Multivariate polynomials in R

Robin K. S. Hankin
Auckland University of Technology

Abstract

In this short article I introduce the **spray** package, which provides some functionality for handling sparse multivariate polynomials; the package is discussed here from a programming perspective. An example from the field of enumerative combinatorics is presented.

Keywords: Multivariate polynomials, R.

1. Introduction

The **multipol** package (Hankin 2008) furnishes the R programming language with functionality for multivariate polynomials. However, the **multipol** package was noted as being inefficient in many common cases: the package stores multivariate polynomials as arrays and this often involves storing many zero elements which consume computational and memory resources unnecessarily.

One suggestion was to use sparse arrays—in which nonzero elements are stored along with an index vector describing their coordinates—instead of arrays. In this short document I introduce the **spray** package which provides functionality for sparse arrays and interprets them as multivariate polynomials.

1.1. Existing work

The **slam** package (Hornik, Meyer, and Buchta 2014) provides some sparse array functionality but is not intended to interpret arbitrary dimensional sparse arrays as multivariate polynomials. The **mpoly** (Kahle 2013) package handles multivariate polynomials but does not accept negative powers, nor is it designed for efficiently processing large multivariate polynomials; I present some timings below. The **mpoly** package is different in philosophy from both the **spray** package and **multipol** in that **mpoly** is more "symbolic" in the sense that it admits—and handles appropriately—named variables, whereas my packages do not make any reference to the *names* of the variables. As Kahle points out, naming the variables allows a richer and more natural suite of functionality; straightforward **mpoly** idiom is somewhat strained in **spray**.

2. Package philosophy

The **spray** package does not interact or depend on **multipol** in any way, owing to the very different design philosophies used. The package uses the C++ Standard Template Library's map class to store and retrieve elements.

A *map* is an associative container that stores values indexed by a key, which is used to sort and uniquely identify the values. In the package, the key is a vector object or a deque object with (signed) integer elements.

2.1. Compile-time options

At compile time, the package offers two options. Firstly one may use the unordered_map class in place of the map class. This option is provided in the interests of efficiency. An unordered map has lookup time $\mathcal{O}(1)$ (compare $\mathcal{O}(\log n)$ for the map class), but overhead is higher.

The other option offered is the nature of the key, which may be either vector class or deque class. Elements of a vector are guaranteed to be contiguous in memory, unlike a deque. This does not appear to make a huge difference to timings, but the default (unordered_map indexed by a vector) appears to be marginally the fastest option.

3. The package in use

To create a sparse array object, one specifies a matrix of indices M with each row corresponding to the position of a nonzero element, and a numeric vector of values:

```
> library("spray")
> M <- matrix(c(0,0,0,1,0,0,1,1,1,2,0,3),ncol=3)
> M

     [,1] [,2] [,3]
[1,]    0    0    1
[2,]    0    0    2
[3,]    0    1    0
[4,]    1    1    3

> S1 <- spray(M, 1:4)
> S1

           val
 0 0 1  =    1
 0 1 0  =    3
 0 0 2  =    2
 1 1 3  =    4
```

Thus S1[0,0,2] = 2. Note that the representation of the spray object does not preserve the order of the index rows in the argument, although a particular index row is associated unambiguously with a unique numeric value. Replace methods work as expected:

```
> S1[diag(3)] <- -3
> S1
```

```
            val
 0 0 1  =    -3
 0 1 0  =    -3
 0 0 2  =     2
 1 1 3  =     4
 1 0 0  =    -3
```

Note that a value with an existing index is overwritten, while new elements are created as necessary. Addition is implemented:

```
> S2 <- spray(matrix(c(6,0,1,7,0,1,8,2,3),nrow=3), c(17,11,-4))
> S2

            val
 6 7 8  =    17
 0 0 2  =    11
 1 1 3  =    -4

> S1+S2

            val
 0 0 1  =    -3
 0 1 0  =    -3
 0 0 2  =    13
 6 7 8  =    17
 1 0 0  =    -3
```

Note that element `[0,0,2]` becomes $2 + 11 = 13$, while element `[1,1,3]` vanishes.

