The `freealg` package: the free algebra in R

Robin K. S. Hankin

2022-01-06

Introduction

The `freealg` package provides some functionality for the free algebra, using the Standard Template library of `C++`, commonly known as the `STL`. It is very like the `mvp` package for multivariate polynomials, but the indeterminates do not commute: multiplication is word concatenation.

The free algebra

A vector space over a commutative field $\mathbb{F}$ (here the reals) is a set $\mathbb{V}$ together with two binary operations, addition and scalar multiplication. Addition, usually written $+$, makes $(\mathbb{V},+)$ an Abelian group. Scalar multiplication is usually denoted by juxtaposition (or sometimes $\times$) and makes $(\mathbb{V},\cdot)$ a semigroup. In addition, for any $a,b\in\mathbb{F}$ and $\mathbf{u},\mathbf{v}\in\mathbb{V}$, the following laws are satisfied:

- Compatibility: $a\left(b\mathbf{v}\right)=(ab)\mathbf{v}$
- Identity: $1\mathbf{v}=\mathbf{v}$, where $1\in\mathbb{F}$ is the multiplicative identity
- Distributivity of vector addition: $a\left(\mathbf{u}+\mathbf{v}\right)=a\mathbf{u}+b\mathbf{v}$
- Distributivity of field addition: $(a+b)\mathbf{v}=a\mathbf{v}+b\mathbf{v}$

An *algebra* is a vector space endowed with a binary operation, usually denoted by either a dot or juxtaposition of vectors, from $\mathbb{V}\times\mathbb{V}$ to $\mathbb{V}$ satisfying:

- Left distributivity: $\mathbf{u}\cdot(\mathbf{v}+\mathbf{w})=\mathbf{u}\cdot\mathbf{v}+\mathbf{u}\cdot\mathbf{w}$.
- Right distributivity: $(\mathbf{u}+\mathbf{v})\cdot\mathbf{w}=\mathbf{u}\cdot\mathbf{w}+\mathbf{v}\cdot\mathbf{w}$.
- Compatibility: $(a\mathbf{u})\cdot(b\mathbf{v})=(ab)(\mathbf{u}\cdot\mathbf{v})$.

There is no requirement for vector multiplication to be commutative. Here we assume associativity so in addition to the axioms above we have

- Associativity: $(\mathbf{u}\cdot\mathbf{v})\cdot\mathbf{w}=\mathbf{u}\cdot(\mathbf{v}\cdot\mathbf{w})$.

The *free associative algebra* is, in one sense, the most general associative algebra. Here we follow standard terminology and drop the word 'associative' while retaining the assumption of associativity. Given a set of indeterminates $\{X_1,X_2,\ldots,X_n\}$ one considers the set of *words*, that is, finite sequences of indeterminates. The vector space is then scalar multiples of words together with a formal vector addition; words are multiplied by concatenation. This system automatically satisfies the axioms of an associative algebra together with the requirement that addition of vectors satisfies distributivity.

The free algebra is the free R-module with a basis consisting of all words over an alphabet of symbols with multiplication of words defined as concatenation. Thus, with an alphabet of $\{x,y,z\}$ and

$$A = \alpha x^2 yx + \beta zy \qquad B = \gamma z + \delta y^4$$

we would have

$$A \cdot B = \left(\alpha x^2 yx + \beta zy\right) \cdot \left(\gamma z + \delta y^4\right) = \alpha\gamma x^2 yxz + \alpha\delta x^2 yxy^4 + \beta\gamma zyz + \beta\delta zy^5$$

and

$$B \cdot A = \left(\gamma z + \delta y^4\right) \cdot \left(\alpha x^2 yx + \beta zy\right) = \alpha\gamma zx^2 yx + \beta\gamma z^2 y + \alpha\delta y^4 x^2 yx + \beta\delta y^4 zy.$$

Note that multiplication is not commutative, but it is associative.

The `STL` `map` class

Here, a *term* is defined to be a scalar multiple of a *word*, and an element of the free algebra is considered to be the sum of a finite number of *terms*.

Thus we can view an element of the free algebra as a map from the set of words to the reals, each word mapping to its coefficient. Note that the empty word maps to the constant term. In `STL` terminology, a `map` is a sorted associative container that contains key-value pairs with unique keys. It is used in the package to map words to their coefficients, and is useful here because search and insertion operations have logarithmic complexity.

Package conventions

The indeterminates are the strictly positive integers (that is, we identify X_i with integer i); a natural and easily implemented extension is to allow each indeterminate X_i to have an inverse X_{-i}.

