mpoly: Multivariate Polynomials in R

by David Kahle

Abstract The **mpoly** package is a general purpose collection of tools for symbolic computing with multivariate polynomials in R. In addition to basic arithmetic, **mpoly** can take derivatives of polynomials, compute Gröbner bases of collections of polynomials, and convert polynomials into a functional form to be evaluated. Among other things, it is hoped that **mpoly** will provide an R-based foundation for the computational needs of algebraic statisticians.

Introduction

At a basic level, R is not designed for symbolic computing. Unlike computer algebra systems founded on rational arithmetic and representations like Mathematica (Wolfram Research, 2012) and Maple (Maplesoft, 2012), R's dependence on floating-point arithmetic naturally lends itself to its purpose—data analysis, numerical optimization, large(ish) data and so on. Nevertheless, the advent of algebraic statistics and its contributions to asymptotic theory in statistical models, experimental design, multiway contingency tables, and disclosure limitation has increased the need for R – as a practical tool for applied statistics – to be able to do some relatively basic operations and routines with multivariate polynomials (Diaconis and Sturmfels, 1998; Drton et al., 2009). **mpoly** is designed to try to fill this void entirely within R (Kahle, 2012). This article is intended to demonstrate its capabilities to a discipline/specialty-neutral audience.

A bit more specifically, the **mpoly** package is a general purpose collection of tools for symbolic computing with multivariate polynomials in R. While new, it is not the first package proposed to deal with multivariate polynomials. We therefore begin with a discussion of the current package intended to fulfill this need, **multipol**, in order to motivate the need for and capabilities of **mpoly** (Hankin, 2010).

Background: the multipol package

R currently has three packages which deal with polynomials – **polynom**, **PolynomF**, and **multipol**. The first two (the second being a reboot of the first) deal exclusively with univariate polynomials and as such are not suitable for the general purpose needs cited in the introduction (Venables et al., 2009; Venables, 2010). The third, **multipol**, implements multivariate polynomials in a natural way suitable for some traditional applications (e.g. simple interactions in generalized linear models) but exhibits two limitations which impede its more general use and suggest a fresh re-envisioning of multivariate polynomials in R is needed (Hankin, 2008). The first limitation has to do with the impossibility of polynomial arithmetic, and the second has to do with storing sparse polynomials.

multipol is an implementation of multivariate polynomials based on the S3 class object `"multipol"`, an unnamed multidimensional array. Consequently, **multipol** is fundamentally dependent on arrays as its basic data structure. This is evident in the construction of a `"multipol"` object:

```
> library("multipol")
Loading required package: abind

Attaching package: 'multipol'

The following object(s) are masked from
  'package:base':

    single

> a <- as.multipol(array(1:12, c(2,3,2)))
> a
, , z^0

    y^0 y^1 y^2
```

```
x^0   1   3   5
x^1   2   4   6

, , z^1

    y^0 y^1 y^2
x^0   7   9  11
x^1   8  10  12
```

In normal notation, a moment's consideration reveals that the polynomial defined above is

$$\begin{aligned} & 1 + 2x + 3y + 4xy + 5y^2 + 6xy^2 + 7z \\ + \quad & 8xz + 9yz + 10xyz + 11y^2z + 12xy^2z \end{aligned}$$

in the ring $R[x, y, z]$. (A similar pretty printing can be enabled by setting `options(showchars = TRUE)`.) Thus, the elements of the array are the coefficients on the terms appearing in the polynomial. From here, the limitations are immediately accessible.

1. Notice that the definition of a multivariate polynomial in the example above makes no reference to the variable names `x` and `y`—they are determined when the `print` method is called. In other words, a `"multipol"` object has no notion of variables whatsoever, the variables are only generated when asked for via the `print` method for class `"multipol"`. This has profound implications on polynomial arithmetic in **multipol**. In particular, arithmetic is only *possible* if two polynomials have the same number of dimensions (variables), and is only *meaningful* if those dimensions represent the same variables in the same order, the latter of which is never checked because there are no names specified which can be checked.