Repeated index rows

If any row of the index matrix is identical to any other row, then this is interpreted as an error. However, on occasion, the user may wish to sum values over repeated index rows and this is done by setting the `addrepeats` argument to `TRUE`:

```
> spray(matrix(0:5,8,3),addrepeats=TRUE)

            val
 0 2 4  =     2
 1 3 5  =     2
 5 1 3  =     1
 2 4 0  =     1
 4 0 2  =     1
 3 5 1  =     1
```

4. Sparse arrays interpreted as multivariate polynomials

One natural and useful interpretation of a sparse array is as a multivariate polynomial:

```
> options(polyform=TRUE)
> S1

 -3*z -3*y +2*z^2 +4*x*y*z^3 -3*x
```

(only the print method has changed; `S1` is as before).

```
> S1*S2

 +12*x^2*y*z^3 -33*y*z^2 -33*x*z^2 +36*x*y*z^5 +22*z^4 +12*x*y*z^4 -16*x^2*y^2*z^6 +12*x*y
```

It is possible to introduce an element of symbolic calculation, exhibiting familiar algebraic identities:

```
> x <- lone(1,2)
> y <- lone(2,2)
> (x+y)^3

 +3*x^2*y +3*x*y^2 +y^3 +x^3

> (1+x+y)^3

1 +3*x^2 +3*y +3*x +6*x*y +3*y^2 +y^3 +x^3 +3*x^2*y +3*x*y^2
```

4.1. Further functionality

Negative indices have a natural intrerpretation as multivariate polynomials:

```
> S1[0,-1,-2] <- 1
> S1

 -3*z -3*y +2*z^2 +4*x*y*z^3 -3*x +y^-1*z^-2
```

And multivariate polynomials have a natural interpretation as functions:

```
> f <- as.function(S1)
> f(matrix(1:9,3,3))

[1]  5550.005 20563.003 52596.002
```

The package also includes the ability to sum across one or more dimensions:

```
> asum(S1,2)

-3 -3*y +y^-2 -3*x +2*y^2 +4*x*y^3
```

Other algebraic operations include substitution and partial differentiation. Consider the homogenous polynomial in three variables; to substitute $y = 1.5$ into this, we use the `subs()` function:

```
> options(polyform=TRUE)
> homog(3,3)

 +x^2*y +y^3 +x^2*z +x^3 +x*y^2 +x*y*z +y^2*z +x*z^2 +z^3 +y*z^2

> subs(homog(3,3),dims=2,1.5)

4.5 +1.5*x^2 +3*y +1.5*x*y +3*x +1.5*y^2
```

5. An example

Suppose we consider a chess knight and ask how many ways are there for the knight to return to its starting square in 6 moves. Such questions are most naturally answered by using generating functions. We define `chess_knight`, a spray object with rows corresponding to the possible moves the chess piece may make:

```
> chess_knight <-
+   spray(matrix(
+       c(1,2,1,-2,-1,2,-1,-2,2,1,2,-1,-2,1,-2,-1),
+       byrow=TRUE,ncol=2))
> options(polyform=FALSE)
> chess_knight

           val
  1  2  =    1
  1 -2  =    1
 -1  2  =    1
 -1 -2  =    1
  2  1  =    1
  2 -1  =    1
 -2  1  =    1
 -2 -1  =    1
```

Then `chess_knight[i,j]` gives the number of ways the piece can move from square `[0,0]` to `[i,j]`; and `(chess_knight^n)[i,j]` gives the number of ways the piece can reach `[i,j]` in `n` moves. To calculate the number of ways that the piece can return to its starting square we simply raise `chess_knight` to the sixth power and extract the `[0,0]` coefficient:

```
> constant(chess_knight^6,drop=TRUE)

[1] 5840
```