It is natural to denote indeterminates $X_1, X_2, X_3 \ldots$ with lower-case letters $a, b, c, \ldots$ and their multiplicative inverses with upper-case letters $A, B, C \ldots$

Thus we might consider $X = 5a + 43xy + 6yx - 17a^3b$ to be the map

```
{[1] -> 5, [24,25] -> 43, [25,24] -> 6, [1,1,1,2] -> -17}
```

In standard template library language, this is a map from a list of signed integers to a double; the header in the package reads

```
typedef std::list<signed int> word; // a 'word' object is a list of signed ints
typedef map <word, double> freealg; // a 'freealg' maps word objects to reals
```

Although there are a number of convenience wrappers in the package, we would create object X as follows:

```r
library("freealg")
X <- freealg(words = list(1, c(24,25), c(25,24), c(1,1,1,2)), coeffs = c(5, 43, 6, -17))
dput(X)
# structure(list(indices = list(1L, c(1L, 1L, 1L, 2L), 24:25, 25:24),
#     coeffs = c(5, -17, 43, 6)), class = "freealg")
X
# free algebra element algebraically equal to
# + 5*a - 17*aaab + 43*xy + 6*yx
```

(the print method translates from integers to letters). Note that the key-value pairs do not have a well-defined order in the `map` class; so the terms of a free algebra object may appear in any order. This does not affect the algebraic value of the object and allows for more efficient storage and manipulation. See also the `mvp` (Hankin 2019a) and `spray` (Hankin 2019b) packages for similar issues.

The package in use

There are a variety of ways of creating free algebra objects:

```
(X <- as.freealg("3aab -2abbax"))  # caret ("^") not yet implemented
# free algebra element algebraically equal to
# + 3*aab - 2*abbax
(Y <- as.freealg("2 -3aab -aBBAA"))  # uppercase letters are inverses
# free algebra element algebraically equal to
# + 2 - 1*aBBAA - 3*aab
(Z <- as.freealg(1:3))
# free algebra element algebraically equal to
# + 1*a + 1*b + 1*c
```

Then the usual arithmetic operations work, for example:

```
X^2         # powers are implemented
# free algebra element algebraically equal to
# + 9*aabaab - 6*aababbax - 6*abbaxaab + 4*abbaxabbax
X+Y         #  'aab' term cancels
# free algebra element algebraically equal to
# + 2 - 1*aBBAA - 2*abbax
1000+Y*Z    # algebra multiplication and addition works as expected
# free algebra element algebraically equal to
# + 1000 + 2*a - 1*aBBA - 1*aBBAAb - 1*aBBAAc - 3*aaba - 3*aabb - 3*aabc + 2*b +
# 2*c
```

Substitution

Algebraic substitution is implemented in the package with the `subs()` function:

```
subs("1+4a",a="xx")
# free algebra element algebraically equal to
# + 1 + 4*xx
p <- as.freealg("1+aab+4aba")
subs(p,a="1+x",b="y+xx")
# free algebra element algebraically equal to
# + 1 + 5*xx + 10*xxx + 5*xxxx + 1*xxy + 6*xy + 4*xyx + 5*y + 4*yx
```

Substitution works nicely with the `magrittr` package:

```
library("magrittr")
"1+aaab+4abaa" %>% subs(b="1+x+3aa")
# free algebra element algebraically equal to
# + 1 + 5*aaa + 15*aaaaa + 1*aaax + 4*axaa
```

Calculus

The package includes functionality to differentiate free algebra expressions. However, the idiom is not as cute as in the `mvp` package because `freealg` uses integers to distinguish the variables, while `mvp` uses symbolic expressions. Starting with a univariate `freealg` example we have:

```
k <- as.freealg("3 + 4aa + 7aaaaaa") # 3 + 4a^2 + 7a^6
k
# free algebra element algebraically equal to
# + 3 + 4*aa + 7*aaaaaa
```

```
deriv(k,1)   # dk/da  = 8a+42a^4
# free algebra element algebraically equal to
# + 7*aaaaa(da) + 7*aaaa(da)a + 7*aaa(da)aa + 7*aa(da)aaa + 4*a(da) + 7*a(da)aaaa
# + 4*(da)a + 7*(da)aaaaa
```

However, because of noncommutativity one has more complex behaviour. Observe that the Leibniz formula, if written $(xy)' = x'y + xy'$, still holds for noncommutative systems:

```
deriv(as.freealg("abaaaa"),1)  # d(aba^4)/da = ba^4 + 4aba^3
# free algebra element algebraically equal to
# + 1*abaaa(da) + 1*abaa(da)a + 1*aba(da)aa + 1*ab(da)aaa + 1*(da)baaaa
```