For example, suppose one wants to add or multiply the polynomials

$$a = 1 + 2x + 3y + 4xy \tag{1}$$

and

$$b = 1 + 2x + 3y + 4xy + 5z + 6xz + 7yz + 8xyz, \tag{2}$$

both in $R[x, y, z]$. Mathematically speaking, it is obvious that the sums and products are well-defined. However, they cannot be computed because they have a different number of variables:

```
> a <- multipol( array(1:4, c(2,2)) )
> b <- multipol( array(1:8, c(2,2,2)) )
> a + b
Error: length(dima) == length(dimb) is
  not TRUE
> a * b
Error: length(dim(a)) == length(dim(b)) is
  not TRUE
```

If the polynomials have the same number of variables, the arithmetic may be incorrect if those variables are different:

```
> options(showchars = TRUE)
> ( a <- multipol( as.array(
+  c('x^0' = 0, 'x^1' = 1) )))
[1] 1*x^1
> ( b <- multipol( as.array(
+  c('y^0' = 0, 'y^1' = 1) )))
[1] 1*x^1
> a + b
[1] 2*x^1
```

The result would seem to indicate that $a + b = x + y = 2x$, which is clearly in error. The location of the problem can easily be found however since we printed the polynomials at each step—regardless of our attempt to force (in a simple way) the labeling of the variables, we achieve the same incorrect result.

2. The array representation does not allow for sparse polynomials. For our purposes, a *sparse (multivariate) polynomial* of multidegree $\boldsymbol{d}$ is one which has "few" terms $c_{\boldsymbol{d}'}\boldsymbol{x}^{\boldsymbol{d}'}$ with multidegree $\boldsymbol{d}' \leq \boldsymbol{d}$ (element-wise) with nonzero coefficients. As an example, consider the polynomial

$$ab^2 + bc^2 + cd^2 + \cdots + yz^2 + za^2. \tag{3}$$

Since **multipol** represents multivariate polynomials with arrays, the representation requires a 26 dimensional array (a dimension for each variable) each with three levels (e.g. `a^0`, `a^1`, `a^2`). This amounts to an array with $3^{26} = 2,541,865,828,329$ cells, all but 26 of which are nonzero—a whopping 20.33TB of space if the coefficients are stored in a double-precision floating point format.

This section has introduced two shortcomings of **multipol** which significantly limit its potential use for most practical applications. It should be noted, however, that **multipol** was not intended for use with "large" polynomials. It was built with enumerative combinatorics in mind, and its inefficiency with larger polynomials was readily acknowledged in Hankin (2008), where it is suggested that perhaps a sparse array representation (as is available in Mathematica) or outsourcing to C++ routines could alleviate the burdens of the array representation. Its inefficiency for different purposes should therefore not be considered a flaw so much as outside the package's scope. Nevertheless, the arithmetic and storage problems caused by the array representation are fatal limitations for any problem which deals with large polynomial rings (i.e., ones with many variables). Enter **mpoly**.

mpoly – the basics

mpoly is a complete reenvisioning of how multivariate polynomials and symbolic computing with multivariate polynomials should be implemented in R. Unlike **multipol**, **mpoly** uses as its most basic data structure the list. This fundamental change allows us to dispense of both issues with **multipol** discussed in the previous subsection.

In **mpoly**, an `"mpoly"` object is an S3 class object which is a list. The elements of the list are each named numeric vectors, with unique names including `coef`. Naturally, each element of an `"mpoly"` object corresponds to a term in the polynomial. The names of the elements correspond to the variables, and the values to the degrees; `coef` is the coefficient of the term. Constructing an `"mpoly"` object in this way is very straightforward using the constructor `mpoly`.

```
> library("mpoly")
> termList <- list(
+   c(coef = 1),
+   c(x = 10, coef = 2),
+   c(x = 2, coef = 3),
+   c(y = 5, coef = 4),
+   c(x = 1, y = 1, coef = 5))
> termList
[[1]]
coef
   1
[[2]]
   x coef
  10    2
[[3]]
   x coef
   2    3
[[4]]
   y coef
   5    4
[[5]]
   x    y coef
   1    1    5
> poly <- mpoly(termList)
> class(poly)
[1] "mpoly"
```

Unlike multivariate polynomials in **multipol**, those in **mpoly** not only have variable names but also an intrinsic variable order which is taken to be the unique names of elements of the `"mpoly"` object minus `coef`.[1] This can be seen with the `vars` function.

```
> vars(poly)
[1] "x" "y"
```

[1] To be clear, this is in the `unique(names(unlist(mpoly)))` sense; the order of the terms matters when determining the intrinsic order.

