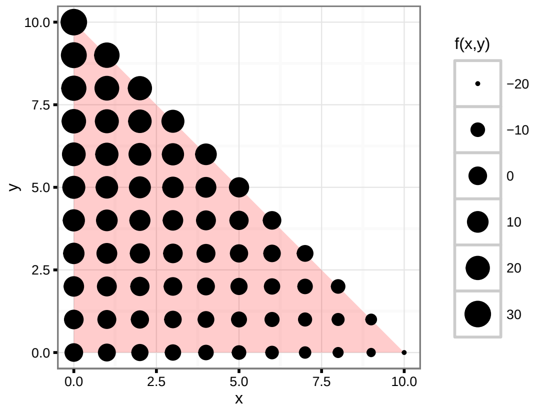
```
latte_max("-2 x + 3 y", c("x + y <= 10", "x >= 0", "y >= 0"))
```

```
## $par
##  x  y
##  0 10
##
## $value
## [1] 30
```

```
latte_min("-2 x + 3 y", c("x + y <= 10", "x >= 0", "y >= 0"))
```

```
## $par
##  x  y
## 10  0
##
## $value
## [1] -20
```

Again we can check that the solution is correct with a simple diagram:



Other applications