The package includes symbolwise multiplicative inverses and these too are differentiable in the package:

```
deriv(as.freealg("A"),1)    # d(a^-1)/da = -a^-2
# free algebra element algebraically equal to
# - 1*A(da)A
```

but again noncommutativity complicates matters:

```
deriv(as.freealg("Aba"),1)    # d(a^-1ba)/da = -a^-2ba + a^-1b
# free algebra element algebraically equal to
# + 1*Ab(da) - 1*A(da)Aba
```

The package can perform multiple partial differentiation by passing a vector second argument. To calculate, say, $\frac{d^3\left(a^3ba^{-1}cb^5\right)}{da^2\,db}$, is straighforward:

```
X <- as.freealg("aaabAcbbbbb")
X
# free algebra element algebraically equal to
# + 1*aaabAcbbbbb
deriv(X,c(1,1,2))
# free algebra element algebraically equal to
# + 2*aaabA(da)A(da)Acbbbb(db) + 2*aaabA(da)A(da)Acbbb(db)b +
# 2*aaabA(da)A(da)Acbb(db)bb + 2*aaabA(da)A(da)Acb(db)bbb +
# 2*aaabA(da)A(da)Ac(db)bbbb + 2*aaa(db)A(da)A(da)Acbbbbb -
# 2*aa(da)bA(da)Acbbbb(db) - 2*aa(da)bA(da)Acbbb(db)b - 2*aa(da)bA(da)Acbb(db)bb
# - 2*aa(da)bA(da)Acb(db)bbb - 2*aa(da)bA(da)Ac(db)bbbb -
# 2*aa(da)(db)A(da)Acbbbbb - 2*a(da)abA(da)Acbbbb(db) - 2*a(da)abA(da)Acbbb(db)b
# - 2*a(da)abA(da)Acbb(db)bb - 2*a(da)abA(da)Acb(db)bbb -
# 2*a(da)abA(da)Ac(db)bbbb - 2*a(da)a(db)A(da)Acbbbbb + 2*a(da)(da)bAcbbbb(db) +
# 2*a(da)(da)bAcbbb(db)b + 2*a(da)(da)bAcbb(db)bb + 2*a(da)(da)bAcb(db)bbb +
# 2*a(da)(da)bAc(db)bbbb + 2*a(da)(da)(db)Acbbbbb - 2*(da)aabA(da)Acbbbb(db) -
# 2*(da)aabA(da)Acbbb(db)b - 2*(da)aabA(da)Acbb(db)bb - 2*(da)aabA(da)Acb(db)bbb
# - 2*(da)aabA(da)Ac(db)bbbb - 2*(da)aa(db)A(da)Acbbbbb + 2*(da)a(da)bAcbbbb(db)
# + 2*(da)a(da)bAcbbb(db)b + 2*(da)a(da)bAcbb(db)bb + 2*(da)a(da)bAcb(db)bbb +
# 2*(da)a(da)bAc(db)bbbb + 2*(da)a(da)(db)Acbbbbb + 2*(da)(da)abAcbbbb(db) +
# 2*(da)(da)abAcbbb(db)b + 2*(da)(da)abAcbb(db)bb + 2*(da)(da)abAcb(db)bbb +
# 2*(da)(da)abAc(db)bbbb + 2*(da)(da)a(db)Acbbbbb
```

We can then perform a number of consistency checks, remembering that multiplication is not commutative:

```
set.seed(0)
(X <- rfalg(maxsize=10,include.negative=TRUE))
# free algebra element algebraically equal to
# + 3*CC + 7*CCbCCCac + 6*CaCb + 4*BBaaCC + 1*a + 5*bbCAAAbA + 2*cc
d <- deriv(X,1)
```

```
deriv(X^3,1) == X^2*d + X*d*X + d*X^2
# [1] TRUE
```

Observe carefully that the usual chain rule is not applicable here. If multiplication is commutative, one has $d(X^3)/da = 3X^2dX/da$, but the above idiom shows that $d(X^3)/da = X^2\,dX/da + X dX/da\,X + dX/da\,X^2$. Now we check Young's theorem, that partial derivatives commute:

```
deriv(X,1:2) == deriv(X,2:1)
# [1] TRUE
```

If differentiation commutes pairwise we may perform any number of partial derivatives in any order with the same result:

```
deriv(X,c(1,2,3,1,2,3)) == deriv(X,c(3,3,2,1,2,1))
# [1] TRUE
```