Viewing a multivariate polynomial as a list is a cumbersome task. To make things easier, a `print` method for `"mpoly"` objects exists and is dispatched when the object is queried by itself.

```
> poly
1  +  2 x^10  +  3 x^2  +  4 y^5  +  5 x y
```

One of the important considerations in polynomial algebra is the ordering of the terms of a multivariate polynomial. Notice the order of the terms presented in the printed version of the `"mpoly"` object; it is the order in which the terms are coded in the `"mpoly"` object itself. This can be changed in either of two ways.

First, it can be changed via the `print` method, which accepts arguments `order` for the total order used (lexicographic, graded lexicographic, and graded reverse lexicographic) and `varorder` for a variable order different than the intrinsic order. When an order is requested but a variable order is not specified, the method messages the user to alert them to the intrinsic variable order being used.[2]

```
> print(poly, order = 'lex')
using variable ordering - x, y
2 x^10  +  3 x^2  +  5 x y  +  4 y^5  +  1
> print(poly, order = 'grlex')
using variable ordering - x, y
2 x^10  +  4 y^5  +  3 x^2  +  5 x y  +  1
> print(poly, order = 'lex',
+  varorder = c('y', 'x'))
4 y^5  +  5 y x  +  2 x^10  +  3 x^2  +  1
> print(poly, order = 'glex',
+  varorder = c('y', 'x'))
2 x^10  +  4 y^5  +  5 y x  +  3 x^2  +  1
```

Second, the elements of the `"mpoly"` object can be reordered to create a new `"mpoly"` object using the `reorder` method.

```
> poly
1  +  2 x^10  +  3 x^2  +  4 y^5  +  5 x y
> (poly2 <- reorder(poly, order = 'lex'))
using variable ordering - x, y
2 x^10  +  3 x^2  +  5 x y  +  4 y^5  +  1
> unclass(poly2)
[[1]]
   x coef
  10    2
[[2]]
   x coef
   2    3
[[3]]
   x    y coef
   1    1    5
[[4]]
   y coef
   5    4
[[5]]
coef
   1
```

Defining polynomials: `mpoly` vs `mp`

The major workhorse of the package is the constructor `mpoly` itself. In particular, polynomial *reduction* (combining of like terms) and *regularization* (combining of coefficients and like variables within terms) are both performed when the multivariate polynomials are constructed with `mpoly`.

```
> termList <- list(
+   c(x = 1, coef = 1),
```

[2] This is a subtle point. It is very possible that a polynomial in the ring $R[x, y]$ is coded with the intrinsic order $y > x$ and that, as a consequence, the typical lexicographic order will not be the one intended. The messaging is used to make clear what order is being used.

```
+   c(x = 1, coef = 2)
+ )
> mpoly(termList) # x + 2x
3 x
> termList <- list(
+   c(x = 5, x = 2, coef = 5, coef = 6, y = 0)
+ )
> mpoly(termList) # x^5 * x^2 * 5 * 6 * y^0
30 x^7
```

While the constructor `mpoly` is nice, it is inconvenient to have to keep specifying lists in order to define polynomials. The `mp` function was designed for this purpose and is intended to make defining multivariate polynomials quick and easy by taking them in as character strings and parsing them into `"mpoly"` objects.

```
> ( p <- mp('10 x + 2 y 3 + x^2 5 y') )
10 x  +  6 y  +  5 x^2 y
> is(p, "mpoly")
[1] TRUE
> unclass(p)
[[1]]
   x coef
   1   10
[[2]]
   y coef
   1    6
[[3]]
   x    y coef
   2    1    5
```

This parsing is a nontrivial process and depends heavily on the specification of the polynomial in the string. The `mp` function must first determine the variables that the user is specifying (which must be separated by spaces for disambiguation) and then construct the list to send to `mpoly` to construct the object. Because it is passed through `mpoly`, the `"mpoly"` object returned by `mp` is reduced and regularized.

```
> mp('x^2 + 10 x 6 x + 10 x 6 x y y 2')
61 x^2  +  120 x^2 y^2
```