(function `constant()` extracts the coefficient corresponding to zero power). One natural generalization would be to arbitrary dimensions. A d-dimensional knight moves two squares in one direction, followed by one square in another direction:

```
> knight <- function(d){
+   n <- d*(d-1)
+   out <- matrix(0,n,d)
+   out[cbind(rep(seq_len(n),each=2),c(t(which(diag(d)==0,arr.ind=TRUE))))] <- seq_len(2)
+   spray(rbind(out,-out,`[<-`(out,out==1,-1),`[<-`(out,out==2,-2)))
+ }
```

Then, considering a four-dimensional chessboard (Figure 1):

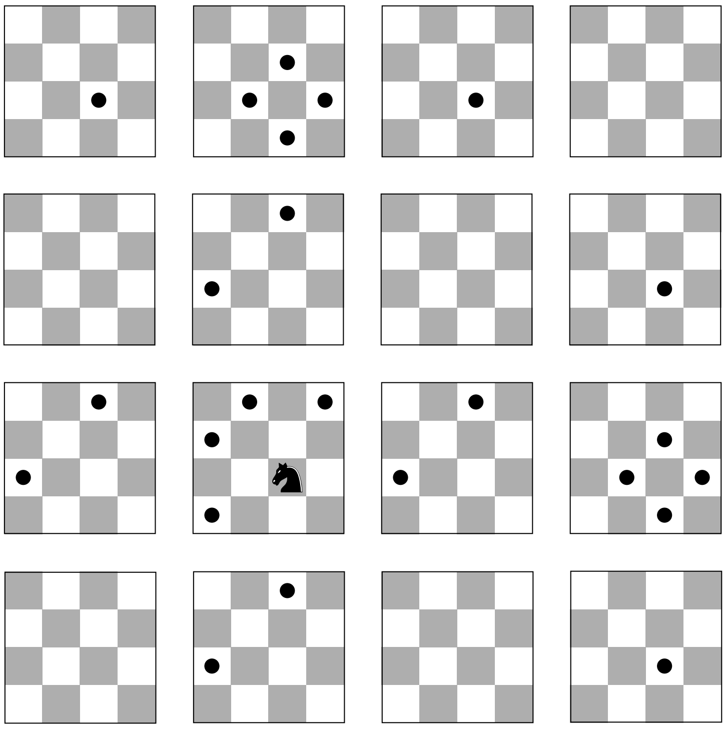

Figure 1: Four-dimensional knight on a $4 \times 4 \times 4 \times 4$ board. Cells attacked by the knight shown by dots

```
> constant(knight(4)^6, drop=TRUE)

[1] 10117920
```

It is in such cases that the efficiency of the map class becomes evident: on my system (3.4 GHz Intel Core i5 iMac)), the above call took just under 0.6 seconds of elapsed time whereas the same[1] calculation took over 173 seconds using **mpoly**.

[1]Because **mpoly** does not accept negative powers, the calculation was equivalent to `(knight(4) + xyz(4)^2)^6`. Also note that the **multipol** package is not able to execute these commands in a reasonable time.

If we want the number of ways to return to the starting point in 6 or fewer moves, we can simply add the unit multinomial and take the sixth power of the sum:

```
> constant((1+knight(4))^6, drop=TRUE)
```

```
[1] 10306561
```

(1.2 seconds for **spray** vs 275 seconds for **mpoly**). For 8 moves, the differences are more pronounced, with **spray** taking 5.1 seconds and **mpoly** requiring more than 1500 seconds).

References

Hankin RKS (2008). “Programmers’ Niche: Multivariate polynomials in R.” *R News*, **8**(1), 41–45. URL `http://CRAN.R-project.org/doc/Rnews/`.

Hornik K, Meyer D, Buchta C (2014). *slam: Sparse Lightweight Arrays and Matrices*. R package version 0.1-32, URL `https://CRAN.R-project.org/package=slam`.

Kahle D (2013). “mpoly: Multivariate Polynomials in R.” *The R Journal*, **5**(1), 162–170. URL `http://journal.r-project.org/archive/2013-1/kahle.pdf`.

Affiliation:

Robin K. S. Hankin
Auckland University of Technology
New Zealand