We can also verify the Leibniz formula for products, remembering to maintain the order of the terms:

```
(X <- rfalg(maxsize=10,include.negative=TRUE))
# free algebra element algebraically equal to
# + 7*C + 6*BCCCABBC + 2*BACBAB + 1*ABaaCA + 3*bbbAcB + 4*cbABAC + 5*cbc
(Y <- rfalg(maxsize=10,include.negative=TRUE))
# free algebra element algebraically equal to
# + 5*a + 2*abcc + 1*b + 4*bb + 7*bcBaBBBACC + 3*cAba + 6*ccBcabA
deriv(X*Y,1) == deriv(X,1)*Y + X*deriv(Y,1)
# [1] TRUE
```

Higher derivatives are somewhat harder:

```
f1 <- function(x){deriv(x,1)}
f2 <- function(x){deriv(x,2)}
f1(f2(X)) == f2(f1(X))  # Young
# [1] TRUE
f1(f2(X*Y)) == X*f1(f2(Y)) + f1(X)*f2(Y) + f2(X)*f1(Y) + f1(f2(X))*Y # Leibniz
# [1] TRUE
```

The print method

The default print method uses uppercase letters to represent multiplicative inverses, but it is possible to set the `usecaret` option, which makes changes the appearance:

```
phi <- rfalg(n=5,inc=TRUE)
phi
# free algebra element algebraically equal to
# + 2*Ba + 4*a + 3*aB + 5*bCb + 1*baca
options("usecaret" = TRUE)
phi
# free algebra element algebraically equal to
# + 2*b^-1a + 4*a + 3*ab^-1 + 5*bc^-1b + 1*baca
options("usecaret" = FALSE)  # reset to default
```

How the `disordR` package is used

It is possible to extract and examine the different components of a `freealg` object. Consider:

```
a <- as.freealg("1 + 4*a + 7*aaac + 1*aab + 5*acac + 3*bbaa + 2*bbab + 6*c")
a
# free algebra element algebraically equal to
# + 1 + 4*a + 7*aaac + 1*aab + 5*acac + 3*bbaa + 2*bbab + 6*c
dput(a)
# structure(list(indices = list(integer(0), 1L, c(1L, 1L, 1L, 3L
# ), c(1L, 1L, 2L), c(1L, 3L, 1L, 3L), c(2L, 2L, 1L, 1L), c(2L,
# 2L, 1L, 2L), 3L), coeffs = c(1, 4, 7, 1, 5, 3, 2, 6)), class = "freealg")
```

We see that, internally, `a` is a two-element list, with the first element being the indices and the second being the coefficients. We may extract the coefficients as follows:

```
coeffs(a)
# A disord object with hash 3c7d8fa26fcfb5b80bbefd1c7c150a7e2dfbab8f and elements
# [1] 1 4 7 1 5 3 2 6
# (in some order)
```

Above, observe how `coeffs()` does not return a vector but an object of class `disord`, defined in the `disordR` package. The hash code is displayed by the `disord` print method and serves to stop misuse of the coefficients. Standard extraction and replacement conditions apply to this object; for example, we cannot extract the "first" element, and trying to do so generates an error from the S4 extraction method of `disordR`:

```
coeffs(a)[1]
# Error in .local(x, i, j = j, ..., drop): if using a regular index to extract, must extract each eleme
```

The error is intentional: the key-value pairs are stored in an implementation-specific order, so there is no well-defined "first" element. Nevertheless, some extraction methods do make sense. We can, for example, find every coefficient that is greater than 3:

```
coeffs(a) > 3
# A disord object with hash 3c7d8fa26fcfb5b80bbefd1c7c150a7e2dfbab8f and elements
# [1] FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE
# (in some order)
```

Above, note that the result is another `disord` object, in this case Boolean. The result cannot be a vector because for the extracted values, as for the parent, have no uniquely defined order. Observe that the result can be used to extract coefficients of `a`:

```
coeffs(a)[coeffs(a) > 3]
# A disord object with hash 49503055c4e4090b32720f9a6a171386a4f59061 and elements
# [1] 4 7 5 6
# (in some order)
```

The hash key prevents one from treating the result of `coeffs()` like a vector in cases where the unspecified order would cause ambiguities. For example:

```
b <- a*2
coeffs(a) + coeffs(b)
# Error in coeffs(a) + coeffs(b): consistent(e1, e2) is not TRUE
coeffs(a) < coeffs(b)
# Error in coeffs(a) < coeffs(b): consistent(e1, e2) is not TRUE
```