Arithmetic, calculus, and algebra

mpoly has much more to offer than simply defining polynomials. Methods are available for addition, subtraction, multiplication, exponentiation and equality as well. Moreover, since `"mpoly"` objects know their variable names intrinsically, we can perform arithmetic with whichever polynomials we like. For example, the arithmetic with a and b from (1) and (2) is easy –

```
> a <- mp('1 + 2 x + 3 y + 4 x y')
> b <- mp(paste('1 + 2 x + 3 y + 4 x y + ',
   '5 z + 6 x z + 7 y z + 8 x y z'))
> a + b
2  +  4 x  +  6 y  +  8 x y  +  5 z
  +  6 x z  +  7 y z  +  8 x y z
> b - a
5 z  +  6 x z  +  7 y z  +  8 x y z
> a * b
1  +  4 x  +  6 y  +  20 x y  +  5 z  + 16 x z
 +  22 y z  +  60 x y z  +  4 x^2  +  16 x^2 y
 +  12 x^2 z  +  40 x^2 y z  +  9 y^2
 +  24 x y^2  +  21 y^2 z  +  52 x y^2 z
 +  16 x^2 y^2  +  32 x^2 y^2 z
```

Exponentiation and equality are also available.

```
> a^2
1  +  4 x  +  6 y  +  20 x y  +  4 x^2  +
```

```
  16 x^2 y  +  9 y^2  +
  24 x y^2  +  16 x^2 y^2
> a == b
[1] FALSE
> ( c <- mpoly(a[c(2,1,4,3)]) ) # reorder a
4 y x  +  3 y  +  2 x  +  1
> a == c
[1] TRUE
```

Here also each of the results are reduced and regularized. While the computations are done entirely in R, they are quite efficient; each of the above calculations is virtually instantaneous.

But **mpoly** does not stop there. The basic operations of the differential calculus, partial differentiation and gradients, are also available to the user and are efficient. A `deriv` method exists for `"mpoly"` objects which can be dispatched,[3] and the `gradient` function is built on `deriv` to compute gradients.

```
> deriv(b, 'x')
8 y z  +  4 y  +  6 z  +  2
> gradient(b)
8 y z  +  4 y  +  6 z  +  2
8 x z  +  4 x  +  7 z  +  3
8 x y  +  6 x  +  7 y  +  5
```

The gradient is a good example of another class object in the **mpoly** package, the `"mpolyList"`. `"mpolyList"` objects are simply lists of `"mpoly"` objects and are used to hold vectors of multivariate polynomials. They can be easily specified using the `mp` function on a vector of character strings.

```
> ( ps <- mp(c('x + y + z', 'x + z^2')) )
x  +  y  +  z
x  +  z^2
> str(ps, 1)
List of 2
 $ :List of 3
  ..- attr(*, "class")= chr "mpoly"
 $ :List of 2
  ..- attr(*, "class")= chr "mpoly"
 - attr(*, "class")= chr "mpolyList"
```

The viewing of `"mpolyList"` objects is made possible by a `print` method for `"mpolyList"` objects just like for `"mpoly"` objects. Moreover addition, subtraction, and multiplication are defined for `"mpolyList"` objects as well; they each operate element-wise.

In algebraic geometry, a Gröbner basis of a polynomial ideal (or collection of polynomials generating an ideal) is a collection of polynomials which generates the same ideal and exhibits other nice properties, such as zero remainders for polynomials in the ideal and unique remainders for those outside it. They are foundational to the study of polynomial ideals and their associated varieties. In addition to differentiation, **mpoly** can also compute Gröbner bases of collections of multivariate polynomials (`"mpolyList"`) by passing the proper objects in the proper syntax to the **rSymPy** package which has an implementation of Buchberger's algorithm (Grothendieck and Bellosta, 2010).[4] The computations are performed by a Java based Python implementation and are quite fast once the Java Virtual Machine (JVM) has been initialized. The Gröbner basis is then returned as an `"mpolyList"` object.

```
> polys <- mp(c('t^4 - x', 't^3 - y',
+  't^2 - z'))
> gb <- grobner(polys)
using variable ordering - t, x, y, z
Loading required package: rJava
> gb
-1 z  +  t^2
t y  -  z^2
-1 y  +  z t
x  -  z^2
y^2  -  z^3
```

[3] `deriv` does not call the default method from **stats**.

[4] Buchberger's algorithm is the standard method of calculating Gröbner bases, however, faster methods are known.

```
> class(gb)
[1] "mpolyList"
```

Moreover, `grobner` can calculate Gröbner bases with respect to various monomial orders and any variable ordering.

```
> polys <- mp(c('x^2 - 2 y^2', 'x y - 3'))
> grobner(polys, varorder = c('x', 'y'))
3 x  -  2 y^3
-9  +  2 y^4
> grobner(polys, varorder = c('x', 'y'),
+  order = 'grlex')
-3 x  +  2 y^3
x^2  -  2 y^2
-3  +  x y
```

The task of computing Gröbner bases is the only job **mpoly** outsources from R. This is because implementations of Gröbner bases algorithms are (1) complex and highly prone to error, (2) scarce and therefore difficult to emulate, and (3) highly iterative; thus it was thought best to leave the routine to more expert programmers and frameworks outside R. Unfortunately, there is currently no Gröbner walk algorithm available to convert a Gröbner basis in one monomial order to a Gröbner basis in another, a technique often used to quickly compute Gröbner bases in more difficult orders (e.g. lexicographic) from "easier" ones (e.g. graded reverse lexicographic), so there are still a number of improvements which can be made.

Evaluating polynomials

Apart from being interesting algebraic objects, polynomials can of course be thought of as functions. To access this functional perspective, we can convert an `"mpoly"` or `"mpolyList"` object to a `function` using the `"mpoly"` method of `as.function`. This is particularly suited to R's strengths since R is geared towards evaluation and numerical optimization

```
> library("ggplot2"); theme_set(theme_bw())
> ( p <- mp('x') * mp('x - .5') *
+  mp('x -  1') ) # x(x-.5)(x-1)
x^3  -  1.5 x^2  +  0.5 x
> f <- as.function(p)
f(x)
> f
function(x){x**3  -  1.5 * x**2  +  0.5 * x}
<environment: 0x1218bc270>
> f(10)
[1] 855
> s <- seq(-.1, 1.1, length.out = 201)
> df <- data.frame(x = s, y = f(s))
> qplot(x, y, data = df, geom = 'path') +
+   geom_hline(yintercept = 0,
+     colour = I('red'))
```

The plot generated is included in Figure 1, where one can see that the function has the correct zeros. For multivariate polynomials the syntax is the same, and `as.function` can provide the function with (1) a vector argument (ideal for passing to optimization routines such as `optim`) or (2) a sequence of arguments.

```
> mpoly <- mp('x + 3 x y + z^2 x')
>
> f <- as.function(mpoly)
f(.) with . = (x, y, z)
> f(1:3)
[1] 16
>
> f <- as.function(mpoly, vector = FALSE)
f(x, y, z)
> f(1, 2, 3)
[1] 16
```

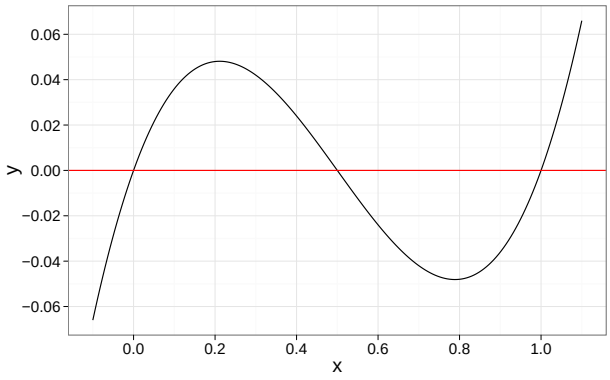

Figure 1: The `as.function` method for `"mpoly"` objects.