Above, neither idiom is well-defined and returns an error; the diagnostic is the hash codes differing. Note, however, that there is not a complete bar on combining coefficients of distinct objects:

```
coeffs(a) + coeffs(b)[coeffs(b) >= 14]
# A disord object with hash 3c7d8fa26fcfb5b80bbefd1c7c150a7e2dfbab8f and elements
# [1] 15 18 21 15 19 17 16 20
# (in some order)
```

The idiom above has a well-defined result because the object to the right of the + sign has length 1 and ordering is not an issue. Replacement methods follow similar desiderata:

```
coeffs(a)
# A disord object with hash 3c7d8fa26fcfb5b80bbefd1c7c150a7e2dfbab8f and elements
# [1] 1 4 7 1 5 3 2 6
# (in some order)
coeffs(a)[coeffs(a) < 3] <- 0  # set any coefficient < 3 to zero
a
# free algebra element algebraically equal to
# + 4*a + 7*aaac + 5*acac + 3*bbaa + 6*c
```

Also

```
(X <- rfalg())
# free algebra element algebraically equal to
# + 4*a + 7*aaac + 1*aab + 5*acac + 3*bbaa + 2*bbab + 6*c
coeffs(X) <- coeffs(X)%%2  # treat coeffs modulo 2
X
# free algebra element algebraically equal to
# + 1*aaac + 1*aab + 1*acac + 1*bbaa
```

We may also perform similar operation with the `words()` of a `freelg` object.

```
X <- as.freealg("1+5x + 7*x*y*x + 6*x*x*x*x*x - 9a*b + x*a - 3a*x - x*y*z*a")
X
# free algebra element algebraically equal to
# + 1 - 9*ab - 3*ax + 5*x + 1*xa + 6*xxxxx + 7*xyx - 1*xyza
```

Suppose we wish to extract only those terms with exactly two. For this, we need `disordR::sapply()`:

```
library("disordR")  # overloads sapply()
wanted <- sapply(words(X),function(x){length(x)==2})
wanted
# A disord object with hash 8ff9a51a78b587a55b163bef23039772167bc9f5 and elements
# [1] FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE
# (in some order)
freealg(words(X)[wanted],coeffs(X)[wanted])
# free algebra element algebraically equal to
# - 9*ab - 3*ax + 1*xa
```

As a final illustration, suppose we wanted to kill every term starting with an `x` that has a large (absolute value exceeds 100, say) coefficient:

```
P <- as.freealg("1 + x + y + 1002*x*y + y*x + 176*x*x*y -5423*x*z*x")
P
# free algebra element algebraically equal to
# + 1 + 1*x + 176*xxy + 1002*xy - 5423*xzx + 1*y + 1*yx
w1 <- sapply(words(P),function(w){identical(w[1],24L)})
w2 <- sapply(coeffs(P),function(w){abs(w)>1000})
wanted <- !(w1 & w2)
freealg(words(P)[wanted],coeffs(P)[wanted])
# free algebra element algebraically equal to
# + 1 + 1*x + 176*xxy + 1*y + 1*yx
```

Alternatively

```
coeffs(P)[w1 & w2] <- 0
P
# free algebra element algebraically equal to
# + 1 + 1*x + 176*xxy + 1*y + 1*yx
```

An example

Haiman (1993) poses the following question, attributed to David Richman:

"find the constant term of $\left(x+y+x^{-1}+y^{-1}\right)^p$ when x and y do not commute".

Package idiom is straightforward. Consider $p=4$:

```
X <- as.freealg("x+y+X+Y")
X^2
# free algebra element algebraically equal to
# + 4 + 1*YY + 1*YX + 1*Yx + 1*XY + 1*XX + 1*Xy + 1*xY + 1*xx + 1*xy + 1*yX +
# 1*yx + 1*yy
constant(X^4)
# [1] 28
```

We could even calculate the first few terms of Sloane's A035610:

```
f <- function(n){constant(as.freealg("x+y+X+Y")^n)}
sapply(c(0,2,4,6,8,10),f)
# [1]     1     4    28   232  2092 19864
```

References

Haiman, M. 1993. "Non-Commutative Rational Power Series and Algebraic Generating Functions." *European Journal of Combinatorics* 14 (4): 335–39.

Hankin, Robin K. S. 2019a. *mvp: Fast Symbolic Multivariate Polynomials*. https://github.com/RobinHankin/mvp.git.

———. 2019b. *Spray: Sparse Arrays and Multivariate Polynomials*. https://github.com/RobinHankin/spray.git.