`"mpolyList"` objects can be converted into functions in the same way. Note that in both cases the user is messaged with the appropriate syntax for the resulting function.

```
> polys <- mp(c('x + 1', 'y^2 + z'))
> f <- as.function(polys)
f(.) with . = (x, y, z)
> f(1:3)
[1] 2 7
```

Another neat example for illustrating this is visualizing Rosenbrock's "banana" function,

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2,$$

which is a common test example for optimization routines (Figure 2).

```
> f <- mp(paste('1 - 2 x + x^2 + 100 x^4 - ',
+  '200 x^2 y + 100 y^2'))
> f <- as.function(f)
> df <- expand.grid(
+  x = seq(-2, 2, .01),
+  y = seq(-1, 3, .01)
+ )
> df$f <- apply(df, 1, f)
> library("scales")
> qplot(x, y, data = df,
+  geom = c('raster', 'contour'),
+  fill = f + .001, z = f,
+    colour = I('white'), bins = 6) +
+  scale_fill_gradient2(
+    low='blue', mid='green', high='red',
+    trans = 'log10', guide = 'none'
+  )
```

Conclusion and future directions

While other functionality is also provided (such as a `terms` method, a function to compute least common multiples, and some combinatorial functions), the presented ones are the main contributions of the **mpoly** package. Put together, they provide a package which is not only user-friendly, efficient, and useful for polynomial algebra, but also a solid foundation upon which further developments can be made to make polynomials more accessible in R. In particular, it provides a nice starting point for any future package dealing with algebraic statistics.

There are, however, several improvements which can be made. First, the primary constructor function `mpoly` can be made significantly faster by outsourcing its job to C++, as previously suggested (Hankin, 2008). Second, improvements can be made to speed up some of the arithmetic, for example addition-chain exponentiation for polynomial exponents (as opposed to iterative multiplication). Last, improvements can be made into the flexibility of the facilitated constructor `mp`. In particular, parenthetical expressions are not currently handled but would be of great practical use.

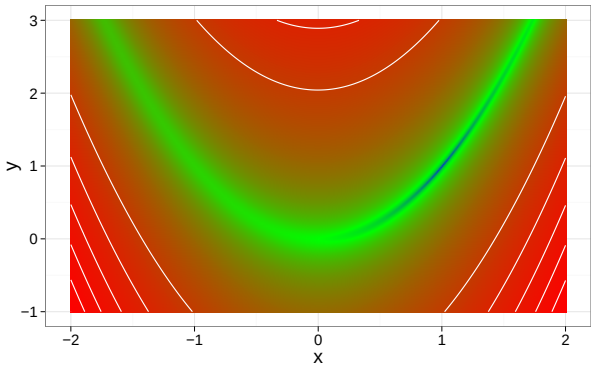

Figure 2: The Rosenbrock function plotted by evaluating an `"mpoly"` object via `as.function` and `apply` with vector arguments.

Acknowledgments

The author would like to thank Martyn Plummer and two anonymous reviewers for useful suggestions which made this and future work better.

Bibliography

P. Diaconis and B. Sturmfels. Algebraic algorithms for sampling from conditional distributions. *The Annals of Statistics*, 26(1):363–397, 1998. [p1]

M. Drton, B. Sturmfels, and S. Sullivant. *Lectures on Algebraic Statistics*. Birkhäuser, 2009. [p1]

G. Grothendieck and C. J. G. Bellosta. *rSymPy: R Interface to SymPy Computer Algebra System*, 2010. URL http://CRAN.R-project.org/package=rSymPy. R package version 0.2-1.1. [p6]

R. K. S. Hankin. Programmers' niche: Multivariate polynomials in R. *R News*, 8(1):41–45, 2008. [p1, 3, 8]

R. K. S. Hankin. *multipol: Multivariate Polynomials*, 2010. URL http://CRAN.R-project.org/package=multipol. R package version 1.0-4. [p1]

D. Kahle. *mpoly: Symbolic Computation and More with Multivariate Polynomials*, 2012. URL http://CRAN.R-project.org/package=mpoly. R package version 0.0.1. [p1]

Maplesoft. *Maple 16*. Waterloo, Canada, 2012. URL http://www.maplesoft.com/. [p1]

B. Venables. *PolynomF: Polynomials in R*, 2010. URL http://CRAN.R-project.org/package=PolynomF. R package version 0.94. [p1]

B. Venables, K. Hornik, and M. Maechler. *polynom: A Collection of Functions to Implement a Class for Univariate Polynomial Manipulations*, 2009. URL http://CRAN.R-project.org/package=polynom. R package version 1.3-6. [p1]

Wolfram Research. *Mathematica 9.0*. Champaign, 2012. URL http://www.wolfram.com/. [p1]

David Kahle
Baylor University
Department of Statistical Science
One Bear Place #97140
Waco, TX 76798
david_kahle@baylor.edu